Graphical User Interface for Search of Mathematical Expressions with Regular Expressions

Takayuki Watabe[1(✉)] and Yoshinori Miyazaki[2]

[1] Graduate School of Science and Technology, Shizuoka University,
Shizuoka, Japan
dgs13012@s.inf.shizuoka.ac.jp

[2] Graduate School of Informatics, Shizuoka University, Shizuoka, Japan
yoshi@inf.shizuoka.ac.jp

Abstract. This paper discusses a pattern-matching method with regular expressions for mathematical expressions on electronic documents. In ordinary regular expressions, a pattern is described as a string with meta-characters. However, strings are unsuitable for mathematical expressions because of their two-dimensional structure (e.g., fractions, superscripts, and subscripts). In addition, meta-characters for regular expressions are frequently used as normal characters, forcing users to type escape characters. Therefore, in this study, we propose a graphical user interface (GUI) to create patterns for mathematical expressions.

Keywords: Mathematical expressions · Pattern-matching · Regular expressions · GUI

1 Introduction

A mathematical expression used for describing mathematical concepts and models is a significant notation. Electronic documents, including web pages and e-books, often contain mathematical expressions and need to retrieve content. Search algorithms for natural language are partially adaptable to mathematical expressions because a mathematical expression consists of natural language characters. However, the layout of the characters in a mathematical expression is different from that in natural language; this suggests a need to develop search algorithms specifically for mathematical expressions.

Some studies on searching mathematical expressions have been conducted [1–3] typically in relation to search engines. Moreover, search algorithms can be used to highlight search results in a document or move to a page that comprises the results. In general, pattern-matching algorithms (string searching algorithms) are suitable for such applications. Previously, we proposed a search algorithm for mathematical expressions similar to a pattern-matching algorithm for natural language [4]. Using this algorithm, our search tool (MathRegExp) allows users to use regular expressions to describe patterns. Regular expressions can make patterns flexible using wildcards, Boolean *or*, the number of occurrences of character(s), etc. Although several studies have attempted

M. Kurosu (Ed.): Human-Computer Interaction, Part I, HCII 2015, LNCS 9169, pp. 438–447, 2015.
DOI: 10.1007/978-3-319-20901-2_41

to implement wildcards for searching mathematical expressions [5, 6], their functions and notations are different from those used in regular expressions. Our regular expressions for mathematical expressions are similar to the original regular expressions for plain texts.

In MathRegExp, a pattern is expressed as a string. However, using strings as the pattern has some problems. First, the two-dimensional layout of characters in mathematical expressions is represented one-dimensionally, preventing intuitive understanding of which expressions are to be matched to a pattern. Second, because meta-characters for regular expressions are frequently used as normal characters (e.g., "+", "(", and ")"), users must use escape characters, which are cumbersome and prone to mistakes. Because of these problems, novice users, including mathematics learners and liberal arts scholars, might find MathRegExp difficult to use. Therefore, in this study, we propose a graphical user interface (GUI) to create patterns for MathRegExp.

A GUI for describing mathematical expressions has been proposed previously [7]. Our GUI satisfies the further demand of the ability to input regular expressions for advanced and complex matching as well. Our GUI displays these as figures to solve the aforementioned problem of escaping meta-characters and provides users with two methods for inputting regular expressions.

2 Patterns

This section outlines the patterns of MathRegExp as strings to introduce its functions. Here, the proposed GUI internally generates patterns as strings. A pattern consists of characters, structures, and regular expressions, which will be discussed below.

2.1 Characters and Structures

Characters are represented in Unicode. Because Unicode contains categories for mathematical expressions, users can describe symbols appearing in various fields, including calculus, geometry, logic, and set theory.

We refer to the particular symbol layouts appearing in a mathematical expression as structures. A notation for a structure is the form \keyword{argument1}[{argument2}[{argument3}]] (where "["and "]" indicate optionality). A keyword specifies a type of structure (e.g., fraction, superscript, or square root), and structures are allowed to be nested. For example, `\frac{1}{\sqrt{2}}` is acceptable for $\frac{1}{\sqrt{2}}$. Table 1 shows a list of structures.

The structures in Table 1 and characters in Unicode enable users to describe summation, definite integration, and maximum.

2.2 Regular Expressions

Regular expressions are divided into five categories, namely, wildcard, character class, quantification, Boolean *or*, and backreference.

Table 1. List of structures

Keyword	Description	Argument			Example
		1	2	3	
sub	subscript	subscript			$_1$
sup	superscript	superscript			^{2}
subsup	subscript and superscript	subscript	superscript		$_0^\infty$
under	underscript	base	underscript		$\lim\limits_{n\to\infty}$
over	overscript	base	overscript		$\vec{a}$
underover	underscript and overscript	base	underscript	overscript	$\sum\limits_{i=0}^{n}$
sqrt	square root	base			$\sqrt{x}$
root	radical with index	base	index		$\sqrt[3]{a}$
frac	fraction	numerator	denominator		$\frac{1}{2}$

A wildcard is represented as “`.`” and matches an arbitrary character or arbitrary structure. For example, `\frac{.}{2}` matches $\frac{\sqrt{x}}{2}$.

A character class matches a character enclosed by “`[`“ and ”`]`”. Therefore, `[xyz]\sup{2}` matches both x^2 and y^2. A negated character class matches a single character that is not contained between “`[^`” and “`]`”. An abbreviated notation using “`-`” in character classes is acceptable: `[0-9]` matches an arbitrary single digit, and `[^stx-z]` matches any character other than “s”, “t”, “x”, “y” and “z”. Structures are unacceptable in character classes.

Quantification is described as “`*`”, “+” and “`?`”. “`*`” represents “zero or more of a preceding element,” “+” matches “one or more of a preceding element,” and “?” matches “zero or one of the preceding element.” As an example, `\sqrt{.+}` matches $\sqrt{2a^2}$.

Boolean *or* is represented as “`|`”. “`|`” separates alternatives, i.e., `x|y` matches both x and y individually.

A scope of quantification and Boolean *or* is specified by enclosing with “`(`” and “`)`”. For example, `(x\sub{.})+` matches x_1, x_2, x_3, etc., and `(\sqrt{.}|.\sup{\frac{1}{2}})` matches $\sqrt{x}$ or $x^{\frac{1}{2}}$. In addition, a structure is assumed to be a character, hence the pattern `\sqrt{.}+` is valid.

A backreference, notated as “\n” (*n* is a number), matches a mathematical expression identical with the expression matched with a part of a pattern enclosed by the *n*th parenthesis (i.e., “`(`” and “`)`”). For example, `\frac{.+}{(.+)}-\frac{.+}{\1}` matches the addition of two fractions with common denominators.

3 GUI

The method of creating patterns is similar to that of inputting ordinary text, namely, repeatedly inserting a character after a cursor. However, it might be necessary to create patterns with untypable characters, structures, or regular expressions.

3.1 GUI for Mathematical Expressions

Characters are input from the keyboard if they are typable. A palette is provided for inputting untypable characters, including Greek letters and mathematical symbols. Structure templates (i.e., empty structures) are displayed when the buttons are clicked. A double-lined rectangle indicates a structure argument, such as those in Fig. 1.

Users can position the cursor at the rectangle and input a pattern as an argument of the structure. The cursor is controlled by clicking or pressing arrow keys. When arrow keys are pressed, the cursor moves in the order of argument1, argument2, and argument3.

A structure is deleted by pressing the delete key after selecting the entire structure using rectangle selection or placing a cursor on one of the empty arguments (double-lined rectangles).

3.2 GUI for Regular Expressions

This chapter presents figures for regular expressions and methods of inputting and editing them.

3.2.1 Figures of Regular Expressions

Regular expressions are displayed on our GUI as figures rather than meta-characters. Table 2 shows the figures.

These figures represent the scope with a rectangle, allowing users to intuitively grasp the scope of mathematical expressions having two-dimensional structures. This representation also solves the problem that parentheses in regular expressions represented as strings have the double meaning of specifying scope and capturing backreferences. Here, the number of capturing (i.e., *n* of capturing in Table 2) is assigned as the order of inputting.

Quantification, Boolean *or*, and capturing are applied to a part of a mathematical expression. When multiple functions are applied to an identical part, merged figures are displayed. In a merged figure, "possibility of absence" (optionality) is represented as a broken-lined rectangle, "repetition" is three vertical lines following a rectangle,

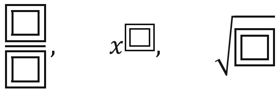
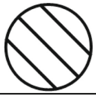
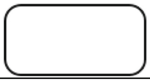
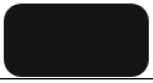
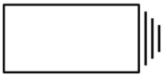
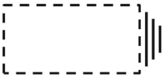
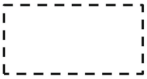
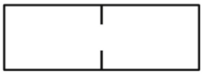
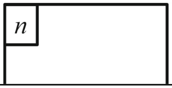
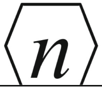


Fig. 1. Templates of structures

Table 2. Figures of Regular Expressions

Description	Figure
Wildcard	
Character class	
Negated character class	
Quantification (+)	
Quantification (*)	
Quantification (?)	
Boolean *or*	
Capturing for backreference	
Backreference	

"Boolean *or*" is a spaced vertical line, and "capturing" is an appended number with a small rectangle. Figure 2 is an example of a merged figure representing `(x|y|z)*`.

3.2.2 Inputting Regular Expressions

A wildcard is inputted by clicking a button, similar to the inputting of untypable characters. The method for inputting a character class, a negated character class, or backreference is identical to that for inputting structures, namely, displaying a rounded rectangle (or a hexagon) and describing contents for character classes (or a reference number).

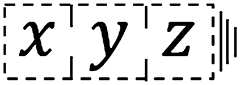


Fig. 2. Merged figure of `(x|y|z)*`

Our GUI provides two input methods for the functions with a scope (i.e., quantification, Boolean *or*, and capturing). The first method is identical to that for inputting structures, displaying a rectangle and describing a part of the pattern in the scope. The second is selecting a part of an already created pattern to specify the scope and assign the function.

We suppose that there are different procedures for creating patterns. Some users might decide to use regular expressions before creating a pattern, for example, a user inputs `\frac{.+}{()}\+\frac{.+}{\1}` as “addition of fractions with a common denominator” and then concretely describes the denominator. Others might add regular expressions to the pattern after describing a mathematical expression, for example, a user searches with a pattern such as `sinx` and then wishes to search trigonometric functions, searching again with `(sin|cos|tan)x`. Our GUI allows users to follow both the procedures by providing multiple input methods. The former users will employ the method for structures and the latter will use selecting and assigning method.

3.2.3 Editing Regular Expressions

Users edit wildcards such as characters, character classes, and capturing as structures.

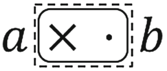
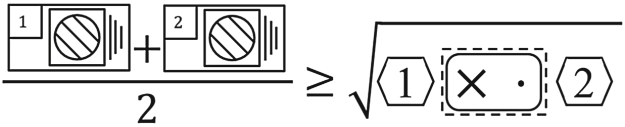
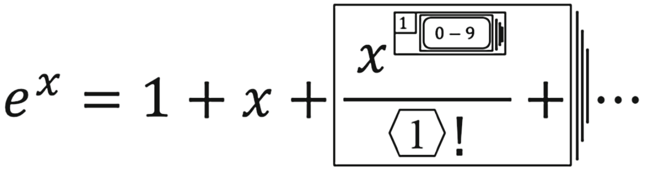
When users edit regular expressions having scope, they must first select the scope. A scope is selected by clicking a rectangle or pressing right (left) cursor keys with the cursor just before (after) the rectangle. When a scope is selected, the cursor disappears and the selected rectangle blinks. The rectangle is blurred by clicking characters or structures in the pattern or pressing cursor keys, and the cursor appears at the appropriate position.

Types of editing regular expressions with a scope are divided into “assigning the function,” “updating the scope,” and “deleting the scope.” Users assign a function using buttons. The buttons for quantification and backreferences behave as toggles. If a function is assigned to a scope, the button remains pressed. The button for Boolean *or* is used for adding alternatives. When the button is clicked, a spaced line is inserted in the rectangle, enabling the user to input a new alternative. Users can update a scope by dragging and resizing the selected rectangle. It is impossible for the scope of Boolean *or* to be updated because the function has multiple pattern parts. A scope is deleted by pressing the delete key with the selected scope. If a scope with Boolean *or* is deleted, the second to the last alternatives are also deleted.

3.3 Examples of Patterns

In Table 3, we show examples of patterns created using our GUI, the corresponding patterns as strings, and the matched mathematical expressions.

Table 3. Patterns and Matched Mathematical Expressions

Multiplication of a and b	
`a[×·]?b`	
$a \times b, a \cdot b, ab$	
Relation between arithmetic and geometric means	
`\frac{(.+)\+(.+)}{2}≥\sqrt{\1[×·]?\2}`	
$\frac{a+b}{2} \geq \sqrt{ab}, \frac{x_1+x_2}{2} \geq \sqrt{x_1 x_2}$	
Maclaurin series of e^x	
`e\sup{x}=1\+x\+(\frac{x\sup{([0-9]+)}}{\2!}\+)+⋯`	
$e^x = 1 + x + \frac{x^2}{2!} + \cdots, e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots$ Note: as a result of performing to specification, this pattern could also match math ematical expressions including $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^3}{3!} + \cdots$.	

4 Implementation of Matching

This section explains how our matching algorithm is implemented. Highlighting matched parts of mathematical expressions in web pages, one application of our algorithm, including replacing, is detailed. Figure 3 is the conceptual diagram of the implementation.

We use the regular expressions library called Onigmo [8] internally. The library can describe recursive patterns in addition to ordinary functions of regular expressions. Recursion is indispensable to our manner of matching mathematical expressions because we use parentheses to represent structure (i.e., "`{`", and "`}`" in `\keyword{content}`). Correspondence of parentheses cannot be treated by ordinal regular expressions. For example, a pattern of `\sqrt{. +}`, which means "a square root including an arbitrary mathematical expression," matches `\sqrt{x}\frac{1}{2}` because `. +` in the pattern matches "`x}\frac{1}{2`." If the `. +` is revised to `[^{}] +`, which means "repetition of a character unless `{`or `}`," to avoid this problem, then the pattern causes another problem that the pattern does not match the mathematical expression `\sqrt{x\sup{2}}`. A recursive pattern solves the problem. The details will be discussed later.

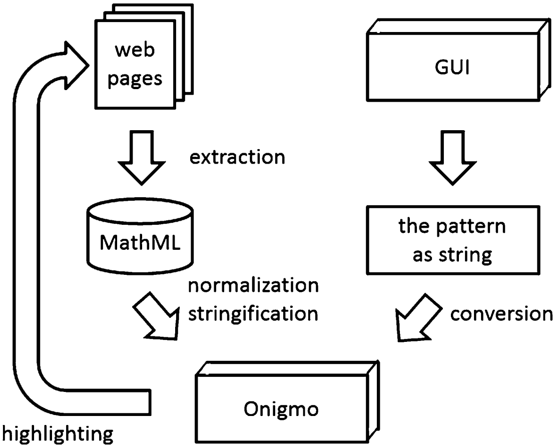


Fig. 3. Conceptual diagram

It is assumed that target mathematical expressions of our matching method are described in a format called MathML [9] Presentation Markup. This format is standard for describing mathematical expressions in electronic papers because it is recommended by W3C and can be used in HTML5. In MathML, a single mathematical expression can be described as various data. For example, two MathML data in Fig. 4 represent the identical mathematical expression: ab^2.

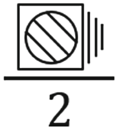
On the left-hand side in Fig. 4, a and b^2 are located on the same line. On the right-hand side, the superscript “2” is attached to ab. Such problems are found in several tags of MathML, which reduces the precision of the pattern-matching process. Therefore, we normalize MathML data to eliminate data variety before pattern-matching. Moreover, normalized data are stringified to a similar format for the patterns described in Sect. 2.

Patterns are created using the GUI, and the GUI generates patterns as strings following the aforementioned format. The patterns, however, are not interpreted appropriately as patterns with Onigmo, hence, they are converted before matching. An example of conversion is shown in Table 4.

```
<math>                      <math>
    <mi>a</mi>                  <msup>
    <msup>                          <mrow>
        <mi>b</mi>                      <mi>a</mi>
        <mn>2</mn>                      <mi>b</mi>
    </msup>                         </mrow>
</math>                             <mn>2</mn>
                                </msup>
                            </math>
```

Fig. 4. Two kinds of data for ab^2

Table 4. Example of conversion of a pattern and a mathematical expression

	A mathematical expression	A pattern
Two-dimensional representation	$\left(a+\frac{\sqrt{3}}{2}\right)x^{n}$	
Stringified representation	`(a+\frac{\sqrt{3}}{2})x\sup{n}`	`\frac{.+}{2}`
Converted representation	`(a+/::::/{/:::/{3/}/}/{2/})x/:/{n/}`	`/::::/{((?<arb>(?<arbc>[^{}:/\\]|/(?![{}:])|\\{|\\}|\\:|\\\\)|(?<arbs>/:+(/{(\g<arbc>|\g<arbs>)*/})+)))+/}/{2/}`

The notable processes of the conversion are "replacing keyword," "setting scope," and "calling pattern recursively."

If keywords are retained in stringified mathematical expressions, normal characters in a pattern can match characters to represent keywords. For example, the pattern of `a` matches `a` in a keyword `frac` in a mathematical expression of `\frac{1}{2}`. To avoid this problem, we replace keywords with sequences of colons (`:`) and a colon as a normal character to `\:`.

In patterns for searching mathematical expressions, a structure behaves as a single character. Therefore, the pattern of `\sqrt{2}` + is acceptable. However, quantification (+) in this pattern plays the role of "one or more of a character of `}`" when it is interpreted as a pattern with Onigmo. In conversion, every structure is enclosed by parentheses for setting the appropriate scope.

The problem that `\sqrt{. +}` matches `\sqrt{x}\frac{1}{2}` is resolved by calling patterns recursively with two notations of Onigmo, `(? <` *name* `>)` and `\g <` *name* `>`. `(? <` *name* `>)` are used for naming a pattern enclosed by parenthesis as `name`, and `\g <` *name* `>` calls the named pattern. We replace wildcard as follows using these functions (indented for readability).

```
(? < arb>
(? < arbc > [^{}:/\\]|/(?![{}:])|\\{|\\}|\\:|\\\\)
|(? < arbs >/: + (/{(\g < arbc > |\g < arbs >)*/}) +)
)
```

The replacement represents "an arbitrary character (named arbc)" or "an arbitrary structure (named arbs)." arbc has the ability to match an arbitrary character including characters for describing structures (i.e., `{`, `}`, `:`, and `/`). In arbs, an argument of a structure is zero or more of arbc or arbs with `\g <` *name* `>` notations, namely, calling arbs in arbs, and making it possible to treat correspondence of parentheses for structures (`{`and`}`).

5 Conclusion

In this paper, we proposed a GUI to facilitate searching mathematical expressions based on a pattern-matching algorithm with regular expressions. Our GUI displays patterns two-dimensionally and reduces the number of escape characters, improving usability, especially for novice users, and preventing error or omission in patterns.

In the future, we aim to add an input method that does not require the use of a mouse. Some existing GUIs for mathematical expressions, including Microsoft Word and LyX, have an input method that enables users to input untypable characters by describing a backslash and a keyword and then pressing a space bar. We plan to expand this function to input not only untypable characters but also structures and regular expressions, allowing adept users to promptly create patterns.

Acknowledgement. This work was supported by JSPS Grant-in-Aid for JSPS Fellows Grant Number 26-2758.

References

1. Yokoi, K., Aizawa, A.: An approach to similarity search for mathematical expressions Using MathML. In: 2nd Workshop Towards Digital Mathematics Library, pp. 27–35 (2009)
2. Miner, R., Munavalli, R.: An approach to mathematical search through query formulation and data normalization. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 342–355. Springer, Heidelberg (2007)
3. Zanibbi, R., Yuan, B.: Keyword and Image-based retrieval of mathematical expressions. In: IS&T/SPIE Electronic Imaging, International Society for Optics and Photonics, vol. 7874 (2011)
4. Watabe, T., Miyazaki, Y.: Pattern matching algorithm for mathematical expressions with a regular expression. IPSJ J. Inf. Process. Soc. Jpan. **56**(5), 1417–1427 (2015)
5. Miller, B.R., Youssef, A.: Technical aspects of the digital library of mathematical functions. Annals of Mathematics and Artificial Intelligence, Springer **38**(1-3), 121–136 (2003)
6. Altamimi, M.E., Youssef, A.S.: Wildcards in math search, implementation issues. In: CAINE/ISCA, pp. 90–96 (2007)
7. Kovalchuk, A., Levitsky, V., Samolyuk, I., Yanchuk, V.: The formulator mathml editor project: user-friendly authoring of content markup documents. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) AISC 2010. LNCS, vol. 6167, pp. 385–397. Springer, Heidelberg (2010)
8. Onigmo. https://github.com/k-takata/Onigmo
9. MathML. http://www.w3.org/TR/MathML/