

# 木構造プログラミングシステム

都 倉 信 樹  
(大阪電気通信大学)

**概要：** ある教育用プログラミングシステムについての報告である。このシステムは

1. プログラムを木構造として、入力・編集・保存する。
2. 具体構文から離れた抽象構文での思考を促進する。
3. 編集系、実行系、代書系を備える。

アルゴリズムから具体的プログラムへの橋渡しの教育実践での利用を期待している。

**キーワード：** 木構造プログラミング,

## 1. はじめに

プログラミング教育に関して、多くのシステムが作られ、教育実践も本格化しつつある。本研究は中高大でのアルゴリズム教育の支援を目指している。[5]でごく簡単な概略を説明したが、ここでより丁寧な説明を試みる。

従来のアルゴリズム教育で用いられたツールには種々ある。当然、プログラム言語が最も重要なツールであるが、1つの難点は言語の習得も同時に求められることになり、学習者の負担がかなり大きく、指導者も苦勞する。アルゴリズムそのものは言語に関わりなく、言語に依存しないアルゴリズムの説明も試みられる。たとえば、フローチャート、NSチャートなどの図式で、アイデアを示せるが、かならずしもうまく行かない。実際に動かさないとアルゴリズムの理解が深まらない。やはり、プログラム化して、実行することが求められる。しかし、実用言語はアルゴリズムの習得に必要なレベルを超えて多くの機能を有し、どうしても敷居が高く、かつ表記も細心の注意をしないとコンパイラからお叱りを受ける。初学者にはなかなか厳しいものとなる。こういう困難を解消しようと多くの努力がなされてきた。

たとえば、高等学校情報科教授用の資料の[6]では、Python, Javascript, Scratch, どんぐり (DNC L)が扱われている。いずれもあまりプログラム言語学習の負担の掛からない方法で、プログラミングを学習することを目指している。他にも種々の提案や実践があるが、それを展望することは本報告の目的ではない。

ここでは、従来型のプログラミング環境とすこし趣の異なる木構造プログラミングという考え方とその実現の一例を報告する。その中心的なアイデアは木構造としてプログラムを扱うということである。

## 2. 木構造とその表現

木構造は親ノードに子ノードがいくつあつら下がるという再帰的な構造であり、出発点は根とよぶノード1つである。多くのアルゴリズムで使われる重要なデータ構造でもある。幾つかの表示法を挙げる。

### 2.1 グラフ表示

データ構造の教科書での木の表現の例を図1に示す。各ノードから子ノードへ枝が描かれている。

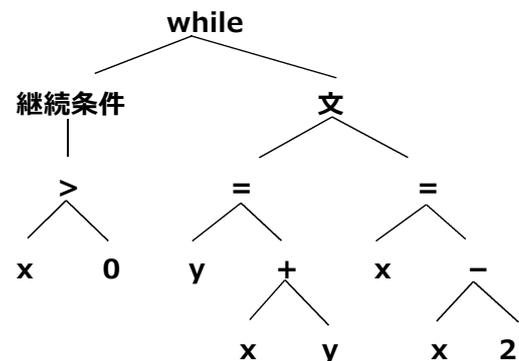


図1 木のグラフ表示の例

階層構造が一目で把握できるという利点がある。コンピュータ画面でどう表現するかが問題になるが、本報告では、つぎのTreeViewを用いる。

## 2.2 TreeView

TreeViewというコントロールはWindows等多くのOSのファイルシステムの内容の表示に用いられており、日常見慣れているであろう。これで簡単なプログラムを表現した例を図2に示す。

1行1ノードであり、子ノードは親ノードより一段インデントを深くして並べる。

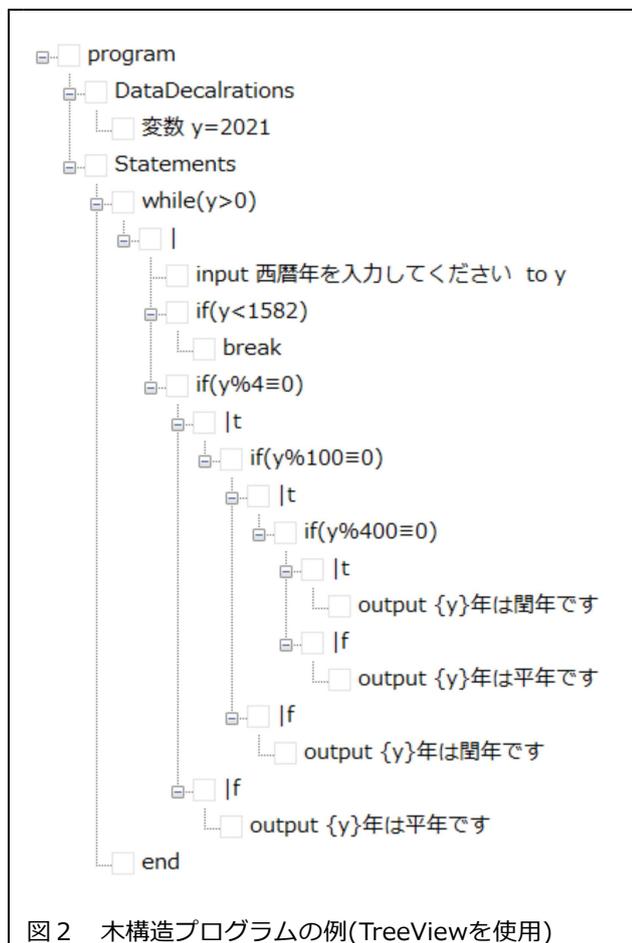


図2 木構造プログラムの例(TreeViewを使用)

ノード間の親子関係を示す線分が引かれている。また、記号田がついているノードがある。これをクリックすると、そのノードを根とする部分木が折りたたまれて根だけが表示され、そのことを示す記号田に代わる。逆に、田をクリックすると、部分木が展開される。また、ノードをクリックして選択するとハイライト表示される。このように展開折りたたみ機能は抽象度のレベルを変えてプログラムを見ることができるという大きな利点がある。

## 2.3 インデントによる表現

上の図2はスクリーンキャプチャであるが、これをテキスト型式で出力して図3に示す。左端にノード

番号と、[ ]内にノードのレベルを表示している。枝はなく、階層関係はインデントの深さだけで表されていることに気づかれるであろう。この情報でTreeViewを復元することができる。「レベルをインデントの深さで表現して、括弧構造の括弧を省ける」と指摘したのが P.Landinである[1]。このアイデアはPython等で採用されよく知られるようになった。

```
F:\C#\¥WriterTestX¥TestData¥CSmaple2b.txt
1 [0] program
2 [1]   DataDeclarations
3 [2]     変数 y=2021
4 [1]   Statements
5 [2]     while(y>0)
6 [3]       |
7 [4]         input 西暦年を入力してください to y
8 [4]         if(y<1582)
9 [5]           break
10 [4]        if(y%4≠0)
11 [5]          |t
12 [6]            if(y%100≡0)
13 [7]              |t
14 [8]                if(y%400≡0)
15 [9]                  |t
16 [10]                    output {y}年は閏年です
17 [9]                  |f
18 [10]                     output {y}年は平年です
19 [7]                |f
20 [8]                  output {y}年は閏年です
21 [5]              |f
22 [6]                output {y}年は平年です
23 [2]     end
```

## 2.4 LDR(Level-Data representation)

TreeView表現をファイルに保存し、また、逆にファイルから復元表示したい。シリアル化と逆シリアル化である。これにはXML、JSON等の方法がよく使われているが、本システムでは、インデントの深さ、つまりレベルを数で表現して、レベルとノード内容(をシリアル化したもの)を並べたテキストで表すLDR(Level-Data Representation)[3][4]で、シリアル化、あるいは、逆シリアル化している。

TreeViewの各ノードには種々のプログラムの情報を保持している。そこから必要なものを整形して出力する機能を代書機能と称している。

### 3. 木構造プログラミング

#### 3.1 木構造プログラミングの発想

このシステムの発想は[2]に溯る。当時ソフトウェア工学で**保守**(maintenance)が大きな問題とされていた。プログラムとドキュメントの乖離を縮めるという意図で、木構造でプログラムを作り、必要なノードに説明を書き加える方法を提案した。木のノードは様々な抽象度レベルをもつが、それぞれのレベルで適切な注記を加えようということであった。当時も木プログラムから具体構文に従うプログラムを生成する「代書系」の構想をもっていた。テンプレートの中に具体的構文への変換法を記述することで、木構造プログラムから、具体的言語でのプログラムを生成する。

#### 3.2 システムの画面

図4は試作システムのプログラム編集時の画面例である。上からメニュー、お知らせを表示するテキストボックス、その下はL,C,R 3つの領域に分割され、左のLが編集中のTreeView(プログラム木)であ

り、中央Cは上下にペインC1,C2に分かれる。C1に挿入可能な構成要素のリストを表示する。その1つを選択すると、その木構造がC2に表示される。Rの上ペインR1には選択中のノードの詳細情報を表示する。Rの下ペインR2には、編集用のボタンがある。

#### 3.3 木構造プログラミングの入力・編集

ノードを1つずつ木に加えるという方法でもプログラムを作れるが、言語機能ごとに部分木を追加するという形で編集する方が楽であるし、機能中心の作業になる。編集中の状況を図4に示す。

たとえば、図4では、input 文で取り込んだ西暦年について、閏年かどうかを判定する。whileの判定を $y > 0$ として、0を入力して終わることを意図していたが、0を入力すると西暦0年についての判定をしてくれる。これをさけるため、ifでbreakする。そのため、inputの次ぎにif1を挿入するところである。ここではC1ペインでif1をクリックするとC2にそのテンプレートが表示される。挿入位置if2を選択して(青→で選択を表示)、「挿入 前に」ボタンをクリックすると、inputとif2の間にif1が入る。

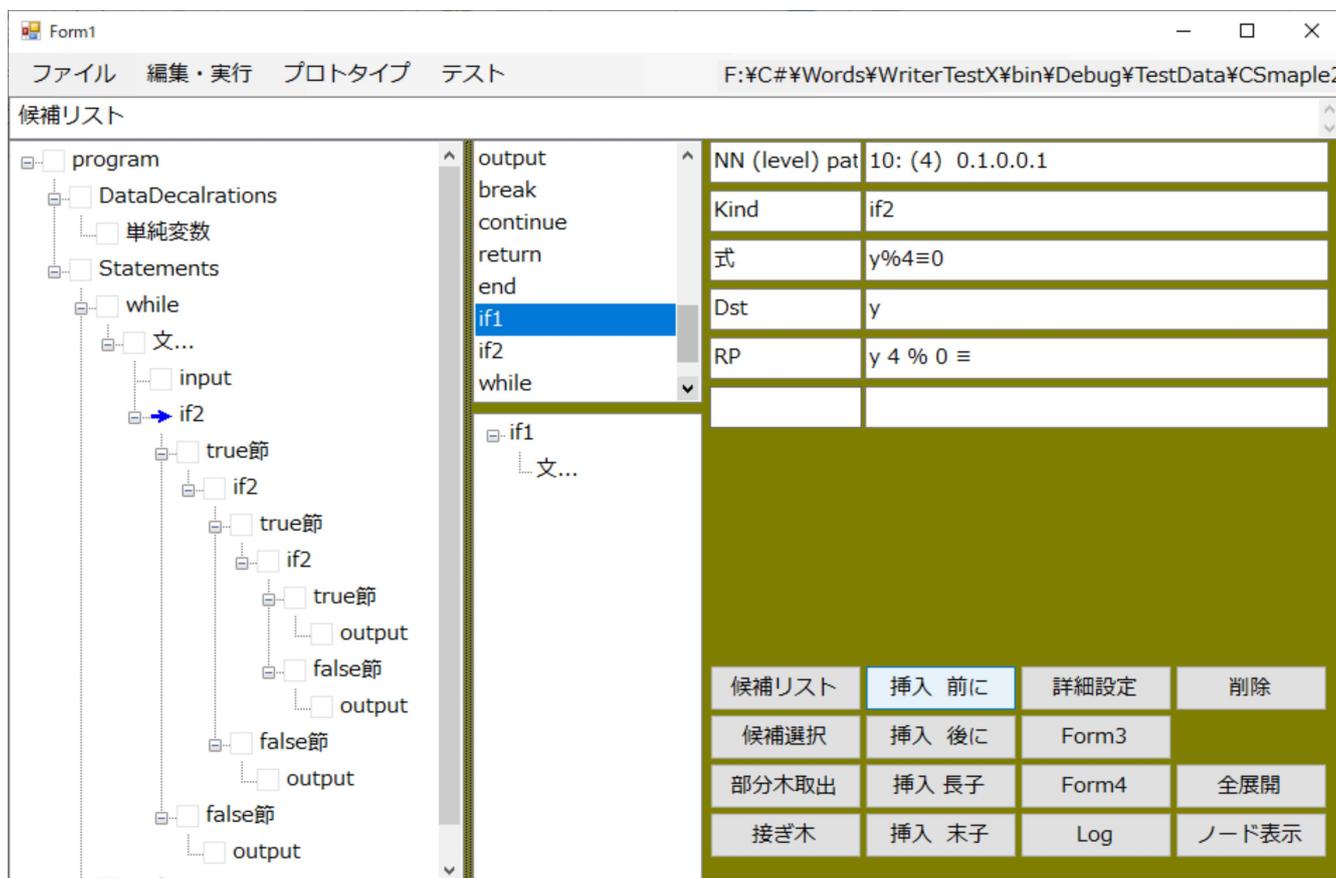


図4 プログラムの入力編集画面

つづいて、文...のところにbreakを選択して、接ぎ木(置換)ボタンをクリックする。これで構造ができる。

次にif1を選択してR 1のテキストボックスの「式」エントリに、 $y < 1582$  と入力し、詳細設定ボタンをクリックする。こうしてできたプログラムが前掲の図2である。

### 3.3 式と制御構造の分離

while(E) do S のように本来、式と実行文は複合している。式はひとつの文字列として扱うが、これを構文解析し、構文木として扱うこともできる。その方が、すべてを抽象構文木で扱うという意味で一貫性があるといえるが、式の木を作ってみるとかなり大きくなることがあり(例えば、図1を完全に木で表現するとノード数が16、深さ4になる)、逆に大局が見えにくいであろう。

式は非常に洗練された表現様式であり、コンパクトに内容を表現する手段である。人類が長年月を掛けて洗練してきたものである。式に比べ実行制御の文の表現様式はまだ歴史が浅いということはあろう。学習者は式についてかなり慣れているということ considering this separation method is used.

## 4. 実行はインタプリティブに行う

### 4.1 式, 制御構造, 変数

編集しただけでは実行出来ない。「実行準備」メニューをクリックして、コンパイラほどではないが準備を行う。ひとつは、変数の準備と変数表の作成である。変数は1つずつノードを用いる。たとえば、配列A[10]と宣言すると、配列を代表するノードAの子ノードとして、A.0、..., A.9という名をもつ単純変数ノードが生成する。参照はA[2], A[i]など通常の形で、たとえば、 $i=5$  のときA[i]は、名前A.5の変数を指す。式は逆ポーランド記法に変換する。こういう準備を行って、実行が可能となる。

### 4.2 実行

現在のところ、まずステップ実行を実現していて、実行中のノードは矢印で表示する。また、変数が変化するとすぐ表示にも反映する。inputは小さい入力専用のウィンドウを開いて入力を待つ。出力はC2ペインに順次書き加えていく。付録の図5にプログラムと対応する出力を示している。

表示実行時の見せ方は種々の可能性があり、今後の検討事項でもある。

### 4.3 実行ログ

実行ログを生成する。これは内部メソッドごとに出力する情報を指定する。かなり大量の情報が出力されるので、これから必要な情報を取り出しやすくする機能を用意している。たとえば、指定した文字列を含む行を間引きすることで情報量を減らす。また、指定した文字列を含む行をインデントしたり、前に空行を置いてパラグラフ化するなどして見やすくする。これを用いてデバグの効率が向上している。現状はデバグ用が主であるが、学習者のために有用な情報出力の選択は検討事項にある。

### 4.4 代書機能

代書機能の対象言語候補のひとつはDNCLである[6]。現仕様とDNCLは距離が近いので、DNCLプログラムを生成することはそう困難はない。このように、具体的構文のある言語に変換して出力するのが代書という作業である。ただし、ノードの種類ごとの変換規則を定めることで変換する範囲に限定している。いわば、直訳レベルに限定しており、高度の意識は想定していないということである。

自然語の翻訳も元言語でのテキストを構文解析し、抽象構文木をもとに対象言語に変換する方法が基本であるが、その対象言語に出力するのと同じことである。プログラム言語間の翻訳はあまり行われませんが、それは必要性が乏しいことと機能差がある場合、翻訳不能となることが原因と考えられる。ここでは抽象言語でのプログラムから出発するので、言語間の機能の差という問題はないが、抽象言語で採用している機能によっては実用言語に変換できないことはありうる。後述の仕様(表1)はごく基本的なものであって、一般の実用言語に翻訳できないことは考えにくい。しかし、現状で欠けているものがあるために実動するプログラムには翻訳できない。それは型である。

アルゴリズムの学習上、型を考慮する必要はあまりなく、[6]の各言語の例題をみても、学生は型はほとんど意識しないだろう。本システムでの言語はほぼこれらと同等であり、型は表だって登場しない。したがって、型をきちんと指定することを求められる言語に代書するというのは無理がある。

実は本システムのインタプリタは型を動的に決め

てデータ処理しているので、内部的には型を扱っている。極端に言えば、ユーザは全く型のことを意識も指定もしないが、代書系が型を推論して代書することも考えられる。ここまで求めるとローカルな変換ではすまなくなってくる。型以外にも指定外の事柄をいろいろ補って矛盾のないプログラムを生成することはどうだろうか。

## 5. 試作システムの言語機能

アルゴリズムの学習・教育のための言語やその支援系は多数提案されている。ここでは、それらを参考にして、基本的な機能を実現しているが、実際に使いつつその適否を検討しなければならないと考えている。現時点での仕様を表1にまとめる。

表1 仕様	
変数	
単純変数	
配列	1次元, 2次元, 3次元
リスト	
式	
演算子	-(~) ! = 代入式 + - * / % < > <= >= == != &&
定数	例 5, 3.14 true "string"
括弧	( ) のみ
変数参照	単純変数, 配列参照 (例. A[3])
関数呼出	(例. f(3, i+j) )
制御構造	
単文に相当	代入, input, output break, continue, end
繰返し	while, for,
条件分岐	2分岐, 多分岐
関数	
数学関数	C#の関数を呼び出す
ユーザ定義関数	in-out引数など
その他	
ノードごとに説明や注意を記入できる	

## 6. いくつかの検討事項

以下のA1~A5の仮説・疑問・問題提起について、皆さんはどうお考えになるか、ご意見をいただければ幸いです。

ば幸いです。

**A1.** アルゴリズムを代入、分岐、繰返しなどの基本要素に分解すれば、それから直接的に機能ごとにプログラムに転換できる。これは従来のように具体言語でアルゴリズムの学習をするのに比べ、より本質的なことに注力でき、学習効果が高まると期待するのだが、どうだろうか？

**A2.** わずかの差かも知れないが、入力量が減少する。シンタクスエラーで入力し直す機会も減るので、生産性の向上に寄与するのではないか。

**A3.** Visual Studioなどの統合開発環境IDEは具体構文でのプログラムをサポートするために、トークンレベルの補完や入力を受け付けつつ種々のチェックを行いプログラマに注意したり、ヒントを出したりする。生産性の向上に非常に役立っている。

たとえば、C#とVisual Basicは機能は極めて近く、機能中心にプログラムできる木構造プログラミングの考えを採用し、C#, Visual BASIC, その他の言語の代書機能を備える形態の統合環境をどうか。

**A4.** 意味が大事なのか、表現が大事なのか？

言語は意味を伝えるための(一般には)一次元の記号の並びである。その記号列は意味を正しく伝えるためにいろいろの約束に従う。また、同じ意味を伝えるにも多種多様な表現が可能である。人間は単に意味を伝えるだけでは飽き足らず、多様な表現の何を選ぶかにも大きな価値、楽しみや喜びを置いているのかもしれない。であるとしたら、本報告のような直接意味を伝えるシステムはコミュニケーションの喜びを奪うとして受容されない恐れはないか。

## 7. むすび

木構造でプログラムをつくる。このとき機能単位ごとに部分木を組み合わせていくので、テキスト指向から機能指向に転換していると考えられる。意味を重視するこのアプローチは学習者にとって、どのような教育効果があるか今後の検証を待ちたい。

筆者はこの開発を通じて、これまでと違う観点から言語機能を見ることが多く、プログラム言語教育にも役立つという期待ももっている。

最後に本報告準備にあたって、種々の支援をいただいた方々に感謝申し上げます。

(mail) nobtokura@gmail.com

参考文献

[1]P.J.Landin:"The next 700 programming languages",  
 C.ACM vol.9,No3, pp.157-166 (1966 March).  
 [2]Y.Nakamoto, M.Iwamoto, M.Hori, K.Hagihara and  
 N.Tokura: An Editor for Documentation in Π-system  
 to Support Software Development and  
 Maintenance,6th In'l Conf. on Software Eng. 330-339,  
 1982.

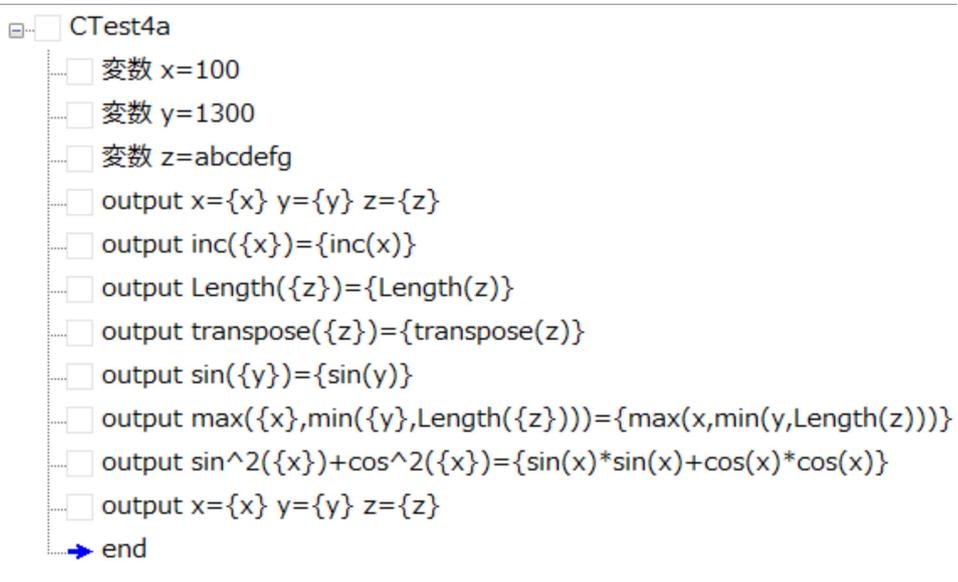
[3]都倉信樹: 入門プログラミング教育につづく科目案  
 SSS2019, 6-1, 2019.8.  
 [4]都倉信樹: ある単純な木表現法の応用について,  
 FIT2019, 6D-7.(Sept. 2019).  
 [5]都倉信樹: 木構造プログラミングシステムの試作,  
 情報処理学会第 83 回全国大会,5G-07,2021.3.  
 [6]兼宗進, 並木美太郎, 長慎也, 長瀧寛之, 長島和平,  
 本多佑希ほか: プログラミング事例集,高等学校教授用資  
 料,東京書籍, 2021.4. <https://ten.tokyo-shoseki.co.jp/>

付録 関数呼出しのテスト例

数学関数はC#の機能呼び出す。

transpose等は組み込み関数機能を作るためのテストである。

outputは補間文字列方式を使い, 出力を簡便にできるようにしている。{ } 内を実行時に評価し結果を  
 文字列表現として挿入してテキストを作る。

 <pre> CTest4a ├─ 変数 x=100 ├─ 変数 y=1300 ├─ 変数 z=abcdefg ├─ output x={x} y={y} z={z} ├─ output inc({x})={inc(x)} ├─ output Length({z})={Length(z)} ├─ output transpose({z})={transpose(z)} ├─ output sin({y})={sin(y)} ├─ output max({x},min({y},Length({z})))={max(x,min(y,Length(z)))} ├─ output sin^2({x})+cos^2({x})={sin(x)*sin(x)+cos(x)*cos(x)} ├─ output x={x} y={y} z={z} └─ end                     </pre>	<p>テストプログラム</p> <p>←ネストの例                  ^ はオペレータではない</p>
<p>出力ペインの出力</p> <pre> x= 100 y= 1300 z= abcdefg inc( 100 )= 101 Length( abcdefg )= 7 transpose( abcdefg )= gfedcba sin( 1300 )= -0.580513008156313 max( 100 ,min( 1300 ,Length( abcdefg )))= 100 sin^2( 100 )+cos^2( 100 )= 1 x= 100 y= 1300 z= abcdefg                     </pre>	
<p>図5 テストプログラムの一例</p>	