

情報処理教育センターにおけるMathematicaの利用法

大塚, 寛

<https://doi.org/10.15017/6770311>

出版情報 : 情報処理教育広報. 20, pp.11-28, 1997. Educational Center For Information Processing, Kyushu University

バージョン :

権利関係 :



情報処理教育センターにおける *Mathematica* の利用法

大塚 寛

ohtsuka@math.kyushu-u.ac.jp

1 はじめに

数式処理システム *Mathematica* は汎用の数式処理としての機能はもちろん、グラフィックスやプログラミング、プレゼンテーション (ノートブック版のみ) 機能にも優れた対話型システムで、情報処理教育センター 4 階の講義室のパソコンで利用できます。本稿では、この *Mathematica* の数式電卓とでも言うべき初歩的な使い方と若干のプログラム作成など、基本的な *Mathematica* の利用法を解説します。この解説が *Mathematica* を演習などで利用するきっかけになれば幸いです。

なお本稿で利用する *Mathematica* は Windows95 上で動作するバージョン 2.2.3 のノートブック版ですが、バージョン 2 以上であれば特に断らない限りノートブック版、ターミナル版のいずれでも動作します。ただしいくつかの機能はノートブック版だけでしか利用出来ません。また Windows95 については、ある程度操作方法が分かるものとして、細かな手順までは説明しません。

2 *Mathematica* の起動と終了

ノートブック版の *Mathematica* を起動するには、タスクバーの左端にある「スタート」から「プログラム (P)」 「**Mathematica**」とメニューをたどり「**Mathematica 2.2.3**」を選択します (図 1)。

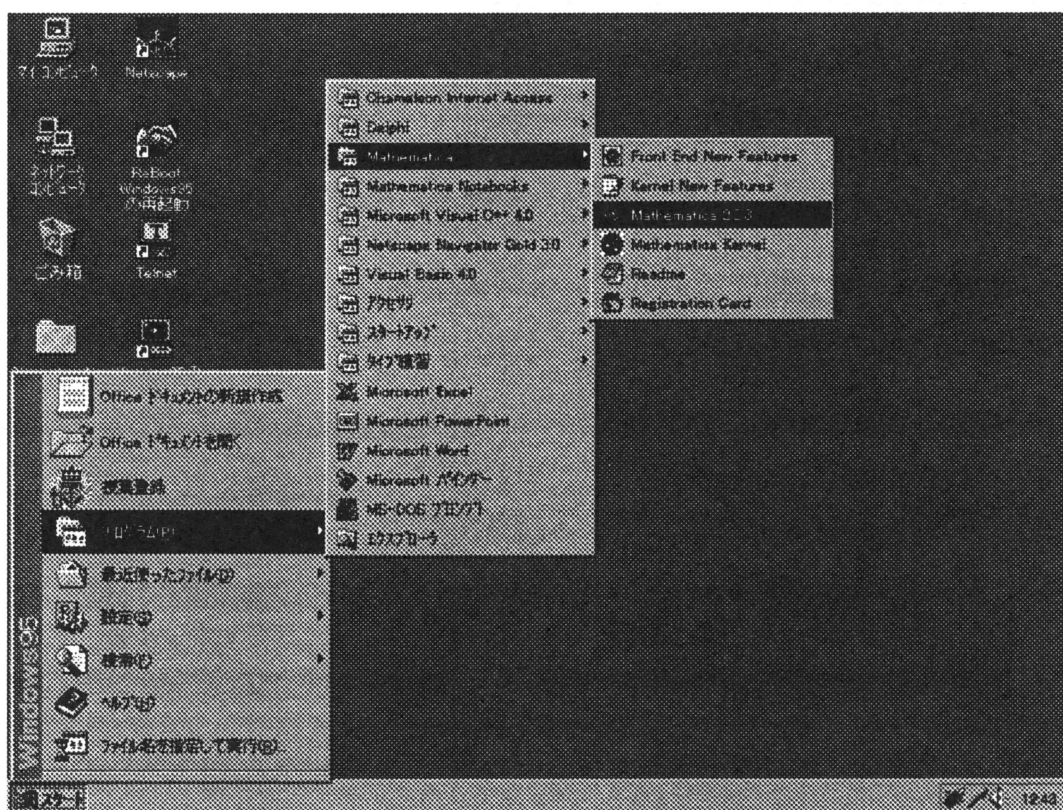


図 1: メニュー画面

同じメニューにある「**Mathematica Kernel**」を選択するとターミナル版の *Mathematica* が起動します。また、メニュー項目「**Mathematica**」の下にある「**Mathematica Notebooks**」にはいくつかのサンプルのノートブックがあるので、ノートブック版 *Mathematica* の利用方法の参考になるでしょう。

Mathematica を起動すると、図 2 のような何も書かれていない画面になり、入力待ちになります。

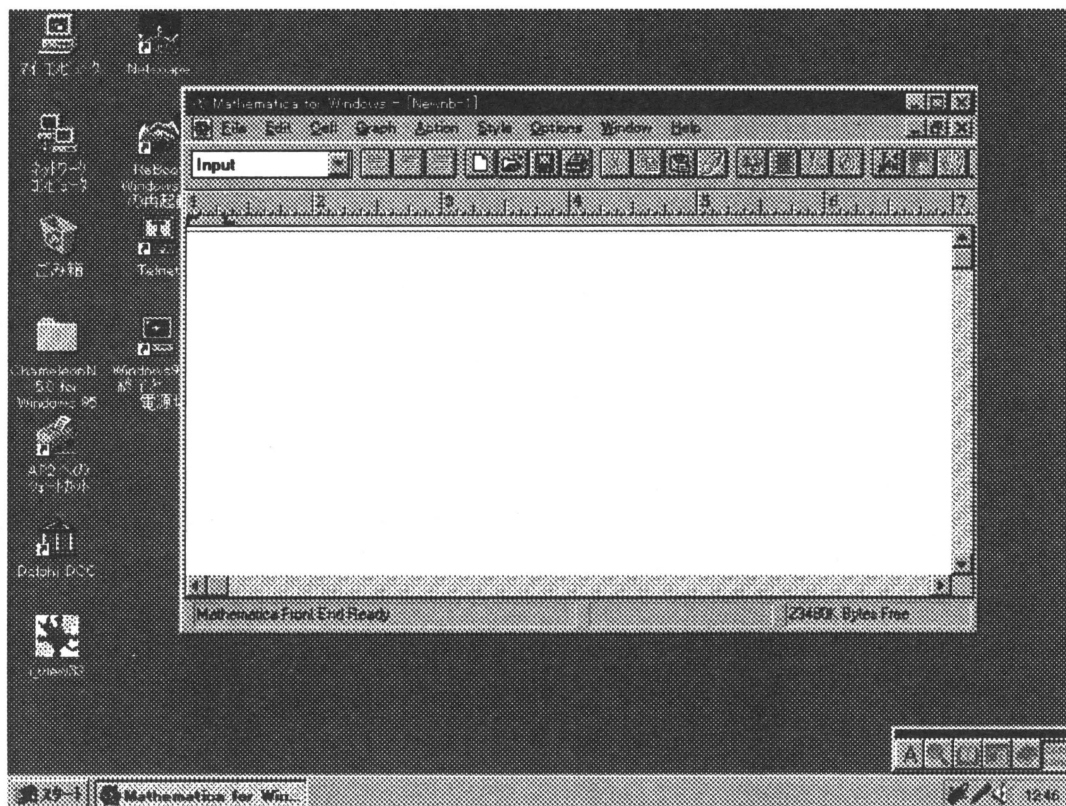


図 2: 初期画面

Mathematica を終了するには、*Mathematica* の「**File**」メニューから「**Exit**」を選びます (図 3)。この時もしすでに何らかの入力を行なっていれば、図 4 のセッション (*Mathematica* を起動してから現在までのノートブックの状態) を保存する旨のダイアログが現れます。*Mathematica* はセッションが新規に作られたものであれば、保存するファイル名として Newnb-1.ma を用意するので、再び利用するような場合は、このファイル名でセッションを保存します。

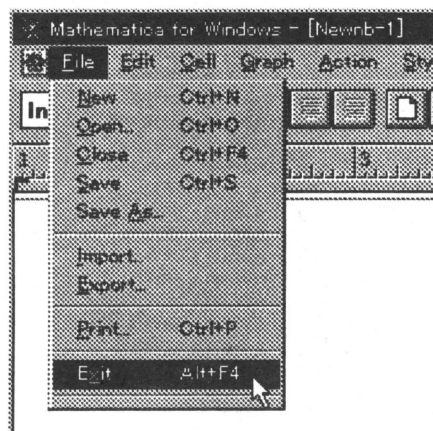


図 3: **File** メニューで **Exit** 選択

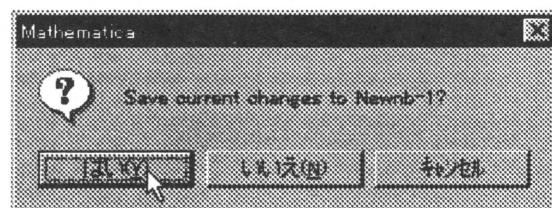


図 4: 「はい」を選択

保存するフォルダ、ファイル名は適当に替えて構いませんが、ファイルの拡張子にはノートブックを表す .ma をつけるようにしてください (図 5)。この処理の後 *Mathematica* は終了します。

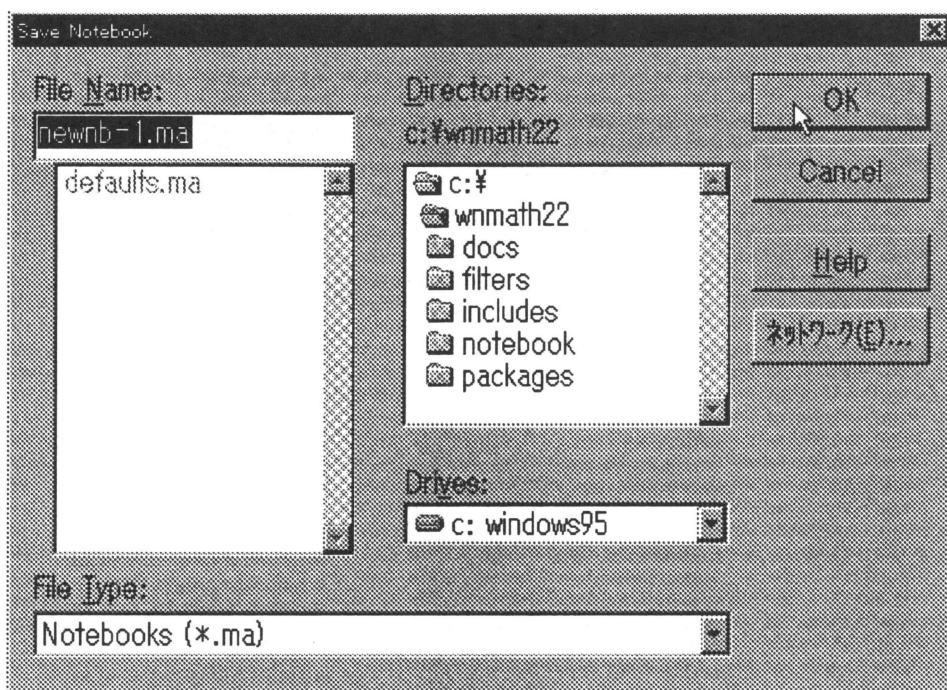


図 5: Save Notebook ダイアログ

Mathematica は保存する際に拡張子が .ma と .mb のファイルを同じフォルダにつくります。 .mb ファイルはファイルサイズが大きくなるので、フロッピーディスクに保存する場合は、まずハードディスクに保存した後、.ma ファイルだけフロッピーディスクにコピーして下さい。 .ma ファイルだけでも Mathematica は動作します。 ftp で WS に転送する場合は、.ma ファイルはアスキーモードで構いませんが、.mb ファイルはバイナリモードで転送する必要があります。

3 数式電卓として使う

まず Mathematica がどういう事をできるか見るために、簡単な計算をさせてみます (図 6)。

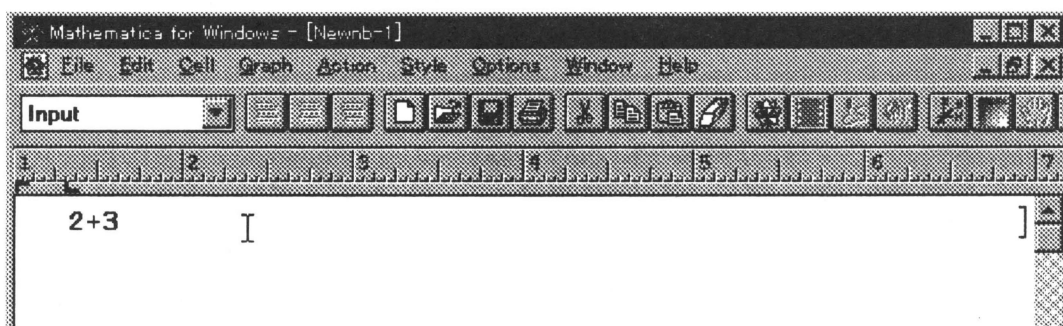


図 6: 入力

2+3 とタイプすると、画面の右側にセルブラケットと呼ばれる Mathematica の入出力等の単位を示す枠が現れます。この時のタスクバーの状態に注目します (図 7)。



図 7: まだ Kernel を起動していない状態のタスクバー

この後、Shift キーを押しながら Enter キーを押す (Shift+Enter と記す) と、タスクバーの状態は図 8 の様になります。



図 8: Kernel を起動した状態のタスクバー

そしてしばらく時間が経ったのち、入出力プロンプト $In[1]:=$, $Out[1]=$ と共に 5 という答えが返ってきます。また、画面の右側のセルブラケットが入れ子になっているのが分かります (図 9)。

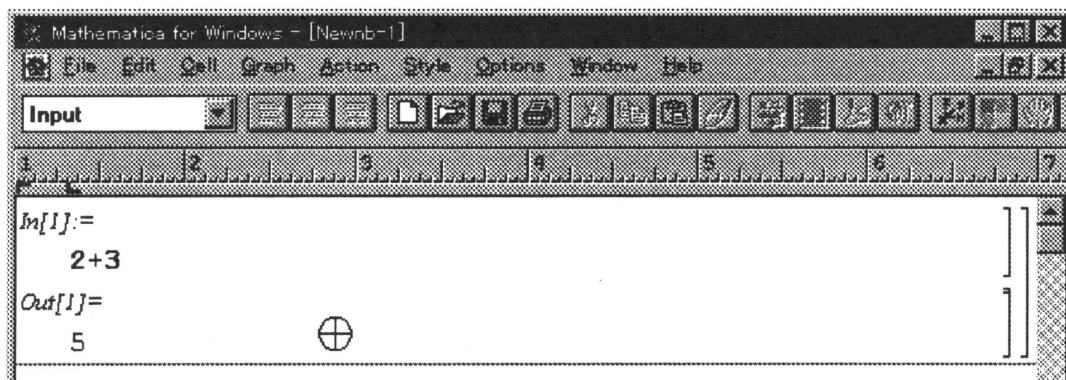


図 9: 答えの表示

実はノートブック版の *Mathematica* を立ち上げた時点では、まだ *Mathematica* の計算部分を受け持つカーネル (ターミナル版の *Mathematica*) は起動しておらず、式を計算する必要ができて初めてカーネルが起動されます。この後は簡単な計算ならすぐに答えが返ってきます。またカーネルとの対比で、ノートブックの表示を行なう部分はフロントエンドとも呼ばれます。

なお Unix での例ですが、ターミナル版の *Mathematica* を起動すると、以下のように入力プロンプト $In[1]:=$ が表示されて入力待ちになります。

```
ohtsuka@pisces ohtsuka> math
Mathematica 2.2 for Solaris
Copyright 1988-94 Wolfram Research, Inc.
-- Motif graphics initialized --
In[1]:=
```

終了するには、Quit あるいは Quit[] とタイプして Enter を入力するか、単に **Ctrl**+D を入力します。この際ノートブック版みたいにセッションが保存出来るわけではありません。

また上と同じ入出力は、 $2+3$ のタイプ後 Enter を入力して得ることが出来ます。

```
In[1]:= 2+3
Out[1]= 5
```

これ以降、特にノートブック版特有のものでない限り、*Mathematica* の入出力は以下のように、ユーザによる入力を **boldface** 体で、*Mathematica* の出力を *type writer* 体で表示し、右側に短いコメントを書きます。

```
Sin[1/2 Pi]          (* sin  $\frac{1}{2}\pi$  を計算させるために、ターミナル版では Enter を、*)
1                    (* ノートブック版では Shift+Enter を入力して、答えを得る *)
```

以下いくつかの式を *Mathematica* に計算させてみますが、その前にいくつかの約束ごとと *Mathematica* で使う特殊記号を挙げておきます。

Mathematica の基本的な約束事

文字 使える文字はアルファベットと数字で、アルファベットは大文字小文字を区別する。名前はアルファベットで始まる。組み込み関数やコマンド、定数名はすべて大文字 (特別な定数は \$) で始まり、名の中にいくつかの大文字を含むものもある。また良く利用する D や Det などを除き、名前は数学で利用する通りの命名法が使われ、短縮されない。

当然ですが、数の計算だけでなく、記号 (symbol) による計算も扱えます。

```

2 a + 7 a (* 代数的計算 2a + 7a *)
9 a (* 記号的な結果 9a を返す *)
(x + y)^4
4 (* (x + y)^4 *)
(x + y) (* そのままでは展開されず、複数行にわたって表示される *)
Expand[%] (* 直前の出力を関数 Expand を使って展開 *)
4 3 2 2 3 4
x + 4 x y + 6 x y + 4 x y + y (* x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4 *)
D[%, x] (* 直前の出力を関数 D を使って x で偏微分 *)
3 2 2 3
4 x + 12 x y + 12 x y + 4 y (* 4x^3 + 12x^2y + 12xy^2 + 4y^3 *)
Simplify[%] (* 直前の出力を関数 Simplify を使って簡単な式に直す *)
3
4 (x + y) (* 4(x + y)^3 *)
D[(x + y)^4, x] (* 元の式を x で偏微分してみる *)
3
4 (x + y) (* 同じであることが分かる *)

```

簡単な 2 次、連立および微分方程式を解いてみます。方程式は「左辺 == 右辺」の形です。

```

Solve[2 x^2 + 5 == 0, x] (* 数字を係数に持つ 2 次方程式を解く *)
5 5 (* 2 つの厳密解  $\pm\sqrt{\frac{5}{2}}i$  を得る *)
{{x -> -I Sqrt[-]}, {x -> I Sqrt[-]}} (* 解は左辺を右辺に置き換えるルー *)
2 2 (* ル演算子 '->' とリストで表される *)
%[[1,1]] (* 直前の出力の解の 1 番目を抽出 *)
5 (* -I Sqrt[5/2] は近似せずに求められるもっともよい解 *)
x -> -I Sqrt[-] (* ルール演算子 (一時的な置き換え) で解を表す *)
2
2 x^2 + 5 /. % (* 2 x^2 + 5 で x を -I Sqrt[5/2] に置き換える *)
0 (* 確かに方程式の解である事が分かる *)
Solve[{2 x + 3 y == 7, 4 x - 3 y == 5}, {x, y}] (* 解を持つ連立方程式を解く *)
{{x -> 2, y -> 1}}
Solve[{x^2 == 1, x^3 == 2}, x] (* 解を持たない連立方程式を解く *)
{} (* 空のリストが返ってくる *)
DSolve[y''[x] + y[x] == 0, y[x], x] (* 微分方程式を解く。''(クォート 2 つ) は 2 回微分 *)
{{y[x] -> C[2] Cos[x] - C[1] Sin[x]}} (* 一般解を得る *)
DSolve[{y''[x] + y[x] == 0, y[0] == 0}, y[x], x] (* 初期値つき微分方程式を解く *)
{{y[x] -> -(C[1] Sin[x])}}

```

ヘルプを使う

Mathematica は数式処理システムなので、どの数学的分野がどのような関数、コマンドを使って扱えるか、もっと詳しく説明すべきでしょうが、それらを全て列挙するのは到底無理なので、ここでは関数、コマンド等を Mathematica の中で調べる方法を説明します。

Mathematica に組み込まれている記号 (関数、コマンド、定数) を調べるには、? を使います。

```

?Plot (* 関数 Plot の標準的な使い方を調べる *)
Plot[f, {x, xmin, xmax}] generates a plot of f as a function of x from xmin to
xmax. Plot[{f1, f2, ... }, {x, xmin, xmax}] plots several functions fi.

```

?によって得られる説明は、必ずしもその関数のすべての使い方を示しているわけではありません。もっと詳しい情報を知りたいければ、??を使います。

```

??Plot                                     (* 関数 Plot と、その属性、オプションを調べる *)
Plot[f, {x, xmin, xmax}] generates a plot of f as a function of x from xmin to
xmax. Plot[{f1, f2, ... }, {x, xmin, xmax}] plots several functions fi.
Attributes[Plot] = {HoldAll, Protected}                                     (* 属性 *)
Options[Plot] =                                     (* オプション。途中 25 個のオプションは省略 *)
{AspectRatio -> GoldenRatio^(-1), Axes -> Automatic, AxesLabel -> None,
:
FormatType -> $FormatType, TextStyle -> $TextStyle}

```

更にこの中の属性やオプションについて知りたいときも、改めて?, ??を使います。

関数名が分かっている場合は、上の方法で調べることができますが、分からない場合は、*Mathematica* の命名法が数学で利用する通りで省略されない事を利用して、考え付く名前の前後にワイルドカード*をつけて?で問い合わせることができます。

```

?Bessel*                                  (* Bessel で始まる名前を列挙する *)
BesselI BesselJ BesselK BesselY

?*Plot                                    (* Plot で終わる名前を列挙する *)
ContourPlot      ListContourPlot ListPlot      Plot
DensityPlot      ListDensityPlot ParametricPlot

?*Log*                                                  (* Log を含む名前を列挙する *)
EllipticLog      LogGamma      LogicalExpand LogIntegral PolyLog      Log

```

これで得られた各々の名前について、改めて?, ??で調べればよいことになります。今までに出てきた関数についても、これらの方法で利用法を調べてみると、もっといろいろな使い方ができることが分かるでしょう。

更にノートブック版では、すべて英語ですが、Windows95 標準のヘルプシステムを利用したオンラインのヘルプが用意されています。これを利用するには、*Mathematica* の「**Help**」メニューで「**Contents**」あるいは「**Search for Help on...**」を選択します。「**Contents**」を選択すると(図 10)、図 11 のような「**Windows ヘルプ**」の *Mathematica* の目次部分が現れます。

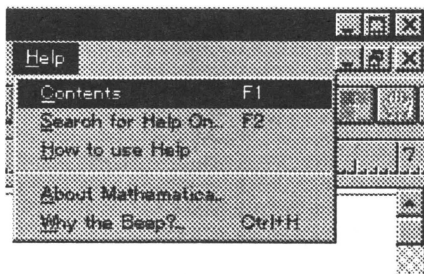


図 10: Help メニューで
Contents 選択

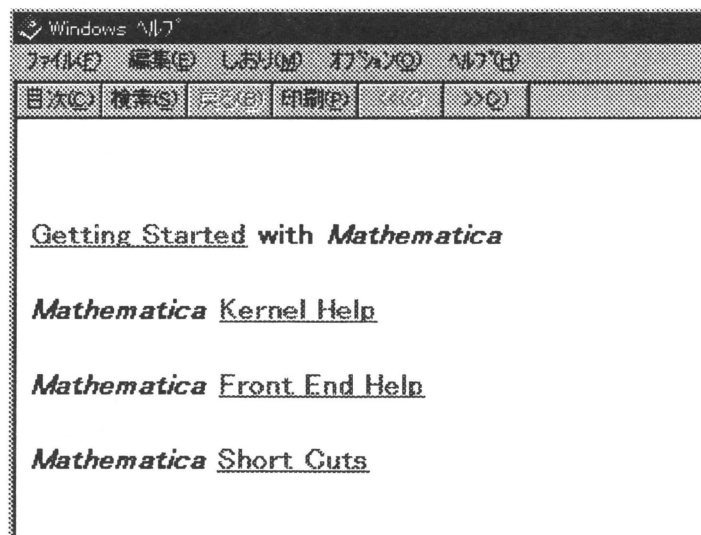


図 11: Windows ヘルプ

「**Search for Help on...**」を選択すると(図 12)、図 13 のようなダイアログが現れます。

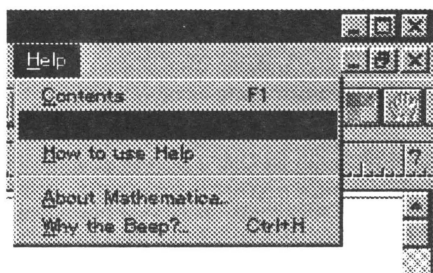


図 12: Help メニューで
Search for Help on... 選択

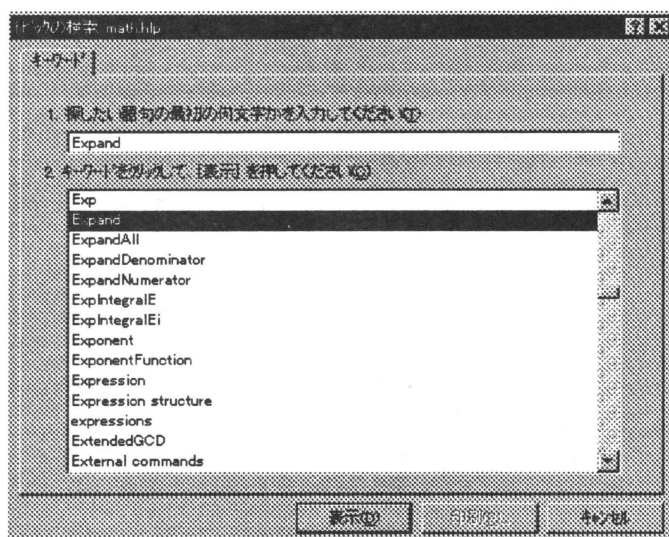


図 13: 「トピックの検索」ダイアログ

ここでは *Mathematica* の目次部分の「Kernel Help」から「Categorical listing」「Algebraic Computation」「Basic Algebra」と最初のリンクを順々にたどって得られる関数 `Expand` を入力しています。この後「表示 (D)」をクリックすると、`Expand` に対応する「Windows ヘルプ」の画面が現れます (図 14)。

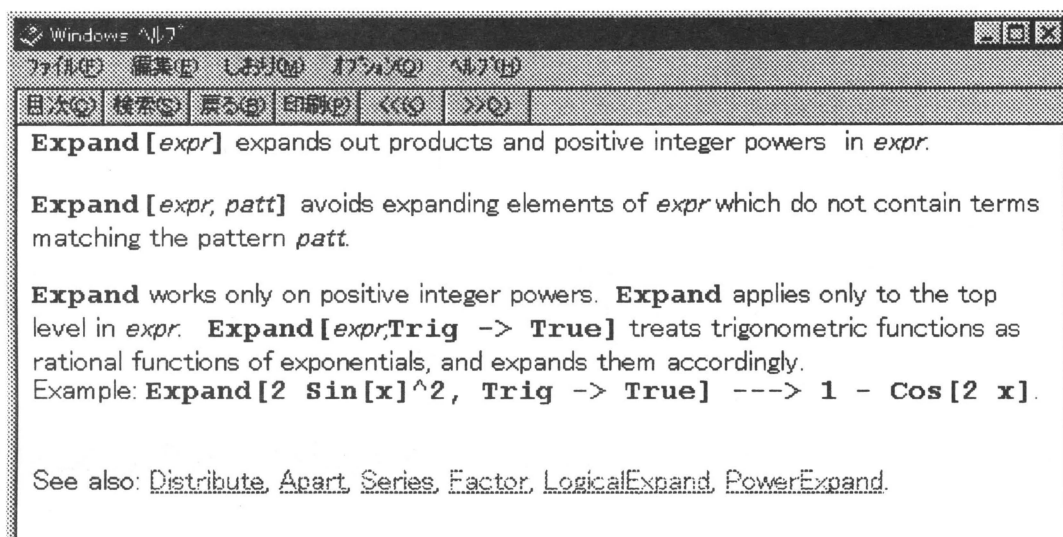


図 14: 関数 `Expand` の解説表示

Mathematica の目次部分 (図 11) の「Kernel Help」以下には、分野毎とアルファベット順に分けて組み込みの関数などの解説があります。目次部分ではその他に「Getting Started」「Front End Help」「Short Cuts」が選べますが、これらにはノートブックの使い方など、*Mathematica* を操作する上で必要なことがすべて載っていると行って過言ではありません。しかもリンクを辿ることによって、関連する項目へ簡単に移動できるので、書籍よりは便利です。*Mathematica* を使って何かをする際には、このオンラインヘルプも忘れずに利用して下さい。

なお Mac や Unix 上のノートブック版 *Mathematica* には、**Function Browser** と呼ばれるオンラインのヘルプより便利な代物がありますが、残念ながら Windows95 版にはついていません。

標準パッケージを使う

Mathematica では組み込み関数やコマンドしか使えないわけではありません。組み込み関数ではないが特定の数学的分野で利用される関数や、様々なユーティリティー類をパッケージとして提供し、これらを *Mathematica* にロードすることで、パッケージに定義されている関数を組み込み関数と同

じ様に利用することができます。

具体的には `c:\wnmath22\Packages` 以下の数学的分野その他を表すサブフォルダに、各分野で提供される標準パッケージが用意されています。

DOS の制限によりフォルダ名が 8 文字までしかありませんが、実際には Preload を除いた以下のフォルダ名で、計 13 個あります。

Algebra	Calculus
DiscreteMath	Examples
Geometry	Graphics
LinearAlgebra	Miscellaneous
NumberTheory	NumericalMath
ProgrammingExamples	Statistics
Utilities	



図 15: `c:\wnmath22\Packages` 以下

ここではグラフィックス関連のパッケージを例にとって、その使い方を紹介します。

Graphics 以下には 21 個のパッケージがあります。右図ではパッケージの拡張子は表示されていませんが、実際は `.m` で、以下のファイル名です。

Animation.m	ArgColors.m
Arrow.m	Colors.m
ComplexMap.m	ContourPlot3D.m
FilledPlot.m	Graphics.m
Graphics3D.m	ImplicitPlot.m
Legend.m	Master.m
MultipleListPlot.m	ParametricPlot3D.m
PlotField.m	PlotField3D.m
Polyhedra.m	Shapes.m
Spline.m	SurfaceOfRevolution.m
ThreeScript.m	



図 16: 図 15 の Graphics 以下

このうち Master.m は、そのフォルダの中の他のパッケージをすべてロードするパッケージで、**Examples**, **ProgrammingExamples** 以外のどのフォルダにもあります。

パッケージをロードするには、ファイルを読み込む一般的な演算子 `<<` かパッケージのロードのための関数 `Needs` を使います。パッケージはそれを含むフォルダ名と共に、「フォルダ名 ‘パッケージ名’」の形で指定します。パッケージ名を囲んでいるのは ‘(バッククォート)’ です。標準パッケージの場合はフルパスで指定する必要はありません。例えば `ComplexMap.m` をロードするには、

```
<< Graphics'ComplexMap'
```

あるいは同じパッケージを 2 度ロードすると問題が生じるので、指定したパッケージがまだロードされていないときにのみロードする `Needs` を使って、

```
Needs["Graphics'ComplexMap"]
```

とします。

パッケージをロードする前に、そのパッケージで定義されている関数を使うと、パッケージをロードした後に問題が生じます。以下はパッケージ `ComplexMap.m` に定義されている関数 `CartesianMap` を使う例ですが、関数 `Needs` による警告に注意してください。

```

CartesianMap[Log, {1., 4.}, {1., 4.}] (* ComplexMap をロードする前 *)
CartesianMap[Log, {1., 4.}, {1., 4.}] (* 入力した式が返り、関数が働かない *)
Needs["Graphics`ComplexMap`"] (* ComplexMap をロードする *)
CartesianMap::shdw: (* CartesianMap に関する警告 *)
Warning: Symbol CartesianMap appears in multiple contexts
{Graphics`ComplexMap`, Global`}; definitions in context
Graphics`ComplexMap` may shadow or be shadowed by other definitions.
CartesianMap[Log, {1., 4.}, {1., 4.}] (* ロード後 CartesianMap を使う *)
CartesianMap[Log, {1., 4.}, {1., 4.}] (* 前回と同じように、関数が働かない *)
Remove[CartesianMap] (* CartesianMap の定義を削除 *)
CartesianMap[Log, {1., 4.}, {1., 4.}] (* 改めて CartesianMap を使う *)
-Graphics- (* 絵は表示しないが、表示されたことを示している *)

```

実はパッケージ `ComplexMap` をロードする前に `CartesianMap` を使ったことで、*Mathematica* が `CartesianMap` なる名前の記号を新たに作り、`ComplexMap` をロードした後に `CartesianMap` を使っても、*Mathematica* が最初に作った `CartesianMap` を利用してしまうからです。

この問題を解決するには、例にあるように `Remove[CartesianMap]` を実行して、最初に作った `CartesianMap` を *Mathematica* から削除します。これには *Mathematica* のコンテキストや名前を検索する順序等が関わっているので、これだけでは正確な説明ではありませんが、一般的にパッケージをロードしたときに、「関数名::shdw:」云々というメッセージが出てきたら、そのメッセージに出てきた関数を `Remove` すれば、問題を解決することが出来ます。

4 プログラムを書く

Mathematica は汎用の高機能な数式処理システムですが、プログラミングも可能です。*Mathematica* が提供する標準パッケージも、*Mathematica* のプログラム言語としての機能を利用して作られています。

Mathematica をプログラム言語として見た場合、従来の手続き型のプログラミングだけでなく、関数型、論理型やオブジェクト指向のプログラミングが可能ほど高度な機能を持っています。しかもその基本的な部分は非常に単純であり、手続き型のプログラミングに関しては、ある程度のプログラミングの経験があれば、それほど多くのことを学ばなくても習得可能なものです。

ここではそのような人を対象に、*Mathematica* で手続き型のプログラミングをするための基本的なさわりの部分を紹介します。なお *Mathematica* の記法で具体例以外は、`type writer` 体の部分は文字通り入力すべき文字列を表し、*italic* 体の部分は入力すべき式を一般的に表すことにします。従って実際には *italic* 体の部分を目的に応じて *Mathematica* の式に置き換えて入力します。

入力のシンタックス

まず *Mathematica* のプログラム、以下では式 (expression) と呼びますが、これはアトムを基本要素として、次の関数の形で再帰的に定義されたものです。ただし、 $n \geq 0$ です。

$$f[arg_1, arg_2, \dots, arg_n]$$

ここでアトムとは、記号 (Symbol) か数か文字列 (String) であり、数には、整数 (Integer)、実数 (Real)、有理数 (Rational)、複素数 (Complex) があります。これらはそれぞれ括弧の中で示されたヘッドによって区別されます。 $f, arg_1, arg_2, \dots, arg_n$ は再び式を表します。ただし f は数等の定数、文字列以外の式です。この式のヘッドは f であり、記号の場合は関数名と呼び、 $arg_1, arg_2, \dots, arg_n$ を引数と呼びます。`Head[expr]` は式 $expr$ の (評価された後の) ヘッドを返します。

今まで出てきた演算子もそれに対応する関数を持ち、その関数の省略形 (エイリアス) として表わされたものです。例えば関数 `Plus` の省略形は $+$ で、 $a+b$ はその完全形 `Plus[a, b]` と同等です。

FullForm[expr] と入力すると、式 expr の (評価された後の) 完全形を得ることができます。

FullForm[a-b]	(* a-b の完全形を調べる *)
Plus[a, Times[-1, b]]	(* a と b の (-1) 倍の和であることが分かる *)

式は再帰的に構成されるので、式のある部分は再び式です。Part[expr, n] あるいはそのエイリアス expr[[n]] は式 expr を評価して得られた式の n 番目の部分を返します。

式の評価

正しい式が入力されれば、Mathematica はその式を以下のように評価 (evaluate) します。

数 式が数の場合は、そのまま数を返す。数は Mathematica の定数である。

記号 式が値が割り当てられていない記号の場合は、そのまま記号を返す。式が値が割り当てられている記号の場合は、その値を返す。記号は Mathematica の変数である。ただし、Pi のような数学上の定数を表す記号は Mathematica の定数であり、その形のまま返される。

文字列 文字列は文字列そのものを返す。文字列は数学的演算の対象にはならない。

関数 原則として関数は、まず引数が評価され、その後関数の本体が、得られた引数の値の下に評価され、その結果が返される。引数は第 1 引数が最初に、その後第 2 引数、..., 第 n 引数と評価が進む。

複合式、つまりセミコロン (;) で区切られた式では、個々の式が左側から順々に評価され、最後の式の結果が返される。特に最後がセミコロンで終わる場合は、値 Null が返されその値は表示されない。これは CompoundExpression のエイリアスである。

制御構造を扱うような特別な関数は、上の評価方法に従わないので注意すること。

関数は原則として call by value と呼ばれる評価方法をとりますが、SetAttributes で関数の属性 (attribute) を HoldFirst, HoldRest, HoldAll に変更することにより、該当する部分の評価を行わずに関数を実行することも出来ます。

リスト - 汎用のデータ構造

Mathematica では、構造を持つデータを表すための一般的なデータ構造として、リストを用意しています。リストは Mathematica の 0 個以上の式 (リストの要素) をコンマ (,) で区切り波括弧 ({, }) で囲んだ形であり、入れ子にも出来ます。これは List のエイリアスです。Mathematica にはリストを扱う様々な関数が用意されており、ここではその一部を紹介します。

ある規則性を持ったリストをつくる場合には、Table や Range ような関数が利用されます。

Table[n^2, {n, 1, 5}]	(* 1 から 5 までを 2 乗した数のリスト *)
{1, 4, 9, 16, 25}	
Range[1, 10]	(* 1 から 10 までの整数のリスト *)
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}	

関数をリストの各要素に適用させる関数として Map 等がありますが、多くの組み込み関数はリストに適用されると、自動的にリスト内の要素に適用されます。このような関数は Listable という属性を持ちます。

Map[Sin, {0, 1/2 Pi, Pi, 3/2 Pi, 2 Pi}]	(* Map を使って Sin をリストに適用 *)
{0, 1, 0, -1, 0}	(* リストの各要素に Sin を適用した結果が返る *)
Sin[{0, 1/2 Pi, Pi, 3/2 Pi, 2 Pi}]	(* Sin をリストに適用 *)
{0, 1, 0, -1, 0}	(* リストの各要素に Sin を適用した結果が返る *)

Mathematica ではリスト構造に様々な意味を持たせています。例えば、集合、ベクトルや行列はそれ専用のデータ構造を持つわけではなく、リストで表現されており、集合の演算、ベクトルや行列

の演算を表す関数はリストを引数にとるようになっています。また前出の Table や制御構造で説明する Do 等、繰り返しを利用する関数はリストで様々なループを指定します。

```
A = {{1, 2}, {3, 5}}; (* 2×2 行列の指定 (出力しない) *)
B = {1, 2}; (* 2 次元 (定数) ベクトルの指定 (出力しない) *)
X = {x, y}; (* 2 次元 (変数) ベクトルの指定 (出力しない) *)
A . X (* 行列とベクトルの積 (関数 Dot のエイリアス) *)
{x + 2 y, 3 x + 5 y}
Solve[A . X == B, X] (* 連立方程式 AX = B を X について解く *)
{{x -> -1, y -> 1}} (* 解を得る *)
```

関数の定義

Mathematica では関数を定義することが Mathematica のプログラムを書くことに当たります。ここでは関数を定義するために最低限知っておくべき事として、割り当て (assignment)、パターン、局所変数の基本的扱いについて解説します。

割り当て

Mathematica で式 *expr* をシンボル *symbol* に割り当てるには、次の 2 通りの方法があります。

即時割り当て *symbol* = *expr* あるいはその完全形 Set[*symbol*, *expr*]
まず *expr* を評価し、その結果が *symbol* に割り当てられます。

遅延割り当て *symbol* := *expr* あるいはその完全形 SetDelayed[*symbol*, *expr*]
割り当ての際には *expr* の評価は行なわれず、後で *symbol* が評価された時に、*expr* が評価されます。

以下の例では、各々 5 回の繰り返し毎に random1, random2 が評価され、その値がリストの要素になりますが、random1(即時割り当て) の場合は、割り当ての際の値が得られるのに対し、random2(遅延割り当て) の場合は、Random[] が呼び出されていることが分かります。

```
random1 = Random[] (* 即時割り当て。Random[] は乱数を発生させる関数 *)
0.386486 (* この時に Random[] が出力した値 *)
Table[random1, {5}] (* random1 の 5 個のリスト *)
{0.386486, 0.386486, 0.386486, 0.386486, 0.386486} (* 同じ要素のリストを得る *)
random2 := Random[] (* 遅延割り当て (出力されない) *)
Table[random2, {5}] (* random2 の 5 個のリスト *)
{0.709568, 0.552134, 0.372644, 0.802331, 0.3843} (* 異なる要素のリストを得る *)
```

パターン

一定の条件を満たす式を、その式の型や条件の組み合わせからなるパターンを利用して指定出来ます。記号 *x* のパターンという形で、いくつか簡単なパターンを以下に挙げます。

x_ *x* は任意の式にマッチする。

x_.head *x* はヘッドが *head* の式にマッチする。

x_.?predicateQ *x* は述語関数 *predicateQ* が True を返す式にマッチする。

何のことか分かりにくいと思いますが、これは Mathematica が式を評価する際に、パターンマッチングを行なうことと関係しています。パターンについてはこれ以上は解説しませんが、パターンは関数を定義する際に、引数のパターン、特に型を指定するために用いられます。

簡単な関数なら、遅延割り当てとパターンを用いて次のような形で定義することが出来ます。もちろん := は SetDelayed のエイリアスなので、式のシンタックスにあっていることは言うまでもありません。

```
userfunc[arg1_, arg2_, ...] := body
```

`userfunc` はこの関数のヘッド、`arg1_, arg2_, ...` は仮引数 (パラメータ) でパターンは `_` に限りません。`body` はその中に `arg1, arg2, ...` その他の変数が現れる可能性のある式です。同じ関数名でも引数の個数が異なれば、更に引数の個数が同じでも引数が異なるパターンであれば、*Mathematica* は異なる関数と見なします。

遅延割り当てとパターンを用いないで定義するとどうなるか、簡単な例をみてみます。

まず遅延割り当てについて、以下の左側が即時割り当ての例、右側が遅延割り当ての例ですが、9 行目の出力値に注目すると、即時割り当てと遅延割り当てでは出力値が異なります。これは遅延割り当てを用いた場合、定義した関数で使用される変数の値を変更出来るのに対し、即時割り当ての場合は、関数定義時の変数の値が使われて、その後の変数の値の変更が意味を成さないことを示しています。なお、この例で利用した関数 `Clear` はその引数に与えられた記号の定義を削除します。また、`?` は組み込みでない関数でも働きます。

```
Clear[add1]
a = 3;
add1[x_] = x + a
3 + x
add1[5]
8 (* a の再定義前 *)
a = 7;
add1[5]
8 (* a の再定義後、前と同じ *)
?add1
Global`add1
add1[x_] = 3 + x (* a は未使用 *)
```

```
Clear[add2]
a = 3;
add2[x_] := x + a

add2[5]
8 (* a の再定義前 *)
a = 7;
add2[5]
12 (* a の再定義後、前と異なる *)
?add2
Global`add2
add2[x_] := x + a (* a を使用 *)
```

次にパターンについて、以下の左側がパターンを使わない例、右側がパターンを使う例ですが、7 行目の出力値に注目すると、パターンを使わない場合は、`x` は `y` にマッチせず、定義した関数が適用されないで入力式 `square[y]` が返るのに対し、パターンを使うと、任意の式にマッチする `x` が `y` にマッチして定義した関数が適用され、 y^3 が返ります。

```
Clear[square]
square[x] := x^2
square[x]
2
x (* square が適用される *)
square[y]
square[y] (* square が適用されない *)

square[2 x]
square[2 x]
```

```
Clear[cube]
cube[x_] := x^3
cube[x]
3
x (* cube が適用される *)
cube[y]
3
y (* cube が適用される *)
cube[2 y]
3
8 y
```

局所変数

プログラムを書く際に、その関数だけでしか利用できない局所変数を使うことにより、関数の予想外の振る舞い等を避けることができます。関数の定義中に現れる変数を局所変数にするには、関数 `Module`, `Block`, `With` を使います。

Module `Module[{x,y,...},expr]` は式 `expr` (本体と呼ぶ) の変数 `x,y,...` を局所変数として用い、`expr` を評価する。さらに、第1引数のリストの要素が、`x=x0` の形になっていれば、局所変数 `x` に初期値 `x0` を設定することになる。

Block `Block[{x,y,...},expr]` は本体 `expr` の変数 `x,y,...` を局所変数として用い、`expr` を評価する。さらに、第1引数のリストの要素が、`x=x0` の形になっていれば、局所変数 `x` に初期値 `x0` を設定することになる。

With `With[{x=x0,y=y0,...},expr]` は本体 `expr` に現れる記号 `x,y,...` をすべて `x0,y0,...` に置き換える。

全く似たような説明ですが、それぞれ微妙に異なります。以下これらの違いを簡単に説明しますが、もし今までにコンパイラ型の言語を利用してきた経験があれば、`Module` を利用することを進めます。

まず `Module` と `Block` では、同じ名前を持つ大域変数と局所変数が現れたときに名前の衝突を避ける方法が異なります。`Module` は局所変数を静的有効範囲 (static scoping) とするのに対し、`Block` は局所変数を動的有効範囲 (dynamic scoping) とします。以下の例では `Module`, `Block` 共に `i` を局所変数に指定していますが、局所変数と同じ名前である記号 `i` にそれらを適用すると、`Module` では局所変数と記号を区別しているのに対し、`Block` では局所変数と記号を区別せず共に局所変数として扱っていることが分かります。

```
Clear[a, i] (* a と i の定義を削除 *)
sumofpowersM[x_] := Module[{i}, Sum[x^i, {i, 1, 5}]] (* Module を使って定義 *)
{sumofpowersM[a], sumofpowersM[i]}
      2      3      4      5      2      3      4      5
{a + a + a + a + a , i + i + i + i + i }
sumofpowersB[x_] := Block[{i}, Sum[x^i, {i, 1, 5}]] (* Block を使って定義 *)
{sumofpowersB[a], sumofpowersB[i]}
      2      3      4      5
{a + a + a + a + a , 3413} (* 1 + 2^2 + 3^3 + 4^4 + 5^5 を得る *)
```

また `Module` において局所変数の値が初期化されており、本体では局所変数に何も割り当てられていないならば、`Module` を `With` に置き換えることが出来ます。

再帰的な関数の定義

Mathematica で関数を定義する場合、定義されるべき関数が定義を与える `:=` の右辺に再び現われても構いません。このようにして与えられる関数を再帰関数と言います。以下は階乗を計算する再帰関数ですが、もちろん非再帰的な関数で定義することも可能です。

```
factR[n_Integer /; n >= 0] := If[n==0, 1, n*factR[n-1]] (* 再帰的定義 *)
factNR[n_Integer /; n >= 0] := Module[{x=0, y=1}, (* 非再帰的定義 *)
  While[!(x++ == n), y *= x]; y] (* While は制御構造にて説明 *)
Table[factR[i] == factNR[i], {i, 0, 10}] (* 値が等しいか調べる *)
{True, True, True, True, True, True, True, True, True, True, True}
```

なお階乗は非負整数でのみ定義されるので、非負整数に対応するパターン `n_Integer /; n >= 0` を用いて定義しています。

制御構造

手続き型プログラミングでは、条件分岐、繰り返しなどの制御構造を多用してプログラムを書きます。*Mathematica* にはこれらの制御構造を扱う関数が用意されており、ここではそのうち良く利用されるものを紹介します。なお制御構造はその性質上、*Mathematica* の式の評価規則の原則には当てはまらない事に注意してください。

条件分岐

条件分岐を行なう関数には If, Which, Switch 等があります。これらの関数は引数を未評価のまま保持し、条件に適合した場合にそれに対応する式のみを評価します。

If If[cond, t, f] は条件 cond を評価した値が真 (True) ならば t を、偽 (False) ならば f を評価して返す。真偽値でなければ、式そのものが未評価のまま返される。If[cond, t, f, u] は cond の値が真偽値でないときに u を評価して返す。

Which Which[test₁, expr₁, test₂, expr₂, ...] は偶数個の引数をとる。奇数番目の引数は真偽値を返す述語で、test₁, test₂, ... の順に評価し、test_i が True を返す最初の述語なら、expr_i を評価してその値を返す。どの述語も True と評価されない場合は Null を返す。

Switch Switch[expr, pattern₁, value₁, pattern₂, value₂, ...] は奇数個の引数をとる。expr を評価し、pattern₁, pattern₂, ... の順に比較し、最初にパターンマッチした pattern_i に対応する body_i を評価してその値を返す。マッチするパターンがない場合は Null を返す。最後のパターンを _ とすることでいずれにもマッチしなかった場合に対応できる。

繰り返し

繰り返しを行なう関数には Do, While, For 等があります。これらの関数は引数を未評価のまま保持し、繰り返しの中で評価します。関数の返り値は Null か、または内部で Return が評価された時は Return の引数を評価した値を返します。

Do Do[expr, {imax}] は式 expr を imax 回評価する。Do[expr, {i, imin, imax}] は変数 i を imin から imax まで 1 ずつ増加させながら式 expr を評価する。Do[expr, {i, imin, imax, di}] は変数 i の増分を di とする。Do[expr, {i, imin, imax}, {j, jmin, jmax}, ...] は多重ループ。反復変数は局所変数として扱われる。

While While[test, expr] は test を評価し、それが True である限り expr を評価する。

For For[start, test, incr, body] は start から評価し、test が True を返さなくなるまで繰り返し body と incr をこの順に評価する。反復変数は局所変数ではないことに注意。

While や For では、Break や Continue を使って繰り返しから抜け出したり、繰り返しの次の反復に移ることができます。

ソーティングを例にとって

延々と Mathematica で利用する関数を紹介してきましたが、ここではそれらを使って簡単なプログラムを書いてみます。と言っても数式処理から離れて、汎用のプログラム言語として利用出来ることを見るために、リストの要素を昇順に並べ替える選択ソートとクイックソートを実装します。

ノートブックの対話型環境でプログラムが簡単に変更できることを見るために、以下のプログラムをエディタで編集して Mathematica にロードするのではなく、ノートブック上で入力して下さい。この時、各行の右端に注釈の形で **Shift**+Enter があるとところで **Shift**+Enter を入力し、それ以外では Enter で改行して下さい。

まず後でいろいろ操作するために、次の関数 swap をリストの要素の交換に用います。

```
swap
SetAttributes[swap, {HoldFirst}] (* Shift+Enter *)
swap[l_Symbol, i_, j_] := ({l[[i]], l[[j]]} = {l[[j]], l[[i]]}; 1) (* Shift+Enter *)
```

また n を引数として 1 から n までの整数を並べ替える次の関数でデータを作る事にします。

```
RandomPermutation
RandomPermutation[n_Integer?Positive] := Module[{t},
  t = Array[{Random[], #}&, n]; t = Sort[t]; Map[#[[2]]&, t] (* Shift+Enter *)
```

選択ソート (関数名: SelectionSort) は *Mathematica* では以下のように実装出来ます。

```

SelectionSort
SelectionSort[list_List] := Module[{l = list, i, n = Length[list], min, minj, j},
  Do[ min = l[[i]]; minj = i;
    Do[If[l[[j]] < min, min = l[[j]]; minj = j], {j, i+1, n}];
    swap[l, i, minj],
    {i, 1, n-1}];
  ]
(* Shift+Enter *)

```

クイックソート (関数名: QuickSort) は *Mathematica* では以下のように実装出来ます。

```

QuickSort
QuickSort[list_List] := Module[{l = list},
  qSort[l, 1, Length[l]]; l]
(* Shift+Enter *)

SetAttributes[qSort, {HoldFirst}]
(* Shift+Enter *)
qSort[l_, n0_, n1_] /; n0 >= n1 := l
(* Shift+Enter *)
qSort[l_, n0_, n1_] :=
  Module[{lm = l[[ Floor[(n0 + n1)/2 ]]], i = n0, j = n1},
    While[ True,
      While[ l[[i]] < lm, i++ ];
      While[ l[[j]] > lm, j-- ];
      If[ i >= j, Break[] ];
      swap[l, i, j];
      i++; j--
    ];
    qSort[ l, n0, i-1 ]; qSort[ l, j+1, n1 ]
  ]
(* Shift+Enter *)

```

シンタックスにミスがあれば、**Shift+Enter** を入力した際にエラーや警告が表示されます。またシンタックスに誤りがなくても、関数が正しく定義されているか、意図しない関数がないか等々? を使って調べることが出来ます。間違いのある関数は、それを定義した入力セルブラケットをアクティブにして、間違った部分を訂正後、改めて**Shift+Enter** を入力します。余計な関数がある場合は **Remove, Clear** を使って余計な関数を削除します。

以上の関数が定義できたとして、ノートブック上の対話型の環境でいろいろ操作してみます。まずそのまま実行してみると、ソート出来るのが分かります。

```

list = RandomPermutation[20]
(* 要素数 20 の list を作る *)
{5, 3, 12, 20, 11, 16, 6, 1, 14, 18, 19, 13, 2, 15, 7, 8, 17, 9, 10, 4}
SelectionSort[list]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
list
(* list は変更されていない *)
{5, 3, 12, 20, 11, 16, 6, 1, 14, 18, 19, 13, 2, 15, 7, 8, 17, 9, 10, 4}
QuickSort[list]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}

```

次にそれぞれどのように実行されているのか実行過程を調べたいときは、リストの変更を行なう **swap** で、変更前のリストを出力するよう書き換えます。変更は **swap** を定義している入力セルブラケットをアクティブにして、**Print[l]** を追加し、**Shift+Enter** を入力するだけです。

```

swap[l_Symbol, i_, j_] := (Print[l]; {l[[i]], l[[j]]} = {l[[j]], l[[i]]}; l)

```

その後の操作は、ノートブックへの新しい入力として最後の方に移って行います。

```

list = RandomPermutation[10] (* 要素数 10 の list を作る *)
{3, 6, 8, 7, 2, 4, 1, 5, 10, 9}
SelectionSort[list]
{3, 6, 8, 7, 2, 4, 1, 5, 10, 9} (* ここから数行は Print の出力、途中略 *)
:
{1, 2, 3, 4, 5, 6, 7, 8, 10, 9}
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10} (* SelectionSort の返り値 *)
QuickSort[list]
{3, 6, 8, 7, 2, 4, 1, 5, 10, 9} (* ここから数行は Print の出力、途中略 *)
:
{1, 2, 3, 4, 5, 6, 7, 8, 10, 9}
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10} (* QuickSort の返り値 *)

```

リストの変更の様子の羅列を見てもおもしろくない場合は、標準パッケージの `Animation.m` を用いて、変更の様子をアニメーションで表示させることも出来ます。

Mathematica のグラフィックスについては特に説明はしませんが、グラフィックスを扱う関数は `-Graphics-` 等で示されるグラフィックオブジェクトを扱い、グラフィックスの表示は副作用 (side effect) でしかありません。従ってある関数を用いて一度表示した後も、その関数が出力したグラフィックオブジェクトさえあれば、グラフィックスを表示する一般的な関数 `Show` を用いて (再計算させることなく) 再び表示したり、他のグラフィックスオブジェクトと合わせて表示したり出来ます。

例で利用する `ListPlot` は数からなるリストを引数にとり、各点の x 座標を $\{1, 2, \dots\}$ としてリストをプロットします。 `ShowAnimation` はグラフィックオブジェクトからなるリストを引数にとり、リストの各要素を順々に表示することにより、アニメーション表示を行ないます。

以下ではまず、大域変数 `lists` に変更されたリストを順々に保存するよう `swap` を変更します (2 行目)。ソートし終えたら、ソートの結果得られた `lists` の各要素から `ListPlot` を使ってグラフィックオブジェクトを作ります (6, 11 行目)。 `Map` を使っているので、グラフィックオブジェクトのリストが得られます。以下の例では `ListPlot` のオプション `DisplayFunction` を `Identity` とすることにより、グラフィックスの表示は行っていない。得られたグラフィックオブジェクトのリストに `ShowAnimation` を適用すると、ターミナル版ではリストが変更される様子がアニメーションで繰り返し表示されます (8, 12 行目)。

```

Needs["Graphics`Animation`"] (* ShowAnimation のため、1 度だけやれば良い *)
swap[l_Symbol, i_, j_] := ({l[[i]], l[[j]]} = {l[[j]], l[[i]]}; AppendTo[lists, l]; l)
list = RandomPermutation[20];
lists = {list}; (* lists の初期値を list だけからなるリストに設定 *)
SelectionSort[list]; (* SelectionSort によるリストの変更を lists に保持 *)
Map[ListPlot[#, DisplayFunction->Identity, PlotStyle->PointSize[0.05]]&, lists]
{ -Graphics-, -Graphics-, ..., -Graphics- }
ShowAnimation[%];
lists = {list} (* lists の初期値を list だけからなるリストに設定 *)
QuickSort[list]; (* QuickSort によるリストの変更を lists に保持 *)
Map[ListPlot[#, DisplayFunction->Identity, PlotStyle->PointSize[0.05]]&, lists];
ShowAnimation[%];

```

ノートブック版では更に図 17 のように、 `ShowAnimation` で表示されたグラフィックスを選択、つまり `ShowAnimation` による出力のグラフィックスすべてのセルブラケットをアクティブにし、 *Mathematica* の「**Graph**」メニューから「**Animate Selected Graphics Ctrl+Y**」を選ぶか **Ctrl**+**y** と入力して、アニメーションを得ます。アニメーションを終了するには、例えば最終行に移ってマウスをクリックするとかします。

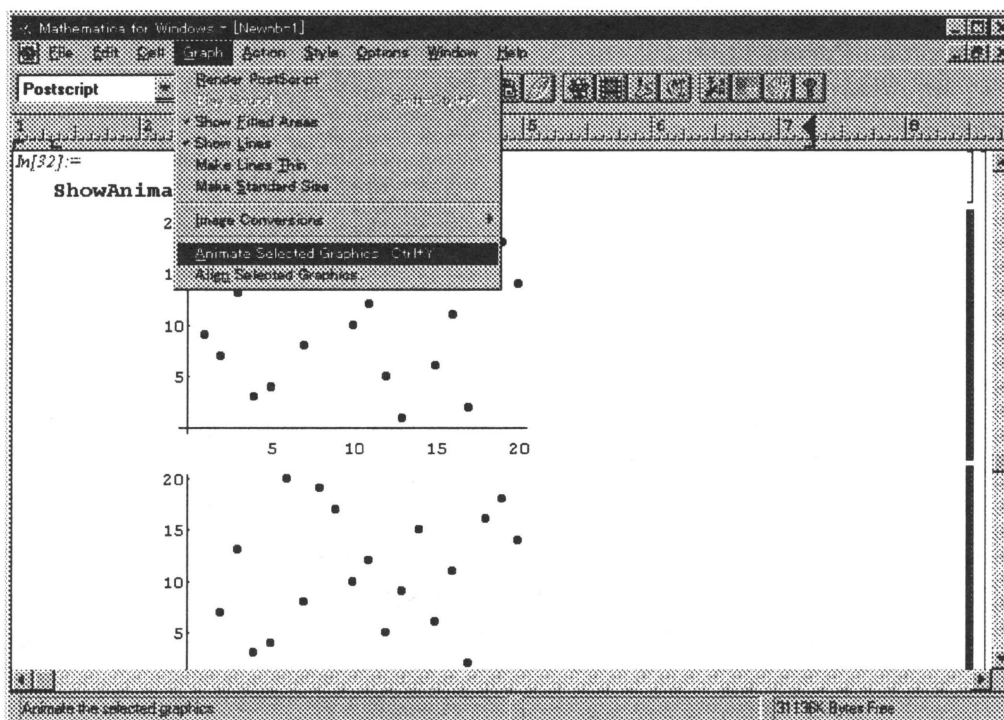


図 17: アニメーション

このように、対話型のシステムであるために、関数を簡単に変更したり、作った関数を調べたり、その結果を表示することがすぐにできます。

5 おわりに

以上駆け足で *Mathematica* の使い方を説明してきましたが、説明が片寄っていたり、必要な説明、特に数式処理や外部ファイルの扱い、トラブルシューティング等が抜けていたりして、まだまだ十分とはいえません。説明が不十分なところは、実際に使ってみたり市販の *Mathematica* の解説書を参考にして、*Mathematica* がどのようなものか自分で調べて貰うのが一番です。

しかしこの解説で、*Mathematica* を数式処理システムとしてだけでなく、Windows95 上でグラフィックスなどが手軽に扱える対話型のプログラム言語として捉えていただければ、そして演習などで利用していただければ幸いです。

最後に参考文献には *Mathematica* のプログラミングに関する日本語の解説書をいくつか挙げておきますが、*Mathematica* の詳しい解説には *Mathematica* の開発者 **Wolfram** の本 [1] が頼りになります。この他にもたくさんの、特に数式処理システムとして特定分野で *Mathematica* を利用するための解説書が出版されています。

参考文献

- [1] S. Wolfram: *Mathematica: A System for Doing Mathematics by Computer*, second edition (第 2 版、日本語版), Addison-Wesley/ アジソンウェスレイジャパン, 1992
- [2] N. Blachman: *Mathematica 実践的アプローチ*, Prentice-Hall/ トッパン, 1992
- [3] R. Maeder: *Mathematica プログラミング技法*, Addison-Wesley/ トッパン, 1992
- [4] N. Blachman: *Mathematica 事典*, トッパン, 1994
- [5] R.J. Gaylord, S.N. Kamin, and P.R. Wellin: *Mathematica プログラミング*, 近代科学社, 1994
- [6] R.D.. Skeel and J.B. Keiper: *Mathematica による数値計算*, 共立出版, 1995