Recognizing Mathematical Expressions Using Tree Transformation

Richard Zanibbi, *Student Member*, *IEEE Computer Society*,
Dorothea Blostein, *Member*, *IEEE*, and James R. Cordy, *Member*, *IEEE*

Abstract—We describe a robust and efficient system for recognizing typeset and handwritten mathematical notation. From a list of symbols with bounding boxes the system analyzes an expression in three successive passes. The Layout Pass constructs a Baseline Structure Tree (BST) describing the two-dimensional arrangement of input symbols. Reading order and operator dominance are used to allow efficient recognition of symbol layout even when symbols deviate greatly from their ideal positions. Next, the Lexical Pass produces a Lexed BST from the initial BST by grouping tokens comprised of multiple input symbols; these include decimal numbers, function names, and symbols comprised of nonoverlapping primitives such as "=". The Lexical Pass also labels vertical structures such as fractions and accents. The Lexed BST is translated into LaTeX. Additional processing, necessary for producing output for symbolic algebra systems, is carried out in the Expression Analysis Pass. The Lexed BST is translated into an Operator Tree, which describes the order and scope of operations in the input expression. The tree manipulations used in each pass are represented compactly using tree transformations. The compiler-like architecture of the system allows robust handling of unexpected input, increases the scalability of the system, and provides the groundwork for handling dialects of mathematical notation.

Index Terms—Document image analysis, recognition of mathematical notation, diagram recognition, tree transformation, graphics recognition.

1 Introduction

Automated recognition of mathematical notation is a challenging pattern recognition problem of great practical importance. Applications include the conversion of scientific papers from printed to electronic form and the reading of scientific documents to visually impaired users. Recognition of handwritten expressions permits users to write mathematical expressions on a data tablet; this is a convenient alternative to input methods such as typing LaTeX expressions or using a structure-based editor for mathematical notation.

Over the past 30 years researchers have investigated many approaches to recognizing mathematical notation. Surveys are available in [1] and [2].

1.1 Challenges

This section briefly reviews some of the challenges that arise in recognition of mathematical notation. First, expressions must be located in a document image that contains a mix of text, expressions, and figures. Expressions can be offset or inline. Various approaches to this problem have been studied [3], [4].

Recognizing mathematical symbols is difficult because a large number of symbols, fonts, typefaces, and font sizes are used [5]. Care must be taken to distinguish between noise and small symbols such as periods and commas.

Recognizing the spatial relationships between symbols (the *symbol layout*) is challenging, particularly for handwritten notation. The blurry distinction between inline and superscript relationships, illustrated by Fig. 1b, makes it difficult to define robust methods for recognizing relationships. A statistical study of superscript versus inline versus subscript relationships in handwritten mathematics expressions is reported in [6]. Fig. 1 shows expressions for which ambiguous layout confuses the order, scope, and even presence of operations. The inexact symbol placement that is common in handwritten notation (Fig. 2a) compounds this problem.

Ambiguous spatial relationships and symbol identities need to be resolved using contextual analysis [7], [8]. Also, contextual analysis is needed to disambiguate the roles of mathematical symbols. For example, a horizontal line may act as a fraction line, subtraction symbol, or as an overbar for Boolean negation. Exploitation of redundancy is a common technique for resolving ambiguities; an example is the redundancy between city name and postal code in address recognition [9]. However, mathematics uses a concise notation, one which provides little redundancy.

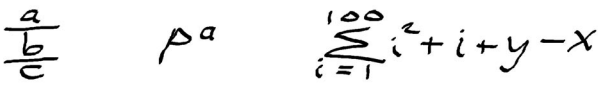
Finally, mathematics notation is not formally defined and many dialects are in use. Similar to natural languages, mathematical symbols and structures are invented or redefined as needed by the users of the notation. Publications about the formatting of mathematical notation are available [10], [11], [12]. However, these are not in a form that can be used as a specification for a mathematics recognition system.

1.2 Mathematics Recognition via Tree Transformation

In this paper, we describe the design and implementation of a mathematics recognition system that makes extensive use of tree transformation. The ideas underlying this approach

• *The authors are with the School of Computing, Queen's University, Kingston, Ontario, Canada, K7L 3N6.*
E-mail: {zanibbi, blostein, cordy}@cs.queensu.ca.

Manuscript received 27 June 2001; revised 15 Feb. 2002; accepted 2 Apr. 2002. Recommended for acceptance by L. Vincent.
For information on obtaining reprints of this article, please send e-mail to: tpami@computer.org, and reference IEEECS Log Number 114438.



(a) (b) (c)

Fig. 1. These expressions illustrate that ambiguous layout can confuse the order, presence, and scope of operators. (a) Which division is performed first? (b) Is a superscripted? (c) What is the extent of the scope of the summation?

may be relevant in any application where syntactic pattern recognition is appropriate. The following strategies are used to structure the recognition system.

We analyze symbol layout in mathematical expressions by searching for linear structures (baselines) in the input and then using these as the basis for finding secondary linear structures. Intelligent search functions are applied in image subregions; the subregions are defined in a symbol-specific way, as described in Section 3. This strategy allows us to exploit the left-to-right reading order of mathematical notation, thereby analyzing layout efficiently without backtracking. Similar layout analysis techniques have been used in applications including parsing of visual languages [13] and recognition of mathematical notation [14], [15]. One of our contributions is to generalize the technique to make it robust enough to handle the irregular symbol layouts present in handwritten expressions (Fig. 2a).

The linear structures (baselines) are organized into a *Baseline Structure Tree* (BST). This tree forms the basis for subsequent, compiler-style processing. Processing is divided into three major passes: 1) The Layout Pass builds an initial BST, 2) the Lexical pass groups and labels compound symbols (e.g., "sin") and structure symbols (e.g., fraction lines), and 3) the Expression Analysis Pass analyzes expression syntax (operator precedence and associativity) and produces an *operator tree*. The operator tree describes an ordered application of operators to operands. This represents the semantics of the mathematical expression, as is needed for evaluating the expression or translating the expression into a Computer Algebra System format.

The use of passes results in robust processing of input: The Layout Pass processes *all* inputs, even those that contain syntax errors or unknown constructs. This produces useful partial results for any input. Also, the use of passes is a helpful structuring tool for recognizing various dialects of mathematical notation. While the core of the Layout Pass is fixed, the symbol class definitions used in the Layout Pass may be easily redefined. Additionally, the Lexical Pass and Expression Analysis Pass may be provided with dialect-specific rules.

All of the processing in our approach is performed using tree manipulations called *tree transformations*. Tree transformations allow the computations we perform to be expressed in a convenient and compact form (see Section 1.3). Our decision to make use of tree transformations stemmed from the observation that both the layout and syntax of mathematical expressions are hierarchical and, as a result, are usually expressed as trees. Trees are used in formatting languages such as LaTeX, for representing the parse of mathematical expressions in compilers [16], and in many other approaches to mathematics recognition (as surveyed in [8]).

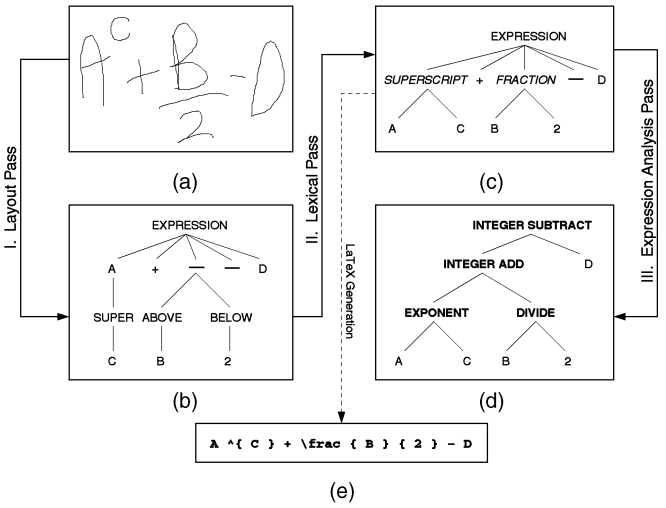


Fig. 2. Overview of processing in DRACULAE.

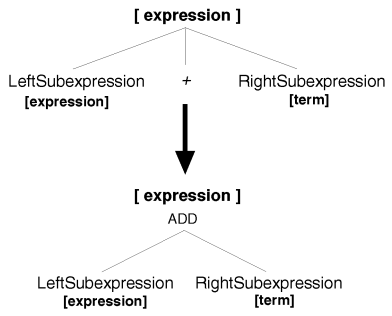
Our implementation is called the Diagram Recognition Application for Computer Understanding of Large Algebraic Expressions (DRACULAE) [8], [17]. For processing online input, DRACULAE is packaged with a user interface and a third-party symbol recognizer, both of which are provided by the Freehand Formula Entry System [18], [19].

DRACULAE obtains linear or close to linear performance on many inputs. The worst case time complexity of DRACULAE, when processing an input of n symbols is $O(n^2 \lg n)$. Worst case inputs are unsyntactic or unlikely. For example, one worst case input consists of a series of $\sum$ symbols, each placed to be a superscript of the preceding one. Most inputs are processed in near linear time. This is particularly impressive because DRACULAE handles handwritten inputs with irregular symbol placements. Many alternative approaches designed to cope with ambiguous layout, such as stochastic grammars [20] and graph transformation [21], involve extensive amounts of search or backtracking. DRACULAE currently recognizes a single dialect of mathematics notation, but has been constructed to allow multiple dialects to be accommodated in the future.

Fig. 2 provides an overview of the processing performed by DRACULAE. Tree transformation, which is used throughout the implementation, is discussed in Section 1.3. The symbol layout model and Baseline Structure Trees are defined in Section 2. The symbol layout model is used by the Layout Pass (Section 3) to convert the input into a Baseline Structure Tree. The Lexical Pass (Section 4) converts this to a Lexed BST. Finally, the Expression Analysis Pass (Section 5) produces an operator tree. Experimental results on handwritten and typeset input are presented in Section 6.

1.3 Tree Transformation

DRACULAE uses trees as its central data structure. The recognition process begins by building a tree that encodes low-level baseline structure. This tree is successively refined and restructured to represent higher levels of understanding at each stage of the process. Tree restructurings are implemented using a programming language construct called *tree transformation*. A tree transformation is a



(a)

```
rule convertAdditionsToOperatorTrees
    replace [expression]
        LeftSubexpression[expression] +
        RightSubexpression[term]
    by
        ADD LeftSubexpression
        RightSubexpression
end rule
```

(b)

Fig. 3. A tree transformation rule from the Expression Analysis Pass. The rule is shown graphically (a) and as TXL code (b). This rule finds all parse subtrees for subexpressions that use an infix binary + operation. Each of these parse subtrees is replaced by an operator subtree explicitly indicating that addition is intended.

restructuring rule that searches a host tree (the *scope*) for subtrees with a particular shape and attribute values (the *pattern*); each matching subtree is replaced with a new subtree (the *replacement*) restructured from the original. Fig. 3 shows a tree transformation rule.

We use the tree transformation language TXL to specify tree transformations in a compact, abstract manner [22], [23]. Originally designed for programming language processing tasks, TXL specifies tree transformations in ASCII text using a by-example style of rule specification (Fig. 3), and provides an efficient, robust parser to rapidly convert trees to and from ASCII text form. TXL transformation rules can be combined and controlled using functional programming constructs and are directly and efficiently executed by the TXL interpreter. The amount of code needed to describe a complex tree transformation in TXL is orders of magnitude less than in a general purpose programming language such as C. DRACULAE is implemented by less than 3,500 lines of TXL code.

2 Symbol Layout in Mathematical Expressions

Mathematical notation uses symbol layout to convey which operators are used and what the arguments to these operators are. An analysis of operator dominance and baselines can be used to recover this information. The following sections define operator dominance, baselines, Baseline Structure Trees, and symbol classes. These define the symbol layout model which forms the basis of the Layout Pass.

2.1 Operator Dominance and Baselines

Operator dominance [24] is a concept used to determine the precedence and arguments of operators.

Range: The *range* of an operator is the expected location of its argument(s) [24]. The ranges DRACULAE uses are described in Section 2.3.

Operator Dominance: Operator A *dominates* operator B if B is in the range of A and A is not in the range of B [24]. An operator *dominates* the symbols that constitute its arguments.

If operator A dominates operator B, then A is of lower precedence than B. For example, in the expression $x + \frac{y-z \cdot d}{a}$ the "+" dominates the fraction line because the fraction line is in the range of the addition sign and the converse is false. Similarly, the fraction line dominates the subtraction and multiplication operators and their arguments. Neither the subtraction or multiplication operator dominates the other, because both are in the range of the other.

Fig. 1a is ambiguous because the operator dominance (and, as a result, precedence) is unclear: The fraction lines are of equal length and arranged vertically, so neither appears to dominate the other. Different dialects of mathematical notation use varying definitions of operator range and dominance. For instance, the ambiguity in Fig. 1a can be resolved by choosing a definition of operator dominance that results in selection of either the top or bottom line.

Baseline and *Start Symbol* are defined using operator dominance and the left-to-right ordering of mathematical notation.

Baseline: A baseline in mathematical notation is a linear horizontal arrangement of symbols, intended to be perceived as adjacent.

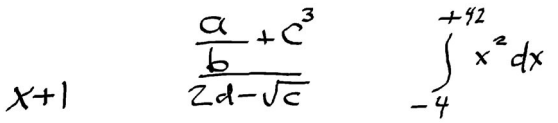
For example, there are two baselines in the expression $x^{2+a} - y$. One baseline contains the symbols $(x, -, y)$ and the other contains $(2, +, a)$. In handwritten expressions, the placement of baseline symbols may deviate far from the ideal horizontal arrangement (Fig. 2a).

Nested Baseline: A nested baseline is a baseline that is either vertically offset from a symbol or contained by a symbol (as in the case of a square root containing an expression comprised of one or more baselines). Nested baselines are used to indicate operator dominance. For example, in the expression $\frac{1}{2}$, the two baselines, 1 and 2, are nested relative to the fraction line. Similarly, in the expression $x^{2+a} - y$, the superscripted baseline $(2, +, a)$ is nested relative to the x.

Dominant Baseline: The dominant baseline of a mathematical expression contains the symbols that are not nested relative to any other symbols in the expression. The dominant baseline of a mathematical expression begins with the Start Symbol of the expression.

Start Symbol: In a mathematical expression, the Start Symbol is the leftmost symbol of the expression which is not on a nested baseline.

Examples of Start Symbols are shown in Fig. 4. The Layout Pass (Section 3) contains algorithms for locating the Start Symbol and subsequent baseline symbols.



(a) (b) (c)

Fig. 4. Examples of Start Symbols. In (a) the leftmost symbol is not on a nested baseline and is the Start Symbol. In (b), the Start Symbol is the wider fraction line. This operator dominates the remaining symbols, which are all located on baselines nested relative to it. Similarly, in (c), the integral is the leftmost nonnested symbol and is the Start Symbol.

2.2 Baseline Structure Trees

A Baseline Structure Tree represents the hierarchical structure of baselines in an expression [17]. The Baseline Structure Tree explicitly captures important aspects of symbol layout without committing to any particular syntactic or semantic interpretation. For instance, a Baseline Structure Tree can be used to represent the symbol layout of "$2+$" despite the fact that this expression is syntactically and semantically invalid. Similarly, a Baseline Structure Tree represents the symbol layout of "$f(x)$" regardless of whether function application or multiplication of variables is intended.

A Baseline Structure Tree (or *BST*) contains two types of nodes: *symbol nodes* and *region nodes*, defined below. These nodes are arranged in levels; any path through the tree encounters symbol nodes and region nodes in alternation. The root of the tree, $EXPRESSION$, is a region node representing the entire image.

Symbol Node: A symbol node represents a mathematical symbol. The symbol node stores the identity of the symbol (as provided by symbol recognition), the class of the symbol (as defined in Table 1), and the attributes of the symbol (the bounding box and centroid coordinates). A symbol node is the root of a subtree of the BST. Suppose S is a symbol represented by symbol node $snode$. The children of $snode$ are region nodes representing image subregions that contain baselines nested relative to S.

Region Node: A region node represents an image region which contains a baseline, possibly with nested baselines. The image region is defined relative to the symbol that is the parent of this region node; the spatial relationship is captured by the *region label*, defined below. The region node is the root of a subtree; the children of the region node are symbols that form the region's dominant baseline.

Region Label: All region nodes in a BST have a region label, one of $ABOVE$, $BELOW$, $SUPER$, $SUBSC$, $UPPER$, $LOWER$, $TLEFT$ (top-left), $BLEFT$ (bottom-left), $CONTAINS$, and $EXPRESSION$. As shown in Fig. 5, the class of a symbol determines what regions are defined relative to the symbol.

In a Baseline Structure Tree, region nodes represent all mathematically important spatial relationships other than horizontal adjacency. Horizontal adjacency has special status because it defines baselines. Symbols that are on the same baseline are represented in the tree as ordered siblings.

These definitions are illustrated using the Baseline Structure Tree shown in Fig. 2b. This tree contains four region nodes ($EXPRESSION$, $SUPER$, $ABOVE$, and $BELOW$) and eight symbol nodes (A + − − D C B 2). The dominant baseline of the whole expression is (A + − − D). The "2" is the sole symbol in the baseline located $BELOW$ the first "−." The "C" is the sole symbol of the baseline that is superscripted ($SUPER$) relative to the "A."

2.3 Symbol Classes

In the Layout Pass, Symbol classes and the parameters c (centroid ratio) and t (threshold ratio) are used to define

TABLE 1
Symbol Classes and Their Associated Attributes

Symbol Class	y-centroid	Thresholds			
		BELOW	ABOVE	SUBSC	SUPER
Non-Scripted unary/binary operators and relations (+,-,=,≥,→,etc.)	$\frac{1}{2}H$	$\frac{1}{2}H$	$\frac{1}{2}H$	–	–
Open Bracket (, {, [	cH	$minY$	$maxY$	–	–
Root ($\sqrt{}$)	cH	$minY$	$maxY$	tH	$H-(tH)$
Variable Range $\Sigma, \int, \Pi, \cup, \cap$	$\frac{1}{2}H$	tH	$H-(tH)$	tH	$H-(tH)$
Plain: Ascender 0...9, A...Z, b,d,f,h,i,k,l,t, $\Gamma, \Delta, \Theta, \Lambda, \Xi, \Phi, \Psi, \Omega, \delta, \theta, \lambda$	cH	tH	$H-(tH)$	tH	$H-(tH)$
Plain: Descender g,p,q,y, $\gamma, \eta, \mu, \rho, \chi, \psi$	$H-(cH)$	$\frac{1}{2}H+t\frac{1}{2}H$	$H-t\frac{1}{2}H$	$\frac{1}{2}H+t\frac{1}{2}H$	$H-t\frac{1}{2}H$
Plain: Centered All other symbols (including Close Brackets)	$\frac{1}{2}H$	tH	$H-(tH)$	tH	$H-(tH)$

The ABOVE, BELOW, SUPER, and SUBSC thresholds are used to define the regions associated with each symbol, as shown in Fig. 5. The values maxY and minY are bounding box coordinates and H is the bounding box height (maxY-MinY). The centroid ratio, c, and the threshold ratio, t, are both in range [0, 0.5], with $t \leq c$.

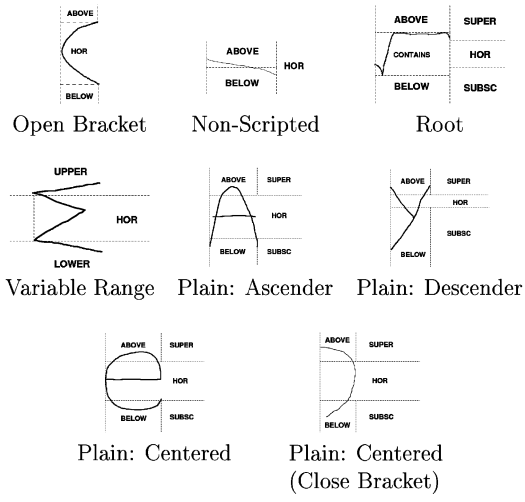


Fig. 5. Regions associated with the different symbol clases. The right end of the HOR, SUPER, SUBSC, UPPER, and LOWER regions is located at the minX coordinate of the next baseline symbol. The left end of the LOWER and UPPER regions is the maxX coordinate of the previous baseline symbol. Y-thresholds for each region are defined in Table 1.

image regions around symbols. A variety of c and t values are tested on the UW-III database in Section 6.1. As described in Section 3, the Layout Pass recognizes the symbols in the dominant baseline of a region, defines subregions around these symbols, and searches for nested baselines in these subregions. This section defines the symbol classes and regions that are used and defines the test for determining whether a symbol lies in a region. These definitions comprise the *symbol layout model*.

The *centroid* of a symbol is a point used to test whether a symbol lies within a region. This is a common technique in the literature on recognition of mathematical notation, first used in the work of Anderson [25]. Collapsing a symbol to a single point allows for simpler geometric analyses. The centroid X-location is always centered in the bounding box, at (minX + maxX)/2. As shown in Table 1, the computation of the centroid Y-location depends on the *centroid ratio* c and on whether the symbol is an ascender, descender, or centered.

A *region* is an axis-parallel box. The region includes the left and bottom edges of the box, but not the right and top edges. The Layout Pass tiles the image with regions. All points in the image belong to exactly one region, so each symbol's centroid is located in exactly one region.

Every symbol is assigned a symbol class, as defined in Table 1. The symbol class determines where nested baselines can be located relative to the symbol. This is illustrated in Fig. 5.

Ambiguity, in the form of overlapping regions, can arise in the region definitions shown in Fig. 5. Consider two adjacent baseline symbols, where the symbol on the left has a $SUPER$ or $UPPER$ region and the symbol on the right has an $UPPER$ region (i.e., is in class Variable Range). The $SUPER$ or $UPPER$ region of the left symbol overlaps the $UPPER$ region of the right symbol. Similarly, the $SUBSC$ or $LOWER$ region of the left symbol overlaps the $LOWER$ region of the right symbol. For example, in the expression

$$x^2 \sum_{i=1}^{10,000} i$$

the symbols "2" and "1" may fall in both the $SUPER$ region of the "x" and the $UPPER$ region of the $\sum$. This ambiguity is resolved in the Layout Pass using analysis of local context (function $CollectRegions$ in Section 3.1).

3 Layout Pass

The Layout Pass produces a Baseline Structure Tree from a list of symbols with bounding boxes. It identifies the dominant baseline of the expression, partitioning any symbols not on the dominant baseline into regions relative to the dominant baseline symbols. This process is applied recursively in the partitioned regions. The left-to-right reading order of mathematical notation is exploited to construct the BST efficiently without backtracking, even when symbol layout is irregular. Extensive research went into defining the search functions $Start$ and Hor, discussed below. The inspiration for this directed search came from the linear positional grammar work of Costagliola et al. [13], where syntax-driven linear scanning of the input is used to parse visual languages. The directionality present in mathematical notation made it possible for us to adapt these ideas for use in the Layout pass.

Each input symbol s has bounding box coordinates denoted $minX(s)$, $minY(s)$, $maxX(s)$, and $maxY(s)$. The Layout Pass begins with a preprocessing step, in which Table 1 is used to assign each input symbol a symbol class $class(s)$, a centroid ($centroidX(s)$, $centroidY(s)$), and region thresholds ($aboveThreshold(s)$, $belowThreshold(s)$, $superThreshold(s)$, $subscThreshold(s)$). After this preprocessing, function $BuildBST$ creates the BST. Section 3.1 defines $BuildBST$ and the most important functions it uses: $ExtractBaseline$, $Start$, Hor, and $CollectRegions$. Supporting functions are defined in Section 3.2.

The major steps in the Layout Pass are as follows. They are illustrated in Fig. 6.

1. The initial Baseline Structure Tree consists of a root $EXPRESSION$ node, with a sorted list L of symbols as children. Symbols are sorted by $minX$ coordinate. R is the image region that contains the entire expression.
2. Find the symbol which begins the dominant baseline in region R. This is computed as $S_1 = Start(L)$. The $Start$ function checks for cases in which symbol S_1 is not the leftmost symbol in list L. For example, the limits of a $\sum$ can begin to the left of the $\sum$.
3. Find $S_2 \ldots S_n$, the rest of the symbols in the baseline that begins with symbol S_1. This is done by function Hor. Care is taken to handle irregular symbol layout, such as in the expression in Fig. 6.
4. Add $S_1 \ldots S_n$, the symbols in the dominant baseline in region R, to the Baseline Structure Tree. The symbol nodes are inserted as offspring of the region node representing R.
5. The symbols of the dominant baseline, $S_1 \ldots S_n$, are used to partition region R into subregions, using the region definitions from Fig. 5. All the symbols in

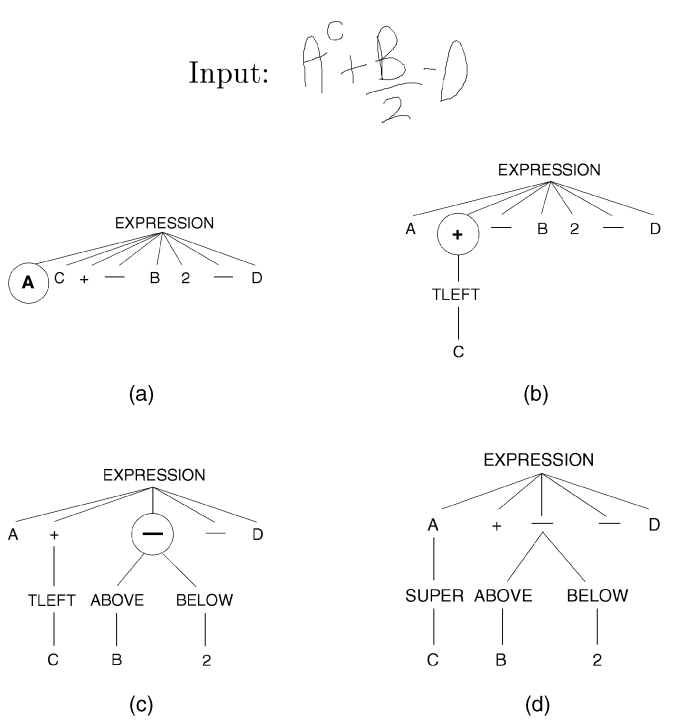


Fig. 6. BST construction by the Layout Pass for the expression in Fig. 2a. (a) The initial BST. This is created in function $BuildBST$ before invoking $ExtractBaseline$. The $Start$ function locates the leftmost symbol in the dominant baseline, indicated here by a circle around the "A." (b) The tree after Hor has found the next baseline symbol ("+"); region partitioning places the "C" into the $TLEFT$ region of the "+." (c) The tree after the third baseline symbol ("-") is located by Hor. (d) The final tree, after the last two baseline symbols have been found, and the $TLEFT$ partitioning has been refined. In this example, the nested baselines do not require further processing as they are single symbols.

list L that are not part of the dominant baseline are assigned to one of these subregions.

6. For each nonempty subregion found in the previous step, add a Region Node to the Baseline Structure Tree. Recursively apply Steps 2 to 6 to each of these regions.

In summary, the Layout pass recursively applies search functions and image partitioning to recognize dominant and nested baselines. The search function $Start$ is used to locate the leftmost symbol of the dominant baseline and Hor is used to locate successive symbols in a baseline.

3.1 Top Level Functions in the Layout Pass

This section and the next section provide a functional specification of DRACULAE's Layout Pass. The input, which is passed to function $BuildBST$, is a list of symbol nodes, annotated with bounding box coordinates. The output is a Baseline Structure Tree describing the layout of these symbols.

Function names are followed by a type specification. The parameter and return value types are BST (Baseline Structure Tree), SNODE (a symbol node, which may be the root of a subtree), RNODE (a region node, which may be the root of a subtree), REGION_LABEL (one of the 10 region labels defined in Section 2.2), SNODE_LIST (a list of symbol nodes), RNODE_LIST (a list of region nodes), REGION_LABEL_LIST (a list of region labels), BOOLEAN, and INTEGER. When several arguments have the same type, integer subscripts are added. Arguments are referenced using the same names, written in lower case.

For list notation, $|list|$ is the number of items in a list, $list - item$ denotes removal of an item from a list, and $(item)$ denotes a list consisting of a single item.

BuildBST ($SNODE\_LIST \rightarrow BST'$): Construct a Baseline Structure Tree from $snode\_list$, the input list of symbol nodes.

1. Let $root$ be a region node labeled $EXPRESSION$. If $|snode\_list| = 0$ Return $root$.
2. Let $snode\_list' = SortSymbolsByMinX(snode\_list)$.
3. Make each symbol node in $snode\_list'$ be a child of $root$.
4. Return $ExtractBaseline(root)$.

ExtractBaseline ($RNODE \rightarrow RNODE'$): Find the dominant baseline in the region represented by $rnode$ and update the part of the BST that is rooted at $rnode$. Make recursive calls to add nested baselines.

1. Let $snode\_list = Symbols(rnode)$. If $|snode\_list| \leq 1$ Return $rnode$.
2. Let $s_{start} = Start(snode\_list)$.
3. Let $baseline\_symbols = Hor(s_{start}, snode\_list)$.
4. Let $updated\_baseline = CollectRegions$ $(baseline\_symbols)$.
5. Update the tree rooted at $rnode$ by discarding the children of $rnode$ and replacing them by the symbol nodes in $updated\_baseline$. (Each symbol node in $updated\_baseline$ is itself the root of a subtree.)
6. Now, use recursion. For each region node $childrnode_i$ that is a child of a symbol node in $updated\_baseline$, replace $childrnode_i$ by $ExtractBaseline(childrnode_i)$.
7. Return $rnode$.

Start ($SNODE\_LIST \rightarrow SNODE'$): Find the symbol node which begins the dominant baseline in $snode\_list$. Compare the last two symbols in $snode\_list$, remove the dominated symbol, and recurse. Symbol s_n dominates the previous symbol, s_{n-1} if 1) $Overlaps(s_n, s_{n-1})$, or 2) $Contains(s_n, s_{n-1})$, or 3) $class(s_n) =$ Variable Range and $\neg IsAdjacent(s_{n-1}, s_n)$. Otherwise, s_{n-1} dominates s_n.

Hor ($(SNODE\_LIST_1, SNODE\_LIST_2) \rightarrow$ $SNODE\_LIST'$): Find the symbols of the baseline that begins with the symbols in $snode\_list_1$ and continues with a subset of the symbols in $snode\_list_2$. The symbols of the baseline are returned as $snode\_list'$. Nonbaseline symbols in $snode\_list_2$ are partitioned into $TLEFT$, $BLEFT$, $ABOVE$, $BELOW$, and $CONTAINS$ regions. Symbols in $TLEFT$ and $BLEFT$ regions are later reassigned by the $CollectRegions$ function.

1. If $|snode\_list_2| = 0$ Return $snode\_list_1$.
2. Let $current\_symbol$ be the last symbol node in $snode\_list_1$.
3. Let $(remaining\_symbols, current\_symbol') = Partition$ $(snode\_list_2, current\_symbol)$
4. In $snode\_list_1$, replace $current\_symbol$ by $current\_symbol'$.
5. If $|remaining\_symbols| = 0$ Return $snode\_list_1$.

6. If $class(current\_symbol') =$ Nonscripted then Return $Hor(ConcatLists(snode\_list_1, (Start(remaining\_symbols))), remaining\_symbols)$.
7. Let $SL = remaining\_symbols$.
8. While $|SL| \neq 0$,
   a. Let l_1 be the first symbol in SL.
   b. If $IsRegularHor(current\_symbol', l_1)$ then Return $Hor(ConcatLists(snode\_list_1, (CheckOverlap(l_1, remaining\_symbols))), remaining\_symbols)$.
   c. Let $SL = SL - l_1$.
9. Let $current\_symbol' = PartitionFinal(remaining\_symbols, current\_symbol')$.
10. Return $ConcatLists(snode\_list_1, (current\_symbol'))$.

CollectRegions $(SNODE\_LIST \rightarrow SNODE\_LIST')$:
$snode\_list$ is a list of symbol nodes whose subtrees contain temporary regions labeled $TLEFT$ and $BLEFT$, created by function Hor. The symbols in a $TLEFT$ region are reassigned to $SUPER$ (a region associated with the preceding baseline symbol) or $UPPER$ (a region associated with the current baseline symbol); similarly, symbols in $BLEFT$ regions are assigned to $SUBSC$ or $LOWER$ regions. For brevity, we show only the $TLEFT$ case here.

1. If $|snode\_list| = 0$ Return $snode\_list$.
2. Let s_1 be the first symbol of $snode\_list$. Let $s_1' = s_1$. Let $snode\_list' = snode\_list - s_1$.
3. If $|snode\_list| > 1$ then
   a. Let s_2 be the second symbol of $snode\_list$. Let $s_2' = s_2$.
   b. Let $(superList, tleftList) = PartitionShared Region(TLEFT, s_1, s_2)$.
   c. Let $s_1' = AddSuper(superList, s_1)$.
   d. Let $s_2' = AddTleft(tleftList, RemoveRegions((TLEFT), s_2))$.
   e. In list $snode\_list'$ replace s_2 by s_2'
4. If $class(s_1') = VariableRange$
   a. Let $upperList = (TLEFT, ABOVE, SUPER)$.
   b. Let $s_1' = MergeRegions(upperList, UPPER, s_1)$.
5. Return $ConcatLists((s_1'), CollectRegions(snode\_list'))$.

3.2 Supporting Functions in the Layout Pass

The following functions, listed in alphabetical order, are used by the top-level functions shown in the previous section.

AddAbove, AddBelow, etc. $((SNODE\_LIST, SNODE) \rightarrow SNODE')$: The symbol nodes in $snode\_list$ become grandchildren of $snode$. For AddAbove, they are placed as children of an ABOVE region node. Functions AddBelow, AddSuper, AddSubsc, AddContains, AddTleft, and AddBleft are defined analogously.

CheckOverlap $((SNODE, SNODE\_LIST) \rightarrow SNODE')$: Look through $snode\_list$ for Nonscripted symbols which horizontally overlap $snode$, tested via the *Overlaps* function. Return the widest such symbol if one exists. If there are no such symbols, return $snode$.

ConcatLists $((SNODE\_LIST_1, SNODE\_LIST_2) \rightarrow SNODE\_LIST')$: Concatenate the symbol node lists $snode\_list_1$ and $snode\_list_2$, returning the resulting list.

Contains $((SNODE_1, SNODE_2) \rightarrow BOOLEAN')$: Return true if $snode_1 \neq snode_2, class\ (snode_1) =$ Root, $minX(snode_1) \leq centroidX(snode_2) < maxX(snode_1)$, and $minY(snode_1) \leq centroidY(snode_2) < maxY(snode_1)$.

HasNonEmptyRegion $((SNODE, REGION\_LABEL) \rightarrow BOOLEAN')$: Return true if $snode$ has a child region node $rnode$ with region label $region\_label$, and $|Symbols(rnode)| > 0$.

IsAdjacent $((SNODE_1, SNODE_2) \rightarrow BOOLEAN')$: Test whether $snode_1$ is horizontally adjacent to $snode_2$, where $snode_1$ may be to the left or right of $snode_2$. Return true if $class(snode_2) \neq$ Nonscripted, $snode_1 \neq snode_2$, and $subscThreshold(snode_2) \leq centroidY(snode_1) < superThreshold(snode_2)$.

IsRegularHor $((SNODE_1, SNODE_2) \rightarrow BOOLEAN')$: Return true if a) $IsAdjacent(snode_2, snode_1)$, or b) $maxY(snode_1) \leq maxY(snode_2)$ and $minY(snode_1) \geq minY(snode_2)$, or c) $class(snode_2)$ is Open Bracket or Close Bracket and $minY(snode_2) \leq centroidY(snode_1) < maxY(snode_2)$.

MergeRegions $((REGION\_LABEL\_LIST, REGION\_LABEL, SNODE) \rightarrow SNODE')$: For every region label in $region\_label\_list$, find all children of $snode$ that have this label. All of these region nodes are then merged into a single region node labeled $region\_label$.

Overlaps $((SNODE_1, SNODE_2) \rightarrow BOOLEAN')$: Test whether $snode_1$ is a Nonscripted symbol that vertically overlaps $snode_2$. For example, in Fig. 2a, the fraction line overlaps the centroid of the "B." Return true if
   a. $snode_1 \neq snode_2$ and
   b. $class(snode_1) =$ Nonscripted, and
   c. $minX(snode_1) \leq centroidX(snode_2) < maxX(snode_1)$, and
   d. $\neg Contains(snode_2, snode_1)$, and
   e. each of i) and ii) are false: i) $class(snode_2)$ is Open Bracket or Close Bracket, $minY(snode_2) \leq centroidY(snode_1) < maxY(snode_2)$ and $minX(snode_2) \leq minX(snode_1)$ ii) $class(snode_2)$ is Nonscripted or Variable Range and $maxX(snode_2) - minX(snode_2) > maxX(snode_1) - minX(snode_1)$.

Partition$((SNODE\_LIST, SNODE) \rightarrow (SNODE\_LIST', SNODE'))$: The symbol nodes in $snode\_list$ are tested for belonging in regions of $snode$. Symbol nodes that fail the test are returned in list $snode\_list'$. Symbol nodes that pass the test are placed below the appropriate child region nodes of $snode$; the updated subtree is returned as $snode'$.

PartitionFinal $((SNODE\_LIST, SNODE) \rightarrow SNODE')$: The symbol nodes in $snode\_list$ are placed below superscript or subscript region nodes relative to $snode$, where $snode$ is the last symbol on a baseline.

PartitionSharedRegion $((REGION\_LABEL, SNODE_1, SNODE_2) \rightarrow (SNODE\_LIST_1', SNODE\_LIST_2'))$:
If $region\_label$ is $TLEFT$, the symbols in the $TLEFT$ region of $snode_2$ are partitioned into two lists: $snode\_list_1'$

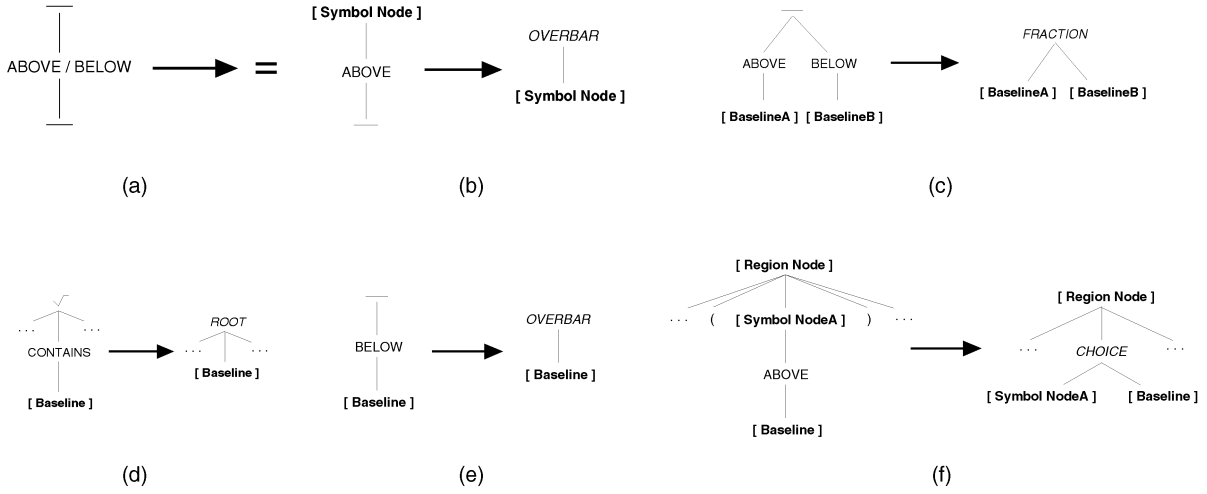


Fig. 7. Examples of Tree Transformation Rules in the Lexical Pass. The pattern (left side) of each rule is searched for depth-first in a BST. If the pattern matches a subtree, the subtree is replaced by the right side of the rule and then searching continues. We use [Region Node] and [Symbol Node] to represent any region node or symbol node respectively. [Baseline] represents a list of symbol nodes, while "..." represents an arbitrary list of region or symbol nodes. Rules (b) and (e) are complementary. There is a complement for rule (f) to recognize simple choice notation where the bottom symbol is found first.

consists of the symbols in the $SUPER$ region of $snode_1$ and $snode\_list'_2$ consists of the symbols in the $UPPER$ region of $snode_2$. Analogous computation is done when $region\_label$ is $BLEFT$, this time partitioning the $BLEFT$ region into a $SUBSC$ and $LOWER$ region (not shown).

1. Let $rnode$ = the child region node of $snode_2$ that has region label $TLEFT$. Let $SL = Symbols(rnode)$.
2. If $class(snode_1)$ = Nonscripted, then Let $snode\_list'_1$ be an empty list.
3. Else if $class(snode_2) \neq$ Variable Range, or $class(snode_2)$ = Variable Range and $HasNonEmpty Region(snode_2, ABOVE)$ is false, then Let $snode\_list'_1 = SL$.
4. Else if $class(snode_2)$=Variable Range and *HasNon EmptyRegions* $(snode_2, ABOVE)$ then $snode\_list'_1 = l_1, l_2, \ldots, l_i)$ where l_1 is the first symbol of SL and l_i is the rightmost symbol in SL such that $IsAdjacent\ (l_i, snode_2)$ holds.
5. Return $(snode\_list'_1, SL - snode\_list'_1)$.

RemoveRegions $((REGION\_LABEL\_LIST, SNODE) \rightarrow SNODE')$: Remove all child region nodes from $snode$ that match any of the labels in $region\_label\_list$.

SortSymbolsByMinX $(SNODE\_LIST \rightarrow SNODE\_LIST')$: Sort $snode\_list$, a list of symbol nodes, into order of increasing $minX$ bounding box coordinate.

Symbols $(RNODE \rightarrow SNODE\_LIST')$: Returns the children of $rnode$ as a list.

4 Lexical Analysis

Following the construction of the Baseline Structure Tree in the Layout Pass, the *Lexical Analysis* pass transforms the BST into a *Lexed BST* using a set of tree transformations that recognizes groups of adjacent input symbols that represent single mathematical symbols. Two kinds of groups of input symbols are recognized in this pass: *compound symbols*, which are single-baseline groups of input symbols that represent a single mathematical symbol (e.g., equal signs, decimal numbers, function names), and *structure symbols*, multibaseline groups of input symbols that imply a mathematical symbol by its local structural context (e.g., fractions, limits, accents on symbols). The result of the Lexical pass is a tree in which these mathematical symbols are explicitly identified for parsing by Expression Analysis in the next pass.

Some example Lexical Analysis transformation rules are shown in Fig. 7. Each of these rules searches the BST for the pattern of a particular mathematical symbol and restructures the tree to provide an explicit label and grouping of the input symbols for the mathematical symbol. As each group of symbols is recognized and relabeled as a mathematical symbol, the bounding box of the recognized unit is computed from its component symbols. Among other uses, these bounding boxes may be used to provide feedback in user interfaces or resolve ambiguities. (Currently, DRACULAE does not make use of these values).

The Lexical Analysis pass is designed to easily accommodate different dialects of mathematics simply by adding or replacing transformation rules for the mathematical symbols of the dialect. For some dialects and mathematical symbols, attention to the ordering of the rules is necessary because the patterns of two or more transformation rules may contain shared symbols or structures or because the pattern of one transformation is only produced after the application of another transformation.

4.1 Compound Symbols

The Lexical Pass begins by applying a set of tree transformation rules to search the BST for compound symbols. Compound symbols are sequences of input symbols on a single baseline that are to be treated as single mathematical symbols, for example the "*s*," "*i*," and "*n*" of the mathematical function name "*sin*," the grouping of sequences of digits into numbers, and the collection of one line above another into an equals sign (Fig. 7a). These

correspond to the treatment of pairs of characters such as "< " and "= " as the single operator "<= " in programming language compilers.

For the mathematical dialect currently recognized by DRACULAE, the Lexical pass recognizes the following compound symbols: decimal numbers (e.g., 1.00, 0.01, .01), function names (e.g., ln, lg, log, exp, sin, cos, tan), and oversegmented symbols (e.g., any of $=, \equiv, \cong, \|, \subseteq, \supseteq, \rightarrow, \leftarrow$).

DRACULAE currently does not use any whitespace analysis. Analysis depends only on input symbol adjacency, not on the amount of whitespace between them. As a result, Lexical Analysis locates function names simply by searching baselines for adjacent letters which form one of the known function names, replacing the group of letters with a single symbol node labeled with the function name. This is adequate when variables and constant names consist of single letters. However, consider "$cost = a * x$," where the current system would identify "$cost$" as as "cos" and "t." In the future we plan to employ whitespace analysis to improve recognition of multiletter function and variable names.

The Lexical pass uses local adjacency to recognize compound symbols. For instance, the two unconnected lines of an equals sign may be represented as two separate lines in the BST, one above or below the other. The Lexical pass uses a tree transformation to search the BST for this pattern, and replaces it with a single symbol node labeled "=" (Fig. 7a). A similar method is used in [26] to detect compound symbols.

4.2 Structure Symbols

Following the recognition of compound symbols, the Lexical pass applies a set of transformation rules to detect structure symbols (Fig. 7b, Fig. 7c, Fig. 7d, Fig. 7e, and Fig. 7f). A structure symbol is a symbol whose role depends on the structure between multiple baselines. Examples are horizontal lines (interpreted as fractions or accents), limits, root signs, and accents. This is analogous to a programming language compiler using context to recognize that a parenthesized subexpression represents an argument list or array index.

Fig. 7b, Fig. 7c, Fig. 7d, Fig. 7e, andFig. 7f show example tree transformation rules used to identify and relabel square roots, fractions, accents, and simple mathematical choice notation in a BST in DRACULAE. At the end of the Lexical pass, normally no region labels remain unless the input BST contains compound and structure symbols not defined in the dialect.

5 Expression Analysis

After Lexical Analysis has identified compound and structure symbols in the Lexed BST, the Expression Analysis pass uses a mathematical expression grammar and a set of tree transformations to create the final *operator tree*. In an operator tree, internal tree nodes are operators and leaf nodes are operands. Operator trees encode all the information necessary to evaluate the represented mathematical expression in the semantics of the dialect.

5.1 Expression Syntax Analysis

The expression grammar specifies the precedence and associativity of mathematical operators in the mathematical dialect using a modification of the traditional context-free expression grammars used in programming language compilers [16]. The Lexed BST produced by the Lexical pass is first linearized into a text string and then parsed using the TXL parser to create an *expression parse tree* analogous to those produced by the syntax pass of a compiler. At present, the DRACULAE expression grammar parses only a subset of the dialect of mathematics recognized by the Lexical pass, but this subset can easily be extended by adding new grammatical forms to the expression grammar.

Expression Analysis returns an error if the parse fails. This could be either because the expression is malformed or because it is outside of the current dialect. Although the expression can always be displayed to the user (because the Layout and Lexical Passes always produce a result), it is inappropriate to evaluate it in these circumstances.

5.2 Expression Semantic Analysis

Semantic analysis consists of analyzing the parsed expression to recognize implied operators (such as adjacent operands meaning multiplication), to analyze the types of operands and infer type conversion operators, and to reorder operands so that they precede their operators in the textual output of the Expression pass. These tasks are achieved using a set of tree transformation rules that search for these patterns in the expression parse tree and then restructure the tree to add the implicit operators and reorder operands. These rules are simplified by the fact that analysis and labeling of structure symbols was already handled in the Lexical pass.

The operator tree output by the Expression Analysis pass is in a form that can be more or less directly translated and executed by a Computer Algebra System such as Mathematica [27] or Maple [28].

6 Test Results

At the time of this writing, the only publicly available ground-truthed set of mathematical expressions is in the University of Washington English/Technical Document Images Database III (UW-III) [29]. The mathematical notation component of UW-III is comprised of 25 ground-truthed document images containing mathematical expressions. Developing methods for evaluating document recognition systems is an active area of research (e.g., [30]). Most of these methods require a large representative corpus of documents with ground truth. In addition to facilitating evaluation, such corpora allow automatic deduction of language definitions and probabilistic contextual information (as has been done for natural language understanding [31], [32]). In contrast, the lack of further corpora of mathematical expressions has resulted in researchers designing recognition systems that describe only a single mathematical dialect, defined using sample expressions and (perhaps largely) introspection. Work is ongoing to establish another corpus of typeset and handwritten mathematical expressions [33].

TABLE 2
UW-III Database Test Results

Threshold Ratio (t)	Centroid Ratio (c) 1/3	1/4
1/3	Tokens Correct: 1642 (86%) Baselines: Correct: 463 (71%) Incorrect: 251 Expressions Correct: 20 (27%)	
1/4	Tokens Correct: 1728 (90%) Baselines: Correct: 513 (79%) Incorrect: 165 Expressions Correct: 28 (38%)	Tokens Correct: 1728 (90%) Baselines: Correct: 513 (79%) Incorrect: 165 Expressions Correct: 28 (38%)
1/6	Tokens Correct: 1679 (87%) Baselines: Correct: 471 (79%) Incorrect: 146 Expressions Correct: 27 (37%)	Tokens Correct: 1679 (87%) Baselines: Correct: 471 (79%) Incorrect: 146 Expressions Correct: 27 (37%)
1/8	Tokens Correct: 1679 (87%) Baselines: Correct: 471 (79%) Incorrect: 146 Expressions Correct: 27 (37%)	Tokens Correct: 1658 (86%) Baselines: Correct: 453 (70%) Incorrect: 155 Expressions Correct: 27 (37%)

Each table entry shows the result of running DRACULAE with different centroid and threshold ratio values c and t (see Table 1). Given are the number of correctly placed tokens, number of correct baselines, number of incorrect baselines, and the number of correct expressions. There are 1,919 tokens, 648 baselines, and 73 expressions in the ground truth. We report the percentage of correct tokens to total ground truth tokens and correct baselines to total ground truth baselines.

Results for mathematical notation recognition have most commonly been presented in terms of recognition success or failure on a small set of sample expressions or using the percentage of correctly recognized expressions in a set of test expressions (e.g., [26], [34]). Recently, some new metrics have been proposed to better characterize errors in baseline structure [35], expression syntax [36], and overall system performance [36].

In Section 6.1, we assess DRACULAE's Lexed BST recognition performance on the UW-III database using two new metrics for baseline structure accuracy, namely, 1) the ratio of correctly recognized baselines to total baselines in the ground truth representation of an expression and 2) the percentage of symbols or tokens in a BST that are located on their correct baselines. In Section 6.2, we describe some informal results concerning the performance and usability of DRACULAE, using the user interface and symbol recognizer provided by the Freehand Formula Entry System [18], [19].

6.1 Results for Typeset Expressions in UW-III

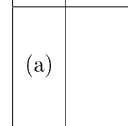
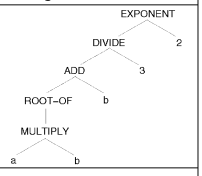
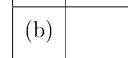
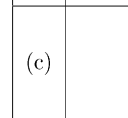
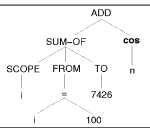
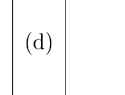
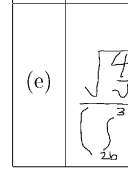
We used the UW-III symbol and bounding box ground truth data to test DRACULAE's Lexed BST output (see Section 4). This test data was not used during system development. The test set was made using the symbol and bounding box ground truth for 23 of the 25 pages in the database (pages 20 and 21 were removed as they contained matrix expressions). Expressions spanning multiple lines were manually broken into separate subexpressions. The final test set contained 73 expressions comprised of 1,917 input symbols, with a mean of 26.3 symbols per expression. The LaTeX ground truth for these expressions contained 648 baselines with 1,919 *tokens*, with means of 3.0 tokens per baseline and 8.9 baselines per expression. Tokens do not always correspond to input symbols, primarily due to groupings of letters in function names (e.g., "sin," "ln") and to accented symbols. Symbols and their accents are often ground truthed as a single symbol in UW-III, though they are represented as two tokens (symbol and accent) in the LaTeX ground truth. Most baselines contain few symbols: 62 percent are comprised of a single token, while 84 percent are comprised of three tokens or less.

We ran DRACULAE using a series of values for the two layout model parameters, t and c. For each test expression we compared DRACULAE's LaTeX output to the UW-III ground truth LaTeX. This comparison was done using a TXL program, as explained below. Table 2 shows the parameter values that were used. No test was performed for $c = 1/4, t = 1/3$ as this results in subscript regions that are higher than the Y-centroid for symbol classes such as Plain Ascender and Descender.

For each test expression, a context-free grammar specified in TXL is used to parse the DRACULAE output and UW-III ground truth LaTeX representations. The parse trees are then compared. Identical trees correspond to perfect structure (Lexed BST) recognition. For nonmatching trees, the TXL program outputs a list of baseline pairs corresponding to the two trees, starting with the first mismatching pair. This list is used, along with images corresponding to the LaTeX strings produced by DRACULAE and the original document images, to manually locate additional errors.

We count errors of two types. The first is the *number of incorrect baselines*, where an incorrect baseline is one in which any of the following are true: 1) The list of tokens on the baseline do not match ground truth, 2) the baseline is nested relative to a token which does not match the ground truth token, or 3) the depth in the BST or region of the baseline does not match ground truth. Table 2 shows the number of properly recognized baselines.

TABLE 3
Recognition Results for Sample Expressions Created Using the Freehand Formula Entry System (FFES)

	Input Expression	LaTeX (Lexed BST)	Operator Tree
(a)		$(\frac{\sqrt{ab}+b}{3})^2$	
(b)		$\overline{a} \vee \overline{b \vee c}$	(Outside Dialect)
(c)		$\sum_{i=100}^{7426} i + \cos n$	
(d)		Z $A_B \sum_{i=}^{2} \sum_{1_j=2}^{2} ij$	(Outside Dialect)
(e)		$\frac{\sqrt{\frac{4x^2}{\sqrt{2a}}}^{4^{2^3}+\frac{3}{4}}}{(\int_{2b}^{3} \frac{x^2}{4+a} dx)^3 x^2} + \frac{-b+\sqrt{b^2-4ac}^2}{2a}$	(Outside Dialect)

For these expressions, threshold ratio $t = 1/6$ and centroid ratio $c = 1/4$ were used.

The second type of error is the *number of misplaced tokens*. A token is *misplaced* if it appears on a baseline other than that in the ground truth. A *properly placed* token appears on the same baseline, at the same depth, in the same region (e.g., superscripted or subscripted), and nested relative to the same parent token as in the ground truth. According to this definition, a token may be *properly placed* on an *incorrect* baseline. Table 2 shows the number of properly placed tokens. Measuring tokens provides a more informative measure than measures based on entire expressions or baselines.

The total number of expressions recognized without error is low. However, the percentage of properly placed tokens is 86-90 percent. This means that DRACULAE places most symbols in the test set on their proper baseline.

The most common source of errors is misdetection of scripted and horizontally adjacent symbols. The definition of superscript, subscript and horizontal regions for descending class symbols in our current symbol layout model appears to be particularly poor. For example, when a "p" is followed on a baseline by a Plain Ascender symbol, this is often misdetected as a superscript.

Other errors include:

1. misdetection of kerned symbols as below rather than subscripted relative to the parent symbol,
2. misdetection of a close bracket as below instead of to the right of a fraction line, because the centroid of the bracket is below the fraction line,
3. a bug in the partitioning routines that yields two additional tokens for one of the expressions, and
4. a small number of additional tokenization errors.

The tests consist of 73 expressions, run for seven combinations of t and c values, for a total of 13,419 input symbols. These tests took 206 seconds to execute as a batch process on a 900MHz Pentium III with 256MB of RAM running Linux. This rate of 65 symbols per second includes the time taken for TXL to reinterpret the DRACULAE source code seven times.

A number of researchers have recently reported properly recognizing the symbol layout of over 90 percent of the mathematical expressions in their test sets [26], [33], [37], [38], [39], [40]. It is difficult to meaningfully compare these results. The test sets used by other authors are generally not publicly available. Also, different authors use different metrics. We view our metrics as an important new tool for evaluating recognition results at a level that is between symbol recognition and operator trees.

A system that makes use of more layout information, such as whitespace and point size information, and/or more sophisticated contextual analyses would perform better than our current system. It is interesting how well DRACULAE is able to perform without such information. In the future we plan to extend our layout model. Some alternative approaches to analyzing layout in mathematical expressions include penalty functions [38], projection profiles [26], defining "strong" and "weak" region areas using a training set [37], virtual link networks [37], [39], convex hulls [7], the generation of multiple interpretations to cope with ambiguity [41], and the incorporation of probabilistic information [20], [42].

We are also beginning to explore recognition of tabular structures such as matrices and lists of expressions [43]. Existing approaches to matrix recognition include [37], [39], [40], [44].

6.2 Testing DRACULAE on Handwritten Expressions Using FFES

We have informally tested DRACULAE's recognition capabilities for handwritten mathematical notation. These tests use the user interface and symbol recognition portions of the Freehand Formula Entry System (FFES) [18], [19]. This

interface allows a user to enter, delete, move, and relabel symbols. DRACULAE is given the current list of symbols with bounding boxes and provides an interpretation of the current expression. Sample expressions are provided in Table 3. Each of the expressions in this table are processed by DRACULAE in well under a second.

Table 3 shows that DRACULAE is robust: All inputs are mapped to LaTeX output. Lexed BST (and LaTeX) output is produced, even if an expression contains unknown and/or unsyntactic baseline structures, as in Table 3b, Table 3d, and Table 3e. The use of operator dominance in the search functions provides some skew tolerance (Table 3b). Large, deeply nested expressions (Table 3e) and nested accents (Table 3b) are handled.

Operator trees are produced for expressions that fall within the dialect defined in the current Expression Analysis pass. In Table 3a, the implicit multiplication of "a" and "b" is made explicit in the operator tree.

The disambiguation of *SUPER/UPPER* and *SUBSC/LOWER* regions is fragile. For example, the limits of adjacent Variable Range Symbols are improperly segmented in Table 3d, where the "1" is mistakenly grouped with the second "$\sum$" symbol. (The LaTeX string has been altered to make this error easily visible.) Analysis of whitespace would correct many such errors.

Some usability results for FFES/DRACULAE were obtained in an experiment comparing online expression entry time using different feedback mechanisms [19]. All 27 participants in the experiment successfully entered the trial expressions and all reported that they found bitmaps produced from DRACULAE's LaTeX output to be useful. Twenty four of the participants (89 percent) reported that they were interested in using a similar system again.

7 Conclusion

We have presented a methodology and implementation (DRACULAE) for rapid, robust recognition of typeset and handwritten mathematical expressions. DRACULAE makes use of search functions that exploit the left-to-right reading order of mathematical notation and operator dominance to recursively and efficiently extract baselines in a mathematical expression.

The Baseline Structure Tree (BST) is a simple hierarchical description of symbol layout in mathematical expressions. Tree transformation is used as an efficient, compact means to express a series of restructurings from an initial list of symbols to an initial BST, to a Lexed BST (translatable to LaTeX), and, finally, to an operator tree (translatable to Computer Algebra System languages).

DRACULAE's architecture is similar to that of a compiler. This provides a framework for coping with dialects, by separating symbol layout analysis, lexical grouping, syntax analysis, and semantic analysis. This architecture also makes the system easy to reconfigure.

Acknowledgments

The authors would like to thank Edward Lank, Nick Willan, and Genarro Costagliola for valuable discussions and inspiration. We are grateful to Steve Smithies, Kevin Novins, and Jim Arvo for making the Freehand Formula Entry System available. We also wish to thank the anonymous reviewers for their helpful comments. This research is supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] D. Blostein and A. Grbavec, "Recognition of Mathematical Notation," *Handbook of Character Recognition and Document Image Analysis,* pp. 557-582, World Scientific, 1997.

[2] K. Chan and D. Yeung, "Mathematical Expression Recognition: A Survey," *Int'l J. Document Analysis and Recognition,* vol. 3, no. 1, pp. 3-15, Aug. 2000.

[3] A. Kacem, A. Belaïd, and M.B. Ahmed, "Automatic Extraction of Printed Mathematical Formulas Using Fuzzy Logic and Propogation of Context," *Int'l. J. Document Analysis and Recognition,* vol. 4, no. 2, pp. 97-108, Dec. 2001.

[4] R.J. Fateman, "How to Find Mathematics on a Scanned Page," *Proc. SPIE,* vol. 3967, pp. 98-109, 1999.

[5] B.P. Berman and R.J. Fateman, "Optical Character Recognition for Typeset Mathematics," *Proc. Int'l Symp. Symbolic and Algebraic Computation,* pp. 348-353, 1994.

[6] Z.X. Wang and C. Faure, "Structural Analysis of Handwritten Mathematical Expressions," *Proc. Ninth Int'l Conf. Pattern Recognition,* pp. 32-34, 1988.

[7] E.G. Miller and P.A. Viola, "Ambiguity and Constraint in Mathematical Expression Recognition," *Proc. 15th Nat'l Conf. Artificial Intelligence,* pp. 784-791, 1998.

[8] R. Zanibbi, "Recognition of Mathematics Notation via Computer Using Baseline Structure," Technical Report, ISBN-0836-0227-2000-439, School of Computing, Queen's Univ., Aug. 2000.

[9] S. Srihari, "From Pixels to Paragraphs: The Use of Contextual Models in Text Recognition," *Proc. Second Int'l. Conf. Document Analysis and Recognition,* pp. 416-423, 1993.

[10] T.W. Chaundy, P.R. Barrett, and C. Batey, *The Printing of Mathematics.* London: Oxford Univ. Press, 1957.

[11] N.J. Higham, *Handbook of Writing for the Mathematical Sciences.* Philadelphia: SIAM, 1993.

[12] D.E. Knuth, TeX *and METAFONT—New Directions in Typesetting.* Digital Press, 1979.

[13] G. Costagliola, A. De Lucia, and S. Orefice, "A Parsing Methodology for the Implementation of Visual Systems," *IEEE Trans. Software Eng.,* vol. 23, no. 12, pp. 777-799, Dec. 1997.

[14] H. Lee and M. Lee, "Understanding Mathematical Expressions Using Procedure-Oriented Transformation," *Pattern Recognition,* vol. 27, no. 3, pp. 447-457, 1994.

[15] K. Inoue, R. Miyazaki, and M. Suzuki, "Optical Recognition of Printed Mathematical Documents," *Proc. Third Asian Technology Conf. Math.,* pp. 280-289, 1998.

[16] A. Aho, V. Jeffrey, and D. Ullman, *Principles of Compiler Design.* Addison-Wesley, 1977.

[17] R. Zanibbi, D. Blostein, and J.R. Cordy, "Baseline Structure Analysis of Handwritten Mathematics Notation," *Proc. Sixth Int'l Conf. Document Analysis and Recognition,* pp. 768-773, 2001.

[18] S. Smithies, K. Novins, and J. Arvo, "Equation Entry and Editing via Handwriting and Gesture Recognition," *Behaviour and Information Technology,* vol. 20, no. 1, pp. 53-67, 2001.

[19] R. Zanibbi, K. Novins, J. Arvo, and K. Zanibbi, "Aiding Manipulation of Handwritten Mathematical Expressions through Style-Preserving Morphs," *Proc. Graphics Interface,* pp. 127-134, 2001.

[20] P.A. Chou, "Recognition of Equations Using a Two-Dimensional Stochastic Context-Free Grammar," *Visual Comm. and Image Processing IV,* pp. 852-863, 1989.

[21] A. Grbavec and D. Blostein, "Mathematics Recognition Using Graph Rewriting," *Proc. Third Int'l Conf. Document Analysis and Recognition,* pp. 417-421, 1995.

[22] J.R. Cordy, I. Charmichael, and R. Halliday, *The TXL Programming Language-Version 10,* Jan. 2000.

[23] J.R. Cordy, C.D. Halpern, and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects," *Computer Languages,* vol. 16, no. 1, pp. 97-107, Jan. 1991.

[24] S. Chang, "A Method for the Structural Analysis of Two-Dimensional Mathematical Expressions," *Information Sciences,* vol. 2, pp. 253-272, 1970.

[25] R.H. Anderson, "Syntax-Directed Recognition of Hand-Printed Two-Dimensional Equations." PhD thesis, Harvard Univ., Jan. 1968.
[26] M. Okamoto and B. Miao, "Recognition of Mathematical Expressions by Using the Layout Structures of Symbols," *Proc. First Int'l Conf. Document Analysis and Recognition,* vol. 1, pp. 242-250, 1991.
[27] S. Wolfram, *The Mathematica Book, version 4.* Cambridge Univ. Press, 1999.
[28] F. Garvan, *The MAPLE Book.* CRC Press, 2001.
[29] I. Phillips, "Methodologies for Using UW Databases for OCR and Image Understanding Systems," *Document Recognition V, SPIE Proc.,* vol. 3305, pp. 112-127, 1998.
[30] I. Phillips and A. Chhabra, "Empirical Performance Evaluation of Graphics Recognition Systems," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 21, no. 9, pp. 849-870, Sept. 1999.
[31] E. Brill, "Transformation-Based Error-Driven Parsing," *Recent Advances in Parsing Technology,* pp. 1-13, Kluwer Academic, 1996.
[32] E. Charniak, *Statistical Language Learning.* MIT Press, 1993.
[33] U. Garain and B.B. Chaudhuri, "On Development and Statistical Analysis of a Corpus for Printed and Handwritten Expressions," *Proc. Fourth Int'l IAPR Workshop Graphics Recognition,* pp. 429-439, Sept. 2001.
[34] A. Belaïd and J. Haton, "A Syntactic Approach for Handwritten Mathematical Formula Recognition," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 6, no. 1, pp. 105-111, Jan. 1984.
[35] M. Okamoto, H. Imai, and K. Takagi, "Performance Evaluation of a Robust Method for Mathematical Expression Recognition," *Proc. Sixth Int'l Conf. Document Analysis and Recognition,* pp. 121-128, 2001.
[36] K. Chan and D. Yeung, "Error Detection, Error Correction and Performance Evaluation in On-Line Mathematical Expression Recognition," *Pattern Recognition,* vol. 34, pp. 1671-1684, 2001.
[37] Y. Eto and M. Suzuki, "Mathematical Formula Recognition Using Virtual Link Network," *Proc. Sixth Int'l Conf. Document Analysis and Recognition,* pp. 762-767, 2001.
[38] R. Fukuda, S. I, F. Tamari, X. Ming, and M. Suzuki, "A Technique of Mathematical Expression Structure Analysis for the Handwriting Input System," *Proc. Fifth Int'l Conf. Document Analysis and Recognition,* pp. 131-134, 1999.
[39] T. Kanahori and M. Suzuki, "A Recognition Method of Matrices by Using Variable Block Pattern Elements Generating Rectangular Area," *Proc. Fourth Int'l IAPR Workshop Graphics Recognition,* pp. 455-469, 2001.
[40] K. Toyozumi, T. Suzuki, K. Mori, and Y. Suenega, "A System for Real-Time Recognition of Handwritten Mathematical Formulas," *Proc. Sixth Int'l Conf. Document Analysis and Recognition,* pp. 1059-1063, 2001.
[41] H. Winkler, H. Fahrner, and M. Lang, "A Soft Decision Approach for Structural Analysis of Handwritten Mathematical Expressions," *Proc. Int'l Conf. Acoustics, Speech and Signal Processing,* pp. 2459-2462, 1995.
[42] C. Faure and Z.X. Wang, "Automatic Perception of the Structure of Handwritten Mathematical Expressions," *Computer Processing of Handwriting,* R. Plamondon and C.G. Leedham, eds., pp. 337-361, World Scientific, 1990.
[43] R. Zanibbi, D. Blostein, and J.R. Cordy, "Directions in Recognizing Tabular Structures of Handwritten Mathematics Notation," *Proc. Fourth Int'l IAPR Workshop Graphics Recognition,* pp. 493-499, Sept. 2001.
[44] R.H. Anderson, "Two-Dimensional Mathematical Notation," *Syntactic Pattern Recognition,* K.S. Fu, ed., pp. 147-177, New York: Springer Verlag, 1977.

Richard Zanibbi received the Bachelor of Music and BA degrees in 1998 and the MSc degree in computing science in 2000 from Queen's University, Kingston, Ontario, Canada, where he is currently pursuing a PhD in computing science. He has worked as a research programmer for Legasys Corporation (1999-2000) and for the Medical Computing Laboratory at Queen's University (2001-present). His research interests include diagram recognition, visual language theory, pattern recognition, and human-computer interaction. He is a student member of the IEEE Computer Society.

Dorothea Blostein received the MSc degree in computer science from Carnegie Mellon University in 1980, and the PhD degree from the University of Illinois in 1987. Since 1988, she has been a professor in the School of Computing, Queen's University, Kingston, Ontario. Dr. Blostein's research interests include pattern recognition and document image analysis. A particular interest is the relationship between generation and recognition of diagrams. She is coauthor of Lime, a sophisticated editor for music notation. Dr. Blostein was chair of GREC 2001, the Fourth IAPR International Workshop on Graphics Recognition. She has served on the program committees for ICPR 2002, Diagrams 2002, WEDELMUSIC 2001, GREC '99, AGTIVE '99, and GREC '97. Dr. Blostein is a member of the IEEE and the IEEE Computer Society.

James R. Cordy is a professor and head of the School of Computing at Queen's University, Kingston, Ontario, Canada. From 1995 to 2000, he was vice president and chief research scientist at Legasys Corporation, a software technology company specializing in legacy software system analysis and renovation. Professor Cordy is the author or coauthor of numerous contributions in computer software systems, including the PL/I subset compiler SP/k (1977), the Toronto Euclid compiler (1980), the S/SL compiler technology (1980), the Concurrent Euclid programming language (1981), the Turing programming language (1983), Turing Plus (1985), Object-Oriented Turing (1992), the orthogonal code generation compiler technology (1986), the TXL programming language (1991), the TXL source transformation system (1995), the LS/2000 year 2000 conversion system (1996), and the LS/AMT software analysis and migration system (1999). He is a member of the IEEE.