数理解析研究所講究録
941 巻 1996 年 229-238

29.

Mathematical Notation Understanding Methodology

趙燕結 Yanjie Zhao(久留米工業大学知能工学研究所)
櫻井鉄也 Tetsuya Sakurai(筑波大学電子・情報工学系)
杉浦洋 Hiroshi Sugiura(名古屋大学工学部)
鳥居達生 Tatsuo Torii(名古屋大学工学部)

29.1 Tacit Agreement, Formalization, and Grammar

Up to the present, we have considered the following mechanisms that play important role in the meaning representation of mathematical notation.

1. Mathematical symbols and their meanings, which are formed historically and are defined temporarily by mathematical domains or other symbols.
2. Fixed two-dimensional structures or patterns of these symbols, which have fixed meanings and can be constructed recursively.
3. Locations of these symbols within these structures, which can not be at arbitrary position and have to comply with tacit typesetting agreements.
4. Tacit agreements about symbol omission, such as omission of parentheses and operators, for efficient communication and conciseness, which are not explicitly expressed by symbol.

The detail discussion for the 1st and 2nd can be read in our former research [1][2] and in Section

2-4 successively. We discuss emphatically the 3rd and 4th in this paper. We classify all the tacit agreements mentioned above into the following three kinds.

1. **Typesetting tacit agreements**: No matter what kinds of writing methods you use, pen-paper, publishing by publisher, computer hand-written input, computer image scanning input, or computer keyboard-mouse input, you have to obey the two-dimensional row, column, and tree structures of mathematical notation [1], in spite of the typesetting agreements are tiny different among publishers and associations in modern times. In addition, there are some tacit agreements for line-breaking, page-breaking, etc (these need further research). These tacit agreements must be observed by any mathematical expression writing.

2. **Determinable tacit agreements**: To omit symbols, several tacit agreements about priority and association law have been accepted generally. For example, $a + b \times c$ means $a + (b \times c)$ in the most situations. These priorities are determinable or definite in a context, in spite of the different kinds of them (e.g., logical operators are prior to relation operators in one priority, and behind relation operators in another).

3. **Indeterminable tacit agreements**: To omit more symbols, several other tacit agreements about priority has been accepted in a quite widespread range, e.g., $\sum_{k=1}^{\infty} a^k \sin k\pi s + a$ often means $(\sum_{k=1}^{\infty} a^k \sin(k\pi s)) + a$; not means $(\sum_{k=1}^{\infty} a^k)(\sin k)\pi s + a$. However, some cases are difficult to determine or indeterminable. For example, what the $\sin(n + m + l)x$ means? $(\sin(n + m + l))x$ or $\sin((n + m + l)x)$? This may be dependent of conventions, different contexts, different fields, and personal habits.

The mechanism "indeterminable tacit agreements" has not been found in programming languages. The problems and conclusions about the 3 mentioned above have not been discussed in contemporary existed researches [3][4] yet.

We have defined a **formal representation** [1] to abstract the structures and locations of mathematical symbols and to express the typesetting and indeterminable tacit agreements. We have also established a grammar (containing the syntax and semantics) [1] to describe the structures and meanings, and to represent the determinable tacit agreements. Here, we clarify the relation of tacit agreement, formal representation and grammar as follows.

1. Formal representation can be used to express typesetting tacit agreements explicitly at least. Formal representation is an abstract model that is independent of the typesetting, i.e., formal representation has not line-breaking, page-breaking, and many other typesetting details. The practical input of mathematical expression is a WYSIWYG representation, i.e., **quasi-formal representation** so called [2], which is dependent of different typesetting methods and techniques (e.g., $\bar{x}_{ij}$).

2. Formal representation can also be used to express all determinable and indeterminable tacit agreements. For example, to parse expression $a + b \times c$, we can simply input it like as $a + \boxed{b \times c}$. Similarly, $\sin(n+1)x$ can be input as $\sin\boxed{(n+1)x}$ or $\boxed{\sin(n+1)}x$.

3. Designing grammar clearly and introducing grammatical categories elaborately can realize the description of determinable and indeterminable tacit agreements. For example, to parse expressions like as $a + b \times c$, we can establish rules like as <calculation> ::= <calculation> + <term> and <term> ::= <term> × <factor>. The indeterminable situations are not so explicit. We discuss them at Section 2 and 4.

4. In the representation of determinable and indeterminable tacit agreements, the more boxes of formal representation we use, the less grammar rules and grammatical categories we need; vice verse. Formal representation is also not unique for the row structure according to the grammar we use. **Formalization** (using box to obtain formalized mathematical expression) is able to be accepted universally, but is dependent of how to input. Certain grammar may not be accepted universally, but need not input so many boxes.

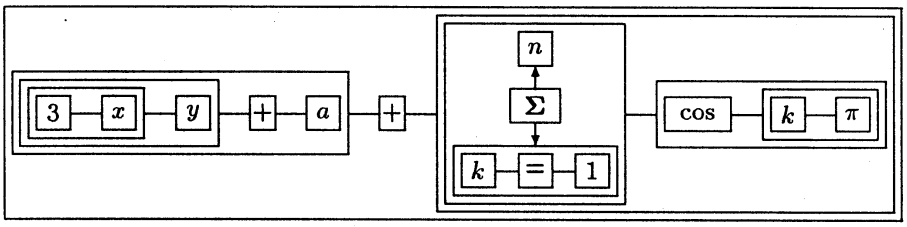
Therefore, comparing with programming languages, the formalization, as a method to express tacit agreements, is not only necessary, but also same important as grammar. Certain formalization and certain grammar construct an understanding method (see Section 2). In this point of view, we can know that enough communication is necessary during human use of mathematical notation to calculate on computer. Formalization is not a special method to aid to input and understand mathematical expressions, it is widespread and inevitable. Other necessities of formalization are shown in [1] and Section 5 of this paper.

29.2 Formalizations and Their Parsing

The formalization methods and their corresponding grammars can be classified into the following three categories according to the three tacit agreements mentioned above.

1) Strong Formalization

Strong formalization requires that every structure or operation has to be wrapped by a box. In this formalization, all tacit agreements have to be expressed by boxes. This means that all determinable and indeterminable tacit agreements can be determined by end user. Therefore, in its corresponding grammar that is called **strong grammar**, only one grammatical category (e.g., <expression>) is necessary in principle. For example, $3xy + a + \sum_{k=1}^{n} \cos k\pi$ can be formalized as below without any tacit agreement.



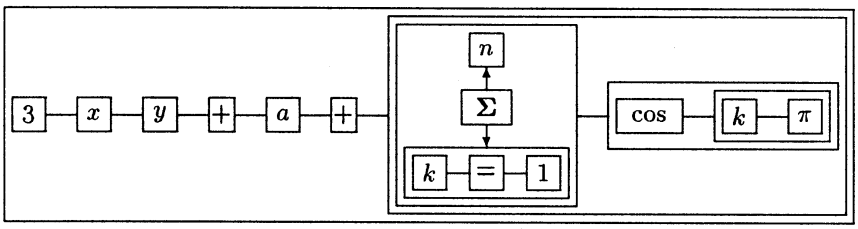
Strong formalization and its strong grammar have the strongest representation power and are suitable to any field. The grammar can also be extended easily by end user. However, excessive boxes are necessary to input.

2) Weak Formalization

Weak formalization requires that some boxes in row structure can be omitted by introducing grammatical categories into its grammar to describe all determinable tacit agreements. Its corresponding grammar that is called **weak grammar** has to introduce the following grammatical categories at least in principle [1] (other similar grammars can also be established):

1. **statement**: declaration, definition, assignment, substitution, equation-solving, control statement, etc., which often need words from native languages to clarify their structures.
2. **relation**: relation operation, such as $\in$, $\subset$, $\subseteq$, $=$, $\neq$, $<$, $\leq$, $\rightarrow$, $\equiv$, etc.
3. **calculation**: addition operation, such as binary $+$, $-$, $\cup$, $\vee$, and unary $+$, $-$, etc.
4. **term**: $\times$, $\cdot$, $/$, $\div$, $\cap$, $\wedge$, mod, $\circ$, etc. and implicit multiplication.
5. **factor**: all other operations, such as $!$, $\#$, $\neg$, ∂, $\square'$, $\bar{}$, $\ddot{}$, fraction, $(\cdot)$, $[\cdot]$, $\{\cdot\}$, $\lceil\cdot\rceil$, $\lfloor\cdot\rfloor$, $|\cdot|$, e , $\int \square d\square$, $\sum \square$, $\max \square$, $\lim \square$, matrix, vector, $\sin x$, $f(x,y)$, a_i, etc.
6. **atom**: minimal independent mathematical object, which does not contain any operation within it, e.g., number, constant (e.g., i and ∞), domain (e.g., $\mathbf{R}$), variable, function, and set. It needs no further analysis.

However, all factors, factor's operands, and atoms have to be wrapped by boxes. For the same example $3xy + a + \sum_{k=1}^{n} \cos k\pi$, its weak formalization is shown below.

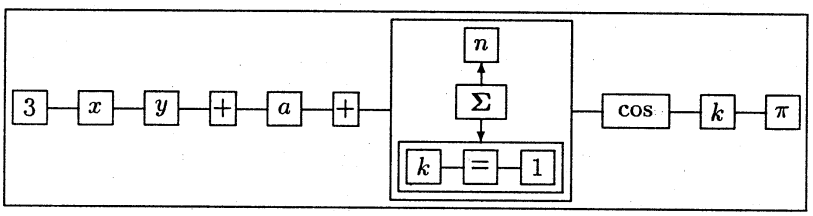


Because all factors can be input by templates [2], weak formalization can be input easily by keyboard-mouse, and there is also a big decrease in the input of necessary boxes. Weak grammar is also not difficult to be extended by end user because it contains only several easily understood

grammatical categories.

3) Free Formalization

Free formalization requires that a row, column, or tree structure has to be put into a box merely, i.e., only are the boxes for expressing typesetting tacit agreements necessary. Thus, this formalization has not any restrictions for using boxes within row structure. All boxes for determinable and indeterminable tacit agreements are omitted. For the same example $3xy + a + \sum_{k=1}^{n} \cos k\pi$, its free formalization is shown below.



To parse, we have to manage to clarify or recover the boundaries of factors in a row structure term. For this purpose, we introduce more grammatical categories besides the categories in weak grammar, according to our typical classification of factors as follows.

1. **open-close_factor**: $(\cdot)$, $[\cdot]$, $\{\cdot\}$, $\lceil\cdot\rceil$, $\lfloor\cdot\rfloor$, $|\cdot|$, $\|\cdot\|$, etc.
2. **defined_function_call_factor**: user defined function call, such as $f(x)$, $f^n(x)$, indexed variable a_{ij}, derivative $f'(x)$, y', differentiation $\dot{x}$, $\ddot{x}$, etc.
3. **reserved_function_call_factor**: like as $\log_n x$, $\tan^n x$, $\max(x, y)$, $\det A$, $\sin^{-1} x$, etc.
4. **complicated_factor**: like as $\sum_{i=0}^{n}$, $\max_{0\le i\le n}$, $\lim_{x\to a}$, $\int r(x)\frac{p(x)dx}{q(x)}$, $\int_a^b f(x)dx$, $\frac{\partial^3 f}{\partial x^2 \partial y}$, etc.
5. **simple_factor**: like as $(...)'$, x^y, $\sqrt{x}$, $\sqrt[n]{x}$, $\frac{x}{y}$, $\bar{x}$.
6. **unary_operator_factor**: $\neg$, $\sim$, $\#$, $!$, etc.

To establish the corresponding grammar that is called **free grammar**, we introduce the corresponding grammatical categories. However, one by one use of these rules to try to match so many factors has to be a right recursive syntactic analysis, that is inconsistent with the whole left recursive top-down parser [1]. Additionally, the parser also performs the corresponding context-sensitive analysis and bottom-up semantic interpretation. Thus there have to be many big backtrackings. To solve the two problems, we use the following two-parsing method (here only discuss the row structure factors) only for parsing the term expression.

Parsing 1 is a preparatory syntactic analysis to wrap factors within a term by boxes to obtain well-formed term and factors. We introduce typical rules in the following order:

1. **term** is a series of multiplication signs, the rest six categories of factors, and atoms.
2. **open-close_factor**'s operand is an open-close symbol series.
3. **defined_function_call_factor**'s operand is a structure $\square(...)$, or $\square$ $(...)$, etc.

4. **reserved_function_call_factor**'s operand is a reserved_function_call_factor, an open-close_factor, a defined_function_call_factor, a unary_operator_factor, a simple_factor, or a series of atoms and multiplication signs.

5. **complicated_factor**'s operand is a complicated_factor, or a series of factors except complicated_factors.

6. **unary_operator_factor**'s operand is an open-close_factor or an atom for !, ¬, and #; an unary_operator_factor only for ¬.

7. **simple_factor**'s operand is a tree or column.

Parsing 2 does syntactic analysis and semantic interpretation for well-formed term and factors that are the same as of the weak grammar.

The grammars for Parsing 1 are not unique. We can establish different grammars to correspond different tacit agreements or conventions. For example, while reading expression $\frac{1}{2}[1+\frac{\sin(n+\frac{1}{2})x}{\sin\frac{1}{2}x}]$, one readily know it means $\frac{1}{2}[1+\frac{\sin((n+\frac{1}{2})x)}{\sin(\frac{1}{2}x)}]$, because the $(n+\frac{1}{2})$ seems to be a coefficient and there is a similar $\frac{1}{2}$ at the denominator. However, this expression can be mistakenly parsed by the above Parsing 1, because it is inconsistent with the convention sin(...). If you prefer the former expression, you can establish your own grammar to permit the operand of sin to contain an open-close_factor following an atom. However, if the operand in the open-close_factor is very complicated, you seem to had better wrap the operand into parentheses. In short, to cope with the indeterminable tacit agreements, we can establish many different unambiguous free grammars to make all the things be determinable in one grammar. This "multi-grammar" scheme is also consistent with the practical situation of mathematical notation using, because there is more than one notation system in use.

In addition, the box wrapping an atom can also be omitted in principle, if we introduce lexical rules and do context-sensitive analysis according to mathematical object declaration and definition. However, that means very low efficient.

Free formalization permit user to input a row structure expression (e.g., by keyboard only) without wrapping any box. This is convenient to input simple 1D expressions. The box using is decreased to the least. In addition, free grammar is also possible to parse formulas input by OCR scanning after recognition and formalization. However, so many free grammars are difficult and dangerous to extend and manage by end user. This problem need further research.

29.3 Understanding Methodology

We have given a knowledge-based understanding method [1][2], which can parse only strong and weak formal representations with simple context-sensitive processing. Here we extend it to cope with all the three formalizations and complicated context-sensitive parsing (see Section 4). This method has become a set of methods, or a methodology.

The three kinds of grammar, strong, weak, and free grammars, in spite of so many common components they have, are different each other in structure pattern, number of rules, and meaning interpretations. In addition, they are not unique. Therefore, we need the same number of knowledge-bases to implement them. However, to parse these knowledge-bases, we need not necessarily the same number of parsers. In fact, we introduce a knowledge representation language that is called **meta-language** to express all these grammars, so that we create only one parser (the first parser, which is called **understanding parser**) to parse meta-language. Additionally, the end-user can also use meta-language to extend and modify certain grammar. Thus meta-language has to be designed to be user-oriented, and then any knowledge representation written in meta-language is translated into a knowledge base that is very efficient and parsing-oriented.

Formal representation can be translated by the parser into a common static meaning representation, which is called **meta-representation** [1]. Following this, it is translated into program, and then, program's execution result is translated into formal representation, and then, quasi-formal representation, for display and editing again [2]. Similarly, the formal representation can also be translated into different typesetting languages. We consider that one programming language needs one translator, but we create only one parser (the second, which is called **translation parser**) to parse all these translators written in meta-language. Similarly, based on this method, all the translations from execution results into formal representation needs only another parser (the third, which is called **reply parser**) too, and all the translations from formal or quasi-formal representations into various typesetting languages (e.g., LaTeX, AMS-TeX, etc.) needs one parser (the fourth, which is called **typesetting parser**) merely. All the knowledge bases can be established through translation from these grammars in meta-language. Whole knowledge-based understanding methodology is shown in Fig.1.

The extensible translation from meta-representation into program is necessary, because **1)** program is a computer simulation of the uncomputable meta-representation; **2)** the translation is a further interpretation of meta-representation, which includes "all" the meanings, but not express "all" explicitly; **3)** meta-representation can be translated into program in various programming languages (e.g., MATHEMATICA, MAPLE, AXIOM, etc), because anyone of them can not satisfy all user's requirements, and one is unable to solve or is incorrect in many calculations and others are able to or correct [5].

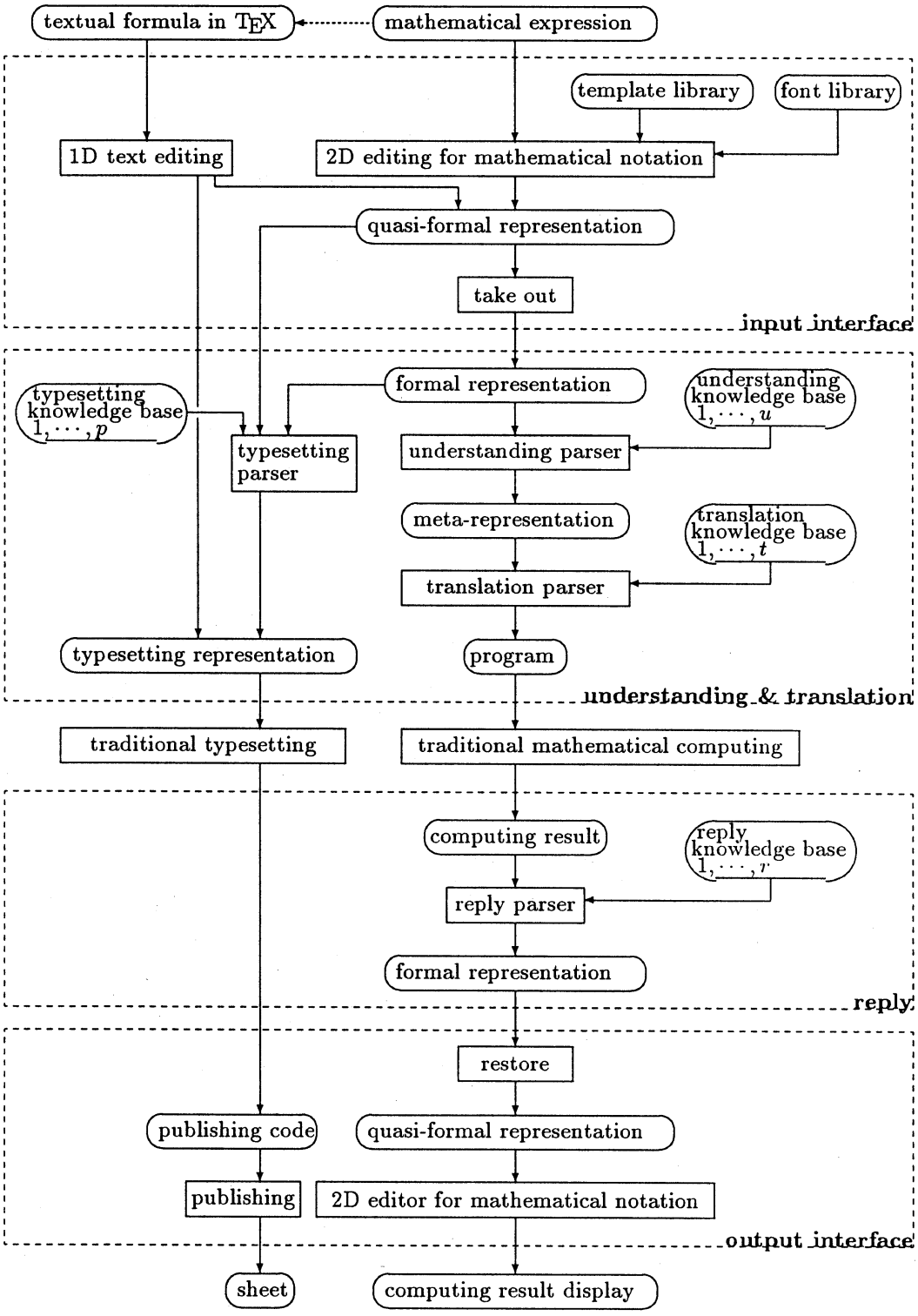


Fig.1 A Methodology of Knowledge-Based Understanding

29.4 Meta-Language

The fundamental construction of the understanding knowledge is the same as of our former one [1]. All grammars are written in meta-language (typically, 3 grammatical categories for strong; 8 for weak; and 24 for free in our current prototyping). In addition, many new actions are attached into some rules for context-sensitive analysis and a big decrease of the number of rules (only about 200). Meta-language is constructed through 11 primitives for declaration, and 46 meta-phrases: 17 meta-structures for rule structure description and context-sensitive processing (e.g., $\square \leq \square < \square$); 13 meta-attributes for marking the grammar, formalization, batch or interaction, specified programming language of a file, and if a reply is a computing result or error message; and 16 meta-actions to denote context-sensitive processing action (e.g., replacement of a number, constant, declared symbol, arbitrary expression, row or column size, etc) and to denote a repetition, selection, and option structures for decreasing number of rules dramatically.

29.5 Formal Representation+Normal Code

The inner data representation of the formal representation is called **1D formal representation**, which is defined as follows:

```
<expression> ::= '<string>' |
    [ row, <expression list> ] | [ column, <expression list> ] |
    [ tree, <expression>{, back_sup, <expression>}{, upper, <expression>}
        {, front_sup, <expression>}{, front_sub, <expression>}
        {, lower, <expression>}{, back_sub, <expression>} ]
<expression list> ::= <expression> | <expression>, <expression list>
<string> ::= <sign in Normal Code> | <sign in Normal Code> <string>
Note: sign ' in <string> is denoted as '' and { } means option.
```

Where the **normal code** is an extensible normalized clear-text code for mathematical notation (containing various style Arabic numerals, Latin, Greek and Cyrillic letters, and mathematical signs). Its font style primitive is denoted like as {`@rm ...`} ; its sign primitive `\...`, and some are same as of TEX and others' not, e.g., `\factorial` for !. Only one \ is in one primitive, e.g., `\not\leq` in TEX is `\notleq` . One sign one primitive, e.g., `\backslash` and `\setminus` in TEX are denoted as `\backslash` . Structured symbols are denoted by formal representation and

primitive, e.g., `\sqrt{...}` in TEX is expressed to be `[column, '\sqrt', ...]`.

The 1D formal representation together with normal code is equivalent to the formula part of TEX. TEX together with necessary boxes can also be understood by our methodology in principle. The reasons that we do not use TEX to express inner data directly are as follows.

1. Formal representation is an abstract model, thus the necessary minimum information for meaning understanding is preserved. This model is simple and efficient for meaning understanding. The typesetting details are unnecessary. TEX is a user-oriented typesetting language, and is very big and complicated.
2. Thus the meaning understanding can be independent of typesetting and input. Formal representation is independent of the development of TEX and publishing technology.
3. Thus sign and its location are separate. While extending, we can only introduce new signs, not new structures. In addition, this separation can obtain more powerful representation, e.g., `\bar{}`, `\overline{}`, `\hat{}`, `\widehat{}`, `\check{}` , `\tilde{}`, `\widetilde{}`, `\grave{}`, `\ddot{}`, etc. can be abstracted to be `[tree, ..., upper, <sign>]`.
4. In less than 20 years of her short history, TEX (or Plain TEX) has derived LATEX, AMS-TEX, AMS-LATEX, and so on and so forth. There is not a standard.

TEX is an amazing achievement. Her original design goal is not an abstract model for mathematical notation understanding. TEX will become a 1D textual input way of Fig.1 in future.

参考文献

[1]Yanjie Zhao, Hiroshi Sugiura, Tatsuo Torii, Tetsuya Sakurai: A Knowledge-Based Method for Mathematical Notations Understanding. *Transactions of Information Processing Society of Japan*, Vol.35, No.11, pp2366-2381, Nov.1994.

[2]趙燕結, 桜井鉄也, 杉浦洋, 鳥居達生: 自然な数式ヒューマン・インタフェースに関する研究. 情報処理学会研究報告 (IPSJ SIG Notes). Vol.95, No.70, pp57-64, 1995.7.

[3]Kajler,N., Soiffer,N.: A Survey of User Interfaces for Computer Algebra Systems (to appear in *the Journal of Symbolic Computation*, preprint: RIACA Technical Report #1), Jun.1994

[4]Soiffer,N.: Mathematical Typesetting in Mathematica. Levelt,A.(Ed.) *ISSAC'95 Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation, Jul.10-12, 1995, Montreal, Canada*, pp140-149, ACM Press 1995.

[5]Wester, M.: A Review of CAS Mathematical Capabilities. Godoy, L.A (Ed.):*Applied Mechanics in the Americas*, Vol.3, pp450-455, American Academy of Mechanics and Asociacion (Postscript copy `ftp math.unm.edu/pub/cas` is also available), 1995.