人　工　知　能　96－1
（1994．10．4）

知識ベースによる数式の意味解釈とその応用

趙燕結* 杉浦洋† 鳥居達生† 桜井鉄也‡
* 久留米工業大学 知能工学研究所
† 名古屋大学 工学部
‡ 筑波大学 電子情報工学系

概要

計算機上での自然な数式記述と，その意味に基づく応用を目指し，知識ベースによる，数式と算法の記述の意味解釈の方法を実現した．この方法は，数式の形式表現と意味表現，数式の構造分析と意味解釈のための知識ベースとそのParser，応用までの変換などから構成される．知識ベースはユーザにより拡張可能であり，様々な分野に特殊化できる．この方法を中心にして，数式の文書処理と意味解釈，及びその計算の実行を統合するシステムを目指す．これにより，数学計算のマン・マシン・インターフェイスを改良することができ，数学計算，算法の表現，数学の教育，科学文書処理などの分野への応用も展開できる．

Knowledge-Based Mathematical Expressions Understanding and Its Applications

Yanjie Zhao* Hiroshi Sugiura† Tatsuo Torii† Tetsuya Sakurai‡
* Intelligence Engineering Laboratory, Kurume Institute of Technology
† Department of Information Engineering, Nagoya University
‡ Institute of Information Sciences and Electronics, University of Tsukuba

Abstract

A knowledge-based method for understanding mathematical and algorithmic notations has been achieved, to aim at the representation of natural mathematical expressions on computer, and to aim at applications based on the meanings of mathematical expressions. The method contains a formal representation and a meaning representation for mathematical expressions, a knowledge base and its parser for their structure analysis and meaning interpretation, and transforms into various applications. It is possible to permit user to extend the knowledge base to be suited for different fields. Around this method, we work for a system which integrates the document processing, meaning interpretation, and computation execution of mathematical expressions. Thereby, we can improve the man-machine interface of mathematical computation and can develop applications in the fields of mathematical computation, algorithm representation, mathematical education, and scientific document processing.

* zhao@lab.kurume-it.ac.jp † sugiura@urus.torii.nuie.nagoya-u.ac.jp ‡ sakurai@is.tsukuba.ac.jp

1. 目的と問題

我々人間は，数学計算の算法の記述に数式を用いる．しかし，数式による算法記述と，それを計算機上で実現するための各種のプログラミング言語の記述の間には，まだ大きなギャップが存在している．そして，数学計算のプログラミングの負担はまだ大きな問題となっている．この負担を少なくすることが我々の研究目的の一つである．それに対する，我々の解決案は自然な数式表現による数学計算のインターフェイスを構築して，数学計算の問題及びその実現算法を自然な数式で記述することである．

数式表現による数学計算に関する研究は,1961 年から 30 年以上にわたって研究され，幾つかの分野 (高級言語設計，パターン認識，文書処理，ソフトウェア仕様言語，数学ソフトウェアのインターフェイス) で，様々な成果をあげた．最近この研究はもう一度活発化している．幾つかのプロトタイプが発表され[2]-[8]，あるものは商品にまで発展している[9]-[15]．これらは，計算機言語のインターフェイスとして開発されたものであり，二次元の構造を持つ数式を直接入力することができる．これらは数式の全体を扱うことを意図せず，システムにとっての必要最小限のサブセットを扱っている．特に宣言の表現は少ない．そのことによって，数式の意味の多義性を避けられるという利点があるが，しかし，1) 表現力が低くなる，2) 普遍性がない，3) 数式表現の拡張機能がないか，もしくは貧弱である．

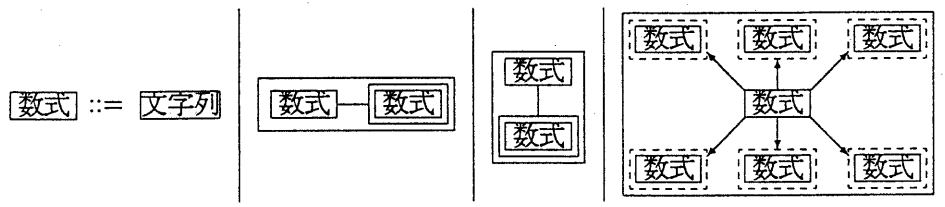
我々の研究は，数式を含んだもう一つの計算機言語を設計するのではなくて，数式自体の意味と構造を研究し[1]，計算機の上に数式による数学計算のインターフェイスを構築し，様々な応用分野 (数学計算，算法の表現，数学の教育，科学文書処理など) に展開することである．この解決案を実現するために，少なくとも下記の問題を解決しなければならない．

1) 数式の形式表現
2) 数式の構造と意味
3) 数式の構造と意味に関する知識表現
4) 数学計算の表現 (数式と算法の混在表現) とその意味解釈
5) 数式の多義性の回避
6) 数式の文書処理と意味解釈，及びその計算の実行を統一するシステム

以下の各章でこれらの問題とその解決について述べる．

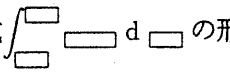
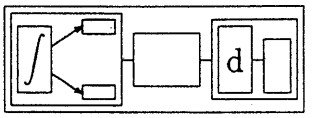
2. 数式の形式表現

数式の印刷イメージは記号の位置関係の記述によって得られる．この記述を抽象して，形式表現[1]という入力の表記を下の図に定義する．



数式の形式表現の中で，一つの数式は一つのボックス ▭ を占めて，ボックスの

中の任意の数式は原子ボックスの中の文字列から横並び，立て並び，六本木という三種類の構造によって構成する．二重ボックス  は同じ構造の繰り返しを意味する．更に，  は構造のオプションを意味する．形式表現によって，数式の記号及び各記号のつながりや，意味の限界などを同時に計算機上に入力，保持できる．形式表現に従った数式入力は，意味に従った数式を構成してゆけるので，人間にとって自然である．勿論入力作業を効率的に行うには，適切なインターフェイスが必要となる．

例えば，定積分式  の形式表現は  である．

3. 数式の構造と意味

数式の形式表現から意味解釈を行った結果を表すために，数式のメタ表現 (meta-representation)[1] という意味の表現を定義する．数式のメタ表現は，下記の数式の関数表記と属する領域である．

function: <関数名>(メタ表現 , ⋯, メタ表現)
domain: <領域名>

例えば，実数領域上の上記の定積分式のメタ表現は下記のように表される．

function: definite_integral(
  function: integral_element( )
  domain: set_of_real_numbers ,
  function: lower_bound( )
  domain: <subset_of_real_numbers> ,
  function: upper_bound( )
  domain: <subset_of_real_numbers> ,
  function: integrand( )
  domain: set_of_real_numbers)
domain: set_of_real_numbers

このようなメタ表現はデータ型，算法，プログラミング言語などとは独立している．数式の意味解釈は，形式表現からメタ表現までの翻訳を意味する．

数式の形式表現から意味を得るために，数式の構造と意味の関係を明らかにする．

数式の構造は，形式表現によって表された数学記号の幾何的連結関係である．数学記号は数式の"単語"として数式の意味を基本的に決めることができる．数学記号は大きく 2 種類に分けられる．

1) 固定された数学記号は，歴史的に形成され，普遍性と意味の不変性が高く，宣言を必要としない．中には，

固定された対象: 数，常数名，領域名 (例: 3.1416, π, e, i, ∞, $\boldsymbol{R}$)

演算子: (例: $\sin\_$; $\sum\_$; $\sqrt{}$; $\int \_ d\_$)

補助記号: 括弧，分離記号，省略記号 (例: $(a_1, a_2, \cdots, a_n)$)

2) 可変数学記号は，変数名，関数名，集合名などであり，宣言 (declaration) が必要である (例えば $x \in \boldsymbol{R}$; $f : \boldsymbol{R} \to \boldsymbol{R}$, via $x \mapsto f(x)$).

それら以外は，自然言語からの習慣用語 (Let, where, via, Solve など) は固定された数学記号とみなす．

数式の構造パターンを固定された数学記号によって決定できる．

宣言の機能は可変数学記号の下記の意味を記述することである．

領域 (domain): 可変数学記号が存在する集合．例えば，$v \in S$の S，

$f : S_1 \to S_2$の S_1と S_2，$\int_a^b f(x)\mathrm{d}x$ の a と b の領域によって束縛変数 x の領域

作用域 (scope): 可変数学記号が文書の中で意味を持つ範囲．2 種類がある:

1) 宣言の次の文から再宣言の前まで

2) 局所的 (例えば，束縛変数の作用域，"$\cdots$ where $v \in S$" のような指示)

解析順序: 1) 通常の順序 (左から右へ，上から下へ，中心から周辺へ)

2) 束縛変数があれば，構造の中の数学記号の解釈順序:

まずは束縛変数の領域，次は束縛変数，最後は束縛変数の作用域

例えば，$\int_a^b f(x)\mathrm{d}x$ に対して，まずは a と b，次は $\mathrm{d}x$ の x，最後は $f(x)$

その他 (関数の定義と呼び出し方など)

数式の意味は，各々の数学記号の個々の意味だけでは決定できない．同じ記号であっても違う構造の中で意味が違うことがある (例えば，$\vec{a}$と $f : A \to B$と $p \to (q \to p)$ の中の$\to$)．しかも，同じ構造で同じ記号でも違う意味を持つことがある (例えば，$x \in \boldsymbol{R}$ に対して，もし $x \geq 0$ ならば，$\sqrt{x}$は実数であるが，もし $x < 0$ ならば，$\sqrt{x}$は複素数になる)．また，同じ構造で同じ記号でも違う関数名が対応できる．例えば，二項演算 $\square \times \square$ は下記の幾つかの意味を持っている．

乗算

整数の乗算 $\boldsymbol{Z} \times \boldsymbol{Z} \to \boldsymbol{Z}$

実数の乗算 $\boldsymbol{R} \times \boldsymbol{R} \to \boldsymbol{R}$

行列の乗算 $\boldsymbol{R}^{l\times m} \times \boldsymbol{R}^{m\times n} \to \boldsymbol{R}^{l\times n}$

$\vdots$

集合の直積 $\cdots$

ベクトルの外積 $\cdots$

$\vdots$

従って，一つの構造に多数の関数名が対応することがある．また，一つの関数名に多数の領域が対応することもある．また，記号から数式を再帰的に構成する時，演算子の優先順序 (例えば，$a + b \times c$ の意味は $a + (b \times c)$ である)，演算子の左，右結合規則 (例えば，$a - b - c$ の意味は $(a - b) - c$ である)，領域の包含関係 (例えば，$\boldsymbol{N} \subset \boldsymbol{R}$が判れば，$\sqrt{2}$が有意義になる) などを考えなければならない．

4. 知識ベースによる数式の意味解釈

数式の構造と意味に関する知識を用いて，数式の意味解釈を実現した．我々はこの知識を知識ベースとして構築し，Parser という解釈プログラムによって，数式の形式表現からメタ表現までの翻訳を実現した[1]．知識ベースは背景知識と一連の規則からなる．

4.1 規則

一般の規則は下記のような記述である.

構造: 一つの構造規則: " < 構文変数 > → < 構造パターン > "
宣言機能: (オプション)
意味 1:
< 関数名 >(< 構文変数 >, ··· , < 構文変数 >)
領域規則 1: " < 値域 > ← < 領域パターン > "
⋮
領域規則 m: ···
⋮
意味 n: ···

上記の意味の下に, 関数と一つの値域から構造に対応するメタ表現を得る.

規則中で, 数学の概念に対応して, 下記の 7 種の構文変数を定義する.

atom 式 は最小独立対象 (数, 常数, 変数, 関数の呼び出し, 領域名など) である.
factor 式 は atom 式と乗算, 加算, 関係, 論理以外の演算子から構成される.
term 式 は factor 式と乗算演算子から構成される.
addition 式 は term 式と加算演算子から構成される.
relation 式 は addition 式と関係演算子から構成される.
logic 式 は relation 式と論理演算子から構成される.
式 は logic 式と文 (宣言, 定義, 方程式の解く, 代入文など) である.

規則中の宣言機能 (declaration effects) は下記のような基本動作から構成される.

宣言中で変数名の位置は構文変数で表す: symbol{< 構文変数 > ···}
その変数が属する領域の位置は構文変数で表す: domain{< 構文変数 > ···}
その変数の作用域の位置は構文変数で表す: scope{< 構文変数 > ···}
その他 (関数の定義と呼び出し方など)

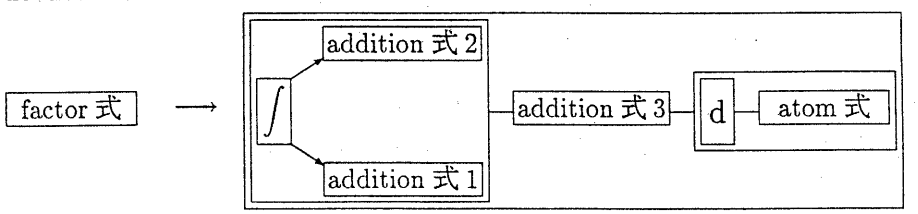
例えば, "$f : A \to B$" を解釈するための構造パターンは"$< expr1 > : < expr2 > \to < expr3 >$"で, "$A \xrightarrow{f} B$" を解釈するための構造パターンは"$< expr2 > \xrightarrow{<expr1>} < expr3 >$"である. 両方の構造は違うが, 宣言機能は同じ:

symbol{ $< expr1 >$ }, domain{ $< expr2 >$ }, codomain{ $< expr3 >$ }

そして, 意味は同じである. 宣言機能によってユーザは知識ベース中の数式の表記手法を拡張することができる.

前の定積分式を解釈するための一つの規則は下のようである.

rule
structure_rule



declaration_effects

symbol(atom 式)

domain(addition 式 1, addition 式 2)

scope(addition 式 3)

meanings

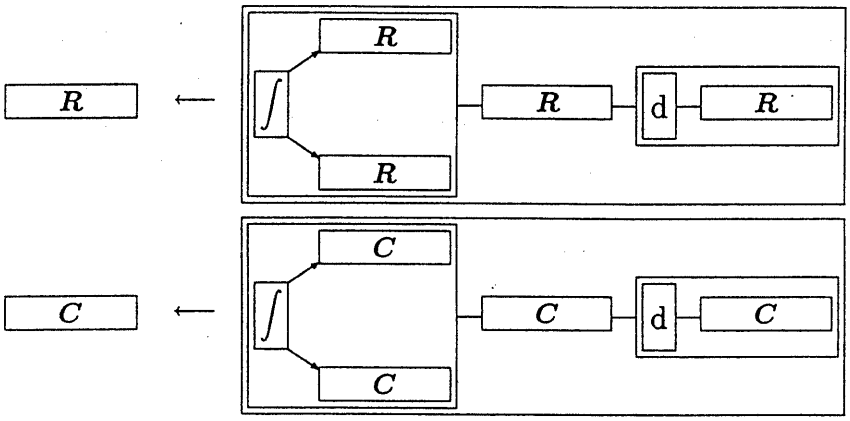
function

define_integral(

integral_element(atom 式), lower_bound(addition 式 1),

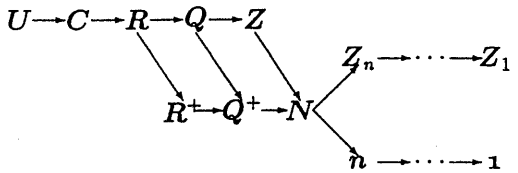
upper_bound(addition 式 2), integrand(addition 式 3))

domains



4.2 背景知識

背景知識は領域名と領域の対応，領域の包含関係などを記述する．我々は下図に示す代表的な領域の包含関係を実現し，規則の中の領域規則の領域名はこの領域のサブセットになる．



ここで，Uは形式的な全集合，Z_nは$\{0,1,\cdots,n-1\}$，nは$\{1,2,\cdots,n\}$を表す(nは1より大きい任意の自然数である)．更に，$\infty \in N$という知識を∞が対応する規則の領域規則に加えた．背景知識により，規則の中の領域規則の数を大幅に減らすことができた．例えば，加法の規則の領域規則 $R \leftarrow R+N$, $R \leftarrow R+Z$, $R \leftarrow R+Q$, $\cdots$ などは，一つの領域規則 $R \leftarrow R+R$ に吸収される．また，$\int_0^{\infty} f(x)\mathrm{d}x$ なども背景知識より$\infty \in N \subset R$だから，領域規則 $R \leftarrow \int_R^R R\mathrm{d}R$ で解釈される．背景知識に基づいて，ベクトルと行列の領域の包含関係を導き出すこともできる．例えば，もし$N \subset R$ならば，$N^{n\times m} \subset R^{n\times m}$.

4.3 Parser

Parserの中では，形式表現と宣言機能の基本動作の処理が固定されている．だから，ユーザは形式表現と宣言機能の基本動作を変更できない．Parserは，領域の名前，領域の包含関係，構文変数の名前，固定された数学記号，メタ表現などを知っていない．それらは全て知識ベースの中にある．従って，ユーザは新しい規則を作って，これら全てを拡張することができる．

Parserは知識ベースによって，top-downの順序で構造の分析を行い，次にbottom-upの順序で意味の解釈を行う．

Parserの扱うデータ構造は，宣言した変数記号の意味を記憶するためのリスト
(変数記号の形式表現，領域，作用域，関数の呼び出し方などのリスト)である．

Parserの算法は，

入力: 数式の形式表現
出力: その数式の意味 — メタ表現
算法概要:

1) 形式表現と対象リストの中の変数記号の形式表現をマッチングする．
もし成功すれば，対象リストの中の登録された意味を出力する．
2) 形式表現と規則の中の構造パターンをマッチングする．
3) 宣言機能があれば，宣言機能の基本動作を実行し，
宣言した意味が対象リストに登録される．
4) 全ての部分構造を再帰的に理解して，それらの関数と領域をもらう．
atom式に到達する時，変数と関数の呼び出しの形式表現と対象リストの対象をマッチングする．もし成功すれば，対象リストの中に登録された意味を出力する．数，常数名，領域名などの形式表現とそれらが対応する規則の中の構造パターンをマッチングする．
5) 部分構造の領域と規則の中の一連の領域パターンをマッチングする．
6) 規則の中の一つの関数と一つの領域を意味として出力する．

5. 数学計算の表現とその意味解釈

人間が用いたり，本に記述された数学計算の数式表現とその実現のプログラミング言語表現の間に大きいギャップが存在している．数式から各種のプログラミング言語へ変換しなければならない．また，新たなプログラミング言語を用いる時には，もう一度プログラムを書き直す必要がある．

我々はこのギャップを埋めるために，数式の意味理解の研究を行っている．しかし，数学計算の表現は**数学記号法** (mathematical notations) だけではなく，計算機時代以来発展した**算法記号法** (algorithmic notations) も含んでいる．従来の数学記号法は数学問題とその証明あるいは計算を表現する．算法記号法の役割は算法の表現である．数式を使った数学計算の表現は双方を混ぜて，3段階で分けることができる．

1) **純数学記号法** (どの数学問題を解くかを表現するため)

例えば，　Solve $f(x) = 0$ for $x \in \boldsymbol{R}$

2) 数学記号法及び一部の算法記号法 (計算方法と算法を表現するため)

例えば，上の問題を Newton 法で解くと，

$x : N \to R$, via $i \mapsto x_i$;

$x_0 \leftarrow$ 出発値; $\epsilon \leftarrow$ 要求精度;

$x_{i+1} \leftarrow x_i - \dfrac{f(x_i)}{f'(x_i)}$ for $i = 0, \cdots, n$ until $\mid x_{n+1} - x_n \mid < \epsilon$;

3) 簡単な数式を含んだプログラミング言語 (算法の効率的表現)

例えば，上の Newton 法に対して，

```
...
float function newton(f, f', x, ε){
  float r;
  r ← x;
  if (f'(x) ≠ 0){
    repeat{ x ← r; r ← x − f(x)/f'(x);} until | r − x | < ε;
    return r;
  }
}
```

実際の数学計算の中にこれらの 3 段階表現は全て必要である．これらの表現は一つの自然な柔軟な数学計算言語になる．このような記述を解釈するために，その一部の算法記号法 (代入文と制御文) の構造と意味に関する知識を加え，知識ベースを拡張した．例えば，上の Newton 法の中に，次のような文を解釈するための規則を実現した:

" ... for *control_variable* = *initial_value*, $\cdots$, *terminal_value* until *condision*; ".

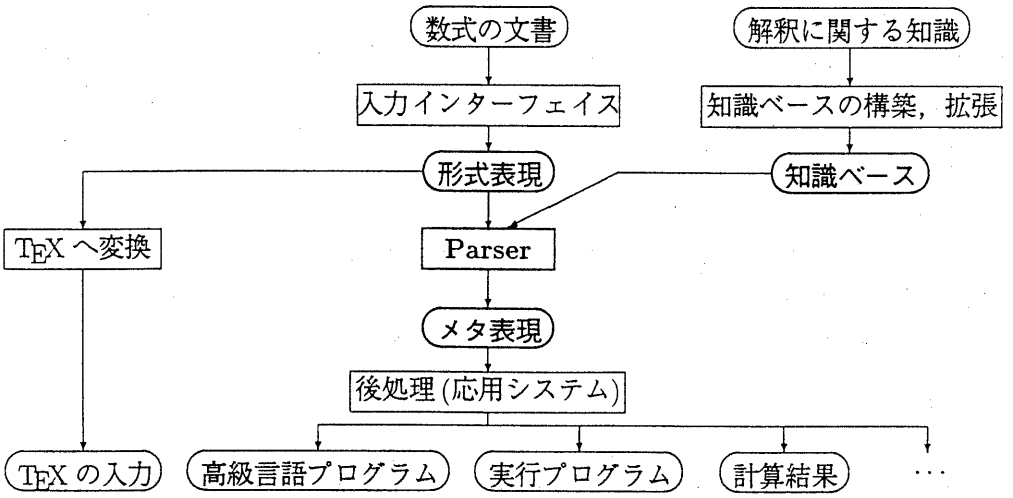
6. 多義性の回避

もし数式の解釈の結果が二つ以上の関数と領域を与えるならば，この数式は多義的数式という．

我々の数式の意味解釈の方法を経て，多義的数式は少なくなる．まず，形式表現によって構造を与えて，一部の多義性を回避できる．次に，宣言により解釈を特定できる．その後，一つの構造に対して一つの規則を作成するから，残る多義性は規則の中に限って存続する．更に，規則の中の領域のパターンに似合って，多義性はすごく減少される．常用の記号法の中では，括弧はまだ多くの意味を持っている．数学計算の文書は，それも基本的には括弧の記号法の意味の宣言によって解釈を一意化する．例えば，(a, b) は一般のベクトルではなく，複素数である場合，" Here, $(\cdot, \cdot)$ means complex number " のような宣言を書くことが必要である．また，括弧に関する構造を解釈するための規則の中に，多数の意味の下に関数名は違うが，領域のパターンが同じである．このような宣言によって，多数の意味から一つの意味を決めることが実現した．

7. 数式表現のインターフェイスとその応用

知識に基づいた数式の意味を解釈する手法を中心として，次のような数式の文書処理と意味解釈，及びその計算の実行を統合するシステム[1]は実現中である．



このシステムの中で，**後処理 (応用システム)** はメタ表現から応用までの変換とその応用自体である．そのためにデータ型と算法を選択することも含んでいる．

8. 実現の現状と今後の課題

知識ベースと Parser を SUN ワークステーションで CESP[16] というオブジェクト指向 PROLOG で実現した．知識ベースと Parser の研究とテストの際に，二次元の形式表現の入力のためのインターフェイスと独立して評価するために，二次元の形式表現と等価な一次元の形式表現を定義し，これを入力して，知識ベースと Parser を経て，メタ表現まで翻訳した．また，後処理の一つの実験として，そのメタ表現を MATHEMATICA[17] の記述に変換して，計算をテストしている．

例えば，前の定積分式の入力は下記のような一次元の形式表現である．

```
[row, [tree, '\int', back_sup, [...], back_sub, [...]],
      [...] , [row, '\rm d', [...]]
]
```

一次元の形式表現の文字列の中に，TEX[18] の数式の表現が全て使われる．

これに対して，意味解釈の結果は下記のようなメタ表現となる．

```
[ definite_integral, set_of_real_numbers,
    [ '.integral_element', [ variable, set_of_real_numbers, ...] ],
    [ '.lower_bound', ...],
    [ '.upper_bound', ...],
    [ '.integrand', ...]
]
```

その中の'.<関数名>' は恒等関数を表す．このメタ表現に対して，MATHEMATICA の式 " Integrate[*integrand*, { *integral_element*, *lower_bound*, *upper_bound* }] " を得ることができる．

現在，知識ベースと Parser は次のような解釈の能力を持っている: 変数の宣言，常数，四則演算 (乗算の見えない演算子も含んでいる)，初等関数，不定積分と 1 変数関数の定積分，導関数，求和式，関数の宣言と定義と呼び出し方，ベクトルと列の宣言と添字変数，階乗，分式，関係式，絶対値，方程式を解く，代入文，記号法の多義性を避けるための宣言，括弧，繰り返し構造など．更に，これらの一部を MATHEMATICA へ変換して，計算をテストしていた．

今後の課題として，より複雑な数式と数学計算の表現の解釈，分厚い公式集によって沢山の数式の解釈テスト，二次元の形式表現の入力と編集，ユーザによる知識ベースの拡張の実現，数式の省略記号の解釈などを研究しなければならない．

参考文献

[1] Yanjie Zhao, Hiroshi Sugiura, Tatsuo Torii, & Tetsuya Sakurai: *A Knowledge-Based Method for Mathematical Notations Understanding* (No.4317), accepted by *Transactions of Information Processing Society of Japan* in Jun.22, 1994.

[2] Soiffer,N.M. & 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.

[3] 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.

[4] Young,D.A. & Wang,P.S.: *GI/S: A Graphical User Interface for Symbolic Computation Systems*, Journal of Symbolic Computation Vol.4 No.3 Dec.1987 pp.365-380 1987.

[5] 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.

[6] Doleh,Y. & 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 Addison-Wesley pp.88-95 1990.

[7] 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.

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

[9] Klerer,M.:*User-Oriented Computer Languages, Analysis & Design*, Macmillan Publishing 1987.

[10] *MathStation*, Ver.1.0 MathSoft, Inc., One Kendall Square, Cambridge,MA,02139 U.S.A. 1989.

[11] Soft Waterhouse, Inc.: *DERIVE User Manual 3rd.Ed.* 1989.

[12] Donnelly,D.: *MathCAD for Introductory Physics*, (MathSoft Inc. Cambridge MA. U.S.A.) Addison-Wesley 1992.

[13] Avitzur,R.: *Milo*, Paracomp Inc. San Francisco, CA U.S.A. (1988).

[14] Bonadio,A. & others: *Theorist Reference Manual, Theorist Learning Guide*, Prescience Corporation 1990.

[15] White,J.& others: *MathKit: Interactive Texts for Math and Science, An Introduction*, Institute for Academic Technology, University of North Carolina at Chapel Hill 1993.

[16] *CESP* Ver.A00 エーアイ言語研究所, 日本〒 247 神奈川県鎌倉市大船 5-1-1 三菱電機（株）, 1994.

[17] Wolfram,S.: *Mathematica, a System for Doing Mathematics by Computer*, Addison-Wesley 1991.

[18] Knuth,D.E.: *The TEX book (Computer & Typesetting/A)*, Addison Wesley 1984.