数式処理 *J.JSSAC (1998)*
Vol. 6, No. 3, pp. 2 - 29

論文

Formalization and Parsing of Mathematical Expressions for Mathematical Computatio

Yanjie ZHAO Tetsuya SAKURAI

Nagasaki Institute of Applied Science University of Tsukuba

Hiroshi SUGIURA and Tatsuo TORII*

Nagoya University

(RECEIVED 1997/7/22 REVISED 1997/12/17)

Abstract. Mathematical expression parsing is very important particularly for the human interface of numeric and algebraic computation systems. In this paper, we define a formalization method and a denotational meaning representation of mathematical expressions. We create grammars and their parser to translate the formalized mathematical expressions into their meaning representation. We also develop a meta-language based system to realize the notation extension. Thus many complicated context-sensitive notations can be parsed, and a lot of ambiguities can also be avoided.

1 Introduction

Since the 1960s, many studies on the input and parsing of mathematical expressions have been made in separate fields: programming languages, document pattern recognition, document preparation, software specification, numeric and algebraic computation systems. 132 references were listed in [9] and 51 in [20]. In these achievements, there are at least the following unsolved problems: 1) grammars for whole mathematical expressions are lacking, 2) the existing parsings of mathematical expressions are dependent on their specified front-end languages, 3) many complicated context-sensitive mathematical notations have not been parsed, 4) the ambiguity problem has not been avoided, 5) notation extension is still very difficult, and 6) the high quality display of mathematical expressions during editing is still lacking.

All existing developments have outgrown their front-end languages to be close to mathematical notation. On the contrary, to resolve the problems mentioned above, we have not designed another similar one, but have started from and focussed on the study of whole mathematical notation at first, and then, develop various applications in diverse areas.

*Present address : Nanzan University

Precisely, we have proposed a box notation system to formalize mathematical expressions [20], have implemented a knowledge-based method to parse the formalized mathematical expressions into a denotational meaning representation, and then into front-end languages of general purpose computer algebra systems (*Mathematica* and *Maple*) to verify the parsing, and had established two prototypes to input mathematical expressions in a high quality display [21][13].

After the references listed in [20], there have been new developments: **1) CAS/PI** [6][7][8], a formula editor and a common graphical user interface of concrete computer algebra systems, and its extensibility through writing its inner list representation of grammar and through plugging software components, **2) MathSpad** [3], a 2D formula editor with a TEX's METAFONT display and an input connection to *Maple*, **3) MathProbe** [17], a hyperdocument of a mathematical dictionary which allows retrieval and editing of very limited 2D formulas and a connection to *Maple*, **4) Oberon Interface** [18], an editor based on Oberon OS with step-by-step formula conversion between 1D and 2D form for both documentation and computation by a connection to *Maple*, **5) OCR input** of the image of published mathematical literature [1], **6) SPEC** [12], a system with a simulated mathematical expression editing for calculating integrals, **7) Scientific WorkPlace** [15], a WYSIWYG document and formula editor with high quality display and a connection with *Maple*, **8) GLEF$_{\mathbf{ATINF}}$** [2], an interface of mathematical inference laboratory *ATINF* for combining inference tools, formula editing, and proof checking into one environment, **9) Mathematica** version 3, [14], which has been equipped with a 2D, programmable, and extensible formula editing interface with high quality display (its fonts are suitable to lower resolution display), **10) Maple** V Release 4 [11], which has an interface of textual parsing input and a 2D high quality of display, and a hyperlink between its worksheets, and **11) Developments in Printed Formula Pattern Recognition** [4][16][5][10], which have achieved the recognition of many mathematical expressions. In addition, many projects are developing based on *Internet*.

All these new contributions have resolved part of the problems mentioned above, i.e., the 1st has been unsolved; the 2nd has been resolved in one way (software bus) in *CAS/PI*; the 3rd has been resolved to a certain extent, e.g., in *Scientific WorkPlace*; the 4th has been unsolved; the 5th has been resolved to a certain extent, e.g., in *CAS/PI* and *Mathematica* Version 3; the 6th has been greatly resolved in *Scientific WorkPlace* and *Mathematica* Version 3. Compared with these new developments in the world mentioned above, our research at the same period still has the following independent features and contributions:

1. Establishment of a syntactic structure and a semantic representation of mathematical notation based on a formalization (to resolve the 1st problem).
2. Definition of a formalization method or a precise model of the structure of mathematical notation to abstract the non-verbal expressions, which is independent of input

methods, and moreover, definition of meaning representation to preserve the meanings of mathematical expressions, which is independent of algorithmic definitions and programming languages (to resolve the 2nd problem).
3. Establishment of powerful grammars to parse much more complicated context-sensitive notations (to resolve the 3rd problem).
4. Clarification of the necessities for the parsing of many grammars, and moreover, development of a knowledge-based parsing method to avoid the ambiguities (to solve the 4th problem).
5. Development of a meta-language to express all the grammars of mathematical notation and to make the grammars user-oriented, and moreover, design of the notation extension mechanisms (to resolve the 5th problem).

In this paper, based on our recent work in [22][23], we expound the parsing part of our research deeper, more completely, and systematically. The input interface of our research was given in [21][13]. We narrate all the contents in the following sections: the topics about the formalization in Section 2, the parsing method in Section 3, the meta-language as well as notation extension in Section 4, and the prototyping in Section 5.

2 Formalization

In order to process mathematical notation on computers, it is necessary to define a precise model or a formalization method for mathematical notation at first. Through this model or method, non-verbal mathematical expressions can be abstracted into verbal and formalized ones. This model is defined as follows.

A formal expression (expression, in short) is a morpheme, or a structure of morphemes. Every expression occupies one rectangle of space denoted by a box. A morpheme is put into a box called atomic box. Every box is located in a structure of boxes. The **formal representation** (FR for short) is defined in Figure 1 (where "expr" for the short of "expression"). **Formalization** means to use boxes to formalize a mathematical expression to obtain its FR.

In Figure 1, the two kinds of undirected line mean horizontal and vertical concatenates. The six kinds of directed arrows mean super-script and sub-script subordinate concatenates. The duet box means the repetition of horizontal or vertical concatenates, or empty. The dashed box means an optional box. The separated "|" means alternation. A morpheme is a string of any characters from natural languages and mathematical signs.

The necessities of the formalization can be summarized as follows:

1. FR can be used to express the universal set of the formalized mathematical expressions.
2. FR can be used to abstract any non-verbal symbol into its verbal one. For example,

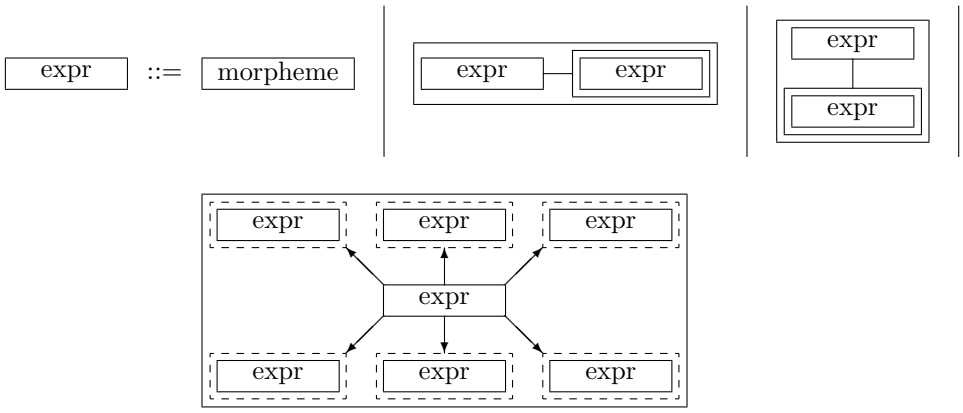
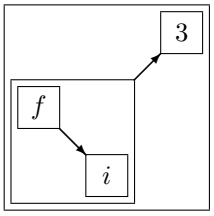


Figure 1: Definition of formal representation

fraction strokes being different in length become same formal stroke; open-close symbols, such as |, (, etc, being different in height become same formal one respectively; square root symbols $\sqrt{<\text{expr}>}$ being different in both length and style become same formal one like as `[column, '\sqrt', <expr>]`; etc.

3. FR is an abstract model for expressing the following **typesetting tacit agreement**s explicitly and normally. No matter what methods one uses, such as pen and paper, printing, computer hand-written input, computer image scanning input, or computer key-mouse input, one has to obey the two-dimensional row, column, and tree structures on a base line. In addition, there are also some rules for line-breaking, page-breaking, etc, which need another study. All these things have become tacit agreements, called **typesetting tacit agreement**s. FR is independent of input methods and the media. It is a fundamental frame for building parsing patterns and input templates of mathematical expressions. FR is also independent of typesetting details, such as the font produced by different makers, font size (e.g., the two x's in x^x are the same), tiny differences in location and gap of symbols (e.g., between $+$ and b in $a+b$ versus between $-$ and b in $-b$), different space and skip, etc. These typesetting details contribute nothing to the parsing.
4. FR can be used to express all the following **determinable tacit agreement**s and **indeterminable tacit agreement**s. To omit symbols, some tacit agreements about priority and associative rule have been widely accepted. For the example of the priority, we can find that any **factor** is at the top priority, such as $n!$, $\lceil x \rceil$, $\mid x \mid$, e^x, $\sin x$, $\int f(x)\mathrm{d}x$, $\sum_{r=0}^{n} f(r)$, $a_{\lceil \frac{n}{2} \rceil}$, etc. The second priority is a **term**, such as $\mathbf{a} \times \mathbf{b}$, $\mathbf{a} \cdot \mathbf{b}$, x/y, $A \cap B$, etc. The third one is called **calculation**, such as $x+y$, $x-y$, $A \cup B$, $-b$, etc. For example, $a+b \times c$ means $a+(b \times c)$ in the most situations. For the example of the associative rule, we can find that some operations satisfy both left and right associative rules, some others only the left associative rule (e.g., $a-b-c=(a-b)-c$).

the correct form: 

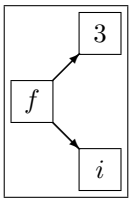
the incorrect form: 

Figure 2: The formal representation of f_i^3 for the meaning $(f_i)^3$

These priorities are determinable or definite in a context, in spite of their different kinds (e.g., logical operators are prior to relation operators in one priority, and posterior to in another), which is called **determinable tacit agreement**s. To omit much more symbols without causing ambiguities, other tacit agreements concerning priority, which is called **indeterminable tacit agreement**s, have been accepted to a certain extent. For example, $\sum_{k=1}^{\infty} a^k \sin k\pi s + a$ 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 are indeterminable. For example, what does the expression $\sin x\sqrt{ax^3 + bx^2 + cx + d}$ means? $(\sin x)\sqrt{ax^3 + bx^2 + cx + d}$ or $\sin(x\sqrt{ax^3 + bx^2 + cx + d})$? This may be dependent on different conventions, contexts, fields, and personal habits. To these two kinds of tacit agreements, FR can be used to express them explicitly. For example, to parse the expression $a + b \times c$, we can simply input it as $a + \boxed{b \times c}$. Similarly, $\sin(n + 1)x$ can be input as $\sin \boxed{(n + 1)x}$ or $\boxed{\sin(n + 1)}x$. Of course, clear design of a grammar and elaborate introduction of its grammatical categories can also realize the description of these tacit agreements in principle. For example, to parse an expression like $a + b \times c$, we can establish rules like as <calculation> ::= <calculation> + <term> and <term> ::= <term> × <factor> ... However the indeterminable situations are not so explicit. Nevertheless, in representation of these tacit agreements, the more boxes we use, the less grammar rules and categories we need; vice verse (see Section 3.4). On the one hand, the **formalization** can be naturally manipulated (at least by key-mouse) by people who know how to construct mathematical expressions on computers, but there are many boxes, and the FR for a same expression also becomes not unique. On the other hand, certain grammars may not be accepted widely, and are difficult to modify and extend.

5. FR can also be used to distinguish some inherent ambiguities. For a simple example, what does the f_i^3 mean? $(f_i)^3$, $(f^3)_i$, or (f_i^3) ? Without a context-sensitive restriction, we can not obtain any answer. However, if f_i^3 means $(f_i)^3$, its FR which is shown in Figure 2 can avoid this kind of problems. For a typical example, what does the $|a|b+c|d|$

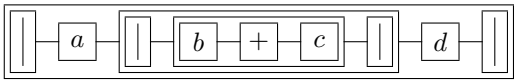


Figure 3: The formal representation of $|a|b+c|d|$ for the meaning $|a \times (|b+c|) \times d|$

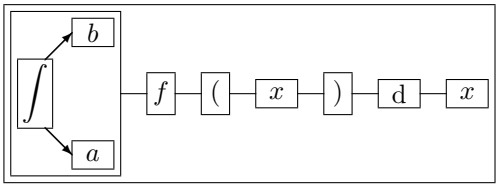


Figure 4: The formal representation of $\int_a^b f(x)\mathrm{d}x$

mean? $(|a| \times b) + (c \times |d|)$, or $|a \times (|b+c|) \times d|$? This is an inherent ambiguity. However, if it means $|a \times (|b+c|) \times d|$, its FR which is shown in Figure 3 can solve this kind of problems, and its corresponding printing image $\left|a|b+c|d\right|$ is unambiguous.

6. FR can be used as the meta-notation of a meta-language to describe grammars of mathematical notation, because Backus-Naur Form (BNF in short) is inconvenient and even impossible to define the grammars of mathematical notation. BNF can not explicitly express the column and tree concatenates, some tacit agreements, and non-verbal symbols, if additional meta-symbols, which are equivalent to FR, are not introduced.

Therefore, in comparison to the input and parsing of natural languages and programming languages, the formalization is not only necessary, but also as important as the grammar. A certain formalization and a certain grammar determine a parsing method (see Section 3.4). From this point of view, enough communication (not only mathematical symbols, but also some tacit agreements) is necessary for human to manipulate and process mathematical notation on computers. Formalization is not a special method to aid the input and parsing of mathematical expressions. It is widespread and inevitable.

As an example, the FR of $\int_a^b f(x)\mathrm{d}x$ is shown in Figure 4. It is not a WYSIWYG input image, but an abstract or inner data representation of the corresponding input image. It is in fact stored in computers for display and parsing in the following list form:

```
[row,[tree,'\int',back_sup,'b',back_sub,'a'],'f','\(','x','\)','{@rm d}','x']
```

Its WYSIWYG input and editing by key-mouse can been found in [21][13].

3 Parsing Method

3.1 Meaning Representation

The meaning of a mathematical expression can at least be dependent on its denotational interpretation, i.e., a fixed individual, operator or auxiliary symbol has its verbal name or denotation (a value, constant name, function name, etc.); a variable, self-defined function, self-defined function call, or polynomial has its declared domain, based on mathematical knowledge. This kind of interpretation can become a static denotational semantics of mathematical expressions, and can be expressed by functions with their codomains.

In this research, our goal is only the translation between mathematical expressions and programs in order to simulate computation meanings of mathematical expressions through the execution of corresponding programs. Therefore, in a utilitarian stand, this static denotational semantics is enough. In a theoretic stand, this semantics can be regarded as an elementary result to approach to the semantics of mathematical notation, which needs further research.

Thus, we define a textual functional meaning representation, called **meta-representation** (MR for short), to describe this kind of static denotational semantics. MR is an object that contains a function name, its codomain name, and all sub-MR, written in verbal words.

$$<\text{MR}> ::= f(<\text{MR}>, ..., <\text{MR}>),\ d \ \mid \ n,\ d$$

where f means a function name; d means a domain; n means an individual name.

MR is a symbolic meaning representation of mathematical expressions on computers. Thus it can be used to express not only computable but also uncomputable meanings. It is independent of programming languages and algorithms, and can be extended and modified by the end user in principle. It includes only the name of functions, but does not give the definitions of functions. It is necessary to translate it into a program written in a programming language for a further interpretation on computers.

MR is also necessary for the following two reasons: 1) Through the MR and a common human interface for the common mathematical expressions, it is possible for users to use more than one front-end language, numerical package, and computer algebra system at the same environment, because one computational problem may be solved incorrectly, incompletely, or inefficiently in one system or language, but may be solved correctly, completely, or efficiently in another. 2) Although many real constants can be expressed by symbols in programming languages, such as π, e, $\sqrt{2}$, $\sin 1$, etc, declarations such as "Let $x \in \mathbf{R}$" can not be completely expressed by any programming language. The mathematical semantics of a real variable or a real number is very complicated. In current research stage, to avoid this problem in MR, only a denotational name to the domain of real numbers is given: `set_of_real_numbers`. During the parsing, its concrete semantics is unknown. In the next stage *computation on computers*, it can be continually interpreted and simulated by a mathematical computation system.

3.2 Grammatical Knowledge Representation

All mathematical symbols can be classified into two kinds: 1) **fixed mathematical symbols**, i.e., fixed individuals, operators and auxiliary symbols, 2) **changeable mathematical symbols**, i.e., variables, self-defined functions, self-defined function calls, and polynomials [20].

The meanings and usages of existing **fixed mathematical symbol**s have been formed historically and have been accepted by the whole world. They are invariant and do not need declarations to say what they are. However, **changeable mathematical symbol**s need declarations to say what they are and what domains they belong to, and they may have different meanings and usages in different contexts.

According to this classification, the **fixed mathematical symbol**s determine the structure pattern of mathematical expressions. Namely, fixed individuals are terminal objects which do not need further analysis. Operators connect their operands to create an operation. Auxiliary symbols are also similar to operators. For example, parentheses, commas and ellipses can construct a structure, e.g., $(a_1, a_2, \cdots, a_n)$, where 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.

Therefore, we can express all the grammatical knowledge of both syntax and domain semantics of mathematical notation in a *frame and rewriting rule* knowledge representation below according to the relation between the structure and its meanings of a mathematical expression [20] (in this paper, the term grammar means both syntax and semantics; the term parsing means both syntactic analysis and semantic interpretation).

A grammar is mainly composed of a **rule base** and its **background knowledge**. The **rule base** is constructed by a series of rules. A rule has the following construction (where a pair of brackets [and] as meta-symbols denote an optional component):

rule{
  structure{ <category> → <structure_pattern> [*action* ...] }
  meanings{
    function{ textual function of meta-representation }
    [
    domains{ <codomain> ← <domain_pattern> [*action* ...];
      <codomain> ← <domain_pattern> [*action* ...];
      ⋮
    }
    ]
  }
  [
  meanings{ ⋯ }

]

⋮

}

Where, <structure_pattern>, <codomain>, and <domain_pattern> are expressed based on FR. <structure_pattern> and every <domain_pattern> have the same structure of boxes in a rule. Every *action* is a procedure to perform a certain context-sensitive processing (being similar to the actions of *augmented transition network grammars* [19]). The whole of **structure** part corresponds to the syntax. "<category> → <structure_pattern>" is called a **structure rule**. The → means "can have the form". "<codomain> ← <domain_pattern>" is called a **domain rule**. The ← means "is mapped from". The **function** part together with one of < codomain >'s composes a MR.

For example, a rule for pattern $\int_{\square}^{\square} \square \mathrm{d} \square$ can be described in a meta-language (see Section 4.1) that is shown as follows (omit all the boxes):

rule{

structure{

$$< \text{factor} > \rightarrow \int_{<\text{calculation2}>}^{<\text{calculation1}>} < \text{term} > \ \textbf{select}\{d, \mathrm{d}\} < \text{atom} >$$

declare_symbol{

symbol{< atom >}

derived_domain{<calculation2>, <calculation1>}

scope{< term >} } }

meanings{

function{integral(< term >, < atom >, < calculation2 >, < calculation1 >)}

domains{

$$\mathbf{R} \ \leftarrow \int_{\mathbf{R}}^{\mathbf{R}} \mathbf{R} \ \textbf{select}\{d, \mathrm{d}\} \ \textbf{void}$$

$$\mathbf{C} \ \leftarrow \int_{\mathbf{C}}^{\mathbf{C}} \mathbf{C} \ \textbf{select}\{d, \mathrm{d}\} \ \textbf{void} \ \} \ \}$$

}

Where the MR within the **function** part is described and defined in advance as the following:

integral(<expression>, <integral_element>, <lower_bound>, <upper_bound>)

All *action*s (see Section 4.1) within this rule are briefly introduced as follows: **declare_symbol**{ <action> ...} is used to declare a symbol by its wrapped *action*s for registration of the symbol itself, symbol's domain, codomain, and scope; **symbol**{<pattern>, ...} is used to declare a symbol and its usage or call form; **derived_domain**{<category>} is used to denote the domain which is derived from the domain of the <category>, and this domain is

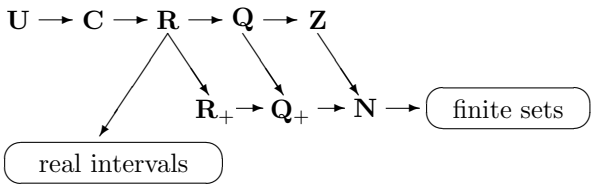


Figure 5: A typical domain inclusion relation

the domain of the declared symbol which is registered by **symbol**; **scope**{<category>, ...} is used to denote that the <category>'s are the scopes of the bound symbol which is registered by **symbol**; **select**{<pattern>, ..., <pattern>} is used to replace itself with one of the <pattern>'s to dramatically decrease the number of rules; **void** is used to replace itself with nothing, to terminate the analysis of current parsing branch.

This kind of rule has the following features: 1) it can be used to express the relation between structure and meanings effectively, intuitively, and easily understood in order to realize its extensibility; 2) it can be used to parse context-sensitive notations through *action*s, and can keep all the knowledge for parsing a notation into one rule by every means. 3) it can be used to effectively process bound variables and indeterminate symbols through the *action*s; 4) it combines a top-down lexical and syntactic analysis, a bottom-up domain analysis and semantic interpretation, and the context-sensitive processing into one parsing in principle.

The **background knowledge** contains two parts: 1) a series of rules for describing the grammar of domain notations appeared in declarations and definitions, and 2) *domain inclusion relation*s to dramatically decrease the number of domain rules. A typical *domain inclusion relation* has been built and shown in Figure 5.

In Figure 5, every arrow means $\supset$, and $\mathbf{U}$ means universal set. The "finite sets", such as $\mathbf{Z}_n$, and the "real intervals", such as $[a, b]$, can also be used. However it is very difficult to describe the relation between various finite sets and between various real intervals in a static knowledge representation, which needs further research in future.

In addition, the knowledge about "$\infty \in \mathbf{N}$" has also been built in **rule base** for parsing ∞. Therefore, a domain name in <domain_pattern> becomes the name of any its subset. However this method requires "small domain matching first". Thus the order of domain rules in **domains** part can not be arbitrary, but has to be dependent on the *domain inclusion relation*s.

For example, in rule for $+$, a series of domain rules, such as $\mathbf{R} \leftarrow \mathbf{R} + \mathbf{N}$, $\mathbf{R} \leftarrow \mathbf{R} + \mathbf{Q}$, $\mathbf{R} \leftarrow \mathbf{R} + \mathbf{R}$, $\mathbf{R} \leftarrow \mathbf{R} + \mathbf{Z}$, etc, are necessary. However, according to the relation in Figure 5, we can absorb them into one domain rule $\mathbf{R} \leftarrow \mathbf{R} + \mathbf{R}$.

Many compound domains, which are constructed by the fundamental domains shown

in Figure 5, such as $\mathbf{R} \times \mathbf{R}$, $\mathbf{R}^{n\times m}$, $\mathbf{Q}[x]$, etc, and their inclusion relations are processed by *action*s. These *action*s declare the whole of compound domain notation, and point out which part is the fundamental domain in its notation. According to Figure 5, their inclusion relations can be determined. For example, $\mathbf{N}^{n\times m} \subset \mathbf{R}^{n\times m}$ because $\mathbf{N}$ and $\mathbf{R}$ are fundamental domains in $\square^{n\times m}$, and $\mathbf{N} \subset \mathbf{R}$. Similarly, $\mathbf{Q}[x] \subset \mathbf{R}[x]$ because $\mathbf{Q}$ and $\mathbf{R}$ are fundamental domains in $\square[x]$, and $\mathbf{Q} \subset \mathbf{R}$, etc.

3.3 Parsing Program

All the grammatical knowledge mentioned above has been built into a knowledge base. We use the knowledge base because the grammar can be modified and extended. The parser on the knowledge-base performs a top-down lexical and syntactic analysis, and then, a bottom-up domain analysis and semantic interpretation. We use the top-down parsing, because we combine the lexical analysis into the syntactic analysis, and because mathematical expressions often include some unique structures being attached with bound variables, such as the x within $\int_a^b f(x)dx$ and its scope is $f(x)$, which are implicitly declared beyond their scopes. Other parsing methods should be studied in future. This parsing program is summarized as follows:

Data Structure is *declaration registration table*s to register any declared or defined symbol or notation, i.e., to memorize its notations, class (variable, function, polynomial, parameter, or indeterminate symbol), domain, codomain, parameterized domain, scopes, assignment state, and its corresponding denotation in MR.

Algorithm is composed of

The input: FR, the output: MR and error message, and the method: as below.

1. If the FR has been declared, return its declared MR.
2. Match the FR with the `structure_pattern` of a rule.
   If they are matched,
   　　if there are *action*s, execute them.
   Else, goto 5.
3. Recursively parse all the sub-FRs through all categories in the `structure_pattern`.
   If one of them is failed, goto 5.
   Obtain all the MRs of the sub-FRs.
4. Match the domains of these MRs with every `domain_pattern` of a **meanings**.
   If they are matched,
   　　if there are *action*s, execute them;
   　　return the MR: a function and a codomain.
   Else,
   　　if there is a next **meanings**, get it and goto 4.

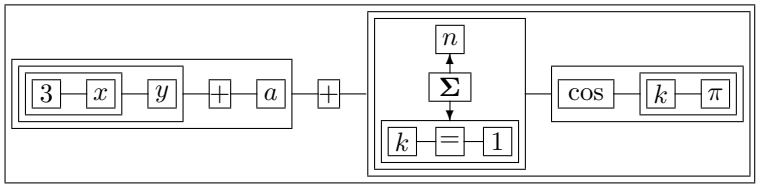


Figure 6: Example of strong formal representation

5. If there is not a succeed rule in the knowledge-base,
   return with error message "fail to parse".
   Else, get the next rule from the knowledge-base, goto 2.

3.4 Formalization and Rule Rewriting

To a same expression, its formalization method is not unique, and the grammar is also not unique. According to the tacit agreements mentioned in Section 2, the formalization methods and their corresponding grammars can be classified into the following three kinds:

1) Strong formalization requires that every structure or operation has to be wrapped by a box, i.e., all tacit agreements have to be expressed by boxes. This means that all determinable and indeterminable tacit agreements are determined by the end user during input (by input of templates). Therefore, in its corresponding grammar, called **strong grammar**, only one grammatical category (e.g., <expression>) is necessary in principle, and the requirement for the order of rules is weak, e.g., it is unnecessary to put the rule for + before the rule for ×.

For example, the strong formalization of $3xy + a + \sum_{k=1}^{n} \cos k\pi$ is shown in Figure 6.

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

2) Weak formalization requires that a portion of boxes in the row structure can be omitted by introducing grammatical categories into the grammar to describe all **determinable tacit agreement**s. Its corresponding grammar, called **weak grammar**, typically contains the following grammatical categories (other similar grammars can also be created).

1. **Sentence**: statements and commands, such as declaration, definition, assignment, substitution, equation solving, etc., which often need words from natural languages to clarify their structures and meanings.
2. **Relation**: various relation operations, such as $\in$, $\notin$, $\subset$, $\subseteq$, $=$, $\neq$, $<$, $\leq$, $\rightarrow$, $\equiv$, etc.

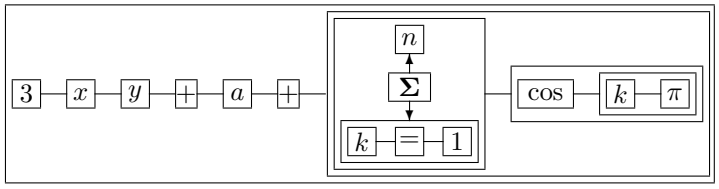


Figure 7: Example of weak formal representation

3. **Calculation**: various addition operations, such as binary $+$, $-$, $\cup$, $\vee$, and unary $+$, $-$, etc.
4. **Term**: various multiplication operations, such as $\times$, $\cdot$, $/$, $\div$, $\cap$, $\wedge$, *mod*, etc. and implicit multiplication.
5. **Factor**: all other operations, such as unary operations $!$, $\#$, $\neg$, ∂, $\square'$, $\bar{}$, $\ddot{}$, etc., fraction stroke, open-close operations $(\cdot)$, $[\cdot]$, $\{\cdot\}$, $\lceil\cdot\rceil$, $\lfloor\cdot\rfloor$, $|\cdot|$, $\|\cdot\|$, etc., exponential operation like as $e^{\square}$, complicated structured operations like $\int_{\square}^{\square}\square \mathrm{d}\square$, $\sum_{\square}^{\square}\square$, $\max_{\square}\square$, $\lim_{\square}\square$, matrix, vector, etc., reserved function calls like as $\sin\square$, self-defined function calls like $f(\square)$, $a_{\square}$, etc.
6. **Atom**: minimal independent mathematical object, which does not contain any operation within it. It has been declared in a certain way, or does not need a declaration. It can be a number, constant name (e.g., imaginary unit i, infinity ∞, etc), domain name (e.g., set of real numbers $\mathbf{R}$), variable name, function name, set name, or indeterminate symbol. It needs no further analysis and interpretation.

The rules of these grammatical categories can be used to describe the operation priority and association rules. However, all factors, their operands, and atoms have to be wrapped by boxes. Moreover, the grammatical categories in a grammar have to be in a determined order.

The weak formalization of $3xy + a + \sum_{k=1}^{n} \cos k\pi$ in Figure 7 can be compared to its strong one in Figure 6.

Because all the factors can be input through the methods template and overlay input by mouse selection [9][13], **weak formalization** can be easily input by key-mouse. There is also a meaningful decrease in the necessary box input. **Weak grammar** is also not difficult to extend even by the end user who understands rewriting rule, because it contains only several easily understood grammatical categories.

3) Free formalization requires that only row, column, and tree structures have to be put into boxes, i.e., the boxes are necessary only for expressing **typesetting tacit agreement**s. Thus, this formalization has no restrictions for using boxes within the row structure. All boxes for determinable and indeterminable tacit agreements can be omitted.

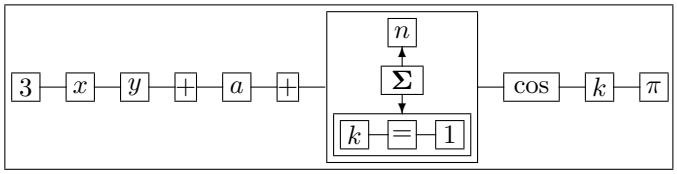


Figure 8: Example of free formal representation

Figure 8 shows the free formalization of the same example $3xy + a + \sum_{k=1}^{n} \cos k\pi$.

To parse this kind of FRs, we have to manage to clarify or recover the boundaries of factors, their operands, and atoms in a row structure **term**. For this purpose, according to our typical classification for **factor**s, we introduce more grammatical categories besides those introduced by **weak formalization** as follows:

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

According to this classification, we can establish the corresponding grammar, called **free grammar**. However, one by one use of these rules to try to match so many **factor**s in a **term** is a right recursive syntactic analysis, because these **factor**s are often lacking their boundaries and omitting multiplication signs between them, and most of the operators appears at the left of their operands (so the corresponding rewriting rule is right recursive like as the form $T \to FT$). This is inconsistent with the whole left recursive top-down parsing (for a list, not a string input, because in **calculation** and **term**, the computing order is often left-to-right (so the corresponding rewriting rule is left recursive like as the form $E \to E + T$, $T \to TF$, etc). Additionally, the parser also performs the context-sensitive analysis and bottom-up semantic interpretation, thus there have to be many big back-tracks. To solve the two problems, we use the following two-parsing method only for the row structure **term**.

Parsing 1 is a preparatory syntactic analysis to wrap **factor**s of a **term** by boxes to obtain a well-formed **term** and its well-formed **factor**s, which still need to wait for confirmation in **Parsing 2**. We introduce the following typical rules of **Parsing 1** in the following order:

1. **Term** is made up of a series of multiplication signs, **atom**s, and the other **factor**s.
2. **Open-close_factor** may contain a series of open-close operations.
3. **Defined_function_call_factor** has a structure like $\square^{\square}(...)$, $\square(...)$, etc.
4. **Reserved_function_call_factor** may have an operand: a reserved_function_call_factor, open-close_factor, defined_function_call_factor, unary_operator_factor, simple_factor, or a series of **atom**s.
5. **Complicated_factor** may have an operand: a complicated_factor, or a series of **factor**s except complicated_factors.
6. **Unary_operator_factor** may have an operand: an open-close_factor or **atom** for !, ¬, and #, or unary_operator_factor for ¬ only.
7. **Simple_factor** operands do not have a row structure.

Parsing 2 does lexical and syntactic analysis and semantic interpretation for a well-formed **term** and its **factor**s that are the same as of the **weak grammar**. For example, to structure $\square(...)$, if it is not a function call declared before, it can be interpreted as a product.

The rules for **Parsing 1** is not unique. We can establish different rules to correspond quite different tacit agreements or conventions. For example, while one reads the expression $\frac{1}{2}[1+\frac{\sin(n+\frac{1}{2})x}{\sin\frac{1}{2}x}]$, one readily knows that 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 will be mistakenly parsed by the above **Parsing 1**, because this example is inconsistent with the convention sin(...). If you prefer the convention of this example, you have to establish another grammar to permit the operand of sin to contain an open-close_factor following an **atom**. In short, to express **indeterminable tacit agreement**s, we can establish many different unambiguous **free grammar**s to make all the desired be determinable in one grammar.

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, this will mean a very complicated analysis and low efficiency in parsing. In fact, the string name is seldom used, and the box wrapping an **atom** is also easy to input by key-mouse. This problem needs further research with regards to other input methods.

Free formalization permits users to input a row structure expression (for example, by keyboard only) without wrapping any box. This is convenient for the input of simple textual expressions. The use of box can be decreased to the minimum. However, so many **free grammar**s are difficult and dangerous to extend by the end user. **Free grammar** has also the problem of grammatically legal but "unreasonable" operand, which requires study in future.

No matter what kind of formalization mentioned above, the parser is only one (see

Section 3.6), and the parsing results have to be the same. For the examples in Figure 6, Figure 7, and Figure 8, the parsing results are the following same MR, if there are $x, y, a \in \mathbf{R}$ and $n \in \mathbf{N}$ in advance (to save space and to read clearly, `.set_of_real_numbers` is denoted as `R`, and `.set_of_natural_numbers` is denoted as `N`).

```
add(add(multiply(multiply(3, N, x, R), R, y, R), R, a, R), R,
    sum(cos(multiply(k, N, irrational_number_pi, R), R), R,
        k, N, 1, N, n, N), R), R
```

3.5 Necessities of Many Grammars

The formalizations and corresponding grammars mentioned above are not unique. They can be combined into a so called **many grammars scheme** to parse various systems of mathematical notations. We summarize its necessities as follows.

1. Operator priority causes the problem of different rule orders. For example, logical operators are prior to relation operators inone grammar, and posterior to in another.
2. Avoidance of ambiguities causes the problem of different rule orders. For example, the rule for derivative such as $f^{(n)}$ is prior to the rule for exponential function such as a^n in one grammar, and random order in another with the agreement that the exponential part of exponential function can not be wrapped by parentheses.
3. Avoidance of ambiguities causes the problem of different meaning interpretations. For example, if the parentheses in expression such as (a, b) means complex number and inner product, and expression such as $\binom{n}{m}$ means binomial coefficient, then the parentheses can not be used to express matrix in this grammar. Similar examples can be found like as $\sin^{-1} x$, $3\frac{1}{2}$, etc.
4. The three formalizations cause the problems of the different structure patterns, and the problem of the number of parsings in different grammars, e.g., the rule for $+$ has to be prior to the rule for $\times$ in **weak grammar**, but unnecessary in **strong grammar**; **term** in **free grammar** needs two-parsing.

In short, the **many grammars scheme** is consistent with the practical situation of mathematical notation use, because there is more than one mathematical notation system in use, and each system is internally unambiguous. The existing systems [9][14] have considered only one grammar, so they could not avoid the ambiguities. However, the **many grammars scheme** can avoid a lot of ambiguities.

3.6 Knowledge-Based Parsing System

In order to realize the **many grammars scheme** to perform more context-sensitive analyses, and to avoid ambiguities, our former knowledge-based parsing method [20] has been extended to be a parsing system, which is discussed below.

The three kinds of grammars mentioned above, in spite of their common components, are different each other in structure pattern, number of rules, order of rules, number of parsings, and meaning interpretations. Therefore, we need the same number of knowledge-bases to implement them. However, to these knowledge-bases, we do not necessarily need the same number of parsers. In fact, we have developed a knowledge representation language, called **meta-language**, to express all these grammars, called *understanding grammar*s. Through a *knowledge base builder*, such a grammar can be converted into an *understanding knowledge base* (being denoted as K^U). Thus, we have created only one parser, called *understanding parser*, on K^U. Through this parser, a FR is translated into a MR, and then, by a *translation parser* on *translation knowledge base*s (being denoted as K^T), MR is translated into a program. The translation from computing result into FR for display needs only another parser, called *reply parser*, on *reply knowledge base*s (being denoted K^R); and the translation from FR to typesetting languages (e.g., TEX) for printing needs only one parser, called *typesetting parser*, on *typesetting knowledge base*s (being denoted K^P). The whole system is shown in Figure 9.

This system has the following features: 1) the input-output interface is independent of the parsing, and the FR is their interface or protocol; 2) the program translation is independent of the parsing, and the MR is their interface or protocol; 3) a parser can correspond more than one knowledge base through **meta-language**. Therefore, the input-output, parsing, program execution, typesetting, and knowledge base preparation can be performed on different machines and in different time.

4 Meta-language and Notation Extension

4.1 Meta-language

A FR based meta-language for representing the grammars of mathematical notation has been developed, and an example of this language for describing a rule for parsing $\int_{\square}^{\square} \square \mathrm{d} \square$ has been shown in Section 3.2. The meta-language includes 50 meta-statements and the code for the fundamental signs. All the definition can be found in [24]. The meta-statements can be divided into the following three classes.

Meta-structures : 9 structures to denote structures of the knowledge representation, such as **rule**, **structure**, **meanings**, **function**, **domains**, etc.

Meta-attributes : 3 attributes to denote attributes of the knowledge representation, such as the class of the grammar (understanding, translation, reply, and typesetting), formalization method (strong, weak, and free), and a title to distinguish same kind of grammars.

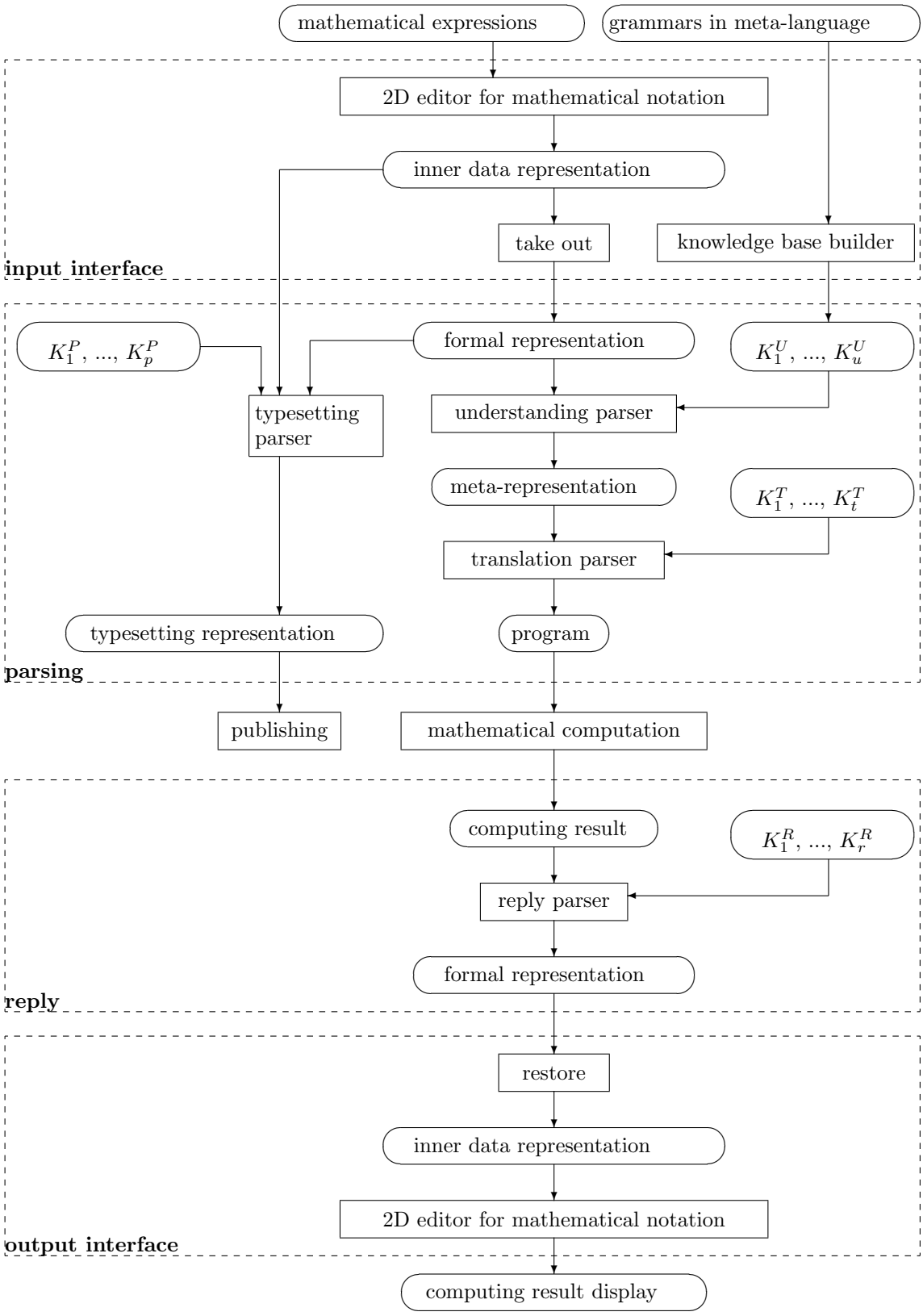


Figure 9: Knowledge-based parsing system based on many grammars scheme

actions : 38 *action*s to denote procedures for context-sensitive processing, and to express concise representation for decreasing the number of rules and patterns, which are classified into the following five kinds.

1. **Declaration and definition**: 18 *action*s to perform the context-sensitive analysis of declaration and definition, such as the registration of a declared symbol, its domain, codomain, assignment state, and scope (examples have been shown in Section 3.2), and also, the specification of parameter, indeterminate, and argument continual recursive analysis (examples will be shown in Section 4.3), etc.
2. **Condition and relation**: 5 *action*s to perform context-sensitive checks about domain consistence or inclusion, declaration state of a changeable symbol (e.g., to confirm if $f(\Box)$ in $f^{\Box}(\Box)$ has been declared as a function call, if not, it is a product $(f^{\Box})(\Box)$; otherwise it means $(f(\Box))^{\Box}$), and the number of components (e.g., to confirm if a matrix has the same number of elements in every row).
3. **Control structure**: 1 *action* to realize the control structure if-then.
4. **Replacement**: 12 *action*s to accomplish meaning replacement for context-sensitive analysis and decreasing the number of rules. For the example of replacement, it is necessary to get the row and column sizes of a matrix to know its codomain (matrix is also regarded as a function); it is also necessary to replace an abnormal or brief notation like as $\Box < \Box \leq \Box$ with its equivalent normal one like as $(\Box < \Box) \wedge (\Box \leq \Box)$, by *action* **reveal**; it is convenient to parse a number by an *action* to avoid the lexical analysis of a number. For the example of decreasing the number of rules, we introduce *action*s **repeat**, **select**, and **option** to describe repetition, selection, and option.
5. **Message output**: 2 *action*s to fulfill the feedback of some operands' interpretation result in the **Parsing 2** of a **free grammar**, and estimate the "weight" of an operand to tell the end user if it is somewhat unreasonable, which needs research in future.

4.2 Meta-language Based Parsing

The meta-language can be used to write and modify the grammars, and then, the parser can parse any input FR according to one of these grammars. Thus we can easily create, modify, and change grammars without programming.

In order to effectively perform context-sensitive analysis and the two-parsing, to speed up the parsing, and to save memory, all the rules of grammars are divided into the following 4 kinds: 1) **well rule** for general rules. 2) **reveal rule** for the rules that contain *action* **reveal** to replace a brief notation with its equivalent normal one, hence no MR is produced. For example, it is necessary to replace a brief expression such as $a < b \leq c$ with its normal one $a < b \wedge b \leq c$ before parsing it. Similarly, to replace $f^n(x)$ with $(f(x))^n$; to replace $\int \frac{dx}{f(x)}$ with $\int \frac{1}{f(x)} dx$; etc. 3) **trivial rule**, and 4) **frame rule**, i.e., the rule for **Parsing 1** in a **free grammar**.

The grammatical knowledge representation written in the meta-language is user-oriented, so it is inefficient in parsing. Thus it is necessary to translate a grammar into a parsing-oriented knowledge base by a *knowledge base builder* (see Figure 9), which can accomplish the following: 1) translation from a grammar in the meta-language into its inner list form, 2) division of the rules into 4 kinds: well, reveal, trivial, and frame, 3) decision of left recursive production in **structure** to determine a syntactic analysis order, 4) establishment of a so called *category-domains corresponding table* for each rule to speed up the domain pattern matching (to avoid many same pattern matchings), and 5) detection of errors in grammars.

The parsing program in Section 3.3 will become quite complicated, if we include the three formalizations, the four kinds of rules, and various *action*s written in the meta-language. Its details can be found in [24].

4.3 Self-definition Extension

Notation extensibility is a very important feature of mathematical notation. There are the following two ways for notation extension: 1) an author defines his own mathematical notations in a context in the time of need, and 2) an author makes a list of mathematical notations in front or back of a document. In this research, the first way is implemented in the method shown in this sub-section; the second way will be studied in Section 4.4.

An operator is often defined for a part of context in a new notation that is different from the existing popular notations and the former defined notations. Many programming languages also have the mechanisms of **self-definition extension**, such as operator definition and overload, but with very limited textual notations. Thus we give an implementable method below.

The notation of an operation definition is composed of an operator sign or a structure of signs, parameter symbols for the operands substitution, and perhaps bound variable symbols with their bounds. Most of the operation definitions have not bound variables. This situation is discussed at first.

The operation definition without bound variable is often formally written to be "$op \stackrel{\text{def}}{=}$ ..." or "$op \stackrel{\triangle}{=}$ ..." etc, where the *op* is the notation of the defined operation and is composed of operator structure and parameters. To distinguish the structure from the parameters, we note that all the parameters have to appear in the right, and have to be declared formally in the right. Thus the operator structure can be obtained through bringing every parameter back to be an open place according to every formal declaration of parameters in the right.

However, it is very difficult to express the distinguishment knowledge mentioned above in a rewriting rule description. The operation definition expression itself is also context-sensitive, because the meaning of a parameter is dependent on its declaration in the right.

Therefore, we use *action*s to realize the distinguishment. The rule for "*op* $\stackrel{\text{def}}{=}$ <expression>, where <parameter declaration>, ..." (<para_dec> for short) is created as follows:

rule{
  structure{
    <sentence> → **any** **select**{ $\stackrel{\text{def}}{=}$, $\stackrel{\triangle}{=}$} <calculation>,
                  where <para_dec1>**repeat**{, <para_dec2> }
      declare_symbol{
          operation{
              any
              parameter_declaration{ <para_dec1> **repeat**{, <para_dec2>} }
              parameter_scope{<calculation>} }
          derived_codomain{<calculation>} }
  }
  meanings{
    function{.defined_operation(**operation**,
          .parameter_declaration(.<para_dec1> **repeat**{, .<para_dec2>}),
          <calculation>)}
  }
}

In this rule, the *action* **any** is used to replace itself with the defined operation notation. The *action* **repeat**{, <pattern>} is a replacement of ", <pattern>, <pattern>, ...". This rule is dependent on the rule for <para_dec>. Thus we have to use the *action* **parameter_declaration** to perform the information transmission from the rule for <para_dec>. The *action* **declare_symbol** is used to define an operation. The *action* **derived_codomain** is used to obtain the operation's codomain. The *action* **operation** brings every parameter (obtained from the *action* **parameter_declaration**) within the **any** back to be an open place, in order to form the pattern of the operation notation. And then, the **operation** registers the pattern, the codomain, and all the scopes of parameters into the *declaration registration table*. Finally, in **function** part, the **operation** replaces itself with the translated textual form of the operation. One of the rules for <para_dec> (will be used in next example) is shown below.

rule{
  structure{
    < para_dec > → <atom> : <domain1> → <domain2>
      declare_parameter{
          symbol{ <atom>, <atom>(**any**) }
          domain{ <domain1> }
          codomain{ <domain2> } }

}
meanings{ **function**{ .declare_function(**symbol**, .<domain1>, .<domain2>) } }
}

Note that not the **declare_symbol**, but the **declare_parameter** is used to declare parameters, because the scope of parameters is not the context following the operation definition, but the scope that is specified by the *action* **parameter_scope** in **operation** in its former rule.

For example, a sentence "$< f,\ g > \stackrel{\text{def}}{=} \int_0^1 f(t)g(t)\mathrm{d}t$, where $f : \mathbf{R} \to \mathbf{R},\ g : \mathbf{R} \to \mathbf{R}$" can be parsed on the two rules mentioned above. Precisely, **any** is replaced by $< f,\ g >$, and **operation** brings it back to be a pattern $< \_,\ \_ >$ according to the declarations of f and g on the rule for <para_dec>. Then $< \_,\ \_ >$, its functional domain of the parameters, and its codomain, which is derived from $\int_0^1 f(t)g(t)\mathrm{d}t$, are registered into the *declaration registration table*. When an operation call like $< u,\ v >$ (u and v are functions declared in advance) is met, the two open places in $< \_,\ \_ >$ are substituted by u and v. When an operation call like $< \ln\sin\pi x,\ \sin 2\pi x >$ is met, i.e., the arguments are expressions, the unique unassigned x is found and become the argument of the two expressions. Finally, the MR of this operation definition is produced to be the following (to save space and to read clearly, `.set_of_real_numbers` is denoted `R`, and `.set_of_natural_numbers` is denoted `N`):

```
.defined_operation(
    lt__comma__gt(f, g), .parameter_declaration(
        .declare_function(.f, .R, .R), .declare_function(.g, .R, .R) ),
    integral(multiply(f(t, R), R, g(t, R), R), R, t, R, 0, N, 1, N), R )
```

Where, if there is not a "." before a function, its codomain follows it; otherwise, its codomain is ignored for concise representation.

The MR of the operation call $< u,\ v >$ is shown below:

```
lt__comma__gt(u, .function_space(.R, .R), v, .function_space(.R, .R)), R
```

The MR of the operation call $< \ln\sin\pi x,\ \sin 2\pi x >$ is the following:

```
lt__comma__gt(
  ln(sin(multiply(irrational_number_pi, R, x, R),R),R),R, .argument(.x),
  sin(multiply(multiply(2, N, irrational_number_pi, R), R, x, R),R), R,
      .argument(.x)), R
```

Similarly, the same rules can also be used to parse other examples, such as "$u(x) \stackrel{\text{def}}{=} ax$, where $x \in \mathbf{R}$", "$\|\boldsymbol{x}\| \stackrel{\text{def}}{=} \sqrt{\sum_{i=1}^{n} {x_i}^2}$, where $\boldsymbol{x} : \mathbf{Z}_n \to \mathbf{R},$ via $i \mapsto x_i$", etc.

If a defined operation notation has a bound variable, it can not be parsed by the rules mentioned above. For example,

$$\bigoplus_{i=m}^{n} a_i \stackrel{\text{def}}{=} \frac{1}{2}a_m + \sum_{i=m+1}^{n} a_i, \text{ where } \boldsymbol{a} : \mathbf{Z}_n \to \mathbf{R}, \text{ via } i \mapsto a_i.$$

There is not an explicit declaration of i and its bounds in the right. Which one is the bound variable in the left? Which one is its bound? How to obtain the pattern $\bigoplus_{\square=\square}^{\square} \square$ corresponding to the left? The bound variable may not appear in the operand. The three i's in $\bigoplus$, $\sum$, and $\mapsto$ are different each other in fact. To gain a clear idea about them, additional natural language description of the bound variable and its bounds is necessary. This needs research in future.

There is another problem in use of an operation definition, i.e., the definition sentence can not specify the knowledge about `domain_pattern` in detail. For example, we create a rule to parse the operation "□ mod □". This rule can include a **structure rule** "<term> → <term> mod <factor>" and several **domain rule**s "$\mathbf{N} \leftarrow \mathbf{N}$ mod $\mathbf{N}$", "$\mathbf{Z} \leftarrow \mathbf{Z}$ mod $\mathbf{R}$", "$\mathbf{Q} \leftarrow \mathbf{Q}$ mod $\mathbf{Q}$", "$\mathbf{R} \leftarrow \mathbf{R}$ mod $\mathbf{R}$", and many others for polynomial. If we self-define it in a context as

$$x \bmod y \stackrel{\text{def}}{=} \begin{cases} x - y\lfloor x/y \rfloor, & y \neq 0 \\ x, & y = 0 \end{cases} \quad \text{where } x, y \in \mathbf{R} \ ,$$

only one **domain rule** "$\mathbf{R} \leftarrow \mathbf{R}$ mod $\mathbf{R}$" is included in fact. Therefore, this problem becomes that we define an operation in a context if we do not care about the lost `domain_pattern`s, or if the operation needs only one **domain rule**. In addition, it is very difficult for end users to extend the rules for operation definition mentioned above. Thus, these rules had better be in a kernel of a grammar, so that can not be modified (see Section 4.4).

4.4 Grammar Extension

The **self-definition extension** mentioned above is very flexible and prior to the definition in grammars. However, it is unthinkable that everything needs self-definition. Similar to the list of symbols in front of a mathematical book, the parsing knowledge is built into a grammar in advance. While the end user wants to introduce a new notation for whole book, and not to define it only for a part of context, he can use the meta-language to realize the so called **grammar extension** discussed and designed below.

In principle, the user can append and modify the user-oriented grammars written in the meta-language. According to the rule construction described in Section 3.2, the end user can append and modify: rule, *domain inclusion relation*, name of grammatical categories, textual **function** part, fundamental signs, and order of rules. However, the end user can not append and modify FR and the meta-language, as well as can not destroy the

domain rules' order, "small domain matching first", within every **meanings**. In practice, appending and modifying a grammar by end users are very dangerous and may cause many problems. If these problems are not considered or resolved, end users had better not be permitted to append and modify any grammar if he is not the grammar's establisher. To realize **grammar extension**, we have considered some dangerous problems and their countermeasures as follows.

1. After a grammar is extended or modified, more than one same `structure_pattern` may appear in different rules; same `domain_pattern` may appear in one rule; a grammatical category may be undefined; trivial rules may form a synonym repetition; a fundamental domain may not be included into a *domain inclusion relation*; domain rules' order in a rule may be destroied. To deal with these problems, a checker has been set up in the *knowledge base builder* to report these errors.
2. After a grammar is changed, some corresponding modification of the originally related rules may be forgot; ambiguities may occur; unexpected input may be accepted. These problems are all undecidable, so they are the end user's responsibility.
3. The modified **function** part is inconsistent with the grammar of the corresponding function in a programming language. This problem can only be found in *translation parsing*.
4. After an extension, the original rules and user appended or modified rules are mixed. It is difficult to perform further maintenance. To solve this problem, a rule index for extension states is necessary. In this index, any rule is in the state of *kernel* (can not be modified), *periphery* (can be modified), *modified*, or *appended*. When errors are found, the history of extension states will be reported.
5. The end user has to consider at least hundreds of notations, their knowledge representations, and their relations, during creating any mathematical notation system. The complexity of this work is enough to cause many errors. In order to lighten this burden, to decrease the redundancy of the knowledge, and to manage the knowledge conveniently, we will establish the following rule library, called **multi-grammar**.

Although grammars are different each other, most of their rules are still similar. There are the following three differences between the rules for the same notation in different grammars: 1) formalizations methods, 2) rules for the two-parsing, and 3) grammatical categories.

Thus if we classify all the rules in different grammars by formalization methods, and only use the grammatical categories of MR, we can find that the rules in one formalization are the same. A rule in a concrete grammar is called **active rule**. A rule in a formalization and in use of only the grammatical categories in MR is called **inactive rule**. All the **inactive rule**s form three **inactive rule libraries** according to the three formalization methods. These rule libraries are called **multi-grammar**s to hint that one can select and

rewrite or "activate" rules in one library to build diverse grammars.

For example, we can activate the below **inactive rule** to form an **active rule** shown in the example in Section 3.2, through rewriting the <expression> to be <factor>, every <integrand> to be <term>, every <upper_bound> to be <calculation1>, every <lower_bound> to be <calculation2>, and every <integral_element> to be <atom>.

rule{
  structure{
    $< \text{expression} > \rightarrow \int_{<\text{lower\_bound}>}^{<\text{upper\_bound}>} < \text{integrand} >$ **select**$\{d, \mathrm{d}\} < \text{integral\_element} >$

      declare_symbol{
        symbol{< integral_element >}
        derived_domain{<lower_bound>, <upper_bound>}
        scope{< integrand >} }
  }
  meanings{
    function{integral(< integrand >,
                    < integral_element >, < lower_bound >, < upper_bound >)}
    domains{

      $\mathbf{R} \leftarrow \int_{\mathbf{R}}^{\mathbf{R}} \mathbf{R}$ **select**$\{d, \mathrm{d}\}$ **void**

      $\mathbf{C} \leftarrow \int_{\mathbf{C}}^{\mathbf{C}} \mathbf{C}$ **select**$\{d, \mathrm{d}\}$ **void** }
  }
}

Multi-grammars can make all rules be produced by the same source, thus the inconsistency of rules among grammars can be avoided. Additionally, we can also establish various well-made grammars in advance for user's selection, which is similar to the manner of style files of TEX, to make **grammar extension** easier and safer.

5 Prototyping

To verify this research, two prototypes have been developed. Their two WYSIWYG human interfaces have basically been implemented in *Tcl/Tk* [13] and in *Prolog* [21]. The parser has been coded in *Prolog*, and the knowledge base builders in *C*. Up to the present, three grammars for the three formalization methods have been established, and each of them contains more than 200 rules. Their knowledge representation covers notations in elementary algebra, elementary function, linear algebra, polynomial algebra, set algebra, logic algebra, calculus, and equation solving. Finally, the parser can translate an input

FR into MR defined in advance, and then, into a *Mathematica* or a *Maple* program. The pictures of the input interface and computation examples have been given in [13][21][22][23]. All these prototypes have together been named to be *nMath*.

The structures, the row, column, and tree, in FR are necessarily minimum kinds of structures. In a practical implementation, in order to decrease the complexity of the grammar, and to increase the efficiency of the input and parsing, other structures should be introduced into *nMath*, such as `matrix`, `fraction`, `square_root`, etc, to form an extended FR.

6 Conclusions

This research has achieved fundamental results of the input and parsing of mathematical notation for improving the environment of mathematical computation. This research has focussed on the formalizing and parsing methods of nearly whole mathematical notation. Through a knowledge-based parsing method, a great portion of traditional modern mathematical notations can be parsed on computers. Many ambiguities can be avoided. The grammars of mathematical notation can be extended by using a meta-language. However, this research is only the first step to approach the input and parsing of whole mathematical notation for mathematical computation. There are still many problems mentioned in this paper need further research, and many valuable applications remain to be developed in future.

Acknowledgements

We deeply acknowledge Prof. Richard J. Fateman (University of California at Berkeley) for his inspired discussion and advice about the abstract model and semantics of mathematical notation, as well as his introduction of references and improvement of the paper writing. We also express grateful thanks to Prof. Yasuyoshi Inagaki (Nagoya University) for his important advice in the description of some standpoints, the use of some technical terms, and improvement of the paper writing.

References

[1] Berman, B.P., & Fateman, R.J.: Optical Character Recognition for Typeset Mathematics, *Proc. ISSAC'94* (Giesbrecht, M., ed.), ACM, 1994, 348-353.

[2] Caferra, R. & Herment, M.: A Generic Graphic Framework for Combining Inference Tools and Editing Proofs and Formulae, *Journal of Symbolic Computation*, **19**(1-3), 1995, 217-243.

[3] Chisholm, P.: MathSpad: `http://www.win.tue.nl/win/cs/wp/mathspad/`, 1993.

[4] Grbavec, A. & Blostein, D.: Mathematics Recognition Using Graph Rewriting, *Proceedings of the Third International Conference on Document Analysis and Recognition*, IEEE, 1995, 417-421.

[5] Ha, J., Haralick, R.M., & Phillips, I.T.: Understanding Mathematical Expressions from Document Images, *Proceedings of the Third International Conference on Document Analysis and Recognition*, IEEE, 1995, 956-959.

[6] Kajler, N.: Building a Computer Algebra Environment by Composition of Collaborative Tools, *Proceedings of DISCO'92* (Fitch,J.P. ed.), LNCS Vol.721, Springer-Verlag, 1992.

[7] Kajler, N.: CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems. *Proc. ISSAC'92* (Wang, P., ed.), ACM, 1992, 376-386.

[8] Kajler, N.: *Environnement graphique distribué pour le Calcul Formel*. Thèse, Docteur en Sciences, Université de Nice-Sophia Antipolis, Mars 1993.

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

[10] Lee, H.J., & Wang, J.S.: Design of a Mathematical Expression Recognition System, *Proceedings of the Third International Conference on Document Analysis and Recognition*, IEEE, 1995, 1084-1087.

[11] Lee, T.: Maple V Release 4: New Features for Engineers and Scientists, *MapleTech*, **3**(1), 1996, 8-13.

[12] Saito, O., Yamauchi, T., & Takahashi, T.: Widespread User Interface for Computer Algebra System (in Japanese), *Transactions of the Japan Society for Industrial and Applied Mathematics*, **4**(4), 1994, 349-357.

[13] Sakurai,T., Zhao,Y., Sugiura,H., & Torii,T.: A User Interface for Natural Mathematical Notations (in Japanese), *Transactions of the Japan Society for Industrial and Applied Mathematics*, **6**(1), 1996, 147-157.

[14] Soiffer, N.: Mathematical Typesetting in Mathematica. Levelt,A.(Ed.) *Proc. ISSAC'95* (Levelt, A., ed.), ACM, 1995, 140-149.

[15] TCI Software Research Inc.: *Scientific WorkPlace*, `http://www.tcisoft.com/tcisoft.html`, 1995.

[16] Twaakyondo, H.M. & Okamoto, M.: Structure Analysis and Recognition of Mathematical Expressions, *Proceedings of the Third International Conference on Document Analysis and Recognition*, IEEE, 1995, 430-437.

[17] Watters, C. & Ho, J.: MathProbe: Active Mathematical Dictionary, *RIAO'94 Conference Proceedings*, New York, Oct.11-13, 1994, 552-569.

[18] Weck, W.: Putting Icons into (Con-)Text, *Proceedings of the Thirteenth International Conference on Technology of Object-Oriented Languages and Systems, TOOLS EUROPE'94*, Versailles, France, 1994.

[19] Winograd, T.: *Language as a Cognitive Process, Volume 1: Syntax*, Addison-Wesley, 1983.

[20] Zhao,Y., Sugiura,H., Torii,T., & Sakurai,T.: A Knowledge-Based Method for Mathematical Notations Understanding, *Transactions of Information Processing Society of Japan*, **35**(11), Nov.1994, 2366-2381.

[21] Zhao, Y., Sakurai, T., Sugiura, H., & Torii, T.: Research in Natural Mathematical Expression Human Interface (in Japanese), *The 61st Human Interface Symposium, Information Processing Society of Japan, SIG Notes*, **95**(70), 1995, 57-64.

[22] Zhao, Y., Sakurai, T., Sugiura, H., & Torii, T.: A Methodology of Parsing Mathematical Notation for Mathematical Computation. *Proc. ISSAC'96* (Lakshman, Y.N., ed.), ACM, 1996, 292-300.

[23] Zhao, Y., Sakurai, T., Sugiura, H., & Torii, T.: Notation Extension Methods in Mathematical Notation Parsing for Mathematical Computation. *Proc. ASCM'96* (Kobayashi, H., ed.), Scientists Inc., 1996, 81-91.

[24] Zhao, Y.: *Knowledge-Based Parsing Method of Mathematical Notation for Improving Mathematical Computation Environment*, PhD thesis, Nagoya University, Sep. 1996.