SAD/Tkinter の使い方

生出勝宣

KEK, Oho, Tsukuba, Ibaraki 305, Japan

oide@acsad1.kek.jp

1997 年 10 月 29 日
(SAD1.0.5.4.4b に対応)

SAD は加速器の設計コードとして1986年からKEKで開発され現在に至っています (その概要はホームページ `http://www-acc-theory.kek.jp/SAD/sad.html` を参照)。最近では EPICS チャネル・アクセスや Python/Tkinter、Tcl/Tk インタープリータなどが組み込まれ、SADScript インタープリータ言語とあわせて、単に加速器の設計やシミュレーションに限らず、汎用のシステムとして利用可能なものになりつつあります。SAD/Tkinter は SAD/FFS/SADScript インタープリータから、Xウインドウのアプリケーションを書く道具、Tk tool kitを使うためのライブラリです。

このマニュアルに書かれている内容、SAD のプログラム及びライブラリは、今後予告なく随時改編されます。このマニュアルの最新版は上記のホームページからいつでもダウンロードできます。

なお、このマニュアルは SAD/Tkinter のすべてをカヴァーできていません。その理由は筆者自身がその全機能を経験・把握していないためです。そこで読者の皆様にはBrent Welch:*Practical Programming in Tcl and Tk*, 1995, *for Tcl7.4 and Tk4.0* を併読されることをお願いします。このマニュアルに説明がなくても、Tcl/Tk に備わっている機能は必ず利用可能です。

目次

1 SADを起動するまで

1.1 SAD計算機のアカウントを取得する。

SAD は主として KEK の SAD 計算機上で開発・運用されています。他の若干のシステムにも移植されていますが、ここでは SAD 計算機上のものを前提に話をすすめます。

KEK で加速器関係の仕事をする方は原則的に SAD 計算機のアカウントを取得できます。その方法は巻末のシステム管理者からのメッセージにしたがってください。

KEK の外部、あるいは国外から利用する場合はシステム管理者に相談してください。

SAD 計算機の主力は現在 acsad2.kek.jp (400MHz, 4CPU, 500MB) および acsad3.kek.jp (300MHz, 6CPU, 1GB) の DEC UNIX サーヴァです。

SAD の利用者には SAD のアップデート情報が電子メールで送付されます。希望者はoide@acsad1.kek.jpに連絡してください。

1.2 SADのプログラムの編集及び起動

SAD 計算機にはXサーヴァから、あるいは telnet ターミナルからログインします。Xのアプリケーションのためには当然Xサーヴァが必要です。ちなみに筆者は通常 Macintosh 上の eXodus をXサーヴァとして使っていますが、SAD/Tkinter にはサーヴァに依存する部分はありません。UNIX に対してSAD のために特別の環境設定をする必要はありません。

SAD のプログラムは単なるアスキーファイルであるのでどんなエディタでも作成できます。作成された入力ファイルを用いて SAD を起動するコマンドは

```
/SAD/bin/gs 入力ファイル
```

です。/SAD/bin をサーチpath に含めておけば単に

```
gs 入力ファイル
```

でも起動できます。ただし、/SAD/bin が/bin、/usr/bin、/usr/local/bin などより上位に指定されていないと、別の gs が動いてしまうことがあります。その場合には

```
which gs
```

で gs がどの gs であるかを確かめることができます。

> 筆者はエディタに emacs を使っていて、sad-mode という emacs lisp によるSAD 用のマクロも用いています。このマクロは自分のホーム・ディレクトリに/users/oide/.emacs というファイルをコピーすれば使用可能になります。これにより例えば編集中のバッファをセーヴせずに SAD の入力ファイルとする、あるいは SAD の書式を自動的に整列するなどの機能が付加されます。しかし、これらは SAD の実行の必要条件ではありませんし、UNIX や emacs の正統的なユーザにはかえって邪魔になるかもしれません。またこのマクロは一部サーヴァのキー配列に依存する機能があるため、どこでも無条件で使えるわけではありません。

2 Hello, World!

ここでは SAD/Tkinter で書いた “Hello, World!” をその最も単純なアプリケーションとして紹介します。このプログラムは(1) Xウインドウに `"Hello"` というボタンを表示し、(2) そのボタンをクリックするとターミナル (標準出力) に `"Hello, World!"` と出力する、というきわめて単純なものです。それは、SAD/Tkinter では、

```
FFS;
  w = Window[];
  b = Button[w,
    Text -> "Hello",
    Command :> Print["Hello, World!"]];
  TkWait[];
```

となります。

2.1 FFS

まず、第 1 行の `FFS;` は SAD の中の `FFS` というサブシステムを起動することを指示しています。`FFS` が実際何であるかは後述するとして、ここでは SAD/Tkinter はすべてこの `FFS` の中で実行されるものとして理解してください。

2.2 部品の定義

第 2 行

```
  w = Window[];
```

はある新しい `Window` という部品 (widget と呼ばれる) を定義しています。`Window` は Xサーヴァの画面上に通常は枠を伴って表示され、その中に各種の部品を詰めることができます。この例では左辺のシンボル `w` にこの新しい `Window` が割り当てられます。

> ちなみに SADScript では大文字と小文字は区別されます。また、システムに備え付けられた関数は単語の区切りの頭文字が大文字になり、他は小文字になります。例えば `ListPlot`, `LinearSolve` などがそれです。ユーザはシステムが定義していなければシンボルとして何文字でも使えます。
>
> ひとつのシンボル、数値、演算子の途中に空白や改行をいれることはできません。文字列はその行の最後に \ をおけば、改行して書くことができます。この場合改行文字はデータの中には含まれません。データの中に改行文字そのものをいれる場合は `\n` を入力します (`stringinput` 参照)。これ以外の場所には自由に改行や空白を入れられます。
>
> 従って SADScript の書式はきわめて自由です。その結果、出来上がったプログラムの見やすさはかなり書き方に依存してしまいます。なお実行速度は書式とは関係ありません。
>
> ところでこの行の最後にあるセミコロンは二つ以上の式を互いに連結してひとつの式をつくる演算子です(10.1.1 参照)。セミコロンを省いた場合、もしその行の最後で式が完結している

とみなされるならば、その式はそこで評価され、その結果が端末に(Out[n]:= の形で)出力されますOut。(なお、この結果は再利用できます。10.9.3 参照)。したがって、そのような出力が必要なければセミコロンをつけるべきです。ある式がセミコロンで終わってその後ろに何も書かれていないときにはセミコロンの後には特殊なシンボル Null があるとみなされます。Null の出力は何もないので上のような事情になるわけです。

右辺のWindow[]の記号[]は関数の引用を表わしています。この中には以下に見るように必要な引数をいれることができます。

2.3 部品の作成、属性の指定

次に第3,4,5行

```
b = Button[w,
  Text -> "Hello",
  Command :> Print["Hello, World!"]];
```

は Window wにいれる部品として Button b を定義しています。まず第 3 行にあるように、最初の引き数 w はこの Button が前に定義した Window wの子供であることを指示しています。ある部品はこのように直接 Window の子供となることもできますし、また他の部品、例えば Frame の子供にして間接的にある Window に表示させることもできます。

次の引数 Text -> "Hello" はこの Button の表面に "Hello"という文字を表示することを指示しています。記号 -> の意味は後述します。このようにある文字列は二重引用符で囲みます。引き数と引き数の区切りはカンマで表わします。

この例にあるように、ある部品の属性の設定を伴う定義は

```
シンボル = 部品[親, 属性 -> 値, ...];
```

という形をとります。属性は同時に何種類も指定して構いません。また、属性は部品を定義した後に

```
シンボル[属性] = 値;
```

のように再指定することもできます。例えば上の例では

```
b = Button[w, Command :> Print["Hello, World!"]];
b[Text] = "Hello";
```

としても結果は同じです。しかし、順番を逆にして

```
b[Text] = "Hello";                                    (ア)
b = Button[w, Command :> Print["Hello, World!"]];
```

としたのでは結果は全然違い、このボタンにはなにも表示されません。これは、(ア) の時点では b はまだButton ではないため、b[Text] = .. に Button の属性を指定するという作用がないからです。

2.4 コマンドの結合

さてその次の引き数

```
Command :> Print["Hello, World!"]
```

はこのボタン b がクリックされたときに実行する SAD のコマンドを指定しています。これも構文上は上述の 属性 -> 値 という形に似ていますが、記号 -> の代わりに :> が用いられています。:> は -> と同等ですが、違いはその右辺の式、今の場合は Print["Hello, World!"]が直には評価されず、与えた式のまま関数に渡されるという点にあります。もし、:> の代わりに -> を使いますと、この場合にはこのボタンを定義した時点で Print["Hello, World!"] が実行され端末に Hello, World! と出力されてしまい、またボタンのほうには Print の返す値 Null が割り当てられてしまいます。このような事態をさけるため :> が使われます。このような演算子は遅延演算子とよばれ、SAD では随所で用いられます。

また、Command のような遅延定義を必要とする属性を再定義するには

```
シンボル[属性] := 値;
```

のように演算子 := を = の代わりに用います（6.3.2 を参照)。

このように、Command には SAD の任意の式（プログラム）を結び付けることができますから、ボタンに対して任意の機能を与えることができます。

2.5 Tkの実行、TkWait

さて、このように定義してきた二つの部品 Window と Button は、画面上には現われていてもまだ何も動作することはできません。それは次の

```
TkWait[];
```

という関数が評価されてはじめて実行されます。TkWait[] は画面上に図 1のウインドウを表示して、イヴェント（例えばマウスクリック）待ちの状態にはいります。そこでこのボタンをクリックする毎に端末には

```
Hello, World!
```

と表示されます。

これでこの章の目的の “Hello, World!” は完成しました。

図 1: Hello, World!

TkWait[] という関数は TkReturn という関数が実行されるまで無限に待ちつづけ、その間ボタンのクリックへの反応などの必要な動作を行います（4.8.1 参照)。関数 TkReturn が呼ばれれば、その引き数を値として返しつつ TkWait[] は実行を終えます。現在の Tkinter はこのTkReturn を発行する、図 2 のような特別のボタンを画面の左上に必ず表示します。このボタンはクリックされると TkReturn["ReturnToSAD"] というコマンドを実行します。また、このボタンのあるウインドウの左上の四角をクリックすると SAD が即座に停止します。ただしこのボタンはデバッグの時以外はあまり有用でないため、将来は標準でなくなる可能性が高いと思われます。

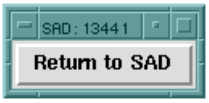

図 2: ReturnToSAD のボタン。ウインドウタイトルにあるのは SAD のプロセス番号。

3　梱包 pack

ひとつの Window の中に複数の部品を詰め込む場合、SAD/Tkinter ではそれらの部品の並び方は梱包者 packer が決定します。それぞれの部品はこの packer に梱包の際の各種の指示を与えることができます。

3.1　梱包の方向、Side

梱包の指示の内で最も基本的なものは、その部品がウインドウの4つの辺のどれに向かって詰め込まれるか、その方向を指定するものです。この方向は属性 Side によって与えられます。例えば、

```
w = Window[];
a = Button[w, Text -> "1 TOP",    Side -> "top"   ];
b = Button[w, Text -> "2 LEFT",   Side -> "left"  ];
c = Button[w, Text -> "3 RIGHT",  Side -> "right" ];
d = Button[w, Text -> "4 BOTTOM", Side -> "bottom"];
```

とすれば、a, b, c, d 4つのボタンが図 3 の左上のようにアレンジされます。また、これらを作成する順番を変えると、ボタンは次図のようにさまざまな様式で配置されます。この図で、ボタンの番号は作成順、文字はSide の方向を表わします。このように、Side の与え方及び作成の順番により、梱包の結果は変わってきます。

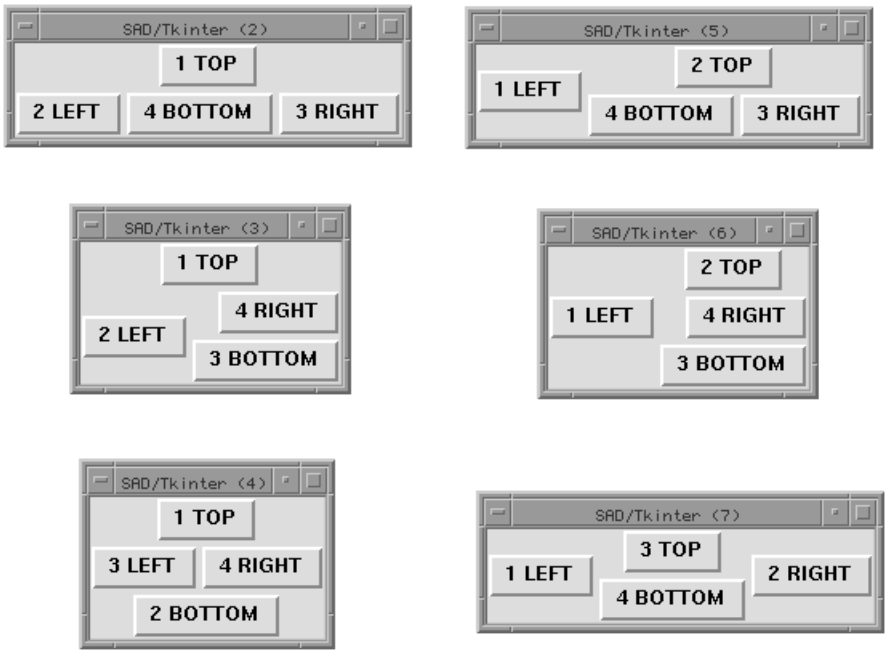

図 3: 梱包の結果は Side パラメータと作成の順番できまる。

3.2 部品の周りのすき間 PadX, PadY

梱包に際して、各部品のまわりにすき間を持たせたい場合があります。属性 PadX 及び PadY によって、それぞれ水平、垂直両方向のすき間を指定できます。この場合単位はピクセルですが、センチメートルやインチでの指定も可能です(後述)。例えば

```
w = Window[];
a = Button[w, Text -> "1 TOP",    Side -> "top",
  PadX -> 20, PadY->10];
b = Button[w, Text -> "2 LEFT",   Side -> "left",
  PadX -> 20, PadY->10];
c = Button[w, Text -> "3 RIGHT",  Side -> "right",
  PadX -> 20, PadY->10];
d = Button[w, Text -> "4 BOTTOM", Side -> "bottom",
  PadX -> 20, PadY->10];
```

とすると、図 4のような結果が得られます。

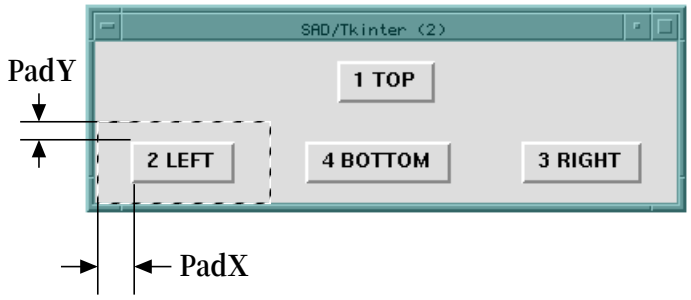

図 4: 部品の周りのすき間は PadX, PadY で確保する。

3.3 部品の内部のすき間 IPadX, IPadY

場合によっては、各部品の内部にすき間を持たせたい場合もあります。属性 IPadX 及び IPadY によって、それぞれ水平、垂直両方向の部品内部のすき間を指定できます。

```
w = Window[];
a = Button[w, Text -> "1 TOP",    Side -> "top",
  PadX -> 20, PadY->10];
b = Button[w, Text -> "2 LEFT",   Side -> "left",
  IPadX -> 20, IPadY->10];
c = Button[w, Text -> "3 RIGHT",  Side -> "right",
  PadX -> 20, PadY->10];
d = Button[w, Text -> "4 BOTTOM", Side -> "bottom",
  PadX -> 20, PadY->10];
```

の結果は図 5 になります。

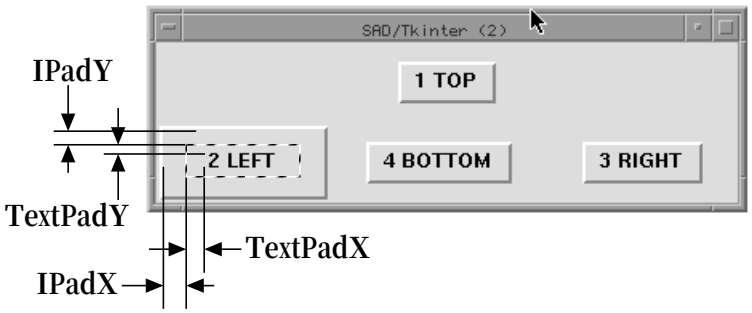

図 5: 部品内部のすき間は IPadX, IPadY 及び TextPadX, TextPadY で確保する。

なお、Button などのいくつかの部品には TextPadX, TextPadY という属性もあり、表示文字の周囲にさらにすき間を確保することができます。

3.4 梱包のすき間の膨張 Expand

今、例えばウインドウの大きさが変更されるなどの原因で、ある部品の周りに余分なすき間が発生したとします。もし、その部品に Expand -> True という属性が指定してあると、その部品は周りのすき間を自分の梱包のすき間とします。次の例は Button の場合の Expand の効果を示すためのもので、図 6 のような結果になります。

```
w1 = Window[MinSize -> {200, 200}];
b1 = Button[w1, Expand -> True,
  Text -> "Expand -> True"];
b2 = Button[w1, Expand -> True,
  Text -> "Expand -> True"];
w2 = Window[MinSize -> {200, 200}];
b3 = Button[w2, Expand -> False,
  Text -> "Expand -> False"];
b4 = Button[w2, Expand -> False,
  Text -> "Expand -> False"];
```

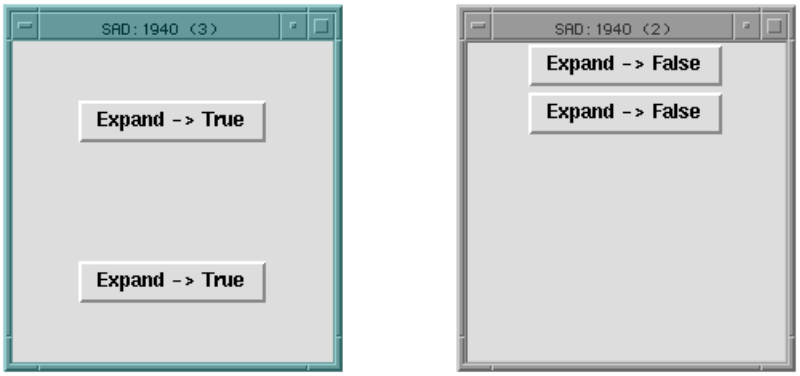

図 6: Expand の効果。

3.5 梱包のすき間の充填 Fill

Fill は Expand -> True などによって与えられた自分の梱包のすき間を自分の表示域として取り込むための属性です。これには Fill -> "x", Fill -> "y", Fill -> "both" の方向の指定が可能です。次の例は Button の場合の Fill の効果を示すためのもので、図 7 のような結果になります。

```
w1 = Window[MinSize -> {250, 200}];
b1 = Button[w1, Expand -> True,
  Fill -> "x",
  Text -> "Expand -> True, Fill ->\"x\""];
b2 = Button[w1, Expand -> True,
  Fill -> "y",
  Text -> "Expand -> True, Fill ->\"y\""];
b3 = Button[w1, Expand -> True,
  Fill -> "both",
  Text -> "Expand -> True, FIll ->\"both\""];
```

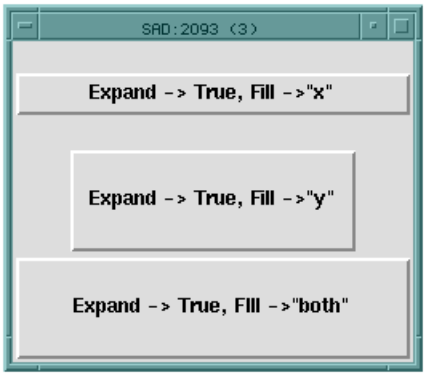

図 7: Fill の効果。

3.6 梱包の際の部品の表示域の基準点 Anchor

Anchor は Expand -> True などによって与えられた自分の梱包のすき間よりも自分の表示域が狭い場合、そのすき間の中のどこに表示域を設定するかを指定します。Anchor は "n", "ne", "e", "se", "s", "sw", "w", "nw", "c" の内の一つの値をとります。"c" は "center" と書いても構いません。Anchor の効果を図 8 に示します。

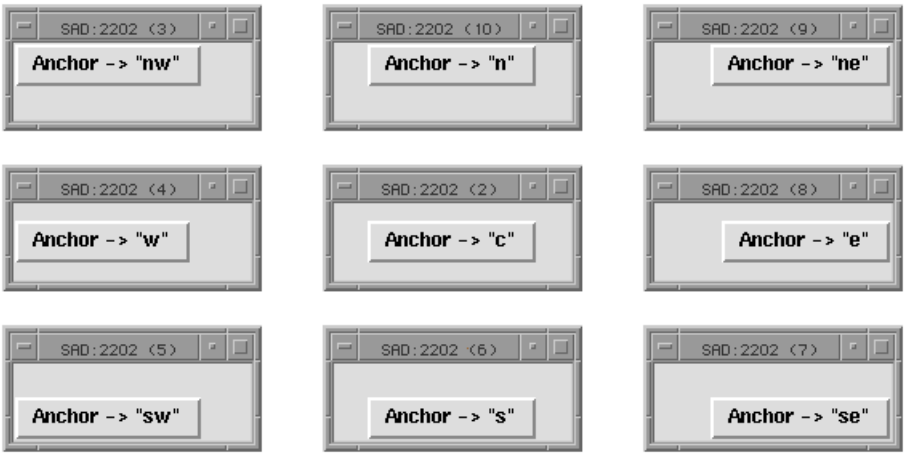

図 8: Anchor の効果。

また、Button などのいくつかの部品には属性 TextAnchor が指定でき、それにより表示テキストを部品の中のどちら側に表示するかを指定できます。TextAnchor のパラメータは Anchor と同じです。

4 SAD/Tkinter の部品の操作

この章では SAD/Tkinter に備え付けられている各種の部品にある程度共通する操作の概要を紹介します。現在までに組み込まれている部品は次の表のとおりです。

表 1: SAD/Tkinter の部品。

部品	機能	コマンド	結合変数
Window	各種部品を収容するウインドウ		
Frame	各種部品を収容する枠		
Button	単純なボタン	Command	TextVariable
CheckButton	ON/OFFのチェックマーク付ボタン	Command	Variable TextVariable
RadioButton	他者択一型のボタン	Command	Variable TextVariable
TextLabel	複数行のテキスト・ラベル		TextVariable
TextMessage	フォーマット可能なテキスト		TextVariable
Entry	文字列の入力枠		TextVariable
ListBox	複数行の文字列からなる選択箱		
ScrollBar	スクロール・バー		
Menu	プルダウン・メニュー	PostCommand	
MenuButton	Menu を割り付けるためのボタン		
Scale	スライド・スケール	Command	Variable
Canvas	グラフィック出力用の画面		
TextEditor	様々な修飾が可能なテキスト入出力		
OptionMenu	いくつかの選択枝から項目を選ぶ		

4.1 結合変数

表1にある、Button, CheckButton, RadioButton, TextLabel, TextMessage, Entry, Scale などの部品には、あるシンボルを結合変数として割り付けることができます。例えば、いまあるボタン d を

```
w = Window[];
d = Button[w, Width -> 20, TextVariable :> date];
update := After[1, date = DateString; update];
update;
```

のように定義したとします。ここで Width -> 20 はボタンの幅を20文字分確保することを指示しています。

TextVariable :> date が問題の結合変数の指定です。このボタンには Text によるボタンの上に書くテキストの指示はなく、代わりにシンボル date が割り当てられています。つまり、以後シンボル date に何らかの値（文字列）が割り当てられると、その値がこのボタンの表面に表

示されることになります。実際、この例では第3行でシンボル update に式 After[1, date = DateString; update] を割り当てています。After は SAD/Tkinter の備え付けの関数で、その第1引き数の時間だけ(単位: 秒)だけ経過した後に第2 引き数の式の実行を予約するというものです。したがって、update が一度呼ばれると、その第2 引き数にはまた update が含まれていますから、結局 update はほぼ 1秒に1回ずつ繰り返し実行されます。シンボル DateString は SAD に備え付けの関数で、その時の時刻を文字列で返します。こうしてシンボル date には時刻の文字列が毎秒設定され、ボタン d の表示も毎秒更新されます。

結合変数はまた Entry (5.8)のように、部品の側から入力したデータをプログラムの中で利用する場合にも有効です。

4.2 イヴェントと部品の結合

それぞれの部品にはXサーヴァで与えられる、マウス・クリックやキーボード入力などの多彩な動作 event にいかに反応するかを指定することができます。前述の、ボタンに対する Command の指定もそれの特殊な例ですが、より一般的には関数 Bind を用いることでどのようなイヴェントにも対応できます。Bind は

```
Bind[部品, イヴェント, 動作];
```

という書式で使います。例えば、

```
w = Window[];
d = Button[w, Text -> "Event"];
Bind[d, "<Enter>", Print[$Event]];
```

としますと、マウス・カーソルがボタン d の上に来たときに端末に $Event の内容が

```
{(Widget:>d),(Tag->""),(Type->"<Enter>"),(X->53),(Y->28),
 (XRoot->769),(YRoot->722),(Height->0),(Width->0),
 (Char->"??"),(KeySym->"??"),(SendEvent->0),(KeyCode->0),
 (State->0),(KeySymNum->0),(Time->15202658)}
```

のように出力されます。$Event はあるイヴェントが発生する度にそのイヴェントの情報が自動的に割り当てられるシンボルです。$Event は実際には 8.7 節にある式の置換を行うための規則のリスト (7 章を参照) の形をしています。表 2 に $Event に現われるシンボルの意味を掲げます。

イヴェントの表現は一般には

```
"<修飾子 - 修飾子 - 型 - 詳細>"
```

とされています。イヴェントの型には表 3 のようなものがあります。

表 2: $Event に現われるシンボル。

シンボル	値	意味
Widget	シンボル	イヴェントを発生した部品
Tag	アイテム番号を表わす文字列またはタグ	イヴェントを発生したアイテム
Type	イヴェントの型名	発生したイヴェントの型名
X, Y	数値(ピクセル)	マウスカーソルの部品内での位置
RootX, RootY	数値(ピクセル)	マウスカーソルのルートウインドウ内での位置
Height, Width	数値(ピクセル)	部品の大きさ
Char	文字	押されたキー
KeySym	文字列	押されたキーを表す
KeySymNum	数値	
KeyCode	数値	押されたキーのコード
Time	数値 (msec)	押された時刻、午前0時 = 0

Key, KeyRelease などにはキーの種類が詳細として付加されます。例えば、

```
"<Key-a>"
"<a>"
"a"
```

などはどれもキー a の押し下げを表わしています。また

```
Return, Escape, Up, Down, Left, Right
```

等々の KeySym も使うことができます。

イヴェントにはいくつかの修飾子をつけることができます。それらはキーボードの修飾キーに対応したものや、ダブル・クリックなどを表わすためのものです。

イヴェントのうち、キーの押し下げに関するイヴェントは焦点 focus の合っている部品のみが反応することができます。部品によっては Entry（5.8 参照）のようにその上でマウスクリックすることにより自動的に焦点が合う部品もありますが、そうでなければ自分で焦点を合わせなければなりません。ある部品に焦点を合わせるには、

```
部品シンボル[Focus$Set]
```

と、また、焦点を外すには

```
部品シンボル[Focus$None]
```

とします。

表 3: イヴェントの型。

型	動作
Button, ButtonPress	マウス・ボタンの押し下げ
ButtonRelease	マウス・ボタンの解放
Enter	マウス・カーソルが部品の内にはいる
Leave	マウス・カーソルが部品の外に出る
Motion	マウス・カーソルが部品の中で動く
Key, KeyPress	キーの押し下げ
KeyRelease	キーの解放
Configure	Window の位置、大きさなどの属性の変更
Destroy	Window が消去された
Expose	Window が露出した
FocusIn	Window に焦点があった
FocusOut	Window から焦点が外れた

表 4: イヴェント修飾子。

修飾子	動作
Control	コントロール・キー
Shift	シフト・キー
Lock	シフト・ロック・キー
Meta, M	メタキー
Alt	Alt キー
Button1 -- Button3	マウス・ボタン1–3
Double	ダブル・クリック
Triple	トリプル・クリック

4.2.1 イヴェントの結合の解除

イヴェントの結合を解除したい場合には、

```
Bind[部品, イヴェント]        または
Bind[部品, イヴェント, ]        または
Bind[部品, イヴェント, Null]
```

とします。

4.3 部品の操作

前にも述べた通り、ある部品は

```
シンボル = 部品[親, 属性 -> 値, ...];
```

という形で最初につくられます。属性 -> 値 の部分は 属性 :> 値 という形をとる場合もあります。以下、このように定義されたシンボルを「部品シンボル」と呼ぶことにします。ここで、親の

部品を省略することも可能です。その場合は、図 2 の ReturnToSAD のボタンが表示されている Window が親の部品とみなされます。

> 同じ属性を重複して指定した場合は、先に書いたものだけが用いられます。また、その部品に定義されていない属性を指定した場合は無視されます。

また、シンボルとしては単純なシンボルのほかに例えば a[1], a[1, 2, 3] などのような引き数付のシンボルでも構いません。このような引き数付のシンボルは多数の部品を機械的に生成する場合に便利です。

なお、上記のように生成された部品シンボルには

```
Widget[シンボル、部品[親, 属性 -> 値, ...]];
```

という値が割り当てられます。従って、ある部品シンボルの頭部（6.2、6.2.3 参照）は常に Widget になります。

属性によっては、複数の値を要求するものもあります。その場合には値としてリスト（7 章を参照）

```
{値1, 値2, ... }
```

が用いられます。

ある部品が生成された後は、

```
部品シンボル[属性] =  値;  または
部品シンボル[属性] := 値;
```

のようにして個別にその属性を変更することができます(個々の部品が具体的にどういう属性をもっているかは後述されます)。値がリストになることもあります。また、属性によっては、全然値を必要としないものもあります。その場合は単に

```
部品シンボル[属性];
```

と書きます。また、ある種の属性はその値を返すことできます。その場合は例えば

```
a = 部品シンボル[属性];
```

の様に使います。さらに、ある種の属性の場合には引き数を要求しながら値も返すというものがあり、その場合は

```
a = 部品シンボル[属性[引き数1, ...]];
```

とします。同一の属性でも書き方によって、値の設定、読み出しの両方を使い分けることができます。

また、一度に多数の属性を後から設定するには関数 Configure により

```
Configure[部品シンボル, 属性 -> 値, ...];
```

のようにします。

部品は一般にはその属性が設定されてもその瞬間には表示を変えません。そのためには、Update、TkSense、TkWait など（4.8 参照）が呼ばれなければなりません。TkWait 実行中は部品の表示は定期的に更新されます。また Update[] は、それまでに定義したすべての部品を更新します。

4.4 部品の親子関係

部品の親子関係はその製作時の指定で決まります。子供が親を指定する場合、親は既に必ず部品として製作されていなければなりません。しかし、子供が必ずしも親の部品の内側に描かれる必要はありません。例えば、Window を自分より小さな Button の子供とすることもできます。

4.5 各部品に共通な属性

さて、各部品にはいくつかの共通する属性があります。それらを表 5 に列挙しておきます。このなかで、梱包関係の属性、Anchor, Expand, Fill, IPadX, IPadY, PadX, PadY, Side は Window には適用されません。

表 5: 各部品に共通な属性。

属性	値(単位)	機能
Focus$None	なし	その部品の焦点を外す。
Focus$Set	なし	その部品に焦点を合わせる。
Lower	なし	その部品のウインドウを背景にさげる。
Raise	なし	その部品のウインドウを最全面にあげる。
Anchor	"n", "ne", "e", "se", "s", "sw", "w", "nw", "center"	梱包の際の部品の起点。
Expand	True, False	梱包の際の周辺のすき間の拡張。
Fill	"x", "y", "both", "none"	梱包の際の周辺のすき間の充填。
IPadX, IPadY	ピクセル	梱包の際の部品の内部のすき間。
PadX, PadY	ピクセル	梱包の際の部品の周辺のすき間。
Side	"top", "bottom", "left", "right"	梱包の向き。
TextAnchor	"n", "ne", "e", "se", "s", "sw", "w", "nw", "center"	表示テキストの配置起点
TextPadX, TextPadY	ピクセル	表示テキストの回りのすき間。
Background	色名称または"#RRGGBB"	背景色
BG	〃	Background の略称
BorderWidth	ピクセルまたは単位付文字列	境界(立体表示される)の幅
BD	〃	BorderWidth 略称
Cursor	形状文字列(図10)	マウス・カーソルの形状
Height	ピクセルまたは単位付文字列	部品の高さ
HighlightColor	色名称または"#RRGGBB"	焦点が合った時の背景色
Relief	"flat", "groove", "raised", "ridge", "sunken"	枠の立体形状
Width	ピクセル、文字数、または単位付文字列	部品の幅

表 5 で「単位付文字列」とは、例えば 1 cm を"1c"、1 inch を "1i"などと表わすものです。

各属性の単位やデフォルト値は部品の種類によって異なります。以後、各部品の中で標準と異なるばあいにはそのつど示します。

4.5.1 部品の枠の立体形状

属性 Relief で指定する、部品の枠の各種の立体形状を図 9 に示します。

図 9: 部品の枠の立体形状。Relief -> 形状 で指定する。

4.5.2 フォント

Font 属性により、文字のフォントを指定する場合は、例えば、

```
Font -> TextFont["times","italic","bold",12]
```

の様に指定します。ただし、どのフォントが使用可能かはXサーヴァに依存します。

4.5.3 色

Background 属性などで、色を指定する場合は、

```
Background -> "red"          または
Background -> "#FF0000"
```

などとします。ここでどのような色名称が使えるかはXサーヴァに依存します。また、RGB指定は"#RRGGBB" のように16進表記の文字列を使います。この桁数はXサーヴァに依存します。

4.5.4 マウスカーソルの形状

ある部品にマウスカーソルが入ったときのカーソルの形状は属性 Cursor を使って、

```
Cursor -> "カーソル名"                    または
Cursor -> "カーソル名 文字色"             または
Cursor -> "カーソル名 文字色 背景色"
```

のように指定することができます。ここでカーソル名称としては図 10 にある備え付けの物のほかに、"@ファイル名" と書けばファイルからもロードできます。

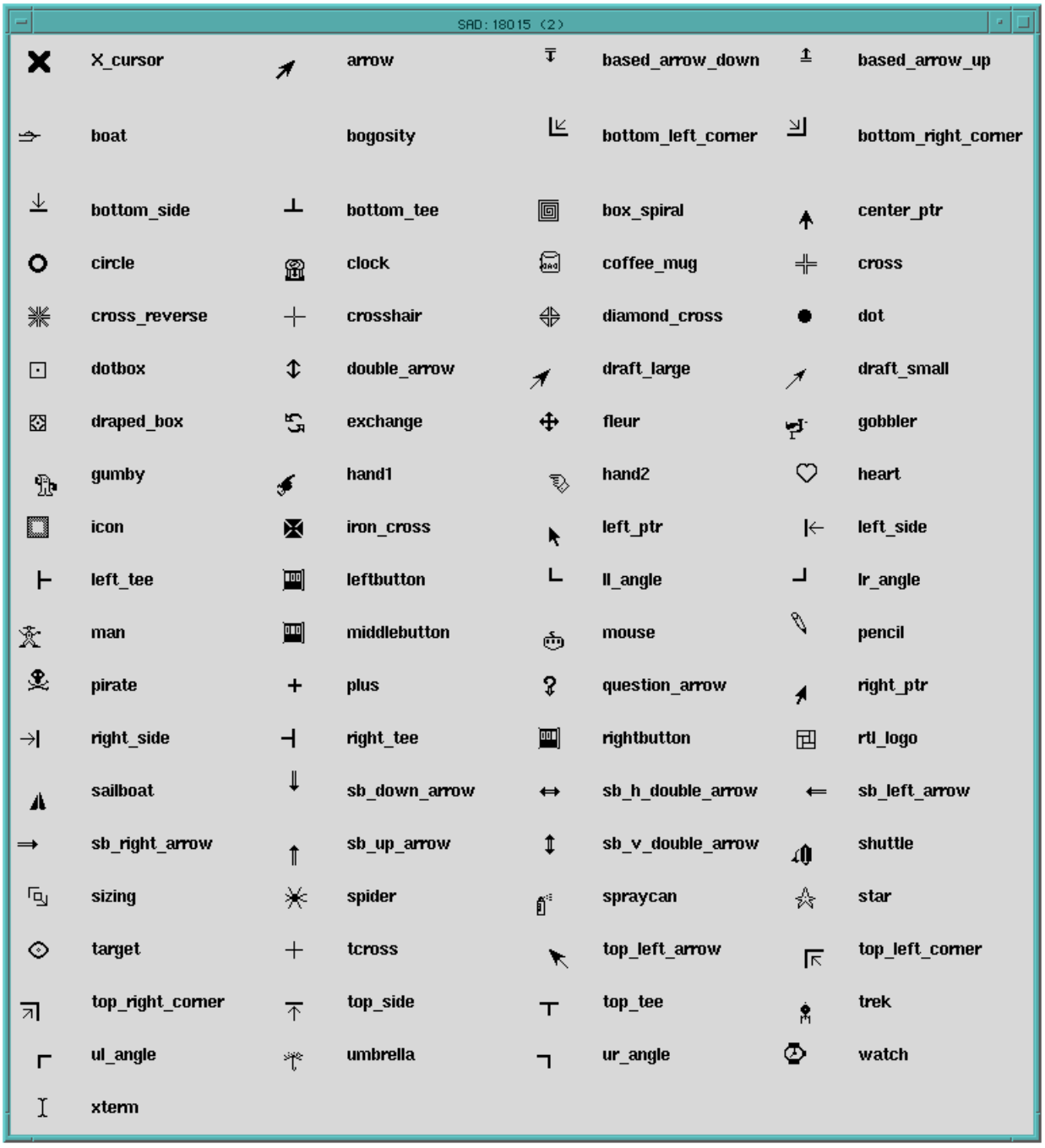

図 10: 備え付けのマウスカーソル。

4.5.5 ビットマップ

いくつかの部品ではビットマップによる塗りつぶしが可能です。ビットマップには備え付けのものとユーザが作製してファイルに保存されているものの2種類があります。この内、備え付けのものは図 11 にある8種類です。ビットマップの指定は文字列で、備え付けのものはその名称で、ファイルのものは "@ファイル名" の様に書きます。なお、ビットマップは Unix の bitmap コマンドで簡単に作製できます。

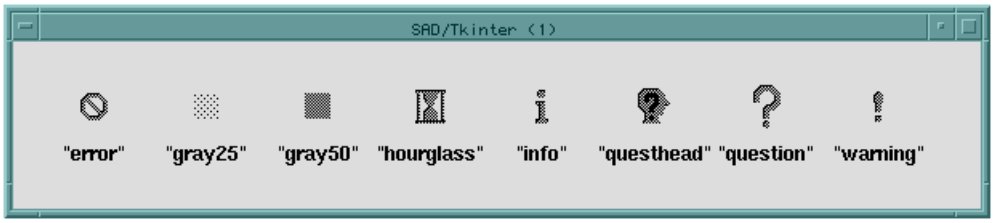

図 11: 備え付けのビットマップ。

4.6 部品の情報の獲得

作成された各々の部品についてはWidgetInformationという関数を用いて各種の情報をうることができます。これは

```
a = WidgetInformation[部品シンボル, 属性];  または
a = WidgetInformation[部品シンボル, 属性, id];
```

のように書きます。指定可能な属性は表 6 に掲げます。

表 6: WidgetInformation の属性。

属性	機能
Geometry	その部品のサイズおよび位置(ピクセル)、"WWWxHHH+dX+dY"
Height	その部品の高さ(ピクセル)
Width	その部品の幅(ピクセル)
X, Y	その部品のウインドウ内の水平、垂直位置(ピクセル)
RootX, RootY	その部品のスクリーン内の水平、垂直位置(ピクセル)
Screen	その部品の表示スクリーン(DISPLAY 変数)
ScreenDepth	その部品の表示スクリーンのビット数
ScreenHeight, ScreenWidth	その部品の表示スクリーンの高さ及び幅(ピクセル)
ReqHeight, ReqWidth	その部品のウインドウに必要な最小高さ、幅

また、属性をリスト（7 章参照）にして

```
a = WidgetInformation[部品シンボル, {属性1, 属性2, ...}];
```

の様にすると結果が属性1，属性2，...　に対応するリストとして返されます。例えば、

```
   w = Window[];
   WidgetInformation[w, {ScreenWidth, ScreenHeight}]
Out[1]:= {2304,1720}
```

のような具合です。

4.6.1 FromGeometry

- 表 6 で属性 Geometry は "幅x高さ+x位置+y位置" という特殊なフォーマットで値（ピクセル）を返しますが、関数 FromGeometry を使うことにより、{幅，高さ，x位置，y位置} という数値リストに変換することができます。逆関数は ToGeometry です。
- 位置はその部品の親の左上隅から右下に測ります。親が指定されていない部品はスクリーン上の位置が返されます。

4.6.2 WidgetGeometry

- WidgetGeometry[部品] はその部品の {幅，高さ，x位置，y位置} というリストを返します。単位はピクセルです。
- WidgetGeometry[部品] は FromGeometry[WidgetInformation[部品, Geometry]] と同値です。

4.6.3 ToGeometry

- ToGeometry[{幅，高さ，x位置，y位置}] は Geometry フォーマットの文字列 "幅x高さ+x位置+y位置" を返します。
- x，y は左上隅から右下に測ります。もし負の数が指定されると右下隅から測られます。
- ToGeometry[{幅，高さ] は "幅x高さ" を返します。
- ToGeometry[{ ，，x位置，y位置}] は "x位置+y位置" を返します。

4.7 部品の消去

プログラムの終了、切り替えなどですでに製作した部品を必要としなくなった場合には、必ず

```
DeleteWidget[部品シンボル1, 部品シンボル2, ... ];
```

のように、関数 DeleteWidget により、不要な部品を消去してください。また、

```
部品シンボル1 =.
```

は DeleteWidget[部品シンボル1] と同様の働きをします。

- DeleteWidget は親に対して行えば、**その親に属する子供も全て消去**されます。
- DeleteAllWidgets[] は全ての部品を消去します。
- DeleteWidget[a] で a 自身が部品でない場合には、a[...] の形のシンボルに割り当てられている部品を**全て**消去します。

ある部品に結合した結合変数は、その変数を最初に結合した部品を消去した時に自動的に消去されます。また、結合変数は

```
DeleteVariable[変数1, 変数2, ... ];
```

のように、DeleteVariable を使って消去することもできます。

4.8 SAD/Tkinter の制御

前にも述べましたが、SAD/Tkinter の部品はただ定義しただけでは表示も何もされません。表示、更新、イヴェントへの反応などは通常、関数 TkWait[], TkSense[]、あるいは Update[] が呼ばれたときにだけ起こります。

4.8.1 TkWait、TkReturn

- TkWait[] はそれまでの部品の定義に基づき、全ての部品を製作・更新し、イヴェント待ちの状態になります。イヴェントが発生すれば各部品の定義に応じた動作が起こされます。
- 結合されたコマンドのなかで TkReturn[式1] が実行されると、TkWait は中断され、式1の値が TkWait[] の結果として返されます。
- ボタン ReturnToSAD（2 参照）は TkReturn["ReturnToSAD"] を実行します。
- TkWait[] はイヴェントに結び付けられたコマンドの中で何重にも呼ぶことができます。これはダイアログ・ボックスなどでユーザーの応答を待つときなどに使うことができます。

4.8.2 TkSense

- TkSense[秒] は指定された秒数だけ TkWait[] の動作をし、その間に TkReturn[式2] が実行されると式2 の結果を直ちに返します。そうでなければ指定時間の後に Null を返します。
- TkSense[] は TkSense[0.3] と同値です。

4.8.3 Update

- Update[] はそれまでの部品の定義に基づき、全ての部品を製作・更新します。

4.8.4 WaitExpression

- WaitExpression[式1] は式1 の値が変化するまで TkWait[] の動作をしつづけます。その間に TkReturn[式2] が実行されると式2 の結果を直ちに返します。

4.8.5 After

- After[秒, 式1] は指定された秒数だけ経過した後に式1 を評価することを予約します。
- After は TkWait や TkSense が呼ばれた時に式1 の実行に移ります。

4.8.6 Bell

- Bell[] は X サーヴァのベルを 1 回鳴らします。

5 SAD/Tkinter の各種部品

5.1 Window

Window は他の部品とは違い、例えば梱包の属性を持たないなど、特異性があります。したがって、これを部品とはしないという考えもあります。

表 7: Window の属性。

属性	入出力値(単位)	機能
Deiconify	なし	アイコン状態・隠れ状態から復帰
Iconify	なし	アイコン状態にする
Geometry	"WWWxHHH+dX+dY"（ピクセル）	サイズおよび位置
MaxSize	$\{H, W\}$（ピクセル）	最大拡大可能サイズ
MinSize	$\{H, W\}$（ピクセル）	最小縮小可能サイズ
OverrideRedirect	True, False	Trueの時、枠なしのウインドウ
State	"normal", "iconic", "withdrawn"	ウインドウの状態
Title	文字列	タイトルバーに書く文字列
Withdraw	なし	ウインドウを隠す

表 7 で属性 Geometry は "幅x高さ+x位置+y位置" という特殊なフォーマットの文字列を要求しますが、関数 ToGeometry(4.6.3 参照) を使えば {幅，高さ，x位置，y位置} から "幅x高さ+x位置+y位置" への変換ができます。

Window にはこれ以外にも特殊な操作が可能です。そのひとつは AdjustWindowGeometry で、Window のサイズを中に梱包された部品の表示に最低必要な大きさに変更するものです。用法:

```
AdjustWindowGeometry[Windowシンボル];
```

5.2 Frame

Frame は Window の中の様々な部品を整列させて収容する枠です。また、この枠の立体的な表示や枠および背景の着色などもできます。例えば

```
w = Window[];
f1 =  Frame[w, Relief -> "raised",
  Side -> "left", BorderWidth -> 5];
b11 = Button[f1, Text -> "Frames",
  Side -> "top", PadX -> 20, PadY -> 10];
b12 = Button[f1, Text -> "are used",
  Side -> "top", PadX -> 20, PadY -> 10];
```

```
b13 = Button[f1, Text -> "to align",
  Side -> "top", PadX -> 20, PadY -> 10];

f2 =  Frame[w, Relief -> "ridge",
  Side -> "left", BorderWidth -> 5];
b21 = Button[f2, Text -> "widgets",
  Side -> "top", PadX -> 20, PadY -> 10];
b22 = Button[f2, Text -> "in a window.",
  Side -> "top", PadX -> 20, PadY -> 10];

f3 =  Frame[f2, Relief -> "sunken",
  Side -> "top", BorderWidth -> 5,
    PadX -> 10, PadY -> 10];
b31 = Button[f3, Text -> "Frames",
  Side -> "left", PadX -> 20, PadY -> 10];
b32 = Button[f3, Text -> "can be nested.",
  Side -> "left", PadX -> 20, PadY -> 10];
```

のようにしますと、図 12 のような結果になります。この場合枠の形状は Relief と BorderWidth で決まりますが、デフォルトではBorderWidth -> 0 なので Relief だけでは何も効果はありません。Frame はいくらでも重ねられます。

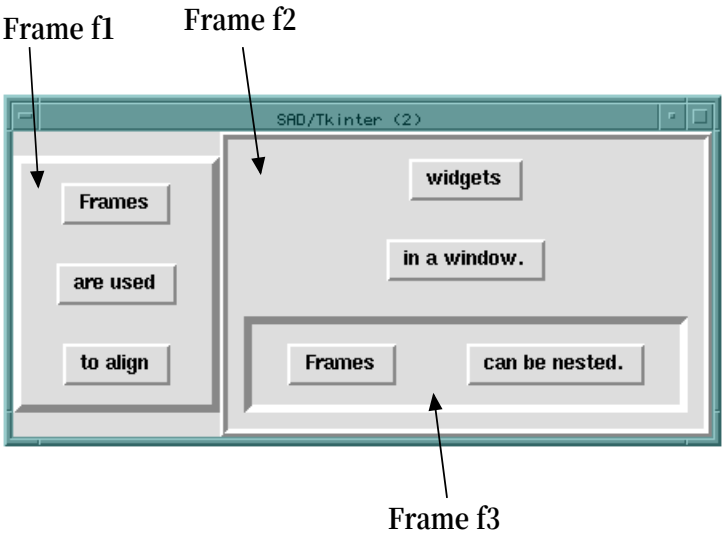

図 12: Frame の用例。

5.3 Button

Button の属性を次の表に列挙します。

表 8: Button の属性。

属性	入出力値(単位)	デフォルト	機能
ActiveBackground	色		マウスカーソルが上にあるときの背景色
ActiveForeground	色		マウスカーソルが上にあるときの文字色
Bitmap			表面にbitmapを表示
Command	式		ボタン押下時に実行する式
DisabledForeground	色		
Foreground	色	"black"	文字色
FG	〃	〃	Foreground の略称
Font	フォント		文字フォント
Text	文字列		ボタン表面の表示文字列
TextVariable	シンボル		ボタン表面の表示文字列を 割り当てるシンボル
BorderWidth(BD)	ピクセル	2	境界(立体表示される)の幅
Width	文字数		ボタンの幅
Relief	"flat", "groove", "raised", "ridge", "sunken"	"raised"	枠の立体形状
TextAnchor	"c", "n", "ne", "e", "se", "s", "sw", "w", "nw", "center"	"c"	表示テキストを揃える方向
TextPadX	ピクセル	9	表示テキストの左右のすき間
TextPadY	ピクセル	3	表示テキストの上下のすき間
Flash	なし	なし	ボタンを一瞬点滅
Invoke	なし	なし	割り当てられた Command を実行

5.4 CheckButton

CheckButtonはON/OFFを表わすチェックマーク付のボタンです。これは Variable -> シンボル で与えられたシンボル(結合変数)にOnValue -> 値 で決められた値が割り当てられると、チェックマークが点灯するようになっています。それ以外の場合には、たとえその値が OffValue -> 値 で決めた値でなくても、チェックマークは消灯します。また、結合変数を評価すると、その時点のボタンの状態に応じてOnValue -> 値 または OffValue -> 値 で設定された値を返します。また、複数の CheckButton が一つの変数を共有する場合は、その評価は最初に定義した CheckButton の状態を返しますが、その変数への値の設定はすべての CheckButton に反映されます。例図は

```
w = Window[];
b1 = CheckButton[w, Text -> "Linac/BT OK",
  Variable :> linac];
b2 = CheckButton[w, Text -> "LER OK",
  Variable :> ler];
b3 = CheckButton[w, Text -> "HER OK",
  Variable :> her];
b4 = CheckButton[w, Text -> "Belle OK",
  Variable :> belle];
linac = 1;
ler = 0;
her = 1;
belle = 1;
```

により作成しました。

図 13: CheckButton の用例。

表 9: CheckButton の属性。

属性	入出力値(単位)	デフォルト	機能
ActiveBackground	色		マウスカーソルが上にあるときの背景色
ActiveForeground	色		マウスカーソルが上にあるときの文字色
Bitmap			ボタン表面にbitmapを表示
Command	式		ボタン押下時に実行する式
DisabledForeground	色		
Foreground(FG)	色	"black"	文字色
Font	フォント		文字フォント
Text	文字列		ボタン表面の表示文字列
TextVariable	シンボル		ボタン表面の表示文字列を割り当てるシンボル
BorderWidth(BD)	ピクセル	2	境界(立体表示される)の幅
Width	文字数		ボタンの幅
Relief	"flat", "groove", "raised", "ridge", "sunken"	"flat"	枠の立体形状
Variable	シンボル		結合変数
OffValue	数値	0	ボタンが OFF の時に返す値
OnValue	数値	1	ボタンを ON に設定する値
TextAnchor	"c", "n", "ne", "e", "se", "s", "sw", "w", "nw", "center"	"c"	表示テキストを揃える方向
TextPadX	ピクセル	9	表示テキストの左右のすき間
TextPadY	ピクセル	3	表示テキストの上下のすき間
Flash	なし	なし	ボタンを一瞬点滅
Invoke	なし	なし	割り当てられた Command を実行
Deselect	なし	なし	ボタンを OFF にする
Select	なし	なし	ボタンを ON にする
Toggle	なし	なし	ボタンの状態を反転する

5.5 RadioButton

RadioButton は他者択一型の選択を行うためのマーク付のボタンです。これは Variable -> シンボル で与えられたシンボル(結合変数)に値が割り当てられると、その値がValue -> 値 で決められた値に一致する RadioButton のマークが点灯するようになっています。それ以外の場合にはマークは消灯します。また、結合変数を評価すると、その時点のボタンの状態に応じて値を返します。例図は

```
w = Window[];
b1 = RadioButton[w, Text -> "2 ns spacing",
  Variable :> sb, Value -> 2];
b2 = RadioButton[w, Text -> "4 ns spacing",
  Variable :> sb, Value -> 4];
b3 = RadioButton[w, Text -> "6 ns spacing",
  Variable :> sb, Value -> 6];
sb = 2;
```

で作成しました。

図 14: RadioButton の用例。

表 10: RadioButton の属性。

属性	入出力値(単位)	デフォルト	機能
ActiveBackground	色		マウスカーソルが上にあるときの背景色
ActiveForeground	色		マウスカーソルが上にあるときの文字色
Bitmap			ボタン表面にbitmapを表示
Command	式		ボタン押下時に実行する式
DisabledForeground	色		
Foreground(FG)	色	"black"	文字色
Font	フォント		文字フォント
Text	文字列		ボタン表面の表示文字列
TextVariable	シンボル		ボタン表面の表示文字列を 割り当てるシンボル
BorderWidth(BD)	ピクセル	2	境界(立体表示される)の幅
Width	文字数		ボタンの幅
Relief	"flat", "groove", "raised", "ridge", "sunken"	"flat"	枠の立体形状
Variable	シンボル		結合変数
Value	数値		ボタンを ON にする値
TextAnchor	"c", "n", "ne", "e", "se", "s", "sw", "w", "nw", "center"	"c"	表示テキストを揃える方向
TextPadX	ピクセル	9	表示テキストの左右のすき間
TextPadY	ピクセル	3	表示テキストの上下のすき間
Flash	なし	なし	ボタンを一瞬点滅
Invoke	なし	なし	割り当てられた Command を実行
Deselect	なし	なし	ボタンを OFF にする
Select	なし	なし	ボタンを ON にする

5.6 TextLabel

TextLabelは1行または複数行の文字列を表示するラベルです。その内容は Text -> 文字列 か或いは TextVariable -> シンボル で指定します。後者の場合にはその結合変数の内容が文字列に変換され表示されます。また、文字列の中に "\n" があればそこで改行します。

表 11: TextLabel の属性。

属性	入出力値(単位)	デフォルト	機能
Bitmap			bitmapを表示
Foreground(FG)	色	"black"	文字色
Font	フォント		文字フォント
Justify	"left", "center", "right"	"center"	行内の語の整列
Text	文字列		表示文字列
TextVariable	シンボル		表示文字列を割り当てるシンボル
BorderWidth(BD)	ピクセル	2	境界(立体表示される)の幅
Width	文字数		表示の幅
Relief	"flat", "groove", "raised", "ridge", "sunken"	"flat"	枠の立体形状
TextAnchor	"c", "n", "ne", "e", "se", "s", "sw", "w", "nw", "center"	"c"	表示テキストを揃える方向
TextPadX	ピクセル	1	表示テキストの左右のすき間
TextPadY	ピクセル	1	表示テキストの上下のすき間

図 15: TextLabel の用例。

5.7 TextMessage

TextMessageは長い文字列を決められた幅でフォーマットしながら表示します。その内容は Text -> 文字列 か或いは TextVariable -> シンボル で指定します。後者の場合にはその結合変数の内容が文字列に変換され表示されます。

表 12: TextMessage の属性。

属性	入出力値(単位)	デフォルト	機能
Aspect	%	150	幅/高さ
Foreground(FG)	色	"black"	文字色
Font	フォント		文字フォント
Justify	"left", "center", "right"	"left"	行内の語の整列
Text	文字列		表示文字列
TextVariable	シンボル		表示文字列を割り当てるシンボル
BorderWidth(BD)	ピクセル	2	境界(立体表示される)の幅
Width	ピクセル	116	表示の幅
Relief	"flat", "groove", "raised", "ridge", "sunken"	"flat"	枠の立体形状
TextAnchor	"c", "n", "ne", "e", "se", "s", "sw", "w", "nw", "center"	"c"	表示テキストを揃える方向
TextPadX	ピクセル	1	表示テキストの左右のすき間
TextPadY	ピクセル	1	表示テキストの上下のすき間

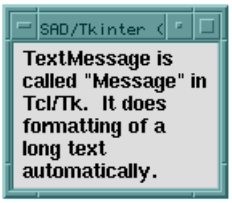

図 16: TextMessage の用例。

5.8 Entry

Entryは文字列をキーボードから入力するための枠です。TextVariable であるシンボルを結合変数に指定することにより、入力された文字列は即座に利用できます。

表 13: Entry の属性。

属性	入出力値(単位)	デフォルト	機能
ExportSelection	True, False	True	選択範囲を X に伝達
Foreground(FG)	色	"black"	文字色
Font	フォント		文字フォント
InsertBackground	色	"black"	挿入カーソルの色
InsertOffTime	msec		挿入カーソルの消灯時間
InsertOnTime	msec		挿入カーソルの点灯時間
InsertWidth	ピクセル	2	挿入カーソルの幅
SelectBackground	色	"green"	選択範囲の背景色
SelectForeground	色	"black"	選択範囲の文字色
SelectBorderWidth	ピクセル	1	選択範囲の境界幅
ShowText	文字	""	内容の代わりに表示する文字
State	"disabled", "normal"	"normal"	disabled = 読み出し専用
XScrollCommand	コマンド		横方向のスクロールバーの割り当て
Justify	"left", "center", "right"	"left"	行内の語の整列
TextVariable	シンボル		入力文字列を割り当てるシンボル
BorderWidth(BD)	ピクセル	2	境界(立体表示される)の幅
Width	文字数	20	表示の幅
Relief	"flat", "groove", "raised", "ridge", "sunken"	"sunken"	枠の立体形状

Entryには表13のような数々の入力編集機能が組み込まれています。

表 14: Entry の入力編集機能。

イヴェント	機能
"<Button-1>"	挿入点を指定
"<Control-Button-1>"	選択範囲を保存して挿入点を指定
"<B1-Motion>"	選択範囲をドラッグで指定
"<Shift-B1-Motion>"	選択範囲の境界をドラッグで変更
"<Double-Button-1>"	1語を選択
"<Triple-Button-1>" "<Control-slash>"	全体を選択
"<Button-2>"	挿入点に貼り込む
"<B2-Motion>"	横スクロール
"<Left>" "<Control-b>"	カーソルを1文字左
"<Shift-Left>"	カーソルを1文字左、選択範囲を拡張
"<Control-Left>" "<Meta-b>"	カーソルを1語左
"<Control-Shift-Left>"	カーソルを1語左、選択範囲を拡張
"<Right>" "<Control-f>"	カーソルを1文字右
"<Shift-Right>"	カーソルを1文字右、選択範囲を拡張
"<Control-Right>" "<Meta-f>"	カーソルを1語右
"<Control-Shift-Right>"	カーソルを1語右、選択範囲を拡張
"<Home>" "<Control-a>"	カーソルを Entry の開始点に
"<Shift-Home>"	カーソルを Entry の開始点に、選択範囲を拡張
"<End>" "<Control-e>"	カーソルを Entry の終点に
"<Shift-End>"	カーソルを Entry の終点に、選択範囲を拡張
"<Select>" "<Control-Space>"	(選択範囲を挿入点に合わせる)
"<Shift-Select>" "<Control-Shift-Space>"	選択範囲を挿入点に合わせる
"<Control-backslash>"	全体を消去
"<Delete>"	左1文字または選択範囲を削除
"<Backspace>" "<Control-h>"	左1文字を削除
"<Control-d>"	右1文字を削除
"<Control-w>"	左1語を削除
"<Meta-d>"	右1語を削除
"<Condrol-k>"	終端まで削除
"<Control-x>"	選択範囲を削除
"<Control-t>"	文字の並び変え

Entryには様々な操作を加えることができます。

次の例は Entry の最も簡単な用例で、その結果は図 17 の様になります。

```
w = Window[];
f1 = Frame[w];
t1 = TextLabel[f1, Text -> "Username: ",
  Side -> "left", PadX -> 10, PadY -> 10];
e1 = Entry[f1, TextVariable :> user,
  Side -> "left", PadX -> 5];

f2 = Frame[w];
t2 = TextLabel[f2, Text -> "Password: ",
  Side->"left", PadX -> 10, PadY -> 10];
e2 = Entry[f2, TextVariable :> pwd,
  Side -> "left", PadX -> 5,
  ShowText -> "*"];
```

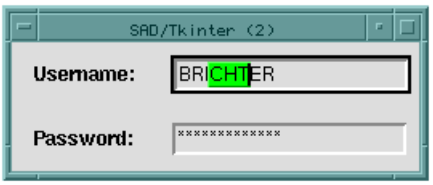

図 17: Entry の用例。

5.9 Scale

Scale は槽 trough とその上をすべるスライダーとからなりたっています。槽はある量の可変範囲を、またスライダーはその現在値を表わします。Scale に結合変数を結び付けておけば、それを通してスライダーの読み出し、設定がともに可能になります。

次の例は Scale の最も簡単な使用例です。

```
w = Window[];
s = Scale[w,
  From -> -10,
  To   ->  10,
  Length -> 200,
  Orient -> "horizontal",
  Variable :> v,
  Command :> Print[{$Arg, v}]];
```

ここで From と To はそれぞれ可変範囲の下限と上限を与えます。また、Length と Orient は Scale 全体の長さと向きを指定します。この場合の結合変数は v で、それはまた、Command のなかでも引用されています。Command はスライダーの位置に変化があった場合に実行される式を指定します。また、$Arg はこのコマンドが実行される時にスライダーの値として渡されるもので、v と同じ値を持っています。この例は図 18のような結果をもたらします。この例ではスライダーが動かされる度に端末に {-3,-3}のようにスライダーの示す値が印刷されます。

図 18: Scale の用例。

Scale にはまた、表 15 のような属性があります。

表 15: Scale の属性。

属性	入出力値(単位)	デフォルト	機能
BigIncrement	数値		大ステップの大きさ
Command	コマンド		スライダーが変化した際に実行
Digits	正整数		値の有効桁
Foreground(FG)	色	"black"	文字色
Font	フォント		ラベル用フォント
From	数値		下限値
To	数値		上限値
Label	文字列		ラベル文字列
Length	ピクセル	100	槽の長さ
Orient	"horizontal" "vertical"	"vertical"	槽の向き
Resolution	数値		結果はこの値の整数倍
ShowValue	True, False	True	値の表示
SliderLength	ピクセル		スライダーの長さ
State	"normal" "active" "disabled"		
TickInterval	数値	0	ティックの間隔、0でティックなし
TroughColor	色		槽の色
Variable	シンボル		結合変数
BorderWidth(BD)	ピクセル	2	テキストの境界の幅
Width	ピクセル	15	槽の太さ
Relief	"flat", "groove", "raised", "ridge", "sunken"	"flat"	枠の立体形状

Scale には次の表に示すいくつかの入力機能が用意されています。

表 16: Scale の入力機能。

イヴェント	機能
"<Button-1>"	槽をクリックするとスライダーが1ステップ進む
"<Control-Button-1>"	スライダーが端まで進む
"<Right>" "<Up>"	スライダーが1ステップ増加
"<Control-Right>" "<Control-Up>"	スライダーが1大ステップ増加
"<Left>" "<Down>"	スライダーが1ステップ減少
"<Control-Left>" "<Control-Down>"	スライダーが1大ステップ減少
"<Home>"	スライダーが最上端或いは最左端に移動
"<End>"	スライダーが最下端或いは最右端に移動

5.10 ScrollBar

ScrollBar は他の部品、Entry, ListBox, TextEditor, Canvas などと一緒に使われ、スクロール・バーの役目を果たします。ScrollBar と他の部品との結合法はきわめて単純で、以下のようにします。

```
physicists =
  {"Copernicus", "Galileo Galilei", "Kepler",
    "Hooke", "Newton", "Euler", "Lagrange",
    "Gauss", "Faraday", "Maxwell", "Boltzmann",
    "Lorentz", "Einstein", "Bohr", "Heisenberg",
    "Schrodinger", "Pauli", "Dirac", "Fermi"};

w = Window[];
sb = ScrollBar[w, Orient -> "vertical",
  Side -> "right", Fill->"y"];
lb = ListBox[w, YScrollCommand :> sb[Set],
  Insert -> {"end", physicists},
  Side -> "right"];
```

このように、(1)ScrollBar を先に定義する。(2)結び付けたい部品の中で

```
スクロールコマンド :> スクロールバーシンボル[Set]
```

とすればOKです。この場合、スクロールコマンドは YScrollCommand、スクロールバーシンボルは sb です。定義する順序を逆にしてはいけません。なお、ListBox については後述します。

ちなみにこの例は次のような結果になります。

ScrollBar の入力機能と属性を表 17・18 にまとめておきます。

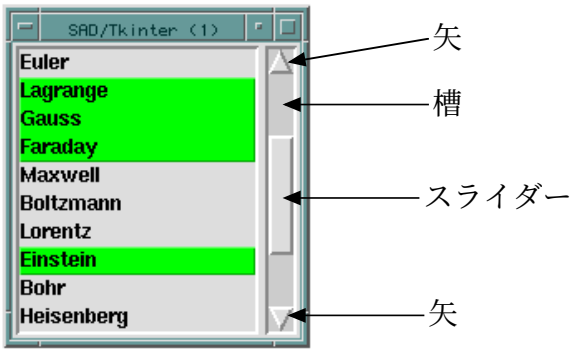

図 19: ScrollBar の用例。

表 17: ScrollBar の入力機能。

イヴェント	機能
"<Button-1>" "<Button-2>"	矢のクリックでスライダーが1ステップ進む
"<B1-Motion>" "<B2-Motion>"	スライダーのドラッグ
"<Control-Button-1>" "<Control-Button-2>"	スライダーを端まで動かす
"<Up>" "<Down>"	1行のスクロール
"<Control-Up>" "<Control-Down>"	1画面の上下スクロール
"<Left>" "<Right>"	1単位の左右スクロール
"<Control-Left>" "<Control-Right>"	1画面の左右スクロール
"<Home>"	スライダーが最上端或いは最左端に移動
"<End>"	スライダーが最下端或いは最右端に移動

表 18: ScrollBar の属性。

属性	入出力値(単位)	デフォルト	機能
Command	コマンド	(自動設定)	スライダーが変化した際に実行
Jump	True, False	False	True:ジャンプ・スクロール
Orient	"horizontal" "vertical"	"vertical"	槽の向き
TroughColor	色		槽の色
BorderWidth(BD)	ピクセル	2	境界の幅
Width	ピクセル		槽の幅

5.11 ListBox

ListBox は文字列の集合から一つまたは複数の要素を選び出すための道具です。基本的な動作は ListBox 作成時に文字列の集合を渡し、選択の完了後に選ばれた項目を受け取るというものです。選択は属性 SelectMode の設定によって単数にも複数にもできます。また、文字列の集合を随時挿入・削除するなど細かい動作も可能です。

SAD では要素の集合体はリスト（7 章を参照）で表わされます。リストは全体を { と } で囲まれ、要素の間を , で区切って表現します。図 19 の例では、

```
physicists =
  {"Copernicus", "Galileo Galilei", "Kepler",
    "Hooke", "Newton", "Euler", "Lagrange",
    "Gauss", "Faraday", "Maxwell", "Boltzmann",
    "Lorentz", "Einstein", "Bohr", "Heisenberg",
    "Schrodinger", "Pauli", "Dirac", "Fermi"};
```

のように、シンボル physicists に右辺の文字列を要素とするリストを割り当てています。さて、ListBox に対しては属性 Insert を用いて、

```
lb = ListBox[w, YScrollCommand :> sb[Set],
  Insert -> {"end", physicists},
  Side -> "right"];
```

のようにリスト physicists を伝達しています。この例ではリストを一旦シンボル physicists に割り当てていますが、もちろん

```
lb = ListBox[w, YScrollCommand :> sb[Set],
  Insert -> {"end",
    {"Copernicus", "Galileo Galilei", "Kepler",
      "Hooke", "Newton", "Euler", "Lagrange",
      "Gauss", "Faraday", "Maxwell", "Boltzmann",
      "Lorentz", "Einstein", "Bohr", "Heisenberg",
      "Schrodinger", "Pauli", "Dirac", "Fermi"}},
  Side -> "right"];
```

のように直接書き込んでもかまいません。

さて、Insert の最初の引き数の "end" は文字列の挿入点を全体の最後にするという意味です。ListBox ではこのような行の指定は表 19 にある何れかの方法で行うことができます。例えば、先頭から挿入する場合には 0 を指定します。

表 19: ListBox の行の指定方法。

値	意味
数値 n	最初から n行目、先頭 = 1
"active"	活性化された行
"anchor"	Anchor 指定された行
"end"	最終行
"@x, y"	座標が（x, y）に最も近い行

ListBox には4つの選択モード SelectMode、"browse", "single", "extended", "multiple" があります。このうち前2者が単数項目の選択、後2者が複数項目の選択です。それぞれのモードでは入力のイヴェントの結合に若干の差異があります。筆者は、通常はこのように4つものモードを使い分ける必要はなく、単に "browse" と "extended" の2者で充分ではないかと思います。ここではとりあえずこの2者の入力機能を説明し、他は参考文献を参照していただくことにします。

5.11.1 "browse" モード

"browse" モードはデフォルトの選択モードです。ここでは選択される項目は一つに限られています。

表 20: "browse" モードの入力機能。

イヴェント	機能
"<Button-1>"	クリックした項目を選択・活性化
"<B1-Motion>"	ドラッグで選択項目が移動
"<Shift-Button-1>"	クリックした項目を活性化、非選択
"<Up>" "<Down>"	活性化項目を1行上下
"<Control-Home>"	最初の行を選択・活性化
"<Control-End>"	最後の行を選択・活性化
"<Left>" "<Right>"	1単位の左右スクロール
"<space>" "<Select>" "<Control-slash>"	活性化した行を選択

5.11.2 "extended" モード

"extended" モードでは任意の複数の項目を選択できます。

表 21: "extended" モードの入力機能。

イヴェント	機能
"<Button-1>"	クリックした項目を選択、選択の起点にする
"<B1-Motion>"	起点からドラッグした範囲を選択
"<Button-Release-1>"	その項目を活性化
"<Shift-Button-1>"	起点からクリックした項目までの範囲を選択
"<Shift-B1-Motion>"	起点からドラッグした範囲を連続的に選択
"<Control-Button-1>"	クリックした項目の選択を反転、起点にする
"<Shift-B1-Motion>"	起点からドラッグした範囲を起点と同じ状態にする
"<Up>" "<Down>"	活性化項目を1行上下、起点にする
"<Shift-Up>" "<Shift-Down>"	活性化項目から選択範囲を拡大する
"<Control-Home>"	最初の行を選択・活性化
"<Control-Shift-Home>"	最初の行までの範囲を選択・活性化
"<Control-End>"	最後の行を選択・活性化
"<Control-Shift-End>"	最後の行までの範囲を選択・活性化
"<Left>" "<Right>"	1単位の左右スクロール
"<space>" "<Select>"	活性化した行を選択
"<Control-slash>"	全体を選択
"<Control-backslash>"	選択を解除
"<Escape>"	直前の選択動作をキャンセル

5.11.3 選択項目のとりだし

選択が行われた後に選択された項目を取り出すには

```
a = リストボックスシンボル[Selection];
```

とすれば選択項目の番号がリストになって返されます。選択項目が単数の場合でも答えは必ずリストです。番号は最初の項目が 1 です。また、選択された文字列を取り出したいときには

```
a = リストボックスシンボル[GetText[Selection]];
```

とすれば、選択された文字列がやはりリストで返ります。また、選択の有無にかかわらず、

```
a = リストボックスシンボル[GetText[番号]];
```

とすれば、その番号の文字列が返ります。

5.11.4 ListBox の属性

ListBox の属性を表 22 に示します。ここで行の指定は表 19 によって行います。

表 22: ListBox の属性。

属性	入出力値(単位)	デフォルト	機能
BorderWidth(BD)	ピクセル	2	境界(立体表示される)の幅
Delete	行1 または {行1, 行2}		行1 または行1 から行2 までを削除
ExportSelection	True, False	True	選択範囲を X に伝達
Foreground(FG)	色	"black"	文字色
Font	フォント		文字フォント
Height	行数		表示の行数
Relief	"flat", "groove", "raised", "ridge", "sunken"	"sunken"	枠の立体形状
See	行		指定した行を表示範囲に移す
SelectBackground	色	"green"	選択範囲の背景色
SelectBorderWidth	ピクセル	1	選択範囲の境界幅
SelectForeground	色	"black"	選択範囲の文字色
SelectMode	"browse" "single" "extended" "multiple"	"browse"	選択モード
Select$Clear	行1 または {行1, 行2}		行1 または行1 から行2 の選択を解除
Select$Set	行1 または {行1, 行2}		行1 または行1 から行2 を選択
SetGrid	True, False	False	True のときサイズの変更に制限
Width	文字数		表示の幅
XScrollCommand	コマンド		横方向のスクロールバーの割り当て
XView	文字位置		横方向の表示位置
YScrollCommand	コマンド		縦方向のスクロールバーの割り当て
YView	行数		縦方向の表示位置

5.12 Menu と MenuButton

Menu はそのなかにいくつかの menu entry とよばれる部品を貼る枠です。貼れるものは Button, CheckButton, RadioButton, Separator, と Cascade です。このうち、3種類のボタンはこれまで登場した3種類の単独のボタンとほぼ同様の動作をします。Separator は部品と部品の間の区切りのための部品です。Cascade は多段式の Menu の連鎖をつくるための部品です。

MenuButton はそこに Menu を割り当てることにより、MenuButtonを押したときにだけ現われるプルダウン・メニューを実現します。MenuButton には Button と同じ属性（表 8 参照）が備わっています。

次の例は MenuButton mb に Menu mn を割り当て、しかる後に mn に各種の部品を追加しています。また、mn に Cascade される Menu mn1 は Cascade が mn に追加される前に定義しておきます。つまり、mn, mn1, Cascade の順に定義しないとうまく動作しません。これは自分が親または子として参照する部品は自分より先に定義されていなければならないという事情によります。

```
w = Window[];
mb = MenuButton[w, Text -> "MENU"];
mn = Menu[mb, TearOff -> True];
mn1 = Menu[mn, TearOff -> True,
  Add -> {
    RadioButton[Text -> "Rare",
      Value -> 1, Variable :> cook],
    RadioButton[Text -> "Medium",
      Value -> 2, Variable :> cook],
    RadioButton[Text -> "Well-done",
      Value -> 3, Variable :> cook]}];
mn[Add] = {
  Button[Text -> "Salad"],
  Separator[],
  Cascade[Text -> "Steak", Menu -> mn1],
  Button[Text -> "Fish"],
  Separator[],
  CheckButton[Text -> "Dessert", Variable :> dessert]};
cook = 1;
dessert = 1;
```

この例は図 20 のような結果になります。

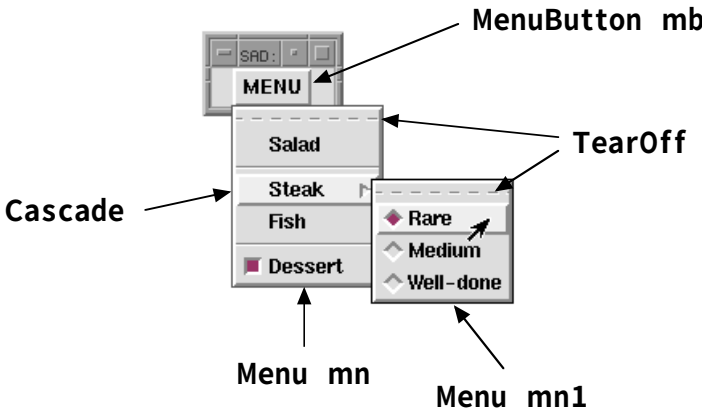

図 20: Menu と MenuButton の用例。

5.12.1 Menu の属性

Menu の属性を表 23 に示します。

表 23: Menu の属性。

属性	入出力値(単位)	デフォルト	機能
BorderWidth(BD)	ピクセル	2	境界(立体表示される)の幅
Delete	部品番号1 {部品番号1, 部品番号2}		部品番号1、或いは部品番号1 から部品番号2 までの部品を削除する
DisabledForeground	色		disabled 状態の部品の文字色
EntryConfigure	{部品番号, 属性 -> 値, ...}		部品番号の部品の属性を変更する
Font	フォント		文字フォント
Foreground(FG)	色	"black"	文字色
Invoke	部品番号		部品番号の部品に割り当てられたコマンドを実行する
PostCommand	コマンド		Menu が現われる直前に実行する式
SelectColor	色	"red"	Check, Radio のセレクタの色
TearOff	True, False	False	True の時 TearOff 部品を付加し、メニューをボタンから剥離・移動可能にする
YPosition	部品番号		部品番号の部品のスクリーン上の位置を返す

表 23 で、「部品番号」とは Menu に貼られた各部品を識別する番号で 1 **から**始まります。また、"active"（今、マウスカーソルの置かれているもの)、"last"（最後の部品)、"@y 座標"（y 座標にある部品）という指定も可能です。

5.12.2 TearOff

表 23 の中で TearOff という属性を True（デフォルトは False）にすると図 20 にある様に各メニューの最上部に破線が付加されます。この破線も一種の部品で、これを選ぶとその Menu が MenuButton から独立したひとつの Window になります。これはあるメニューを持続的に使用する場合大変便利です。TearOff が True の場合は TearOff が部品番号 1 となり、他の部品は番号 2 から始まります。

5.12.3 Menu に貼られる部品の属性

Menu に貼られる部品は対応する単独部品の属性の一部を持っています。次にそれらを示します。

表 24: Menu に貼られる部品の属性。

属性	入出力値(単位)	デフォルト	機能
ActiveBackground	色		マウスカーソルが上にあるときの背景色
ActiveForeground	色		マウスカーソルが上にあるときの文字色
Accelerator			
Bitmap			ボタン表面にbitmapを表示
Command	式		選択時に実行する式
Font	フォント		文字フォント
Foreground(FG)	色	"black"	文字色
Justify	"center" "left" "right"	"center"	テキストの整列
OffValue	数値		Checkbutton がOFFの時の値
OnValue	数値		Checkbutton をONにする値
SelectColor	色	"red"	チェックマークの色
State	"normal" "active" "disabled"	"normal"	エントリーの状態
Text	文字列		ボタン表面の表示文字列
Underline	数値		下線を付ける文字の位置、0 が最初の文字
Value	数値		RadioButton をONにする値
Variable	シンボル		結合変数

5.12.4 Underline

Menu に貼られる部品に属性 Underline -> 文字位置を指定すると、その位置（0 を最初の文字とする）の文字に下線が付加されます。そしてメニューが表示されている状態でその文字のキーを打つと、その部品のコマンドが実行されます。また、<space> 及び <Return> キーを打つと、現在選ばれている部品のコマンドが実行されます。さらに <escape> キーはメニューの表示を消します。

5.13 OptionMenu

OptionMenu はいくつかのメニュー項目の中から一つを選択し、選択されたものの名称をボタン表面に表示するという機能を一度に実現します。次の例

```
w = Window[];
om = OptionMenu[w, TextVariable -> africananimal,
  Items -> {"lion", "giraffe", "zebra",
    "hippopotamus", "rhinoceros"}];
africananimal = "giraffe";
```

は Items で指定した文字列のリスト（7 章を参照）の中の一項目が TextVariable で指定した結合変数、africananimal に割り当てられます。そしてその変数の値がボタン表面に図 21(A) のように表示されます。また、マウスで選択する際にはメニューが現在の設定値を中心に図 21(B)のように展開されます。

OptionMenu には MenuButton と同様に、Menu を貼り付けることもできます。また、OptionMenu の属性は Items 以外は Button と同じです（表 8 参照）。

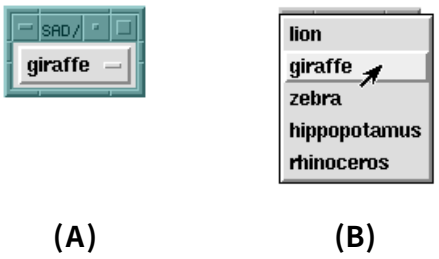

図 21: OptionMenu の用例。

5.14 その他の部品

SAD には上記の他に Canvas や TextEditor などの、より複雑な部品が存在します。ここではまず SAD 言語の構成要素と文法について先に解説し、その後に残りの部品についての説明をつづけたいと思います。

6 SAD の構成要素

6.1 原子

SAD の最も基本的な構成要素(原子 Atom)には次のものがあります。

- 実数。浮動小数点8バイト長。
- 文字列。1文字1バイトで任意長。
- シンボル。英字、$、数字からなる任意長の名前。数字で開始してはいけない。大小文字は区別される。
- パターン。他の要素との照合をはかるための要素。

これらの原子は次のように入力されます。

6.1.1 実数の入力表記

通常の表記法のほかに、FORTRANやCのようにE あるいは e で 10 の冪乗を表わすことができます。なお、複素数は原子ではなく、2 + 3 * I などのように虚数単位をあらわすシンボル I を用いた式で表わします。また、現在の仕様では実数と整数の区別はなく、演算の結果はすべて実数になるので丸め誤差に注意しなければなりません。

6.1.2 文字列の入力方法

文字列は""で囲みます。\ を使うと特殊な入力が可能です。

表 25: \による特殊な入力。

\e	escape
\n	改行
\r	return
\t	tab
\\	backslash
nnn	ASCIIコードが 8 進数で *nnn* の文字
\その他の文字	その他の文字
行末の\	次の行に続く、改行文字は含まれない

6.2 複合要素、式

SAD には要素の複合体である複合要素、あるいは式と呼ばれるものがあります。式は、一般には

```
要素0[要素1, 要素2, ...]
```

のような格好をしています。要素の数はいくつでもかまいません。各要素は原子でもよいし、別の式でもかまいません。例えば a[b, ...][c, ...]...[d, ...] の様な形も式のひとつです。また、要素0をこの式の頭部 head と呼びます。

6.2.1 演算子による式の表記

上のような式のままでも既に SAD の言語構造はすべて表現可能で、実際内部の処理はそうなっています。これが SAD もそれに属するところの「関数構文言語」の特徴です。つまり、この言語にはこれ以上特別な構文は存在しないという利点があります。例えば *Mathematica* や emacs-lisp など幅広く利用されている言語の中にも関数構文言語は存在します。

しかし、上の格好のままでは単純な数式を書くのにも伝統的な表記法からはかけ離れてしまい、読み書きが難しくなります。そこで SAD では上の要素0 が特別なシンボルの場合に伝統的な演算子による表記が可能になっています。例えば

```
a + b * c  は  Plus[a, Times[b, c]]
```

と同値です。ここで問題になるのは演算子の優先順位です。表26、27は SAD で使われる演算子とその優先順位を示しています。この表で上にくるものほど優先的に評価されます。横線で区切られた範囲のものは同一の優先度をもち、その場合は式の左側から演算されます。

表 26: 演算子とその優先順位。

演算子	全表現	グループ化
()		
#シンボル	Slot[シンボル]	
%正整数	Out[正整数]	
_ を含むシンボル		
式1 :: 式2	MessageName[式1, 式2]	
式1 ? 式2	PatternTest[式1, 式2]	
{式1, 式2, ... }	List[式1, 式2, ...]	
式1[式2, ...]	式1[式2, ...]	(式1[式2])[式3]
式1[[式2, ...]]	Part[式1, 式2, ...]	(式1[[式2]])[[式3]]
式++	Increment[式]	
式--	Decrement[式]	
++式	Increment[Null, 式]	
--式	Decrement[Null, 式]	
式1 @ 式2	式1[式2]	式1[式2[式3]]
式1 /@ 式2	Map[式1, 式2]	式1 /@ (式2 /@ 式3)
式1 //@ 式2	MapAll[式1, 式2]	式1 //@ (式2 //@ 式3)
式1 @@ 式2	Apply[式1, 式2]	式1 @@ (式2 @@ 式3)
式1 // 式2 // 式3	StringJoin[式1, 式2, 式3]	
式1 . 式2 . 式3	Dot[式1, 式2, 式3]	
式1 ^ 式2	Power[式1, 式2]	式1 ^ (式2 ^ 式3)
-式	Times[-1, 式]	
+式	式	
式1 / 式2	式1 * 式2 ^-1	
式1 * 式2 * 式3	Times[式1, 式2, 式3]	
(式1 式2 式3)	Times[式1, 式2, 式3]	
式1 + 式2 + 式3	Plus[式1, 式2, 式3]	
式1 - 式2	式1 + (-1 * 式2)	
式1 == 式2	Equal[式1, 式2]	
式1 <> 式2	Unequal[式1, 式2]	
式1 >< 式2	Unequal[式1, 式2]	
式1 > 式2	Greater[式1, 式2]	
式1 >= 式2	GreaterEqual[式1, 式2]	
式1 => 式2	GreaterEqual[式1, 式2]	
式1 < 式2	Less[式1, 式2]	
式1 <= 式2	LessEqual[式1, 式2]	
式1 =< 式2	LessEqual[式1, 式2]	
式1 === 式2	SameQ[式1, 式2]	
式1 <=> 式2	UnsameQ[式1, 式2]	

表 27: 演算子とその優先順位(続き)。

演算子	全表現	グループ化
~ 式	Not[式]	
式1 && 式2 && 式3	And[式1, 式2, 式3]	
式1 || 式2 || 式3	Or[式1, 式2, 式3]	
式 ..	Repeated[式]	
式 ...	RepeatedNull[式]	
式1 | 式2 | 式3	Alternatives[式1, 式2, 式3]	
シンボル:式	Pattern[シンボル, 式]	
式1 -> 式2	Rule[式1, 式2]	式1 -> (式2 -> 式3)
式1 :> 式2	RuleDelayed[式1, 式2]	式1 :> (式2 :> 式3)
式1 /. 式2	ReplaceAll[式1, 式2]	
式1 //. 式2	ReplaceRepeated[式1, 式2]	
式1 += 式2	AddTo[式1, 式2]	
式1 -= 式2	SubtractFrom[式1, 式2]	
式1 *= 式2	TimesBy[式1, 式2]	
式1 /= 式2	DivideBy[式1, 式2]	
式 &	Function[式]	
式1 = 式2	Set[式1, 式2]	式1 = (式2 = 式3)
式1 := 式2	SetDelayed[式1, 式2]	
式1 ^= 式2	UpSet[式1, 式2]	
式1 ^:= 式2	UpSetDelayed[式1, 式2]	
式 =.	UnSet[式]	
式1 ; 式2 ; 式3	CompoundExpression[式1, 式2, 式3]	
式1 ; 式2 ;	CompoundExpression[式1, 式2, Null]	

6.2.2 特別な演算子の用法

シンボルの間、或いは式の間の空白は、前の式が完結しないうちは、Times (*)とみなされます。

連続した関係演算子で結ばれた式、例えば a <= b < c などは Inequality[a, LessEqual, b, Less, c] と変換されます。これにより中央の式が重複して評価されるのが防がれます。ここで、関係演算子とは、Equal(==), Unequal(<>), Less(<), LessEqual(<=), Greater(>), GreaterEqual(>=), SameQ(===), 及び UnsameQ(<=>) です。

6.2.3 頭部 **head** の取り出し

ある式の頭部は Head という関数によって取り出します。f[x, y, ...] の頭部は f です。また、原子についても Head は次のような値を返します。

```
   Head[f[x, y]]
Out[1]:= f
   Head[1]
Out[2]:= Real
   Head["abc"]
Out[3]:= String
   Head[abc]
Out[4]:= Symbol
   Head[abc_]
Out[5]:= Pattern
```

ちなみに複素数の頭部は Complex です。

ある式の持つ意味はほとんどの場合その頭部によって決まります。式はある場合には関数の引用であり、評価されると何かの作用を起こしつつ、形の違う式を結果として返します。また、別の場合にはある種のデータの集合体とみなされ、そのような場合には評価されても形は変わりません。

6.2.4 式の部分の取り出し

- ある式の部分は Part という関数で取り出すことができます。ある式fの $n1$ 番目の要素は f[[$n1$]]で取り出します。取り出した式の更に $n2$ 番目の要素は f[[$n1$, $n2$]] で取り出します。この場合 Part[f, $n1$, $n2$, ...] と f[[$n1$, $n2$, ...]] は同値です。また、f[[$n1$]][[$n2$]] と f[[$n1$, $n2$]] は同値です。
- インデクス $n1$, $n2$, ... は 0 が頭部、1 が最初の要素、となっており、負の場合は最後の要素から逆に数えた順になります。

関数 Part にはもう少し特殊な使い方がありますが、それはリストの項 7 で説明します。

6.3 シンボル

シンボルは SAD のなかではある原子、式、或いは関数を割り当てるために用います。シンボルは何も割り当てがない場合にはシンボルそれ自身が割り当てられていると考えられます。したがって、割り当てなしでいきなり使うこともできます。

一度 On[General::newsym] としておけば、以後、新しいシンボルの生成を検出することができます（10.8.3 参照）。

いくつかのシンボルにはシステムが前もって定義を与えています。例えば、組込みの関数がそれです。このようなシンボルはその名前を構成する各語の先頭が大文字、他が小文字で表記されています。

6.3.1 Set

シンボルに値を割り当てるには関数 Set（演算子 =）を用います。

- シンボル1 = 式1 は式1を評価した値をシンボル1 に割り当てます。
- 式（シンボル1 = 式1）自身は式1 の値を返します。
- シンボル1 = シンボル2 = ... = 式1 は式1 を 1度だけ評価し、その結果をシンボル1，シンボル2，... に割り当てます。
- {シンボル1，シンボル2，... } = {式1，式2，... } はシンボル1に式1の値、シンボル2 に式2 の値、... を割り当てます。この場合シンボルの数と式の数は一致しなければなりません。

このほか、後述のように、Set はリストの要素への値の割り当てや関数の定義にも用いられます。

6.3.2 SetDelayed

- シンボル1 := 式1 は式1 を評価せず、そのままの形で、シンボル1に割り当てます。
- 右辺を評価しないという点以外は SetDelayed と Set は同一です。
- 右辺を部分的に評価するには後述の With を用います。

6.3.3 AddTo, SubtractFrom, TimesBy, DivideBy, AppendTo, PrependTo

SAD にはシンボルの現在の値にある演算をして、またその結果を同じシンボルに再び割り当てるという関数(演算子)があります。表 28 にそれらを列挙します。

表 28: シンボルの再割り当てを行う関数。

関数	演算子表記	等価な式
AddTo	シンボル += 式	シンボル = シンボル + 式
SubtractFrom	シンボル -= 式	シンボル = シンボル - 式
TimesBy	シンボル *= 式	シンボル = シンボル * 式
DivideBy	シンボル /= 式	シンボル = シンボル / 式
AppendTo		シンボル = Append[シンボル, 式]
PrependTo		シンボル = Prepend[シンボル, 式]

6.3.4 特別な定数シンボル

表 29 のシンボル達はシステムがあらかじめ定数値を割り当てているものです。これらのシンボルには属性 Constant が与えられており (?? 参照)、入力時に直ちに評価されます。

表 29: 特別な定数シンボル。

シンボル	値
True	1
False	0
Pi	2 * ArcSin[1]
I	Complex[0, 1]
GoldenRatio	(1 + Sqrt[2]) / 5
Degree	Pi / 180
SpeedOfLight	299792458
Infinity	(INF)
INF	(INF)
NaN	(NaN)

6.3.5 シンボルへの値の割り当ての解除

あるシンボルに対する値の割り当てを解除したい場合は 10.7 にある Clear や Unset (=.) を用います。

6.4 式の評価

SAD は式を次の様に評価します。

- その要素がすべて定数と演算子から成り立っている部分式は入力した時点で評価されます。
- それ以外の式はその式の入力が完結した時点で評価されます。
- 関数の引き数はその関数の引き数の属性 (8.5 参照) にしたがって評価されます。

7　リスト

リストは式或いはデータの集合を表わすものとして、SAD ではきわめて頻繁に用いられます。リストは全体を {} で囲み、要素間を , で区切って表わします。もちろんリストの要素は何であっても構いませんのでリストの中に何重にもリストを重ねることができます。また、リストとはその頭部がシンボル List であるような式にすぎません。つまり、{1, 2, 3} と List[1, 2, 3]は同値です。従って、リストに対して作用できる関数は、多くの場合、頭部が List でない一般の式に対しても作用できます。

7.1　リストの演算

算術演算はリストの各要素に対して並列的に実行され、結果はまたリストです。この場合両被演算項がリストの場合は、両方の長さが一致していなければなりません。次の例の最後の行はこの条件を満たさなかったので、演算が行なわれていません。

```
   {1, 2, 3} * 2
Out[1]:= {2,4,6}
   {1, 2, 3} + {4, 5, 6}
Out[2]:= {5,7,9}
   {1, 2, 3} + {{4, 5}, {6, 7}, {8, 9}}
Out[3]:= {{5,6},{8,9},{11,12}}
   {1, 2, 3} + {4, 5, 6, 7}
Out[4]:= ({1,2,3}+{4,5,6,7})
```

また、多くの数学関数はリストに作用すると、その各要素に並列に作用し、結果をリストにして返します。

以上のような特性は多数のデータを取り扱うときにはきわめて重要で、できるだけリスト全体に対して演算や関数の作用を行うようプログラムすることにより、実行速度の向上させることができます。

> このようなプログラミングの方向は、一言でいえば「できるだけ添え字を使わない」ということで表わされます。この方法は、ある程度までは、プログラムを簡略にし読み易くするという効果も伴います。しかし、ある点を越えると読み易さと実行効率が相反するようになり、そのどこでバランスをとるかは簡単には答えられない問題です。

リストはまた、ヴェクトル、行列、テンソルを表現するものともみなされます。

7.2 リストの生成

リストは各要素を直接指定して生成しますが、ある種の規則からも作り出すこともできます。

7.2.1 Table、反復指定子

Table は与えられた数の要素をもつリストを生成します。

- Table[式1，{数1}] は式1 を数1 回評価した値からなるリスト、
- Table[式1，{i，数1}] はシンボルi を1から数1 まで1ずつ増加させながら、式1 を評価した値からなるリスト、
- Table[式1，{i，数1，数2}] はシンボルi を数1 から数2 まで1ずつ増加させながら、式1 を評価した値からなるリスト、
- Table[式1，{i，数1，数2，数3}] はシンボルi を数1 から数2 まで数3 ずつ足しながら、式1 を評価した値からなるリスト、

を返します。この場合、シンボルi はこの Table の中だけで意味をもちます。もちろん、上で数1などと書いたところは、ある実数を与える式、という意味で、定数である必要はありません。ただし、これらの値は反復が始まる時点で評価され、反復の途中で変更しても反復の回数には影響しません。

上で使われた {i，数1，数2，数3} の様な指定の仕方は反復指定子 iterator と呼ばれ、他の Do，Sum，Product などでも使われます。また、Table[式1，反復指定子1，... ，反復指定子n] は
Table[Table[式1，反復指定子n]，反復指定子1，... ，反復指定子$n-1$] と同値でn次元のテンソルを表わします。この場合後に書いた反復指定子ほど速く変化します。

7.2.2 Range

- Range[数1] は 1 から数1 までの増分1 のリストを返します。
- Range[数1，数2] は数1 から数2 までの増分1 のリストを返します。
- Range[数1，数2，数3] は数1 から数2 までの増分数3 のリストを返します。

7.2.3 IdentityMatrix

- IdentityMatrix[n] は n行 n列の単位行列を返します。

7.2.4 DiagonalMatrix

- DiagonalMatrix[リスト1] はリスト1 の各要素を対角成分とする正方行列を返します。

7.3 リストの操作

以下の諸関数はリストの頭部が List でなくても成り立ちます。多くの場合、全体の頭部と部分リストの頭部の一致が必要です。例えば、Dimensions[a[a[1, 2, 3], a[4, 5, 6]]] は {2, 3} ですが、
Dimensions[a[{1, 2, 3}, a[4, 5, 6]]] は {2} です。

7.3.1 Length

- Length[リスト1] はリスト1 の要素の個数を返します。
- Length[原子] は 0 です。

7.3.2 Dimensions

- Dimensions[リスト1] はリスト1 をテンソルとみなして、その各次元の要素数をリストにしてかえします。Dimensions[{{1, 2, 3}, {4, 5, 6}}] ⇒ {2, 3}。したがって、各要素の要素数が揃っていないようなリストは次元が浅くなります。例えば、Dimensions[{{1, 2, 3}, {4, 5}}] ⇒ {2}。
- Dimensions[原子] は {} です。

7.3.3 Depth

- Depth[リスト1] はリスト1 の最大階数+1、つまり最大何重の部分リスト(子リスト)から成り立っているか +1、を返します。Depth[{1, 2, 3}] ⇒ 2。Depth[{1, {{2, 3}, 4}, 5}] ⇒ 4。
- Depth[原子] は1です。
- 頭部の一致は無視されます。Depth[1, a[2, 3]] ⇒ 3。

7.3.4 Level, 階数指定子

- Level[リスト1, 階数1] はリスト1 の 1階から階数1 までの間に含まれるすべての要素をリストにして返します。
- Level[{1, 2, {{3, 4}, 5}, 2] ⇒ {1, 2, {3, 4}, 5, {{3, 4}, 5}}。
- 階数1 が負の場合はリスト1 の 1階から各々の枝の(最上階 + 階数1 + 1)階までのすべての要素のリストを返します。「負の階数」の意味は以下の場合も同様です。
- Level[{1, 2, {{3, 4}, 5}, -1]
  ⇒ {1, 2, 3, 4, {3, 4}, 5, {{3, 4}, 5}}。
- Level[リスト1, {階数1}] はリスト1の(階数1)階の要素のリストを返します。

- Level[リスト1，{階数1，階数2}] はリスト1の(階数1)階から(階数2)階までの要素のリストを返します。
- 頭部の一致は無視されます。
- 階数0 はリスト全体を指します。

上の Level の第 2 引き数のようにリストあるいは式の階数の範囲を指定する方法は階数指定子とよばれ、Map，Apply，Position，Count，Cases，DeleteCases，MapIndexed，Scan に共通です。

7.3.5 Take、要素指定子a

- Take[リスト1，$n1$] はリスト1 の 1 番目から $n1$ 番目の要素をリストにして返します。
- $n1$ が負の場合は Take[リスト1，$n1$] はリスト1 の(全長+$n1+1$) 番目の要素から最後の要素をリストにして返します。-1 が最後の要素です。
- Take[リスト1，{$n1$，$n2$}] はリスト1 の $n1$ から $n2$ までの要素をリストにして返します。
- Take[リスト1，{$n1$}] はリスト1 の $n1$ 番目の要素だけをリストにして返します。
- 要素指定子を 0 にしても、頭部を意味することはありません。

いずれの場合も結果の頭部はリスト1 の頭部と同じになります。上のような要素および要素の範囲の指定（要素指定子a ）は Drop などに共通です。

7.3.6 Drop

- Drop[リスト1，要素指定子a 1] はリスト1 から要素指定子a 1 で指定した要素を取り除いたリストを返します。この場合頭部はリスト1 の頭部と同じになります。
- 要素指定子a は Take 7.3.5 を参照してください。

7.3.7 First

- First[リスト1] はリスト1の最初の要素を返します。

7.3.8 Last

- Last[リスト1] はリスト1の最後の要素を返します。

7.3.9 Rest

- Rest[リスト1] はリスト1の最初の要素を取り除いた残りをリストにして返します。この場合頭部はリスト1の頭部と同じになります。

7.3.10 Reverse

- Reverse[リスト1] はリスト1の要素を全て逆に並び変えたリストを返します。

7.3.11 Append

- Append[リスト1，要素1] はリスト1の最後尾に要素1を追加したリストを返します。

7.3.12 Prepend

- Prepend[リスト1，要素1] はリスト1の先頭に要素1を追加したリストを返します。

7.3.13 Join

- Join[リスト1，リスト2] はリスト1とリスト2の両方の要素をこの順に結合したリストを返します。この場合両方の頭部は揃っていなければならず、結果の頭部も同じになります。

7.3.14 Flatten

- Flatten[リスト1] はリスト1の全ての階数の全ての要素を第1階に平坦に並べたリストをかえします。Flatten[{1，2，{{3，4}，5}}] ⇒ {1，2，3，4，5}。この場合頭部の一致が必要です。Flatten[{1，2，{a[3，4]，5}}] ⇒ {1，2，a[3，4]，5}。
- Flatten[リスト1，階数1] は1階から階数1までを平坦にします。
- Flatten[リスト1，階数1，頭部1] は、1階から階数1までで且つ頭部が頭部1であるもののみを平坦にします。

7.3.15 Thread

- Thread[{リスト1，リスト2，...}] はリスト1，リスト2，... のそれぞれの第 n要素からなるリスト達のリストを返します。
- Thread[{{1，2，3}，{4，5，6}}] ⇒ {{1，4}，{2，5}，{3，6}}。
- 頭部の一致が必要です。
- リスト1，リスト2，... の中に最初のリストと長さが違うもの、頭部が違うもの、リストでないものが含まれる場合はその要素は結果のどのリストにも共通して含まれることになります。Thread[{{1，2，3}，7，{4，5，6}}] ⇒ {{1，7，4}，{2，7，5}，{3，7，6}}。
- 長方形のリストに対しては Thread と Transpose は同一の結果を返します。

7.3.16 Partition

- Partition[リスト1, n] はリスト1 の要素を頭から n 個ずつ分配したリスト達のリストを返します。
- リスト1 の長さが n の倍数でない時には余りの要素は捨てられます。
- Partition[{1, 2, 3, 4, 5}, 2] ⇒ {{1, 2}, {3, 4}}。
- Partition[リスト1, n, m] はリスト1 の要素を頭から m 個おきに、重複を許して n 個ずつ分配したリスト達のリストを返します。
- Partition[{1, 2, 3, 4, 5, 6, 7}, 3, 2] ⇒
  {{1, 2, 3}, {3, 4, 5}, {5, 6, 7}}。
- Partition[リスト1, n] は Partition[リスト1, n, n] と同値です。

7.3.17 Sort

- Sort[リスト1] はリスト1 の各要素を標準的な順序（表 30 参照）にしたがって並べ替えたリストを返します。
- Sort[{4, 2, 1, 3}] ⇒ {1, 2, 3, 4}。
- Sort[リスト1, テスト1] は並べ替えを2変数関数テスト1 に従って行います。テスト1 は True または False を返します。そのような関数の定義の仕方は 8 関数の定義を参照してください。
- Sort[{4, 2, 1, 3}, Greater] ⇒ {4, 3, 2, 1}。

表 30: 標準的な順序。型の違うものはこの表の上から順に並ぶ。

式の型	型のなかでの順序
実数	小さいものから大きいものへ。
文字列	最初の異なる文字の順に並べる。各文字はまず「無」が先頭で、次にアルファベットでないものが ASCII コードの順に並び、最後にアルファベットが aAbBcC...zZ のように並ぶ。
シンボル 及び パターン原子	名前を表わす文字列の順序
リスト	要素数の少ないものから。要素数が同じ場合頭部の順。頭部が同じ場合は第一要素の順、以下同様。

7.3.18 Union

- Union[リスト1，リスト2，...] はリスト1，リスト2，... のいずれかに含まれる要素達を重複せず集め、標準的な順序（表 30 参照）に従って並べ変えたリストを返します。
- Union[{3，-1，"c"}，{"c"，3，3}，{1，"a"}] ⇒ {-1，1，3，"a"，"c"}。
- Union[リスト1] はリスト1 の中の要素達を重複せず、標準的な順序に従って並べ変えたリストを返します。

7.3.19 Intersection

- Intersection[リスト1，リスト2，...] はリスト1，リスト2，... のいずれにも含まれる要素達を、標準的な順序（表 30 参照）に従って並べ変えたリストを返します。
- Intersection[{3，1，-2}，{1，"a"}，{1，1}] ⇒ {1}。

7.3.20 Complement

- Complement[リスト0，リスト1，リスト2，...] はリスト0 の要素で、リスト1，リスト2，... のいずれの要素でもないもの達を集め、標準的な順序（表 30 参照）に従って並べ変えたリストを返します。
- Complement[{3，2，1，3，0}，{2，1}，{4，1}] ⇒ {0，3}。

7.4 リストの要素に作用する関数

7.4.1 Part, [[]]

リストの個々の要素の取り出しは、リストもひとつの式ですから、式の要素の取り出しを行う関数 Part（演算子 [[]])で出来ます。

- リストfの $n1$ 番目の要素は f[[$n1$]]で取り出します。取り出したリストまたは式の更に$n2$ 番目の要素は f[[$n1$，$n2$]] で取り出します。この場合 Part[f，$n1$，$n2$，...] と f[[$n1$，$n2$，...]] は同値です。また、f[[$n1$]][[$n2$]] と f[[$n1$，$n2$]] は同値です。
- インデクス $n1$，$n2$，... は0が頭部、1が最初の要素、となっており、負の場合は最後の要素から逆に数えた順になります。
- Partでは頭部の一致は必要ありません。
- f[[{$n11$，$n12$，..}，$n2$，]]は第 1 インデクスが $n11$，$n12$，...で、第 2 インデクスが$n2$ 以下で指定される要素になるような要素達のリストを返します。{{1，2}，{3，4}，{5，6}，{7，8}}[[{2，4}，2]] ⇒ {4，8}。

- あるインデクスが省略または Null にされると、そのインデクスが 1 から要素数までの全ての値に対して、ほかのインデクスが指定の値になるようなリストを返します。{{1, 2}, {3, 4}, {5, 6}, {7, 8}}[[, 2]] ⇒ {2, 4, 6, 8}。

7.4.2 Insert, 要素指定子b

- Insert[リスト1, 新要素1, $n1$] はリスト1 の $n1$ 番目の位置に新要素1 を挿入したリストを返します。
- Insert[リスト1, 新要素1, {$n1$, $n2$, ... }] はリスト1の {$n1$, $n2$, ...} の位置（リスト1[[$n1$, $n2$, ...]]） に新要素1 を挿入したリストを返します。
- Insert[リスト1, 新要素1, {{$n1$, ...}, {$n2$, ... }}] はリスト1 の幾つかの位置達に新要素1 を挿入したリストを返します。

上のような要素の位置の指定法(要素指定子b)は Take の場合とは異なっています。これは、Delete, Extract, FlattenAt, MapAt等で用いられます。

7.4.3 Delete

- Delete[リスト1, 要素指定子b1] はリスト1の要素指定子b1（Insert 7.4.2 を参照）で指定される要素を取り除いたリストを返します。

7.4.4 ReplacePart

- ReplacePart[リスト1, 新要素1, 要素指定子b1] はリスト1 の要素指定子b1（Insert 7.4.2 を参照）で指定される位置達に新要素1 を代入したリストを返します。すなわち:
- ReplacePart[リスト1, 新要素1, $n1$] はリスト1 の $n1$ の位置に新要素1 を代入したリストを返します。
- ReplacePart[リスト1, 新要素1, {$n1$, $n2$, ... }] はリスト1 の {$n1$, $n2$, ...}の位置(リスト1[[$n1$, $n2$, ...]]) に新要素1 を代入したリストを返します。
- ReplacePart[リスト1, 新要素1, {{$n1$, ...}, {$n2$, ... }}] はリスト1の幾つかの位置達に新要素1 を代入したリストを返します。

例えば ReplacePart[a, new, ...]としたときに、ReplacePart という語感からは、元のリスト a の一部も変更されるかのような感じを受けますが、そうではありません。ReplacePart は一部を置き換えた結果のリストを返すだけなので、もし元のリストも変更したければその結果をシンボル a に再度 Set するか、以下に述べる様に Set の左辺に Part を用いる必要があります。

7.4.5 Extract

- Extract[リスト1，要素指定子b1] はリスト1 の要素指定子b1（Insert 7.4.2 を参照）で指定される要素達を返します。
- Extract[リスト1，要素指定子b1，関数1] はリスト1（式でもよい）の要素指定子b1（Insert 7.4.2 を参照）で指定される要素達に**それを評価せずに関数1 を適用した結果達**を返します。
- Extract[Hold[{Sin[1], Cos[1]}], {1, 2}, Hold] ⇒ Hold[Cos[1]]
  (Hold 10.6.1 を参照)。

7.4.6 FlattenAt

- FlattenAt[リスト1，要素指定子b1] はリスト1 の要素指定子b1（Insert 7.4.2 を参照）で指定される要素のすぐ下に属する要素達を表に取り出して同列にならべたリストを返します。
- FlattenAt[{1, {2, {3, 4}}, 5}, {2}] ⇒ {1, 2, {3, 4}, 5}。
- 上の例のように、表に出るのはその要素の直下のもの、2 と {3, 4} だけで、更に深いもの、3 や 4は表には出ません。
- 頭部の一致は必要ありません。
- FlattenAt[{1, a[2, {3, 4}], b[5]}, {{2}, {3}}] ⇒ {1, 2, {3, 4}, 5}。

7.5 リストの要素の値の設定

いまあるシンボル a にリスト(あるいは式）が割り当てられているとき、演算子 Part を Set、SetDelayed の式の左辺に用いて

```
a[[n1, n2, ...]] = 式1
```

のようにすると、シンボル a に割り当てられたリスト(式）の n1，n2，... 番目の要素が右辺の値に置き換わります。これはあたかも他の言語における配列への代入に似ています。

この方法と上述の ReplacePart とを混同しないように注意してください。

8　関数の定義

SAD では必要な関数を随時定義して使うことができます。最も単純な例として、今、

```
   f[x_] := x^2;
```

の様に定義すれば、

```
   f[4]              :fはその引き数を2乗する。
Out[1]:= 16
   f[1 + a]          :引き数は式でもよい。
Out[2]:= ((1+a)^2)
   f[3x + x^2]
Out[3]:= (((3 x)+(x^2))^2)
```

となります。ここで、上の定義式の左辺の x_ につけるアンダースコア(_)はきわめて重要な意味を持っています。シンボル1_ という記号は、SAD では、「シンボル1という名前を持ったパターン原子」とみなされます。もしアンダースコアを書き落とすと、

```
   f[x] := x^2;        :左辺の引き数に_がない。

   f[4]
Out[1]:= f[4]         :fはもはや2乗の関数ではない。
   f[x]
Out[2]:= (x^2)        :それでも引き数が x ならば右辺の値になる。
```

のように、全然違った結果になります。実は後者のような書き方もエラーではなく、有用な場合があります。

8.1　引き数の置換

x_ というパターンは、関数の評価に際して、

(1) 任意の形の引き数に照合する。

(2) 右辺の定義式のなかに現われる同じ名前のシンボル(x)を、全て引き数の値に置き換え、その後に右辺を評価する。

という動作をします。従って、右辺の x が右辺の評価の際に何であるかはその関数が呼ばれた時に決まります。上の第2項の右辺のシンボルの置き換えでは、右辺にあからさまに現われているシンボルだけを置き換えます。つまり、右辺の式の評価の結果、同じ名前のシンボルが登場してもそれはもはや引き数とは関係ありません。例えば、

```
   a := x^3;
   f[x_] := x^2 + a;

   f[4]
Out[1]:= (16+(x^3))
```

の様に、f[4]はまず、右辺の x を引き数 4 で 4^2 + a と置き換えてからその評価にかかります。シンボル a を評価すると同じ名前のシンボル x に出会いますが、これはもはや引き数の値とは関係がありません。もちろん、

```
   f[x_] := (a := x^3; x^2 + a);

   f[4]
Out[1]:= 80
```

とすれば、今度は右辺の a の定義の中に x があらわに書かれているのでそれは引き数に置換されています。ちなみに、上で a := x^3 のように a の定義が SetDelayed で書かれているので、a を定義した時点では a の右辺は評価されません。にもかかわらず、a の定義は、この場合、x^3 ではなく、4^3 になっています。つまり、引き数の置換はあくまで置き換えであって、評価ではないということです。置き換えは右辺のなかの全ての場所で同時に起こり、それを阻止することはできません。

8.2 引き数の違いによる同一シンボルへの複数の関数定義

SADでは関数の定義に際して、その引き数の形態を選ぶことにより、同一のシンボルに対して異なった定義を与えることができます。最も簡単な例として、

```
   f[x_] := Sin[x] / x;
   f[0] = 1;
```

と関数 f を定義したとします。もし第1行だけだとすると、引き数が 0 の時に0/0となり、実行エラーを生じます。そこで第2行のように引き数の特殊な値に対する定義を書き加えるだけで、引き数が 0 の時には第2行の定義が実行され、エラーを防止できます。SAD はある関数を評価する場合、その引き数が関数の定義の引き数のパターンと照合し、照合すればその定義を実行します。この場合、より特殊な定義から順に照合します。従って、上の例では引き数 0 はどちらのパターンにも照合しますが、第2行の方が第1行よりも特殊なので第2行が優先的に照合され、実行されることになります。

もちろん上のような単純な場合は定義を一つにして、右辺の中に条件判断を設けても同じ結果を得ることができます。しかし、一般にはパターンの違いにより定義を区別するほうが記述が簡素になり、実行の効率も向上します。また、以下に見るように、引き数の個数の違いや式の形態などもパターンによれば簡単に区別できます。

8.3 関数の定義の解除

あるシンボルに対する関数の定義を解除したい場合は 10.7 にある Clear や Unset (=.) を用います。

8.4 パターンの変種

以下に「パターン式」というのは、パターンを含んでいるかもしれない式を指します。パターンでない一般の式は自分自身と同じものだけに照合します。

8.4.1 一つまたはそれ以上の個数の系列への照合

x__ というパターンは、関数の評価に際して、

（1）一つまたはそれ以上の任意の個数の任意の引き数の系列に照合します。

（2）右辺の定義式のなかに現われる同じ名前のシンボル(x)を、全て引き数の系列に置き換え、その後に右辺を評価する。

という動作をします。ここで「系列」Sequence とは特殊なリストで、他のリストあるいは式の中に入ると、その要素がそのまま親のリストの要素になる、という性質をもっています。例えば、

```
   f[x__] := {x};
```

と関数 f を定義すると、f[1] は {1}、f[1, 2, 3] は {1, 2, 3} 等々を返します。

照合に際しては最初のパターンから、系列の長さ1の場合から順に試されます。例えば、

```
   f[x__,y__] := {{x}, {y}};
   f[1, 2, 3]
Out[1]:= {{1},{2,3}}
```

の様に照合します。

8.4.2 ゼロまたはそれ以上の個数の系列への照合

x___ というパターンは、関数の評価に際して、

（1）ゼロまたはそれ以上の任意の個数の任意の引き数の系列に照合します。

（2）右辺の定義式のなかに現われる同じ名前のシンボル(x)を、全て引き数の系列に置き換え、その後に右辺を評価する。

という動作をします。ただし、照合に際しては最初のパターンから、系列の長さ0の場合から順に試されます。例えば、

```
   f[x___,y___] := {{x}, {y}};
   f[1, 2, 3]
Out[1]:= {{},{1,2,3}}
```

の様に照合します。

8.4.3 シンボルのないパターン

- _・__・___ というパターンは照合の性質はそれぞれ上に述べたものと同じですが、右辺の評価の際には何らの置き換えも起しません。これらは引き数の形態のみを判定する場合には有用です。

8.4.4 頭部を指定するパターン

- シンボル1_頭部1 というパターンは頭部が頭部1 の引き数に対してだけ照合します。
- 頭部1としては、現在はシンボルだけが指定可能です。

例えば、x_Real は実数に対してだけ照合します。

8.4.5 式に対しての照合

x : パターン式1 は関数の評価に際して、

（1）パターン式1 に照合する引き数に対してだけ照合する。

（2）右辺の定義式のなかに現われる同じ名前のシンボル（x）を、全て引き数に置き換え、また、パターン式1 の中に含まれるパターン達についても同様の置き換えを行う。その後に右辺を評価する。

- 例:

```
f[p : a|b|c] := Print[p];
```

は引き数が a, b, c のいずれかである場合にだけそれを Print します。

- パターン式1 は照合の前には評価されません。例えば、

```
   Clear[a,f];
   f[p : a] := Print[p];     (ア)
   a = 1;                    (イ)
   f[1]                      (ウ)
Out[4]:= f[1]
```

のように、（ア）の時点では a には値がないため、この関数の引き数はシンボル a に対してだけ照合します。従って、（イ）で a に後から値が設定されても、（ウ）のように、その新しい値には照合しません。

8.4.6 PatternTest

- パターン式1 ? テスト1 という式はまず、引き数がパターン式1 に照合するかどうかを調べます。もし照合すれば、式

  テスト1[照合した値]

  を評価し、その結果が True (0でない実数)ならば、元の引き数がパターン式1 ? テスト1 に照合したと判定します。

ここでテスト1としては、通常、次章に述べる純関数を使用します。

8.4.7 Alternatives

- パターン式1 | パターン式2 | ... はパターン式1, パターン式2, ... のいずれかに照合するパターンです。

8.4.8 Repeated, ..

- パターン式1 .. はそのいずれの要素もパターン式1 に照合する一つまたはそれ以上の任意の個数の引き数の系列に照合するパターンです。

8.4.9 RepeatedNull, ...

- パターン式1 ... はそのいずれの要素もパターン式1 に照合するゼロまたはそれ以上の任意の個数の引き数の系列に照合するパターンです。

8.4.10 照合しないときの値の指定

- パターン式1 : 式2 は引き数がパターン式1 に照合しないとき、パターン式1 に式2 の値を与え、照合させるパターンです。これはパターンに引き数が省略された時のデフォルト値を与えるひとつの方法です。

  この表記法と上述の「式に対する照合」の表記法は同じ形式なので混乱のもとになっています。この場合「パターン式1」の部分がシンボルの場合にのみ「式に対する照合」と解釈されます。

8.4.11 パターンを含む式

一般にパターン式(パターンを含む式)のなかには同じシンボルを持つパターンを含めても構いません。その場合は、同一のシンボルは同一の対象に照合しなければならないという条件があります。例えば、f[1, 2] はf[x_, y_] には照合しますが、f[x_, x_]には照合しません。

8.5 引き数の評価

さて、多くの関数ではその引き数は関数の定義との照合に先立って評価されます。しかし、例えば関数 Set の左辺、SetDelayed の両辺、Table の各引き数のように引き数を評価せず、そのままの形で関数に渡さなければならない場合も生じます。このような引き数の評価の属性は SetAttributes で指定することができます。

また、関数 Evaluate，Unevaluated（10.6.3、10.6.4）により、個々の場合の評価の有無を指定することもできます。

8.5.1 SetAttributes

- SetAttributes[シンボル1，HoldAll] はシンボル1 の全ての引き数を評価しないように指定します。
- SetAttributes[シンボル1，HoldFirst] はシンボル1 の最初の引き数だけを評価しないように指定します。
- SetAttributes[シンボル1，HoldRest] はシンボル1 の 2 番目以後の引き数を評価しないように指定します。
- SetAttributes[シンボル1，HoldNone] はシンボル1 のすべての引き数を評価するように指定します。
- SetAttributes[シンボル1，Constant] はシンボル1 を定数とみなし、それが入力された時点で直ちに評価されます。
- SetAttributes[{シンボル1，シンボル2，...}，属性1] はシンボル1，シンボル2，...に属性1 を与えます。
- SetAttributes[シンボル1，{属性1，属性2，...}] はシンボル1に属性1，属性2，... を与えます。

8.5.2 Null

ある関数を引用するときに f [... ,, ...] の様にカンマの間に何も書かない場合は、そこにシンボル Null があるとみなされます。

8.6 上方値

さて、これまで述べてきたシンボルへの関数の定義の割り当てはあくまでそのシンボルが関数の頭部になるようなものに限られていました。これに対して関数を割り当てられるシンボルが別の式の引き数に書かれているような関数の定義の仕方があります。例えば今、演算子 UpsetDelayed (^:=) により、

```
(seed = x_) ^:= SeedRandom[x];
```

とすると、以後、seed = n という式は SeedRandom[n] を実行することになります (SeedRandom は 11.7.3 参照)。ここで、UpsetDelayed の左辺の頭部は Set ですから、シンボル seed はその第一引き数に現われています。このような引き数のシンボルに対する関数の割り当てを上方値 upvalue の割り当てと呼び、きわめて応用範囲の広いものです。(これに対して通常の関数の定義を下方値 downvalue を呼びます。

上方値の定義の解除は Clear 10.7.1 により行います。

8.6.1 Upset

- 頭部0[引き数1, 引き数2, ...] ^= 右辺 は引き数1, 引き数2, ...がシンボルであるか、頭部がシンボルである式の場合に、それらのシンボル**全て**に対して右辺を評価した値を上方値として割り当てます。
- 引き数がシンボル[...][...] ... [...] のような形でもそのシンボルに上方値を割り当てます。
- 左辺の第 2 階以上に書かれたシンボルに対しては割り当てが行われません。
- シンボルが Protect により保護されている場合には割り当てが行われません。
- 頭部0 はどのようなパターン式でも構いません。

8.6.2 UpsetDelayed

- 頭部0[引き数1, 引き数2, ...] ^:= 右辺 は引き数1, 引き数2, ...がシンボルであるか、頭部がシンボルである式の場合に、それらのシンボル全てに対して右辺を評価せず、そのままの形で上方値として割り当てます。
- 引き数がシンボル[...][...] ... [...] のような形でもそのシンボルに上方値を割り当てます。
- 左辺の第 2 階以上に書かれたシンボルに対しては割り当てが行われません。
- シンボルが Protect により保護されている場合には割り当てが行われません。
- 頭部0 はどのようなパターン式でも構いません。

8.7 式の置換

SAD ではある式の一部を別の式に置き換えることができます。このような置換は、すでに関数の評価の際に引き数のパターンに対して実行されていることをみました。置換は ReplaceAll (演算子 /.) を用いて、どのような式に対しても実行できます。

8.7.1 ReplaceAll, /.

式0 /. パターン式1 -> 式1 は

- 式0 の値の各部分（頭部を除く）の中にパターン式1に照合するものがあれば、それを式1の値に置き換える。
- 式1 の中に、パターン式1 に含まれるパターンに一致するシンボルがさらにあれば、それらを式0 の照合された値に置き換える。

という作用をします。例: {a, b} /. x_ -> x + 1 ⇒ {a + 1, b + 1}。

演算子 /. の右辺には規則を与えます。規則とは、頭部が Rule（->）または RuleDelayed（:>）の式、あるいは**そのような式達から成るリスト**です。

8.7.2 Rule

- パターン式1 -> 式1 は置換の規則を与えます。この場合両辺ともまえもって評価されます。

8.7.3 RuleDelayed

- パターン式1 :> 式1 は置換の規則を与えます。この場合左辺はまえもって評価されますが、右辺は置換が行われた後に評価されます。

規則で左辺を評価しない簡単な方法はいまのところありませんが必要に応じて導入します。

8.7.4 ReplaceRepeated, //.

式0 //. 規則1 は置換が行われる限り、規則1による置換を続行します。

8.7.5 With

With はある式の一部を別の式で置き換えた後にその式を評価します。

- With[{式1, ...}、式0] は式0 の中の式1 に照合する部分を、式1 を評価した値で置き換えた後、式0 を評価します。式1 は照合の前には評価されません。
- With[{式1 = 式2, ...}、式0] は式0 の中の式1 に照合する部分を、式2を評価した値で置き換えた後、式0を評価します。式1 は照合の前には評価されません。
- With[{式1 := 式2, ...}、式0] は式0 の中の式1 に照合する部分を、式2 で置き換えた後、式0 を評価します。式1 は照合の前には評価されません。式2 は置換の前には評価されません。

例えば、SetDelayed や RuleDelayed の右辺のように普通は評価されない式でも、With を使えばその一部だけを評価することができます。また、With は複雑な式を見やすくするためにも有効です。更に、式の中で定数になる部分をあらかじめ評価しておくことにより、実行の効率を向上させることもできます。

With は後に述べる Module や Block に構造が似ていますが、機能はまったく違います。後者達が局所的なシンボルを生成するのに対して、With はそのような作用はしません。例えば、上の式1に現われるシンボルは照合の時に使われるだけで、式0 の評価の時には式1 そのものは既に置換されています。

8.8 シンボルの視野

関数の定義では、多くの場合その関数の内部だけで使用するシンボルが必要になります。そのようなシンボルを局所シンボルと呼びます。局所シンボルは通常関数 Module で定義します。Module には二つの引き数があり、Module[{局所シンボル1, ...}, 式1] の様に用います。第一の引き数は局所シンボルのリストです。これにより、式1 の中に現われる同じ名前のシンボルは全てこの局所シンボルであるとみなされ評価されます。Module はそのように式1を評価しその結果を返します。例えば、

```
a := b;          (ア)
Module[{b},      (イ)
  b = c;         (ウ)
  a + b          (エ)
]
```

としますと、(ア)は Module の外で単にシンボル a にシンボル b を割り当てています。ここで SetDelayed が用いられているため、右辺のシンボル b はまだ評価されていません。(イ)はこの Module の中だけで有効な局所シンボル b を定義しています。(ウ)と(エ)がこの Module の第二引き数にあたります。まず、(ウ)では局所シンボル b に別のシンボル c を割り当てています。次に(エ)ではシンボル a と局所シンボル b の和を求めてそれをこの Module の結果にしようとしていますが、その結果は b + c になり、c + c ではありません。シンボル a にはシンボル b が評価されずに割り当てられており、シンボル b が評価されると一見シンボル c になりそうに思えますが、実はModule **の外で書かれたシンボルと** Module **の内側で定義された局所シンボルは全く別のものである**という規則があるので、上のような結果になります。

局所シンボルは Module の実行が終わると、それへの値の割り当ては全て解除されます。

8.8.1 Module

- Module[{局所シンボル1, ...}, 式0] はこの Module の中だけで有効な局所シンボル1, ...を定義してから、式0を評価しその結果を返します。
- Module[{局所シンボル1 = 値1, ...}, 式0] は局所シンボル1 を定義し、且つその初期値として値1 を割り当てから式0 を評価します。
- Module[{{局所シンボル1, ...} = {値1, ...}, ...}, 式0] のように局所シンボル1, ... への初期値の割り当てをリスト形式で行うこともできます。

8.8.2 Block

関数 Block は局所シンボルは定義せず、大域的シンボルの**値だけ**を局所的に設定・変更するのに用います。宣言された大域シンボルの値は Block の終了とともに元に戻ります。

- Block[{大域シンボル1, ...}, 式0] は局所的に使用する大域シンボル1, ...を宣言してから、式0を評価しその結果を返します。
- Block[{大域シンボル1 = 値1, ...}, 式0] は局所的に使用する大域シンボル1 を宣言し、且つその初期値として値1を割り当てから式0を評価します。
- Block[{{大域シンボル1, ...} = {値1, ...}, ...}, 式0] のように局所的に使用する大域シンボル1, ... への初期値の割り当てをリスト形式で行うこともできます。

8.8.3 局所シンボルの表示

- 局所シンボル自身が ToString や Print などで文字列に変換される場合はシンボル名$*nnn*のようにシンボル名の後ろに文脈番号がつけられます。この文脈番号は Module が呼ばれる度に 1 ずつ増加します。

8.9 関数ライブラリ

ユーザーは関数の定義をファイルに書くことによりライブラリを構築できます。そのようなファイルは単に SAD の入力ファイルと同じ書式で書けばよいのです。ただし、複数の式を並べる場合には、各々の式はセミコロン（CompoundExpression 10.1.1 参照）で区切らなければなりません。さもないと、式と式の間に演算子 Times があるとみなされます。

ライブラリのロードには Get（13.1.5）か AutoLoad（10.7.6）を用います。

9　構造的演算

9.1　純関数

さて、これまであるシンボル f に対して関数 f[...] を定義する方法を述べてきましたが、SAD にはこのようなシンボル f を使わなくても定義できる関数の形式があります。これを純関数 pure function と呼びます。純関数は構造的演算に際して多用されます。

9.1.1　純関数1，演算子 & と Slot

- (式1)& は純関数を表わします。式1は Slot（#，#n，##，##n）を含む式です。純関数の引き数はこの Slot を通して受け渡されます。
- (式1)&[引き数1，引き数2，..] は純関数を引き数1，...に対して適用します。式1が、その引き数をこの Slot を通して受け渡されつつ評価され、その結果が返されます。
- # は引き数1を表わします。
- #n は引き数nを表わします。
- ## は引き数全体の系列を表わします。
- ##n は引き数n以降の系列を表わします。

例えば、(Sin[#] / Cos[#2])&[a, b] ⇒ Sin[a] / Cos[b] など。

純関数それ自身は評価されても形を変えません。

純関数の中で純関数を含む式を多重に使用することは可能です。ただし、その場合、Slot と引き数の対応がわかりにくくなることが避け難いので、なるべくならば With により純関数に名前をつけるなどの工夫が望ましいでしょう。また、場合によっては Slot ではどうしても引き数の対応ができない場合も発生します。

9.1.2　純関数2 Function

純関数の多重使用により、Slot による対応が不可能なときには Function を用いて Slot によらない引き数の対応をおこなう純関数を定義できます。

- Function[{シンボル1，...}，式1] は純関数で、その引き数はシンボル1，...に対応します。式1はシンボル1，...を含む式で、純関数の胴体です。
- Function[{x, y}, Sin[x] / Cos[y]][a, b] ⇒ Sin[a] / Cos[b]。
- 引き数のシンボルは純関数の胴体に露に書かれたもののみが対応する値に置き換わります。

9.2 構造的演算とは

SAD では個々の原子、リストの要素に対しては勿論様々な演算が可能ですが、それ以上にあるリストや式の全体に対して一度に演算をすることができます。例えば、算術演算はリストに対してはその要素に並列に作用することは既に述べました。実は一般の関数に対してもそのような演算が可能です。ここではリストや式の全体に作用させる演算を「構造的演算」と呼ぶことにします。構造的演算の代表的なものが Map（演算子 /@)と Apply（演算子 @@）です。これらは SAD ではきわめて多用されるため、特別な演算子まで与えられています。

いま、ある2階のリスト l があるとして、その各要素の長さを要素とするリストを造りたいとしましょう。この問題は Map(/@) を使えば、

```
Length /@ l
```

だけで書くことが出来ます。一般にある関数 f に対して、f /@ $\{a_1, a_2, ...\}$ は $\{f[a_1], f[a_2], ...\}$ を実行します。

次に、数値を要素とするあるリスト l に対して、その合計、平均値、2 乗平均値を求めたいとします。これらは Apply(@@) を使って、それぞれ

```
Plus @@ l
Plus @@ l / Length[l]
Plus @@ (l^2) / Length[l]
```

と書かれます。一般にある関数 f に対して、f @@ $\{a_1, a_2, ...\}$ は $f[a_1, a_2, ...]$ を実行します。

この様に、構造的演算により、複雑な演算をきわめて簡潔に書くことが出来ます。構造的演算は単に簡潔さだけでなく、実行速度の向上にも寄与します。しかし、構造的演算を一つの式の中で幾重にも重ねますと、プログラムの可読性は低下してしまいますので注意してください。

9.3 様々な構造的演算

以下において演算の作用するリストは必ずしも頭部が List である必要はありません。

これらの内、Map, MapAll, MapIndexed, Apply, Scan, Cases, DeleteCases に対しては、階数指定子（Level 7.3.4 を参照）により演算の範囲を指定できます。

9.3.1 Map, /@

- f /@ リスト1 はリスト1 の各要素に関数 f を作用させます。
- Map[f, リスト1, 階数指定子1] はリスト1の階数指定子1（Level 7.3.4 を参照）で指定される階の各要素に関数 f を作用させます。
- f /@ リスト1 と Map[f, リスト1] は Map[f, リスト1, {1}] と同値です。

9.3.2 MapAll, //@

- *f* //@ リスト1 はリスト1 のすべての階（0 階を含む）の全要素に関数 *f* を作用させます。
- *f* //@ リスト1 はMap[*f*, リスト1, {0, Infinity}] と同値です。
- 結果の頭部はもとの式の頭部になります。
- MapAll[*f*, リスト1, Heads -> True] はリスト1のすべての階の全要素と頭部に関数 *f* を作用させます。

9.3.3 MapIndexed

- MapIndexed[*f*, リスト1] はリスト1 の各要素に関数 *f* を作用させますが、そのときその要素のリスト1のなかでの位置を第二引き数として *f* に渡します。
- MapIndexed[*f*, リスト1, 階数指定子1] はリスト1 の階数指定子1（Level 7.3.4 を参照）で指定される階の各要素に関数 *f* を作用させますが、その要素のリスト1 のなかでの位置を第二引き数として *f* に渡します。

9.3.4 Apply, @@

- *f* @@ リスト1 はリスト1 の全要素からなる系列を引き数として関数 *f* を評価します。
- Apply[*f*, リスト1, 階数指定子1] はリスト1の階数指定子1（Level 7.3.4 を参照）で指定される階の全要素からなる系列を引き数として関数 *f* を評価します。
- *f* @@ リスト1 と Apply[*f*, リスト1] は Apply[*f*, リスト1, {0}] と同値です。
- 結果の頭部は、その階に Apply が作用していなければ、もとの式の頭部が保存されます。例えば、Apply[f, g[h[1, 2], h[3, 4]], {1}] ⇒ g[f[1, 2], f[3, 4]]。

9.3.5 Scan

- Scan[関数1, リスト1] は関数1をリスト1の1階の各要素に適用します。最期に Null を返します。
- Scan[関数1, リスト1, 階数指定子1] は関数1をリスト1の、階数指定子1（Level 7.3.4 を参照）で指定される各要素に適用します。最後にNullを返します。
- Scan[関数1, リスト1] は Do[関数1[リスト1[[i]]], {i, Length[リスト1]}]と同じ結果になりますが、一般に速度が速く効率的です。（「添字はできるだけ避けよ。」）

9.3.6 Position

- Position[式1，パターン式1] は式1の全部分式の中でパターン式1に照合するものの位置達をリストにして返します。
- Position[式1，パターン式1，階数指定子1] は階数指定子1（Level 7.3.4 を参照）で指定される階数の、パターン式1に照合する部分式の位置達を返します。
- Position[式1，パターン式1，階数指定子1，$n1$] は階数指定子1で指定される階数の、パターン式1に照合する部分式の位置達の内の最初の $n1$ 個を返します。

9.3.7 Count

- Count[式1，パターン式1] は式1の全部分式の中でパターン式1に照合するものの個数を返します。
- Count[式1，パターン式1，階数指定子1] は階数指定子1（Level 7.3.4 を参照）で指定される階数の、パターン式1に照合する部分式の個数を返します。
- Count[式1，パターン式1，階数指定子1，$n1$] は階数指定子1で指定される階数の、パターン式1に照合する部分式の位置達の内の最初の $n1$ 個の部分式の個数を返します。

9.3.8 Cases

- Cases[リスト1，パターン式1] はリスト1 の要素の中でパターン式1 に照合するもの達をリストにして返します。
- Cases[リスト1，パターン式1，階数指定子1] は階数指定子1（Level 7.3.4 参照）で指定される階数の、パターン式1に照合する部分式達を返します。
- Cases[リスト1，パターン式1，階数指定子1，$n1$] は階数指定子1 で指定される階数の、パターン式1に照合する部分式達の内の最初の $n1$ 個を返します。
- 上記のパターン式1をパターン式1 -> 変更値1のように書くと、結果に対してこの規則による置換が行われます。

9.3.9 DeleteCases

- DeleteCases[リスト1，パターン式1] はリスト1 の要素の中でパターン式1 に照合するもの達を除いたリストを返します。
- DeleteCases[リスト1，パターン式1，階数指定子1] は階数指定子1（Level 7.3.4 を参照）で指定される階数の、パターン式1 に照合する部分式達を除いたリストを返します。

9.3.10 MapAt

- MapAt[関数1，リスト1，$n1$] はリスト1の$n1$ 番目の位置に関数1 を適用し、その結果に置き換えたリストを返します。
- MapAt[関数1，リスト1，{$n1$，$n2$，... }] はリスト1の {$n1$，$n2$，...} の位置（リスト1[[$n1$，$n2$，...]]） に関数1 を適用し、その結果に置き換えたリストを返します。
- MapAt[関数1，リスト1，{{$n1$，...}，{$n2$，... }}] はリスト1 の幾つかの位置達に関数1 を適用し、その結果に置き換えたリストを返します。

9.3.11 MapThread

- MapThread[f, {{a_1，a_2，...},{b_1，b_2，...}，...}] は {f[a_1，a_2，...]，f[b_1，b_2，...]，...} を返します。
- MapThread[関数1，{式1，式2，...}，階数指定子1] は階数指定子1 で指定される式1，式2，... の部分式達に関数1 を適用します。

9.3.12 Nest

- Nest[関数1，式1，n] は式1 に関数1 を
- Nest[f, x, 3] ⇒ f[f[f[3]]] 。

9.3.13 Select

- Select[リスト1，関数1] はリスト1の各要素に関数1を適用し、その結果が True（非0）になるもの達から成るリストを返します。
- Select[リスト1，関数1，$n1$] はリスト1の各要素に関数1を適用し、その結果が True（非0）になるもの達の内、最初の $n1$ 個から成るリストを返します。

9.3.14 SwitchCases

- SwitchCases[リスト1，{パターン式1，パターン式2，...}] はリスト1の各要素をパターン式1，パターン式2，...に照合するものに分類したリストを返します。
- SwitchCases[{1, "a", x, 4}, {_Real, _String}] ⇒ {{1, 4}, {"a"}}。
- もしどれにも分類されないものも結果に必要な場合はパターン式のリストの最後に_を加えれば済みます。

9.3.15 SelectCases

- SelectCases[リスト1，{関数1，関数2，...}] はリスト1の各要素を、関数1，関数2，...を適用した結果が True（非0）になるものに分類したリストを返します。
- SelectCases[{1，2，3，4}，{(#<2)&，(#>=3)&}] ⇒ {{1}，{3，4}}。
- もしどれにも分類されないものも結果に必要な場合は関数のリストの最後に True& を加えれば済みます。

10　プログラミング

SAD では条件判断やループといったプログラムの流れの制御も関数によって行います。

10.1　式の連結

10.1.1　CompoundExpression

- 式1; 式2; ...　; 式n は式1から式nをこの順に評価し、最後の式の結果を返します。
- 式1; 式2; ...　; 式n; は式1から式nをこの順に評価し、Null を返します。TracePrint 10.9.1 の中では各式が評価の前にそのまま印刷されます。

10.1.2　Goto と Label

SAD には実際にはあまり使われませんが Goto もあります。

- Goto[ラベル1] はその Goto と同一またはそれより下の階の CompoundExpression に含まれる Label[ラベル1の値]のところまで制御を移します。ここでラベル1は任意の式でよい。Label 自身はなにもせず、自分自身の式を返します。

10.2　条件式

SAD の条件演算子、論理演算子は実数値0を偽、0でない実数を真とします。0以外の実数はどれも真なのですが、実際の演算では真の場合には実数値1が返されます。論理式の結果はこのように実数ですから、他の算術式の中に混ぜても構いません。また、システムでは True, False の二つのシンボルを用意していますが、これらは単にそれぞれ実数1と0と同値です。また、条件判断を要求される場合に普通の算術式を書いてもかまいません。

条件演算子の中で、SameQ(===) と UnsameQ(<=>)以外のものは真または偽以外の値、すなわち条件式そのものをかえすことがあります。例えば、条件式 a == 1 で、もし a に何も値が割り当てられていないときは、この条件式の結果は真でも偽でもなく、じつはこの式　a == 1 そのものです。また、a に実数値以外の値が割り当てられているときも (aの値) == 1 のような式が返されます。(このようになるひとつの理由は、a == 1 の様な条件式は、いくつかの関数の中で方程式としての意味を持つからです)。これに対して、演算子 SameQ(===) と UnsameQ(<=>) は必ず真または偽の結果を返します。

10.2.1　SameQ, ===

- 式1 === 式2　は両辺の結果が SAD の要素として同一の値(式)のとき True(1)、それ以外は False(0) を返します。

10.2.2 UnsameQ, <=>

- 式1 <=> 式2 は両辺の結果が SAD の要素として同一の値(式)でない時 True(1)、それ以外は False(0) を返します。

10.2.3 MatchQ

- MatchQ[式1, パターン式1] は式1 がパターン式1 に照合するとき True(1)、照合しないとき False(0) を返します。

10.2.4 MemberQ

- MemberQ[式1, パターン式1] は式1 の全ての部分式の中にパターン式1 に照合するものが存在するとき True(1)、照合しないとき False(0) を返します。

- MemberQ[式1, パターン式1、階数指定子1] は式1 の階数指定子1 (Level 7.3.4 を参照) で指定される階数の部分式の中にパターン式1 に照合するものが存在するとき True(1)、照合しないとき False(0) を返します。

10.2.5 FreeQ

- FreeQ[式1, パターン式1] は式1 の全ての部分式の中にパターン式1 に照合するものがひとつも存在しないとき True(1)、ひとつでも照合するとき False(0) を返します。

- FreeQ[式1, パターン式1、階数指定子1] は式1 の階数指定子1 (Level 7.3.4 を参照) で指定される階数の部分式の中にパターン式1 に照合するものがひとつも存在しないとき True(1)、ひとつでも照合するとき False(0) を返します。

10.2.6 VectorQ

- VectorQ[式1] は式1 の値が一階のリストの時 True(1) になり、それ以外は False(0) になります。

- VectorQ[式1, テスト1] は式1 の値が、その要素がすべて関数テスト1 を作用させた時真 (非0) の場合 True(1) になり、それ以外は False(0) になります。

10.2.7 MatrixQ

- MatrixQ[式1] は式1 の値が行列の時 True(1) になり、それ以外は False(0) になります。

- MatrixQ[式1, テスト1] は式1 の値が行列で、その各要素がすべて関数テスト1 を作用させた時真 (非0) の場合 True(1) になり、それ以外は False(0) になります。

10.2.8 ComplexQ

- ComplexQ[式1] は式1 の値が実数でない複素数、あるいは、少なくともひとつの要素がそのような数であるリストの時 True(1) になり、それ以外は False(0) になります。

10.3 条件判断

10.3.1 If

- If[式1，真1] は式1 を評価し、その結果が True ならば式真1 を評価しその結果を返します。そうでなければ Null を返します。

- If[式1，真1，偽1] は式1 を評価し、その結果が True ならば式真1 を、偽ならば式偽1 を評価しその結果を返します。どちらでもなければ Null を返します。

- If[式1，真1，偽1，他1] は式1を評価し、その結果が True ならば式真1 を、偽ならば式偽1 を、どちらでもなければ式他1 を評価しその結果を返します。

10.3.2 Or, ||

- 式1 || 式2 || ... は論理和ですが、その結果が真になるまで、式1、式2、...を評価し続けます。真になれば残りの式はそれ以上評価されず、True(1) が返されます。したがって、実質的な条件判断としても利用できます。

10.3.3 And, &&

- 式1 && 式2 && ... は論理積ですが、その結果が偽になるまで、式1、式2、...を評価し続けます。偽になれば残りの式はそれ以上評価されず、False(0) が返されます。したがって、実質的な条件判断としても利用できます。

10.3.4 Not, ~

- ~式1 は論理否定で、式1 を評価し、その結果が 0 以外の実数ならば False(0)、0 ならば True(1)、それ以外は ~式1 の式そのものを返します。

10.3.5 Switch

- Switch[式0，パターン1，式1，パターン2，式2，...] はまず式0 を評価し、その結果がパターン1に照合すれば式1 を評価し値を返します。そうでなければパターン2，...と照合されるまで進みます。もしどのパターンとも照合しなければ Switch の式そのものが返ります。

- もしどれとも照合しないときに式a を評価したければ、引き数の最後に _，式a と書けば実行できます。

10.3.6 Which

- Which[条件1, 式1, 条件2, 式2, ...] は条件1 を評価し真になれば式1を評価し値を返します。そうでなければ条件2, ... と真になるまで進みます。もしどの条件も真でなければ Which の式そのものが返ります。
- もしどれも真でないときに式a を評価したければ、引き数の最後に True, 式a と書けば実行できます。

10.4 ループ

10.4.1 Do

- Do[式1, 反復指定子] は反復指定子（7.2.1 を参照）で指定された回数だけ式1 を評価し、最期に Null を返します。
- 途中で Break[] が呼ばれると反復は中断します。
- 途中で Continue[] が呼ばれると式1の評価は中断しますが、反復は続きます。

10.4.2 While

- While[条件式1, 式2] は条件式1 を毎回評価し、結果が真(非0)である限り式2 を評価しつづけます。最期に Null を返します。
- 途中で Break[] が呼ばれると反復は中断します。

10.4.3 For

- For[開始1, 条件1, 増分1, 式1] はまず、開始1 を評価します。次ぎに条件1が真である限り、式1 と増分1 をこの順に評価します。
- 途中で Break[] が呼ばれると反復は中断します。

10.4.4 Scan

- Scan[関数1, リスト1] は関数1 をリスト1 の1階の各要素に適用します。最期に Null を返します。
- Scan[関数1, リスト1, 階数指定子1] は関数1 をリスト1の、階数指定子1（Level 7.3.4 を参照）で指定される各要素に適用します。最後に Null を返します。
- Scan[関数1, リスト1] は Do[関数1[リスト1[[i]]], {i, Length[リスト1]}] と同じ結果になりますが、一般に速度が速く効率的です。(「添字はできるだけ避けよ。」)

10.4.5 Sum

- Sum[式1, 反復指定子] は反復指定子 (Table 7.2.1 を参照)で指定された回数だけ式1を評価し、それらの結果の合計を返します。

10.4.6 Product

- Product[式1, 反復指定子] は反復指定子 (Table 7.2.1 を参照)で指定された回数だけ式1を評価し、それらの結果の積を返します。

10.5 プログラムの中断、例外処理

10.5.1 Break

- Break[] はDo, While, For の実行を中断し、それらの次の式に制御を移します。

10.5.2 Continue

- Continue[] は Do のその回の実行を中断しますが、反復は継続します。

10.5.3 Return

- Return[式1] は関数の実行を中断し、式1の値をその関数の値として返します。

10.5.4 Exit

- Exit[] は SAD を終了させます。

10.5.5 Throw

- Throw[式1] は Catch の中で用いられ、プログラムの実行を中断し、式1の値を Catch の値として返します。

10.5.6 Catch

- Catch[式1] は式1を評価し、その値を返します。途中で Throw[式2] が呼ばれると、プログラムはそこで中断し、式2の値を返します。

10.6 式の評価の制御

10.6.1 Hold

- Hold[式1] は式1を評価せず、Hold[式1] の形のままの式を結果として返します。
- Hold[Sin[1]] ⇒ Hold[Sin[1]]。
- 式1の一部を評価するには With 8.7.5 を使います。
- ある式の一部を評価せずに取り出すには Extract 7.4.5 と Hold を組み合わせて使います。
- Extract[Hold[{Sin[1], Cos[1]}], {1, 2}, Hold] ⇒ Hold[Cos[1]]。

10.6.2 ReleaseHold

- ReleaseHold[Hold[式1]] は式1 を評価し、その結果を返します。
- ReleaseHold[Hold[Sqrt[4]]] ⇒ 2。
- 引き数が上の様に頭部に Hold を持たない場合、ReleaseHold[式1] は式1 を評価し、その結果を返します。
- ReleaseHold[{Hold[Sqrt[4]]}] ⇒ {Hold[Sqrt[4]]}。

10.6.3 Evaluate

- f [..., Evaluate[式1], ...] は関数 f の引き数の評価の有無の指定にかかわらず、式1 を評価した値を関数f に渡します。
- Hold[Evaluate[Sqrt[4]]] ⇒ Hold[2]。
- 上の形のように関数の引き数の頭部以外に Evaluate が現われる場合は Evaluate[式1] はそれが**もし評価されれば**式1 を評価した値を返します。
- Evaluate[Sqrt[4]] ⇒ 2。
- Hold[{Evaluate[Sqrt[4]]}] ⇒ Hold[{Evaluate[Sqrt[4]]}]。
- 式1の一部を評価するには With 8.7.5 を使います。

10.6.4 Unevaluated

- f [..., Unevaluated[式1], ...] は関数 f の引き数の評価の有無の指定にかかわらず、式1 を評価せずそのままの形で関数 f に渡します。

10.7 シンボルの設定、診断

10.7.1 Clear

- Clear[シンボル1，シンボル2，...] はシンボル1，シンボル2，... に割り当てられている全ての値及び関数の定義を解除します。
- Clear[シンボル1[引き数1，...]，...] はシンボル1 に割り当てられている上方値 8.6 のうち、シンボル1[引き数1，...] の形を含むものを解除します。

10.7.2 Unset, =.

- シンボル1 =. はシンボル1 に割り当てられている全ての値及び関数の定義を解除します。
- シンボル1[引き数1，...] =. はシンボル1[引き数1，...] に割り当てられている関数の定義だけを解除します。
- 上方値 8.6 は Unset では解除できません。
- 部品シンボルに対しては部品シンボル1 =. は DeleteWidget[部品シンボル1] と同じ働きをします。

10.7.3 Names

- Names[文字列パターン1] はその名前が文字列パターン1（12.5 文字列の照合を参照）に照合する、既に定義されたシンボル名達をリストにして返します。
- Names["Arc*"] ⇒
{"ArcTanh", "ArcCosh", "ArcSinh", "ArcTan", "ArcCos", "ArcSin"}。

10.7.4 Protect

- Protect[シンボル1，シンボル2，...] はこれ以後のシンボル1，シンボル2，... への値や関数の割り当てを禁止します。
- 局所シンボル（8.8）は Protect できません。
- システムが用意する関数やシンボルは Protect されています。

10.7.5 Unprotect

- Unprotect[シンボル1，シンボル2，...] はこれ以後のシンボル1，シンボル2，... への値や関数の割り当てを許可します。

10.7.6 AutoLoad

必要に応じてある関数ライブラリをロードするには AutoLoad を使います。

- AutoLoad[シンボル1，シンボル2，...，ファイル1] はシンボル1，シンボル2，... が次に評価される時にファイル1 の内容が評価されるという仮の定義をします。
- ファイル1 にはシンボル1，シンボル2，... の真の定義を書いておきます。

10.7.7 Order

- Order[式1，式2] は式1 の値が式2 の値に対して標準的な順序（表 30 を参照）で前の時 1、等しい時 0、後ろの時 -1 になります。

10.8 メッセージとエラー処理

SAD は関数の実行中にエラーが発生するとそれに対応するメッセージを生成します。そしてその関数の実行は中断され、関数の結果はその関数を呼び出した形そのものになります。その時点でメッセージは端末に出力されますが、関数 Off により出力を抑制することもできます。また、メッセージの重要度に応じて、その後のプログラムの実行は中断または継続されます。また、メッセージの発生は関数 Check により検出することができます。

次の例

```
  Log[1, 2, 3] + Sin[1, 2]
???General::narg:  Number of arguments is expected 1 or 2 in Log[1,2,3]
???General::narg:  Number of arguments is expected 1 in Sin[1,2]
Out[4]:= (Log[1,2,3]+Sin[1,2])
```

はこのようなメッセージが発生した例です。ここで General::narg がこのメッセージを識別する式です。この例ではまず Log[1, 2, 3] でエラーが発生していますが、そのまま実行が継続され次の Sin[1, 2] を実行し、そこでまた同じメッセージを生成しています。これらは同じメッセージでありながら末尾が異なっています。実はこの General::narg というメッセージは

```
   General::narg
Out[5]:= "Number of arguments is expected `1`"
```

というもので、この中の `1` には別の文字列が代入されるようになっています。また、行頭の ??? と行末の in ... 以降はシステムが付加するものです。

ユーザーはまた、自分のプログラムのなかで随時メッセージを発行することができます。そのようなメッセージの登録は関数 MessageName(演算子 ::) で行います。

10.8.1 MessageName, ::

- シンボル1::タグ1 はシンボル1 とタグ1 で識別される、登録されたメッセージを返します。
- シンボル1::タグ1 のメッセージが未登録の場合は General::タグ1 のメッセージが返されます。
- シンボル1::タグ1 = メッセージ1 はシンボル1 とタグ1 で識別されるメッセージ1 を登録します。シンボル1 とタグ1 は任意の式で構いませんが通常はシンボルが用いられます。なお、シンボル1 とタグ1は評価されずそのままの形で用いられます。
- Print[Definition[MessageName]] により登録されているメッセージをすべて端末に表示できます。

10.8.2 Off

- Off[シンボル1::タグ1] はメッセージシンボル1::タグ1 の出力を以後停止します。
- Off されたメッセージは Check で検出できません。

10.8.3 On

- On[シンボル1::タグ1] はメッセージシンボル1::タグ1 の出力を以後可能にします。
- On[General::newsym] とすると、以後新しい名前のシンボルが生成される度にメッセージ General::newsym が出力されます。これはシンボルの書き誤りなどを検出する場合の助けになります。

10.8.4 $MessageList

- シンボル $MessageList にはその入力行の実行中にそれまでに発生したメッセージの識別子のリストが割り当てられます。各識別子には Hold が被されています。

10.8.5 MessageList

- MessageList[n] は出力行nの実行中に発生したメッセージの識別子のリストを返します。各識別子には Hold が被されています。
- 次の出力行の番号はシンボル $Line に割り当てられています。Out 10.9.3 参照。

10.8.6 Check

- Check[式1，式2] は式1 を評価し、その途中でメッセージが発生した場合、式2 を評価してその値を返します。エラーが発生しない場合は式1 を評価した値を返します。
- Check[式1，式2，シンボル1::タグ1，...] はシンボル1::タグ1，... で識別されるメッセージが発生した場合だけ式2 を評価します。
- Off されたメッセージには Check は反応しません。

10.8.7 Message

- Message[シンボル1::タグ1] はシンボル1::タグ1 で識別されるメッセージを生成します。
- Message[シンボル1::タグ1，文字列1，...] はシンボル1::タグ1 の中の ‘1‘，... という文字列を文字列1，...に置き換えたものを表示メッセージとして生成します。

10.9 デバッグの方法

SAD の現状ではデバッグの道具はあまり充実していません。原始的な方法としては

- TracePrint[式1] により、エラーの発生箇所を特定する。
- Print[式1] あるいは Print[Definition[式1]] などを要所要所に挿入する。
- On[General::newsym] により新しいシンボルの生成を検出する（10.8.3 参照）。
- プログラムの最上位レヴェルにシンボル end を挿入する。

などが考えられます。

しばしば発生する誤りは

- 大文字と小文字の書き誤り。
- セミコロンの書き忘れ。
- セミコロンとカンマの書き誤り。
- 関数の引き数の型、個数の違い。

などです。関数の用法が未定義の場合、いくつかの関数はその呼ばれた式そのものを結果として返すだけでメッセージを発生しないことに注意する必要があります。

10.9.1 TracePrint

- TracePrint[式1] は CompoundExpression (;) のそれぞれの式を、その実行の直前に端末に表示します。

10.9.2 Definition

- Definition[シンボル1] はシンボル1 に割り当てられている定義式を返します。

10.9.3 Out、%

- Out[n] は端末に Out[n] := ... と既に出力された、n 番目の評価結果を再評価して返します。
- % は最後の評価結果を返します。
- %Out[n] と同値です。
- 最後の評価結果の番号はシンボル $Line に割り当てられています。この番号を自分でリセットすることもできますが、その場合、新しい $Line 以前の Out[$n$] の値は失われます。
- $Line は式の評価が完結する度に 1 ずつ増加します。式の評価結果が Null のときは $Line は更新しません。

10.9.4 シンボル end

入力プログラムの最上位の式と式の切れ目にシンボル end を挿入しておくと、SAD はそこで実行を中断し、端末からの入力待ち状態になります。そこでシンボルの状態などを対話式に確認することができます。実行を再開するには in 77 と入力します。

10.9.5 MemoryCheck

- MemoryCheck[] はその時点のメモリーの使用状況を返します。また、システムのバグのためにメモリーが破壊されているかどうかをチェックし、もし破壊されていればメッセージを出力します。
- もしメッセージが出力された場合は筆者までレポートをお願いします。

10.9.6 STACKSIZ

- STACKSIZ は SADScript インタープリータの実行に必要なスタックの大きさを指定するシンボルです。
- 最初の FFS セッションが**始まる前**に STACKSIZ = n; と指定することにより、以後のスタックの大きさを変えることができます。デフォルトは STACKSIZ = 200000 です。
- FFS セッションが開始された後は STACKSIZ を変更しても効果はありません。

10.10 システムとの相互作用

10.10.1 System

- System[コマンド文字列] はコマンド文字列であらわされるシステムのコマンドを実行します。
- コマンドの実行結果を読みたい場合は OpenRead（13.1.1 参照）を用います。

10.10.2 Environment

- Environment[環境変数文字列] は環境変数文字列で表わされる環境変数の値を文字列で返します。
- GetEnv と Environment は同値です。

10.10.3 Directory

- Directory[] は現在作業中のディレクトリ名を返します。

10.10.4 SetDirectory

- SetDirectory[ディレクトリ名1] は作業ディレクトリをディレクトリ名1 に設定し、その完全名称を返します。

10.10.5 HomeDirectory

- HomeDirectory[] はホーム・ヂレクトリ名を返します。

10.10.6 GetPID

- GetPID[] は SAD のプロセスID 番号を返します。

10.10.7 GetUID

- GetUID[] は SAD のユーザID 番号を返します。

10.10.8 GetGID

- GetGID[] は SAD のグループID 番号を返します。

10.11 プロセス制御

10.11.1 Pause

- Pause[秒] は指定された秒数だけプログラムの実行を休止します。
- 1 秒以下の指定も可能です。
- Sleep と Pause は同値です。

10.11.2 Fork

- Fork[] は実行中のプロセスのコピーを作り、サブプロセスとして実行します。
- Fork[] は親のプロセスには子のプロセスのプロセス番号を返し、子のプロセスには 0 を返します。

10.11.3 Wait

- Wait[] はサブプロセスの実行終了を待ちます。
- Wait[] は {サブプロセス番号，終了コード} というリストを返します。サブプロセスが正常終了したときは終了コードは 0 です。

10.12 その他の関数

10.12.1 Date

- Date[] はその時点の時刻を {年，月，日，時，分，秒} のリストで返します。

10.12.2 Day

- Day[] はその時点の曜日名を返します。
- Day[{年，月，日}] はその日付の曜日名を返します。
- Day[{年，月，日，時，分，秒}] はその日付の曜日名を返します。

10.12.3 FromDate

- FromDate[{年，月，日，時，分，秒}] はリスト表示の時刻の、1900 年 1 月 1 日 0 時 0 分 0 秒から計った経過時間を秒単位の数値で返します。

10.12.4 ToDate

- ToDate[秒] は1900 年 1 月 1 日 0 時 0 分 0 秒から計った秒単位の経過時間を{年，月，日，時，分，秒}のリストに直して返します。

10.12.5 DateString

- DateString は呼ばれたときの時刻を文字列にして返します。

10.12.6 TimeUsed

- TimeUsed[] はその SAD セッションが消費した CPU 時間を秒単位で返します。

10.12.7 Timing

- Timing[式1] は式1 を評価し、{式1の結果，消費した CPU 時間（秒)} というリストを返します。

11 数値関数

11.1 初等関数

SAD には以下の初等関数が備わっています。これらは原則として複素数にも適用できます。また、これらの関数はリストに対してはその要素に並列に作用します。

- `ArcCos ArcCosh ArcSin ArcSinh ArcTan ArcTanh Cos Cosh Exp Log Sin Sinh Sqrt Tan Tanh`。

11.1.1 Log

- `Log[`z`]` は z の自然対数。
- `Log[`a`, `z`]` は z の底 a の対数、$= \log_a z$。

11.1.2 ArcTan

- `ArcTan[`z`]` = $\tan^{-1} z$。
- `ArcTan[x, y]` = $\arg(x+iy)$、$\pm\pi$ の間の値をとる。

11.2 特殊関数

現在までに備えられている特殊関数は以下の通りです。ご覧の通りあまり豊富ではありませんが、実際の要求に応じて整備します。これらは原則として複素数にも適用可能ですが、実数のみに限定されるものもあります。もし、複素数引き数が必要な場合は要求され次第対処します。以下で引き数が u, v, w, z, ν で表わされているのものは複素数を受け付けます。また、これらの関数はリストに対してはその要素に並列に作用します。

11.2.1 BesselI

- `BesselI[`ν`, `z`]` = $I_\nu(z)$ 。

11.2.2 BesselJ

- `BesselJ[`ν`, `z`]` = $J_\nu(z)$ 。

11.2.3 BesselK

- `BesselK[`ν`, `z`]` = $K_\nu(z)$ 。

11.2.4 BesselY

- BesselY[ν, z] = $Y_\nu(z)$ 。

11.2.5 Gamma

- Gamma[z]:

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt \ 。$$

- Gamma[a, x]:

$$\Gamma(a,x) = \int_x^\infty t^{a-1}e^{-t}dt \ 。$$

11.2.6 Factorial

- Factorial[z] は Gamma[z + 1] と同値です。

11.2.7 LogGamma

- LogGamma[z] は定義は Log[Gamma[z]] と同値ですが、計算上、Gamma[z] のオーヴァーフローを避けることができます。

11.2.8 LogGamma1

- LogGamma1[z] は定義は Log[Gamma[z + 1]] と同値ですが、計算上、Gamma[z + 1] のオーヴァーフローを避けることができます。

11.2.9 GammaRegularizedQ

- GammaRegularizedQ[a, x]:

$$Q(a,x) = \frac{\Gamma(a,x)}{\Gamma(a)} \ 。$$

11.2.10 GammaRegularized

- GammaRegularized[a, x] は GammaRegularizedQ[a, x] と同値です。

11.2.11 GammaRegularizedP

- GammaRegularizedP[a, x]:

$$P(a,x) = 1 - Q(a,x) = 1 - \frac{\Gamma(a,x)}{\Gamma(a)} \ 。$$

11.2.12 Erf

- Erf[z]:

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt \ 。$$

11.2.13 Erfc

- Erfc[z]:

$$\mathrm{erfc}(z) = 1 - \mathrm{erf}(z) \ 。$$

11.3 数値関数

SAD には以下の数値関数が備わっていいます。

- Abs Ceiling Floor Max Min Mod Round Sign。

この内、Abs, Ceiling, Floor, Round, Signはリストに対してはその要素に並列に作用します。

11.4 複素数演算

- Complex ComplexQ Conjugate Im Re。

11.4.1 Complex

- Complex[x, y] は複素数 x + y * I を表わします。ここで x、y は複素数でも構いません。
- シンボル I は Complex[0, 1] と同値です。

11.4.2 ComplexQ

- ComplexQ[式1] は式1 の値が実数でない複素数、あるいは、少なくともひとつの要素がそのような数であるリストの時 True(1) になり、それ以外は False(0) になります。

11.5 フーリエ変換

11.5.1 Fourier

- Fourier[リスト1] は複素数から成るリスト1= $\{x_1, x_2, ...\}$ の高速フーリエ変換

$$\tilde{x}_m = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} x_k \exp\left(\frac{2\pi i k m}{N}\right)$$

を求め、結果を複素数のリストにして返します。ここで N はリスト1 の長さです。N が 2の冪乗でないときには、2の冪乗になるまで後ろに 0 を補ってから変換します。

11.5.2 InverseFourier

- Fourier[リスト1] は複素数から成るリスト1= $\{x_1, x_2, ...\}$ の高速フーリエ逆変換

$$\tilde{x}_m = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} x_k \exp\left(-\frac{2\pi i k m}{N}\right)$$

を求め、結果を複素数のリストにして返します。ここで N はリスト1 の長さです。N が 2の冪乗でないときには、2の冪乗になるまで後ろに 0 を補ってから変換します。

11.6 行列演算

以下でヴェクトルは1 階の、行列は2 階のリストで表現されます。この場合行列は行ヴェクトルのリストであるとみなします。SAD には以下の行列演算の関数が備わっています。

- Det Dot Eigensystem IdentityMatrix Inner LinearSolve Outer SingularValues Transpose。

11.6.1 Dot, .

- 行列1 . 行列2 は二つの行列の積を返します。

11.6.2 Transpose

- Transpose[行列1] は行列1 の転置行列を返します。

11.6.3 LinearSolve

- LinearSolve[行列1, ヴェクトル1] は方程式 行列1 . 解 == ヴェクトル1 の singular value decomposition による解を求めます。行列1は正方形である必要はありません。
- 線形回帰は LinearSolve で簡単に求めることができます。
- LinearSolve[行列1, ヴェクトル1, Tolerance -> ε] は singular value decomposition のしきい値をεに設定して解を求めます。Tolerance のデフォルト値は 10^{-18} です。

11.6.4 SingularValues

- SingularValues[行列1] は行列1の singular value decomposition を求めます。
- SingularValues[行列1] はリスト {u, w, v} を返しますが、行列1の実効的逆行列は Transpose[v] . DiagonalMatrix[w] . u で表わされます。つまり、ヴェクトル w は singular values の実効的逆数のリストです。
- SingularValues[行列1, Tolerance -> ε] は singular value decomposition のしきい値をεに設定します。Tolerance のデフォルト値は 10^{-18} です。

ここで「実効的逆数」とは、ある singular value をw_i、最大の singular value を $w_{\max}$ とした時、$w_i/(\varepsilon^2 w_{\max}^2 + w_i^2)$ で表わされる量です。

11.6.5 Eigensystem

- Eigensystem[行列1] は正方行列行列1 に対して、{固有値達，右固有ヴェクトル達} というリストを返します。

11.6.6 Inner

- Inner[積1，リスト1，リスト2，和1] はリスト1とリスト2の一般化された内積を返します。ここで積1 と和1 はそれぞれ積と和の役割を果たす関数です。

11.6.7 Outer

- Outer[積1，リスト1，リスト2，...] はリスト1，リスト2，...の一般化された外積を返します。ここで積1 は積の役割を果たす関数です。
- リスト1，リスト2，... の頭部は List でなくても構いませんが、頭部の一致は必要です。

11.7 擬似乱数

11.7.1 Random

- Random[] は0と1の間の一様擬似乱数を返します。
- Random[n] は0と1の間のn個の一様擬似乱数達をリストにして返します。
- Random[$n1$，$n2$，...] は0と1の間の一様擬似乱数達を、各階数の要素数が$n1$，$n2$，... になるテンソルにして返します。後のインデクスほど速く動きます。

11.7.2 GaussRandom

- GaussRandom[] は平均値0、分散1の正規擬似乱数を返します。
- GaussRandom[n] は平均値0、分散1のn個の正規擬似乱数達をリストにして返します。
- GaussRandom[$n1$, $n2$, ...] は平均値0、分散1の正規擬似乱数達を、各階数の要素数が$n1$, $n2$, ... になるテンソルにして返します。後のインデクスほど速く動きます。
- 絶対値がシンボル GCUT の値を上回るものは排除されます。GCUT の初期値は 10^{35} です。

11.7.3 SeedRandom

- SeedRandom[] はその時点の擬似乱数の種の値を返します。
- SeedRandom[奇数1] は擬似乱数の種を奇数1 に設定します。奇数1 の絶対値は2^{31}を越えません。

11.8 方程式の近似解

11.8.1 FindRoot

- FindRoot[式1 == 式2, {未知数1, 初期値1}, {未知数2, 初期値2}, ..., オプション] は方程式式1 == 式2 の近似解を未知数1、未知数2, ... を未知数として求めます。未知数1, ... はシンボルで、式1 == 式2 の中にあからさまに現われなければなりません。
- 未知数1、未知数2, ... の初期値は初期値1 ... で指定されます。
- FindRoot は結果を {未知数1 -> 解1, ... } というリストで返します。
- FindRoot[x == Cos[x], {x, 0}] ⇒ {x -> .739085133215161}。
- FindRoot[{式11 == 式12, 式21 == 式22 ... }, {未知数1, 初期値1}, {未知数2, 初期値2}, ..., オプション] は連立方程式の近似解を求めます。
- 各未知数の指定を {未知数, 初期値, {下限値, 上限値}}のようにすると、未知数の検索範囲を制限することができます。
- FindRoot はニュートン法を基本に解を探します。
- 解が存在しない場合は残差の2乗和の極小値を解として返します。
- FindRoot には表 31 のオプションがあります。

表 31: FindRoot のオプション。

オプション	値	デフォルト値	効果
AccuracyGoal	数値	10^{-20}	2乗残差の両辺の2乗の最大値に対する相対精度
MaxIterations	数値	50	探査中に方程式を評価する最大回数

11.9 非線形回帰

11.9.1 Fit

- Fit[リスト1，式0，シンボル0，{パラメータ1，初期値}，... ，オプション] はリスト1 で表わされるデータ点に対して、パラメータ1，... を動かして式0 の χ^2-Fit を求めます。
- 式0 はシンボル0，パラメータ1，... を**あらわに含む式**です。Fit の二番目以降の引き数は前もって評価されないため、そのような「あらわに含む式」を得るためにここに Evaluate[式00] と書かねばならないことがあります (Evaluate 10.6.3 参照)。ここで式00 は式0 を結果としてもたらす式です。
- 各パラメータを {パラメータ，初期値，{下限値，上限値}} とするとそのパラメータの検索範囲を制限することができます。
- リスト1 は {{$x1$，$y1$，$\delta y1$}，{$x2$，$y2$，$\delta y2$}，...} という構造です。三番目の数値はエラーバーの大きさで、χ^2 の重みとなります。
- 各データ点の三番目の数値は指定する場合は全てのデータ点に対して指定しなければなりません。
- 各データ点の三番目の数値がない場合は各点の重みは均等になり、エラーバーの大きさは χ^2 が Fit の結果に一致するように決まるものとします。
- リスト1 のデータの構造は ListPlot 14.3.11 と共通です。
- FitPlot 14.8 により結果をグラフ表示できます。
- Fit は シンボル -> 値という規則のリストを結果として返します。ここでシンボルの意味は表 32 に示します。
- オプション MaxIterations で探査中の式0 の評価の回数の上限を指定できます（デフォルト： 40)。

11.10 関数の最小化

表 32: Fit の結果を表わすシンボル。ここで n、m はそれぞれデータ点、パラメータの数を表わします。

シンボル	値	意味
パラメータ1，...	実数値	各パラメータの最適値
ChiSquare	実数値	$\chi^2 = \sum_i^n \frac{(f(x_i) - y_i)^2}{\sigma_i^2}$
GoodnessOfFit	実数値	$\Gamma\left(\frac{n-m}{2}, \frac{\chi^2}{2}\right) / \Gamma\left(\frac{n-m}{2}\right)$
ConfidenceInterval	要素数 m のリスト	パラメータ1，... の推定域
CovarianceMatrix	$m \times m$ 行列	共分散行列

12　文字列の処理

12.1　部分文字列の取り出し

- 文字列1[n]　は文字列1 のn番目の文字を返します。
- nが負のときは、文字列1[n]　は文字列1 の最後から -n番目の文字を返します。
- 文字列1[$n1$, $n2$]　は文字列1 のn番目の文字から、$n2$ 番目の文字までを含む部分文字列を返します。$n1$、$n2$ 共に負でも構いません。

12.2　文字列への変換

SAD の原子や式はすべて文字列に変換することができます。一部の関数、Print, Write, StringJoin 等の引き数は自動的に文字列に変換されます。

12.2.1　ToString

- ToString[式1]　は式1を表現する文字列を返します。
- ToString[式1, FormatType -> InputForm]　は式1が文字列の時、それを入力形式に変換します。入力形式とは、それをPrint, Write などで出力したものをそのまま入力すると元の文字列になるような形式です。
- 式1に含まれる実数値は $FORM の値に従って変換されます。

12.2.2　$FORM

$FORM は実数値を文字列に変換する際のフォーマットを指定するシンボルです。この値は一度指定するとその値は保存されます。$FORM には文字列を与えます。

- $FORM = "桁数1.少数部1"　は変換の全桁数を桁数1、少数部の長さを少数部1 で指定します。もし変換する数が指定の範囲内で表現しきれない時は自動的に指数表示に切り替わります。
- $FORM = "F桁数1.少数部1"　は変換の全桁数を桁数1、少数部の長さを少数部1 で指定します。もし変換する数が指定の範囲内で表現しきれない時は桁数の長さだけ "*" が表示されます。
- $FORM = "S桁数1.少数部1"　は変換の全桁数を桁数1、少数部の長さを少数部1 で指定します。もし変換する数が指定の範囲内で表現しきれない時は自動的に指数表示に切り替わります。先導する空白は削除され、また終端部の 0 は削除されます。
- $FORM = "M桁数1.少数部1"　は変換の全桁数を桁数1、少数部の長さを少数部1 で指定します。もし変換する数が指定の範囲内で表現しきれない時は自動的に指数表示に切り替わります。先導する空白は削除され、また終端部の 0 は削除されます。指数部の表示は "E" でなく " 10^" になります。

- $FORM = "" は標準のフォーマット ("S18.15") です。

12.2.3 PageWidth

PageWidth は出力ファイルの Record の長さを指定します。その初期値は 131 または端末の表示幅 - 1 の内小さい方です。PageWidth を越える出力は自動的に改行が挿入されます。

12.2.4 StandardForm

- StandardForm[式1] は $FORM = ""; PageWidth = 2147483647 に設定してから式1 を評価し、その結果を返します。また実行後に $FORM 及び PageWidth の値を以前の値に戻します。
- StandardForm[$FORM = フォーマット1; 式1] とすれば局所的に $FORM をフォーマット1 に設定できます。

12.2.5 SymbolName

- SymbolName[シンボル1] はシンボル1 の文字列表現を返します。これは ToString[シンボル1] と同値です。

12.3 文字列の結合

12.3.1 StringJoin, //

- 文字列1 // 文字列2 // ... は文字列1, 文字列2, .. をこの順に結合した文字列を返します。
- 式1 // 式2 // ... は式1, 式2, .. をこの順に文字列に変換した結果の文字列を結合した文字列を返します。

12.4 文字列の比較

12.4.1 Equal, ==

- 文字列1 == 文字列2 は文字列1 と文字列2 が等しいとき True(1)、等しくないとき False(0) を返します。両辺の型が異なるときはこの式そのものを返します。

12.4.2 Unequal, <>

- 文字列1 <> 文字列2 は文字列1 と文字列2 が異なるとき True(1)、等しいとき False(0) を返します。両辺の型が異なるときはこの式そのものを返します。

12.5 文字列の照合

StringMatchQ などいくつかの関数ではワイルドカードによる文字列の照合が可能です。これはあくまでも文字列としての照合であり、すでに述べたパターンによる照合とは全く別のものです。

12.5.1 ワイルドカード

- "*" はゼロまたはそれ以上の任意の長さの任意の文字列に照合します。
- "%" は任意の1文字に照合します。
- "{...}" はそれに囲まれた文字達の内に含まれる任意の1文字に照合します。
- "{^...}" はそれに囲まれた文字達の内に含まれない任意の1文字に照合します。
- "...|...|..." は"|"で区切られた部分のうちの少なくともひとつに照合します。

12.5.2 StringMatchQ

- StringMatchQ[文字列1，文字列パターン1] は文字列1 が文字列パターン1 に照合するとき True(1)、照合しないとき False(0) を返します。
- StringMatchQ["aabcdee", "a*e"] ⇒ True。

12.6 文字列の演算

12.6.1 StringLength

- StringLength[文字列1] は文字列1 の文字数を返します。

12.6.2 StringPosition

- StringPosition[文字列0，文字列1] は文字列0 に含まれる部分文字列文字列1の位置達を{{始点1，終点1}，{始点2，終点2}} の様にリストにして返します。StringPosition["abccddcce","cc"] ⇒ {{3，4}，{7，8}}。
- 位置達は互いに重なることがあります。
- StringPosition[文字列0，文字列1，n] は最初のn 個の位置達だけを返します。
- StringPosition[文字列0，{文字列1，文字列2，...}] は文字列1，文字列2，...の全ての位置達を返します。

12.6.3 StringInsert

- StringInsert[文字列0，文字列1，n] は文字列0 の位置n（負でもよい)に文字列1 を挿入した文字列を返します。

12.6.4 StringDrop

- StringDrop[文字列1, n] は文字列1 の最初のn 文字を除いた文字列を返します。
- StringDrop[文字列1, -n] は文字列1 の最後のn 文字を除いた文字列を返します。
- StringDrop[文字列1, {n}] は文字列1 のn 番目の文字を除いた文字列を返します。
- StringDrop[文字列1, {$n1$, $n2$}] は文字列1 の $n1$ から $n2$ までの文字を除いた文字列を返します。

12.6.5 StringFill

- StringFill[文字列1, 文字列2, n] は文字列1 の後ろに文字列2 を繰り返し補って、長さ n の文字列を作ります。
- 文字列1 が n よりも短いときは n 文字までに切り捨てられます。
- StringFill["abc", "def", 10] ⇒ "abcdefdefd"。

12.6.6 ToCharacterCode

- ToCharacterCode[文字列1] は文字列1のそれぞれの文字の ASCII コードをリストにして返します。
- ToCharacterCode["Hello!"] ⇒ {72, 101, 108, 108, 111, 33}。

12.6.7 FromCharacterCode

- FromCharacterCode[リスト1] は ASCII コードから成るリスト1 のそれぞれに対応する文字から成る文字列を返します。
- FromCharacterCode[{72, 101, 108, 108, 111, 33}] ⇒ "Hello!"。

12.6.8 Characters

- Characters[文字列1] は文字列1に含まれる各文字から成るリストを返します。
- Characters["A string."]
  ⇒ {"A", " ", "s", "t", "r", "i", "n", "g", "."}

12.6.9 ToUpperCase

- ToUpperCase[文字列1] は文字列1 に含まれる英小文字を全て英大文字に変換した文字列を返します。

12.6.10 ToLowerCase

- ToLowerCase[文字列1] は文字列1 に含まれる英大文字を全て英小文字に変換した文字列を返します。

12.7 文字列の式としての評価

12.7.1 ToExpression

- ToExpression[文字列1] は文字列1が表現する任意の式を評価し、その結果を返します。
- ToExpression["Sqrt[2I]"] ⇒ 1 + I。

12.7.2 Symbol

- Symbol[文字列1] は文字列1をその名前とするシンボルを作り、それを評価します。

13　入出力

SAD ではファイル入出力は原則としてチャネルを通して行われます。チャネルはOpenRead, OpenWrite, OpenAppend などによりファイルと結び付けられています。また、$Input、$Output などにより、そのときの入出力の流れをチャネルとして使用することもできます。このほかにパイプもチャネルとして利用できます。

13.1　ファイル入力

基本的なファイル入力は次のように行われます。

```
a = OpenRead["test.dat"];    (ア)
...
x = Read[a, Real];           (イ)
...
Close[a];                    (ウ)
```

まず、(ア) のように、入力したいファイル名(文字列)を OpenRead に入れることにより、ある入力チャネルを開きます。そのチャネル番号はシンボル a に割り当てられます。次に(イ)のように、チャネル a を用いて必要な入力動作を繰り返します。作業が終了したら、(ウ)のように、Close でそのチャネルを解放します。

入力ファイルはいくつかの Record から成り立っているとみなします。Record は改行文字 ("\n")で区切られた任意の長さの文字列です。改行文字自身は Record には含まれません。また、Record の区切りの文字を変更することもできません。

13.1.1　OpenRead

- OpenRead[ファイル名1]　はファイル名1（文字列）で表わされるファイルに対して入力チャネルを開き、そのチャネル番号を返します。
- OpenRead["!コマンド1"]　はシステムのコマンドであるコマンド1 を実行し、その出力結果に対して入力チャネルを開き、そのチャネル番号を返します。例:　OpenRead["!ls -l"]。
- 何らかのエラーが発生した場合は OpenRead は番号ではなく、シンボル $Failed を返します。

13.1.2　Read

関数 Read は

```
Read[チャネル1, 対象1, オプション1, ...]
```

のように用います。ここでチャネル1 は OpenRead で開いたチャネル番号、または $Input です。$Input はその時点の入力の流れを表わします。

Read の二番目の引き数は読み込まれる対象の指定をします。それぞれの対象には三番目以降の引き数で指定可能なオプションがあります。オプションはシンボル -> 値の様な規則で表わします。オプションの作用は表 33 に示します。オプション省略時は表のデフォルト値に設定されます。

Read の二番目の引き数に対象達のリストを指定すると、Read はそのリストにしたがってファイルから次々と対応する対象達を読み込み、結果を指定と同型のリストにして返します。また、この対象リストの階数は任意です。

Read に際してファイルが終端に達した場合、Read はシンボル EndOfFile を返します。

- Read[チャネル1] はチャネル1 から式を読み込み、それを評価した結果を返します。この場合、ひとつの Record で式が完結しない場合、完結するまで読み進みます。
- Read[チャネル1, Word, オプション1, ..] はチャネル1 から 1語を読み込み、それを返します。
- Read[チャネル1, Real, オプション1, ..] はチャネル1 から 1語を読み込み、それを式として評価した値を返します。
- Read[チャネル1, Expression, オプション1, ..] はチャネル1 から 1語を読み込み、それを式として評価した結果を返します。
- Read[チャネル1, Character, オプション1, ..] はチャネル1 から1文字を読み込み、それを返します。
- Read[チャネル1, String, オプション1, ..] はチャネル1 からその Record の終端までを読み込み、それを文字列として返します。

表 33: Read のオプション。

オプション	デフォルト値	作用
WordSeparators	" ,\t"	語の区切りを表わす文字達
NullWords	False	Trueの時, 語の区切り字が連続したところに語 "" があるとみなす。
ReadNewRecord	True	False の時、次の Record を読まず、語 "" を返す。

13.1.3 Skip

- Skip[チャネル1, 対象1, n, オプション1, ...] は Read[チャネル1, 対象1, オプション1, ...]を n回繰り返し、その最後の結果を返します。
- Skip[チャネル1, 対象1] は Read[チャネル1, 対象1]と同値です。

13.1.4 Close

- Close[チャネル1] はチャネル1 を解放します。

13.1.5 Get

- Get[ファイル名1] はファイル名1（文字列）に書かれた式達を次々に評価し、その最後の結果を返します。
- Get に対しては OpenRead, Close は必要ありません。
- 関数ライブラリのロードには Get を使います。
- AutoLoad（10.7.6）を使えば、必要な場合だけ関数をロードすることができます。

13.2 ファイル出力

基本的なファイル出力は次のように行われます。

```
a = OpenWrite["test.dat"];    (ア)
...
Write[a, x, y, ...];           (イ)
...
Close[a];                     (ウ)
```

まず、(ア)のように、出力したいファイル名(文字列)を OpenWrite に入れることにより、ある出力チャネルを開きます。そのチャネル番号はシンボル a に割り当てられます。次に(イ)のように、チャネル a を用いて必要な出力動作を繰り返します。作業が終了したら、(ウ)のように、Close でそのチャネルを解放します。

出力チャネルは OpenWrite の他に、OpenAppend でも開くことができます。この場合ファイルはその終端から書き込まれます。

出力ファイルの Record の長さはシンボル PageWidth で決まります。PageWidth の初期値は 131 または端末の表示幅 - 1 の小さいほうです。PageWidth を越える出力は自動的に改行されます。

13.2.1 OpenWrite

- OpenWrite[ファイル名1] はファイル名1（文字列）で表わされるファイルに対して出力チャネルを開き、そのチャネル番号を返します。ファイルはその先頭から書き込まれます。
- 何らかのエラーが発生した場合は OpenWrite は番号ではなく、シンボル $Failed を返します。

13.2.2 OpenAppend

- OpenAppend[ファイル名1] はファイル名1（文字列）で表わされるファイルに対して出力チャネルを開き、そのチャネル番号を返します。ファイルはその終端から書き込まれます。
- 何らかのエラーが発生した場合は OpenWrite は番号ではなく、シンボル $Failed を返します。

13.2.3 Write

- Write[チャネル1, 式1, 式2, ...] はチャネル1 に対して、式1 を評価し、その結果を文字列に変換した結果を出力します。この動作を式2 以降にも行い、最後に改行文字を出力します。
- 式と式の間には区切り字はなにも入りません。
- 数値の文字列への変換は $FORM の値に従います。途中で $FORM を変更したいときには Write[..., $FORM = フォーマット1; 式1, ...] のようにすれば式1 以降の $FORM がフォーマット1 に変更されます。元のフォーマットを保存したいときには StandardForm などを利用します。
- チャネル1としては OpenWrite, OpenAppend で開いたチャネル番号、または $Output が指定できます。$Output はその時点の出力の流れを表わします。

13.2.4 Print

- Print[式1, 式2, ...] はその時点での出力の流れ $Output に対して、式1 を評価し、その結果を文字列に変換した結果を出力します。この動作を式2 以降にも行い、最後に改行文字を出力します。
- 式と式の間には区切り字はなにも入りません。
- 数値の文字列への変換は $FORM の値に従います。途中で $FORM を変更したいときには Print[..., $FORM = フォーマット1; 式1, ...] のようにすれば式1 以降の $FORM がフォーマット1 に変更されます。元のフォーマットを保存したいときには StandardForm などを利用します。

13.2.5 WriteString

- WriteString[チャネル1, 式1, 式2, ...] はチャネル1 に対して、式1 を評価し、その結果を文字列に変換した結果を出力します。この動作を式2 以降にも行います。最後に改行文字は出力しません。
- 式と式の間には区切り字はなにも入りません。
- 数値の文字列への変換は $FORM の値に従います。途中で $FORM を変更したいときには WriteString[..., $FORM = フォーマット1; 式1, ...] のようにすれば式1 以降の $FORM がフォーマット1 に変更されます。元のフォーマットを保存したいときには StandardForm などを利用します。
- チャネル1としては OpenWrite, OpenAppend で開いたチャネル番号、または $Output が指定できます。$Output はその時点の出力の流れを表わします。

13.2.6 Close

- Close[チャネル1] はチャネル1 を解放します。

14　グラフィックス

14.1　グラフィックスの例

次の例は、SAD/Tkinter による最も簡単なグラフィックスの例題です。

```
FFS;

  w = Window[];                                          (ア)
  c = Canvas[w, Height -> 400, Width -> 600];            (イ)
  $DisplayFunction = CanvasDrawer;                       (ウ)
  Canvas$Widget = c;                                     (エ)

  data = Table[{x, Sin[x] + 0.1 * GaussRandom[]},
    {x, -Pi, Pi, Pi/10}];                                (オ)
  ListPlot[data];                                        (カ)
  TkWait[];                                              (キ)
```

この例は次の図のようにWindow w（の中の Canvas c）にグラフを表示します。

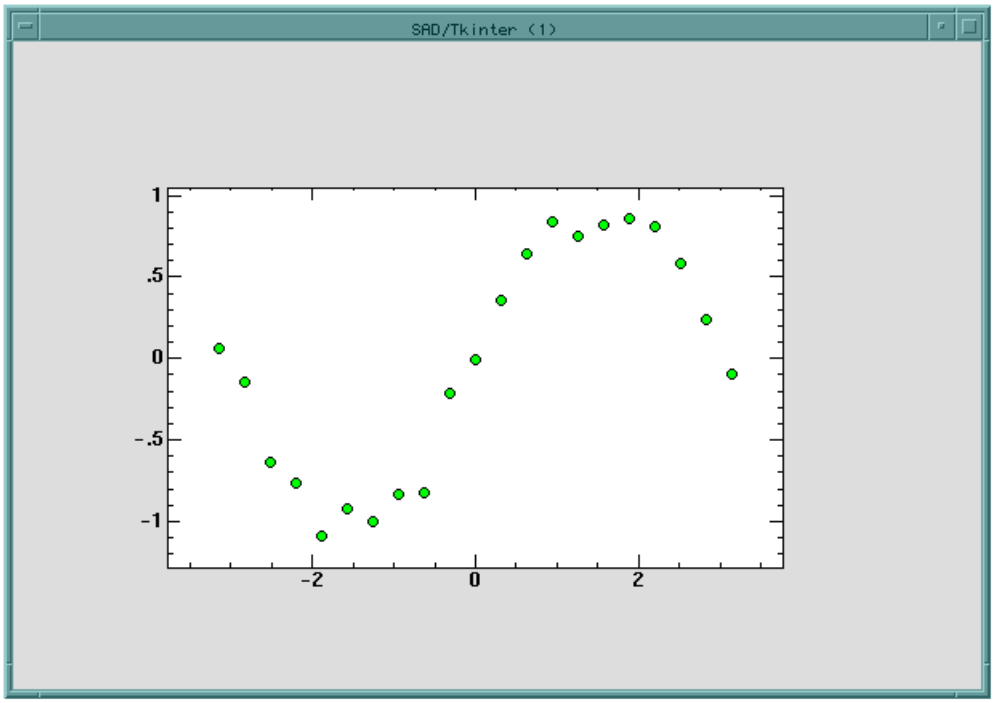

図 22: ListPlot による簡単なグラフの例。

上の例で、まず、(ア)、(キ) については通常の Tkinter の操作ですので、それぞれ 2.2、2.5 を参照してください。

（イ）は Tkinter の部品 Canvas を Window w の中につくります。またそのサイズを幅 600 ピクセル、高さ 400 ピクセルに指定しています。

（ウ）は以下で用いられるグラフィックスの出力関数を指定しています（14.2 参照)。

（エ）はグラフィックスの出力先になる Canvas部品 c を指定しています。

（オ）はこれから表示するグラフのためのデータを作っています。データは $\{\{x1, y1\}, \{x2, y2\}, \ldots\}$ のような形をしています。この例は sine 曲線に正規擬似乱数を乗せたものです。Table 7.2.1、GaussRandom 11.7.2 参照。

（カ）は（オ）で作った data からグラフを出力します。SAD ではグラフィックスは ListPlot、Plot、FitPlot 等により生成されるオブジェクト、頭部 Graphics を持った式で表わされます。このオブジェクトは ListPlot、Plot、FitPlot や Show により出力されます。これらの関数は生成した Graphics オブジェクトを結果として返します。

この例のように、グラフの位置、範囲等々はなにも指定しなくても自動的に調節されますが、オプションを指定することにより自由に変更することができます。

グラフィックスの出力関数として TopDrawer を選ぶ（デフォルト）場合は、(ア)--(エ)、及び（キ）の操作は不要です。

14.2 グラフィックスの出力関数

出力関数はオプション DisplayFunction で指定されます。DisplayFunction のデフォルト値は $DisplayFunction であり、$DisplayFunction には前もって TopDrawer が割り当てられています。

出力関数はこのほかに前の例にあるように、CanvasDrawer が選べます。CanvasDrawer は SAD/Tkinter の部品のひとつである Canvas に出力を行います。

出力されるグラフィックスは TopDrawer、CanvasDrawer でできるだけ同じになるように調整されていますが、機能の違いにより、また実装がおくれているために不完全な部分が多々あります。必要に応じて整備していくつもりです。

TopDrawer、CanvasDrawer 以外の出力関数が指定される（例えば DisplayFunction -> Identity）と、グラフィックスはどこにも出力されません。これは ListPlot などに Graphics オブジェクトだけを生成させるために用いられます。

14.3 グラフの作成

SAD には現在、表 34 にあるグラフ作成のための関数が備わっています。

いずれの関数も引き数としてオプションが指定できます。これらのオプションの多くはいずれの関数にも共通です。オプションはオプションシンボル -> 値という形をとります。表 35 にこれらのオプションを掲げます。また、これらの他にもそれぞれの関数に固有のオプションがありますが、それらは各関数の項で説明します。

以下、ListPlot を例にとってそれらのオプションを説明します。

表 34: グラフ作成のための関数。

関数	作成するグラフ
ListPlot	数値データのプロット
Plot	関数のグラフのプロット
ColumnPlot	棒グラフ
OpticsPlot	ビームラインに沿ったデータ、ビーム光学系のプロット
FitPlot	非線形回帰のグラフのプロット

14.3.1 プロットするデータの範囲、PlotRange

プロットするデータの範囲は通常データ自身から自動的に設定されますが、オプション PlotRange により調節可能です。シンボル Automatic は自動設定値を用いることを指示します。

- PlotRange -> {ymin, ymax} は y 軸の範囲を下限 ymin と上限 ymax に指定します。
- PlotRange -> {ymin, Automatic} は y 軸の範囲を下限 ymin に、上限は自動設定値に指定します。
- PlotRange -> {{xmin, xmax}, {ymin, ymax}} は x 軸の範囲を下限 xmin と上限 xmax に指定し、y 軸の範囲を下限 ymin と上限 ymax に指定します。
- PlotRange -> {{xmin, xmax}, Automatic} は x 軸の範囲を下限 xmin と上限 xmax に指定し、y 軸の範囲を自動設定値に指定します。
- 図 22 の例で ListPlot[data, PlotRange -> {{-10, 10}, {-2, 3}}] とすると図 23の様になります。

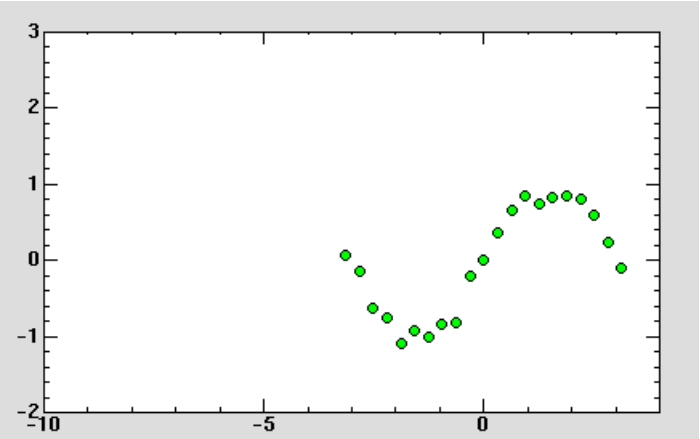

図 23: PlotRange による表示するデータの範囲の指定。

表 35: グラフ作成関数のオプション。

オプション	値	デフォルト値	効果
AspectRatio	実数値	GoldenRatio	横/縦比
DisplayFunction	出力関数	TopDrawer	グラフィック出力関数
Epilog	グラフィックス原子のリスト	{}	グラフ出力後に描く図形
ErrorBarTickSize	実数値	1	エラーバーの鍵の相対長
Frame	True, False	True	グラフの外枠の表示
FrameLabel	{下,左,上,右}	{"","","",""}	座標軸のラベル文字列
Initialize	True, False	True	座標系等の初期化の有無
Plot	True, False	ListPlot: True Plot: False	データ点のマーカ表示
PlotColor	色	"black"	グラフの線の色
PlotJoined	True, False	ListPlot: False Plot: True	データ点を線で結ぶか否か
PlotLabel	文字列	""	グラフ全体につけるラベル
PlotRange	y軸の範囲または {x範囲, y範囲}	{Automatic, Automatic}	プロットするデータの領域
PlotRegion	{{xmin, xmax}, {ymin, ymax}}	{{0,1},{0,1}}	プロットの相対位置
PointSize	実数値	1	マーカの大きさの相対比
PointColor	色	"green"	マーカの内部色
Prolog	グラフィックス原子のリスト	{}	グラフ出力前に描く図形
Scale	{xscale, yscale}	{Linear, Linear}	Linear または Log を選ぶ
Tags	True False	False	Canvas$ID にアイテム番号を記録
Background	色	"#ffffd0"	グラフの枠内の色

14.3.2 プロットの Canvas の中での位置の指定、PlotRegion

プロットは通常指定された Canvas 画面から自動的にその位置及び大きさが決まりますが、PlotRegion により自由な位置、大きさに変更できます。

- 自動的に決まる位置の範囲を{{0, 1}, {0, 1}}として、PlotRegion -> {{x下限, x上限}, {y下限, y上限}} のように自動値に対する相対値で指定します。
- 図 22 の例で ListPlot[data, PlotRegion -> {{0, 0.5}, {0.5, 1}}] とすると図 24 の様になります。

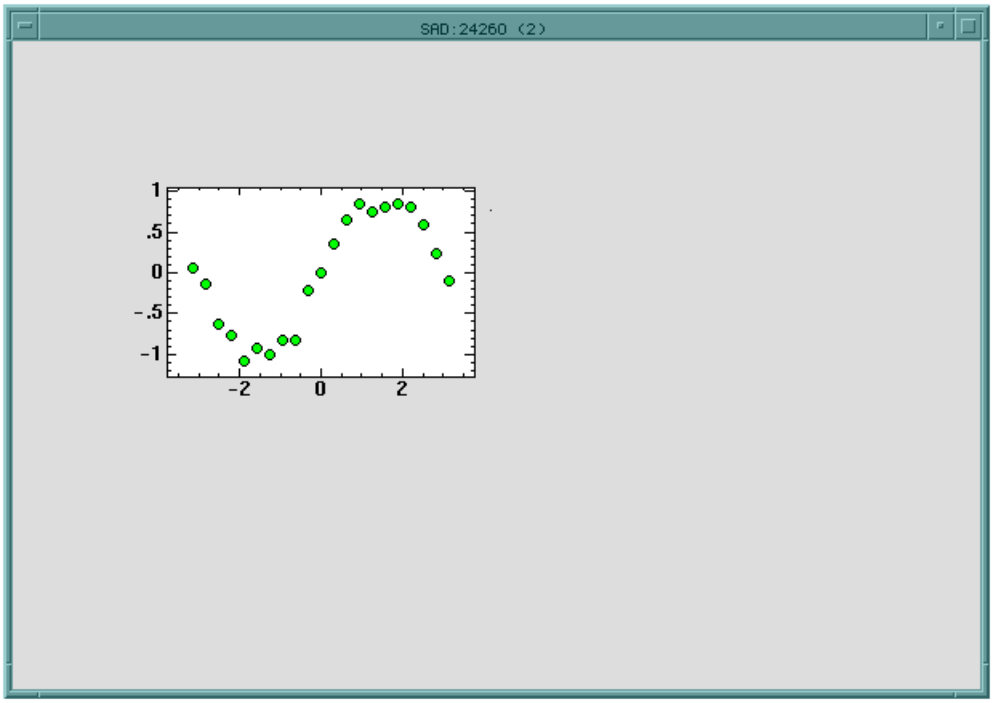

図 24: PlotRegion による表示位置の指定。

14.3.3 横/縦比、AspectRatio

- AspectRatio -> 数値 により、プロットの横/縦比を指定できます。
- 図 22 の例で ListPlot[data, AspectRatio->1] とすると図 25 の様になります。

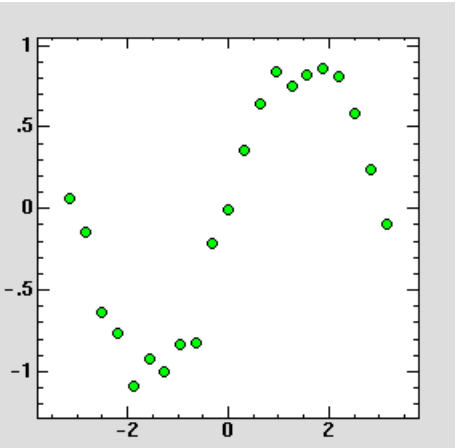

図 25: AspectRatio によるプロットの横/縦比の指定。

14.3.4 グラフの外枠及び目盛の有無、Frame

14.3.5 グラフの外枠につけるラベル、FrameLabel

- FrameLabel -> {下ラベル文字列、左ラベル文字列} により、各々の軸にラベルをつけられます。
- 図 22 の例で ListPlot[data, FrameLabel -> {"Bottom", "Left"}] とすると図 26 の様になります。

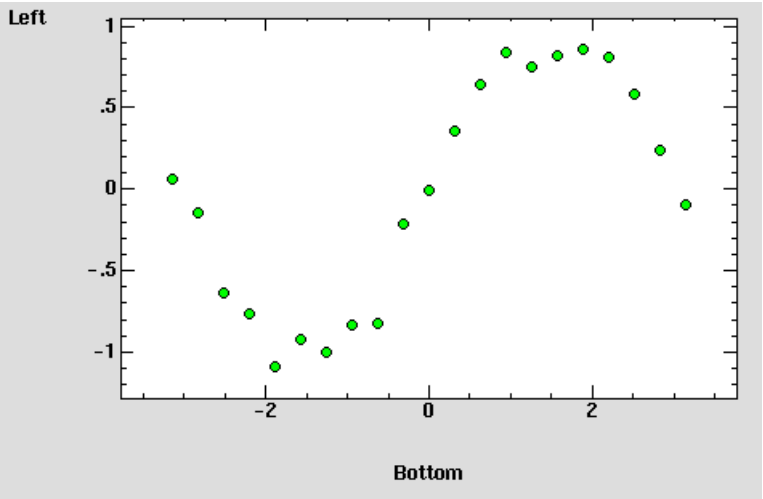

図 26: FrameLabel による枠のラベルの指定。

14.3.6 グラフ全体のラベル、PlotLabel

- PlotLabel -> 文字列により、プロット全体にラベルをつけられます。
- 図 22 の例で ListPlot[data, PlotLabel -> "PlotLabel"] とすると図 27の様になります。

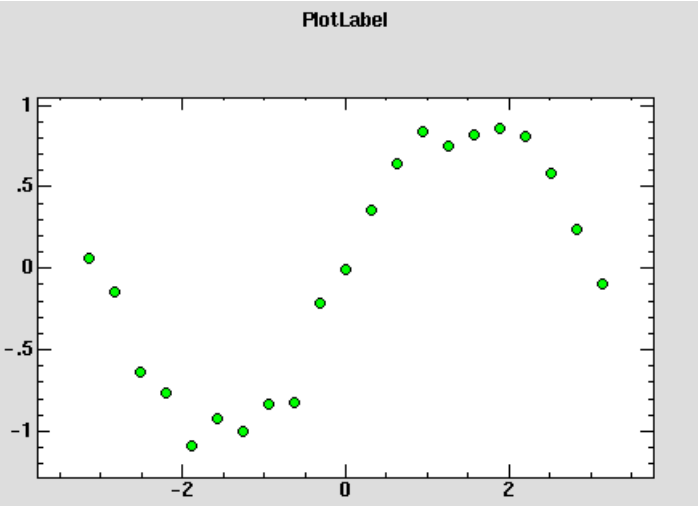

図 27: PlotLabel による全体のラベルの指定。

14.3.7 プロットの前後での図形の書き込み、Prolog と Epilog

- Prolog -> グラフィックス原子のリスト により、プロットの前に別の図形達を書き込むことができます。
- Epilog -> グラフィックス原子のリスト により、プロットの後に別の図形達を書き込むことができます。
- 図 22 の例で ListPlot[data, Prolog -> {Rectangle[{-1, -0.5}, {1, 0.5}, FillColor -> "gray"]}] とすると図 28の様になります (Rectangle は後述 (14.9) のグラフィックス原子のひとつで、長方形を描きます)。

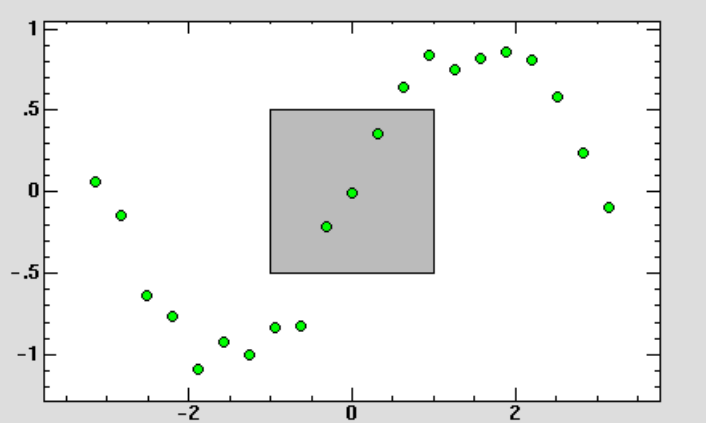

図 28: Prolog でプロットの前に図形を書き込める。

14.3.8 データ点を線で結ぶ、PlotJoined 及びデータ点の表示、Plot

- PlotJoined -> True により、データ点を線で結ぶことができます。
- 結ぶ線の色は PlotColor で指定できます。また、データ点のマーカの表示は Plot で入切できます。
- 図 22 の例で ListPlot[data, PlotJoined -> True] とすると図 29の様になります。
- 図 22 の例で ListPlot[data, PlotJoined -> True, Plot -> False] とすると図 30 の様になります。

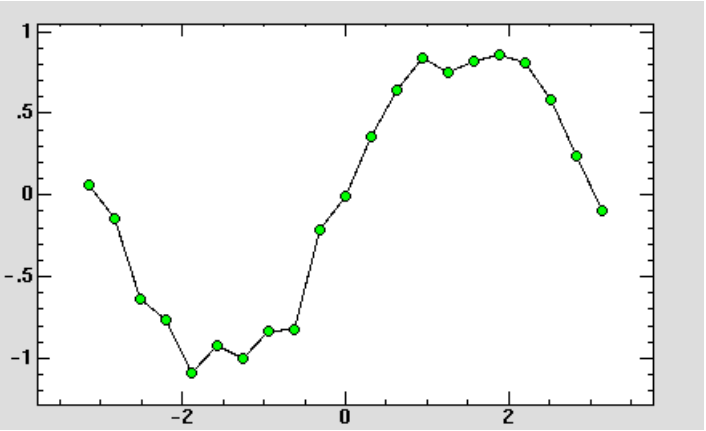

図 29: PlotJoined -> True でデータ点を線で結ぶ。

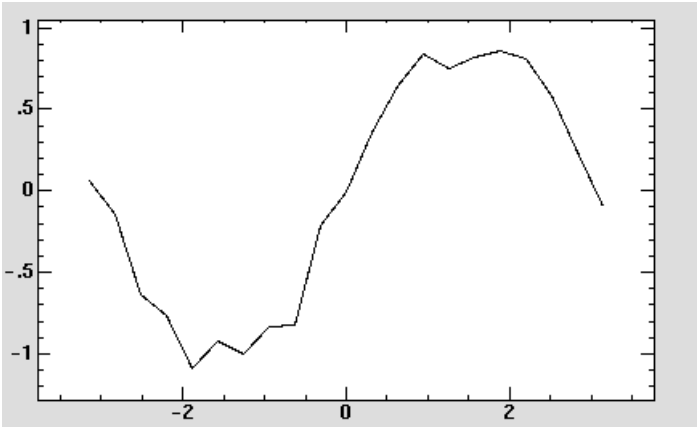

図 30: Plot -> False でデータ点のマーカを消せる。

14.3.9 マーカの大きさと色、PointSize、PointColor

- PointSize -> 数値 でマーカの大きさを指定することができます。この場合標準値を 1 とし、そこからの倍率で大きさを表わします。
- PointColor -> 色 でマーカの内部の色を指定できます。色の指定は 4.5.3 を見てください。
- 図 22 の例で ListPlot[data, PointSize -> 3, PointColor -> "White"] とすると図 31 の様になります。

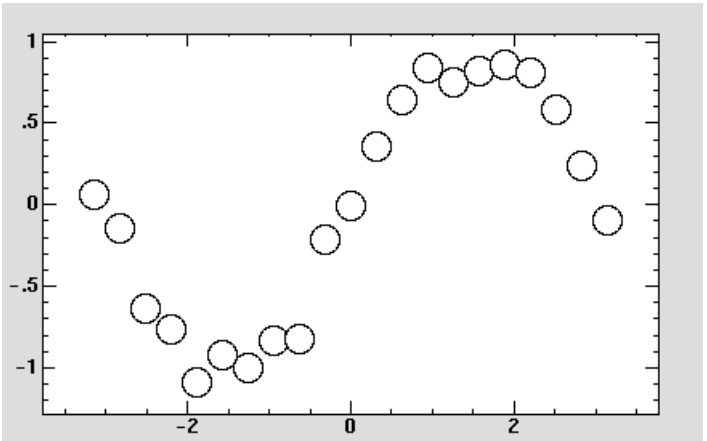

図 31: PointSize、PointColor によるデータ点のマーカの大きさと内部色の指定。

14.3.10 対数グラフ、Scale

- Scale -> {スケールx, スケールy} で x および y 軸のスケールをそれぞれ Linear または Log に指定できます。
- Scale -> スケールy は Scale -> Linear, スケールy に同値です。
- 次の例は図 32 を作ります。

```
x = Range[0, 5, 0.5];
data=Thread[{x, Exp[-x^2 / 2] / Sqrt[2 Pi]}];
ListPlot[data, Scale -> {Linear, Log}];
```

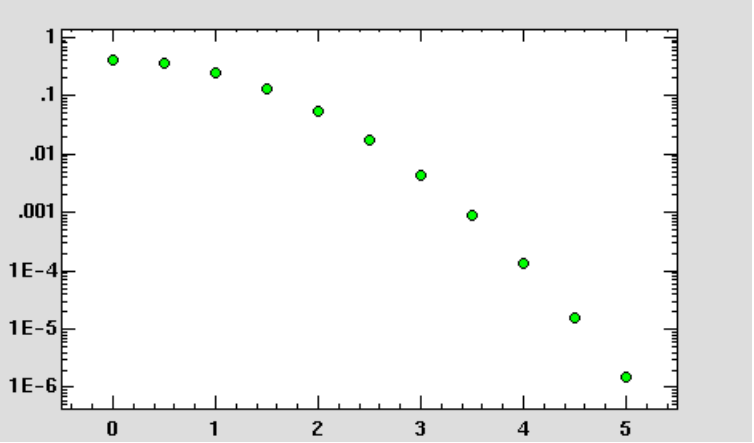

図 32: Scale -> {Linear, Log} (Scale -> Log でもよい) による片対数グラフ。

14.3.11 ListPlot、及びエラーバーの表示

ListPlot[リスト1, オプション1, ...] は上述のように数値データのリスト、リスト1 からグラフを作ります。

- リスト1 は {{$x1$, $y1$}, {$x2$, $y2$}, ...} の形で各データ点を表わします。
- リスト1 が 1 次元のデータ {$y1$, $y2$, ...} のときは x 軸は 1, 2, ... が対応するものとみなします。
- リスト1 が {{$x1$, $y1$、$\delta y1$}, ...} の様に各々が 3 個の数値から成り立っている場合は、3 番目の数値は y 方向のエラーバーの長さを表わします。リスト1 が {{$x1$, $y1$、$\delta x1$, $\delta y1$}, ...} の様に各々が 4 個の数値から成り立っている場合は、3 番目の数値は x 方向、4 番目の数値はy 方向のそれぞれのエラーバーの長さを表わします。
- エラーバーの鍵の長さはオプション ErrorBarTickSize で指定できます。
- いずれの場合も、リスト1 は長方形、即ち各要素が同じ形をしていなければなりません。

- 次の例は図 22 の例に加えて、各点 y 方向 0.1 のエラーバーを付加し、ListPlotで表示したものです。結果は図 33 になります。

```
data1 = Append[#, 0.1]& /@ data;
ListPlot[data1];
```

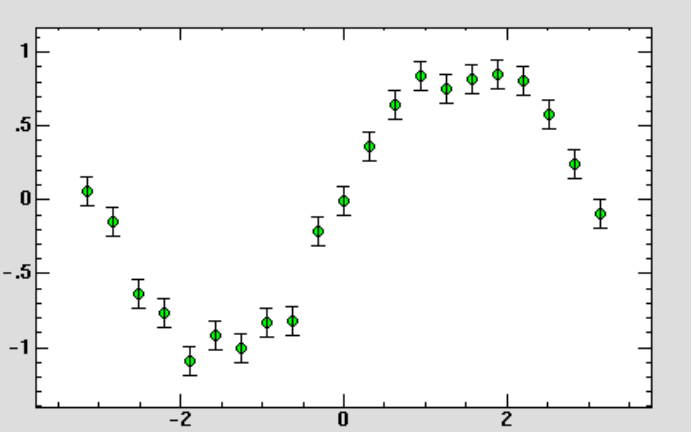

図 33: ListPlot によるエラーバーの表示。

- 次の例は図 22 の例に加えて、各点 x 方向 0.2、y 方向 0.1 のエラーバーを付加し、ListPlotで表示したものです。この例ではさらに鍵の長さを 0 にしています。結果は図 34 になります。

```
data2 = Join[#, {0.2, 0.1}]& /@ data;
ListPlot[data1, ErrorBarTickSize -> 0];
```

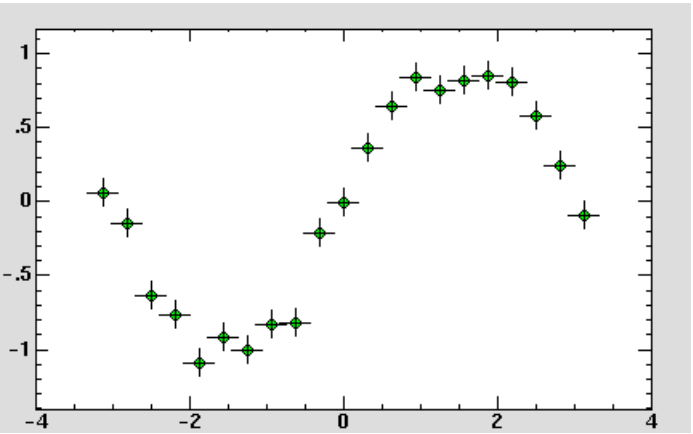

図 34: ListPlot による両方向のエラーバーの表示。ErrorBarTickSize -> 0 としています。

14.4 Plot

Plot は 1変数の関数のプロットを作ります。例えば、単純に次のようにすると図 35 のようなプロットになります。

```
Plot[Sin[x], {x, -Pi, Pi}];
```

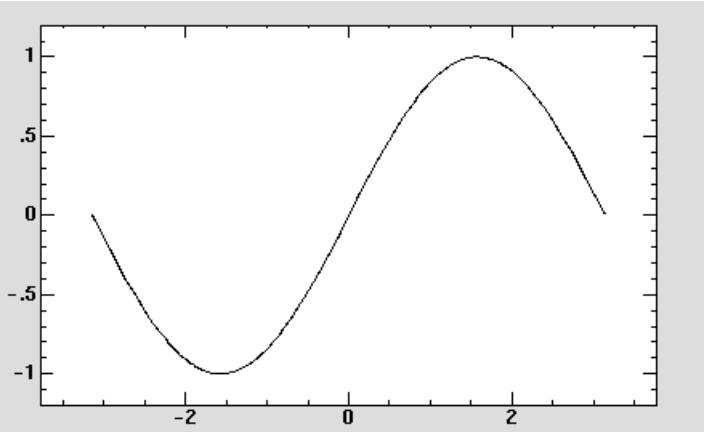

図 35: Plot による関数 Sin[x] の表示。

また、

```
Plot[{Sin[x], Cos[x]}, {x, -Pi, Pi}];
```

のように第一引き数を複数の式のリストとすると図 36 のようなプロットになります。この場合、各式の表示色は {"black", "red", "blue", "green", "grey", "magenta", "cyan"} の順に循環的に選択されますが、オプション PlotColor -> 色リストを指定することにより自由に設定することができます。

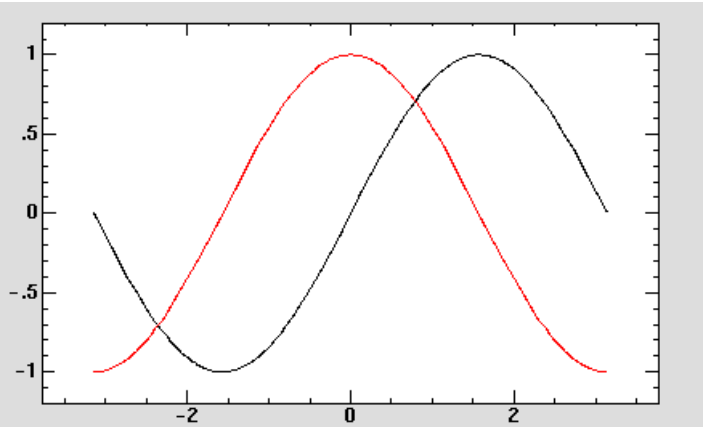

図 36: Plot による複数の関数 Sin[x], Cos[x] の同時表示。Cos[x] の表示色は "red" です。

Plot の構文は以下の通りです。

- Plot[式1，{シンボル1，下限値，上限値}，オプション，...] または
  Plot[{式1，式2，...}，{シンボル1，下限値，上限値}，オプション，...]。
- ここで式1，式2，... は「独立変数」シンボル1 をあらわに含む式です。

14.4.1 Plot のオプション

Plot には表 35 のオプションに加えて、表 36 にある Plot に固有のオプションも指定できます。

表 36: Plot に固有のオプション。

オプション	値	デフォルト値	効果
MaxBend	実数値	0.04	各線分の曲がり角の最大値
PlotDivision	実数値	250	表示区間の最大分割数
PlotPoints	実数値	25	分割点の点数の初期値
PlotColor	色または色リスト	{"black", "red", "blue", "green", "gray", "magenta", "cyan", "yellow"}	プロット線達の表示色

14.5 ColumnPlot による棒グラフの作成

- ColumnPlot[データ1，オプション，...] は棒グラフを作成します。
- データ1 が 1 次元のリストの場合、単純な棒グラフを作成します。
- ColumnPlot[1，4，9，16，25，16，9，4，1] ⇒ 図 37。

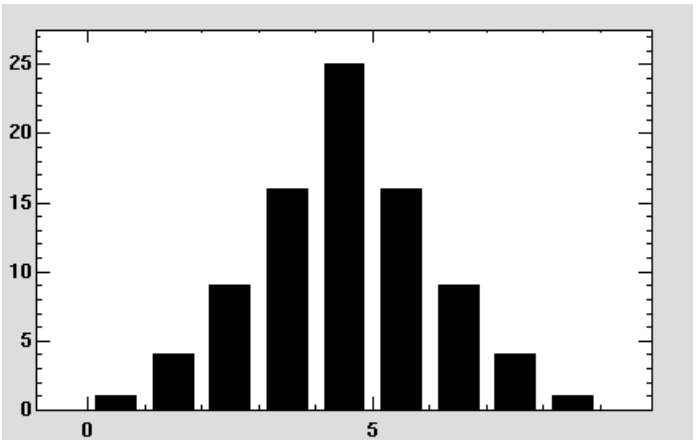

図 37: 1次元のリストによる ColumnPlot。

- データ1 が 2 次元のリストの場合、一つの枠に複数の棒をまとめた棒グラフを作成します。この場合、データ1 は長方形でなければなりません。
- `ColumnPlot[{{1, 3}, {4, 5}, {9, 7}, {16, 9}, {25, 11}}]` ⇒ 図 38。
- 各棒はオプション `FillColor` の値に従って色分けされます。

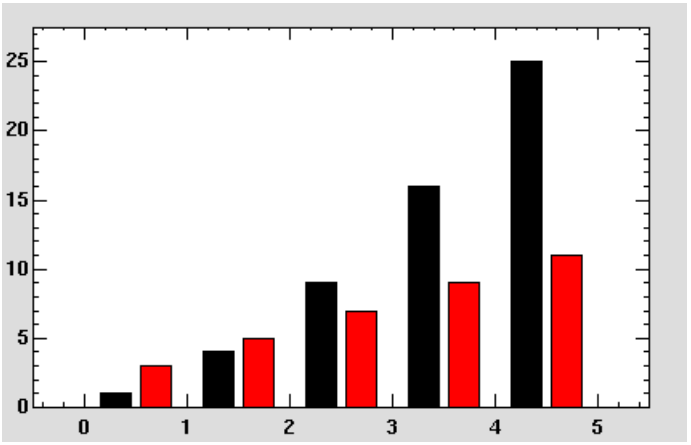

図 38: 2次元のリストによる `ColumnPlot`。

- データ1 が 3 次元の場合、第3階のデータは累積され棒を積み上げて表示されます。
- `ColumnPlot[{{{1, 5, 9}, {2, 5, 7}}, {{3, 3, 3}, {4, 3, 2}}, {{9, 2, 1}, {6, 5, 4}}}]` ⇒ 図 39。
- 積み上げはオプション `MeshStyle` の値に従って、充填パターンを変えて表示されます。

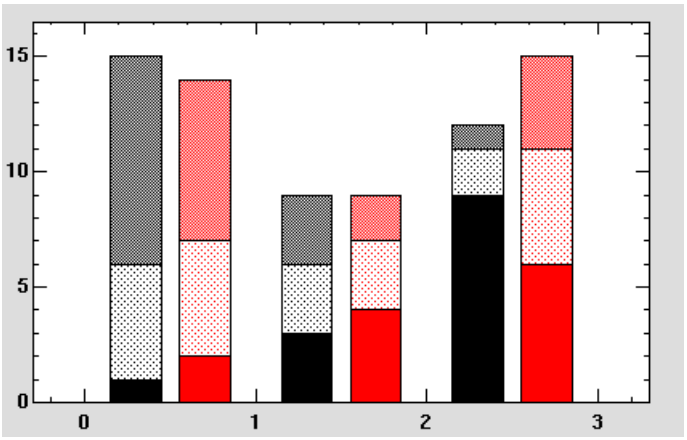

図 39: 3次元のリストによる `ColumnPlot`。

14.5.1 ColumnPlot のオプション

ColumnPlot には表 35 のオプションに加えて、表 37 にある ColumnPlot に固有のオプションも指定できます。

表 37: ColumnPlot に固有のオプション。

オプション	値	デフォルト値	効果
ColumnOffset	数値	0.15	棒と枠のすき間の比率
FillColor	色または色リスト	{"black", "red", "blue", "green", "gray", "magenta", "cyan", "yellow"}	棒達の表示色
MeshStyle	ビットマップまたはそのリスト	{Null, "gray25", "gray50"}	積み上げの表示のための充填パターン。Nullは 100% 塗る。
Orientation	Horozontal, Vertical	Vertical	棒の方向

14.6 グラフの合成, Show

ListPlot や Plot で作成したグラフは合成してひとつのグラフとして表示することができます。次の例

```
g1 = ListPlot[data, DisplayFunction -> Identity];
g2 = Plot[Sin[x], {x, 0, 2Pi}, DisplayFunction -> Identity];
Show[g1, g2];
```

は図 40 のようなグラフを作ります。 ここで、ListPlot 及び Plot には DisplayFunction ->

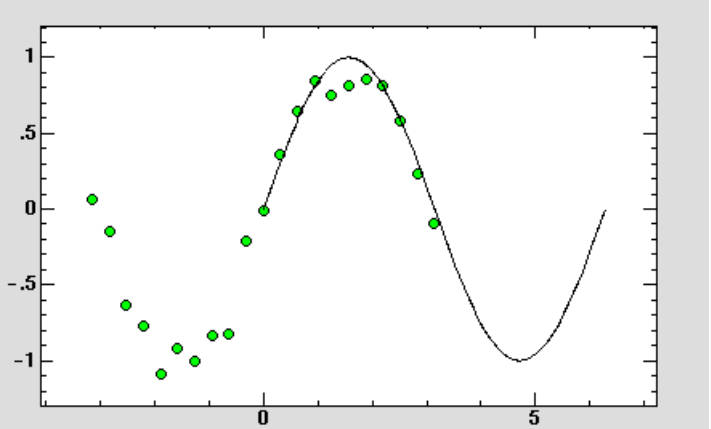

図 40: 複数のグラフの合成。

Identity というオプションをつけておきます。これは、各々の関数の段階ではプロットを実行せず、結果のグラフィックス・オブジェクトをそれぞれシンボル g1, g2 に割り当てるだけにする為です。そして、最後の Show ではじめてプロットを実行します。図 40 で軸の範囲が g1, g2 の両方を包む様に設定されることに注意してください。

- Show[グラフィックス1, グラフィックス2, ... , オプション1, ...] はいくつかのグラフィックスを合成してひとつのプロットを作ります。ここでグラフィックス1, ... は Graphics オブジェクトまたはそれらのリストです。
- 各グラフィックスのオプションのなかで択一しなければならない場合は、最初のグラフィックスのものがえらばれます。
- Show のオプションは各グラフィックスのオプションに優先します。
- グラフの範囲は全グラフィックスを包むように自動設定されますが、PlotRange を Show に対して指定すれば自由に設定できます。

14.7 グラフの表示位置の設定

Canvas の中でのあるグラフの表示位置は通常はその Canvas の大きさから自動的に決められます。しかし、関数 Rectangle を使えばその位置を自由に設定でき、またひとつの Canvas に複数のグラフを表示することもできます。例えば、

```
g1 = ListPlot[data, DisplayFunction -> Identity];
g2 = Plot[Sin[x], {x, -Pi, Pi}, DisplayFunction -> Identity];
Show[Graphics[ {
  Rectangle[{0, 0},   {1, 0.5}, g1],
  Rectangle[{0, 0.5}, {1, 1},   g2]} ]];
```

は図 41 のようなグラフを作ります。 ここで Rectangle 自身は Graphics オブジェクトでは

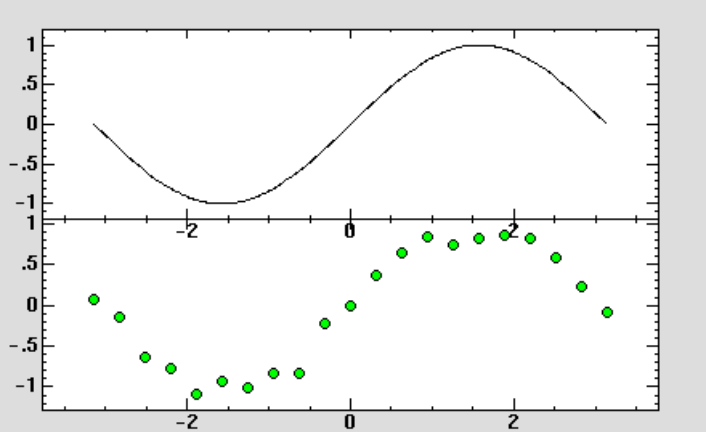

図 41: Rectangle によるグラフの表示位置の設定。

なくグラフィックス原子なので、Show に渡すためには（複数の場合はリストにして）Graphics をかぶせなければなりません。

- Rectangle[{x下限，y下限}，{x上限，y上限}，グラフィックス1] は{x下限，y下限}，{x上限，y上限} で指定される長方形の内部にグラフィックス1 を描くグラフィックス原子です。
- 標準のグラフの位置は {0，0}，{1，1} で、Rectangle のパラメータはそれを基準に相対値で表わします。
- Rectangle の第三引き数がグラフィックスでない場合は Rectangle は長方形のグラフィックス原子を表わします（14.9.3 参照）。

14.8 FitPlot

FitPlot は非線形回帰関数 Fit(11.9.1 参照) と ListPlot, Plot を組み合わせたものです。

- FitPlot[リスト1, 式0, シンボル0, {パラメータ1, 初期値}, ... , オプション] はリスト1 で表わされるデータ点に対して、パラメータ1, ... を動かして式0 の χ^2-Fit を求め、その結果をプロットします。
- 式0 はシンボル0, パラメータ1, ... を**あらわに含む式**です。FitPlot の二番目以降の引き数は前もって評価されないため、そのような「あらわに含む式」を得るためにここに Evaluate[式00] と書かねばならないことがあります (Evaluate 10.6.3 参照)。ここで式00 は式0 を結果としてもたらす式です。
- 各パラメータを {パラメータ, 初期値, {下限値, 上限値}} とするとそのパラメータの検索範囲を制限することができます。
- リスト1 のデータの構造は ListPlot と共通です。エラーバーがある場合はそれが χ^2 の重みとなります。エラーバーがない場合は重みは均等になります。
- FitPlot は {Fit の結果、Graphics オブジェクト} というリストを結果として返します。
- 次の例は図 33 の例に対し FitPlot を用いてsine 曲線による χ^2-Fit を求め、その結果をプロットしたものです (図 42)。

```
data1 = Append[#, 0.1]& /@ data;
FitPlot[data1, e Sin[f x + g], x, {e, 1}, {f, 1}, {g, 0}];
```

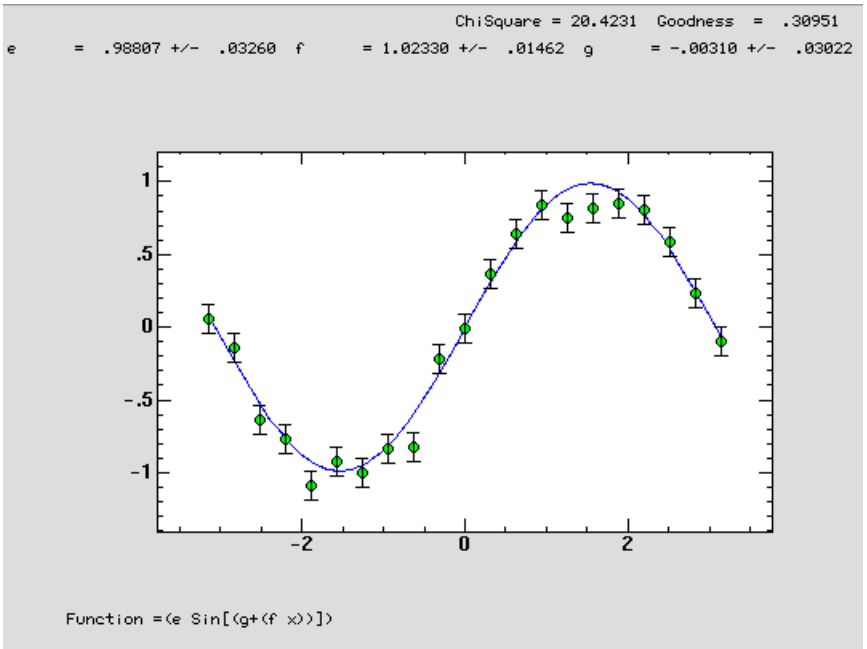

図 42: FitPlot の例。Fit の式は e*Sin[f x + g]、独立変数は x、パラメータはe, f, g です。

14.9 グラフィックス原子

グラフィックス原子は SAD のグラフィックスを構成する基本的な要素です。グラフィックス原子はシンボル Graphics を被せられることにより Graphics オブジェクトとなり、Show によりプロットされるものになります。ListPlot や Plot の返すものもこのような Graphics オブジェクトに他なりません。

現在のところ、Point, Line, Text, Rectangle がこのようなグラフィックス原子として定義されています。各原子はオプション指定により様々な属性をもつことができます。このようなオプションも必要に応じて充実させるつもりです。

14.9.1 Point

- Point[リスト1, オプション1, ...] はデータ点を表わすリスト1の各点のマーカを表わします。
- リスト1 の形式は ListPlot 14.3.11 と同じです。したがって、エラーバーを伴うことがあります。
- オプションは表 38 に示されます。

表 38: Point のオプション。

オプション	値	デフォルト値	効果
ErrorBarTickSize	実数値	1	エラーバーの鍵の相対長
PointSize	実数値	1	マーカの大きさの相対比
PointColor	色	"green"	マーカの内部色

14.9.2 Line

- Line[リスト1，オプション1，...] はデータ点を表わすリスト1 の各点を結ぶ線を表わします。
- リスト1 の形式は ListPlot 14.3.11 と同じです。したがって、データ点の各点にエラーバーを伴うことがあります。
- オプションは表 39 に示されます。

表 39: Line のオプション。

オプション	値	デフォルト値	効果
ErrorBarTickSize	実数値	1	エラーバーの鍵の相対長
Plot	True, False	False	データ点のマーカ表示
PlotJoined	True, False	True	データ点を線で結ぶか否か
PlotColor	色	"black"	グラフの線の色
PointSize	実数値	1	マーカの大きさの相対比
PointColor	色	"green"	マーカの内部色

14.9.3 Rectangle

- Rectangle[{$x1$, $y1$}, {$x2$, $y2$}, オプション1, ...] はデータ点 {$x1$, $y1$}, {$x2$, $y2$} で決まる長方形を表わします。
- Rectangle[{$x1$, $y1$}, {$x2$, $y2$}, Graphics[...]] は長方形の中に別のグラフィックスを描いたものを表わします (14.7 参照)。
- オプションは表 40 に示されます。

表 40: Rectangle のオプション。

オプション	値	デフォルト値	効果
FillColor	色	Null	内部の色、Null の時は枠だけを描く
MeshStyle	ビットマップ	Null	内部を塗るパターン、Null のときは完全に塗る
PlotColor	色	"black"	枠の色
Tags	文字列		付加するタグ(15.1.3 参照)

14.9.4 Text

- Text[{文字列1, {x, y}}, オプション1, ...] は位置 {x, y} に表示する文字列を表わします。
- 座標 {x, y} は Canvas の左下隅を {0, 0} とし、右上が正方向にとる CanvasDrawer、TopDrawer に固有の座標です。また、通常はグラフの左下が {2.4, 1.8} になります。
- 座標として {Scaled[x], Scaled[y]} の様に指定すると、データ点から決まる座標になります。
- オプションは表 41 に示されます。

表 41: Text のオプション。

オプション	値	デフォルト値	効果
PlotColor	色	"black"	テキストの色
Tags	文字列		付加するタグ(15.1.3 参照)
TextAlign	"left", "center", "right"	"left"	テキストの基準位置
TextFont	フォント	$DefaultFont	フォント (4.5.2 参照)
TextSize	実数値	1	大きさ、標準サイズに対する相対比

15 Canvas

前章で見た様に、Canvas は ListPlot などのプロットを表示する領域として用いられます。単なるグラフの表示ならば、ListPlot などを使うだけで充分なのですが、場合によっては Canvas に対して直接作用を及ぼす必要が生じます。例えば、描かれた図形の一部をクリックして何かある動作を行ったり、図形の移動や属性の変更を行うなどの仕事です。ここでは簡単な例をもとにこのような Canvas 特有の操作について説明します。

15.1 Canvas のアイテム

Canvas に描かれる部品はいくつかのアイテム item に分類されます。表 42 はそれらのアイテムの一覧です。

表 42: Canvas のアイテム。

アイテム	描かれる図形
Arc	扇形、弦、弓形
Bitmap	ビットマップ
Image	イメージ
Line	線分
Oval	楕円
Polygon	多角形
Rectangle	長方形
Text	文字列
Window	他の部品をはめ込む枠

これらのアイテムは Canvas 上に例えば次のように作成されます。

```
FFS;

  w = Window[];
  c = Canvas[w, Width -> 300, Height -> 300];

  c[Create$Rectangle] = {50, 50, 250, 250,
    Tags -> "rectarc rectoval"}};
  c[Create$Oval] = {50, 50, 250, 250,
    Fill -> "blue", Stipple -> "gray25",
    Tags -> "rectoval"};
  c[Create$Arc] = {50, 50, 250, 250,
    Style -> "pieslice", Fill -> "red",
    Start -> 60, Extent -> 90,
    Tags -> "rectarc"};

  TkWait[];
```

この例では Rectangle, Oval, Arc の3つのアイテムを図 43 のように描きます。

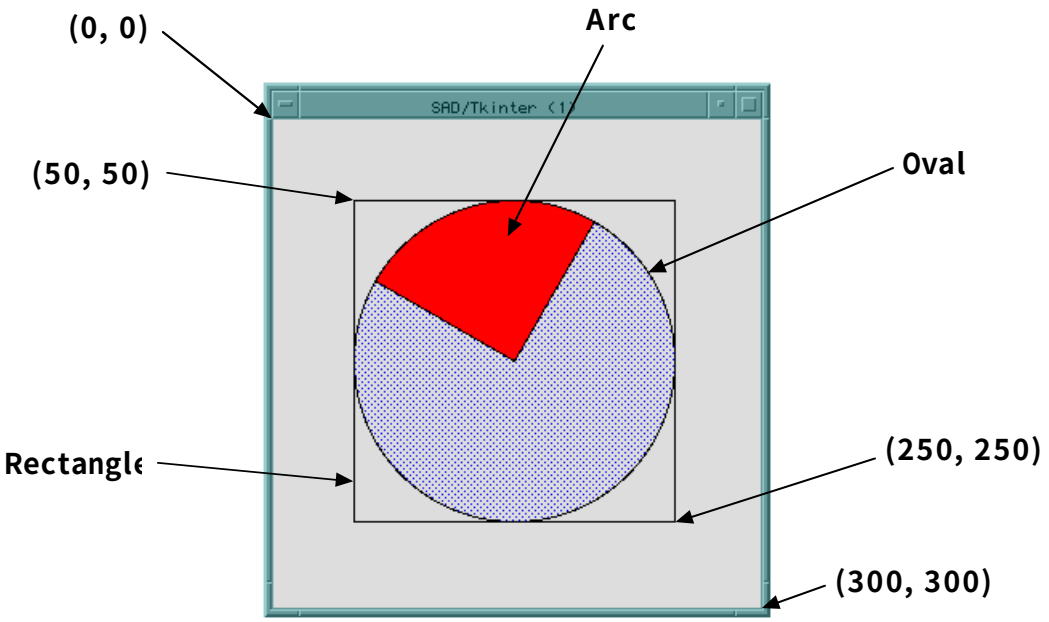

図 43: Canvas の例。座標は図のように与えられます。

このように、Canvas にあるアイテムを追加するには

```
Canvasシンボル[Create$アイテム] = {パラメータ1, ..., パラメータn,
  属性1 -> 値1, 属性2 -> 値2, ...}
```

のように書きます。ここでパラメータ1, ..., パラメータn, は一組またはそれ以上の座標です。これらは別々に書いても、リストにまとめても構いません。上の例では Create$Rectangle の 4 個の引き数がこのパラメータに当たります。次に属性の指定を必要に応じて行います。パラメータ達は必ず属性の指定よりも先に書かなければなりません。

15.1.1 Canvas の座標

Canvas では座標の指定は Window の座標の指定と同様に左上隅を (0, 0) とし、右下に増加する座標で、単位はピクセルです。この座標と、CanvasDrawer の座標とは別のものなので、混同しないようにしてください。

15.1.2 アイテム番号

ひとつの Canvas に対して、製作された各アイテムには製作順に**アイテム番号**が自動的につけられます。この番号は 1 から順に 1 ずつ増加します。途中で Delete などの操作でアイテムが消去されても、同じ番号が再利用されることはありません。この番号は後に各アイテムに対して様々な操作を及ぼすときに使われます。

15.1.3 Tags

Canvas のアイテムには Tags という属性があり、各アイテムをグループに分けて、名称を付けることができます。これにより後でグループ毎に様々な操作を与えることが容易にできます。名称は空白、{}、 などを含まない任意の文字列です。

ひとつのアイテムが複数のグループに所属することも可能です。例えば上の例では、Rectangle には "rectarc" と "rectoval" の二つのグループが指定されています（書き方が多少妙ですがこうしてください)。

例えば、上の例でさらに

```
c[Move] = {"rectoval", 30, -20};
c[Update];
```

などとしますと、図 44 のような結果になります。

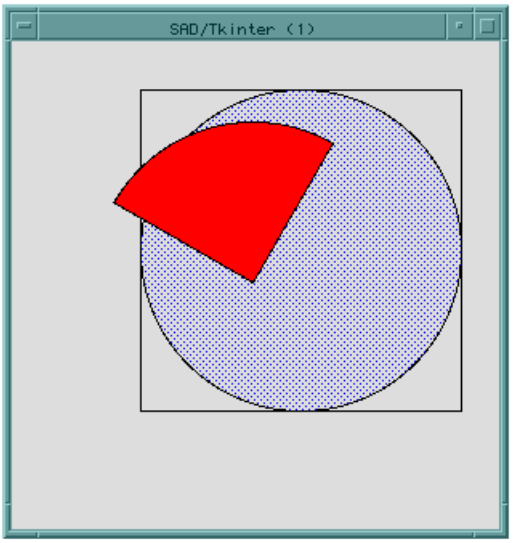

図 44: Tags でグループ化したアイテム達の移動。Rectangle と Oval を含むグループ ("rectoval") が (30, -20) だけ移動しました。

15.1.4 Arc

Arc は扇形、弓形、弧を描きます。Create$Arc はその Arc の弧を一部とする楕円の外接長方形の二つの対角頂点の座標を最初の 4 つのパラメータとします。

表 43 にその属性を掲げます。

表 43: Arc の属性。

属性	値	デフォルト	効果
Extent	角度(度)	360	弧の反時計方向の開き角
Fill	色	無指定	内部色
Outline	色	"black"	輪郭線の色
Start	角度(度)	0	弧の開始角、反時計方向に計る
Stipple	ビットマップ		塗りつぶしのビットマップ
Style	"pieslice", "chord", "arc"	"pieslice"	それぞれ扇形、弓形、弧の選択
Tags	文字列		そのアイテムにつけるタグ達
Width	数値(ピクセル)	1	輪郭線の幅

また、次の例は図 45 の様な結果になります。

```
w = Window[];
c = Canvas[w, Width -> 380, Height -> 140];
styles = {"pieslice", "chord", "arc"};
x = 20;
Scan[
  (c[Create$Arc] = {x, 20, x + 80, 100,
    Style -> #, Fill -> "red",
    Start -> -30, Extent -> -120};
   c[Create$Text] = {x + 40, 120,
     Text -> #};
   x += 120)&,
  styles];
```

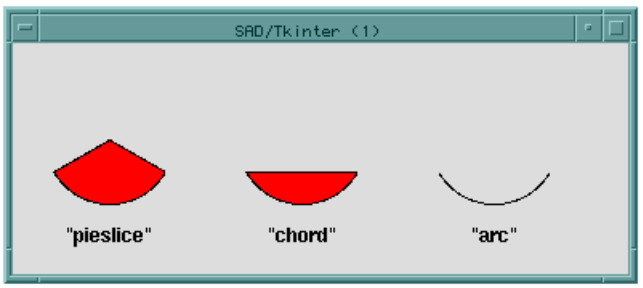

図 45: Arc の 3 種の Style。

15.1.5 Bitmap

Bitmap は Canvas 上にビットマップ（4.5.5 参照）を表示します。Create$Bitmap は位置をを指定するひと組の座標をパラメータとします。また、その座標と Bitmap との相対位置は属性 Anchor で指定します。Bitmap の属性は表 44 に掲げます。また、次の例は 4.5.5 節の図 11 を作ります。

```
w = Window[];
c = Canvas[w, Width -> 600, Height -> 100];
bitmap = {"error", "gray25", "gray50", "hourglass",
  "info", "questhead", "question", "warning"};
x = 50;
Scan[(
  c[Create$Bitmap]={x, 40, Bitmap -> #};
  c[Create$Text]={x, 70, Text -> "\""//#//"\""};
  x += 70)&,
  bitmap];
```

表 44: Bitmap の属性。

属性	値	デフォルト	効果
Anchor	"c" "n" "ne" "e" "se" "s" "sw" "w" "nw"	"c"	座標との相対位置
Background	色	無指定	背景色
Bitmap	ビットマップ		ビットマップ（文字列または "@ファイル名"）
Foreground	色	"black"	前景色
Tags	文字列		そのアイテムにつけるタグ達

15.1.6 Image

表 45: Image の属性。

属性	値	デフォルト	効果
Anchor	"c" "n" "ne" "e" "se" "s" "sw" "w" "nw"	"c"	座標との相対位置
Image	イメージ		イメージ
Tags	文字列		そのアイテムにつけるタグ達

15.1.7 Line

Line は折線を描きます。Create$Line のパラメータは座標を$x_1, y_1, x_2, y_2$, ... と順に並べたものです。属性 Smooth が True のときは各点はスプライン曲線で結ばれ、False のときは直線で結ばれます。属性 CapStyle と JoinStyle は線分の端部と接合部の形状を指定します（図 46 参照）。Line の属性は表 46 に掲げます。また、次の例は図 46 を作ります。

```
w = Window[];
c = Canvas[w, Width -> 640, Height -> 270];
options = {
  CapStyle -> "butt",
  CapStyle -> "projecting",
  CapStyle -> "round",
  JoinStyle -> "bevel",
  JoinStyle -> "miter",
  JoinStyle -> "round"};
{x, y} = {20, 90};
Scan[(
  c[Create$Line] = {x, y, x + 110, y - 60, x + 180, y,
    Width -> 20, #};
  c[Create$Text] = {x + 90, y + 20,
    Text -> #[[1]]//" -> \""//#[[2]]//"\""};
  x += 210;
  If[x > 500, x = 20; y += 120])&,
  options];
```

表 46: Line の属性。

属性	値	デフォルト	効果
Arrow	"none" "first" "last" "both"	"none"	矢頭の有無
ArrowShape	{接続部長, 全長, 全幅}		矢頭の形状(ピクセル)
CapStyle	"butt" "projecting" "round"	"butt"	端部の形状
Fill	色	"black"	線の色
JoinStyle	"bevel" "miter" "round"	"round"	接合部の形状
Smooth	True False	False	True: スプライン、False: 折線
SplineSteps	数値		スプラインの線分数
Stipple	ビットマップ		塗りつぶしのビットマップ
Tags	文字列		そのアイテムにつけるタグ達
Width	数値(ピクセル)	1	線の幅

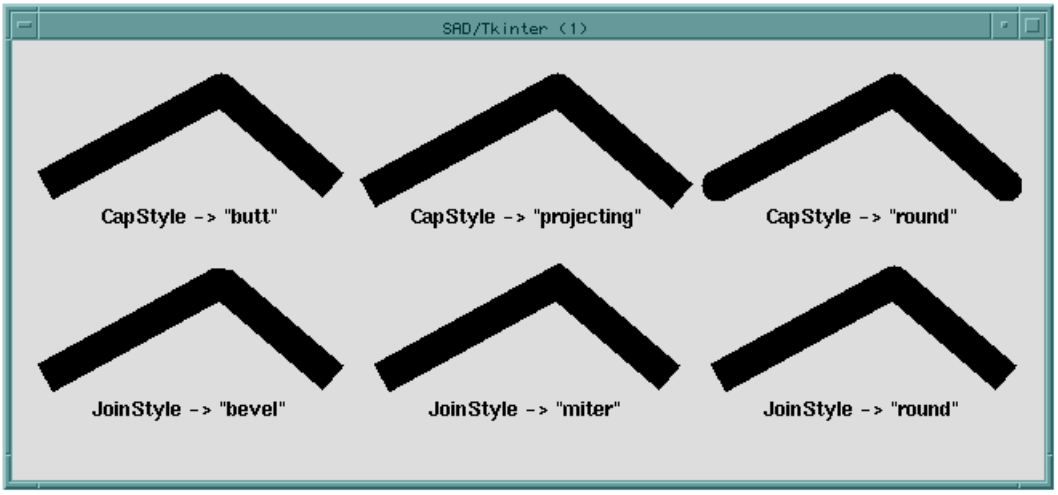

図 46: Line の CapStyle と JoinStyle 属性。

15.1.8 Oval

Oval は楕円を描きます。Create$Oval はその楕円の外接長方形の二つの対角頂点の座標を最初の 4 つのパラメータとします。

表 47: Oval の属性。

属性	値	デフォルト	効果
Fill	色	無指定	内部色
Outline	色	"black"	輪郭線の色
Stipple	ビットマップ		塗りつぶしのビットマップ
Tags	文字列		そのアイテムにつけるタグ達
Width	数値(ピクセル)	1	輪郭線の幅

15.1.9 Polygon

Polygon は閉じた他辺形を描きます。ただし、外形線はなく、全体が一体として扱われます。Create$Polygon のパラメータは頂点の座標を$x_1, y_1, x_2, y_2$, ... と順に並べたものです。属性 Smooth が True のときは頂点はスプライン曲線で結ばれ、False のときは直線で結ばれます。表 48 に属性を掲げます。次の例は図 47 に成ります。

```
w = Window[];
c = Canvas[w, Width -> 600, Height -> 160];
param := {x, y, x + 40, y - 40,
  x + 80, y, x + 120, y + 40,
  x + 160, y, x + 120, y - 40,
  x + 80, y, x + 40, y + 40};
{x, y} = {20, 60};
c[Create$Polygon] = {param, Smooth -> False};
c[Create$Text] = {x + 80, y + 60,
  Text -> "Smooth -> False"};
{x, y} = {200, 60};
c[Create$Polygon] = {param, Smooth -> True};
c[Create$Text] = {x + 80, y + 60,
  Text -> "Smooth -> True"};
{x, y} = {380, 60};
c[Create$Polygon] = {param, Smooth -> True,
  SplineSteps -> 3};
c[Create$Text] = {x + 80, y + 60,
  Text -> "Smooth -> True"};
c[Create$Text] = {x + 80, y + 76,
  Text -> "SplineSteps -> 3"};
```

表 48: Polygon の属性。

属性	値	デフォルト	効果
Fill	色	"black"	全体色
Smooth	True False	False	True: スプライン、False: 折線
SplineSteps	数値		スプラインの線分数
Stipple	ビットマップ		塗りつぶしのビットマップ
Tags	文字列		そのアイテムにつけるタグ達

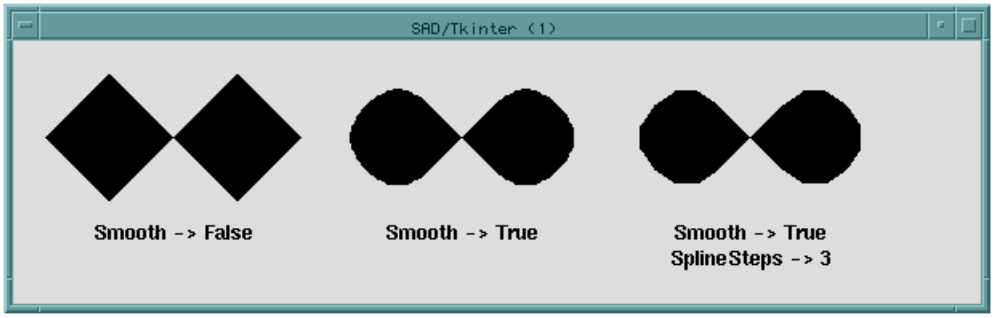

図 47: Polygon の例。Smooth -> True により、スプラインが用いられる。

15.1.10 Rectangle

Rectangle は傾きのない長方形です。Create$Rectangle はその二つの対角頂点の座標をパラメータとします。

表 49: Rectangle の属性。

属性	値	デフォルト	効果
Fill	色	無指定	内部色
Outline	色	"black"	輪郭線の色
Stipple	ビットマップ		塗りつぶしのビットマップ
Tags	文字列		そのアイテムにつけるタグ達
Width	数値(ピクセル)	1	輪郭線の幅

15.1.11 Text

Text は文字列を表示します。また Entry 部品のようにそれを編集することもできますが、その説明は省きます。Create$Text は位置を指定するひと組の座標をパラメータとします。また、その座標と Text との相対位置は属性 Anchor と Justify で指定します。Text の属性は表 50 に掲げます。

表 50: Text の属性。

属性	値	デフォルト	効果
Anchor	"c" "n" "ne" "e" "se" "s" "sw" "w" "nw"	"c"	座標との相対位置
Fill	色	"black"	文字の色
Font	フォント		フォント
Justify	"left" "right" "center"	"left"	文字の揃え方
Stipple	ビットマップ		塗りつぶしのビットマップ
Tags	文字列		そのアイテムにつけるタグ達
Text	文字列	""	表示文字列

15.1.12 Window

Window は Canvas の上に他の任意の Tkinter の部品を貼るのに用います。Create$Window は貼り付ける部品の位置を指定するひと組の座標をパラメータとし、属性 Window で部品を指定します。例えば次のようにすると、図 48 の様な結果が得られます。

```
w = Window[];
c = Canvas[w, Height -> 300, Width -> 400];
$DisplayFunction = CanvasDrawer;
Canvas$Widget = c;
Plot[Sin[x], {x, -Pi, Pi}];
b = Button[c, Text -> "Button on Canvas"];
c[Create$Window] = {150, 100, Window -> b};
```

私がこれまで試した限りでは、上に貼る Frame などを透明にすることはできないようです。

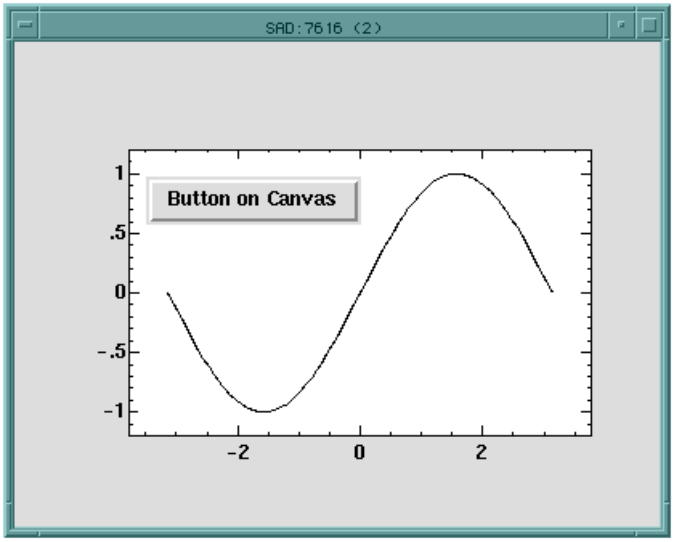

図 48: Canvas の中の Window の例。

15.2 アイテムの操作

Canvas 上に製作されたアイテムに対しては後から様々な操作を及ぼすことができます。以下の説明で c はある Canvas 部品を表わすものとします。

15.2.1 属性の変更および調査、ItemConfigure

- c[ItemConfigure] = {タグまたはアイテム番号, 属性 -> 値, ...} はアイテム番号で指定されるアイテム、またはタグで指定されるアイテム達の属性を変更します。また、アイテムの「上下」とはその時点の表示の重なり具合が前面にあるものを上とします。後から製作されたアイテムほど上に来ますが、変更可能です。
- c[ItemConfigure[アイテム番号, 属性]] はアイテム番号で指定されるアイテムの属性の設定値を返します。

15.2.2 タグの付加、AddTag

- c[AddTag$Above] = {タグ1, タグ2またはアイテム番号2} はアイテム番号2 で指定されるアイテム、またはタグ2 で指定されるものの内の最前面のアイテムの**直上のアイテム**にタグ1 を付加します。
- c[AddTag$All] = タグ1 はそれまでに製作されたすべてのアイテム達にタグ1 を付加します。
- c[AddTag$Below] = {タグ1, タグ2またはアイテム番号2} はアイテム番号2 で指定されるアイテム、またはタグ2 で指定されるものの内の最後面のアイテムの**直下のアイテム**にタグ1 を付加します。

- c[AddTag$Closest] = {タグ1，$x$，$y$} は位置（$x$ y）に最も近いアイテム達の内の最前面のアイテムにタグ1 を付加します。
- c[AddTag$Enclosed] = {タグ1，$x1$，$y1$，$x2$，$y2$} は指定される長方形に**完全に含まれる**アイテム達にタグ1 を付加します。
- c[AddTag$WithTag] = {タグ1，タグ2またはアイテム番号2} はアイテム番号2 で指定されるアイテム、またはタグ2 で指定されるアイテム達にタグ1 を付加します。

15.2.3 アイテムの移動、Move

- c[Move] = {タグまたはアイテム番号，Δx，Δy} はアイテム番号で指定されるアイテム、またはタグで指定されるアイテム達を（Δx，Δy）だけ移動させます。

15.2.4 位置の変更および調査、Coords

- c[Coords] = {アイテム番号，$x1$，$y1$，...} はアイテム番号で指定されるアイテムの位置を $x1$，$y1$，...に変更します。
- c[Coords[アイテム番号]] はアイテム番号で指定されるアイテムの位置を返します。

15.2.5 アイテムの消去、Delete

- c[Delete] = {タグ1またはアイテム番号1，...} はタグ1またはアイテム番号1，...で指定されるアイテム達を消去します。

15.2.6 アイテムの表示面の重なりの移動

- c[Lower] = タグまたはアイテム番号 はアイテム番号で指定されるアイテム、またはタグで指定されるアイテム達を表示の最後面に移動します。
- c[Lower] = {タグ1 またはアイテム番号1，タグ2 またはアイテム番号2} はアイテム番号1 で指定されるアイテム、またはタグ1 で指定されるアイテム達を、タグ2 またはアイテム番号2 のアイテムの直下の表示面に移動します。
- c[Raise] = タグまたはアイテム番号 はアイテム番号で指定されるアイテム、またはタグで指定されるアイテム達を表示の最前面に移動します。
- c[Raise] = {タグ1 またはアイテム番号1，タグ2またはアイテム番号2} はアイテム番号1 で指定されるアイテム、またはタグ1で指定されるアイテム達を、タグ2 またはアイテム番号2 のアイテムの直上の表示面に移動します。

15.2.7 タグの解除、Dtag

- c[DTag] = タグ1 はタグ1 へのアイテムの割り当てを解除します。
- c[DTag] = {タグ1, タグ2} はタグ1で指定されるアイテム達へのタグ2の割り当てを解除します。

15.2.8 タグの調査、GetTags

- c[GetTags[タグまたはアイテム番号]] はアイテム番号で指定されるアイテム、またはタグで指定されるものの中の最初のアイテムに付加されているタグ達をリストにして返します。

15.3 アイテムへのイヴェントの結合

Canvas の各アイテム、あるいはタグのつけられた各グループには Bind により、それぞれ独立にイヴェントを結合することができます。その方法は、例えば

```
c = Canvas[ ... ];
c[Create$Oval] = { ... , Tags -> "a"};
Bind[c, "Button-1", Print["OK"], Tags -> "a"];
```

の様に Bind にオプション Tags ->（**タグまたはアイテム番号、あるいはそれらのリスト**）を加えるだけで、あとは通常の部品に対する結合（4.2 参照）と全く同様です。また、変数 $Event にイヴェントを発生したアイテムの番号あるいはタグが返されます。

15.4 プロット関数で製作したグラフでのアイテムの操作

15.4.1 Canvas$ID によるアイテム番号の記録

ListPlot や Plot などのプロット関数も Canvas のアイテムの組み合わせでグラフを表現します。したがって、それらのアイテム番号が知られれば、個々のアイテムに対して上記の各種の操作を加えることができます。ListPlot が製作したアイテムの番号は、オプション Tags -> True により、その都度シンボル Canvas$ID に記録されるようになっています。例えば、

```
w = Window[];
c = Canvas[w, Width -> 600, Height -> 400];
$DisplayFunction = CanvasDrawer;
Canvas$Widget = c;

data = {{1, 2, 0.5}, {2, 3, 2},
  {3, 5, 0.5}, {4, 2, 0.5}, {5, -2, 0.5}};
ListPlot[data, Tags -> True,
  Plot -> True, PlotJoined -> True,
  PointSize -> 2];
Print[Canvas$ID];
```

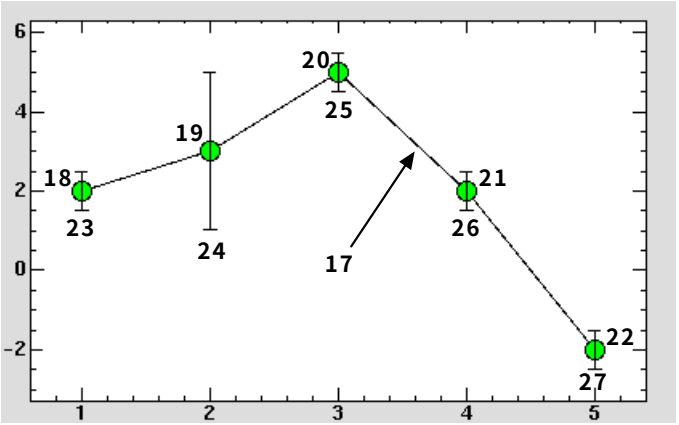

図 49: ListPlot でプロットしたグラフのアイテム番号。Tags -> True により、Canvas$ID にアイテム番号が記録されます。この場合、グラフの線には番号 17、データ点のマーカには番号 18–22、エラーバーには番号 23–27 が割り当てられます。Canvas$ID の値は {{17, 27}} になります。(図上の番号は説明のためのもので、実際には表示されません。)

と 5点のエラーバー付のデータを図 49 のようにプロットしたとします。

こうすると、このリストプロットの直後では Canvas$IDの値は {{17, 27}} になります。そして、実際にはグラフの線には番号 17、データ点のマーカには番号 18–22、エラーバーには番号 23–27 が順に割り振られます。つまり、Canvas$ID は

```
{{開始1, 終了1}, {開始2, 終了2}, ...}
```

という形式で表わされています。それぞれの要素リストは描かれるグラフィックス原子 (14.9 参照) に対応します。この場合 16 以前のアイテム番号には座標軸や目盛などアイテムが割り当てられていますが、それらの数はグラフの条件に左右されるので、どれが何であるとは簡単には言えません。Canvas$ID の返すものは常にデータ、あるいは直接 Epilog などで書き込みを支持されたグラフィックス原子に対応するアイテムの番号だけです。また、アイテムは、グラフの線、マーカ、エラーバーの順に並び、アイテム番号はデータの順に並びます。エラーバーは両方向にあっても常にひとつのアイテムです。したがってアイテム番号とデータ点の対応は一意的です。PlotRange により表示されるデータ点が制限される場合でも、アイテムは必ず製作され、アイテム番号も付加されます。ただし、グラフの線は破線などの場合にはひとつのアイテムにはなりません。

Show により複数のプロットを同時に表示するような場合にも Canvas$ID にはプロット毎、あるいはグラフィックス原子毎にアイテム番号が {開始, 終了} というリストで記録されます。

Canvas$ID の値は ListPlot などのオプション Initialize が True のとき (デフォルト) {} にリセットされます。そうでない時にはそれまでの Canvas$ID に次々に追加されていきます。

15.4.2 Bind によるアイテムへのイヴェントの結合

以上の方法で各アイテムのアイテム番号がわかりましたから、今度はそれを使って各アイテムにイヴェントを結び付けてみましょう。これは基本的には 15.3 で説明したことの応用です。まず、上の例に加えて、

```
ld = Length[data];
markers = Canvas$ID[[1, 1]] + Range[ld];
Bind[c, "<Button-1>", cmd, Tags -> markers];
```

とすると、markers にはマーカ達のアイテム番号がリストになって入ります。この場合は {18, 19, 20, 21, 22} になります。次に、Bind で Tags -> markers と指定することにより、各マーカはイヴェント "<Button-1>"（マウスボタン1 の押し下げ）が発生する度に式 cmd を評価実行することに割り当てられました。そこで、例えば

```
(switch[#] = True)& /@ markers;                          (ア)
markcolor[True] = "green";                               (イ)
markcolor[False] = "red";                                (ウ)
errorbarcolor[True] = "black";                           (エ)
errorbarcolor[False] = "gray";                           (オ)

cmd := Module[{id = ToExpression[Tag /. $Event]},        (カ)
  switch[id] = ~switch[id];                              (キ)
  c[ItemConfigure] = {id,
    Fill -> markcolor[switch[id]]};                      (ク)
  c[ItemConfigure] = {id + ld,
    Fill -> errorbarcolor[switch[id]]}];                 (ケ)
```

とすると、各マーカをクリックする度に図 50 の様にそのマーカとエラーバーの色をトグルすることができます。 ここでまず、（ア）はシンボル switch をマーカの状態を保持する関数として定義し

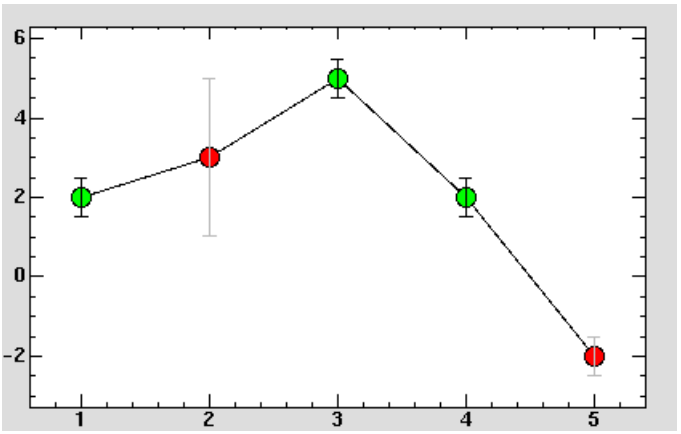

図 50: マーカをクリックするとそのマーカとエラーバーの色がトグルする。

ています。（イ）から（オ）はそれぞれの状態に対応する色の定義です。（カ）から（ケ）がシン

ボル cmd の定義ですが、まず（カ）では局所シンボル id にシンボル Tag の $Event を用いた置換により、クリックされたアイテムのアイテム番号を取り出しています。$Event はこの場合

```
{(Widget:>c),(Tag->"20"),(Type->"<Button>"),(X->291),(Y->124),(XRoot->693),
 (YRoot->542),(Height->0),(Width->0),(Char->"??"),(KeySym->"??"),
 (SendEvent->0),(KeyCode->1),(State->0),(KeySymNum->1),(Time->6784003)}
```

の様な規則のリストです。$Event は大域的なシンボルですが、イヴェントが発生し結合されたコマンドが呼ばれる度に値が局所的に設定されます（8.8 節参照)。$Event に含まれるシンボルについては表 2 を参照してください。

(キ)は switch のトグルです。(ク)、(ケ) は番号 id のマーカ及び番号 id + ld のエラーバーの色を設定し直しています。こうして、アイテムとイヴェントを結合する基本的な動作は確認されました。

16　部品の合成

SAD/Tkinter では、ユーザーがこれまで説明してきたようなシステムに備え付けの部品を組み合わせて新しい部品を定義して備え付けの部品と似たように使うことができます。

16.1　LabeledEntry

いま、単純な例として、TextLabel と Entry を組み合わせた LebeledEntry という部品を定義したいとします。この部品はある Frame のなかの左側に TextLabel を、その右に Entry を並べるだけのものです。そのひとつの方法は、例えば

```
(a_ = LabeledEntry[w_, opt___]) ^:= (                    (ア)
  DeleteWidget[a];                                       (イ)
  Clear[a];                                              (ウ)
  a[Frame] = Frame[w, opt];                              (エ)
  a[Label] = TextLabel[a[Frame], Side -> "left", opt];   (オ)
  a[Entry] = Entry[a[Frame], Side -> "left", opt];       (カ)

  a[b_] := (
    a[Frame][b];                                         (キ)
    a[Label][b];                                         (ク)
    a[Entry][b]);                                        (ケ)

  (a[b_] = c_) ^:= (
    a[Frame][b] = c;                                     (コ)
    a[Label][b] = c;                                     (サ)
    a[Entry][b] = c);                                    (シ)

  (a[b_] := c_) ^:= (
    a[Frame][b] := c;                                    (ス)
    a[Label][b] := c;                                    (セ)
    a[Entry][b] := c);                                   (ソ)
  );
```

のようなものになるでしょう。ここで、まず、（ア）は、この LabeledEntry という新しい部品が普通の部品と同様に

```
シンボル = LabeledEntry[親, オプション, ...]
```

のような形で定義するために上方値（8.6 参照）を用いています。

（イ）と（ウ）はシンボル a などにそれまでに割り当てられているかもしれない部品や値の割り当てを解除しています。

この LabeledEntry という部品は Frame, TextLabel, Entry という 3 つの従属部品から成り立っていますが、それらの従属部品の定義が（エ)--(カ）です。左辺の形はそれらが従属部品であることにふさわしい書き方ではないでしょうか。ここで、オプション opt は 3 つの従属部品に共通にあたえられています。つまり、この 3 種の**いずれにも同じ属性**が指定されます。例えば、

Background という属性は3 つに共通ですからどれにも当てはまります。しかし、例えば Text という属性は TextLabel にしかありませんから TextLabel だけに適用され、ほかの部品では無視されます。また、Side という属性は（オ）（カ） の様に TextLabel と Entry では先に書いた Side -> "left" が優先されるため、Frame に対してだけ有効となります。もし、属性を個別に設定したければ、例えば

```
シンボル[Entry][属性] = 値
```

のようにすればよいのです。

さて、（キ）から（ソ）まではこの LabeledEntry 全体に対して共通の操作、属性の変更を普通の部品と同様の形式で行えるようにするためのいくつかの定義です。

この LabeledEntry を使って、例えば

```
w = Window[];
le1 = LabeledEntry[w,
  Text -> "username: ",Side->"top"];
le2 = LabeledEntry[w,
  Text -> "password: ",Side->"top", ShowText -> "*"];
```

とすると図 51 のようなパネルが作られます。

図 51: 合成された部品 LabeledEntry の例。

なお、このLabeledEntry で作られた部品全体を DeleteWidget（4.7 参照）で消去することもできます。

以上のように、LabeledEntry のような簡単な場合には部品の合成はほぼうまくいきます。さらに複雑な場合にどの程度のものが可能であるかは未知数ですが挑戦してみてください。

17 例題

17.1 SimpleDialog

以下に示す SimpleDialog は、多くのプログラムで多用される、操作者との単純な応答を行うパネルです。まず、その定義は、

```
SimpleDialog[message_, texts_]:=Module[
   {w, tl, fr, i, b, l = Length[texts], r},      (ア) 局所シンボルの定義
   w = Window[];
   tl = TextLabel[w, Text -> message,
     PadX -> 20, PadY -> 10];                    (イ) 引き数 message の表示
   fr = Frame[w];                                (ウ) ボタンを並べる Frame
   i = 1;
   Scan[(                                        (エ) Do を用いてもよい。
     With[{m = #},                               (オ) # は texts の各要素
       b[i] = Button[fr, Text -> m,              (カ) 各ボタン b[i] の定義
         Command :> TkReturn[m],                 (キ) コマンドの設定
         BD -> 4, PadX -> 10, PadY -> 10,
         Side -> "left", Relief -> "ridge"]];
     i++)&,
     texts];
   b[l][Relief] = "raised";                      (ク) l は texts の長さ
   Bind[b[l], "<Key-Return>",                    (ケ) 最後のボタンをデフォルト
     TkReturn[texts[l]]];                            (Rreturn キーで反応)
   b[l][Focus$Set];                              (コ) キー応答のための焦点
   r = TkWait[];                                 (サ) 応答待ち
   w =.;                                         (シ) この Window を削除
   Update[];                                     (ス) 表示を更新
   r];                                           (セ) 押されたキーラヴェルを返す
```

です。この使い方は

```
シンボル = SimpleDialog[メッセージ, ボタンリスト]
```

のようにします。ここでメッセージは文字列、ボタンリストはボタンの表面に書く文字列のリストです。例えば、

```
result = SimpleDialog[
  "Do you want to save?",
  {"Cancel", "Don't save", "Save"}];
```

の様にすると、図 52 のような Window が現われます。ここでいずれかのボタンをクリックすると、その名称が返され、シンボル result に割り当てられます。 また、最後のボタン（この例では "Save"）はデフォルトとして（ケ)(コ）により Return キーを押してもクリックと同じ結果になるように設定されています。

図 52: SimpleDialog の例。

SimpleDialog は Module を利用した関数で、作られるウインドウ及び内部の部品はすべて（ア）で定義した局所シンボルに割り当てられ、実行後には（シ)(ス）によりすべて消去されるようになっています。

作成されるボタンの数は制限ありません。内部では（エ）の Scan により、ボタンは b[i] に割り当てられています。（オ）で With が用いられていますが、それは（キ）のコマンドの TkReturn[m] の m の値を前もって決めたいためです。With により、m は値 #、即ち texts の各要素になるわけですが、ここを Command :> TkReturn[#] と書きますと、:> の右辺が評価されないため、# がそのまま残ってしまい、後で TkReturn[#] としても # が未定義になってしまいます。

（サ）では TkWait[] により応答を待ちます。この SimpleDialog 自身が別のボタンの応答コマンドとして使用される場合などでは、TkWait[] が幾重にも重なりますが、それは構いません。

17.2 グラフの大きさをウインドウの大きさに追従させる

実際のアプリケーションではグラフを描くウインドウの大きさを操作者が必要に応じて変更したくなる場合が多々あります。次の例はこのような場合にそのウインドウの中に描かれたグラフを、そのウインドウの大きさに応じて再描画するものです。

```
FFS;
  $DisplayFunction = CanvasDrawer;
  w = Window[];
  c1 = Canvas[w, Height -> 100, Width -> 150,
    Side -> "left",
    Expand -> True, Fill -> "both"];           (ア)
  c2 = Canvas[w, Height -> 100, Width -> 150,
    Side -> "left",
    Expand -> True, Fill -> "both"];           (イ)
  pl:=(
    Canvas$Widget = c1;
    Plot[Sin[x],{x,-Pi,Pi}];
    Canvas$Widget = c2;
    Plot[Cos[x],{x,-Pi,Pi}]);                  (ウ)
  pl;
  Bind[w, "<Configure>", pl];                  (エ)
  TkWait[];
```

この例は（ア)(イ）で作った左右に並ぶ二つの Canvas c1, c2 に、(ウ）のコマンド pl でそれぞれにグラフを出力します。ここで、Canvas の属性に Expand -> True, Fill -> "both" と指定することにより、ウインドウの伸縮に Canvas が追随して変化することを保証します。そして、(エ）でコマンド pl とウインドウの属性の変更に伴うイヴェント"<Configure>" を結合することにより、ウインドウの伸縮の度に再描画してグラフの大きさを追従させることができます。(ただし、その場合、各 Canvas は最初に指定したサイズを最小サイズとして保持しようとするため、あまりウインドウが小さくなると、二つのグラフの大きさが不均等になります。)

この例を図 53 に示します。

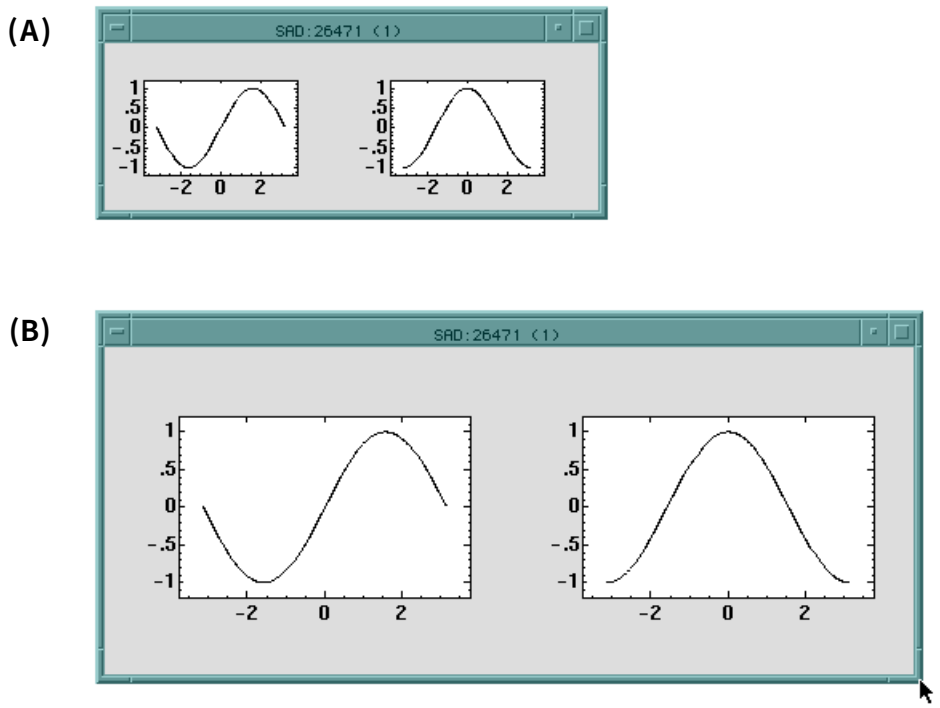

図 53: イヴェント "<Configure>" によりウインドウのサイズの変更にグラフの大きさを追従させることができる。(A) 初期状態。(B) マウスによりウインドウを拡大した例。

17.3 グラフのズーミング

さて、次の例は、Plot で書いたグラフ上でドラッグした領域を拡大してプロットしなおすという、世の中によくある機能を実現したものです。

```
FFS;

  StartSelect := Module[                                          (ア)
    {{x, y, c} = {X, Y, Widget}/.$Event},
    With[{c},
      c[Create$Line]={x, y, x, y,
        Fill -> "blue", Stipple -> "gray50",
```

```
      Tags -> "selection"}];
  Bind[c, "<Motion>", DragSelect];                          (イ)
  Bind[c, "<ButtonRelease-1>", RedrawSelect];               (ウ)
  selstart=selend={x,y}];                                   (エ)

DragSelect := Module[                                       (オ)
  {{x, y, c} = {X, Y, Widget}/.$Event},
  With[{c, x0 = selstart[[1]], y0 = selstart[[2]]},
    c[Delete] = "selection";
    c[Create$Line] = {x0, y0, x, y0, x, y, x0, y, x0, y0,
      Fill -> "blue", Stipple -> "gray50",
      Tags -> "selection"};                                 (カ)
    selend = {x,y}]];

RedrawSelect := Module[                                     (キ)
  {c = Widget/.$Event},
  c[Delete] = "selection";
  If[selstart <> selend,
    z0 = (selstart - Canvas$Offset) / Canvas$Scale;
    z1 = (selend   - Canvas$Offset) / Canvas$Scale;
    pl[z0, z1]];
  Bind[c, "<Motion>"];                                      (ク)
  Bind[c, "<ButtonRelease-1>"]];                            (ケ)

pl[z0_,z1_] :=                                              (コ)
  Plot[f[x],
    {x, Min[z0[[1]], z1[[1]]], Max[z0[[1]], z1[[1]]]},
    PlotRange ->
      {Min[z0[[2]], z1[[2]]], Max[z0[[2]], z1[[2]]]}];
plinit := pl[{-1, -1.2}, {1, 1.2}];

w = Window[];
c = Canvas[w, Width -> 600, Height -> 400];
$DisplayFunction = CanvasDrawer;
Canvas$Widget = c;

f[0] = 0;
f[x_] := Sin[1/x];

plinit;
Bind[c, "<Button-1>", StartSelect];                         (サ)
Bind[c, "<M-Button-1>", plinit];                            (シ)

TkWait[];
```

この例はまず初期状態では図 54(A) のようなプロットをつくります。そこで図のようにマウスボタン1 を押しながら長方形をドラッグし、ボタンを離すと(B) のように選択部分が x、y 共に拡大されて再プロットされます。

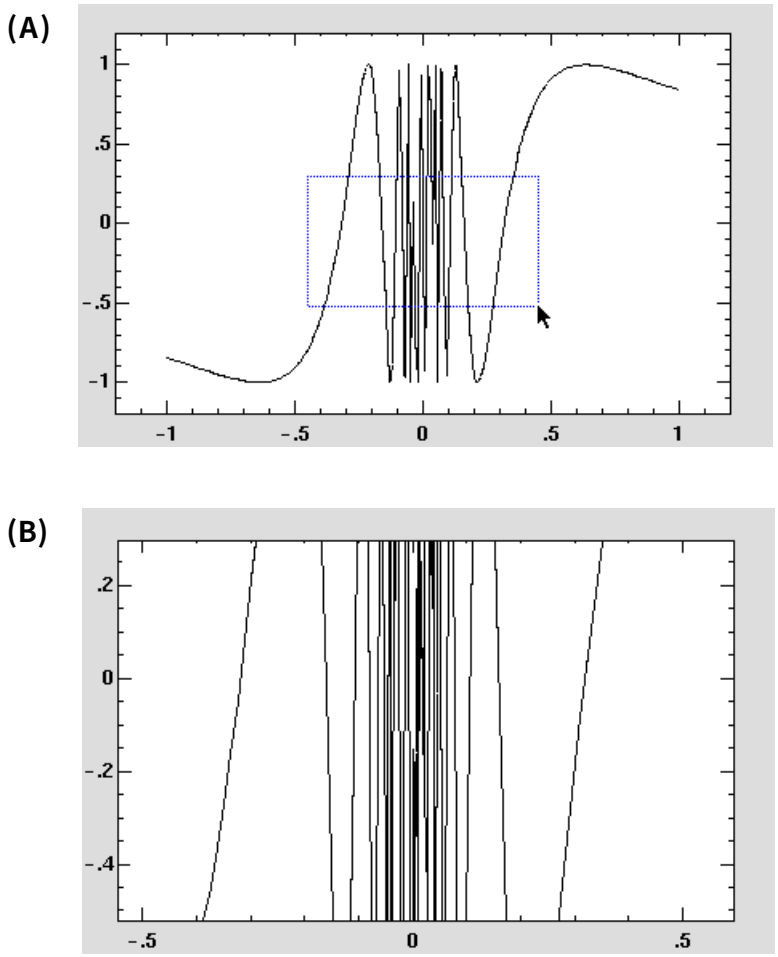

図 54: (A) マウスでドラッグした範囲を (B) 拡大して表示する。

この選択、拡大の動作は 3 つのイヴェントによって行われます。まず、ボタンの押し下げで、これは（ア）のStartSelect を呼びます。StartSelect はその中で（イ)(ウ）にあるように、マウスカーソルの移動とボタンの解放を DragSelect と RedrawSelect に結合します。そして（エ）で選択範囲の起点を記録します。次に DragSelect はその時点の選択範囲を四辺形で書き直します。その四辺形にはタグ "selection" がつけられています。最後に RedrawSelect は選択範囲を確定させ、Canvas 座標を Plot のデータ座標に換算して再プロットします。また、(ク)(ケ）のように、移動とボタンの解放のふたつのイヴェントへの結合を解除しています。このようにして、図 54(B) にあるような拡大図が得られます。

また、この例では（シ）でメタ-ボタン1 のイヴェントにより、プロットを初期状態に戻すようにしてあります。

謝辞

SAD/Tkinter の開発にあたっては山本昇（Python/Tkinter の導入）・赤坂展昌（部品の開発）両氏のご助力を、また、このマニュアルの編集においては小磯晴代氏のご協力を頂きましたことを深く感謝いたします。

索引

D

E

F

G

H

I

N

O

P

R

S

T

U

V

W

X

Y

Z

SAD 計算機へのユーザー登録手続きについて
－システム管理者より－

1．"SAD計算機登録申込書" を入手する。

"SAD計算機登録申込書" は アップルトークネットワーク上の"II TR Users 2, ACC MacII Server in Accelerator"から入手できます。Guestでサーバに接続した後、"SAD計算機"フォルダーの中にある"SAD計算機登録申込書"を御自分のディスクにコピーしてください。

２．"SAD計算機登録申込書" に必要な情報を記入する。

３．記入した"SAD計算機登録申込書"のファイル名を申請者の名前に変更する。

４．記入済み、名前変更済みの登録申込書を提出する。

登録申込書を管理者までお届けください。"SAD計算機"フォルダー中の"登録申込用紙投入箱"フォルダーに記入済みの申込書をコピーしていただければ結構です。

5.SAD計算機管理者にユーザー登録したい旨連絡する。

登録申込書を"登録申込用紙投入箱"フォルダーにコピーした後、e-mail等の方法で管理者までお知らせください。管理者は連絡があったときにしかこのフォルダーをチェックしません。現在の管理者は、

三増俊広（mimashi@kekvax.kek.jp）

山本昇（yamamoto@kekvax.kek.jp）

の両名です。

6.所外の利用者の方に。
KEK所員以外の方には計算機使用について誓約書を出していただいています。"SAD計算機"フォルダー内の誓約書を良くお読みの上、自署された誓約書を管理者までご提出下さい。

SAD計算機アカウント取得申込書

申込日：

名前：

所属：

職名：

連絡先住所（KEK以外の場合）：

居室番号：　　email address:

内線：　　ポケットベル：

希望するUser name ：

初期Password　：

このパスワードは初期値として使われます。アカウント取得後直ちに他の
パスワードに変更してください。

使用したいShell：

KEK内UNIXユーザID番号(uid)：

KEK内UNIXグループID番号(gid):

その他：

システム管理者記録用
Disk Quota:
/users
/users1
/scratch1

other group: