Knuth の形式微分アルゴリズムの実現

竹内　章　山口　忠　加地郁夫

(昭和47年11月30日受理)

The Realization of Knuth's Formal Differentiation Algorithm

Akira TAKEUCHI Tadashi YAMAGUCHI Ikuo KAJI

(Received November 30, 1972)

Abstract

This paper demonstrates the application of Knuth's algorithm using list processing for formal differentiation. The algorithm of formal differentiation is composed of the next three main steps.

In step one, the mathematical expression is analyzed and translated into a right threaded tree.

Step two consists of an actual application of derivative rules aud generates the derivative tree from the right threaded tree. We applied Knuth's algorithm to this step.

In the last step, the tree obtained in the first step is translated into a mathematical expression.

To realize these algorithms, we have used the PL/I language which provides a list and string-processing facility.

1. 緒　　言

リスト処理の一技法である Right Threaded Tree (R. T. T.) を使用した Knuth の形式微分アルゴリズム[1]の考えを用い，形式微分プログラムを開発した。このアルゴリズムでの処理は大部分リスト処理，ストリング処理であるので PL/I 言語を用いた。そのため，リスト処理向き言語「MIX」用に書かれていた Knuth のアルゴリズムを PL/I に対して適用できるよう，改良を行なった。さらにこの形式微分アルゴリズムを実現するために，全く新たに算術式から tree へ，tree から算術式へ効率良く変換するアルゴリズムを開発した。さらに次の段階として，FORTRAN 言語のプログラムの中で微分結果を使用できるようにするため，次の機能を持たせた。

(1) 入力式は「name (X)＝算術式」なる形である。この時，微分は X について行われる。

(2) 出力式は (1) の入力式に対して「D$i$name (X)＝i 次導関数式」なる形である。

2. 形式微分アルゴリズム実現の過程

2-1. 形式微分アルゴリズム実現の過程

このアルゴリズムの骨組みを形成しているのは，算術式から tree へ変換するアルゴリズム，Knuth のアルゴリズムを応用した導関数 tree 生成アルゴリズム，さらに tree から算術式へ変

* 電気工学科　系統工学講座

換するアルゴリズムである。

n 次導関数を求めるステップは次の様になる。

Step 1. 微分すべき式を読み込む。式を分解し変数，定数，オペレーションを逆ポーランド記法により変換する。

Step 2. 算術式から tree へ変換する。(アルゴリズム T)

Step 3. 導関数 tree を生成する。(アルゴリズム D)

Step 4. n 次導関数ならば **Step 5** へ，そうでなければ **Step 3** へ戻る。

Step 5. **Step 3** で得られた導関数 tree を算術式に変換し出力する。(アルゴリズム E)

2-2. アルゴリズムの説明

以下でアルゴリズム D，アルゴリズム T，アルゴリズム E を説明する。

2-2-1. 導関数生成アルゴリズム (アルゴリズム D)

R. T. T. の各ノードの構造は図-1 に示される。表-1 に各々の項目の説明がある。このノードの構造は図-2 の様に PL/I の構造体配列を使用することにより簡単に実現することができる。

このアルゴリズムは既存の R. T. T. をポストオーダ[1]で探索しながら，ノードの TYPE に応じて異なる微分規則を実行して導関数 tree を生成するもので，図-3 にフローチャートを示す。なお DIFF [TYPE] は TYPE が表わしている微分規則を実行するルーチンであり，筆者のアルゴリズムでは，定数，変数，＋，－，＊，／，＊＊，アナリーマイナス，LOG, SIN, COS, TAN, COT, SEC, COSEC, SQRT, EXP を処理する機能を与えた。なお，RLINK (P) という表記法は，P というポインター変数によって示されたノードの RLINK 部の内容を意味する。

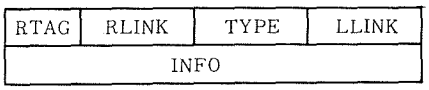


図-1. R. T. T. のノードの構造

```
DCL 1 NODE (2000) ALIGNED,
    2 INFO CHAR (6) VAR,
    2 RTAG BIT (1),
    2 TYPE BIN FIXED,
    2 LLINK BIN FIXED,
    2 RLINK BIN FIXED;
```

図-2. R. T. T. を構成するノードの PL/I 表現

例を挙げると，

$$Y=X**2+A/X$$

は図-4 の R. T. T. で表わされる。左肩の番号はポストオーダを表わしている。

(1) X の微分 TYPE＝0

規則 D (X)＝1 (イ) の tree を構成する。

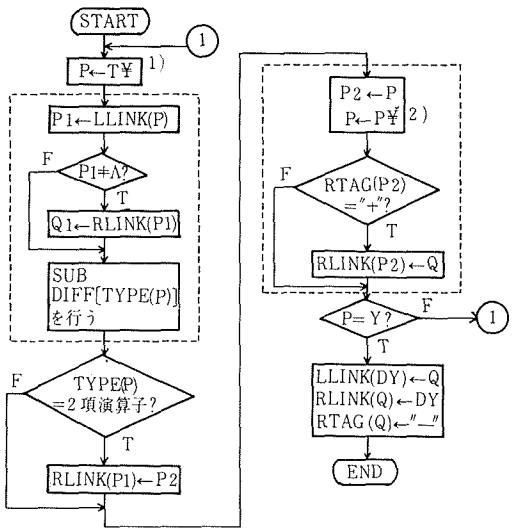
(2) A の微分 TYPE＝0

規則 D (A)＝0 (ロ) の tree を構成する。

(3) ＊＊ の微分 TYPE＝6

規則 D (u＊＊v)＝D (u)＊(v＊(u＊＊(v－1)))＋(LOG u＊D (v))＊(u＊＊v)

今，D (v)＝0，v－1＝1，D (u)＝1 であ



1) tree のポストオーダ I における最初のノードの番地
2) tree のポストオーダーにおける次のノードの番地

図-3. アルゴリズム D フローチャート

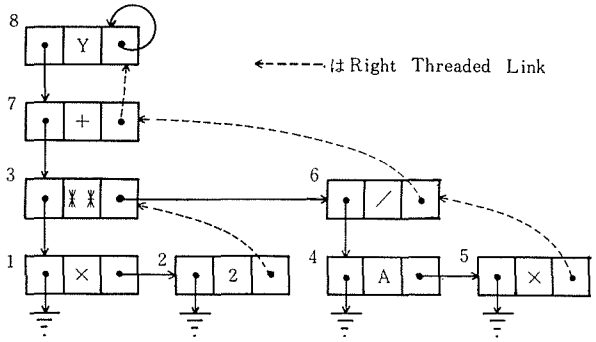


図-4. Y=X**2+A/X の R. T. T.

表-1

項　目	内　容
LLINK	そのノードの左に続く tree の番地を示す。
RLINK	そのノードの右に続く tree の番地を示す。
RTAG	そのノードが Right Threaded Link を持てば '1' B で，持たなければ '0' B を示す1ビットからなる。
INFO	変数，定数，オペレーションがストリングの形で入る。最大6文字である。
TYPE	INFO の表わす内容により夫々異なる整数を与える。 例　INFO の内容　TYPE の値 定数　0 変数　1 +　2 −　3 *　4 /　5 ⋮　⋮

るから，この場合 v*u を表わす tree を構成する。(ハ) の (a) 図は ** に微分規則を実行する直前のそれぞれのポインター変数の指している位置を表わしており，(b) 図は実行後，構成された部分導関数 tree である。このようにすべてのノードを探索した後，Q には (ニ) 図の tree が構成されていて，これが求めるべき導関数 tree である。例で良くわかるように，ポストオーダを使用する利点はオペランドノードの導関数部分 tree を構成してしまった後で，オペレーションノードに到着するところにある。

2-2-2. 算術式から Right Threaded Tree 変換アルゴリズム (アルゴリズム T)

R. T. T. のポストオーダと算術式を逆ポーランド記法変換したオーダが一致するということを，R. T. T. を構成する際に利用した。まず算術式の逆ポーランド記法変換ストリング配列をつくる。この変換によるオーダとスタックを利用し，tree 構成関数 TREE 1, TREE 2, TREE 3 を適当に働かせることにより効率良く，しかも簡単に R. T. T. へ変換するアルゴリズム T を開発した。

このアルゴリズムは次のステップから成り立っている。

Step T 1.

逆ポーランド記法変換配列 (POLISH) から1つ読み込む。

Step T 2. [POLISH はオペレーションか？]

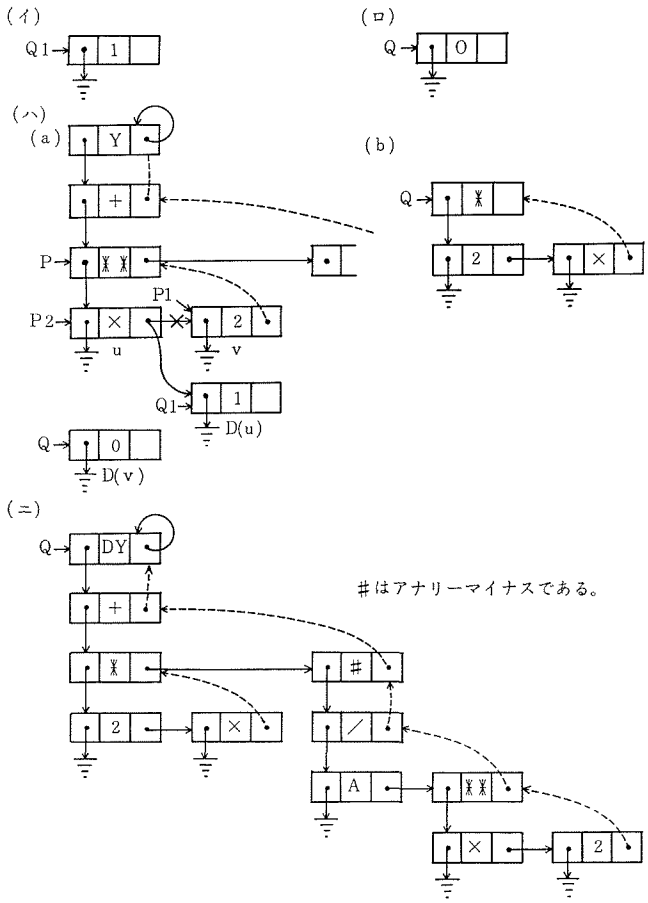


図-5. 導関数 tree 生成例

もしオペレーションならば **Step T 3** へ，でなければ TREE 3 を実行し，その tree の番地をスタックへプッシュダウンする。それから **Step T 5** へ行く。

Step T 3. ［POLISH は二項オペレータか？］

二項オペレータならばスタックを順次2個ポップアップし TREE 1 を実行し，その tree の番地をスタックへプッシュダウンする。ほかの場合には **Step T 4** へ行く。

Step T 4. ［アナリーオペレータ］

スタックを1個ポップアップし TREE 2 を実行し，その tree の番地をスタックへプッシュダウンする。それから **Step T 5** へ行く。

Step T 5. ［終りか？］

POLISH 配列すべてが処理されたならば，リストヘッドを取付ける。終りでなければ **Step T**

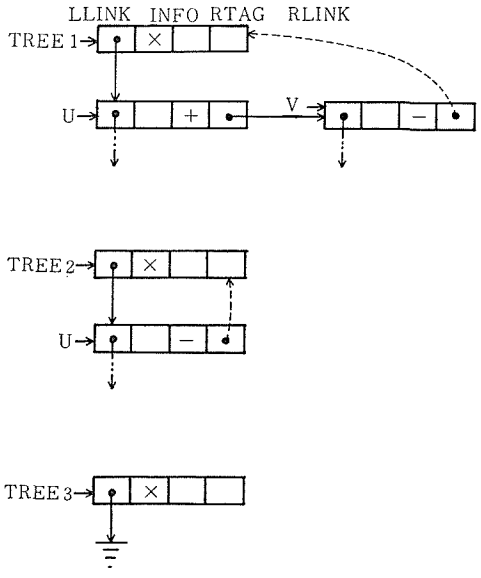


図-6. tree 構成関数

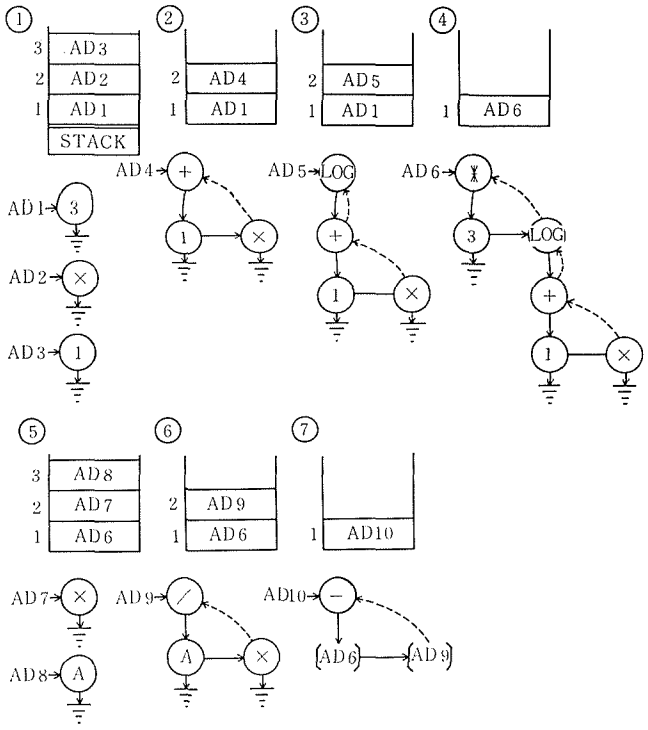


図-7. R. T. T. 構成例

1 へ行く。

算術式

$$Y=3*LOG\ (1+X)-A/X$$

から図-8 に示されている R. T. T. へ変換する過程を図-7 に示す。この場合，算術式の逆ポーランド記法変換した式は

3 1 X + LOG * A X / −

となっている。

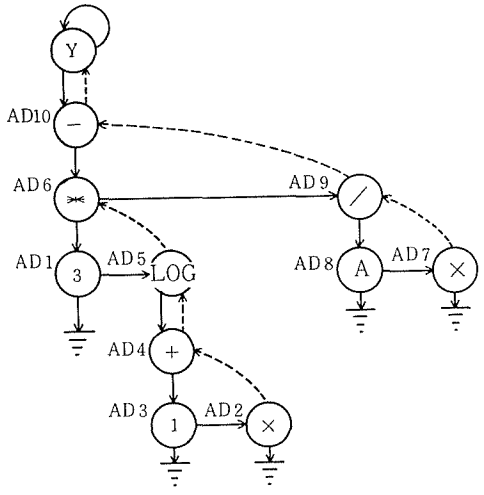


図-8. 完成した R. T. T.

2-2-3. Right Threaded Tree から算術式変換アルゴリズム（アルゴリズム E）

(1) Right Threaded Tree と括弧との対応

R. T. T. を通常の算術式へと変換する際，tree の構造により括弧がつく場合がある。どのような時に括弧がつくということは，以下の規則による。

[規則 1]

オペレーションに表-2 のレベルを与える。あるオペレーションノードに対して，その根となっているノードのオペレーションのレベルの方が高いか，あるいは根のノードのオペレーションが単項オペレータであれば括弧がつく。

規則 1 だけでは図-9 のような R. T. T. は A*B/C*D と変換されてしまい，実際の式 A*B/(C*D) を表わさない。従って次の規則が必要になる。

表-2

オペレーション	レベル
**	3
*, /	2
+, −	1
unary operator	0

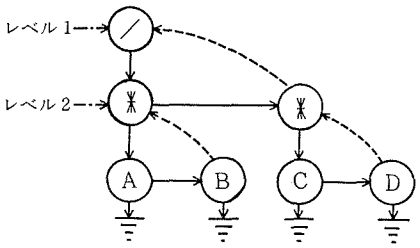
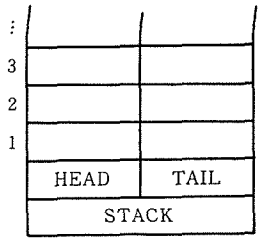
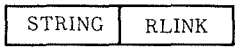


図-9. A*B/(C*D) を表わす R. T. T.



HEAD; 一方向リストの最頭部番地
TAIL; 一方向リストの最尾部番地

図-10. アルゴリズム E で使用するスタックの概要



STRING; tree を構成しているノードの INFO 部のコピー，最大6文字
RLINK; 右に続くノードの番地

図-11. S ノードの概要

[規則 2]

図-9で示されているレベル1の位置にあるノードが除算であり，レベル2の位置にあるノードが除算または乗算であって，しかもレベル1にあるノードに対して右オペランドであれば（すなわち，Right Threaded Link を持てば），そのオペランド部分に括弧がつく。

(2) アルゴリズム E

このアルゴリズムでは，tree のポストオーダと図-10で示されるスタックを使用して，図-11のストリングを項目として右リンクを持つノード（S ノード）からなる一方向リストの連結を行う。この方法を用いることにより「ストリングの最大長は 256 (FACOM 230-60 PL/I の場合)」という制限に対処することができた。

アルゴリズム E は次のステップから成り立っている。

Step E 1. [初期化]

tree のポストオーダにおける最初のノードを訪れる。

Step E 2. [オペレータノードか？]

オペレータノードである場合は **Step E 3** へ，そうでなければ S ノードに INFO 部を複写し，その HEAD, TAIL（この場合は HEAD=TAIL）番地をスタックへプッシュダウンする。それから **Step E 5** へ行く。

Step E 3. [単項オペレータか？]

単項オペレータであれば，現在スタックが示している一方向リストに "(", ")", 単項オペレータをそれぞれ複写した S ノードを連結し，その HEAD, TAIL 番地をスタックへプッシュダウンする。それから **Step E 5** へ行く。ほかの場合には（二項オペレータの場合）**Step E 4** へ行く。

Step E 4. [規則1，規則2を満足するか？]

満足するならば，現在スタックが表わしている一方向リストの HEAD, TAIL 番地を順に2個ポップアップし，その両リストと二項オペレータを複写した S ノードを連結し，さらに "(", ")" を表わす S ノードを連結し，できた一方向リストの HEAD, TAIL 番地をスタックへプッシュダウンする。それから **Step E 5** へ行く。規則を満足しない場合には，括弧をつけずに

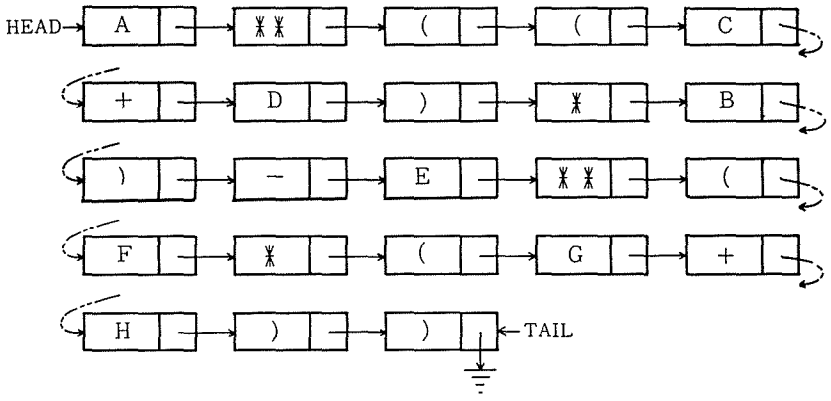


図-12

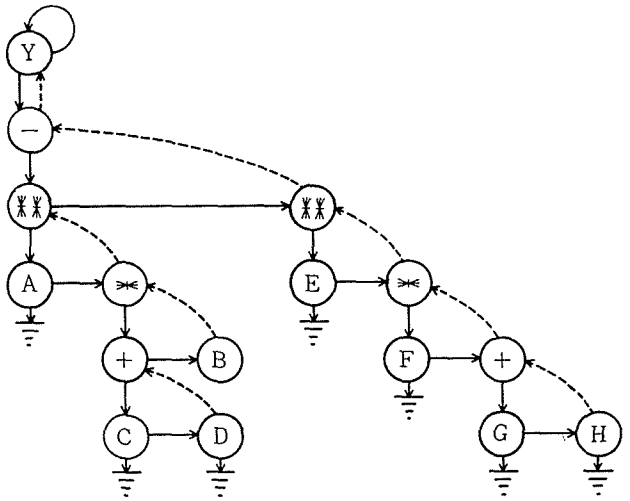


図-13. 一方向リスト

Step E 5 へ行く。

Step E 5.

R. T. T. のすべてのノードの探索が終れば一方向リストの出力を行う。終らなければその R. T. T. のポストオーダにおける次のノードを訪れ **Step E 2** へ行く。

結局，図-12 の R. T. T. は図-13 で示されるような一方向リストに変換される。

R. T. T. はポストオーダでの探索がほかの tree に比べて速く行なえるという性質を持っている。アルゴリズム D，アルゴリズム T，アルゴリズム E ではこれを利用し，ポストオーダ探索を用いている。その結果，計算効率が良くなっている。

2-3. 計 算 結 果

被微分関数として

TEST(X)=SIN(COS(LOG(A＊X＊＊B)))

を入力した時，その一次，二次，三次導関数がそれぞれ D 1 TEST (X)，D 2 TEST (X)，D 3 TEST (X) として求まる。なおこの微分の実行時間は 4165 (*ms*) であった。

```
TEST(X)=SIN(COS(LOG(A*X**B)))

***** DERIVATIVE FUNCTION *****
D1TEST(X)=
(-(A*B*X**(B-1)/(A*X**B)*SIN(LOG(A*X**B))))*COS(COS(LOG(A*X**
B)))

D2TEST(X)=
(-((A*B*(B-1)*X**(B-1-1)/(A*X**B)-A*B*X**(B-1)*A*B*X**(B-1)/(
A*X**B)**2)*SIN(LOG(A*X**B))+A*B*X**(B-1)/(A*X**B)*A*B*X**(B-
1)/(A*X**B)*COS(LOG(A*X**B))))*COS(COS(LOG(A*X**B)))+(-(A*B*X
**(B-1)/(A*X**B)*SIN(LOG(A*X**B))))*(-((-(A*B*X**(B-1)/(A*X**
B)*SIN(LOG(A*X**B))))*SIN(COS(LOG(A*X**B)))))

D3TEST(X)=
(-((A*B*(B-1)*(B-1-1)*X**(B-1-1-1)/(A*X**B)-A*B*(B-1)*X**(B-1
-1)*A*B*X**(B-1)/(A*X**B)**2-(A*B*(B-1)*X**(B-1-1)*A*B*X**(B-
1)+A*B*X**(B-1)*A*B*(B-1)*X**(B-1-1))/(A*X**B)**2-A*B*X**(B-1
)*A*B*X**(B-1)*A*B*X**(B-1)*2*A*X**B/(A*X**B)**2**2)*SIN(LOG(
A*X**B))+(A*B*(B-1)*X**(B-1-1)/(A*X**B)-A*B*X**(B-1)*A*B*X**(
B-1)/(A*X**B)**2)*A*B*X**(B-1)/(A*X**B)*COS(LOG(A*X**B))+(A*B
*(B-1)*X**(B-1-1)/(A*X**B)-A*B*X**(B-1)*A*B*X**(B-1)/(A*X**B)
**2)*A*B*X**(B-1)/(A*X**B)*COS(LOG(A*X**B))+A*B*X**(B-1)/(A*X
**B)*((A*B*(B-1)*X**(B-1-1)/(A*X**B)-A*B*X**(B-1)*A*B*X**(B-1
)/(A*X**B)**2)*COS(LOG(A*X**B))+A*B*X**(B-1)/(A*X**B)*(-(A*B*
X**(B-1)/(A*X**B)*SIN(LOG(A*X**B)))))*COS(COS(LOG(A*X**B)))
+(-((A*B*(B-1)*X**(B-1-1)/(A*X**B)-A*B*X**(B-1)*A*B*X**(B-1)/
(A*X**B)**2)*SIN(LOG(A*X**B))+A*B*X**(B-1)/(A*X**B)*A*B*X**(B
-1)/(A*X**B)*COS(LOG(A*X**B))))*(-((-(A*B*X**(B-1)/(A*X**B)*SIN
(LOG(A*X**B))))*SIN(COS(LOG(A*X**B)))))+(-((A*B*(B-1)*X**(B-1
-1)/(A*X**B)-A*B*X**(B-1)*A*B*X**(B-1)/(A*X**B)**2)*SIN(LOG(A
*X**B))+A*B*X**(B-1)/(A*X**B)*A*B*X**(B-1)/(A*X**B)*COS(LOG(A
*X**B))))*(-((-(A*B*X**(B-1)/(A*X**B)*SIN(LOG(A*X**B))))*SIN(
COS(LOG(A*X**B)))))+(-(A*B*X**(B-1)/(A*X**B)*SIN(LOG(A*X**B))
))*(-((-((A*B*(B-1)*X**(B-1-1)/(A*X**B)-A*B*X**(B-1)*A*B*X**(
B-1)/(A*X**B)**2)*SIN(LOG(A*X**B))+A*B*X**(B-1)/(A*X**B)*A*B*
X**(B-1)/(A*X**B)*COS(LOG(A*X**B))))*SIN(COS(LOG(A*X**B)))+(-
(A*B*X**(B-1)/(A*X**B)*SIN(LOG(A*X**B))))*(-(A*B*X**(B-1)/(A*
X**B)*SIN(LOG(A*X**B))))*COS(COS(LOG(A*X**B)))))
```

図-14. 計 算 結 果 例

3. 今 後 の 問 題

(1) 計算結果を見ればわかるように，得られた導関数式はまだ完全に簡単化された式とはなっていない。この傾向は高次導関数ほど著しくなっていく。従って，導関数 tree を構成した段階で tree の形を検索しながら簡単化を行うアルゴリズムの開発が必要である。

(2) 数値定数として浮動小数点まで取扱えるようにすること。

(3) 取扱える基本関数を増やすこと。

(4) 偏導関数が求められること。

(5) FORTRAN プログラムの中で導関数表現が使用できるようにすること。

4. 結　　語

この PL/I 言語で書かれたプログラムは約 100 (*sec*) のコンパイルおよび結合，編集時間を必要とする。そのため，このプログラムは実行形式で共同ファイルに格納されていて，データとして被微分関数を与えると導関数が求まる様になっている。

今後の問題のうち，(1) は多項式の形式演算にまでその範囲が広がる恐れがあり，実現は多少困難であるが (5) は PL/I 言語のファイル機能を使用することにより，近々，実現の見通しがついている。この方法は従来の数値的解析法に比べて，誤差をほとんど意識しなくともよいという大きなメリットがある。形式積分，多項式の形式的処理の機能を持たせれば，さらに強力なものとなる。計算機の容量が増大し，その演算時間がより速くなるに従い，今まで数値解析法に頼っていた部分を形式演算に置き換えて使用するという場合が多くなろう。

終りに，本研究における計算は北海道大学大型計算センターで行ったことを付記して謝辞にかえたい。

参 考 文 献

1) Knuth, D. E.: The Art of Computer Programming VOL. 1, Addison-Wesley (1969).
2) 藤野精一：リスト処理と言語解析，朝倉書店。
3) FACOM 230-60 PL/I 文法編，使用手引書。