Vol. 35 No. 11 Transactions of Information Processing Society of Japan Nov. 1994

Regular Paper

A Knowledge-Based Method for Mathematical Notations Understanding

YANJIE ZHAO,[†] HIROSHI SUGIURA,[††] TATSUO TORII[††] and TETSUYA SAKURAI[†††]

The understanding of mathematical notation by computer is still an unsolved problem. Resolving this problem can lead to many important applications, particularly in the man-machine interface of mathematical computation. We begin our research from fundamental concepts and formalization of all mathematical notations. Based on these concepts and formalization, we study the syntax and semantics of mathematical notation. According to the syntax and semantics, we implement a knowledge base to solve the problem of knowledge representation of mathematical notation, and implement a parser to translate mathematical expressions into their equivalent functional representations, which are independent of different applications. It is also possible to extend the knowledge base by the user for satisfying the feature of immediate extensibility of mathematical notation. Around the knowledge base and the parser, we can establish a universal interface for different applications systems, to combine the understanding of the formalized mathematical expressions with the document processing of them.

1. Introduction

Mathematical notation has become a language for serious mathematical thinking. However up to the present, human beings have difficulty to directly input computational problems in mathematical notation into a computer. We still have to translate between mathematical expressions and computer language programs. From a long-term point of view, our research aims to do the following:

1. To study the structure and meaning of all mathematical expressions so as to recognize their linguistic and logical properties.
2. To integrate document understanding with document processing of mathematical expressions.
3. To improve the man-machine interface in many application fields, e.g., mathematical computation (numerical computation, computer algebra, and automated theorem proving), computer aided education, scientific document processing (even automatic proof-reading), algorithm representation, and software specifications.

† Intelligence Engineering Laboratory, Kurume Institute of Technology
†† Department of Information Engineering, Faculty of Engineering, Nagoya University
††† Institute of Information Sciences and Electronics, University of Tsukuba

Research in mathematical notation is only a part of the historical research of mathematics before the age of the computer.[1] Modern mathematical notation began at the start of this century.[2] The research in computer processing of modern mathematical notation has spread to the following fields since 1961.

Early Programming Languages: early high-level programming languages which contain some mathematical notations since 1961.[3)-13)] It was at this time that the necessity and applicability of mathematical notation in computer science was realized.

Pattern Recognition: recognizing images of mathematical notations.[14)-19)] The first two-dimensional grammar for some mathematical notations was developed in this field.[14),16)]

Document Processing: editing, typesetting and printing all natural, original, and orthodox mathematical expressions.[20)-27)] The interactive editing and typesetting of two-dimensional mathematical expressions was achieved.

Software Specification: executable languages with mathematical symbols for expressing formal specifications of computing systems and their fast prototyping.[28)-32)]

Mathematical Software Interface: the two-dimensional formula and graphical interface of mathematical computation systems since 1985.[25),33)-46)] This kind of interface permits the user to input two-dimensional mathematical expressions directly, and translates the user's input into the input of several different computer algebra systems.

In conclusion, the document processing of all mathematical expressions has achieved some success. On the other hand, the mathematical software interfaces mentioned above use a necessarily minimal sub-set of mathematical notations, and do not consider the whole (e.g., the **declarations in mathematical literature** in particular). Because they are developed as computer languages, they avoid the ambiguities of mathematical notation, but they have the following problems in view of the use of natural mathematical notation.

1. They lose part of the representation power of mathematical notation.
2. They are different and have no universality.
3. They are not easy to extend by the user.

A user-friendly, two-dimensional editor to input, typeset and print mathematical expressions has been implemented. However, rigorous research in computer processing of natural mathematical notation is still in its infancy. Much research has outgrown a very-high-level programming language to approach the representation of natural mathematical notation. In contrast, we do not design and implement a new very-high-level programming language, but we begin our research from whole natural mathematical notation. We give a knowledge-based method for understanding mathematical expressions in this paper, and then, plan to develop many applications in different fields.

In Chapter 2, we begin by giving fundamental concepts about mathematical notation. In Chapter 3, we make a formalization of mathematical notation to lay a foundation for our research. In Chapter 4, we study the structure and meaning of the formalized mathematical expressions to lay a base for our knowledge-based understanding method described in the following two chapters. In Chapter 5, we implement a knowledge base. In Chapter 6, we implement a parser based on the knowledge base. In Chapter 7, we put forward a scheme to realize a system of many applications with a universal interface for mathematical notation.

In Chapter 3, we define a formal representation which represents the recursive two-dimensional structure of mathematical notation. We divide the problem of understanding mathematical notation into two sub-problems. The first is how to input the formal representation. There are several ways to do it, e.g., direct input by mouse and keyboard, translation from usual text data, and recognition from optical data. The second is how to understand the formal representation. In this paper, we start from the first, and then, concentrate on the second.

2. Fundamental Concepts

What we are concerned with in our research is only modern mathematical notation. We give the fundamental concepts about modern mathematical notation as follows.

Mathematical expression (in a broad sense) means any expression (i.e., formula, definition, graph, text, table, image, etc.) in mathematical literature.

Mathematical expression (in a narrow sense) (math expression, for short) means formula, definition, as well as related text (e.g., let, define, where, for, if, given, etc., because the native verbal language is always needed in order to interpret and amplify the meaning or usage of the non-verbal logogram representations[47)]) in mathematical literature. This concept in the narrow sense is used in our current research.

String used in mathematics (string, for short) means a sequence of characters to construct a minimal meaningful part of a math expression. Namely, a string is the morpheme of math expressions, i.e., numeral, the core part or index part of an object, operator, separator or ellipsis in math expressions.

Character used in mathematics (character, for short) means any member of any character set, such as a Latin letter, Greek letter, Arabic numeral, special letter, mathematical sign, or even a character of any native language (even a chinese character or a japanese katakana character), used in mathematics.

Mathematical notation (math notation, for short) means the set of construction rules of math expressions from strings ; a logographic writing system for representing mathematics but its spoken equivalent varies from language to language.[47]

While reading a math expression, we see the visual image of it at first, then we can understand its meanings through its structure and with the knowledge of mathematics. Math expressions have complicated two-dimensional structures, but they are not arbitrary graphics in a coordinate system. Their components do not overlap each other. The locations and geometric adjacents among a math expression's components relate with its meaning representation. We can find the following features about their structures.

1. A string is a morpheme of a math expression. For example, the $\frown$, f, k and 2 are strings in a function name $\widehat{f}_{k_2}$. The sin and cos are also morphemes in $\sin^2 x+\cos^2 x$.
2. Math expressions have the following three kinds of fundamental structures : 1) A row of expressions to represent operators and their operands (e.g., $\int f(x)\mathrm{d}x$ and $ab+bc+ca$), and a sequence arranged horizontally (e.g., $a, b, c,\cdots$), 2) A column of expressions to represent other operators and their operands (e.g., $\frac{a}{b}$), and a sequence arranged vertically, and 3) A superscript-subscript tree of expressions to represent operation (e.g., x^3 and $\bar{x}$) and modification to make new mathematical objects from the original one (e.g., $\widehat{f}, f_1, f^{<1>}$ and $\widehat{f}_1^{<1>}$ are made from f).
3. Any math expression can be derivated recursively from the three structures (e.g., $\frac{e^x}{x}+y^{\Sigma_{i=1}^{n} z_i}$). Math notation, as compared with native languages, do not rely on phonetics to delimit structure boundaries and distinguish different meanings. They have complicated recursive structures and can be strictly expands according to these structures. So they have stronger representation power and can avoid many ambiguities.[47]

Next, we propose an abstract data structure and a formal grammar.

3. Formalization

The formalization of math notation can achieve the following intentions :

1. To give the formal definition of the universal set of math expressions,
2. To determine the data structures of math expressions, and
3. To define a two-dimensional meta-language for describing the grammar of math expressions, because Backus-Naur Form is inconvenient for non-verbal math notation.

Now we give a brief definition of the formal representation of math expressions as follows.

A **formal expression** (expression, in short) is a string, or a structure of strings. Every expression occupies one rectangle of space denoted by a box, like this : $\boxed{\text{expression}}$ A string is put into a box, which is called an atomic box. Every box is located in a structure of boxes.

We define the **formal representation of mathematical expressions** (formal representation or 2D formal representation, for short) in **Fig. 1**.

where, the two undirected lines "—" mean horizontal and vertical concatenate adjacent relations of two boxes. The six directed arrows "→" mean super-script and sub-script subordinate

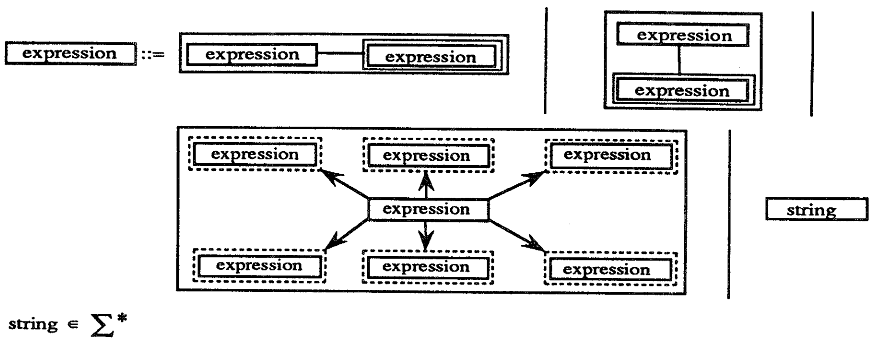


Fig. 1 Definition of 2D formal representation.

adjacent relations of two boxes. The duet box $\boxed{\boxed{\text{expression}}}$ means the repetition of adjacent relations from 0 time. The dashed box $\boxed{\text{expression}}$ means option of the box $\boxed{\text{expression}}$. The "|" means disjunction. The "::=" means definition. Σ^* is the set of strings with the character set Σ, which contains any characters of native languages and mathematical signs. In our implementation of the formal representation (i.e., the equivalent 1D formal representation, see an example following the Fig. 10 in Chapter 6), a string is a sequence of primitives and boxes of TEX.[21] Therefore, we can express the typesetting information (e.g., style, size, position, etc.) of a string in TEX within an atomic box.

From the definition above, every math expression is a structure of a row or column of undirected linkages of boxes, a directed hexa-tree of boxes, or only an atomic box. The set of expressions produced by the formal representation is called to be **Universal Set of Mathematical Expressions**, and denoted by U. If we denote the set of all math expressions as M, then it is sure that $M \subset U$.

Boxes in the formal representation are used to express not only the geometrical structures of math expressions, but also the **boundaries of meanings** to avoid many ambiguities and to represent the user's intention correctly. The formal representation holds both the image pattern and the meaning structure of math expressions. For example, if f_i^3 means $(f_i)^3$, its formal representation is in **Fig. 2**.

An operand of an operator and a factor which occupies a row of boxes have to be put into one

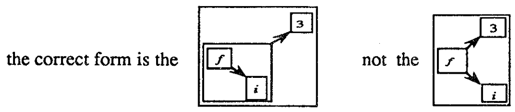


Fig. 2 Example of correct use of boxes.

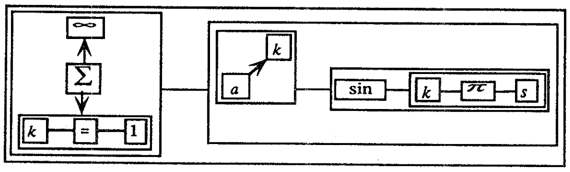


Fig. 3 Example of meaningful use of boxes.

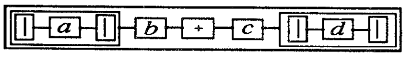


Fig. 4 Example of box usage for open-close structures.

box. For example, if $\sum_{k=1}^{\infty} a^k \sin k\pi s$ means $\sum_{k=1}^{\infty} (a^k \sin(k\pi s))$, it has to be represented as **Fig. 3**.

The open-close structure (i.e., $(\cdot)$, $|\cdot|$, $\lceil\cdot\rceil$, etc.) must be within one box and the expressions between the open and close signs must be within one box. For instance, we have to build the expression $|a|b+c|d|$ with the meaning $(|a|\times b)+(c\times|d|)$ as shown in **Fig. 4**, to distinguish from another meaning $|a\times(|b+c|)\times d|$.

In addition, the same strings with different sizes are regarded as the same string, e.g., in xe^x, the two x are the same string. But the letters with different styles are regarded as different letters, e.g., A, $\boldsymbol{A}$, A, **A**, etc.

Someone may think that it is very troublesome to input these box structures, considering both the meaning and the visual image of math expressions. But our aim is to establish a man-machine interface for scientists and engineers who work with mathematics. It is a natural way for people who are familiar with mathematics to construct math expressions with suitable structures.

In the sense of the formal representation, we study the knowledge about the relation between structures and meanings of math expressions accurately in the next chapter.

4. Structures and Meanings

The strings can be constructed into an independent object or operator which is often called a mathematical symbol (defined in 4.1). Math expressions are constructed recursively by mathematical symbols (developed in 4.4). The meanings of math expressions (explained in 4.2) are determined by their structures and the domains they belong to (discussed in 4.3), and also rely on declarations of their variables (revealed in 4.5).

4.1 Mathematical Symbols

Mathematical Symbol (math symbol, for short) means an expression of an object (number, constant, variable, set, etc.), operator ($+$, sin, $\int dx$, etc.), delimiter (e.g., parentheses), separator (e.g., comma), or ellipsis ($\cdots$, etc.) used in math expressions.

```
function: <function_name> ( [meta-representation] , ... , [meta-representation] )
domain: <domain_name>
```

Fig. 5 Definition of meta-representation.

Each math symbol plays a different role in the structure of a math expression. Here, **structure** means a math symbol adjacent relation defined by formal representation. All math symbols can be classified into fixed math symbols and changeable math symbols.

Fixed math symbols are the math symbols that are used to describe fixed individuals, operators and auxiliary symbols as follows:

Fixed individuals contain
  numbers, constant names (such as π, e, $\emptyset$) and **domain names** (such as $\boldsymbol{R}$).

Operators (must have operands) are $+$, $-$, sin, !, $\int dx$, $\sum_{i=m}^{n}$, $|\cdot|$, etc.

Auxiliary symbols (can not be independent of other symbols) contain
  delimiters to change operation priority, such as $(\cdot)$ in $(a+b)\times c$, or to wrap something as a whole to distinguish other things, such as $(\cdot)$ in $f(x, y)$,
  separators as a boundary between two components, such as the comma in $f(x_1, x_2)$,
  ellipses to show a repetition structure, such as $\cdots$ in $x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\cdots$.

The meanings and usages of existing fixed math symbols have been formed historically and have been accepted by the whole world. Fixed math symbols are invariant components of math expressions. They do not need declarations to say what they are.

Changeable math symbols are the math symbols that are used to describe variables. Therefore, they need declarations to say what they are and what domains they belong to. They can have different meanings and usages in different documents, or even in the same document. Some examples of changeable math symbols are variable x, vector $\boldsymbol{a}$, matrix A, function f, set A, and arbitrary constant C_1.

From this classification, we find that the fixed math symbols determine the structure of math expressions. Namely, fixed individuals are independent objects which do not need further analysis. Operators connect their operands to

```
function: definite_integral (
        function: integral_element ([ ])
        domain: set_of_real_number                          ,
        function: lower_bound([ ])
        domain: <subset of set_of_real_number>              ,
        function: upper_bound([ ])
        domain: <subset of set_of_real_number>              ,
        function: integrand([ ])
        domain: set_of_real_number                          )
domain: set_of_real_number
```

Fig. 6 Example of meta-representation for definite integral expression.

create an operation. Auxiliary symbols are also similar to operators. For example, parentheses, commas and ellipses can construct a structure $(a_1, a_2, \cdots, a_n)$. They connect $a_1, a_2, \cdots, a_n$ together to form an object, i.e., the comma seems to be a connection operator, the ellipsis means a repetition structure, and finally, the parentheses wrap them as a whole to distinguish them from other objects. Contrarily, changeable math symbols are changeable components in a structure.

4.2 Meanings of Mathematical Expressions

The meanings of a math expression is dependent on its interpretation,[48),49)] i.e., fixed math symbols have their names (values, constant names, function names, etc.) in a native language, and changeable math symbols have their declared domains (sets of objects from which they abstract), based on mathematical knowledge. This kind of interpretation is the denotational semantics of math expressions, and can be represented as the form of functions with their domains.[48)] Therefore, the meanings of any math expression can be denoted by composed functions with the codomains of these functions.

In our research, we define a meaning representation. It only says what to do, not how to do, so it is independent of any concrete data types and algorithms. In this paper, the math notation **understanding** means a translation by computer from a formal representation into its meta-representation defined as below.

Meta-representation of a math expression is an object which contains function and its codomain (both names are written in English) to denote a meaning of the math expression, i.e., meta-representation is an object shown in **Fig. 5**.

For example, the math symbol $\int_{\Box}^{\Box} \Box \, \mathrm{d} \Box$ on domain $\boldsymbol{R}$ (set of real numbers) can have the meta-representation shown in **Fig. 6**.

4.3 Relation between Structures and Meanings

The same fixed math symbols in different structures can describe different meanings, e.g., $\rightarrow$in $\vec{a}$ and in f: $A \rightarrow B$ and in $p \rightarrow (q \rightarrow p)$. Therefore, we have to distinguish different meanings according to different fixed math symbols in different structures.

Nevertheless, the same fixed math symbol in the same structure can also have different meanings, because the changeable math symbols could belong to different domains, and the output of a function may be in a different codomain according to different input in the same domain. In addition, one structure may correspond to different function names. For example, the structure of binary operation $\Box \times \Box$ means :

1. product of
   (a) integers $\boldsymbol{Z} \times \boldsymbol{Z} \rightarrow \boldsymbol{Z}$
   (b) real numbers $\boldsymbol{R} \times \boldsymbol{R} \rightarrow \boldsymbol{R}$
   (c) matrics $\boldsymbol{R}^{l \times m} \times \boldsymbol{R}^{m \times n} \rightarrow \boldsymbol{R}^{l \times n}$
   (d) $\vdots$
2. Cartesian product of $\cdots$
3. vector cross product of $\cdots$
4. $\vdots$

Therefore, one structure can have different function names, and one function name can have different codomains with its parameters on different domains. In other words, one structure corresponds to different meanings—the function and its codomain.

In light of what is mentioned above, knowledge about the inclusion relations of domains is also very important. With this knowledge, after defining operator $\sqrt{\ }$ on $\boldsymbol{R}$, we can understand $\sqrt{2}$ because natural number set $\boldsymbol{N} \subset \boldsymbol{R}$.

4.4 Construction of Mathematical Expressions

Math expressions can often be expressed concisely and accurately while omitting many parentheses because of the following two agreements.

First, operators permit a computational **priority order**. For example, factors are the top priority, such as $n!, \lceil x \rceil, |x|, e^x, \sin x, \int f(x)$ $\mathrm{d}x$, $\sum_{r=0}^{n} f(r)$, $a_{\lceil \frac{n}{2} \rceil}$, etc. ; and then, second priority term, such as $\boldsymbol{a} \times \boldsymbol{b}$, $\boldsymbol{a} \cdot \boldsymbol{b}$, x/y, $A \cap B$, follows ; and then, the third one is addition, such as binary $+, -, \cup$, and unary $+, -$, etc.

Secondly, some binary operators satisfy the **associative law**, i.e., $\forall x \forall y \forall z ((x \Box y) \Box z = x \Box (y \Box z) = x \Box y \Box z)$, where $\Box$ denotes an arbitrary binary operator with left-right operands. Some binary operators with the same priority satisfy only the **left associative law**, e.g., for subtraction, $a-b-c=(a-b)-c \neq a-(b-c)$; or only the **right associative law**, e.g., $a ** b ** c = a ** (b ** c) \neq (a ** b) ** c$ in FORTRAN.

The construction of a math expression finally attains statements, which often need words or phrases of native languages as components, e.g., "Let", "Solve" and "for" in "Let $x \in \boldsymbol{R}$" and "Solve $f(x)=0$ for $x \in \boldsymbol{R}$". These words are not math symbols ; but they are necessary for understanding.

4.5 Declaration of Changeable Mathematical Symbols

Declaration is a math expression which is used to specify the meaning of changeable math symbols, and to define new math notations immediately. It influences the structure and meaning of other math expressions through defining structures of new math notations, and through defining the domains and scopes of their changeable math symbols. We call this kind of influence to be a declaration effect.

Declaration effects are semantic structures to show the domains, usages and scopes of declared symbols in the structures of declarations.

We explain various declaration effects (domain, usage and scope) as follows.

The **domain** of a declared changeable math symbol is a set in which it exists. The description of domain in a declaration can be of the following three kinds : 1) A definition with equivalence. For example, "Let $x=c$" means that x is a declared symbol, and has the same domain as c. 2) Declared variables are elements of a domain, e.g., "Let $x, y \in \boldsymbol{R}$". 3) We can declare function f as "Let $f \in B^A$" with the notation in 2). But the notation "$f: A \to B$", with the same meaning, is popular in use. Through the later notation, we can clearly recognize the domain A and the codomain B of f.

Some function declarations also contain definition to say how to do, and/or its **usage** to say how to use. For example, when we read "For $x \in \boldsymbol{R}, f[x] \overset{\mathrm{def}}{=} x^2$", we know that the declared symbol is f, that its domain is $\boldsymbol{R}$, that its codomain relies on the definition x^2, that the x is a typical element which is meanful only within the definition, and that the usage of f is also specified, i.e., it must appear in its scope at the form $f[\ \ ]$.

The specified **scope** in a declaration can be divided into two kinds: 1) Usual Scope means that the scope of a declared symbol is its following document till its next declaration, or its "removing declaration" (e.g., in Mathematica[50]), or the end of the whole document. 2) Localized Scope covers only a specified area of the document. One situation is a "post-scope" declaration, e.g., "Let $y=f(x)$, where $x \in \boldsymbol{R}$". Another is **bound variable**.[48] A bound variable is often declared within an operation. For example, x in $\int_a^b f(x)\mathrm{d}x$ is a bound variable; its domain relies on the domains of a and b as well as the definition of definite integral; its scope is $f(x)$. Therefore, the analysis sequence of the operation with bound variables is domains of bound variables, and then, the bound variables, and finally, the scopes of the bound variables, i.e., a and b, and then, x, and finally, $f(x)$.

5. Knowledge Base

All considerations in Chapter 4 are implemented by a knowledge-based method, which contains a knowledge base discussed in this chapter and its parser discussed in the next chapter. The knowledge base contains a series of rules being explained as follows.

5.1 Rule

The structure and meanings of a math expression can be represented by a **rule**. Math expressions can be understood through a series of rules recursively. An abstract hierarchy of a rule is shown below.

Rule is composed of

- **Structure**, which contains a **structure rule:** ⟨category⟩ → ⟨structure_pattern⟩
- **Declaration Effects**, as an option
- **Meanings** 1, which contains
  - **Function**, with the form ⟨function_name⟩ (⟨category⟩ or ⟨function⟩ ,⋯)
  - **Domain Rule** 1, which has the form ⟨codomain⟩ ← ⟨domain_pattern⟩
  - ⋮
  - **Domain Rule** m,⋯
- ⋮
- **Meanings** n,⋯

The **structure pattern** describes the common structure of a class of math expressions. It is composed of fixed math symbols and substructures. All its sub-structures are denoted by categories (non-terminal symbols), and the fixed math symbols as terminal symbols appear in rules explicitly. While creating structure rules, we must support the two agreements in 4.4, i.e., the priority order can be implemented through arranging rules into a certain sequence. A typical priority order has been defined in 5.2. The associative law can be represented in the rules of the following forms: $\langle A \rangle \to \langle A \rangle \ \square\ \langle B \rangle$ for the left associative law only; and $\langle A \rangle \to \langle B \rangle\ \square\ \langle A \rangle$ for the right associative law only. Here, $\square$ denotes an arbitrary binary operator, A and B denote categories. While analyzing a row of boxes, the former begins from the back end and the latter begins from the front end. For example, in order to accept the structure of binary operator $\times$ with its two operands, we can define a structure rule: $\boxed{\text{term}} \to \boxed{\text{term}} \times \boxed{\text{factor}}$. In addition, the words written in native languages are also regarded as fixed math symbols.

The **declaration effects** only appear in the rules for describing declarations and operations using bound variables to represent the semantic structure of declaration. They are necessary because the semantic structures have to be represented explicitly, so that the language of math expressions can be extended by the user (see 5.3).

One of the **meanings** describes one group of the meanings of a structure. It contains the naming function of the structure and its corresponding domain rules. The function and a codomain are composed of a meaning or result of understanding. The **domain pattern** has the same framework as its corresponding structure pattern. For the given example $\boxed{}\times\boxed{}$ mentioned in 4.3, we can create the domain rules like as : $\boldsymbol{R}\leftarrow\boldsymbol{R}\times\boldsymbol{R}$ for real number product, $\boldsymbol{R}^n\leftarrow\boldsymbol{R}^n\times\boldsymbol{R}^n$ for real vector cross product, etc.

Note that there is a mixture of the terminology "function" $y=f(x)$ based on two ideas[48] : the idea of a function as a many-one correspondence, and the idea of a function as a variable y which ranges in relation to another variable x, so that the value of y is always fixed by that of x. We hold the second idea as an agreement during creating any domain rules.

The whole knowledge base is composed of a series of rules and **background knowledge**. Background knowledge contains the correspondences of domain symbols to domain names and the domain inclusion relations. It can also be extended by the user.

5.2 Categories

The naming of categories is convenient for knowledge representation and user extension, because there are only seven categories and they all correspond to mathematical concepts. Through the seven categories, we can organize almost all math expressions into the knowledge base in a typical priority order (the bottom is prior to the top) as follows :

1. **Expression** is a statement, such as a declaration, a definition, an equation-solving, or an assignment, which is constructed by logic_expressions.
2. **Logic_expression** is an expression which is constructed by relation_expressions and logic operators, e.g., $\vee$, $\wedge$, etc.
3. **Relation_expression** is an expression which is constructed by addition_expressions and relation operators, e.g., $\in$, $\notin$, $\subset$, $\subseteq$, $=$, $\neq$, $>$, $\geq$, etc.
4. **Addition_expression** is an expression which is constructed by term_expressions and addition operators, e.g., binary $+$, $-$, $\cup$, etc. and unary $+$, $-$.
5. **Term_expression** is an expression which is constructed by factor_expressions, multiplication operators (e.g., $\times$, $\cdot$, $/$, $\div$, $\cap$, etc.), function composition operator $\circ$, as well as omitted or invisible operators between two factors (e.g., ab means a times b).
6. **Factor_expression** is an expression which is constructed by atom_expressions, unary operators (e.g., $\neg$, !, #, etc.), fraction stroke, open-close operators (e.g., $(\cdot)$, $[\cdot]$, $\{\cdot\}$, $\lceil\cdot\rceil$, $|\cdot|$, etc.) exponential operation (e.g., e^x), complicated structured operations (e.g., $\int f(x)\mathrm{d}x$, $\sum_{r=0}^{n} f(r)$, etc.), $\sin x$, $f(x, y)$ and indexed variables.
7. **Atom_expression** is a minimal independent mathematical object which does not contain any expression within it. It has been declared in a declaration, or does not need declaration. It can be a number, constant name (e.g., imaginary unit i,[27] base of natural logarithm e,[27] number π, and infinity ∞), domain name (e.g., set of real numbers $\boldsymbol{R}$), variable name, function name, or set name. It needs no further analysis and interpretation.

5.3 Declaration Effects

To implement the declaration effects, we use a series of primitives in a rule to describe all necessary information about a declaration, and to activate corresponding implementation actions in the parser. All primitives used in this knowledge base are introduced as follows :

symbol (formal expression of category,$\cdots$) specifies where the declared symbols are.

equivalent (category) shows that the declared symbol has the same domain as that of the category.

domain (category,$\cdots$) specifies where the domain of the declared symbol is.

codomain (category,$\cdots$) specifies where the codomain of the declared symbol is.

typical_element (category,$\cdots$) specifies where the typical variable is.

definition (category) specifies where a definition is.

usage (⟨rule⟩) describes the usage of the declared symbol through the rule.

scope (category,$\cdots$) specifies where the scope of the declared symbol is.

All primitives can be composed of different groups for different declared symbols in a declaration. In one group, the primitive **symbol** must appear, others are optional. The omitting of the primitive **scope** means that the usual scope is specified.

For example, we define the following two different structure patterns for function declarations $f : A\rightarrow B$ and $A\xrightarrow{f}B$.

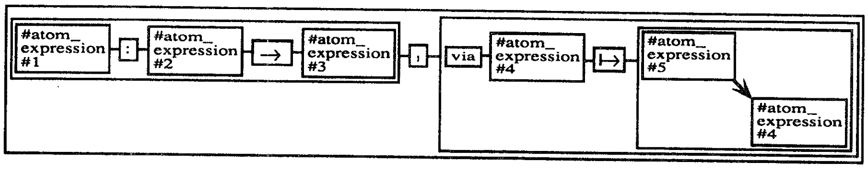


Fig. 7 A structure pattern for a vector and its indexed element.

Structure pattern 1 :

$$\boxed{\#\text{ atom\_expression }\#\ 1} : \boxed{\#\text{ atom\_expression }\#\ 2} \longrightarrow \boxed{\#\text{ atom\_expression }\#\ 3}$$

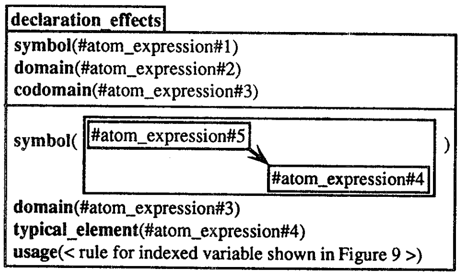
Structure pattern 2 :

$$\boxed{\#\text{ atom\_expression }\#\ 2} \xrightarrow{\boxed{\#\text{ atom\_expression }\#\ 1}} \boxed{\#\text{ atom\_expression }\#\ 3}$$

The two structure patterns correspond to the same declaration effect as follows :

symbol (# atom_expression # 1)
domain (# atom_expression # 2)
codomain (# atom_expression # 3)

These primitives memorize the function name, its domain and codomain for succeed understanding. Obviously, the user can define new structure patterns whithout modifying the parser, so the user (not only the programmer) can extend different representations easily. Therefore, the immediate extensibility of math notation can be achieved.



declaration_effects

symbol(#atom_expression#1)
domain(#atom_expression#2)
codomain(#atom_expression#3)

symbol(#atom_expression#5 #atom_expression#4)
domain(#atom_expression#3)
typical_element(#atom_expression#4)
usage(< rule for indexed variable shown in Figure 9 >)

Fig. 8 Declaration effects for a vector and its indexed element.

Another example shows how to process the mapping structure with usage to describe math expressions such as the following declaration for a real vector and its indexed element. "$\boldsymbol{a}$: $\boldsymbol{n}\rightarrow \boldsymbol{R}$, via $i \mapsto a_i$". Its structure pattern is defined in **Fig. 7**. It corresponds to the following two groups of declaration effects in **Fig. 8**.

According to these primitives, the parser memorizes the formal representation of the two declared symbols, their domains and codomain, and creates a connection between the declared symbols and the usage, and then, understands the vector's indexed variable, such as a_1, according to the rule in the usage shown in **Fig. 9**.

Owing to use declaration effects, the parser only knows the formal representation and the primitives in declaration effects. Therefore, the formal representation and the primitives can not be modified and extended by the user. The parser does not know any function names, domain names, domain relations, category names, and fixed math symbols. Therefore, the user can extend and specify all of them by creating new rules. Of course, a friendly interface for user extension needs to be developed in

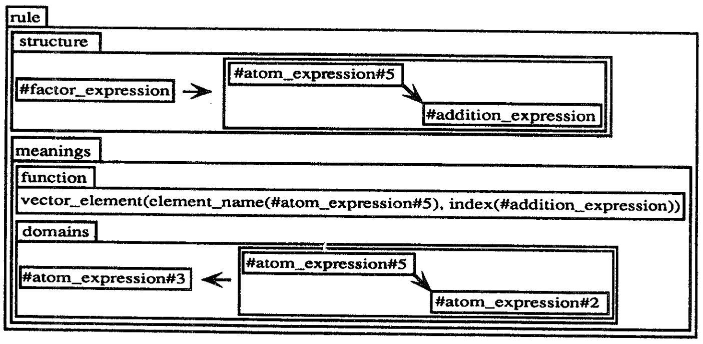


Fig. 9 A rule for usage of indexed variable.

the future. This interface supports a two-dimensional knowledge representation language which is revealed in an implementation example in the next chapter.

6. Parser and Implementation

A parser has been implemented to accomplish top-down structure analysis and bottom-up meaning interpretation based on the knowledge base mentioned above, which is stated briefly as follows :

Data Structure :

An object list used to memorize every declared object's meaning, i.e., its formal representation, domain, codomain, usage, scope, etc.

Algorithm :

Input : *formal_representation*

Outout : its *meta-representation*

Step 1. If there are mathematical objects memorized in the object list, make a match between the *formal_representation* and the object in the object list one by one ; if matched, obtain a *function* and a *domain* of the object as output, return with "successful".

Step 2. Pick up rules from knowledge base one by one, to find a rule such that the structure pattern in this rule matches with the *formal_representation.* If no rule can be found in the knowledge base, make the parser exit with "unknown".

Step 3. If the rule has declaration effects, execute their primitives to memorize declaration information into the object list, and determine the analysis order of sub-structures.

Step 4. Understand all sub-structures recursively using this algorithm. After executing this algorithm for each sub-structure, the *function* and *domain* of the sub-structure are obtained. If one of them failed, go to Step 2.

Step 5. Pick up domain rules one by one from the meanings list of the rule, to find a domain rule whose domain pattern matches with the domain pattern obtained by all sub-structures. If both are matched, the function in current meanings and the codomain in the domain rule are composed of the output. Then, return with "successful". If no domain rule can be found in the rule, goto Step 2.

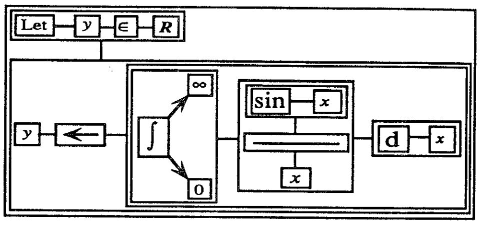


Fig. 10 Formal representation of implementation example.

In this algorithm, we are not concerned with the problem of ambiguities. It is not difficult to modify this algorithm to be able to find ambiguities in a math expression. A math expression is **ambiguous**, if it leads to more than one [*function, domain*] after parsing. We will need another study to handle the problem of ambiguities of math expressions.

The knowledge base and the parser have been implemented in CESP[51] (an object-oriented PROLOG). An implementation example is given as follows.

Let $y \in \boldsymbol{R}$, $y \leftarrow \int_0^{\infty} \frac{\sin x}{x} dx$, where, $\boldsymbol{R}$ is a set of real numbers, and $\leftarrow$ means assignment. This problem can be input as a 2D formal representation shown in **Fig. 10** by a two-dimensional editor (remains to be implemented, which is similar to MathType[26]).

In our implementation, this example is input as a 1D formal representation, which is equivalent to the definition of the 2D formal representation, shown below.

```
[column,
  [row, '\rm Let', 'y', '\in', '\boldmath R'],
  [row, 'y', '\leftarrow',
    [row, [tree, '\int', back_sup, '\infty',
    back_sub, '0'],
          [column, [row, '\sin', 'x'], '\over',
          'x'],
          [row, '\rm d', 'x']
    ]
  ]
]
```

The integral part within this example can be understood according to the following rule in the knowledge base in **Fig. 11**, which is represented in a two-dimensional knowledge representation language (remains to be developed

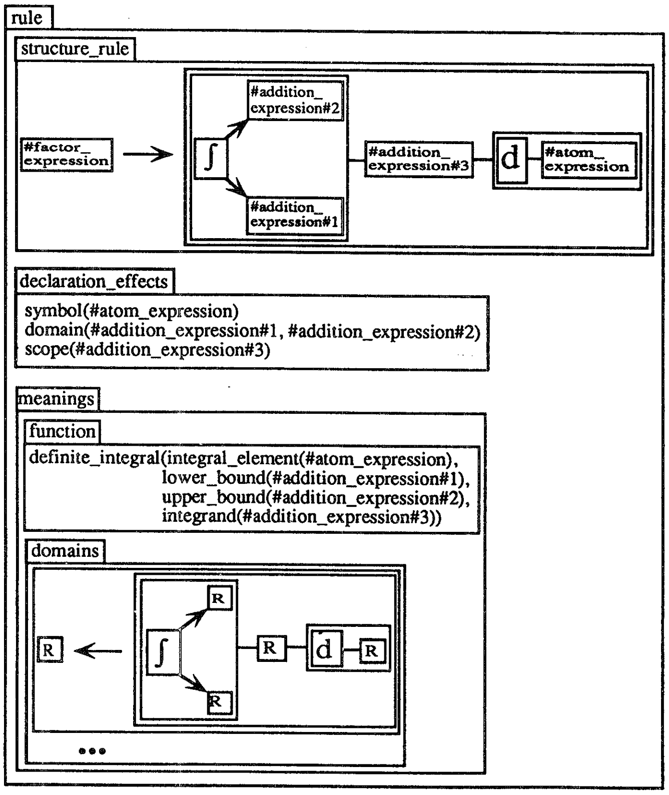

rule
structure_rule
#factor_ expression
#addition_ expression#2
#addition_ expression#1
#addition_ expression#3
d
#atom_ expression
declaration_effects
symbol(#atom_expression)
domain(#addition_expression#1, #addition_expression#2)
scope(#addition_expression#3)
meanings
function
definite_integral(integral_element(#atom_expression), lower_bound(#addition_expression#1), upper_bound(#addition_expression#2), integrand(#addition_expression#3))
domains
R
R
R
R
d
R
•••

Fig. 11 A rule for understanding definite integral expression.

in future).

Finally, the parser translates the 1D formal representation into the meta-representation as follows in a list form: [[⟨function_name⟩ , ⟨meta-representation⟩ ,⋯], ⟨domain_name⟩]. Here, R is the abbreviation of the output set_of_real_numbers in this example; and N means set_of_natural_numbers.

```
[[computation,
    [[declare, [variable, y], [domain, R]], void],
    [[assignment,
        [[assigned_variable, [[declared_variable, y], R]], R],
        [[assign_expression,
          [[definite_integral,
              [[integral_element, x], R],
              [[lower_bound, [[natural_number, 0], N]], N],
              [[upper_bound, [infinity, R]], R],
              [[integrand, [[fraction,
                  [[numerator, [[sin, [[declared_variable, x], R]], R]], R],
                  [[denominator, [[declared_variable, x], R]], R]
              ], R]], R]
          ], R]
        ], R]
    ], void]
  ], void
]
```

Up to the present, the implemented knowledge base and the parser have had the ability to understand the following kinds of formulas: 1) variable declarations, 2) fixed constants, 3) four arithmetic operations which contains the omitted or invisible multiplication operator between two factors, 4) elementary functions, 5) indefinite and definite integrals, 6) summation, 7) function declaration, definition, and function-call, 8) vector declaration and indexed variable, 9) factorial, 10) fraction, 11) relation =, etc. It has been shown that our knowledge-based method is verified to be correct and effective. We also plan to accomplish an application to produce and verify a library which contains the printing images and the meanings of almost all mathematical formulas.

Our knowledge-based method is an extension of context-free grammar parsing. This method is different from other extensions and parsers because of the different language it processes. Our method has the following two distinctive mechanisms for processing math notation merely: 1) declaration effects for describing semantic structures and satisfying immediate extensibility, and 2) the mechanism such that the meanings of objects are determined by declaration at first.

7. System

Based on the research of this paper, we put forward a scheme for a system to integrate document processing with document understanding of math notation in **Fig. 12**.

Here, the **Interface for User Extension** provides the user with a way to create and extend the knowledge base through the two-dimensional knowledge representation language shown in Fig. 11.

The **Post-processors** are application systems. If the application is mathematical computation, the post-processor contains the following three processings: **1) Selection of Data Types:** The domain of a variable in mathematics has to correspond to one of several data types, e.g., we may select one among float type, double float type, arbitrary float type or symbol type to represent the set of real numbers $\boldsymbol{R}$. **2) Selection of Algorithms:** An operation in mathematics can correspond to many algorithms, e.g., for computing $\int_a^b f(x)\,dx$, we have to select the most suitable algorithm from many. **3) Translation into programs:** The meta-representation whose data types and algorithms have been

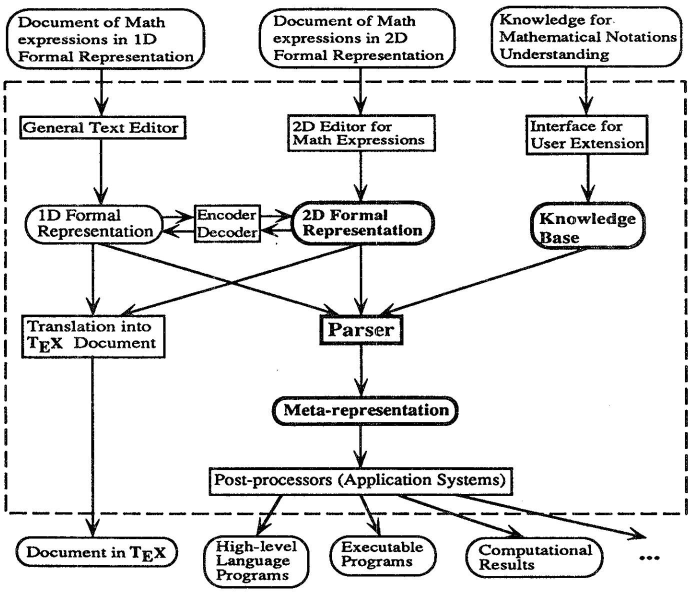

Document of Math expressions in 1D Formal Representation
Document of Math expressions in 2D Formal Representation
Knowledge for Mathematical Notations Understanding
General Text Editor
2D Editor for Math Expressions
Interface for User Extension
1D Formal Representation
Encoder
Decoder
2D Formal Representation
Knowledge Base
Translation into TEX Document
Parser
Meta-representation
Post-processors (Application Systems)
Document in TEX
High-level Language Programs
Executable Programs
Computational Results
...

Fig. 12 A system for integrating document processing with document understanding.

determined is translated into high-level language programs, or the inputs of existing application systems directly. For example, we can translate the meta-representation of the example shown in Fig. 10 into a statement of Mathematica[50] like the following

y=Integrate[Sin[x]/x,{x, 0, Infinity}]

because Mathematica is a language of functions and operations.

Up to the present, we have only implemented the kernel of this system, i.e., the parser and the knowledge base. Therefore, we believe that it is valuable and necessary to take plenty of time to implement the other parts of this system, which will contain at least one post-processor in the future.

8. Conclusion

We began with a research in mathematical notation and their applications. The fundamental concepts and formalization were considered at first. Based on them, we explored the principles of structure and meaning of mathematical notation, and achieved a knowledge-based method for understanding mathematical notation. This method has the following features: 1) independence of different applications, 2) knowledge base extensibility by user, 3) close connection of syntactic analysis and semantic interpretation, and 4) brevity of the knowledge representation. The perfect results and applications still require much research (e.g., ambiguities, ellipses, etc.) in the future.

Acknowledgements We express much gratitude to Prof. Taketomo Mitsui (Nagoya Univ.) who contributed much advice, provided many important references, and improved the writing style of this paper. We also express grateful thanks to Visiting Prof. Libo Lo (Univ. of Michigan) for his discussion and advice particularly in metamathematics. We would also like to acknowledge Mr. Osamu Takata (Toyota Central R&D Labs.) for his advice and help in the implementation.

References

1) Cajori, F.: *A History of Mathematical Notations*, Vol. 1 & 2, The Open Court Publishing (1928-1929).

2) Kline, M.: *Mathematical Thought, from Ancient to Modern Times*, Oxford University Press (1972).

3) Sammet, J. E.: *Programming Languages: History and Fundamentals*, Prentice-Hall (1969).

4) Wells, M. B.: MADCAP: a Scientific Compiler for a Displayed Formula Textbook Language, *Comm. ACM*, Vol. 4, No. 1, pp. 31-36 (1961).

5) Balke, K. G. and Carter, G. L.: The COLASL Automatic Coding Language, *Symbolic Languages in Data Processing*, pp. 501-537, Gordon and Breach (1962).

6) Iverson, K. E.: *A Programming Language*, John Wiley & Sons (1962).

7) Gawlik, H. J.: MIRFAC: a Compiler Based on Standard Mathematical Notation and Plain English, *Comm. ACM*, Vol. 6, No. 9, pp. 545-547 (1963).

8) Klerer, M. and May, J.: An Experiment in a User Oriented Computer System, *Comm. ACM*, Vol. 7, No. 5, pp. 290-294 (1964).

9) Klerer, M.: Interactive Programming and Automated Mathematics, Klerer, M. and Reinfelds, J. (ed.): *Interactive Systems for Experimental Applied Mathematics* (Proceedings of the ACM, Symposium Washington, U.S.A., Aug. 1967), pp. 3-10, Academic Press (1968).

10) Klerer, M., Grossman, F. and Amann, C. H.: Design Philosophy for an Interactive Keyboard Terminal, Klerer, M. and Reinfelds, J. (ed.): *Interactive Systems for Experimental Applied Mathematics* (Proceedings of the ACM, Symposium Washington, U.S.A., Aug. 1967), pp. 183-191, Academic Press (1968).

11) Klerer, M.: Experimental Study of a Two-dimensional Language vs Fortran for First-course Programmers, *Int. J. Man-Mach. Stud.*, Vol. 20, pp. 445-467 (1984).

12) Klerer, M., Grossman, F. and Klerer, R.: The Automated Programmer System: Language Design Issues for Scientific-Mathematical-Engineering Applications Programming, Boudreaux, J. C., Hamill, B. W. and Jernigan, R. (ed.): *The Role of Language in Problem Solving 2*, pp. 245-260, Elsevier Science Publishers (1987).

13) Klerer, M.: *User-Oriented Computer Languages, Analysis & Design*, Macmillan Publishing (1987).

14) Anderson, R. H.: Syntax-Directed Recognition of Hand-printed Two-Dimensional Mathematics, Klerer, M. and Reinfelds, J. (ed.): *Interactive Systems for Experimental Applied Mathematics*, (Proceedings of the ACM, Symposium Washington, U.S.A., Aug. 1967), pp. 436-459, Academic

Press (1968).
15) Fu, K. S.: *Syntactic Methods in Pattern Recognition*, pp. 245-252, Academic Press (1974).
16) Anderson, R. H.: Two-Dimensional Mathematical Notation, Fu, K. S. (ed.): *Syntactic Pattern Recognition, Application*, pp. 147-177, Springer-Verlag (1977).
17) Chang, S. K.: A Method for the Structural Analysis of Two-Dimensional Mathematical Expressions, *Inf. Sci.*, Vol. 2, pp. 253-272 (1970).
18) Wang, Z. X. and Faure, C.: Structural Analysis of Handwritten Mathematical Expressions, *Proceedings of the 9th International Conference on Pattern Recognition*, Nov. 14-17, 1988, Rome, Italy, Vol. 1, pp. 32-34, IEEE Computer Society Press (1988).
19) Nakayama, Y.: A Prototype Pen-input Mathematical Formula Editor, Maurer, H. (ed.): *Educational Multimedia and Hypermedia Annual* (Proceedings of ED-MEDIA93 World Conference on Educational Multimedia and Hypermedia, Orlando, Florida, U.S.A., Jun. 23-26, 1993), pp. 400-407 (1993).
20) Kernighan, B. W. and Cherry, L. L.: A System for Typesetting Mathematics, *Comm. ACM*, Vol. 18, No. 3, pp. 151-157 (1975).
21) Knuth, D. E.: *The TEX Book (Computer & Typesetting/A)*, Addison Wesley (1984).
22) Schär, H.: Die Integration mathematischer Formeln in den Dokumenteneditor Lara, *Informationstechnik it*, 28 Jahrgang, Heft 6, pp. 352-360 (1986).
23) Crisanti, E., Formigoni, A. and Bruna, P. L.: Easy TEX: Towards Interactive Formulae Input for Scientific Documents Input with TEX, Dêsarmênien, J. (ed.): *Lecture Notes in Computer Science, TEX for Scientific Documentation* (2nd. European Conference, Strasbourg, France Jun. 19-21, 1986 Proceedings), pp. 55-64, Springer-Verlag (1986).
24) Holdam, J. and Nϕrgaard, C.: Gipsy: a Grammar Based Interactive Document Processing System, Hewson, R. (ed.): *Proceedings of 3rd. International Conference on Text Processing Systems*, Oct. 22-24, 1986, Dublin, Ireland, pp. 141-147 (1986).
25) Arnon, D., Beach, R., Mcisaac, K. and Waldspurger, C.: CaminoReal: An Interactive Mathematical Notebook, Vliet, J. C. van (ed.): *Proceedings of the International Conference on Electronic Publishing, Document Manipulation, and Typography*, Apr. 20-22, 1988, Nice, France, pp. 1-18, Cambridge University Press (1988).
26) Design Science Inc.: *MathType 2.0 Mathematical Equation Editor User Manual* (1989).
27) ISO 31/XI: Mathematical Signs and Symbols for Use in the Physical Science and Technology (1978).
28) Spivey, J. M.: *Understanding Z, A Specification Language and Its Formal Semantics*, Cambridge University Press (1988).
29) Spivey, J. M.: *The Z Notation, a Reference Manual*, Prentice Hall International (1992).
30) Norcliffe, A.: The Wider Uses of the Z Specification Language in Mathematical Modelling, Johnson, J. H. and Loomes, M. J. (ed.): *The Mathematical Revolution Inspired by Computing* (Proceedings of a Conference in the Institute of Mathematics and its Applications, Brighton Polytechnic, Apr. 1989), pp. 145-155, Oxford University Press (1991).
31) Boute, R. T.: Syntactic and Semantic Aspects of Formal System Description, *Microprocessing and Microprogramming*, Vol. 27, pp. 155-162 (1989).
32) Boute, R. T.: Funmath: Towards a General Formalism for System Description in Engineering Applications, Silvester, P. P. (ed.): Advances in Electrical Engineering Software, pp. 215-226, Springer-Verlag (1990).
33) Soiffer, N. M. and Smith, C. J.: MathScribe: A User Interface for Computer Algebra Systems, Char, B. W. (ed.): *Proceedings of the ACM 1986 Symposium on Symbolic and Algebraic Computation*, pp. 7-12, Jul. 21-23, 1986, Waterloo, Ontario, Canada (1986).
34) Soiffer, N. M.: The Design of a User Interface for Computer Algebra Systems, Ph. D. Thesis, University of California, Berkeley, Report No. UCB/CSD 91/626 Apr. 1991 (1991).
35) Donnelly, D.: *MathCAD for Introductory Physics*, MathSoft Inc. Cambridge MA. U.S.A., Addison-Wesley (1992).
36) *MathStation*, Ver.1.0, MathSoft, Inc., One Kendall Square, Cambridge, MA, 02139 U.S.A. (1989).
37) Soft Waterhouse, Inc.: *DERIVE User Manual 3rd. Ed.* (1989).
38) Young, D. A. and Wang, P. S.: GI/S: A Graphical User Interface for Symbolic Computation Systems, *J. Symbolic Computation*, Vol. 4, No. 3, pp. 365-380 (1987).
39) Wang, P. S.: Integrating Symbolic, Numeric, and Graphics Computing Techniques, *Mathematical Aspects of Scientific Software, The IMA Volumes in Mathematics and Its Applications*, Vol. 14, pp. 197-208, Springer-Verlag (1988).
40) Doleh, Y. and Wang, P. S.: SUI: a System

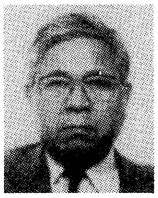
Independent User Interface for an Integrated Scientific Computing Environment, *ACM Proceedings of the International Symposium on Symbolic and Algebraic Computation*, Tokyo, Japan, Aug. 20-24, 1990, pp. 88-95, Addison-Wesley (1990).
41) Avitzur, R.: *Milo*, Paracomp Inc., San Francisco, U.S.A. (1988).
42) Bonadio, A. and others: *Theorist Reference Manual*, Prescience Corporation (1990).
43) Bonadio, A. and others: *Theorist Learning Guide*, Prescience Corporation (1990).
44) Kajler, N.: CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems, *Proceedings of ISSAC'92*, Berkeley, U.S.A., Jul. 1992, pp. 376-386 (1992).
45) Kajler, N.: Environnement graphique distribué pour le Calcul Formel, Thèse, Docteur en Sciences, Université de Nice-Sophia Antipolis, Mars 1993(1993).
46) White, J. and others: *MathKit: Interactive Texts for Math and Science, An Introduction*, Institute for Academic Technology, University of North Carolina at Chapel Hill (1993).
47) Crystal, D.: *The Cambridge Encyclopedia of Language*, Cambridge University Press (1987).
48) Kleene, S. C.: *Introduction to Metamathematics*, North-Holland Publishing (1952).
49) Feys, R. and Fitch, F. B. (ed.): *Dictionary of Symbols of Mathematical Logic*, North-Holland Publishing (1969).
50) Wolfram, S.: *Mathematica, a System for Doing Mathematics by Computer*, Addison-Wesley (1991).
51) CESP Ver.A00 by AI Languages Research Institute, Mitsubishi-Denki, 5-1-1 Ofuna, Kamakura, Kanagawa, 247 Japan (1994).

(Received October 25, 1993)
(Accepted June 20, 1994)

Yanjie Zhao, born in Beijing, China, in 1957, graduated from Computer Science Department of Nanjing University, China, in 1982 with the B.S. degree. He became an assistant since 1982 and a lecturer since 1987 in Radio-Electronics Department of Beijing Normal University, China. He also had the M.S. degree in 1989 in Radio-Electronics Department of Beijing Normal University, China. After expired from the Doctor program from 1991 to 1994 in Depertment of Information Engineering, Nagoya University, Japan, he became an assistant in Intelligence Engineering Laboratory of Kurume Institute of Technology, Japan. His research interests are in Mathematical Notations Understanding and Manipulating, Language Processing, and Document Processing. He is a member of IPSJ and SIAMJ.

Hiroshi Sugiura is an associate professor of Department of Information Engineering, School of Engineering, Nagoya University. He is interested in numerical integration, numerical method for ODE's and integro-differential equations.

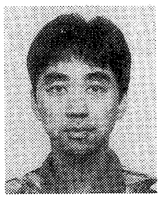
He was born in Mie profecture in 1952. He received his B.S. and M.S. degrees from Department of Mathematics, School of Science Nagoya University. He completed his doctoral course at Department of Information Engineering, School of Engineering, Nagoya University in 1981. Since then, he was with the Department as a research assistant and received D.E. degree from Nagoya University in 1991. And he was promoted to an associate professor in 1992. He is a member of IPSJ.



Tatsuo Torii is a professor of Department of Information Engineering, School of Engineering, Nagoya University. He has been engaged in research about numerical analysis and mathematical software. His special subjects are function approximation and numerical integration based on the FFT technique.

He was born in Kumamoto prefecture in 1934. He graduated from Department of Electricity, Kyushu Institute of Technology in 1957. After his 7 years career being with Shin Nihon Chisso Corp., he joined Department of Applied Physics, Faculty of Engineering, Osaka University as a research assistant in 1964.

He received D.E. from Osaka University in 1972. In 1975, he joined Department of Information Engineering, School of Engineering, Nagoya University as an associate professor. He was promoted to an assistant professor in 1976, and to a professor in 1985. He is a member of IPSJ.



Tetsuya Sakurai, born in 1961, graduated from Department of Information Engineering of Nagoya University in 1986, and also had the D.E. degree in Department of Information Engineering of Nagoya University in 1992. He became an assistant since 1986 in Department of Information Engineering of Nayoya University and lacturer since 1993 in Institute of Information Sciences and Electronics of University of Tsukuba. His research interests are in Root Finding Methods for Nonlinear Equations, Application of Rational Approximation, and Mathematical Software. He is a member of IPSJ and SIAMJ.