Mapleプログラミングの基礎

Michael Monagan　原著

加藤 満 監修　藤波 拓哉・水出 洋 訳

概 要

これは Maple プログラミングのチュートリアルです. このチュートリアルの目的は, 数値計算や線形代数を行う簡単な Maple プログラムを書いたり, 記号式や数式を簡略化したり変換したりするプログラムを作る方法を示すことです. 電卓として対話しながらMaple を使うことに読者が慣れていることを前提にしております.

目 次

1 はじめに

他のプログラミング言語に慣れている人への一言

- Maple は手続き型プログラミング言語です. また, たくさんの関数型プログラミング機能も持っております. BASIC, Pascal, Algol, C, Lisp あるいは Fortran でプログラムを書いた経験があれば, Maple の計算プログラムを書くのはすぐに出来るでしょう.
- Maple は C や Pascal のように厳しく型を使っておりません. 型宣言は不用です. この点に関しては Maple は Basic や Lisp によく似ております. しかし Maple にも型は存在します. 型のチェックは実行時になされるので, それを行うときははっきりとプログラムに書いておかなければなりません.
- Maple は対話型で, そのプログラミング言語はインタプリタで解釈されます. このインタプリタの負荷のために大規模な計算プログラムを実行するのに適しておりません. けれども高精度の数値計算を行ったり, 計算式を作る道具としては適切なのもです.

この文書は Maple バージョン *V Release 2* に基づいております.[*1)] Maple の開発は現在も続いております. 新しいバージョンが 1, 2 年毎に出てきて, Maple の数学的機能の変更だけでなく, プログラミング言語やユーザインターフェースの変更も行っております. こうした理由から, 私はこの言語の現状の機能から将来なくなる機能なども説明し, さらに Maple の将来のバージョンに取り込まれると思われることなども説明することにしました.

Maple V Language Reference Manual は Maple プログラミングの主要な文献です. これは *Springer-Verlag* から出版されております. ISBN 番号は 0-387-87621-3 です. 他の役立つ情報源は, *Waterloo Maple Softwears* から入手できます. *Maple V Release 2* の *Release Notes* や *Springer-Verlag* から出版されている *First Leaves : Tutorial Introduction to Maple* , それに Maple の中にある ? コマンドで呼び出せる膨大なオンライン文書です. この文書を書く理由の一つは, 私がここで述べた重要ないくつかの事柄は, 他に発表していなかったり文書に埋もれてしまっていることが挙げられます.

1.1 評価

始めに, Maple と従来のプログラミング言語とのもっとも重要な差異を指摘しておきます. 識別子に値が割当てられていなければ, それはそれ自身を表します. これは記号 (*symbol*) と呼ばれてます. 記号は方程式の未知数, 多項式の変数, 積和の添数などを表すのに用いられます. 次の Maple 割当て文を考えてみましょう.

```
> p := x^2+4*x+4;
```

$$p := x^2+4x+4$$

ここでは識別子 p には数式 x^2+4x+4 が割当てられております. 識別子 x には値が割当てられておりません. これはまさに記号で未知数なのです. 識別子 p には値が割当てられております. こちらのほうはプログラミング変数のようなもので, その値はちょうど通常のプログラミング変数のように次の計算で使われます. p の値は何でしょうか ?

```
> p;
```

$$x^2+4x+4$$

[*1)] (訳注) この文書のコマンドや出力は, すべて *Maple V Release 5* に改めてあります. プログラムの一部にエラーなどの修正を行なったものもあります.

それは数式 x^2+4x+4 です. それでは x の値は何でしょうか?

```
> x;
```

$$x$$

それは記号 x です. 変数には記号を含む値を割当てることができるので, 評価 (*evaluation*) の問題が起こってきます. 次の文を考えてみましょう.

```
> x := 3;
```

$$x := 3$$

```
> p;
```

$$25$$

ここでは x に値 3 を割当てて, Maple に p の値を出力するように要請しました. Maple はどんな値を出力するか? 多項式 x^2+4x+4 を出力するのでしょうか, あるいは多項式を評価して $3^2+4\times 3+4$ を計算しその答 25 を出すのでしょうか? 見られるとおり Maple は後者をとりました. この評価の問題は時々刻々出てきて, それがプログラムの効率や意味に影響を与えます. Maple と従来のプログラミング言語との差異, すなわち識別子はプログラミング変数と数学的変数の両方に用いられることはすばらしいことです. しかしこの２つの混同に注意しなければなりません. ユーザが出逢ういろいろな問題点は識別子が記号とプログラミング変数の両方に使えるということに関係があります. それでは, $\int p\,dx$ を計算しようとすればどんなことが起こるでしょうか?

```
> int(p,x);
```

Error, (in int) wrong number (or type) of arguments

積分関数 `int` にエラーが起こりました. ここでは積分変数 x は記号とみなされております. しかし前のほうで x には整数 3 が代入されました. Maple はこの引数を使って `int` 関数を評価するので, 25 を 3 に関して積分しようとしました. 3 に関しての積分することはまったくナンセンスです! x を記号に戻すにはどうするのでしょうか? Maple では次のようにして変数 x を未割当て (*unassign*) とします.

```
> x := 'x';
```

$$x := x$$

```
> int(p,x);
```

$$\frac{1}{3}x^3+2x^2+4x$$

1.2 式 : 和, 積, ベキ, 関数

Maple では数式たとえば $\sin(x+\pi/2)$ や $x^3y^3-2/3$ のようなものは, 式 (*expressions*) と呼ばれております. これらは記号, 数字, 算術演算子および関数から成り立っております. 記号は `sin, x, y, Pi` のようなものです. 数字は $12, \frac{2}{3}, 2.1$ などです. 算術演算子は $+$ (加法), $-$ (減法), $*$ (乗法), $/$ (除法) および `^`(ベキ乗) です. 関数の例として `sin (x), f(x,y), min (x1,x2,x3,x4)` があります. たとえば, 多項式 $p=x^2y+3x^3z+2$ は Maple では次のように入力されます.

```
> p := x^2*y+3*x^3*z + 2;
```

$$p := x^2 y + 3x^3 z + 2$$

そして式 $\sin(x+\pi/2)e^{-x}$ は次のように入力されます.

```
> sin( x+Pi/2 )*exp( -x );
```

$$\cos(x)\exp(-x)$$

Maple は $\sin(x+\pi/2)$ を $\cos(x)$ と簡単化したことに注意して下さい. Maple の式は数式木 (*expression trees*) すなわちコンピュータ用語でいう *DAG* (非巡回的有向グラフ) で表されます. 我々が式を使うために Maple 関数のプログラムを書くとき, 基本的には計算木を扱っていることになります. この計算木を調べる 3 つの基本的ルーチンが `type`, `op` および `nops` です. `type` 関数

$$type(f, t)$$

は式 f が型 t ならば値 *true* を返すものです. 基本の型は `string, integer, fraction, float, '+', '*', '^', function` です. `whattype` 関数も式の型を出力するのに役立ちます. たとえば, 我々の多項式 p は 3 項の和です. したがって

```
> type( p, integer );
```

$$false$$

```
> whattype( p );
```

$$+$$

```
> type( p, '+' );
```

$$true$$

このバッククォート文字 ‘ の使い方に注意して下さい. Maple ではバッククォートは, `/`, `.` などの奇妙な文字を含む文字列に使用されます. バッククォートはフォワードクォート (アポストロフィ) ’ やダブルクォート " と同じものではありません.

他の 2 つの数値型が `rational` と `numeric` です. 型 `rational` は有理数すなわち整数と分数を指します. 型 `float` は浮動小数点数すなわち小数点を含む数字を指します. 型 `numeric` はこれらの数字のいずれか, すなわち型 `rational` か `float` の数字を指します. Maple ユーザは Maple が有理数と近似値を区別していることに注意して下さい. 小数点が存在することが重要なのです! 次の文を考えて下さい.

```
> 2/3;
```

$$\frac{2}{3}$$

```
# That is a rational number, the following is a floating point number
```
(これは有理数で, 次の数字は浮動小数点数です)

```
> 2/3.0;
```

$$.6666666667$$

> **技術者に注意**：あなたがたの大部分の人は, Fortran や C のソフトウェアを使ったりプログラミングしているでしょう. Fortran や C では常に小数点を用います. すなわち 1/2 の代わり 0.5 と書くように学んでいるでしょう. Maple では, 式をタイプするときはたぶん 1/2 とするでしょう. データをタイプするときはたぶん 0.5 を用いるでしょう. Maple では 1/2 と 0.5 は同じものではありません.

我々の例の p はいくつかの項の和です. これはいくつの項を持っているでしょうか？ nops 関数は式のオペランドの個数 (***n**umber of **op**erands*) を返します. 和の場合はそれに含まれる項数になります. 積の場合はそれに含まれる項数です. したがって, 次のようになります.

```
> nops( p );
```

$$3$$

op 関数は式中のオペランド (***op**erand*) を 1 つ抽出するために用いられます. i が 1 から f のnops までの範囲にあるとき, 次の構文

$$op(i, f)$$

は式 f の i 番目のオペランドを抽出するものです. 我々の例では, これは和 f の i 番目の項を抽出することになります.

```
> op( 1, p );
```

$$x^2 y$$

```
> op( 2, p );
```

$$3 x^3 z$$

```
> op( 3, p );
```

$$2$$

op 関数は次のようにも用いられます.

$$op(i..j, f)$$

これは f の i から j までのオペランドの列を返します. たとえば, 次のようになります.

```
> op( 1..3, p );
```

$$x^2 y,\ 3 x^3 z,\ 2$$

役立つ短縮形は op(f) で, これは op(1..nops(f), f) と同じで, f のすべてのオペランドの列を作ります. 和 p の第 2 項は何でしょうか？ これは 3 つの因子の積です.

```
> type( op( 2, p ), '*' );        # It is a product    ( これは積です )
```

$$true$$

```
> nops(op(2,p));          # It has 3 factors  ( これは 3 因子を持っています )
```

3

```
> op(1,op(2,p));          # Here is the first factor  ( これは第一因子です )
```

3

```
> op(op(2,p));            # Here is a sequence of all 3 factors
                          ( これは 3 因子のすべての列です )
```

$3, x^3, z$

添数付き記号名と関数

Maple は 2 種類の変数を持っていて, それを記号名 (*names*) と呼んでおります. それは型 `string` を持つ `x, sin, Pi` のような文字列と, 型 `indexed` を持つ $A_1, A_{i,j}, A_{ij}$ のような添数付き記号名すなわち添字付き変数です. これらの例は Maple では `A[1], A[i,j], A[i][j]` と入力されます. Maple のほとんどの関数はこの 2 種類の変数を受け入れます. Maple の型である `name` は文字列か添字のいずれかを指します.

例

```
> type(a,string);
```

true

```
> type(a,name);
```

true

```
> whattype(A[1]);
```

indexed

```
> type(A[1],indexed);
```

true

```
> type(A[1],name);
```

true

f が添数付き記号名のとき, `nops` 関数は添数の個数を返し, `op` 関数は i 番目の添数を返します.
また op(0, f) は添数の記号名を返します.

例

```
> nops(A[i,j]);
```

2

```
> op(1,A[i,j]);
```

$$i$$

```
> op(0,A[i,j]);
```

$$A$$

```
> nops(A[i][j]);
```

$$1$$

```
> op(1,A[i][j]);
```

$$j$$

```
> op(0,A[i][j]);
```

$$A_i$$

関数は添数付き記号名とよく似た動作をします. 関数の呼び出しの構文は次のとおりです.

$$f(x_1, x_2, \cdots,)$$

ここで f は関数の記号名で, $x_1, x_2, \cdots$ は引数です. nops 関数は引数の個数を返し, op 関数は i 番目の引数を返します. また op(0, f) は関数の記号名を返します.

例

```
> nops(f(x,y,z));
```

$$3$$

```
> op(1..3,f(x,y,z));
```

$$x, y, z$$

```
> op(0,f(x,y,z));
```

$$f$$

これで Maple の式を作ったり, 一部を抽出したりすることができるようになります. 練習問題 6 は Maple の式の一部を抽出する Maple プログラムを書く問題です. ここではいくつかの例を使って, それを対話的にどのように行うかを示して終わりとしましょう.

```
> f := sin(x[1])^2*(1-cos(Pi*x[2]));
```

$$f := \sin(x_1)^2 \ (1 - \cos(\pi\, x_2))$$

```
> whattype(f);
```

$$*$$

```
> op(1,f);
```

$$\sin(x_1)^2$$

```
> op(1,op(1,f));
```

$$\sin(x_1)$$

```
> op(1,op(1,op(1,f)));
```

$$x_1$$

1.3 文： 割当て文, 条件文, 繰返し文

Maple の割当て文, if 文, for 文, while 文の構文は Algol 60 から採ったものです. 割当て文は次のようになっています.

$$name := expr$$

ここで *expr* は任意の式で, name は変数記号名です. ここで, いずれ出てくる評価問題について述べておきましょう. この問題が生じるのは, 変数に数字を割当てることができるように, 記号を含む式にも変数を割り当てることができるからです. 前にやった p への割当て文を考えてみましょう.

```
> p := x^2+4*x+4;
```

$$p := x^2+4x+4$$

この場合は記号名 p に数式を割当て, その数式は記号 x を明示的に含んでおります. それでは同じ記号名を割当て文の両辺で使ったらどうなるでしょうか？

```
> p := p+x;
```

$$p := x^2+5x+4$$

何ら変わったことは起こりません. それではここの評価規則はどのように適用されたかを見てみましょう. 記号名 p には右辺 $p+x$ を評価し, さらに簡単化した結果が割当てられています. $p+x$ を評価するとき, p の値は多項式 x^2+4x+4 で x の値はまさに x そのものです. したがって評価結果は式 $(x^2+4x+4)+x$ です. 次にこれが簡単化されて最終結果 x^2+5x+4 になりました. それでは難題を出しましょう. p に事前に値が割当てていなかったらどうなるでしょうか？ 見てみましょう. p の代わりに q を用いると次のようになります.

```
> q := q+x;
Warning, Recursive definition of name
```

$$q := q+x$$

今度は明らかにユーザが最初に, q にある値を割当てたことを忘れていたとしましょう. これはあり得るケースです. それでは続けて Maple にどんなことが起こるか見てみましょう. 前の例のように q が x のよう

な値を持つことを Maple は絶対許しません.

上の文は Maple の警告をもたらしました. 記号名 q には今度は q を含む数式が割当てられました. q を評価しようとすれば今度はどうなるでしょうか？ $q+x$ を評価しなければなりません. しかし $q+x$ を評価するためには, また q を評価しなければなりません. ここには無限の評価の繰返しが起こります. 筆者の研究室のシステムでは, q を評価しようとすると, 特別なクラッシュが起こり Maple は死んでしまいました.

```
> q := q+x^2;
Error, too many levels of recursion
```

このような警告から回復するには, 単に変数 q を未割当てにする, すなわち, `q := 'q';` とします. 一般に大変コストがかかるので, Maple はすべての再帰的割当てを検出しないことに注意して下さい.

> **注意**: 上の例の q のような記号名の再帰的定義は, 評価の際 Maple がスタック領域をはみ出したときクラッシュするのが普通です. システムによっては Maple は死んでしまいます. 他のシステムでは, 死ぬ前に問題があると報告して停止します. 読者の所で Maple がクラッシュしたことがあれば, いま述べたことが理由である見込みが大いにありえます. アルゴリズムの中に本当の無限ループがあれば, もちろん Maple は死んでしまいます.

通常のプログラミング言語ではすべての変数は値を持っていますから, この再帰的定義の問題は起こりません. 値が割当てられていなければ, 間違いなくエラーです. 言語にもよりますが, 変数はデフォルト値をとるか, その時点でたまたまメモリにあった適当な値をとるか, あるいはまれな場合ですが言語がそれを検出してエラーを出そうとします.

Maple の条件文は次の構文をとります.

if *expr* **then** *statseq*
[**elif** *expr* **then** *statseq*]*
[**elif** *statseq*]
fi

ここで *statseq* はセミコロンで区切られた文の列で, [ ] はオプションの部分を表し, * はゼロ回以上反復される部分を表します. 典型的な **if** 文は次のとおりです.

```
if x < 0 then -1 elif x = 0 then 0 else 1 fi
```

for 文は 2 つの構文を持っております. 第 1 構文は

[**for** *name*][**from** *expr*][**by** *expr*][**to** *expr*][**while** *expr*]
do *statseq* **od**

したがって **for, from, by, to, while** の各節は省略可能です. 省略された場合は, そのデフォルト値はそれぞれダミー変数, 1, 1, ∞ , true となります. **for** ループの典型的な例は次のとおりです.

```
for i to 10 do print(i^2) od;
```

for, from, by および **to** の節を省略すれば, いわゆる **while** ループになります.

```
i := 10^10+1;
while not isprime(i) do i := i + 2 od;
```

```
print(i);
```

for ループと **while** ループを組み合わせるとすばらしいものになることがあります. たとえば, 10^{10} より大きい最初の素数を探すこの例は次のように書けます.

```
for i from 10^10+1 by 2 while not isprime(i) do od;
print(i);
```

上の例は繰返しが終了したあとでも繰返しの指標の値を呼び出すことができることを示しております.

for 文の第 2 構文はいわゆる **for-in** 文です. これは頻繁に起こる繰返しにまさにうってつけのものです.

```
for i to nops(s) do f(op(i,s)) od;
```

ここで *s* は和でもよいが, 一般には Maple の式あるいはリストや集合のようなデータ構造 (これについては次節を参照して下さい) でもよいのです. これは **for-in** 文を使うと次のように書けます.

```
for i in s do f(i) od;
```

for-in 文の構文は次のとおりです.

[**for** *name*] [**in** *expr*] [**while** *expr*]
do *statseq* **od**

2 データ構造

非常に複雑なプログラムでは, データの操作や保存を行ったりします. データの表現法は, 我々の書くアルゴリズムやプログラムの実行速度に影響します. Maple には良いデータ構造がいくつかあります. ここで調べるのは, 式列, リスト (あるいはベクトル), 集合, 表 (あるいはハッシュ表), 配列 です. Maple にはレコードや結合型リストはありません. これらが Maple でどのように実現されるかについては, この節の終わりで述べます.

2.1 式列

式列とはコンマで区切られた式の列のことです. たとえば,

```
> s := 1, 4, 9, 16, 25;
```

$$s := 1, 4, 9, 16, 25$$

```
> t := sin, cos, tan;
```

$$t := sin, cos, tan$$

となります. 式列の式列は 1 つの式列にまとめられます. すなわち, 式列は結合的 (*associative*) です. たとえば,

```
> s := 1, (4, 9, 16), 25;
```

$$s := 1, 4, 9, 16, 25$$

```
> s, s;
```

$$s := 1, 4, 9, 16, 25, 1, 4, 9, 16, 25$$

となります. 特別な記号である NULL は空の式列として用いられます. 式列はいろいろな目的に用いられます. 次の節では式列からリストや集合がどのように作られるかを示します. ここでは関数呼び出しは実際に式列から作られることに注意しておきます. たとえば, Maple の `min` 関数と `max` 関数は引数として任意個数をとること, すなわち引数の式列を受付けます.

```
> max(s);
```

$$25$$

```
> min(s, 0, s);
```

$$0$$

> **注意**： `op` 関数と `nops` 関数は式列に適用できません. その理由は式列自身が関数の引数になるからです. したがって, 呼び出し `op(s);` と `nops(s);` はそれぞれ `op(1, 4, 9, 16, 25)` と `nops(1, 4, 9, 16, 25)` と同値で, これはエラーとなります. `op` や `nops` を使用したければ, あらかじめ式列をリストに直して下さい.

`seq` 関数は式列を作るのに非常に役に立ちます. 2 種類の **for** 文に対応して, 2 種類の使い方があります. 最初の 1 つは次のとおりです.

$$seq(f(i), i = m..n)$$

`seq` の動作はちょうど次の繰返し文をプログラムしたようになります.

```
s := NULL;
for i from m by 1 to n do s:=s, f(i) od;
```

たとえば次のようになります.

```
> seq( i^2, i=1..5 );
```

$$1, 4, 9, 16, 25$$

```
> s := NULL; for i from 1 to 5 do s:=s, i^2 od;
```

$$s :=$$
$$s := 1$$
$$s := 1, 4$$
$$s := 1, 4, 9$$
$$s := 1, 4, 9, 16$$
$$s := 1, 4, 9, 16, 25$$

seq は中間の式列を作らないので for 文より効率が良いことに注意して下さい. seq の他の使い方は

$$seq(\ f(i),\ i = a\ )$$

で, これは次のものと同値です.

```
seq( f(op(i,a)), i=1..nops(a) )
```

ここに 2 つの面白い例があります. coeff 関数は x に関する多項式の i 次の項の係数を求めるものです. Maple の D 関数は微分演算子です.

```
> a := 3*x^3+y*x-11;
```

$$a := 3x^3 + yx - 11$$

```
> seq( coeff(a,x,i), i=0..degree(a,x));
```

$$-11, y, 0, 3$$

```
> seq( D(f), f=[sin,cos,tan,exp,ln] );
```

$$cos, -sin, 1+tan^2, exp, a \to \frac{1}{a}$$

2.2 リストと集合

リスト, 集合, 関数は式列から作られます. リストはオブジェクトをまとめたデータ構造です. 角括弧はリストを作るときに用いられます. たとえば次のようになります.

```
> l := [x,1,1-z,x];
```

$$l := [x, 1, 1-z, x]$$

```
> whattype(l);
```

$$list$$

空のリストは[]で表します. 集合もオブジェクトをまとめたものです. リストと集合の差異は, 集合から重複したものが除かれるということです. 大括弧は集合に用いられます. たとえば次のとおりです.

```
> s := {x,1,1-z,x};
```

$$s := \{1, x, 1-z\}$$

```
> whattype(s);
```

$$set$$

空集合は{}で表します. nops 関数はリストや集合の要素の数を返し, op 関数は i 番目の要素を抽出します. 式列, リスト, 集合 の i 番目の要素を呼び出すのに, 添数記法を使うこともできます. たとえば, 次のとおりです.

```
> op(1,s);
```

```
> s[1];
```

$$1$$

```
> op(1..3,s);
```

$$1, x, 1-z$$

```
> s[1..3];
```

$$\{1, x, 1-z\}$$

次の繰返し文は, リストや集合が要素 x を含むときに true を出力し, そうでないときに false を出力します.

```
for i to nops(s) while s[i]<>x do od;
if i > nops(s) then print(false) else print(true) fi;
```

`member` 関数を使ってもこれと同じことができます. `member(`x`,`s`)` は要素 x がリストあるいは集合 s に属していれば, true を返します. 次のようにすると, リスト l に新しい要素を追加できます.

```
l := [op(l),x];
```

`subsop` 関数を使いリストから i 番目の要素を削除できます.

```
l := subsop(i=NULL,l);
```

この `subsop` 関数は任意の型の式に適用できます. 集合演算子は `union`, `intersect`, `minus` で, たとえば次のとおりです.

```
> t := {x,z,u};
```

$$t := \{x, z, u\}$$

```
> s union t;
```

$$\{x, z, 1-z, u\}$$

Maple ユーザが多分注目することは, Maple が集合の要素を予測できない奇妙な順序で並べることでしょう. 重複した要素を削除したり, 集合の和や積や差を作ったりするするときに Maple が用いるアルゴリズムは, 入力した集合の要素を最初に整列することからすべて始まります. Maple はマシンアドレス (*machine address*) に従って, すなわち要素がコンピュータ内のメモリにある順序で整列します. このことは要素がメモリ内の置かれている場所に左右されるので, その順序は奇妙で予測できないものになるのです. Maple の集合はこのように実装されているので, 集合演算を非常に速く行うことができます.

2.3 表

表やハッシュ表は効率的表を書くのに非常に役立ちます. 表とは離散的なデータ間の 1 対多関数です. たとえば, ここに英語の色名のフランス語とドイツ語の訳語の表があります.

```
> COLOUR[red] := rouge,rot;
```

$$COLOUR_{red} := rouge, rot$$

```
> COLOUR[blue] := bleu, blau;
```

$$COLOUR_{blue} := bleu,\ blau$$

```
> COLOUR[yellow] := jaune, gelb;
```

$$COLOUR_{yellow} := jaune,\ gelb$$

COLOUR 表の定義域は英語で書かれた色名です. この表の値域はフランス語とドイツ語で書かれた色名の式列です. 一般に, 定義域と値域の値は 0 個あるいはそれ以上の値の式列です. 表の定義域の値はキー (*key*) とか指標 (*indices*) と呼ばれております. Maple の `indices` 関数はこの式列を返します. 表の値域の値は値 (*value*) とかエントリ (*entries*) と呼ばれております. Maple の `entries` 関数はその式列を返します. たとえば次のとおりです.

```
> indices(COLOUR);
```

$$[red],\ [yellow],\ [blue]$$

```
> entries(COLOUR);
```

$$[rouge, rot],\ [jaune, gelb],\ [bleu, blau]$$

`indices` 関数と `entries` 関数が表の指標とエントリを返す順序は, 必ずしもそれが表に入力された順序と同じではありません. これは Maple がハッシングを用いているからで, その結果探索は非常に速くなりますが, エントリが作られた順序は失われました. しかし, `indices` 関数と `entries` 関数の出力は 1 対 1 に対応しております.

表を操作するにはどうするでしょうか? 表のキーすなわち指標が与えられると, 対応するエントリはすばやく探すことができます. すなわち次の操作は

```
> COLOUR[red];
```

$$rouge,\ rot$$

色 red のフランス語名とドイツ語名を返します. どの位速いでしょうか? 表の大きさにかかわらず, 処理時間は一定です. 1000 の異なる色を表が持っていても, その表を探索する時間は非常に速いのです. これが表の基本的な使用法です. 情報は割当て文で表に入力され, 表探索は表の添数を用いて行われます. これ以外に表の操作があるでしょうか? `assigned` 関数を用いてあるエントリが表にあるかどうかを検査できます. そして, 未割当て文によって表からエントリを削除できます. たとえば次のようになります.

```
> assigned(COLOUR[blue]);
```

$$true$$

```
> COLOUR[blue] := 'COLOUR[blue]';
```

$$COLOUR_{blue} := COLOUR_{blue}$$

```
> assigned(COLOUR[blue]);
```

$$false$$

```
> print(COLOUR);
```

$$table([\ red = (rouge, rot)\ yellow = (jaune, gelb)\ ])$$

2.4 配列

1 次元配列は次の array コマンドを用いて作られます.

$$array(\ m..n\ );$$

これは指標 $m, m+1, \cdots, n$ をもつ配列を作ります. エントリは, 表のときと同じように割当て文で配列に挿入されます. たとえば次のようになります.

```
v := array(1..n);
v[1] := a;
for i from 2 to n do v[i] := a*v[i-1] mod n od;
```

1 次元配列は指標を 1 個持っている表と似ていますが, こちらの指標は固定された範囲内の整数に限定されております. 配列は表よりも効率よい呼び出しができて, 指標の範囲検査もできます. ここに数列を整列するために使用された 1 次元配列の例があります. たとえば, 上と同じ配列 v =array(1..n) が与えられているとします. ここに書かれたコードは, バブルソートを用いて v を昇順に整列するものです.

```
for i to n - 1 do
   for j from i + 1 to n do
      if v[i] > v[j] then temp := v[i]; v[i] := v[j]; v[j] := temp fi
   od
od;
```

1 次元配列の別の応用は中間的なデータ構造に使うのもです. たとえば, 多項式 $a(x) = \sum_{i=0}^{m} a_i x^i$ を係数のリスト $[a_0, \cdots, a_m]$ で表すとします. 次数 n の多項式 b も与えられているとします. 配列 c を用いて次のように積 $a \times b$ を求めます.

```
m := nops(a) - 1; # degree of a
n := nops(b) - 1; # degree of b
c := array(0..m+n); # allocate storage for the product
for i from 0 to m + n do c[i] := 0 od;
for i from 0 to m do
   for j from 0 to n do
      c[i+j] := c[i+j] + a[i+1] * b[j+1]
   od
od:
```

```
[seq(c[i] ,i=0..n+m)]; # put the product in a list
```

2 次元配列およびもっと高次の配列も同じように動作します. 2 次元配列は次のコマンドで作られます.

$$array(c..d, m..n);$$

1 例として, 3 変数 x_1, x_2, x_3 の対称多項式から成るベクトルが与えられているとします.

```
v := array(1..4):
v[1] := 1:
v[2] := x[1] + x[2] + x[3]:
v[3] := x[1]*x[2] + x[1]*x[3] + x[2]*x[3]:
v[4] := x[1]*x[2]*x[3]:
```

$M_{i,j}$ を v_i の x_j に関する導関数とする 2 次元配列 M を作ってみましょう.

```
> M := array(1..4,1..3):
> for i to 4 do for j to 3 do M[i,j] := diff(v[i],x[j]) od od:
> M;
```

$$M$$

```
> eval(M);
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ x_2+x_3 & x_1+x_3 & x_1+x_2 \\ x_2x_3 & x_1x_3 & x_1x_2 \end{bmatrix}$$

> **注意**: M の値は配列 M の名前そのものであることに注意して下さい. 配列と表の評価規則は特殊なものです. その理由は未割当てエントリーをもつ配列が使えるようにしているためです. ここではこれについて詳細に述べません. 当分の間, 配列や表を出力したり, 手続きから配列や表を返すときはいつも eval 関数を使って下さい.

array に関する詳細な情報が欲しいときは, ?array で調べて下さい. また添数 1 から始まる 1 次元配列はベクトルを表すのに用いられ, 添数 1 から始まる 2 次元配列は行列を表すのに用いられることに注意して下さい. 「**行列とベクトル**」の節も参照して下さい.

2.5 レコード

Maple は Pascal の record や C の struct のようなレコード型データ構造を表面的には持っておりません. レコード型データ構造とは, 異質のオブジェクトの集まりすなわち必ずしも同一の型でないオブジェクトの集合を保持するデータ構造のことです.

レコード型データ構造が欲しくなる場合の 1 例は, 四元数を表すデータ構造を探しているときでしょう. 四元数とは, a, b, c, d を実数としたとき, $a+bi+cj+dk$ の形をした数です. 四元数を表示するには, 4 つの数値 a, b, c, d のみ保存すればよいのです. 他の例は $Q[x]$ 内で行う多項式の因数分解を表示するデータ構造

です. $a(x)$ の因数分解は次のようになります.

$$a(x) = u \times f_1^{e_1} \times \cdots \times f_n^{e_n}$$

ここで因子 $f_i \in Q[x]$ の各々はモニックかつ既約です. 我々は因子 f_i と指数 e_i と単位 $u \in Q$ を保持する必要があります. Maple にはレコードを表す方法はいくつかあります. もっとも簡単でもっともわかりやすい方法はリストを用いることです. すなわち, 四元数 $a+bi+cj+dk$ をリスト $[a,b,c,d]$ で表し, $a(x)$ の因数分解は f を $[f_i, e_i]$ の形のリストのリストとして $[u, f]$ のリストで表すことです. このデータ構造の要素を参照するには添数を用います. たとえば因数分解の単位元の部分は `a[1]` で与えられます. 次の例に示すように, 要素を参照するのに記号を使いたければ, 名前を定数と定義するマクロ (*macro*) 機能を用いることができます.

```
> a := [-1/2,[[x+1,2],[x-1,1]]];
```

$$a := \left[-\frac{1}{2}, [[x+1, 2], [x-1, 1]]\right]$$

```
> macro(unit=1,factors=2,base=1,exponent=2);
> a[unit];
```

$$-1/2$$

```
> a[factors][1][base];
```

$$x+1$$

```
> a[unit]*a[factors][1][base]^a[factors][1][exponent]*
    a[2][2][1]^a[2][2][2];
```

$$-\frac{1}{2}(x+1)^2(x-1)$$

Maple でレコードを表す第 2 の方法は関数呼び出しを用いることです. すなわち, $a+bi+cj+dk$ を `QUARTERNION(i,j,k,l)` で表します. この表示方法の利点は, 関数に対するいろいろな演算方法を Maple に伝えることができることです. これについてはあとで詳細に説明しましょう. ここでは関数をきれいに出力する (*pretty print*) 方法についてだけ述べておきましょう. 次の例がそれを示しております.

```
> QUARTERNION(2,3,0,1);
```

$$\mathrm{QUARTERNION}(2,3,0,1)$$

```
> 'print/QUARTERNION' := proc(a,b,c,d) a + b*'i' + c*'j' + d*'k' end:

> QUARTERNION(2,3,0,1);
```

$$2+3i+k$$

ここでは出力用手続きすなわち出力用サブルーチンを定義しました. このルーチンは, Maple から得られた異なる `QUARTERNION` ごとに 1 度ずつ呼び出され, その結果を表示するものです. 手続きの中にクォートを使っていることに注意して下さい. これは出力の中に名前 $i, j, k,$ が欲しいのであって, 他で何かに使われている変数 i, j, k の値が欲しいわけではないからです.

レコードを表す第 3 の方法は, それを体の名前で表した多変数の多項式と考え, その係数の値を保存する

ことです. これは体が数体で, その上で算術演算をしたいときにとくに役立ちます. たとえば, 上で述べた出力表示のように四元数を変数 i, j, k の多項式として表します. すなわち,

```
> z1 := 2 + 3*i + k;
```

$$z1 := 2 + 3i + k$$

```
> z2 := 2 - 3*i + 2*j + 2*k;
```

$$z2 := 2 - 3i + 2j + 2k$$

```
> coeff( z1, i ); # the coefficient in i
```

$$3$$

```
> z1 + z2;
```

$$4 + 3k + 2j$$

これはすばらしいものに見えるけれども, i, j, k は繰返し文の変数として利用するので, i, j, k の名前を使うことは勧められません!

2.6 結合型リスト

Maple リストは結合型リストではありません. Maple リストは実はそのエントリのポインタの配列です. Maple リストと配列の違う点はリストは読み込みだけです. すなわち Maple リストの要素には割当てができません. 結合型リストは再帰的データ構造です. この差異は簡単な例題を調べればわかります. 多項式 $a(x) = \sum_{i=0}^{n} a_i x^i$ を Maple で表示することを考えましょう. 1 つの方法は係数 $a_0, \cdots, a_n$ を Maple リストに保存することです. たとえば, 多項式 $p = x^4 + 3x^2 + 2x + 11$ を次のように表します.

```
[11, 2, 3, 0, 1]
```

多項式の次数はリストの中のエントリの (個数 -1) となります. もう 1 つの別の方法は結合型リストを使用することです.

```
[ 1, [ 0, [ 3, [ 2, [ 11, NIL ]]]]]
```

この結合型リストは再帰的データ構造であることがわかります. これは Lisp プログラム言語の用語で伝統的に *CAR* と *CDR* と呼ばれている 2 つの値をもつリストか, あるいは空の結合型リストを表す特別な値 NIL のいずれかとなります. 第 1 フィールドはデータの値を保持し, 我々の例では係数です. 第 2 フィールドは他の結合型リストのポインタで残りのデータの値を保持します. 結合型リストで表された多項式の次数を求めるには, 結合型リストの深さを計算する必要があります. p を結合型リストとすると, 次の繰返し文を使ってこれを行うことができます.

```
for n from 0 while p <> NIL do p := p[2] od;
```

さて, Maple リストの代わりに結合型リストを用いて多項式を表示する理由は何でしょうか? その主たる理由は結合型リストの先頭に新しいエントリを追加するのに, 線形時間でなく一定時間で行いたいからです. たとえば, 多項式 p に項 $5x^5$ を追加したいとしましょう. Maple リスト表現では次のようにして 6 個のエントリをもつ新リストを作らなければなりません.

```
[op(p),5];
```

この新しい Maple リストには少なくとも 6 ワードの記憶場所が必要です. こんな馬鹿なことはしないで下さい. `op` 呼び出しは `op(1..nops(p),p)` を短くしたもので, Maple リスト p のすべてのエントリの式列を作ります. これは一定時間で実行でき記憶場所を必要としませんが, 今度は式列 `(11,2,3,0,1),5` となり, これが作られるべきより長い新しい式列 `11,2,3,0,1,5` となります. この場合 6 ワードの記憶場所が割当てに必要となります. 一般に, 次数 n の多項式に $a_{n+1}x^{n+1}$ を追加するのに $O(n)$ の処理時間と記憶場所が必要です. しかし結合型リストの場合はどうでしょうか？ 項 $5x^5$ を追加するには, 次のようにします.

```
[5,p];
```

これは長さが 2 の新しい Maple リストが必要となるだけなので, 一定の記憶場所しか必要としません. p が大域変数ならば線形の時間が必要となりますが, p が手続き内の局所変数やパラメータならば一定の時間しかかかりません. この評価問題については手続きの節で説明します. いまのところ, この操作の実行時間も結合型リストの場合 $O(1)$ であると仮定しておきます.

> **Lisp プログラマーに注意**: Maple リストの要素は変更できません. REPLACA や REPLACDR と同じものはありません. Maple のリスト, 集合, 式列, 関数はデータ構造を読み込むだけです. 配列と表のみが変更可能です.

3 Maple の手続き

3.1 引数, 局所変数, RETURN, ERROR

Maple の手続きは次の構文をとります.

proc (*nameseq*)
[**local** *nameseq* ;]
[**option** *nameseq* ;]
statseq
end

ここで *nameseq* はコンマで区切られた記号の列で, *statseq* はセミコロンで区切られた文の列です. ここに与えられた x, y に対して x^2+y^2 を求める簡単な手続きがあります.

```
proc(x, y)  x^2+y^2  end
```

この手続きは 2 つの引数 x と y を持っています. 局所変数やオプションがなく, 1 個の文があるだけです. この手続きによって返される値は x^2+y^2 です. 一般に手続きによって返される値は, 明示的な `return` 文がなければ最後に計算された値となります. 明示的な `return` 文をもつ手続きの 1 例は次の `MEMBER` 手続きです. `MEMBER(x,L)` は x がリスト L の中にあれば true を返し, なければ false を返します.

```
MEMBER := proc(x, L) local v;
   for v in L do if v=x then RETURN(true) fi od;
```

```
    false
end;
```

この MEMBER 手続きは局所変数 v を持ち, 大域的なユーザ変数 v と無関係です. 手続きの中にでてくる変数は引数でもなく局所変数でもなければ大域変数です.

ERROR 関数は手続き内のエラーメッセージを作るのに用いられます. たとえば, 次の MEMBER ルーチンは引数が実際にリストであるかどうかを検査し, そうでなければ適切なエラーメッセージを出します.

```
MEMBER := proc(x, L)  local v;
   if not type(L, list) then ERROR('2nd argument must be a list') fi;
   for v in L do if v=x then RETURN(true) fi od;
   false
end;
```

Maple V Release 2[*2)] を利用すれば, 簡単に引数検査を行うもっと効率的なプログラムが書けます.

```
MEMBER := proc(x, L::list)  local v;
   for v in L do if v=x then RETURN(true) fi od;
   false
end;
```

これは Pascal や C のような型宣言ではないことに注意して下さい. これはすぐ上の MEMBER のプログラムで明示的に書いた型検査を実行時に行うようにした短縮版なのです. しかもプログラムのエラーを追跡するときに非常に役に立つエラーメッセージも出してくれます. たとえば次のとおりです.

```
> MEMBER([1, x, x^2], x);
Error, MEMBER expects its 2nd argument, L, to be of type list, but received x
```

3.2 手続きの実行のトレース： printlevel

ここにユークリッドアルゴリズムを用いて, 2 つの非負の整数の最大公約数を求める Maple 手続きがあります. たとえば 3 は 2 つの整数 21 と 15 を割り切る最大の整数ですから, 21 と 15 の最大公約数は 3 です. ところで, ユークリッド法は数学におけるもっとも古くから知られているアルゴリズムの 1 つです. これは紀元前 300 年頃までさかのぼります.

```
GCD :=proc(a, b)  local c,d,r;
   c := a;
   d := b;
   while d<>0 do r:=irem(c, d); c:=d; d:=r od;
   c
end;
```

irem 関数は 2 つの整数の整数剰余 (***i****nteger* ***rem****ainder*) を計算するものです. この GCD ルーチンは実際にうまく動くでしょうか？ 手続きの実行を調べるもっとも簡単な道具は printlevel 機能です. printlevel 変数は初期値として 1 が割当てられている大域変数です. これをもっと高い値に設定すれば, すべての割当て文や手続きへの入口と出口のトレースが出力されます. GCD(21, 15) を計算したときにどんなことが

[*2)] (訳注) 手続き宣言の中の「:」が *Maple V Release 4* から「::」に改められました.

起こるか見てみましょう.

```
> printlevel := 100:
> GCD(21, 15);
```

{--> enter GCD, args = 21, 15

$$c := 21$$
$$d := 15$$
$$r := 6$$
$$c := 15$$
$$d := 6$$
$$r := 3$$
$$c := 6$$
$$d := 3$$
$$r := 0$$
$$c := 3$$
$$d := 0$$
$$3$$

<-- exit GCD (now at top level) = 3 }

$$3$$

GCD 手続きへの入力引数が返される値と一緒に表示されていることがわかります. 各割当て文の実行結果も表示されております.

Maple GCD 手続きに関する興味深い点は, Maple が任意精度の整数に関する四則演算を使っているから, このルーチンは任意の大きさの整数に対して働くことです. たとえば, 100! と 2^{100} の最大公約数が求められます.

```
> GCD(100!, 2^100);
```

$$15845632502852867518708790067 2$$

この GCD 手続きは次のように再帰的にも書けるし, 入力引数の型検査も含めることができます.

```
GCD := proc(a::integer, b::integer)
   if b=0 then a else GCD(b, irem(a,b)) fi
end;
```

この再帰版は簡単なので理解しやすいでしょう. printlevel 機能が計算の流れをどのように示すのかを見るために, この再帰版のトレースを見てみましょう.

```
> GCD(15, 21);
```

{--> enter GCD, args = 15, 21

{--> enter GCD, args = 21, 15

{--> enter GCD, args = 15, 6

{--> enter GCD, args = 6, 3

```
{--> enter GCD, args = 3, 0

                                    3

<-- exit GCD (now in GCD) = 3 }

                                    3

<-- exit GCD (now in GCD) = 3 }

                                    3

<-- exit GCD (now in GCD) = 3 }

                                    3

<-- exit GCD (now in GCD) = 3 }

                                    3

<-- exit GCD (now at top level) = 3 }

                                    3
```

3.3 矢印演算子

数式を計算するだけの手続きの場合は, 矢印構文と呼ばれる別の構文があります. これは代数でよく用いられている関数の構文を真似したものです. 1 引数の関数の場合, この構文は次のとおりです.

$$(\mathit{symbol}) -> [\textbf{local}\ \mathit{nameseq};]\ \mathit{expr}.$$

引数が 0 個以上あるときは, 引数を括弧で囲んでおきます.

$$(\mathit{nameseq}) -> [\textbf{local}\ \mathit{nameseq};]\ \mathit{expr}$$

x^2+y^2 を計算する例は次のように書けば十分です.

```
(x, y) -> x^2+y^2;
```

Maple V Release 2[*3)] では, この構文は拡張されて手続き本体が **if** 文でもよいことになっています. したがって区分的多項式をたとえば次のように定義することができます.

```
x -> if x<0 then 0 elif x<1 then x elif x<2 then 2-x else 0 fi;
```

3.4 有効範囲の規則： 引数, 局所変数, 大域変数

Maple では, 入れ子になった手続きが許されています. たとえば, 次のように書くことができます.

```
f1 := proc(x)  local g; g:=x->x+1; x*g(x) end;
```

[*3)](訳注) `piecewise` 文を使うと x -> piecewise (x<0, 0, x<1, x, x<2, x-2, 0) と書けます.

手続き f_1 は手続きとなる局所変数 g を持っています. f_1 は $x*(x+1)$ を計算しております. しかし, 入れ子型引数と局所変数は入れ子型有効範囲の規則を使用しておりません. たとえば, 上の手続きは次の手続きと同じではありません.

```
f2 := proc(x)  local g; g:=()->x+1; x*g() end;
```

なぜなら, g の手続き内の x の参照は, f_2 の引数 x を参照することにならないからです. これは大域変数 x を参照しているのです. 次の例を考えてみましょう.

```
> f1(a);
```

$$a(a+1)$$

```
> f2(a);
```

$$a(x+1)$$

```
> x := 7;
```

$$x := 7$$

```
> f2(a);
```

$$8a$$

同様に外部の有効範囲にある局所変数を参照できません.

> **コンピュータ科学者への注意**: Maple では入れ子型手続きが許され, これは関数の値として返されますが, 入れ子型辞書式の有効範囲を認めていないので, その最終結果を直接に返すことはできません. たとえば, `f ->x ->f(x+1)` はシフト演算子を定義しておりません. これは Maple の将来のバージョンとみなされております.

3.5 評価規則： 実引数と仮引数, 局所変数, 大域変数

関数呼び出し `f(x1,x2,...,xn)` を考えてみましょう. この関数呼び出しの実行は次のように行われます. 関数名 f が評価されます. 次に引数 $x1, x2, \cdots, xn$ が左から右へ評価されます. それから f が手続きであると決まれば, この手続きが評価された引数に基づいて実行されます. この評価規則には `eval, assigned, seq` を含めて 6 個だけ例外があります. それでは手続き内の変数の評価はどうなるでしょうか？ 次の手続きを考えてみましょう.

```
f := proc()  local p; p:=x^2+4*x+4; x:=5; p end;
```

ここで p は局所変数で, 引数はありません. しかし変数 x は何でしょうか？ 「**はじめに**」の節で次の問題を考えました.

```
> p := x^2+4*x+4;
```

$$p := x^2+4x+4$$

```
> x := 3;
```

$$x := 3$$

```
> p;
```

$$25$$

この場合は p と x は大域変数です. 大域変数は完全評価すなわち再帰的に評価されるので, その結果は 25 となります. 上の手続きのように p が局所変数のときはどうなるでしょうか？ すなわち,

```
f( );
```

を実行した結果どうなるでしょうか？

次の手続きのように p が引数のときはどうなるでしょうか？

```
> x := 'x';        # Make x a symbol first
```

$$x := x$$

```
> g := proc(p)  x:=5; p end:
> g(x^2+4*x+4);
```

$$x^2+4x+4$$

効率と好みの理由から, Maple の設計者は局所変数と引数は 1 レベル評価を行うことに決めました. 大域変数のみ完全評価を行います. `eval` 関数は局所変数と引数を完全評価するために用いられ, 大域変数の 1 レベル評価は必要なときに行うことにしております. たとえば, 次のとおりです.

```
> x := 'x';        # Make x a symbol first
```

$$x := x$$

```
> g := proc(p)  global x; x:=5; eval(p) end:
> g(x^2+4*x+4);
```

$$49$$

3.6 漸化式とオプション記憶

フィボナッチ数 F_n は線形漸化式 $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ によって定義されます. これは直接, 次のようにプログラム化されます.

```
F := proc(n)
   if n = 0 then 0 elif n = 1 then 1 else F(n-1)+F(n-2) fi
end;
```

これで最初の数個のフィボナッチ数が得られます.

```
> seq( F(i), i=0..10 );
```

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55$$

しかし, これはフィボナッチ数を求める効率的方法ではありません. 実際, もっとも速いコンピュータでもこの手続きを用いて `F(100)` を求めることはできないでしょう. もし手続き `F` の呼び出される回数を数えると, 同じ引数が繰返し呼び出されているのがわかるでしょう. 次の値を計算するとき, それより前の 2 つの値を記憶しておいたほうがよいのは明らかでしょう. これは次のような繰返し文で行えます.

```
F := proc(n) local fnm1,fnm2,f;
   if n = 0 then RETURN(0) fi;
   fnm2 := 0;
   fnm1 := 1;
   for i to n-1 do f := fnm1 + fnm2; fnm2 := fnm1; fnm1 := f od;
   fnm1
end;
```

これをプログラム化する別の方法はオプション記憶 (*option remember*) を用いることです. このオプションは, 必要な時に利用できるように, 計算されたときの値を保存しておくために用いられます. 次の手続きを考えてみましょう.

```
F := proc(n) option remember;
   if n = 0 then 0 elif n = 1 then 1 else F(n-1)+F(n) fi
end;
```

このプログラムは `F(100)` を素早く計算します. 各 Maple 手続きはそれぞれ付随する記憶 (*remember*) 表を持っております. この表の指標は引数で, 表の項目は関数の値です. n のとき `F` が呼び出されると, Maple はまず `F(n)` がすでに計算されているかどうかを調べるために `F` の記憶表 (*remember table*) を調べます. もしその値があれば, `F` の記憶表からその結果を取り出して返します. もしその値がなければ, 手続き `F` のプログラムを実行し, 自動的に n と `F(n)` を対にして `F` の記憶表に保存します.

いわゆる関数割当て (*functional assignment*) を用いて, 関数値を明示的に記憶表に保存する方法も説明しましておきましょう. この方法は記憶表の中に指定した値のみ保存するので, オプション記憶よりも柔軟です.

```
F := proc(n) F(n) := F(n-1)+F(n-2) end;
F(0) := 0;
F(1) := 1;
```

3.7 型と一律操作

`type` 関数は, 入力の型に応じて違う動作をするルーチンを書くときに用います. たとえば, 次の `DIFF` ルーチンは与えられた変数 x の多項式で表された数式を微分するものです.

```
DIFF := proc( a::algebraic, x::name) local u,v;
   if type(a,numeric) then 0
   elif type(a,name) then if a = x then 1 else 0 fi
   elif type(a,'+') then map( DIFF, a, x )
```

```
    elif type(a,'*') then u := op(1,a); v := a/u;
       DIFF(u,x)*v + DIFF(v,x)*u
    elif type(a,anything^integer) then
       u := op(1,a); v := op(2,a); v*DIFF(u,x)*u^(v-1)
    else ERROR('don't know how to differentiate',a)
    fi
end;
```

DIFF の手続きの中で型は異なる 2 つの目的で用いられております. 第 1 の用法は型の検査です. 入力は必ず代数的な記号式や数式でなければならないし, 微分する変数は記号名でなければなりません. 第 2 の用法は入力の型を調べて, それが和か積かに従ってどんな微分規則が適用できるかを決定することです. この DIFF の例は非常に有用な関数である map 関数の使い方を示しているので, ここでこれを説明しましょう. これは次の構文をとります.

$$\mathrm{map}\left(f, a, x_1, \cdots, x_n \right)$$

この意味は, 関数 f に補助的な引数 $x_1, \cdots, x_n$ を渡して, 数式 a のオペランドに関数 f を適用するものです. 上の DIFF の手続きでは 1 個の補助引数 x があります. しかし, たいていは補助引数がありません. 形式的には, これは次の列を計算するのと同値で,

```
seq( f( op(1,a), x1, ··· , xn ), i=1..nops(a) );
```

この列から f の型と同じ型の値を作るものです. 次に例をいくつか挙げておきます.

```
> p := x^3+2*x+1;
```

$$p := x^3 + 2x + 1$$

```
> map( F, p );
```

$$F(x^3) + F(2x) + F(1)$$

```
> map( x -> x^2, p );
```

$$x^6 + 4x^2 + 1$$

```
> map( degree, p, x );
```

$$4$$

DIFF 関数はまた構造的 (*structured*) な型の使い方も示しております. 型 anything, name, '+', '*' は単純型です. 型 anything ^ integer は構造的な型です. これは値がベキで, その底は何でもよくすなわち任意の型でよいが, 指数部は整数でなければいけないことを意味しています. これは次のように書いたのと同値です.

```
if type(a,'^') and type(op(2,a),integer) then
```

構造的な型によって, 長い型の検査を簡潔な検査でおきかえることができます. 他のよくある例で説明しましょう. 多くのルーチンでは引数として, 記号名や数式の集合やリストを用います. たとえば, solve コマンドはいくつかの未知数に関する連立方程式を解くもので, たとえば次のようになります.

```
> solve( {x+y=2, x-y=3}, {x,y} );
```

$$\left\{y = -\frac{1}{2}, x = \frac{5}{2}\right\}$$

ゼロ個以上の方程式が型 `set(equation)` で検査でき, ゼロ個の未知数の集合は型 `set(name)` で検査できます. しかし, `solve` コマンドでは暗黙的にゼロに等しい代数式を解くことができます. すなわち上の例は次のように入力することができます.

```
> solve( {x+y-2, x-y-3}, {x,y} );
```

$$\left\{y = -\frac{1}{2}, x = \frac{5}{2}\right\}$$

したがって, 第 1 引数の型は方程式の集合か代数式の集合のいずれかで,
型 {`set(algebraic)`, `set(equation)`} となります. これは `set({algebraic,equation})` と同じでないことに注意してください. なぜでしょうか?

これらと異なる型に関する情報は, オンラインヘルプから `?type` で得られます.

3.8 引数の個数の変数 : `args` と`nargs`

Maple では関数の引数の個数を変数とする関数があります. そのような関数の 1 例が `max` 関数です. ここにその関数をプログラム化した最初の試みがあります.

```
MAX := proc(x1) local maximum,i;
   maximum := x1;
   for i from 2 to nargs do
      if args[i] > maximum then maximum := args[i] fi
   od;
   maximum
end;
```

特別な変数である `nargs` は引数の個数で, 変数 `args` は引数の列です. したがって `args[i]` は i 番目の引数です.

3.9 結果を未評価で返すこと

いま書いた `MAX` の手続きは, 数字の引数に対してのみに働きます. Maple 関数 `max` 試してみれば, 記号引数でも動作することがわかるでしょう. 次の文を見て下さい.

```
> MAX(1,2,x);
Error, (in MAX) cannot evaluate boolean
> max(1,2,x);
```

$$\max(2, x)$$

Maple は非数値に対して `args[i] < maximum` かどうかは計算できないので, 手続き `MAX` は実行できません. 数字がいくつかあるときの MAX はその答えを計算し, そうでないときは `MAX(x,y)` を `MAX(x,y)` のままにしておきたい, そうすれば `sin(x)` を `sin(x)` のままにしておくように `MAX(x,y)` を記号的に

計算できます.

さらにいま書いた MAX 手続きは, numeric 型の数字に対してのみ働きます. しかし, MAX が $\pi > \sqrt{2}$ を理解すれば素敵でしょう. このような MAX 関数を作るのに役立つのは, 2 つの実数の強力な比較を与える signum 関数を用いることです. signum 関数は $x < 0$ ならば -1 を返し, $x \geq 0$ ならば $+1$ を返し, それ以外は未評価 (*unevaluated*) で, すなわち, signum(x) を返します. たとえば次のようになります.

```
> signum(sqrt(2)-1);
```

$$1$$

```
> signum(sqrt(2)-Pi);
```

$$-1$$

```
> signum(a-b);
```

$$\mathrm{signum}(a-b)$$

この signum 関数を使って, MAX 関数をもっとスマートにし, 記号引数も扱えるようにしましょう.

```
MAX := proc() local a,i,j,n,s;
   n := nargs;
   # First, put the arguments in an array
   a := array(1..n);
   for i to n do a[i] := args[i] od;
   # Compare a[i] with a[j] for 1 <= i < j <= n
   i := 1;
   while i < n do
      j := i+1;
      while j <= n do
         s := signum(a[i]-a[j]);
         if s = 1 then # i.e.  a[i] >= a[j]
            a[j] := a[n]; n := n-1;
         elif s = -1 then # i.e.  a[i] < a[j]
            a[i] := a[j]; a[j] := a[n]; j := n; n := n-1; i := i-1;
         else # cannot determine the sign
            j := j+1
         fi
      od;
      i := i+1
   od;
   if n = 1 then RETURN(a[1]) fi;
   'MAX'( seq(a[i], i=1..n) );
end;
```

上記のコードでもっとも面白いのは最後の行です. バッククォート ' は実行中に MAX 関数の呼び出しを防止するもので, 無限ループに落ち入らないようにしています. その代わりに未評価の関数呼び出し MAX (. . .) が返され, 最大値が計算できなかったことを示します. しかし, 若干の簡単化はたとえば次のように行ないま

す.

```
> MAX( x, 1, sqrt(2), x+1 );
```

$$\mathrm{MAX}\left(x+1,\sqrt{2}\right)$$

3.10 簡単化と変換則

代数的に記述されるすなわち変換則によって行なわれる簡単化を導入したいことがよくあります. たとえば, 関数 f が与えられていて, f が可換で結合的であるとしましょう. 実際 max はそのような関数です. すなわち, $\max(a,b)=\max(b,a)$ で $\max(a,\max(b,c))=\max(\max(a,b),c)=\max(a,b,c)$ が成り立ちます. Maple ではこれらの性質をどのように実現しているでしょうか? 我々が欲しいのは, max 関数を含む数式を書く標準的 (*canonical*) 方法です. 引数は整列して可換性を実現できます. 結合性については入れ子になった max の呼び出しを非入れ子にします. すなわち, $\max(\max(a,b),c)$ と $\max(a,\max(b,c))$ の両方を $\max(a,b,c)$ に変換します. 実際は $\max(\max(a))=\max(a)$ すなわち max をベキ等にしても実現できます. 次の MAX 関数はその通りにしております.

```
MAX := proc() local a;
   a := [args];
   a := map( flatten, a, MAX ); # unnest nested MAX calls
   'MAX'( op(sort(a)) );
end;
flatten := proc(x,f)
   if type(x,function) and op(0,x) = f then op(x) else x fi
end;
```

たとえば次のようになります.

```
> MAX(a,MAX(c,b),a);
```

$$\mathrm{MAX}(a,a,b,c)$$

$\max(a,a)=a$ という性質も認識できるようにすべきであることもわかります. こうするには引数をリストに入れる代わりに, 集合に入れれば重複した要素は除かれます. また集合は自動的に整列するから, 整列の呼び出しも省略できることになります. したがって次のようになります.

```
MAX := proc() local a;
   a := {args};
   a := map( flatten, a, MAX );
   'MAX'( op(a) );
end;
```

```
> MAX(a,MAX(c,b),a);
```

$$\mathrm{MAX}(a,b,c)$$

読者は我々の MAX 手続きがしていることに少し当惑しているでしょう. `printlevel` 変数に正の整数を割り当てれば, 実行されるすべての文を追跡できることはずっと前からわかっていました. しかし, この簡

単な追跡機能を使って得られる出力は, たいていの場合多すぎます. この場合, `flatten` 手続きからも出力が得られます. `trace` 関数はそれらと違って手続きを選別して追跡するのに用いられます. これを使って `MAX` 手続きを追跡してみましょう.

```
> trace(MAX);
```

$$MAX$$

```
> MAX(a,MAX(b,a),c);
```

```
{--> enter MAX, args = b, a
```

$$a := \{a, b\}$$
$$a := \{a, b\}$$
$$\mathrm{MAX}(a, b)$$

```
<-- exit MAX (now at top level) = MAX(a,b) }
{--> enter MAX, args = a, MAX(a,b), c
```

$$a := \{a, c, \mathrm{MAX}(a, b)\}$$
$$a := \{a, b, c\}$$
$$\mathrm{MAX}(a, b, c)$$

```
<-- exit MAX (now at top level) = MAX(a,c,b) }
```

$$\mathrm{MAX}(a, b, c)$$

3.11 オプション引数とデフォルト値

多くの Maple ルーチンはオプション引数を受け入れます. これを用いてユーザはすべての引数の値を決めることはしないで, デフォルト値を使うことがよくあります. 例として関数 `plot`, `factor`, `collect`, `series` が挙げられます. `degree` 関数を考えてみましょう. `degree` 関数は 1 変数の単一な多項式はその次数を求め, 多変数の多項式は次数の和を計算するもので, たとえば次のようになります.

```
> p := x^3+2*x+1;
```

$$p := x^3 + 2x + 1$$

```
> degree(p);
```

$$3$$

```
> q := 3*x^2*y+2*y^2-x*z+7;
```

$$q := 3x^2y + 2y^2 - xz + 7$$

```
> degree(q);
```

$$3$$

ときには特定の変数たとえば x の次数を求めたいことがあるでしょう. これは degree 関数にオプションの第 2 引数を指定してできます. たとえば次のようになります.

```
> degree(p,x);
```

$$3$$

```
> degree(q,x);
```

$$2$$

この degree 関数はどのようにプログラム化するのでしょうか？ 入力は数式で, 第 2 のオプション引数が与えられたときは変数の名前か名前の集合であると仮定しましょう. そうすると次のように書けます.

```
DEGREE := proc( a::algebraic, x::{name, set(name)}) local s,t;
   if nargs = 1 then # determine the variable(s) for the user
      s := indets(a); # the set of all the variables of a
      if not type(s,set(name)) then ERROR(‘input not a polynomial‘) fi;
      DEGREE(a,s)
   elif type(a,constant) then 0
   elif type(a,name) then
      if type(x,name) then if a = x then 1 else 0 fi
      else if member(a,x) then 1 else 0 fi
      fi
   elif type(a,‘+‘) then max( seq( DEGREE(t,x), t=a ) )
   elif type(a,‘*‘) then
      s := 0;
      for t in a do s := s + DEGREE(t,x) od;
      s
   elif type(a,algebraic^integer) then DEGREE(op(1,a),x) * op(2,a)
   else ERROR(‘cannot compute degree‘)
   fi
end;
```

ここで用いられている indets 関数は, 入力に現れた未知数 (*indeterminates*) (すなわち変数) の集合を返すものです. ここで用いられている各規則を研究し, それぞれがどのような順序で実行されるかを調べることを読者の宿題に残しておきましょう.

3.12 引数を通して結果を返すこと

Maple の多くの関数は 1 つ以上の値を返します. もちろん, 1 つ以上の値を列やリストで返すことは常に可能です. しかし, 他のプログラミング言語と同様に引数を通して値を返すこともできて, しばしばこれが便利になります. たとえば, 多項式の長割算を行う divide 関数を考えてみましょう. divide(a,b) の呼び出しは, 多項式 b が多項式 a を割り切って剰余がないとき, そのときに限って true を返します. たとえば次のようになります.

```
> divide(x^3-1,x-1);
```

$$true$$

しかし通常は, a を b で割って, 商 q で何かをしたいでしょう. Maple では次のようにこれを実行します. a を b で割ったときの商を割り当てる記号名を, 第 3 引数としてたとえば次のように `divide` 関数に与えます.

```
> if divide(x^3-1,x-1, 'q' ) then print(q) fi;
```

$$x^2+x+1$$

`divide` 関数には q の値でなく記号名 q を渡すために, クォートを用いていることに注意して下さい. 他の例を採り上げ, オプション引数に値を割り当てるプログラムの書き方を研究しましょう. 値 x がリスト L にあるかどうかを検査する `MEMBER` 関数を考えてみましょう. 我々の関数を修正して, `MEMBER(x, L, 'p')` はリスト L の中に x があるかどうかを返し, さらに記号名 p には L 中に最初に現れた x の位置を割り当てるとしましょう.

```
MEMBER := proc(x, L::list,p::name) local i;
   for i to nops(L) do
      if x = L[i] then
         if nargs = 3 then p := i fi;
         RETURN(true)
      fi
   od;
   false
end;
```

次に例をあげましょう.

```
> MEMBER(4,[1,3,5],'position');
```

$$false$$

```
> position;
```

$$position$$

```
> MEMBER(3,[1,3,5],'position');
```

$$true$$

```
> position;
```

$$2$$

`MEMBER` 手続き内の仮引数 p の割当文の効果は, 実引数 `position` に割当られることがわかります.

4 Maple のプログラミング

4.1 Maple の行列・ベクトル計算

Maple のベクトルは添数 1 から始まる 1 次元配列で表わされ, 行列は 1 から始まる行と列の添数を持つ 2 次元配列で表わされます. ここで 5×5 ヒルベルト行列を作る 1 つの方法をお見せしましょう. ヒルベルト

行列は対称行列で, その (i,j) 要素が $1/(i+j-1)$ であることを思い出して下さい.

```
> H := array(1..5,1..5):
> for i to 5 do for j to 5 do H[i,j] := 1/(i+j-1) od od;
> H;
```

$$H$$

```
> eval(H);
```

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}$$

注意: H の値は行列 H の記号名そのものであることに注意して下さい. 配列の評価規則, したがって行列とベクトルの評価規則は特別な規則です. それは技術的な理由からです. 当分の間, 行列やベクトルを出力したいときはいつも `eval` 関数を用いて下さい.

`linalg` パッケージには, Maple でベクトルや行列を計算するための多くの関数が入っております. 上の行列はこの線形代数パッケージの中の `matrix` コマンドを用いて, 次のように作ることもできます.

```
linalg[matrix](5,5,(i,j) -> 1/(i+j-1));
```

Maple は行列 H の行列式や逆行列を求めることができるし, 他の行列演算も行うことができます. 演算のリストについては `?linalg` を参照して下さい.

ガウス消去法を用いて行列のいわゆる行操作を行うプログラムをここでお見せしましょう. `GaussianElimination(A,'r')` はそれによって得られる上三角行列を計算します. これはオプションの第 2 引数も持っていて, これが与えられるとそれにその行列の階数が割当てられます.

```
GaussianElimination := proc(A::matrix(rational),rank::name)
local m,n,i,j,B,r,c,t;

   m := linalg[rowdim](A); # the number of rows of the matrix
   n := linalg[coldim](A); # the number of colums of the matrix

   B := array(1..m,1..n);
   for i to m do for j to n do B[i,j] := A[i,j] od od;

   r := 1; # r and c are row and column indices
   for c to n while r <= m do
      for i from r to m while B[i,c] = 0 do od; # search for a pivot
      if i <= m then
         if i <> r then # interchange row i with row r
            for j from c to n do
               t := B[i,j]; B[i,j] := B[r,j]; B[r,j] := t
```

```
            od
         fi;
         for i from r+1 to m do
            if B[i,c] <> 0 then
               t := B[i,c]/B[r,c];
               for j from c+1 to n do B[i,j] := B[i,j]-t*B[r,j] od;
               B[i,c] := 0
            fi
         od;
         r := r + 1                    # go to next row
      fi
   od;                    # go to next column

   if nargs>1 then rank := r-1 fi;
   eval(B)

end:
```

型 `matrix(rational)` は行列 (Maple の 2 次元配列) の要素をすべて有理数であることを指定しております.

4.2 Maple の数値計算

Maple の浮動小数点数は小数点数として入力するか, あるいは直接に `Float` 関数を用いて入力します.

$$\texttt{Float(m,e)} = m * 10^e$$

ここで仮数 m は任意の大きさの整数で, 指数 e はマシンサイズの整数に限定されて通常は 31 ビットです. たとえば, 数値 3.1 は `3.1` と入力されるか `Float(31,-1)` と入力されます. `op` 関数は浮動小数点数の仮数や指数の抽出に用いることができます. 注意 : 浮動小数点定数 0.0 は特別に扱われて 0 と簡単化されます, つまり自動的に整数 0 になります.

ここで範囲 $[0,1)$ 内で正確に 6 桁の精度の小数となる一様乱数を生成する一様乱数発生プログラムをお見せします.

```
> UniformInteger := rand(0..10^6-1):
> UniformFloat := proc() Float(UniformInteger(),-6) end:
> seq( UniformFloat(), i=1..6 );
```

.669081, .693270, .073697, .143563, .718976, .830538

Maple 組込み関数 `rand` は与えられた範囲内の乱数を返さないことに注意して下さい. その代わりに, これは呼び出されたときに与えられた範囲内のランダムな整数を返す Maple 手続き (乱数発生プログラム) を与えます.

小数に丸めるときは浮動小数点演算が行われます. 使用される精度は大域変数 `Digits` で制御され, このデフォルト値は 10 です. しかし任意の値に設定もできます. `evalf` 関数は正確な記号定数を浮動小数点近似するのに用いられます. たとえば次のようになります.

```
> Digits := 25:
```

```
> sin(1.0);
```

$$.841470984807896506652502 3$$

```
> sin(1);
```

$$\sin(1)$$

```
> evalf(%);
```

$$.841470984807896506652502 3$$

Maple は初等関数や $J_v(x), \Gamma(x), \zeta(x)$ のような特殊関数について知っています. Maple ではこれらはそれぞれ `BesselJ(v,x)`, `GAMMA(x)`, `Zeta(x)` で表わします. これはいずれもいろいろな数列の和を計算し高精度まで求められます.

使用される浮動小数点演算モデルでは, 得られた結果の相対誤差は $10^{1-\mathrm{Digits}}$ 以下となります. これはハードウェアで浮動小数点演算を実現している方式よりも強力で, しかもこのような精度の結果を得るにはそれは中間計算を高精度で行う必要があります. ここにテーラー級数の和を求める例があります. 小さな x に対して次の誤差関数

$$\mathrm{erf}\,(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,\mathrm{d}t$$

を計算したいとしましょう. 誤差関数について良く知らなければ, Maple でこれを少し作図してみて下さい. 小さな x, $-1 < x < 1$ に対して $\mathrm{erf}(x)$ を計算するには, $x = 0$ のまわりの $\mathrm{erf}(x)$ のテーラー級数を用いて,

$$\frac{\sqrt{\pi}}{2}\mathrm{erf}(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{n!\,(2n+1)} = x - \frac{x^3}{3} + \frac{x^5}{10} - \frac{x^7}{42} + \frac{x^9}{216} - \cdots$$

(エラーチェックをしないと) 次のように書けます.

```
ErfSmall := proc(a) local n,x,x2,result,term,sumold,sumnew;
   x := evalf(a); # evaluate the input at Digits precision
   Digits := Digits + 2; # add some guard digits
   sumold := 0;
   term := x;
   sumnew := x;
   x2 := x^2;
   for n from 1 while sumold <> sumnew do
      sumold := sumnew;
      term := - term * x2 / n;
      sumnew := sumold + term / (2*n + 1);
   od;
   result := evalf( 2/sqrt(Pi)*sumnew );
   Digits := Digits-2;
   evalf(result) # round the result to Digits precision
end;
```

大きな x に対してはこのルーチンは不正確で効率が悪くなります. 大きな x に対して $\mathrm{erf}(x)$ を計算する方法

については練習問題を参照して下さい.

第 2 の例として, 整数の平方根を 20 桁まで計算するニュートン反復法を書くことを考えてみましょう. 方程式 $f(x)=0$ を解くニュートン反復法は,

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

で表わされることを思い出して下さい. いまの例では, $\sqrt{a}$ を計算するために $x^2-a=0$ を解くことになります. したがって反復法は

$$x_{k+1} = (x_k + \frac{a}{x_k})/2$$

となります. 次のルーチンはこの計算を行うものです.

```
SqrtNewton := proc(a::integer) local xk, xkm1;
   if a < 0 then ERROR('square root of a negative integer') fi;
   Digits := 55; # add some guard digits
   xkm1 := 0;
   xk := evalf(a/2); # initial floating point approximation
   while abs(xk-xkm1) > abs(xk)*10^(-50) do
      xkm1 := xk;
      print(xk);
      xk := (xk + a/xk)/2;
   od;
   Digits := 50;
   evalf(xk); # round the result to 50 Digits
end;
```

各反復法の結果を示すために `print` 文を入れておきました. 次にそれを示しましょう.

```
> SqrtNewton(2);
```

```
                                1.
1.5000000000000000000000000000000000000000000000000000000000
1.4166666666666666666666666666666666666666666666666666666666
1.4142156862745098039215686274509803921568627450980392156
1.4142135623746899106262955788901349101165596221157440456
1.4142135623730950488016896235205302436149819257761974296
1.4142135623730950488016887242096980785696718753772340025
1.4142135673730950488016887242096980785696718753769480731
1.4142135673730950488016887242096980785696718753769
```

上の例でわかるように, $f'(x_k)$ が零に近くなく x_k が f の根に十分近ければ, ニュートン反復法は 2 次収束することが知られています. 非常に高精度すなわち Digits $>$ 1000 の場合, 初期段階で最高精度になっていないから, 各反復を最高精度で行いたくないかも知れません. 少し桁数を修正するだけでも反復法はうまくゆくのでしょうか? `SqrtNewton` の手続きで用いられたニュートン反復法を, 各段階の桁数が倍になるように近似的に修正してみて下さい. こうするとこの反復法はどの位速くなるでしょうか?

Maple の浮動小数点方式はソフトウェアで実現されているので, ハードウェアの浮動小数点演算よりかな

り遅いのです. さらに Maple はハードウェア浮動小数点演算の評価に使う `evalhf` と呼ばれる関数を持っております. この関数は浮動小数点用の C ライブラリ組込みルーチンを使用します. だから Maple ソフトウェア浮動小数演算よりかなり速いのですが, これはコンパイルされないのでハードウェア浮動小数演算よりも遅くなります. 詳細については `?evalhf` を見て下さい.

4.3 Maple の多項式計算

多項式や有理関数の計算は Maple の長所です. ここで多項式のユークリッドノルムを計算するプログラムを採り上げましょう. これは多項式 $a(x)=\sum_{i=0}^{n} a_i x^i$ が与えられたとき $\sqrt{\sum_{i=0}^{n} a_i^2}$ を計算するものです.

```
EuclideanNorm := proc(a)
   sqrt( convert( map( x -> x^2, [coeffs( expand(a) )] ), '+') )
end;
```

このプログラムを内側から読んでいくと, まず入力された多項式が展開されます. `coeffs` 関数は係数の式列を返すのでそれをリストにします. このリストの各要素を 2 乗して新しいリストを作ります. それからこの 2 乗リストが和に変換されます.

なぜ多項式は展開されるのでしょうか? `coeff` 関数と `coeffs` 関数では入力多項式を必ず展開 (*expanded*) するのは, そうしないと係数が計算できないからです. 次の例でそれが明らかでしょう.

```
> p := x^3 - (x-3)*(x^2+x) + 1;
```

$$p := x^3 - (x-3)(x^2+x) + 1$$

```
> coeffs(p);
Error, invalid arguments to coeffs
> expand(p);
```

$$2x^2 + 3x + 1$$

```
> coeffs(expand(p));
```

$$2, 3, 1$$

```
> EuclideanNorm(p);
```

$$\sqrt{14}$$

> **注意**: 多項式のある変数に関する係数を求めたければ, 常に最初にその変数について多項式を展開したほうがよいでしょう. `expand` 関数を用いることができますが, これはすべての変数について多項式展開を行うものです. 別の方法として `collect` 関数を使うことができます. `collect(p,x)` は x に関して多項式 p を展開するものです. `?collect` で他の詳細な事項や多変数の多項式の扱い方を調べて下さい.

`EuclideanNorm` 手続きは数値係数の平方和の平方根を求めるという意味で, 多変数の多項式に対して動作するものです. たとえば多項式 $p = ux^2 + y^2 + v$ が与えられれば, `EuclideanNorm` 手続きは $\sqrt{3}$ を返

します. しかし, この多項式を x, y に関する多項式とみると, その係数は u, v の記号係数となります. 実際は多項式の変数が何であるかを, EuclideanNorm ルーチンに知らせることができればよいと思うでしょう. 次のように変数をオプション引数とすればそれができます.

```
EuclideanNorm := proc( a, v::{name,set(name),list(name)})
   if nargs = 1 then
      sqrt( convert( map( x -> x^2, [coeffs(expand(a))] ), '+' ) )
   else
      sqrt( convert( map( x -> x^2, [coeffs(expand(a),v)] ), '+') )
   fi
end;
```

型 {name, set(name), list(name)} は, 第 2 引数 v が単一の変数や変数の集合あるいは変数のリストでもよいことを示しております. coeffs 関数自身はこの引数を第 2 のオプション引数として受け付けることに注意して下さい. 最後にこのルーチンは入力が多項式でなくてもよいことにして, それを追加すると次のようになります.

```
EuclideanNorm := proc( a, v::{name,list(name),set(name)})
   if nargs = 1 then
      if not type(a,polynom) then
         ERROR('1st argument is not a polynomial',a) fi;
      sqrt( convert( map( x -> x^2, [coeffs(expand(a))] ), '+' ) )
   else
      if not type(a,polynom(anything,v)) then
         ERROR('1st argument is not a polynomial in',v) fi;
      sqrt( convert( map( x -> x^2, [coeffs(expand(a),v)] ), '+' ) )
   fi
end;
```

型 polynom は次の一般的な構文

$$\mathrm{polynom}(R, X)$$

を持っていて, これは係数が型 R の変数 X の多項式を表しています. 1 例が polynom(rational,x) で, これは有理係数を持つ x に関する 1 変数多項式すなわち $Q[x]$ の多項式を規定しております. R と X が指定されていなければ, 上の最初の例のように, その数式はすべての変数に関する多項式でなければいけません.

多項式の計算を行う基本的な関数は degree, coeff, expand, divide, collect です. Maple にはこれ以外にも多項式計算を行う関数が多くあり, 多項式の除算, 最大公約数, 終結式などの機能があります. さらに Maple は有限体を含むいろいろな数体の上で多項式を因子分解することもできます. 機能のリストは ?polynom のオンラインヘルプを参照して下さい.

有限体上で多項式計算を行う機能を説明するために, $GF(2)$ 上で次数 n の最初の原始 3 項式 (*primitive trinomial*) が存在するときそれを求めるプログラムを挙げて終わりにしましょう. すなわち, x が $GF(2)[x]/(a)$ の原始要素であるような $x^n + x^m + 1$ の形の既約多項式 a を求めたいのです. iquo 関数は 2 つの整数の整数商 (***i**nteger **quo**tient*) を計算するものです.

```
trinomial := proc( n::integer ) local i,t;
   for i to iquo(n+1,2) do
```

```
        t := x^n+x^i+1;
            if Primitive(t) mod 2 then RETURN(t) fi;
    od;
    FAIL
end;
```

4.4 手続きの読込みと保存：read とsave

対話しながら 1 行か 2 行の Maple プログラムを書けるでしょう. しかし大きなプログラムはファイルに保存したいでしょう. 普通はエディタを使って Maple プログラムを書きファイルに保存して, それを使う前に Maple に読込みます. プログラムは read コマンドを用いて Maple に読込むことができます. たとえば, Maple 手続き MAX をファイル MAX に書込んであるとすれば, 次のようにしてこのファイルを Maple に読込みます.

```
read MAX;
```

Maple セッションで計算された Maple 手続きや数式を, 次の型をした save コマンドを用いて Maple 内でファイルに保存することができます.

```
save f1, f2, ···, filename;
```

これは変数 f1, f2, ··· の値をテキスト形式でファイル filename に保存するものです. さらに Maple データを内部形式すなわちいわゆる “.m” 形式で保存することもできます. この形式は非常にコンパクトで Maple で読込むのが速くなります. 単に次のようにファイル名に “.m” を追加するだけです.

```
save f1, f2, ···, ‘filename.m‘;
```

これは f1, f2, ··· の値をファイル filename.m に保存するものです. これは read コマンドを用いると Maple に読み戻すことができます.

```
read ‘filename.m‘;
```

4.5 Maple プログラムのデバッグ

もっとも簡単なデバッグ用ツールは出力レベル (*printlevel*) 機能です. printlevel は大域変数で初期には 1 が割当てられております. これにもっと高い値を設定すれば, すべての割当て文, 手続きの入口と出口のトレースが出力されます. printlevel の値が高ければ高いほど, それだけ手続きの深い実行レベルがトレースされます. しかし, printlevel 機能から得られる結果はしばしば大量になります. trace 関数を使うと, 指定された関数の実行のみトレースすることができます. この 2 つのツールの使用例はすでにこの文書で書きました. ここで少し詳しくお話しましょう. printlevel 変数を 3 以上にすると, 実行時エラーが起きたとき, Maple はエラーが起きた時点で呼び出し列のスタックトレースを出力します. とくに, いま実行されているすべての手続きの引数およびエラーが起きた手続きで実行された局所変数の値と文を出力します. このことをもっとも簡単に説明する例を挙げておきます.

```
> f := proc(x) local y; y := 1; g(x,y); end:
> g := proc(u,v) local s,t; s := 0; t := v/s; s+t end:
> printlevel := 4:
```

```
> f(3);
 f called with arguments: 3
 #(f,2): g(X,Y)
 g called with arguments: X, Y
 #(g,2): t := v/s;
Error, (in g) division by zero
 locals defined as: s = 0, t = t
```

4.6 他の Maple 機能とのインタフェース

関数 f を手続きとしてプログラム化して, その性質を Maple に知らせる方法をこれまで述べてきました. これと違ったやり方を考えるために, f を含む数式を微分することを Maple に教えたいとしましょう. f を数値的に計算して作図できるようにする方法や f を含む式を簡単化する方法などを, Maple に教えたいと考えてもよいでしょう. どうすればよいでしょうか? 多くの Maple ルーチンは, これらの f 関数に関する仕事を教えることができるようなインタフェースを持っております. そのようなルーチンは `diff`, `evalf`, `expand`, `combine`, `simplify`, `series` などです. 関数 W を微分する方法を Maple に教えるには, `'diff/W'` と呼ばれるルーチンを書きます. この `diff` ルーチンが $W(g)$ を含む式 $f(x)$ で呼び出されると, `diff` ルーチンは `'diff/W'(g,x)` を呼び出して x に関する $W(g)$ の導関数を計算します. $W'(x) = W(x)/(1+W(x))$ であることを知っているとしましょう. そうするとこの連鎖則を明示的に示す次のような手続きを書くことができます.

```
'diff/W' := proc(g,x) diff(g,x) * W(g)/(1+W(g)) end;
```

そうすると次のようになります.

```
> diff(W(x),x);
```

$$\frac{W(x)}{1+W(x)}$$

```
> diff(W(x^2),x);
```

$$2\,\frac{xW(x^2)}{1+W(x^2)}$$

2 番目の例として, 第 1 種チェビシェフ多項式 $T_n(x)$ の数式操作を考えましょう. この多項式は Maple では `T(n,x)` と表すことができます. さらに特定の n の値に対して `T(n,x)` を x の多項式として展開したいとしましょう. Maple の `expand` 関数にルーチン `'expand/T'`を書いてその方法を教えることができます. `expand` は `T(n,x)` に逢うと, `'expand/T'(n,x)` を呼出します. $T_n(x)$ は線形漸化式 $T_n(0) = 1$, $T_n(1) = x$, $T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x)$ を満足していることを思い出して下さい. したがって次のように書けます.

```
'expand/T':= proc(n,x) option remember;
   if n = 0 then 1
   elif n = 1 then x
   elif not type(n,integer) then T(n,x) # can't do anything
   else expand(2*x*T(n-1,x) - T(n-2,x))
   fi
end;
```

このルーチンは再帰的ですが, remember オプションが用いられているので `T(100,x)` を非常に速く計算します. いくつか例を挙げましょう.

```
> T(4,x);
```

$$\mathrm{T}(4,x)$$

```
> expand(T(4,x));
```

$$8x^4 - 8x^2 + 1$$

```
> expand(T(100,x));
```

$$1 - 5000\,x^2 \cdots \text{途中省略} \cdots + 633825300114114700748351602688\,x^{100}$$

また関数 f を数値的に計算する方法を Maple に教えるために, `'evalf/f'(x)` が $f(x)$ を数値的に計算できるように Maple 手続き `'evalf/f'` を定義することもできます. たとえば, 前に示した数値の平方根を計算するニュートン反復法を使いたいとしましょう. この関数は次のようになるでしょう.

```
'evalf/Sqrt' := proc(a) local x,xk,xkm1;
   x := evalf(a); # evaluate the argument in floating point
   if not type(a,numeric) then RETURN( Sqrt(x) ) fi;
   if x<0 then ERROR('square root of a negative number') fi;
   Digits := Digits + 3; # add some guard digits
   xkm1 := 0;
   xk := evalf(x/2); # initial floating point approximation
   while abs(xk-xkm1) > abs(xk)*10^(-Digits) do
      xkm1 := xk;
      xk := (xk + x/xk)/2;
   od;
   Digits := Digits - 3;
   evalf(xk); # round the result to Digits precision
end;
> x := Sqrt(3);
```

$$x := \mathrm{Sqrt}(3)$$

```
> evalf(x);
```

$$1.732050808$$

```
> Digits := 50;
```

$$Digits := 50$$

```
> evalf(Sqrt(3));
```

$$1.7320508075688772935274463415058723669428052538104$$

4.7 外部プログラムの呼び出し

C や Fortran で書かれた外部プログラムを Maple で呼び出したいことがしばしばあります. 実際, そうしたい 2 つの理由があります. 明らかに, Maple では出来ないが他のプログラムで出来るとき, Maple のアルゴリズムを作り直す代わりに他のプログラムを利用することは意味があります. 他の理由は効率性です. Maple は数式処理は効率が良いのですが, 機械精度の数値計算は一般に効率が良くありません. そこでたとえば, 固有ベクトルを数値計算で求めるのに Fortran ライブラリルーチンを利用したくなるでしょう.
Maple のデータの受け渡しはファイルによって行わなければなりません. Maple プログラムは外部プログラムで必要とされるデータを入力ファイルとして書き出しておかなければなりません. Maple は `system` コマンドを用いて外部プログラムを実行させることができます. 外部プログラムは入力ファイルからデータを呼び出し, その結果を出力ファイルに書き出さなければなりません. プログラムの実行が終了したあとで, Maple はその結果を出力ファイルから呼び出し, Maple に取り込みます. こういうことを行う Maple プログラミングの概略は次のようになります.

```
interface(quiet=true);
writeto(input);
... # write any data into the file input
writeto(terminal);
interface(quiet=false);
system(...); # execute the external program
read output;
... # continue processing in Maple
```

これをもう少し詳細に調べてみましょう. 最初のコマンド `interface(quiet=true)` は Maple のすべての診断すなわちメッセージや警告で使用されるいろいろな出力をオフにし, それらが `input` ファイルに書き出されないようにします. データが `input` ファイルに書かれたあとで, これは `interface(quiet=false)` でリセットされます.

次のコマンド `writeto(input);` は `input` と名付けられたファイルを出力用に開きます. この時点から Maple のすべての出力はこのファイルに入ります. 出力は通常 `lprint` か `printf` コマンドで作られます. コマンド `writeto` はこのファイルを上書きします. すでにファイルの中にあるものにデータを追加したければ, `writeto` の代わりに `appendto` コマンドを使って下さい.

すべてのデータがこのファイルに書かれたあとで, コマンド `writeto(terminal);` を用いて出力をターミナルに戻すと暗黙のうちに閉じられます.

`system` コマンドは外部プログラムを実行するために用います. `system` コマンドが行う正確な呼び出しはシステムに依存しています. Unix では, この呼び出しは次のようになるでしょう.

```
system('foo < in > out');
```

外部プログラム `foo` は入力ファイル `in` 上で実行され, その結果はファイル `out` に書込まれます. `system` コマンドは外部プログラムが正常に終了したかどうかを示す値を返します. Unix の場合, 正常な終了なら 0 を返します.

最後に, データが Maple の書式で書かれていれば, `read` コマンドを用いて `out` ファイルの結果を Maple に読込みます. 任意のデータを読込むのは以下に述べられている `readline` か `sscanf` か `readdata` コマンドを用います.

このモードでは Maple は外部プログラムをサブルーチンのように呼び出します. 結果を Maple に読込ん

だあとで Maple は実行を続行します. 実はこれが外部プログラムを呼び出す非常に簡単な方法で, デバッグも簡単です. ファイルを検査して, データが決められた書式になっていることを確かめることができます. しかし, テキストファイルを使ったデータの受け渡しはそれほど効率よい方法でないようです. 多くのアプリケーションでは外部プログラムでなされた仕事が重要なものならば, この非効率性は問題にならないでしょう.

4.8 数値データのファイル入出力

ここで Maple から得たデータを他のプログラムで読めるようにファイル出力する方法や Maple が読み込めるように外部プログラムでデータを書式化する方法を, もっと詳細に議論しましょう.

Maple *V Release 2* にファイル入出力機能として, ルーチン `printf, sscanf, readline` があります. ルーチン `printf` は一般的な書式の出力に用います. これは C の printf ルーチンにならって作られたもので, 代数式の出力用の % a を受け付ける以外は同じです. ここで `printf` を用いて浮動小数点データを出力する典型的な 1 例を挙げます.

```
> printf('A float in fixed point:%f and scientific notation:%e\n', x, x );
```

```
A float in fixed point: 0.577216 and scientific notation: 5.772157e−01
```

`printf` 関数は第 1 引数として出力するべきテキストを指定する文字列を受取り, % 制御列を用いてその値はどのように出力すべきかを第 2 引数として与えます. 上の例では, 2 つの値が % f と % e 制御列で出力されています. エスケープ列 \n は改行するために用います. 他の役立つ制御列は整数と文字列を出力する % d と % s で, さらに制御列 % a は Maple 代数式を出力するためのものです. 他のオプションについては `?printf` を参照して下さい.

`readline` ルーチンは, テキストファイルから 1 行のテキストファイルを文字列として読むために用います. `sscanf` ルーチンは文字列を調べて数字を抽出するために用います. `printf` の逆を行なうものです. したがってあるデータを読むためには, `readline` を用いてファイルの 1 行ずつを読んで, 各行ごとに `sscanf` を用いてデータを抽出します. ユーテリティルーチン `readdata` を書くと空白やタブで区切られた数値データのカラムを持つテキストファイルを読むことができます. カラム 1 個以上指定されていると, `readdata` ルーチンはデータのリストやデータのリストのリストを返します. ファイル `foo` が次のようになっているとしましょう.

```
2.1   3.2   1.2
2.4   3.1   3.4
3.9   1.1   5.5
```

このデータを読んだ結果がここにあります.

```
> readlib(readdata):  # load the raaddata function from the library
> readdata(foo,2); # read the first two columns of the data
```

```
[[2.1, 3.2], [2.4, 3.1], [3.9, 1.1]]
```

4.9 Fortran と C の出力

`fortran` コマンドと `C` コマンドは Fortran や C のコンパイラに適した書式で Maple の数式を出力するものです. これは Maple で数式や数式のベクトルや行列を開発したとき, これらの数式を Fortran や C のサブルーチンにしたいときに役立ちます. どのようにして Maple の数式を Fortran や C に変換するのでしょうか？ Maple 関数 `fortran` と `C` はこのためのものです. 1 例を挙げてみましょう. 区間 [2,4] の 5 桁の精度で補誤差関数 erbc(x)=1-erb(x) の近似となる次の Maple 多項式を作ったとしましょう.

```
f := −3.902704411 x + 1.890740683 − 1.714727839 x^3 + 3.465590348 x^2
   − .0003861021174 x^7 + .5101467996 x^4 − .09119265524 x^5 + .009063185478 x^6
```

この補誤差関数がどんなものであるかはこの例題の目的にはさほど重要ではありません. 実は統計学の正規分布に関係があるのです. 近似 f をどのように作ったかを知ることもここでは重要ではありません. 我々の Fortran と C ライブラリにはこの関数は組み込まれていないので, 与えられた範囲内でこの関数のおよその近似を必要とします. 興味のある読者は, コマンド `chebyshev(erfc(x),x=2..4,10^(-5))` を用いてチェビイシェフ級数近似を作り, 以前に多項式に展開するために書いた `'expand/T'` ルーチンを使いましょう. 上の近似 f を効率良く求めるために, この多項式をホーナー形式にしたのです. それから Fortran コードをつくりたいのです. 次のようになるでしょう.

```
> h := convert(f,horner);
    h := 1.890740683 + ( −3.902704411 + (3.465590348 + ( −1.714727839 + ( .5101467996
      + (−.09119265524 + ( .009063185478 − .0003861021174 x) x )x ) x ) x ) x ) x
> fortran(h);
t0 = 0.1890741E1+(-0.3902704E1+(0.346559E1+(-0.1714728E1+(0.510146E0+(-0.9119266E-1+(0.90631
#85E-2-0.3861021E-3*x)*x)*x)*x*)*x)*x)*x
```

数式は 1 行より長いので, Maple は継続記号を付け 2 行の Fortran コードを生成しました. 浮動小数点数は単精度の Fortran `E` 表現に自動的に変換され, 7 桁で打ち切られました. Maple はその結果を変数 `t0` に代入しました. ここで変数 *r* が割当てられたファイル `temp.c` に C のプログラムを出力しましょう. この `C` 関数は最初に Maple に読み込まれていなければいけないことに注意して下さい.

```
> readlib(C):
> C([r=h], filename='temp.c' );
```

ファイル `'temp.c'` を見ると, 次のようになっていることがわかります.

```
r = 0.1890740683E1+(−0.3902704411E1+(0.3465590348E1+(−0.1714727839E1
+(0.5101467996+(−0.9119265524E−1+(0.9063185478E−2−0.3861021174E−3
*x)*x)*x)*x)*x)*x)*x)*x
```

読者はこのコマンドの追加情報機能とオプションについては, `?fortran` や `?C` でヘルプページを調べて下さい.

5 練習問題

(1) `remove1(x,L)` は値 x がリスト L の中にあれば最初に出てくる x を削除し, その中になければ `FAIL` を返すものです. この `remove1` の Maple 手続きを書きなさい.

(2) 数値のリストの分散を求める `variance` という Maple 手続きを書きなさい. `variance(x)` は

$$\frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)^2$$

を計算するものです. ただし, n はリスト x の要素の個数で, μ はリスト x の中の数値の平均です. リストが空ならば, ルーチンはエラーを出力して下さい.

(3) $m \times n$ 行列 A のフロベニウス・ノルムを計算する Maple 手続きを書きなさい. フロベニウス・ノルムは次のように表されます.

$$\sqrt{\Sigma_{i=1}^{m}\Sigma_{j=1}^{n}|A_{ij}|^2}$$

(4) 「配列」の節にあったバブルソート (*bublesort*) アルゴリズムよりも速く数値配列を整列するアルゴリズム, たとえばシェルソート, クイックソート, マージンソート, ヒープソートなどの Maple 手続きを書きなさい. その Maple 手続きを修正して, 配列の 2 つの要素を比較するときに用いる論理関数 f をオプションの第 2 引数として受け付けるようにしなさい.

(5) フィボナッチ多項式 $F_n(x)$ を計算する Maple 手続きを書きなさい. この多項式は線形漸化式 $F_0(x) = 1,\ F_1(x) = x,\ F_n(x) = xF_{n-1}(x) + F_{n-2}(x)$ を満足します. 最初の 10 個のフィボナッチ多項式を計算し, 因子分解しなさい. そのプログラムは $F_{50}(x)$ を計算できますか ?

(6) Maple の数式が与えられたとき, その数式の構造をツリーとして返す, すなわち,

- 原始的なオブジェクト (整数と文字列) の場合はそのオブジェクトそのものを返す.
- 原始的なオブジェクトでない場合は, そのオブジェクトの型をを第 1 要素とし, 残りの要素はそのオブジェクトのオペランドの構造とするリストを返す.

とする `structure` という Maple 手続きを書きなさい.

例: `structure(x^3*sin(x)/cos(x));` は次のものを返します.

$$[*,\ [\hat{}\,,\ x,\ 3],\ [function,\ \sin,\ x],\ [\hat{}\,,\ [function,\ cos,\ x],\ -1]]$$

ルーチンでは Maple の型 `integer, fraction, float, '+', '*', '^', string, indexed, function, range, equation, set, list, seriese, table` を扱えるようにしなさい. 次の入力であなたの関数を検査してみて下さい.

```
Int( exp(-t)*sqrt(t)*ln(t), t=0..infinity ) =
int( exp(-t)*sqrt(t)*ln(t), t=0..infinity );
```

(7) `DIFF` 手続きを拡張して, 一般なベキ, 関数 ln, exp, sin, cos を微分できるようにし, 与えられた入力をどのように微分するかわからないとき, エラーを出す代わりに未評価値を返すようにしなさい. `DIFF` を拡張し

て連続微分すなわち関数 DIFF を DIFF するようにしなさい. たとえば, DIFF(DIFF(f(x,y),x),y) は DIFF(DIFF(f(x,y),y),x) と同一の結果を出力するようにして下さい.

(8) 数値の集合と非負の整数 n が与えられたとき, 大きさ n のすべての組合せのリストを返すルーチン comb を書きなさい. たとえば, 次のようになります.

```
> comb( a,b,c,d, 3 );
```

$$\{\{a,b,c\},\{a,b,d\},\{a,c,d\},\{b,c,d\}\}$$

このルーチンを修正して, 重複した要素を含むリストでも働くようにしなさい. たとえば, 次のようになります.

```
> comb( [a,b,b,c], 2 );
```

$$[[a,b],[a,c],[b,b],[b,c]]$$

(9) Maple 関数 degree は与えられた変数 (s) の多項式の総次数を求めるものです. たとえば, 次の多項式では,

```
> p := x^3*y^3 + x^4 + y^5:
> degree(p,x);  # degree in x
```

$$4$$

```
> degree(p,{x,y})  # total degree in x and y
```

$$6$$

となります. 我々は Maple でこの degree 関数をプログラム化する方法を DEGREE 関数で示しました. しかしこの degree 関数と DEGREE 関数は整数のベキに対してのみ働きます. この DEGREE 関数を拡張して, 一般的な数式の次数, たとえば有理数や記号式のベキに対しても働くようにしなさい.

```
> f := x^(3/2) + x + x^(1/2);
```

$$f := x^{\frac{3}{2}} + x + x^{\frac{1}{2}}$$

```
> DEGREE(h,x);
```

$$\frac{3}{2}$$

```
> h := (x^n + x^(n-1) + x^2) * y^m;
```

$$h := (x^n + x^{(n-1)} + x^2)\,y^m$$

```
> DEGREE(h,x);
```

$$\max(n,\,2)$$

```
> DEGREE(h,{x,y});
```

$$\max(n,\,2) + m$$

(10) 次のように表される 1 変数多項式 $a(x)=\sum_{i=0}^{m} a_i x^i$ と $b(x)=\sum_{i=0}^{n} b_i x^i$ の和と積を計算する Maple 手続きを書きなさい.

(a) 係数の Maple リスト $[a_n, a_{n-1}, \cdots, a_1, a_0]$
(b) 係数の結合リスト $[a_n, [a_{n-1}, \cdots, [a_1, [a_0, NIL]]\cdots]]$

(11) 上の問題のデータ構造では, スパースな多項式 (係数に零が多い多項式) は零が明らさまに出てくるので効率が悪い. たとえば, 多項式 $x^{10}+x+1$ を考えてみましょう. これを Maple リストとして表すと,

$$[10, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]$$

となり, 結合リストとして表すと

$$[10, [0, [0, [0, [0, [0, [0, [0, [0[1[1, NILL]]]]]]]]]]]$$

となります. スパース多項式のもっとも効率良い表現は非零の項のみ保存するものです. $e_k > e_{k+1} \geq 0$ として, k 番目の非零の項 $a_k x^{e_k}$ を $[a_k, e_k]$ として保存することにしましょう. 今度は次のように表されたスパースな 1 変量多項式の和と積を求める手続きを書いて下さい.

(a) 非零の項から成る Maple リスト
(b) 非零の項から成る結合リスト

(12) ガウス整数 $\mathbb{Z}(i)$ は整係数を持つ複素数, すなわち $a,b \in \mathbb{Z}$ で $i=\sqrt{-1}$ のとき $a+bi$ の形をした数の集合です. このガウス整数に関するもっと興味あることは, それがユークリッド整域を形成するのかすなわち素因子分解されるかということです. 素因子分解できるならば, ユークリッドアルゴリズムを用いて 2 つのガウス整数の GCD を計算できるからです. ガウス整数 $x=a+bi$ が与えられたとき, $||x|| = a^2+b^2$ を x のノルムとします. $x=gy+r$, $||r|| < ||y||$ となる 2 つの非零なガウス整数の剰余 r を求める Maple 手続き `REM` を書いて下さい. この剰余ルーチンを用いて, 2 つのガウス整数の GCD を求めなさい.

(13) $\mathbb{Z}$ における GCD 計算で整数的記号的に簡単化する Maple の n 変数手続き GCD を書きなさい. たとえば, `GCD(b,GCD(b,a),-a)` が入力されたときは, このルーチンは GCD(a,b) を返します. 整数が入力された場合はルーチンは直接 GCD を計算します. 記号名が入力された場合は, GCD ルーチンは次のことを知っているべきです.

(a) $GCD(a,b) = GCD(b,a)$ (GCD は可換である)
(b) $GCD(GCD(a,b,),c) = GCD(a,GCD(b,c)) = GCD(a,b,c)$ (GCD は結合的である)
(c) $GCD(0,a) = GCD(a)$ (単位元は 0 である)
(d) $GCD(a) = GCD(-a) = abs(x)$

(14) 変数リスト $[v_1, \cdots, v_m]$ に関する総次数 n の単項式のリストを求める Maple 手続き `monomial(v,n)` を書きなさい. たとえば, `monomial( [u, v], 3)` はリスト $[u^3, u^2v, v^3]$ を返します.
ヒント : t に関する積

$$\prod_{i=1}^{m} \frac{1}{1-v_i t}$$

のテーラー級数展開を考え, t^n で整理しなさい. Maple の `taylor` コマンドを参照して下さい.

(15) 相異なる x に対して点列 $(x_0, y_0), (x_1, y_1), \cdots, (x_n, y_n)$ が与えられたとき, Maple ライブラリ関数 `interp` はニュートン補間法を用いてそれらの点を補間する n 次以下の多項式を求めるものです. それらの点に対する 3 次自然スプラインは区分的 3 次多項式で, 各区間 $x_i < x \le x_{i+1}$ は 3 次多項式

$$f_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i \qquad 1 \le i \le n$$

で定義され, $4n$ 個の未知係数は次の $4n$ 個の条件

$$\begin{aligned} f_i(x_{i-1}) &= y_{i-1}, & f_i(x_i) &= y_i, & i &= 1, \cdots, n \\ f_i'(x_i) &= f_{i+1}'(x_{i-1}), & f_i''(x_i) &= f_{i+1}''(x_{i-1}), & i &= 1, \cdots, n-1 \\ f_1''(x_0) &= 0, & f_n''(x_n) &= 0 \end{aligned}$$

によって一意に決定されます. これらの条件は得られた区分的多項式が C^2 連続であることを表しております. 点 (x_i, y_i) の入力が `[x0, y0, x1, y1, ···, xn, yn]` の形のリストと変数で与えられたとき, `[f1, f2, ···, fn]` を出力する Maple 手続きを書きなさい. こうするには未知係数を持つ区分的多項式を作る必要があるので, Maple を用いてその導関数を計算し, 得られた方程式を解きます. たとえば, 次のようになります.

```
> spline( [0,1,2,3], [0,1,1,2], x);
```

$$\left[-\frac{1}{3}x^3 + \frac{4}{3}x,\ \frac{2}{3}x^3 - 3x^2 + \frac{13}{3}x - 1,\ -\frac{1}{3}x^3 + 3x^2 - \frac{23}{3}x + 7 \right]$$

(16) 区分的関数を表すデータ構造を設計しなさい. 前問の区分的 3 次スプラインの表現は Maple の他の機能とインタフェースを持っておりません. そうしないで, 区分的多項式を次のように未評価関数で表現してみましょう.

$$\mathtt{IF}\,(c_1, f_1, \cdots, c_{n-1}, f_{n-1}, f_n)$$

は

$$\begin{aligned} &\text{if } c_1 \quad \text{then } f_1 \\ &\text{if } c_2 \quad \text{then } f_2 \\ &\cdots \\ &\qquad\quad \text{else } f_n \end{aligned}$$

を表すとします.

- 純粋に数値で表される条件に対する区分的多項式を簡単化する `IF` という Maple 手続きを書きなさい.
- IF 式が浮動小数点で評価され, その IF 式が作図できるような ‘evalf/IF‘ という手続きを書きなさい.
- IF 式を微分する ‘diff/IF‘ という手続きを書きなさい.

以上の手続きはたとえば次のような結果を出すでしょう.

```
> IF( x<0, sin(x), cos(x) );
```

$$\mathrm{IF}\,(x < 0,\ \sin(x),\ \cos(x))$$

```
> diff(",x);
```

$$\mathrm{IF}\left(x<0,\ \cos(x),\ -\sin(x)\right)$$

```
> IF( Pi/3<0, sin(Pi/3), cos(Pi/3) );
```

$$\mathrm{IF}\left(\frac{1}{3}\,Pi<0,\ \frac{1}{2}\,3^{\frac{1}{2}},\ \frac{1}{2}\right)$$

```
> evalf(");
```

$$.5000000000$$

`IF` 手続きを修正して, 入れ子になった IF 式を簡単化出来るようにして, 上の例のような定数条件を扱えるようにしなさい. たとえば, その IF は次のような結果を示すことになります.

```
> IF( x<0, 0, IF( x<1, x^2, IF(x>2, 0, 1-x^2) ) );
```

$$\mathrm{IF}\left(x<0,0,x<1,x^2,2<x,0,1-x^2\right)$$

```
> IF( Pi/3<0, sin(Pi/3), cos(Pi/3) );
```

$$\frac{1}{2}$$

(17) 次の反復法はハートレー法として知られております.

$$x_{k+1}=x_k-f(x_k)/f'(x_k)/\left(1-\frac{f(x_k)\,f''(x_k)}{2\,f'(x_k)^2}\right)$$

Maple でこの反復法をプログラムして, これが 3 次収束することを確かめなさい. この収束は 3 次であることを証明しなさい.

(18) 常微分方程式 (*ODE*) のすべての関数は閉じた形で解けなくても, コマンド `dsoleve` は常微分方程式を解析的に解きます. 常微分方程式

$$y'(x)=f(x,y(x))$$

と初期条件 $y(0)=y_0$ が与えられていて, 級数解の最初の数項を求めたいとします.

$$y(x)=\sum_{k=0}^{\infty}y_k x^k$$

$f(x,y(x))$ と初期条件 y_0 が入力されたとき, 線形方程式を解く Maple 手続きをを書きなさい. すなわち,

$$y(x)=y_0+\sum_{k=1}^{n}y_i x^i$$

として, この有限和を常微分方程式に代入し, 係数比較して未知数 y_i について解きます. この結果を次数 n までに切断するには `taylor` コマンドを使うとよいことに注意して下さい. 次の常微分方程式

$$y'(x)=1+x^2-2xy(x)+y(x)^2,\qquad y(0)=1$$

をあなたの Maple 手続きで調べて, 次の級数解を求めてみて下さい.

$$y=1+2x+x^2+x^3+x^4+x^5+O(x^6)$$

Maple の `dsolve` コマンドを用いてこの解を解析的に求め, 上の級数解と一致することをチェックして下さい.
ニュートン反復法を用いて, 2 次収束する級数 $y(x)$ を求めなさい. k 番目の近似値

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

は $O(x^{2^k})$ の精度です. したがって $x_0 = y_0$ から出発して, $x_2 = y_0 + y_1 x + y_2 x^2 + y_3 x^3$ を求めていきます. この級数を求める Maple 手続きを書きなさい.

(19) GaussianElimination 手続きを修正して, 有理数から成る長方形行列について完全な行縮約を行って, この行列を行縮約化階段形にして下さい. さらにその手続きを修正して要素が数式から成る行列を扱えるようにしなさい. 中間結果を簡略化するには `simplify` 関数を使用しなさい.

(20) erf(x) を計算するのにテーラー級数を用いると,

$$\frac{\sqrt{\pi}}{2}\operatorname{erf}(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{n!\,(2n+1)}$$

収束が遅いから x が大きくなると次第に非効率になってきます. 正負の項が打ち消し合うために精度も次第に悪くなってきます. 大きな x に対する erf(x) の漸近級数の項を収束するまで和をとっていく Maple 手続きを書きなさい.

$$\frac{\sqrt{\pi}}{2}\operatorname{erf}(x) = 1 - \frac{2\sqrt{\pi}}{x} e^{-2x} \sum_{n=1}^{\infty} \frac{(-1)^n 1 \times 3 \times \cdots \times 2n-1}{2^n x^{2n}}$$

級数を `Digits` の有効桁数の精度で収束させるには, x はどの位まで大きくてよいでしょうか? テーラー級数とこの漸近級数の両者を用いて, すべての x に対して `Digits` の有効桁数の精度まで erf(x) を計算する手続きを書きなさい.