



An efficient syntactic approach to structural analysis of on-line handwritten mathematical expressions

Kam-Fai Chan, Dit-Yan Yeung*

Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

Received 29 September 1998; accepted 2 March 1999

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, a DCG parser is potentially inefficient due to its frequent use of backtracking. Thus, we propose some methods here to increase the efficiency of the parsing process. Experiments done on some commonly seen mathematical expressions show that our proposed methods can achieve quite satisfactory speedup, making mathematical expression recognition more feasible for real-world applications. © 2000 Pattern Recognition Society. Published by Elsevier Science Ltd. All rights reserved.

Keywords: Definite clause grammar; Document processing; Mathematical expression recognition; Structural analysis

1. Introduction

Many documents in scientific 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 Ref. [1]. Another method is to define a set of keywords to represent special characters, as in LATEX [2]. However, working with specially designed keyboards or keywords requires intensive training. Alternatively, by taking advantage of pen-based computing technologies, one can simply write mathematical expressions on an electronic tablet for the computer to recognize them.

Mathematical expression recognition consists of two major stages: *symbol recognition* and *structural analysis*. Character recognition, as the most common type of symbol recognition problems, has been an active research area for more than three decades [3]. Structural analysis of two-dimensional patterns also has a long history [4]. However, as emphasized in Refs. [5–7], very few papers have addressed specific problems related to mathematical expression recognition.

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

It is well known that parsing can be done in polynomial time with Earley's algorithm [8] while most of the other types of parsers take exponential time. However, as Covington [9] tried to argue, exponential parsers can be fast when the length of the sentence to parse is short. Also, a long sentence can usually be broken up into shorter sentences that can be parsed separately.

*Corresponding author. Tel.: + 852-2358-6977; fax: + 852-2358-1477

E-mail address: dyyeung@cs.ust.hk (D-Y. Yeung)

In this paper, we will mainly focus on the structural analysis aspect of mathematical expression recognition. First of all, we will review some related work. Then, we will discuss some problems that have to be overcome during the structural analysis stage. Afterwards, we will propose to use definite clause grammar (DCG) as a formalism to define a set of replacement rules for parsing mathematical expressions. Unlike some parsers which may take quite a long time to construct even when all the grammar rules are available, DCG rules are already in a readily executable form. However, a DCG parser is potentially inefficient due to its frequent use of backtracking. Thus, we will propose some methods for increasing the efficiency of the parsing process. In addition, we will explain how our proposed approach works through use of an illustrative example. Finally, we will present and discuss some experimental results which are then followed by some concluding remarks.

2. Related work

One of the earliest papers on mathematical expression recognition was presented by Anderson [10] in 1968. He used a purely top-down approach for parsing mathematical expressions. The algorithm starts with one ultimate syntactic goal and tries to partition the problem (i.e. goal) into sub-goals, until either all sub-goals have been satisfied or all possibilities have been exhausted in vain. The algorithm is syntax-directed since it is guided by some grammar rules. However, experiments showed that the algorithm is not very efficient due to the partitioning strategy used for the rules which involve two non-terminal symbols on the right-hand side. As a result, up to $n - 1$ partitions can be generated by a set of n characters, and each of these partitions may further generate more partitions.

In 1970, Chang [4] proposed a method for the structural analysis of two-dimensional mathematical expressions. The algorithm mainly makes use of the ideas of operator precedence and operator dominance. It consists of two major steps, grouping operator sequences and building a structure tree. Efficiency was taken into consideration in the proposed algorithm. However, the methods described are quite tedious. It is not straight forward to understand how they actually work in practical examples.

In 1971, Martin [11] discussed some issues relating to both computer input and output of mathematical expressions. For the input case, however, not enough technical details about the replacement rules used were provided in the paper, but it raised the question of ambiguities found in mathematical expressions though with no solutions provided. In addition, it also proposed some methods to make the parsing process more efficient, but again without real implementation.

Some papers related to this topic only dealt with some specific parts of the recognition process. For example, Wang and Faure [12] applied a statistical approach for determining some relationships among symbols in mathematical expressions, such as *on the same line*, *exponent* and *subscript*. Pfeiffer [13] designed a parser for context-free languages in order to parse two-dimensional structures like mathematical expressions. However, all the discussions in that paper are limited to parsing in a theoretical sense with no real examples shown. Grbavec and Blöstein [14] used a graph rewriting approach for the understanding of mathematical expressions. Their system made use of knowledge about notational conventions to avoid the need for backtracking.

Other papers in the 1980s and 1990s investigated both the character recognition and structural analysis stages with emphasis on some specific themes. Beláid and Haton [5] worked on some simple mathematical expressions and elaborated more on solving the ambiguity problem by taking advantage of contextual information. Lee and Lee [7,15] proposed a method for recognizing symbols in mathematical expressions. Their aim was to translate the expressions from two-dimensional structures into one-dimensional character strings. Dimitriadis and Coronado [6], instead, put emphasis on the detection and correction of errors.

Chou [16] proposed to use a two-dimensional stochastic context-free grammar for the recognition of printed mathematical expressions. His approach was designed for handling noise and random variations. In the grammar, each production rule has an associated probability. The main task of the process is to find the most probable parse tree for the input expression. The overall probability of a parse tree is computed by multiplying together the probabilities for all production rules used in a successful parse. As a result, the process is computationally quite expensive.

Okamoto and Miao [17] took advantage of some specific knowledge of notational conventions of mathematics. Their method can find the structures of expressions without the need for parsing. Twaakyondo and Okamoto [18] extended the work of Okamoto and Miao [17] and Okamoto and Miyazawa [19] by using two strategies, namely, top-down and bottom-up structure processing methods. Again, with their approach, structures can be obtained without parsing. On the other hand, Lee and Wang [20] built a symbol relation tree for an expression and used some heuristics to correct recognition errors. Like the previous two, this method also does not require parsing.

Ha et al. [21] defined an expression tree as an abstraction of a mathematical expression. The construction of such an expression tree can be done through top-down (finding all the primitive objects) and bottom-up (resolving spatial relationships among objects) processes.

Recently, an approach based on hidden Markov models for character recognition was proposed [22]. The resulting mathematical expressions are recognized using a soft-decision approach [23]. Such an approach can ensure that alternative solutions are generated and explored under ambiguous cases.

3. 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 two 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 denoting only a variable name. Although it is somewhat unusual, “ $a2$ ” may be used to represent the multiplication of a by 2.
2. The same group of characters can have different meanings under different contexts. For example, “ dx ” has different meanings in “ $\int x^2 dx$ ” and in “ $cy + dx$ ”. In the first expression, “ dx ” is part of the integral. However, in the second one, the same two letters become the multiplication of two variables.

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

3.1. Grouping symbols

Before we can interpret the symbols, we must first group them properly into units. This can be done by using as heuristics some conventions in writing mathematical expressions. 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, i.e., 2 and 10.
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.

3.2. 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 into one single unit. For example, $(a + b)$ is a unit which holds the sum of a and b .
2. Some binding symbols, like fraction line, $\sqrt{\quad}$ 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 together give 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 $a + b/c$, the meaning becomes $a + (b/c)$ due to the fact that “/” has higher precedence than “+”. The operator “+” is said to dominate “/”. However, in $(a + b)/c$, the meaning becomes $(a + b)/c$ since “/” dominates “+” in this case.

4. 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 [4,10,13]. To remedy such weaknesses, we propose to use definite clause grammar (DCG) [24] as a formalism to concisely and precisely describe our set of replacement rules for parsing mathematical expressions. Note that a grammar expressed in DCG is highly declarative and can be directly executed by a Prolog interpreter.

However, DCG parsers are known to be potentially inefficient due to backtracking. In this section, we will propose some methods for increasing the efficiency of the parsing process.

4.1. Basic notations for DCG

DCG is similar to BNF, with some minor notational differences summarized 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 are some major differences 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. In

addition, arguments can be added to non-terminal symbols of the grammars.

4.2. Conventional backtracking parsing in DCG

The simplest way of parsing a two-dimensional expression is to first 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.


Now, suppose that the parser we are going to use is a DCG parser and we need to create it from scratch. How many rules do we need?

In general, the simplest expressions are the ones that involve arithmetic operations. As we know, all the binary arithmetic operators are left-associative. However, top-down parsers, such as a DCG parser, cannot handle left-recursive grammars. This problem can be solved easily by transforming those left-recursive grammars to right-recursive ones. However, although the strings generated by any left-recursive grammar and its corresponding right-recursive grammar can be the same, their internal structures may be different. Hence, some fixing efforts may be required subsequently.

Anyhow, the grammar rules for arithmetic operations are extremely simple. They are as follows:

```
parse-equation(A) --> equation(A).
equation( [=, A, B] --> expr(A), [=], expr(B).
expr([Op, A, B] --> term(A), [Op],
    {is_add_sub(Op)}, expr(B).
expr(A) --> term(A).
term([Op, A, B] --> factor(A), [Op],
    {is_mul_div(Op)}, term(B).
term(A) --> factor(A).
```

Note that multiplication and division have higher precedence than addition and subtraction. Such precedence relationships can be implemented easily by having mul-

$$\frac{6x + 4y}{2} = 3x + 2y$$


$$(6x + 4y) / 2 = 3x + 2y$$

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

multiple levels in the grammar rules. In general, the operators at a level always have higher precedence than the ones above them.

Similar techniques can also be applied to the unary operator, as well as spatial operators like implicit multiplication, subscript, exponent and parentheses. Here are the grammar rules:

```
factor([neg, A] --> [-], sub_expr(A).
factor(A) --> sub_expr(A).
sub_expr(MENAME) --> sub_term(MENAME).
sub_term(MENAME) --> sub_factor(A),
    sub_expr(B), {is_adjacent(A, B, MENAME)}.
sub_term(A) --> sub_factor(A).
sub_factor(MENAME) --> expr_unit(A),
    sub_expr(B), {is_sub_exp(A, B, MENAME)}.
sub_factor(A) --> expr_unit(A).
expr_unit(MENAME) --> ['(', expr(S1), [')'],
    {add_expr_unit(S1, MENAME)}.
expr_unit(A) --> [A], {is_expr(A)}.
```

In order to handle functions, indefinite integral, fraction and square root, the following rules are needed:

```
sub_expr(MENAME) --> function(MENAME).
sub_expr(MENAME) --> integral(MENAME).
function(MENAME) -->
    is_function_name(Function),
    arg_term(A), function(B),
    {add_function(Function, A, B, MENAME)}.
function(MENAME) -->
    is_function_name(Function), arg_term(A),
    {add_function(Function, A, MENAME)}.
is_function_name(Function) --> [F],
    {is_function_name(F, Function)}.
arg_term(MENAME) --> arg_factor(A),
    arg_term(B), {is_adjacent(A, B, MENAME)}.
arg_term(A) --> arg_factor(A).
arg_factor(MENAME) --> expr_unit(A),
    arg_factor(B), {arg_sub_exp(A, B, MENAME)}.
arg_factor(A) --> expr_unit(A).
integral(MENAME) --> [integral],
    sub_expr(A), is_d, expr_unit(B),
    {store_expr([integral, A, B], MENAME)}.
is_d --> [D], {is_d(D)}.
expr_unit(MENAME) --> [frac, '{',
    expr(A), ['}', '{'], expr(B), ['}'],
    {store_expr([frac, A, B], MENAME)}.
expr_unit(MENAME) --> [root, '{',
    expr(A), ['}', '{'], expr(B), ['}'],
    {store_expr([root, A, B], MENAME)}.
```

Notice that it usually takes comparatively longer time for a DCG parser with the above grammar rules to return the tree structure of an expression, because some sub-structures may be re-generated again and again during the backtracking steps. Therefore, the bigger the

structure is, the longer the time it takes. Fig. 2 depicts the tree structure for the expression shown before in Fig. 1.

4.3. Parsing with left-factored rules

Although the grammar rules in the previous section are highly comprehensible, they are not very efficient from the implementation point of view. For example, in the following two grammar rules,

```
expr ([Op, A, B]) --> term(A), [Op],
    {is_add_sub(Op)}, expr(B).
expr(A) --> term(A),
```

we must first find term(A). If the next symbol Op is neither an addition operator nor a subtraction operator, we then backtrack to the second rule. However, in the second rule, the same step of finding term(A) is repeated again. To tackle this problem, we can perform left factoring on the same rule to give the following result:

```
expr(B) --> term(A), more_term(A, B).
more_term(A, [Op, A, B]) --> [Op],
    {is_add_sub(Op)}, expr(B).
more_term(A, A) --> [ ].
```

In the above left-factored grammar rule, the result of term(A) is passed into the next sub-goal more_term(A, B). If the respective operator is found, we then continue to process more terms. Otherwise, the input structure is returned as output.

The main idea of left factoring is to rewrite some grammar rules so that decisions can be deferred until enough input tokens have been seen in order to make the right choice [25]. The following is the set of grammar rules corresponding to the rule set in the previous section, with some of the rules replaced by left-factored ones as shown below:

```
term(B) --> factor(A), more_factor(A, B).
more_factor(A, [Op, A, B]) --> [Op],
    {is_mul_div(Op)}, term(B).
more_factor(A, A) --> [ ].
function(MEName) -->
    is_function_name(Function), arg_term(A),
    more_function(Function, A, MEName).
```

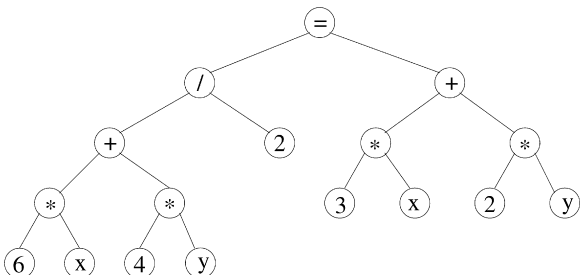


Fig. 2. Tree structure of the mathematical expression in Fig. 1.

```
more_function(Function, A, MEName) -->
    function(B),
    {add_function(Function, A, B, MEName)}.
more_function(Function, A, MEName) --> [ ],
    {add_function(Function, A, MEName)}.

sub_term(MEName) --> sub_factor(A),
    more_sub_term(A, MEName).

more_sub_term(A, MEName) --> sub_expr(B),
    {is_adjacent(A, B, MEName)}.
more_sub_term(A, A) --> [ ].

sub_factor(MEName) --> expr_unit(A),
    more_sub_factor(A, MEName).

more_sub_factor(A, MEName) -->
    sub_expr(B), {is_sub_exp(A, B, MEName)}.
more_sub_factor(A, A) --> [ ].

arg_term(MEName) --> arg_factor(A),
    more_arg_term(A, MEName).

more_arg_term(A, MEName) -->
    arg_term(B), {is_adjacent(A, B, MEName)}.
more_arg_term(A, A) --> [ ].

arg_factor(MEName) --> expr_unit(A),
    more_arg_factor(A, MEName).

more_arg_factor(A, MEName) -->
    arg_factor(B), {arg_sub_exp(A, B, MEName)}.
more_arg_factor(A, A) --> [ ].
```

4.4. Parsing with binding symbol preprocessing

As mentioned in Section 2, binding symbols always dominate their neighbors. For example, in the expression shown in Fig. 1, the fraction line in $(6x + 4y)/2$ 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 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.

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.

The grammar rules corresponding to binding symbol preprocessing are as follows:

```
parse_expr(A) --> preprocess([ ], B),
    {expr(A, B, [ ])}.

```

4.5. 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 AI planning [26].

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. Some symbols in an expression, for example, \int and dx in an indefinite integral expression, enclose a sub-expression in between.

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.

Here is the list of relevant DCG rules for parsing with hierarchical decomposition:

```

parse_equation(A) --> preprocess([], B),
{equation(A, B, [])}.
parse_sub_expr(A) --> preprocess([], B),
{sub_expr(A, B, [])}.

preprocess(A, C) --> [integral],
find_integral([], E1),
preprocess([E1 | A], C).
preprocess(A, C) --> [Begin],
{fence_symbols(Type, Begin, End)},
find_expr(Type, End, [], E2),
preprocess([E2 | A], C).
preprocess(A, C) --> [B], preprocess([B | A], C).
preprocess(A, B) --> [], {reverse(A, B)}.
    
```

```

find_integral(A, MENAME) --> [D], {is_d(D)},
expr_unit(B), {add_integral(A, B, MENAME)}.
find_integral(A, C) --> [B],
find_integral([B | A], C).

find_expr(Type, End, A, MENAME) --> [End],
{add_expr(Type, A, MENAME)}.
find_expr(Type, End, A, C) --> [B],
find_expr(Type, End, [B | A], C).
    
```

As mentioned before, although the strings generated can be the same, the internal structures may be different if we rewrite some left-associative grammar rules into right-associative ones. Hence, we need a procedure for fixing the resulting structure to reflect the correct associativity between operators and their operands. The following is such procedure written in Prolog, which is self-explanatory:

```

transform([Op1, A1, [Op2, B1, C1]], Struct):-
left_assoc_ops_with_same_preced(Op1, Op2),
transform(A1, A2), transform(B1, B2),
(atomic(C1),
Struct = [Op2, [Op1, A2, B2], C1]
; transform([Op2, [Op1, A2, B2], C1], Struct)
).
transform([Op, A1], [Op, A2]):-
transform(A1, A2).
transform([Op, A1, B1], [Op, A2, B2]):-
transform(A1, A2), transform(B1, B2).
transform([Op, A1, B1, C1], [Op, A2, B2, C2]):-
transform(A1, A2), transform(B1, B2),
transform(C1, C2).
transform(A, A).
    
```

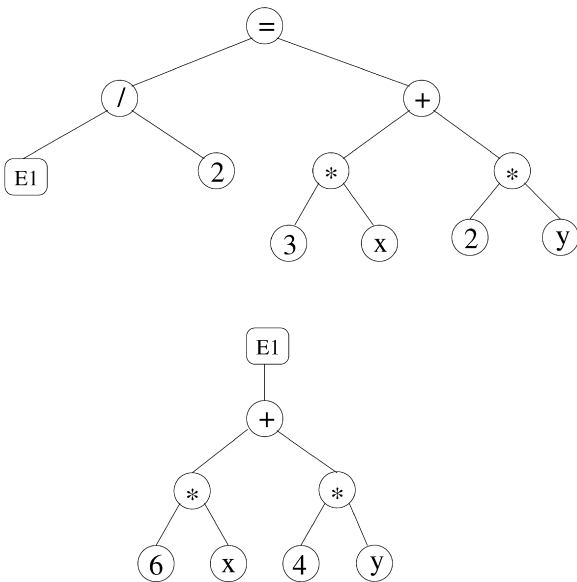


Fig. 3. Tree structures generated as a result of parsing with binding symbol preprocessing.

5. Experimental results and discussions

In this experiment, we perform tests on a number of different expressions which were extracted from Ref. [27]. Expressions are grouped into four domains, namely, elementary algebra, trigonometric functions, geometry and indefinite integrals. In each domain, there are three sizes of expressions, i.e. small, medium and large. Each size consists of five different expressions. Totally, there are 60 expressions.

Initially, the input is simply a sequence of points. After some segmentation steps, we then use the character recognition method proposed in Ref. [28]. Due to the high accuracy achieved by the method and the fact that those 0 expressions are neatly written, all the characters and symbols in the expressions can be recognized without errors. The recognized characters and symbols are then converted to objects with associated attributes, including location, size, and identity. Note that the objects can be put in an arbitrary order for our subsequent processing.

The next step is to group the objects. Here we use a method similar to the one used in Ref. [19]. Afterwards, we perform parsing using different techniques as described above and then compare their efficiency.

Note that time may not be a very good measure of efficiency since it may differ from machines to machines. Hence, instead we use the number of logical inferences as a machine-independent performance measure. Table 1 shows the differences between conventional backtracking parsers with and without the use of left factoring.

The result shows that the set of grammar rules we used plays an important role in terms of efficiency. Parsing with rules which are not left-factored gives us an exponential running time with respect to the size of the expressions. However, with the left-factored version, the time taken is greatly reduced since all the intermediate results are fully utilized and there is much less repetitive construction of intermediate structures.

For binding symbol preprocessing, saving is possible only when such symbols appear in the expressions.

Table 2 shows the differences between hierarchical decomposition parsing that uses left factoring only and that uses binding and fence symbol preprocessing as well.

Our results show that speedup can be achieved for those expressions that contain some binding and fence symbols.

In order to show the potential for practical use with hierarchical decomposition parsing, we also tabulate the time taken for parsing different sizes of expressions in different domains. Our recognition system implemented in Prolog runs on a Sun SPARC 10 workstation. The timer starts when the list of objects is passed to the parsing procedure and ends when the final structure is returned. Table 3 summarizes the result.

Notice that the time required for recognizing the structures of some mathematical expressions of typical sizes

Table 1
Differences between conventional backtracking parsers with and without the use of left factoring

Expressions	Number of logical inferences required for conventional backtracking parsing	
	Without left factoring	With left factoring
Elementary algebra		
$y = x + b/4a$	89 014	1435
$a^2 - b^2 = (a - b)(a + b)$	258 420	2979
$r = \frac{16ab^2c + 256a^3e - 3b^4 - 64a^2bd}{256a^4}$	19 161 397	14 158
Trigonometric functions		
$\cos \alpha = \frac{1}{\sec \alpha}$	56 307	1422
$\tan^2 \alpha = \frac{1 - \cos 2\alpha}{1 + \cos 2\alpha}$	215 795	3459
$\tan \frac{(B - C)}{2} = \frac{b - c}{b + c} \tan \frac{(B + C)}{2}$	5 356 302	5390
Geometry		
$r = \sqrt{x^2 + y^2}$	164 576	1646
$(x - a)y^2 = -x^2(x + a)$	411 192	3745
$p = \sqrt{\frac{(ac + bd)(ab + cd)}{(ad + bc)}}$	374 511 627	7144
Indefinite integrals		
$\int e^x dx = e^x$	108 026	2682
$\int \frac{u}{v} dx = \frac{bx}{d} + \frac{k}{d^2} \log v$	183 078	4280
$\int \frac{\sqrt{a^2 - x^2}}{x^4} dx = -\frac{\sqrt{(a^2 - x^2)^3}}{3a^2x^3}$	52 307 610	10 314

Table 2

Differences between hierarchical decomposition parsing with and without the use of binding and fence symbol preprocessing

Expressions	Number of logical inferences required for hierarchical decomposition parsing	
	With left factoring only	With left factoring, binding and fence symbol preprocessing
$y^2 \pm \sqrt{u-p} \left(y - \frac{f}{2(u-p)} \right) + \frac{u}{2} = 0$	6242	3772
$\sin A = \frac{2}{bc} \sqrt{s(s-a)(s-b)(s-c)}$	30 271	5173
$\frac{2ab}{a+b} \cos \frac{1}{2} C = \sqrt{ab \left(1 - \frac{c^2}{(a+b)^2} \right)}$	11 905	6317
$\int x e^{-x^2} dx = -\frac{1}{2} e^{-x^2}$	16 000	4307
$\int x \sqrt{(a^2 - x^2)^3} dx = -\frac{1}{5} \sqrt{(a^2 - x^2)^5}$	10 084	4919

Table 3

Time required for recognizing the structures of different expressions with hierarchical decomposition parsing

Expression domain	Time required for hierarchical decomposition parsing (in seconds)								
	Small size			Median size			Large size		
	Min.	Median	Max.	Min.	Median	Max.	Min.	Median	Max.
Elementary algebra	0.02	0.03	0.05	0.05	0.07	0.08	0.08	0.15	0.25
Trigonometric functions	0.02	0.02	0.05	0.05	0.07	0.07	0.08	0.10	0.15
Geometry	0.02	0.03	0.05	0.05	0.08	0.10	0.10	0.12	0.15
Indefinite integrals	0.02	0.05	0.05	0.07	0.08	0.08	0.10	0.15	0.17

ranges from 0.02 to 0.25 s. Nevertheless, the parser used is relatively simple. In fact, the whole parser has been listed in the previous section.

6. Conclusion

Pen-based computing offers us a natural human-computer interface, such as an on-line mathematical expression 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. In addition, the replacement rules are already in their executable form so that no exact programming is needed for implementing the rules.

Since our methods do not make use of stroke order information, they may also be used for off-line mathematical expression recognition. However, some problems in mathematical expression recognition have not been addressed in this paper, including ambiguity resolution, error detection, and error correction. With a clear and concise formalism in the parsing phase, these issues will be relatively easy to tackle, using, for example, some error-correcting parsing techniques. Detail investigation of these issues will be provided in a separate paper.

7. Summary

In a mathematical expression, characters and symbols are typically arranged as a complex two-dimensional structure, possibly of different character and symbol sizes. This makes the recognition process more complicated even when all the individual characters and symbols can be recognized correctly. Moreover, to ensure that

a mathematical expression recognition system is useful in practice, its recognition speed is also an important factor to consider.

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, a DCG parser is potentially inefficient due to its frequent use of backtracking. Thus we propose some methods here to increase the efficiency of the parsing process.

Some experiments are done on 60 commonly seen mathematical expressions that are in four domains, namely, elementary algebra, trigonometric functions, geometry and indefinite integrals. The results show that the set of grammar rules we used plays an important role in terms of efficiency. Parsing with rules which are not left-factored gives us an exponential running time with respect to the size of the expressions. However, with the left-factored version, the time taken is greatly reduced since all the intermediate results are fully utilized and there is much less repetitive construction of intermediate structures. In addition, we also show that our proposed methods can achieve quite satisfactory speedup, making mathematical expression recognition more feasible for real-world applications.

Since our methods do not make use of stroke order information, they may also be used for off-line mathematical expression recognition. However, some problems in mathematical expression recognition have not been addressed in this paper, including ambiguity resolution, error detection, and error correction. With a clear and concise formalism in the parsing phase, these issues will be relatively easy to tackle and will be addressed in our future research.

Acknowledgements

This research work is supported in part by the Hong Kong Research Grants Council (RGC) under Competitive Earmarked Research Grants HKUST 746/96E and HKUST 6081/97E awarded to the second author.

References

- [1] F. Grossman, R.J. Klerer, M. Klerer, A language for high-level programming of mathematical applications, in Proceedings of the International Conference on Computer Languages, Miami Beach, FL, 1988, pp. 31–40.
- [2] L. Lamport, *Latex – A Document Preparation System – User's Guide and Reference Manual*, Addison-Wesley, Reading, MA, 1985.
- [3] C.C. Tappert, C.Y. Suen, T. Wakahara, The state of the art in on-line handwriting recognition, *IEEE Trans. Pattern Anal. Mach. Intell.* 12 (8) (1990) 787–808.
- [4] S.K. Chang, A method for the structural analysis of 2-D mathematical expressions, *Information Sciences* 2 (3) (1970) 253–272.
- [5] A. Beláid, J.-P. Haton, A syntactic approach for handwritten mathematical formula recognition, *IEEE Trans. Pattern Anal. Mach. Intell.* 6 (1) (1984) 105–111.
- [6] Y.A. Dimitriadis, J.L. Coronado, Towards an ART based mathematical editor, that uses on-line handwritten symbol recognition, *Pattern Recognition* 28 (6) (1995) 807–822.
- [7] H.-J. Lee, M.-C. Lee, Understanding mathematical expressions using procedure-oriented transformation, *Pattern Recognition* 27 (3) (1994) 447–457.
- [8] J. Earley, An efficient context-free parsing algorithm, *Comm. ACM* 13 (1970) 94–102.
- [9] M.A. Covington, *Natural Language Processing for Prolog Programmers*, Prentice-Hall, Englewood Cliffs, NJ, 1994.
- [10] R.H. Anderson, Syntax-directed recognition of hand-printed 2-D mathematics, in: M. Klerer, J. Reinfelds (Eds.), *Interactive Systems for Experimental Applied Mathematics*, Academic Press, New York, 1968, pp. 436–459.
- [11] W.A. Martin, Computer input/output of mathematical expressions, in *Proceedings of the Second Symposium on Symbolic Algebraic Manipulation*, Los Angeles, CA, 1971, pp. 78–89.
- [12] Z.X. Wang, C. Faure, Structural analysis of handwritten mathematical expressions, in *Proceedings of the 9th International Conference on Pattern Recognition*, Rome, Italy, 1988, pp. 32–34.
- [13] J.J. Pfeiffer, Jr., Parsing graphs representing two dimensional figures, in *Proceedings of the IEEE Workshop on Visual Languages*, Seattle, WA, pp. 200–206 (1992).
- [14] A. Grbavec, D. Blostein, Mathematics recognition using graph rewriting, in *Proceedings of the Third International Conference on Document Analysis and Recognition*, Montreal, Canada, 1995, pp. 417–421.
- [15] H.-J. Lee, M.-C. Lee, Understanding mathematical expressions in a printed document, in *Proceedings of the Second International Conference on Document Analysis and Recognition*, Tsukuba Science City, Japan, 1993, pp. 502–505.
- [16] P.A. Chou, Recognition of equations using a two-dimensional stochastic context-free grammar, in *Proceedings of the SPIE Visual Communications and Image Processing IV*, Philadelphia, PA, vol. 1199, 1989, pp. 852–863.
- [17] M. Okamoto, B. Miao, Recognition of mathematical expressions by using the layout structures of symbols, in *Proceedings of the First International Conference on Document Analysis and Recognition*, Saint-Malo, France, 1991, pp. 242–250.
- [18] H.M. Twaakyondo, M. Okamoto, Structure analysis and recognition of mathematical expressions, in *Proceedings of the Third International Conference on Document Analysis and Recognition*, Montreal, Canada, 1995, pp. 430–437.
- [19] M. Okamoto, A. Miyazawa, An experimental implementation of a document recognition system for papers containing mathematical expressions, in: H.S. Baird, H. Bunke, K. Yamamoto (Eds.), *Structured Document Image Analysis*, Springer, Berlin, 1992, pp. 36–53.

- [20] H.-J. Lee, J.-S. Wang, Design of a mathematical expression recognition system, in *Proceedings of the Third International Conference on Document Analysis and Recognition*, Montreal, Canada, 1995, pp. 1084–1087.
- [21] J. Ha, R.M. Haralick, I.T. Phillips, Understanding mathematical expressions from document images, in *Proceedings of the Third International Conference on Document Analysis and Recognition*, Montreal, Canada, 1995, pp. 956–959.
- [22] M. Koschinski, H.-J. Winkler, M. Lang, Segmentation and recognition of symbols within handwritten mathematical expressions, in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Detroit, MI, Vol. 4, 1995, pp. 2439–2442.
- [23] H.-J. Winkler, H. Fahrner, M. Lang, A soft-decision approach for structural analysis of handwritten mathematical expressions, in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Detroit, MI, Vol. 4, 1995, 2459–2462.
- [24] F. Pereira, D. Warren, Definite clause grammars for language analysis – a survey of the formalism and comparison with augmented transition networks, *Artif. Intell.* 13 (1980) 231–278.
- [25] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.
- [26] S.J. Russell, P. Norvig (Eds.), *Artificial Intelligence: A Modern Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- [27] D. Zwillingner (Ed.), *CRC Standard Mathematical Tables and Formulae*, 30th ed., CRC Press, Boca Raton, 1996.
- [28] K.F. Chan, D.Y. Yeung, Elastic structural matching for on-line handwritten alphanumeric character recognition, in *Proceedings of the 14th International Conference on Pattern Recognition*, Brisbane, Australia, 1998, pp. 1508–1511.

About the Author—KAM-FAI CHAN received his B.Sc. degree from Radford University, M.Sc. degree from the University of South Carolina, and Ph.D degree from the Hong Kong University of Science and Technology, all in computer science. He is currently a postdoctoral research associate in the Department of Computer Science at the Hong Kong University of Science and Technology. His major research interests include pattern recognition, logic programming and Chinese computing.

About the Author—DIT-YAN YEUNG received his B.Sc.(Eng.) degree in electrical engineering and M.Phil. degree in computer science from the University of Hong Kong, and his Ph.D. degree in computer science from the University of Southern California in Los Angeles. From 1989 to 1990, he was an assistant professor at the Illinois Institute of Technology in Chicago. He is currently an associate professor in the Department of Computer Science at the Hong Kong University of Science and Technology. His current research interests are in the theory and applications of pattern recognition, machine learning, and neural networks. He frequently serves as a paper reviewer for a number of international journals and conferences, including *Pattern Recognition*, *Pattern Recognition Letters*, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *IEEE Transactions on Image Processing*, and *IEEE Transactions on Neural Networks*.