Towards Efficient Structural Analysis of Mathematical Expressions

Kam-Fai Chan & Dit-Yan Yeung

Department of Computer Science
Hong Kong University of Science and Technology

Abstract. Machine recognition of mathematical expressions is not trivial even when all the individual characters and symbols in an expression can be recognized correctly. In this paper, we propose to use Definite Clause Grammar (DCG) as a formalism to define a set of replacement rules for parsing mathematical expressions. With DCG, we are not only able to define the replacement rules concisely, but their definitions are also in a readily executable form. However, backtracking parsers like Prolog interpreters, which execute DCG directly, are by nature inefficient. Thus we propose some methods here to increase the efficiency of the parsing process. Experiments done on some typical mathematical expressions show that our proposed methods can achieve speedup ranging from 10 to 70 times, making mathematical expression recognition more feasible for real-world applications.

Keywords: Definite Clause Grammar, document processing, mathematical expression recognition, structural analysis

1 Introduction

Many documents in science and engineering disciplines contain mathematical expressions. The input of mathematical expressions into computers is often more difficult than the input of plain text, because mathematical expressions typically consist of special symbols and Greek letters in addition to English letters and digits. With such a large number of characters and symbols, the commonly used type of keyboard has to be specially modified in order to accommodate all the keys needed, as done in [6]. Another method is to define a set of keywords to represent special characters, as in LaTeX [7]. However, working with specially designed keyboards or keywords requires intensive training. Alternatively, by taking advantage of pen-based computing technologies, one could simply write mathematical expressions on an electronic tablet and the computer will be able to recognize them.

Mathematical expression recognition consists of two major stages: *character recognition* and *structural analysis*. Character recognition has been an active research area for more than three decades. Structural analysis of two-dimensional patterns also has a long history [4]. However, as emphasized in [2, 5, 8], very few papers have addressed specific problems related to mathematical expression recognition.

In a mathematical expression, characters and symbols can be spatially arranged as a complex two-dimensional structure, possibly of different character and symbol sizes. This makes the recognition process more complicated even when all the individual characters and symbols can be recognized correctly. Moreover, to ensure that a mathematical expression recognition system is useful in practice, its recognition speed should also be taken into consideration.

In this paper, we will mainly focus on the structural analysis aspect of mathematical expression recognition. First of all, we will discuss some problems which have to be overcome during structural analysis. Afterwards, we will explain how our proposed methods work through use of an illustrative example. Finally, some experimental results will be presented.

2 Problems in Structural Analysis of Mathematical Expressions

Mathematical expressions are two-dimensional structures. This nature and some other properties make their recognition non-trivial in many ways. Here are some examples:

1. The relationships among symbols in a mathematical expression sometimes depend on their relative positions. For example, in the expression "a^2", 2 is the superscript of a representing the square of a. However, in "a_2", 2 is the subscript of a representing only a variable name. Although it is somewhat unusual, "$a2$" can be used to represent the multiplication of a and 2.
2. The same group of characters can have different meanings in different contexts. For example, "dx" has different meanings in "$\int x^2 dx$" and in "$cy+dx$".

These problems have to be taken into consideration when we process mathematical expressions in the following steps.

2.1 Grouping Symbols

Before we can interpret the symbols, we must first group them properly into units. This can be done by using some conventions in writing mathematical expressions as heuristics. Some of these conventions are as follows:

1. Digits which together form a unit should be of the same size and be written on the same horizontal line. For example, 210 is only one unit but 2^{10} consists of two units which are 2 and 10 respectively.
2. Some letters together may form a unit, like some trigonometric functions such as `tan`, `sin` and `cos`. Before considering a group of letters as a concatenation of variables, we have to first check whether they are in fact some predefined function names.
3. Symbols other than letters and digits should be considered as separate units.

2.2 Determining Relationships Among Symbols

Determining the relationships among symbols, to some extent, can be viewed as grouping several smaller units into one larger unit. Again, some conventions can be used as heuristics:

1. Some fence symbols, such as parentheses, group the enclosed units as one single unit. For example, $(a+b)$ is a unit which holds the sum of a and b.
2. Some binding symbols, like fraction line, $\sqrt{\ }$ and $\sum$, dominate their neighboring expressions. For example, in $\sum_{i=1}^{10} i$, three units, i.e., 10, $i = 1$, and i are bound to the symbol $\sum$ which gives meaning to the expression as the sum of 1, 2, ..., 10.
3. The ideas of operator precedence and operator dominance [4] can also be used for grouping units. For example, in $a + \frac{b}{c}$, the meaning becomes $a + (b/c)$ due to the fact that / has higher precedence than +. The operator + is said to dominate /. However, in $\frac{a+b}{c}$, the meaning becomes $(a+b)/c$ since / dominates + in this case.

3 Parsing with Binding Symbol Preprocessing and Hierarchical Decomposition

Most previous works in mathematical expression recognition did not put much emphasis on explaining how the replacement rules are used for structural analysis, or the explanations are too tedious and sometimes too *ad hoc* [1, 4, 11]. To remedy such weaknesses, we propose to use Definite Clause Grammar (DCG) [10] as a formalism to describe our set of replacement rules for parsing mathematical expressions. Note that a grammar written in DCG is highly declarative and can be directly executed by a Prolog interpreter.

However, backtracking parsers, like Prolog interpreters, are known to be inefficient. This makes most backtracking parsers not useful in practice for the recognition of mathematical expressions. In this paper, we propose some methods for increasing the efficiency of the parsing process.

3.1 Basic Notations for DCG

DCG is similar to BNF, with some minor notational differences as follows:

1. ":::=" is replaced by "-->".
2. Non-terminals are not put inside brackets any more. Instead, terminals are now in square brackets.
3. Symbols are separated by commas and each rule is terminated by a full stop.

There is a major difference between DCG and BNF though. In DCG, some Prolog predicates (enclosed inside { }) can be put in the body of any rule so that the semantics of a rule can be incorporated into its syntax.

3.2 Conventional Backtracking Parsing

The simplest way of parsing a two-dimensional expression is to translate it into its equivalent one-dimensional representation and then parse it with an existing parser. Since there already exist many compilers or interpreters for parsing string-based mathematical expressions, some extra work can be saved by taking this approach. Fig. 1 shows an example of such translation.

$$\frac{6\ x\ +\ 4\ y}{2} = 3\ x + 2\ y \longrightarrow (6\,x+4\,y)/2=3\,x+2\,y$$

Fig. 1. Translating an expression from its two-dimensional form into a one-dimensional representation

Notice that it usually takes comparatively long time for a backtracking parser to return the tree structure of an expression, because some sub-structures may have to be re-generated again and again during the backtracking steps. Therefore, the bigger the structure is, the longer time it takes. Fig. 2 depicts the tree structure for the expression in Fig. 1.

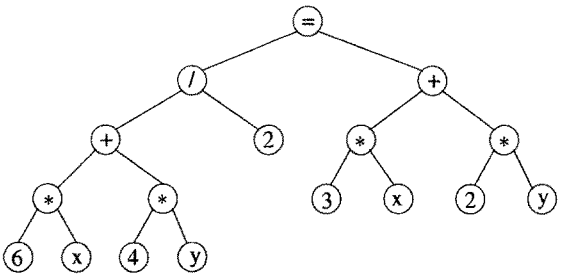

Fig. 2. Tree structure of the mathematical expression in Fig. 1

3.3 Parsing with Binding Symbol Preprocessing

As mentioned in Section 2, binding symbols always dominate their neighbors. For example, in the previous expression, the fraction line in $\frac{6x-4y}{2}$ dominates the sub-expressions $6x-4y$ and 2. Instead of putting them in a one-dimensional form for further parsing, we can directly parse the two sub-expressions first and then construct the final structure of the fraction from the intermediate results. The resulting structure will be stored in memory, with a name introduced to

denote the fraction that the structure represents. There is no need to generate the structure for this fraction again during the subsequent processing.

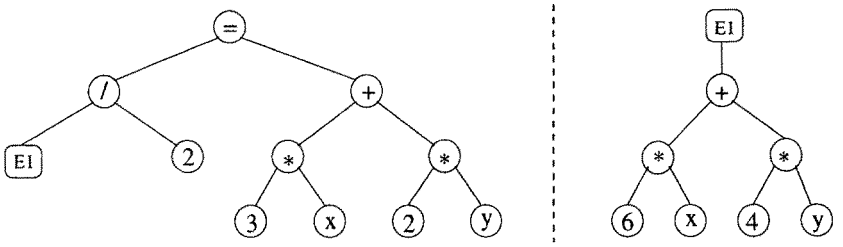

Fig. 3. Tree structures generated as a result of parsing with binding symbol preprocessing

The resulting tree structures are shown in Fig. 3. As shown, the original tree structure is now partitioned into two sub-structures. This eliminates some repeated generation steps, and therefore can lead to significant speedup.

3.4 Parsing with Hierarchical Decomposition

The above idea can be extended to further partition the sub-structures into even smaller structures. Instead of parsing the entire expression, we will parse all the sub-expressions first and then parse the resulting expression. This idea is similar to hierarchical decomposition in planning [12].

Sub-expressions are detected using the following rules:

1. Parentheses have higher precedence than the other operators. Whatever enclosed inside a pair of parentheses should form an expression.
2. Subscript, superscript and implicit multiplication have higher precedence than arithmetic operators.

With these, we can perform some preprocessing steps for finding sub-expressions. Each sub-expression is then parsed separately. Afterwards, we can compose the final tree structure from a set of sub-structures. Fig. 4 shows the resulting sub-structures for the same expression shown before.

In order to demonstrate how declarative the DCG rules can be, we list below all the relevant DCG rules for parsing with hierarchical decomposition.

The following shows the DCG rules for preprocessing. Basically, we keep the concatenation of non-operators in a reverse order. Once an equality sign or arithmetic operator is met, it can then process the list as a sub-expression. Note that parentheses will be processed first since they have higher precedence.

```
preprocess(A, [E, = | C]) --> [=], preprocess([], C),
        { reverse(A, B), factor(E, B, []) }.
preprocess(A, [E, + | C]) --> [+], preprocess([], C),
```

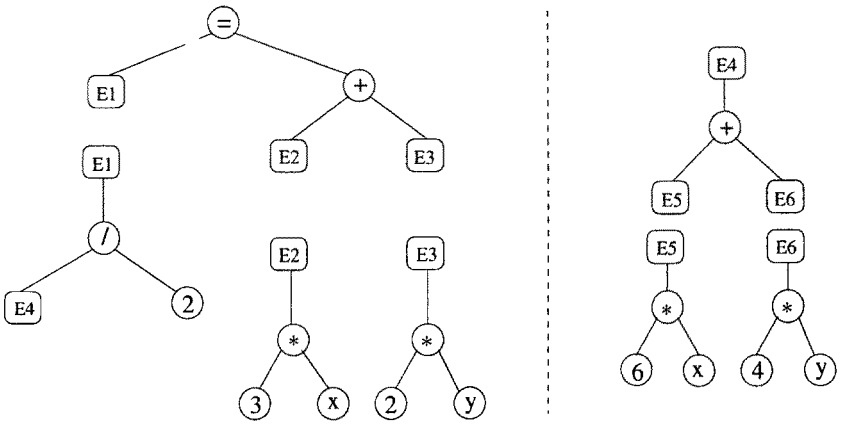

Fig. 4. Tree structures generated as a result of parsing with hierarchical decomposition

```
        { reverse(A, B), factor(E, B, []) }.
preprocess(A, [E, - | C]) --> [-], preprocess([], C),
        { reverse(A, B), factor(E, B, []) }.
preprocess(A, [E, * | C]) --> [*], preprocess([], C),
        { reverse(A, B), factor(E, B, []) }.
preprocess(A, [E, / | C]) --> [/], preprocess([], C),
        { reverse(A, B), factor(E, B, []) }.
preprocess(A, C) --> ['('], paren([], E2), preprocess([E2 | A], C).
preprocess(A, C) --> [B], preprocess([B | A], C).
preprocess(A, [E]) --> epsilon,
        { A == [], E == []; reverse(A, B), factor(E, B, []) }.
```

The following are the major DCG rules used for parsing the expressions. They are self-explanatory. In both `parse_equation(A)` and `parse_expr(A)`, we perform preprocessing before doing the original parse.

```
parse_equation(A) --> preprocess([], B), { equation(A, B, []) }.
parse_expr(A) --> preprocess([], B), { expr(A, B, []) }.
equation([=, A, B]) --> expr(A), [=], expr(B).
expr([+, A, B]) --> term(A), [+], expr(B).
expr([-, A, B]) --> term(A), [-], expr(B).
expr(A) --> term(A).
term([*, A, B]) --> factor(A), [*], term(B).
term([/, A, B]) --> factor(A), [/], term(B).
term(A) --> [A], { is_expr(A) }.
factor(A) --> [A], { is_expr(A) }.
```

4 Experimental Results

In this experiment, we use the character recognition method proposed in [3]. All the characters and symbols recognized are converted to objects with associated

attributes, including location, size, and identity. Note that the objects can be put in arbitrary order for our subsequent processing.

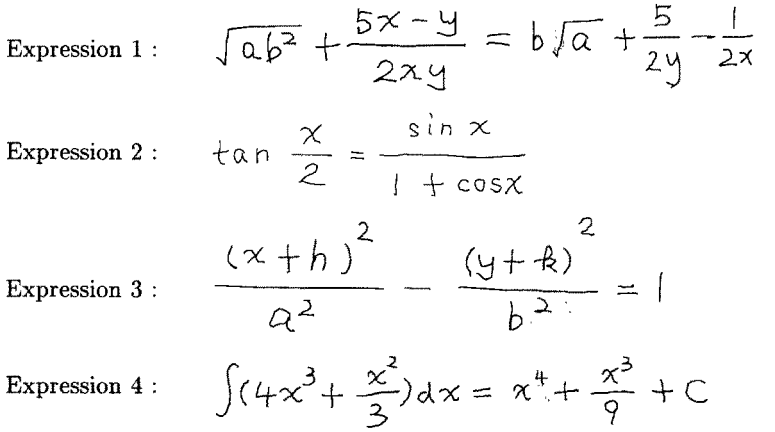

Fig. 5. Some expressions used in the experiment

The next step is to group the objects. Here we use a method similar to the one used in [9]. Afterwards, we perform parsing using different techniques as described above and then compare their efficiency.

We have tested a number of different expressions. Four typical examples are shown here for illustration (Fig. 5). Our recognition system implemented in Prolog runs on a Sun SPARC 2 workstation. The recorded time starts when the list of objects is passed to the parsing procedure and ends when the final structure is returned.

Expression	Time in seconds required for the parsing		
	Conventional backtracking parsing	Parsing with binding symbol preprocessing	Hierarchical decomposition parsing with binding symbol preprocessing
1	1.97	0.27	0.17
2	4.03	0.32	0.07
3	8.63	0.98	0.13
4	8.68	1.63	0.12

Notice that the speedup achieved by hierarchical decomposition parsing with binding symbol preprocessing is very significant. It ranges from 10 to 70 times for the test cases shown above. Other examples tested also show similar speedup performance.

5 Conclusion

Pen-based computing offers us a natural human-computer interface, such as an on-line mathematical editor. Such an editor, however, cannot be put into practical use without a sophisticated mathematical expression recognition subsystem.

In this paper, we have proposed and demonstrated some methods for defining replacement rules in a clear and concise manner for parsing mathematical expressions. More importantly, it manages to offer the much needed speed for practical use.

Since our methods do not make use of stroke order information, they can also be used for off-line mathematical expression recognition. However, some problems in mathematical expression recognition are not addressed in this paper, including ambiguity resolution, error detection, and error correction. These are issues to be addressed in our future research.

References

1. R. H. Anderson. Syntax-directed recognition of hand-printed 2-D mathematics. In M. Klerer and J. Reinfelds, editors, *Interactive Systems for Experimental Applied Mathematics*, pages 436–459. Academic Press, New York, 1968.
2. A. Belâid and J.-P. Haton. A syntactic approach for handwritten mathematical formula recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(1), Jan. 1984.
3. K. F. Chan. and D. Y. Yeung. Elastic structural matching in recognizing on-line handwritten alphanumeric characters. Technical Report HKUST-CS98-07, Dept. of Computer Science, Hong Kong University of Science and Technology, Mar. 1998.
4. S. K. Chang. A method for the structural analysis of 2-D mathematical expressions. *Information Sciences*, 2(3):253–272, 1970.
5. Y. A. Dimitriadis and J. L. Coronado. Towards an ART based mathematical editor, that uses on-line handwritten symbol recognition. *Pattern Recognition*, 28(6):807–822, 1995.
6. F. Grossman, R. J. Klerer, and M. Klerer. A language for high-level programming of mathematical applications. *Proceedings of the International Conference on Computer Languages*, pages 31–40, Miami Beach, FL, 1988.
7. L. Lamport, editor. *LaTeX – A Document Preparation System – User's Guide and Reference Manual.* Addison-Wesley, Reading, MA, 1985.
8. H.-J. Lee and M.-C. Lee. Understanding mathematical expressions using procedure-oriented transformation. *Pattern Recognition*, 27(3):447–457, 1994.
9. M. Okamoto and A. 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, Berlin, 1992.
10. F. Pereira and D. Warren. Definite clause grammars for language analysis – a survey of the formalism and comparison with augmented transition networks. *Artificial Intelligence*, 13, 1980.
11. J. J. Pfeiffer, Jr. Parsing graphs representing two dimensional figures. *Proceedings of the IEEE Workshop on Visual Languages*, pages 200–206, Seattle, WA, 1992.
12. S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, N.J., 1995.