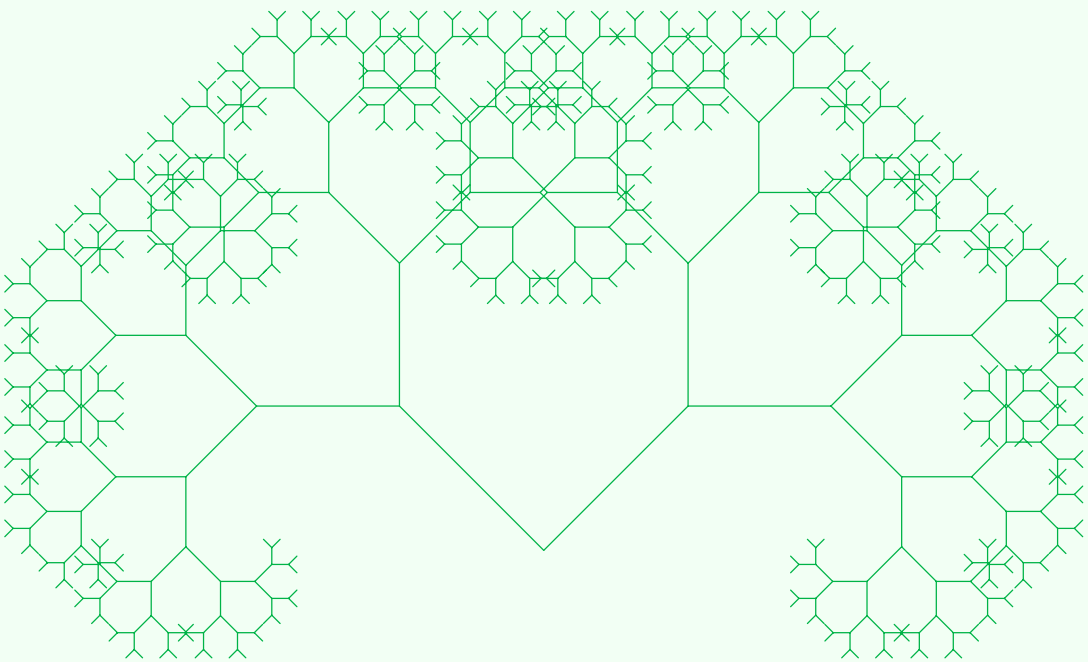
METAFONT&
METAPOSTで

楽々お絵かき

みなも 著



目　次

第 1 部 導入編(初級)

まずは METAFONT や METAPOST の動かし方と，
最低限の文法知識をまとめておきましょう。
もっとも，
「使いながら覚える」主義の人は，文法の項目をとばしてもかまいません。
文法については，今後折を見て順次触れていきますので，
初めから総括的に理解する必要はありません。

第 1 講

起動！ — の前に

1 METAFONT 言語

TEX の生みの親である Donald E. Knuth 氏は，TEX システムと同時に METAFONT システムも開発されました。これは，TEX で使用するフォントを生成するための言語ですが，フォントの形状を記述するだけでなく，図形の記述という目で見ても，非常に優れたものであります。これをフォントの生成ではなく，図版の生成に生かしたものが，John D. Hobby 氏による METAPOST というシステムです。METAFONT と METAPOST はほぼ同じ文法で，METAFONT がフォントを生成するのに対し，METAPOST は PostScript ファイルを生成するという点が違います。

METAFONT と METAPOST をあわせたものは，META 言語とか META ファミリーとよばれています。この本の目的は，この META 言語の入門 — とりわけ，META 言語で「絵」を描くために必要な知識をまとめていくことにあります。ただし,「まとめる」といっても，最初から厳密な文法を前面に押し出すことはしません。まずはいくつかの例に触れ，実際に手を動かして META 言語の感覚に慣れてもらいながら，適宜必要な知識を織り交ぜ，ある程度実感がつかめてから詳しい文法に入る，という形式を心がけていますので，軽い気持ちで読み始めて下さい。

なお，この本は，拙作の LATEX パッケージ MEPOTEX を使って作られています。これは，TEX ソース内に METAPOST ソースを埋め込むことができるパッケージです。埋め込まれた METAPOST ソースはタイプセット時に自動的に抽出され，METAPOST に渡されます。さらに，抽出されたソースをもとに METAPOST によって生成された PostScript ファイルは，(EPS 画像として) 自動的に文中に貼り付けられます[*1]。METAFONT がフォントを，METAPOST が PostScript ファイルを生成しても，生成された結果を確認するのは初心者には敷居の高いものに見えるかもしれません。しかし，MEPOTEX であれば，普通の TEX ソースをタイプセットしてプレビューするのと同じ手間で結果を「見る」ことができますので，初心者にも，META 言語マクロを作っている上級者にも重宝するものと自負しています。この本では適宜 MEPOTEX の使い方 — 基本線は METAPOST に同じ — も織り交ぜていきますので，以降，META 言語といった場合には MEPOTEX の METAPOST ソース部分も指しているものとお考え下さい。

2 実験用ディレクトリの作成

さて，これからこの本では，META 言語を使っていろいろ実験しながら，その使い方を学んでいこうとしているわけですが，その前に，実験用のディレクトリ (フォルダともい

[*1]なお，EPS ファイルの取り込みには LATEX の標準グラフィックパッケージである `graphicx.sty` を利用しています。

います) を作っておきましょう。これは，META 言語も TEX と同じく起動したディレクトリにログファイルなどをはき出すため，変なところで起動すると，そこにログファイルを残してしまうからです。実験が終わった際にそれらをまとめて消しやすいよう，特定のディレクトリで実験するよう決めておけば，後始末が楽です。

ところで，ディレクトリの作り方などの操作は OS によって違います。筆者は Windows パソコンしかもっていませんので，以下の文章は Windows の場合の説明となることをご了承下さい[*2]。Windows の場合で，たとえば D ドライブのルートにすでにある，meta と言う名前のディレクトリの下に，新たに test という名前のディレクトリを作るには，以下のようにします[*3]。

(1) エクスプローラで D ドライブの meta ディレクトリを表示させます。
(2) メニューから［**ファイル**］→［**新規作成**］→［**フォルダ**］を選択します。
(3) すると「**新しいフォルダ**」という名前のディレクトリが作られます。これをマウスで選択し，名前の部分をもう一度クリックすると名前が変更できるようになりますので，test と入力して Enter キーで確定させます[*4]。

3 DOS 窓でのカレントディレクトリの変更

METAFONT や METAPOST を直接起動するとき，Windows ではいわゆる DOS 窓で作業をします。ここでは，DOS 窓を開いて，カレントディレクトリ[*5]を先程作った実験用ディレクトリに変更する方法を説明します。

DOS 窓を開くには，［**スタート**］ボタンを押し，［**プログラム**］→［**コマンドプロンプト**］とメニューを選択します[*6]。すると DOS 窓が開き，画面のカーソルのある行に

```
C:¥Documents and Settings¥ユーザー名>
```

などと表示されると思います[*7]。これが，Windows (の DOS 窓) のプロンプト (入力要求) で，C:¥Documents〜 はカレントディレクトリを表しています。これを先程作った実験用ディレクトリに変更します。実験用ディレクトリを D ドライブに作った場合は，

```
D:
```

と入力して Enter キーを押すと，プロンプトに表示されているカレントディレクトリが，D: で始まるものに変わるはずです[*8]。次に，

```
cd ¥meta¥test
```

と入力すると，プロンプトが D:¥meta¥test> に変わって，カレントディレクトリが先程作った実験用ディレクトリに移ったことがわかると思います。

さて，これで準備は完了です。次講以降では実際に METAFONT や METAPOST を起動してみましょう。

[*2] 他の OS の場合はその OS で TEX や META 言語を使われている方にご相談を。⟵ 無責任。

[*3] Windows の設定によっては，細かい点が上記と違うこともあり得ますので，その際は Windows の解説書かマニュアルをご確認下さい。

[*4] 後で作業しやすいよう，ディレクトリ名は半角アルファベットで構成して下さい。

[*5] 現在の作業用のディレクトリのことです。

[*6] Windows95/98 では［**プログラム**］→［MS-DOS **プロンプト**］です。

[*7] 環境の設定や Windows の種類によって違うことがあります。例えば，Windows95/98 の初期設定では C:¥WINDOWS> です。

[*8] C ドライブに実験用ディレクトリを作った場合はこの操作は不要です。

第 2 講 METAFONT を対話形式で動かす

1 起　動！

今回は最初と言うことで，まず METAFONT を起動して，簡単な命令を実行させてみることにしましょう[*1]。

一般的には，METAFONT を使う際，フォントの形を記述した METAFONT ソースを，予めエディタなどで作成し，それを METAFONT に処理させるわけですが，処理させた結果を確認するには少し知識が必要ですので，ここでは「METAFONT を起動して METAFONT から何か反応が返ってくる様子を見るところまで」を試してみます。

さて，METAFONT をソースファイルを指定せずに起動しますと，オンラインで対話的に入力を要求してきます。入力した命令に対する画像を，online display とよばれる窓に逐一表示させることもできます。実用性はあまりありませんが，今回のように「ちょっと使ってみる」にはちょうどいいと思いますので，以下，実際に手を動かして試してみて下さい。

まず，前講の方法で実験用のディレクトリを作り，(Windows の場合) DOS 窓を開いて，カレントディレクトリを実験用のディレクトリに移します。そして，DOS 窓のプロンプト (入力要求) に対して，

```
mf
```
[*2]

とだけ打ち込んで Enter キーを押して下さい。これで METAFONT が起動するはずです[*3]。他の OS の場合も，コマンドを入力できる画面を開いて起動コマンド (普通 `mf`) を打ち込むか，アイコンを探してクリックなどで起動できるはずです。

うまく起動すれば，`This is METAFONT.` 云々と表示されて，カーソルのある行が

```
**
```

になっていると思います。これが METAFONT のプロンプト (入力要求) で，`*` 2 つの場合は，ファイル名入力要求です。今回はファイル名を入力せず，対話的に使うので，

```
¥relax
```
[*4]

とだけ入力して Enter キーを押して下さい。プロンプトが

```
*
```

[*1] METAFONT に興味がないという人は，**第2講** 〜 **第4講** をとばして下さってかまいません。

[*2] 一時期，online display を使う METAFONT の起動コマンドが `mfwin` だったことがあります。`mf` で起動して online display を使えなかった場合は，`mfwin` を試してみてください。

[*3] 起動しない場合，METAFONT が入っていない，METAFONT のプログラム名が `mf` (や `mfwin`) ではない，METAFONT のプログラムの在処にパスが通っていない，などの理由が考えられます。現在の TEX の配布パッケージでは，不足フォントの自動生成のため，TEX 導入時に METAFONT も導入し，パスも通すようになっていますので，起動しない場合，もう一度導入時のマニュアルなどを確認 — 友達にやってもらった人はその友達に相談 — して下さい。

[*4] `¥` だけでもかまいません。TEX と異なり，`¥relax` は `¥` と `relax` の 2 つの命令からなっていて，これらはともに「空」に展開されます。ここでの `¥` は「入力ファイルはないよ」程度の意味になります。

に変わったと思います。これも METAFONT のプロンプトで，* 1 つの場合は，コマンド入力要求です。これで準備完了です。

2 対話形式と online display

では，METAFONT に命令を打ち込んで「絵」を描かせてみましょう。プロンプト * に対して，

```
draw (15,0)..(0,75)..(15,150);
```

と入力 (最後に Enter キー — 以下も同様) し，さらに，

```
showit;
```

と入力すると，新しいウインドウが開くと思います。これを手前に表示させると，右図のように，上で指示した

```
draw (15,0)..(0,75)..(15,150);
```

に対応する曲線 (指定の点を通る曲線) が表示されていると思います。これが METAFONT の online display とよばれているものです。これでこれまでに描いた「絵」を確認できます。

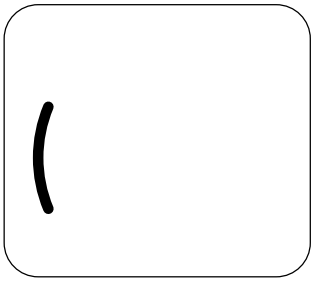


図 1.2.1

引き続き，次のように入力してみて下さい。各行入力後 Enter です。この例のように，META 言語では複数の文 (; で区切られた部分) を一度に入力できます。

```
draw (60,120)--(90,150)--(120,120); showit;
draw (240,120)--(270,150)--(300,120); showit;
drawdot (360,110); drawdot (360,40); showit;
```

showit; を指示するたびに，入力した図形要素が，online display に追加されていき，右図のように「絵」が完成していくと思います。

「絵」が完成したら METAFONT を終了しましょう。METAFONT を終了させるためには，プロンプト * に対して，

```
end.
```

と入力します。すると，

```
Transcript written on mfput.log.
```

と表示され，DOS 窓のプロンプトに戻るはずです。mfput というのは，ファイル名を指定せずに METAFONT を起動したときの，デフォルトのファイル名です。エディタなどで，カレントディレクトリ (実験用ディレクトリのはず) に生成されている mfput.log を表示させて，中身を確認してみて下さい。なお，end. の前に，

```
shipit;
```

とすると，今描いた「絵」をちゃんとフォントとして出力してくれます。ファイル名は mfput.2602gf で，今描いた「絵 = 文字」1 個だけを含んでいます。これを表示させる方法は，次講で説明する内容にかかりますので，ここでは省略します。

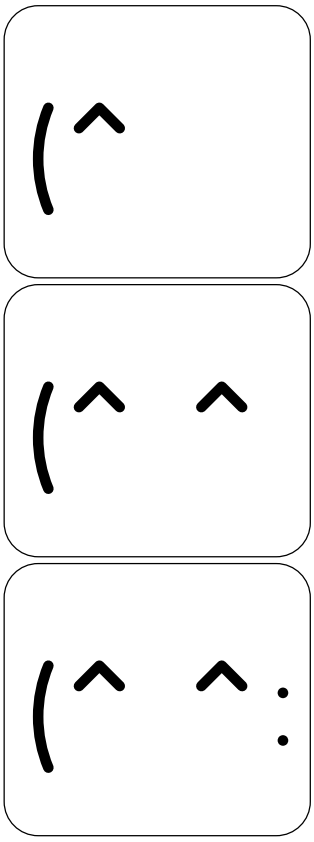


図 1.2.2

第 3 講 METAFONT の機能

1 METAFONT の作るファイル

METAFONT は，TEX ソースをタイプセットしてプレビューするのに必要な「フォント」を作るプログラムです。具体的には，次のようなファイルをはき出してくれます。

- 拡張子 .tfm のファイル (TFM[*1] ファイル) — 文字のサイズなどの情報が記される
- 拡張子 . **解像度** gf のファイル (GF[*2] ファイル) — 実際の文字の形が記される
- 拡張子 .log のファイル — ログ (実行時の各種メッセージの記録) が記される

GF ファイルには，文字の形状を解像度に応じてデジタル化したものが入っています。当然 (出力するプリンタ等の) 解像度に依存します。このファイルを使うのは，TEX そのものではなく，ビューアです[*3]。ただし，GF ファイルはサイズが大きいので，ビューアが直接扱うのは GF ファイルそのものではなく，適当な方法で圧縮したファイルになります。どういう圧縮形式を受け入れるかはビューアによって違いますが，一般的によく使われているのは，いわゆる PK ファイル — GF ファイルを GFtoPK というプログラムに渡して得られる，拡張子「. **解像度** pk」のファイル — です。少し意外に思う人もいるかもしれませんが，前述のように TEX 本体は GF ファイルや PK ファイルを利用しません。TEX は文字のサイズなどの情報から，文字を配置する場所を指定するだけで，その文字の形状がどんなものであるかには関与しないのです。したがって，TEX が利用するのは TFM ファイルだけです。TFM ファイルは解像度に依存しません。

以上をまとめると，以下のようになります。

- METAFONT が作る「フォント」とは TFM ファイルと GF ファイルのこと。
- TEX が使うのは TFM ファイル。
- GF ファイルは普通 GFtoPK を用いて PK ファイルにして使う。
- PK ファイルを使うのはビューア。
- 最終的に TFM ファイルと PK ファイルさえあれば，新しく作ったフォントを TEX ソースに盛り込み，タイプセットしてプレビューすることができる。他のファイル (GF ファイル，ログファイル) は消去してよい。

なお，最近の TEX やビューアは優れていますので，TFM ファイルや PK ファイルがなければ，METAFONT を自動的に起動し，これらのファイルを作ってくれた上に，しかるべき場所に保存までしてくれるようになっているのですが，この「フォント自動生成」の機能は，フォントを自分で作る際にはあまり有効に機能しません。なぜなら，フォントを作るためには，その仕上がり具合を目で確認し，修正を加えていく，という作業も当然必要

[*1] TEX Font Metric の略です。
[*2] Generic Font の略です。
[*3] TEX が出力する DVI ファイルを表示・印刷するプログラムです。dviout などが有名です。

となるのですが，フォントの自動作成ではすでにあるフォントは作成されませんので，修正を加えるたびに所定の場所に格納されたフォント（TFM と PK）を探し出して消すことになり，かえって面倒なためです。

したがって，実験用のディレクトリにソースを格納し，そこから手動で METAFONT を起動するほうが，フォントを作るには便利です。こうすれば，上記のファイルはすべて実験用のディレクトリにできますので，管理もしやすく，また，自動起動と違って，すでにファイルがあっても新しいファイルで上書きしてくれますので，更新にもとくに気を遣うことはありません。手動の面倒くささは，お使いのエディタのマクロ機能を使えばかなり軽減できるはずですので，エディタマクロを作るか，あるいは誰かが METAFONT 用のエディタマクロを作ってくれていないか探すといいかもしれません。

2 METAFONT のモード

METAFONT の作るファイルのうち GF ファイルは解像度等に依存しますので，METAFONT 起動時には解像度や出力装置に応じた各種パラメータを与えてやる必要があります。これらのデータは，プリンタの解像度・種類毎にまとめられ，`modes.mf` というファイルに格納されていますが，METAFONT は通常これをプリロードしていますので，ユーザーはモード名を METAFONT に伝えるだけでよくなっています。

たとえば，`mode=CanonCX;` とすれば，（キヤノンのレーザープリンタの）300 dpi[*4]用フォントを作るモードに設定されます。さしあたって dpi 数さえお持ちのプリンタに合わせれば印刷はできます（し，この本が目標にしているのは「フォント」というより「絵文字」ですので，プリンタ毎の細かい設定は実際にはあまり関係ありません）ので，以下の表の，必要な解像度（dpi 数）に対応したモード名を使えば十分だと思います。

dpi 数	モード名	dpi 数	モード名	dpi 数	モード名
96	`atarins`	118	`pcprevw`	160	`nectzo`
180	`lqlores`	200	`highfax`	240	`canonlbp`
300	`CanonCX`	320	`neclm`	360	`nechi`
400	`nexthi`	600	`canonex`	720	`esphi`
1200	`ultre`	2400	`supre`	特殊	`proof`

少し変わっているのが，フォントを「校正」するための `proof` というモードで，

- 2602 dpi という非常に高い解像度のフォントを作る。
- 校正段階なので TFM ファイルは生成しない。
- フォント名をタイトルとして GF ファイルに埋め込む。
- `labels` が指定されていればラベル（点の位置につけた標識）も埋め込む。
- これで作られた GF ファイルを `GFtoDVI` というプログラムに渡せば，DVI ファイル（ビューアで見ることができる）が得られ，その中には作ったフォントが gray フォントで大きく表示され，埋め込んだラベルも確認できる。

という，特殊な設定になっています。なお，モードを指定せずに起動したときは，`proof` モードでの起動になります。

[*4] dots per inch つまり 1 インチあたりのドット数です。

第 4 講 METAFONT でフォントを作る

1 実験の準備

実験用のディレクトリ (カレントディレクトリ) に生成された TFM ファイルと PK ファイルは，それぞれ TEX とビューアに認識されなければ意味がありません。

TEX がどこから TFM ファイルを探してくるかは，通常 TEX , METAFONT 等の設定ファイルである texmf.cnf に書き込まれている TFMFONTS で決まります。通常これは

```
TFMFONTS = .;{$TEXMF/fonts,$VARTEXFONTS}/tfm//
```

と設定されており，カレントディレクトリ (ピリオド . で表されます) の TFM ファイルを検索する設定になっていますので，とくに変更する必要はありません。

ビューアの設定は，ビューアによって違いますので，それぞれのマニュアルを確認して下さい。dviout の場合なら，メニューから [Option]→[Setup Parameters] を選び，[Font] ページの -TEXPK の先頭に .¥^s.^dpk; を追加すれば，カレントディレクトリを検索してくれます。— もちろん，すでにこの設定がある場合は何もしなくてかまいません。

2 ソースファイル・モードを指定した METAFONT の起動

METAFONT の命令を使ってフォントの形状等を記述したファイルを METAFONT ソースといい，通常拡張子を .mf にします。たとえば test.mf という METAFONT ソースがあるとき，

```
mf test  ( 拡張子の .mf は省略可)
```

の形で METAFONT を起動すれば，METAFONT は test.mf に書かれた命令を順に処理していきます[*1]。この場合，デフォルトの proof モードになります[*2]。proof モード以外のモード (たとえば CanonCX モード) で起動するには，ファイルの読み込みの前に，mode=CanonCX; を指示しなければなりません。ところで，前講では METAFONT を起動し，ファイル名要求 ** に対し ¥relax (¥ だけでもよい) を，コマンド入力要求 * に対し draw (15,0)..(0,75)..(15,150); や showit; などを入力していきましたが，これら一連の命令を，次のように起動時に一気に指定することもできます。

```
mf ¥draw (15,0)..(0,75)..(15,150); showit;
```

この仕様を利用して，モードとファイル名を同時に設定して起動するには，

```
mf ¥mode=CanonCX; input test
```

のようにします[*3]。

[*1] 通常 METAFONT ソースファイルは end で終わりますので，最後の行まで処理が終われば METAFONT も終了します。

[*2] 通常 METAFONT ソースはモードに依存しないように記述しますので，proof モードになります。特定のモードに依存するようにしたい場合は，ソースの先頭に mode=**モード名**; を記述します。

[*3] ¥ で命令を受け付けるようにし，mode に値を入れた後で，input 命令でファイル入力を指示しています。

3 proof モードの使い方

では，ここで実際にフォントを作ってみましょう。この本には，実験用に test.mf というファイルが同梱されています。これは (中身をのぞけば想像がつくと思いますが)，A，B 2 文字だけを含むフォントのソースです[*4]。これを実験用のディレクトリにコピーして下さい。

そして (Windows なら) DOS 窓を開いて，カレントディレクトリを実験用のディレクトリに移して，

```
mf test
```

と入力して下さい。METAFONT が起動し，ファイルを処理します。モードはデフォルトの proof です。2 文字しか入っていないので処理は一瞬で終わると思います。処理が終わったら，エクスプローラなどで実験用ディレクトリの中身を確認して下さい。test.log と test.2602gf という 2 つのファイルができていれば成功です。

次に，できたフォントを dviout 等のビューアで拡大表示させてみましょう。DOS 窓で，

```
gftodvi test
```

としてできた test.dvi をビューアで見れば，今作ったフォントが，gray フォントを使って拡大表示されていると思います。文字 A の定義には labels を埋め込んでいますので，1 ページ目のフォント (A に対応する左向きのマンボウ) には，描画に使った点の番号が書き込まれていると思います。

4 解像度に合わせたモード

次は，

```
mf ¥mode=モード名; input test
```

としてみましょう。「**モード名**」の部分はお持ちのプリンタの解像度に合わせて，**第3講 2項**の表から選んで下さい[*5]。test.log , test.tfm と test.〜gf (〜 は解像度の dpi 数) という 3 つのファイルができていれば成功ですので，さらに

```
gftopk test.〜gf
```

(〜 は解像度の dpi 数)

で，test.〜pk というファイルを作成して下さい。

このフォントを TEX で使ってみましょう。作ったフォントを TEX で使う最も簡単な方法は，TEX のプリミティブ ¥font を使う方法です[*6]。次のような TEX ソースを作って，実験ディレクトリに保存し，LATEX でタイプセットし，できた DVI ファイルをビューアで見て下さい。— うまく表示できましたでしょうか。

```
¥documentclass{jarticle}
¥font¥tes=test
¥begin{document}
A は {¥tes A}，B は {¥tes B} です。
¥end{document}
```

[*4] 正確には A , B の文字コードに割り振られた 2 文字です。文字の形が A や B の形をしているわけではありません。

[*5] モード名は大文字・小文字を間違えないよう入力して下さい。

[*6] もし，LATEX の ¥large などサイズ変更コマンドに対応しようとするなら，METAFONT ソース，TEX ソースともそれなりに設定が必要ですが，サイズ固定の「絵文字」なら ¥font で十分です。

第 5 講 METAPOST で画像ファイルを作る

1 METAPOST の起動方法

本講では METAPOST を使ってみることにします[*1]。

さて，METAPOST でも METAFONT と同様，対話的にコマンドを入力することができるのですが，online display に相当するものがありませんので，まずはソースファイル名を起動時に与える方法を試してみましょう。この本には，実験用に `test.mp` というファイルが同梱されています。これは (中身をのぞけば想像がつくと思いますが)，2 つの画像を含む METAPOST ソースです。これを実験用のディレクトリにコピーして下さい。

METAPOST の実行プログラム名は `mpost` であることが多いようです。もちろん違う名前で移植されることもあり得ますし，TEX の配布セットによっては含まれていない (OS によっては移植されていない) こともありますので，うまくいかないときは，配布セットの内容を確認してみて下さい。

では起動してみましょう。(Windows なら) DOS 窓を開いて，カレントディレクトリを実験用のディレクトリに移して，

```
mpost test  ( 拡張子の .mp は省略可)
```

と入力して下さい。METAPOST が起動し，ファイルを処理します。エクスプローラなどで確認して `test.log` , `test.1` , `test.2` というファイルができていれば成功です。拡張子 `.log` のファイルはログファイルです。残りの拡張子が数字のファイルが，生成された画像ファイルです。METAFONT と異なり，画像は 1 つずつ異なるファイルに格納されます。ファイルの拡張子の数字は，ソース中に記述した画像番号が使われています。

先に説明しましたように，この画像ファイルは PostScript という言語で記述されており，PostScript に対応したプログラムあるいはプリンタで，表示もしくは印刷できます。また，EPS ファイル[*2] として TEX ファイルに「貼り付け」ることもできます。

たとえば Windows の場合，GSview というプログラムで PostScript ファイルを表示できます。GSview を起動し，メニューから `[File]`→`[Open]` を選んで，`test.1` または `test.2` を選択して下さい[*3]。うまく「マンボウ」の絵が表示されたでしょうか。

2 画像を TEX で使う

METAPOST で作成した PostScript ファイルを TEX ファイルに貼り付けるには，LATEX の標準パッケージである `graphicx.sty` を利用すれば便利です。このパッケージ

[*1] METAPOST によって作られた「画像」を TEX ファイルに「貼り付け」る以外の使い方をしないならば，次講で説明する MEPOLEX が使えれば十分ですので，本講は読み飛ばして下さって結構です。

[*2] Encapsulated PostScript ファイルの略です。

[*3] ダイヤログの下方にある「ファイルの種類」は `All Files(*.*)` にして下さい。さもないと，拡張子が数字のファイルは表示されません。

には ¥includegraphics というコマンドがあります。これによって METAPOST で作成したファイルを EPS ファイルとして TEX ファイルに貼り付けてみましょう。

まず，¥includegraphics は拡張子でファイルの種別を判定しますので，画像ファイルの拡張子を .eps に変更します。たとえば次の通りです。

```
test.1 ⟶ test1.eps
test.2 ⟶ test2.eps
```

単に拡張子を変えるだけでは 2 つのファイル名が同じになってしまうため，ベース名の方にも手を加えました。上のような変更を加えた上で，以下のような TEX ソースを作り，実験用ディレクトリに保存して，TEX でタイプセットして下さい。

```
¥documentclass{jarticle}
¥usepackage{graphicx}
¥begin{document}
これが 1 つ目の画像です。¥includegraphics{test1.eps}

これが 2 つ目の画像です。¥includegraphics{test2.eps}
¥end{document}
```

できた DVI ファイルを，EPS 画像の貼り付けに対応したビューア (dviout など) でプレビューすれば，画像が貼り付けられていることがわかると思います。

3 連立方程式を解かせる

META 言語が画像の描画において優れている点として，描画のために必要な点を予め計算しなくても，METAFONT や METAPOST に点の決定条件を与えるだけで自動的に計算してくれる，というものがあります。わかりやすい例で言うと，2 直線の交点をもとに新たな図形要素をつくるとき，その 2 直線を与えるだけで交点を計算してくれます。これは，つきつめて言えば，連立 1 次方程式が解けるということでもあります。「直線の交点」となるとそれなりに文法を説明しなければなりませんので，ここでは右の単純な連立方程式を解かせてみることにしましょう[*4]。

$$\begin{cases} v+w+x+y=2 \\ 2v+w-3x-y=3 \\ 2v+3w-5x+y=-3 \\ 3v+2w+x-y=15 \end{cases}$$

まず，METAPOST または METAFONT を，ファイル名を与えずに起動して，ファイル名入力要求 ** に対して ¥relax を入力し，コマンド入力プロンプト * を表示させて下さい。その上で，次のように上記の連立方程式を入力していきます。

```
v+w+x+y=2; 2v+w-3x-y=3; 2v+3w-5x+y=-3; 3v+2w+x-y=15;
```

この 4 つの式を入力した時点で，META 言語はすでに上記の連立方程式を解いて，それぞれの解を変数の値としてもっています。したがって，

```
show x;
```

などと入力すれば >>2 のように，それぞれの変数の値 (解) が表示されます。

[*4] z は META 言語ではマクロ名に使われていますので，それ以外の文字を使っています。

第 6 講 MEPOTEX の使い方

1 オリジナルモードと隠蔽モード

MEPOTEX は拙作のパッケージで，TEX ソース中に METAPOST ソースを埋め込めるようにしたものです。すでに説明しましたように，MEPOTEX では,「埋め込まれた METAPOST ソースの抽出 → METAPOST の実行 → 生成された PostScript ファイルの取り込み」を自動的に行うため，組版のために必要な操作は通常の LATEX ソースに対する操作と同じだけで済みます。

MEPOTEX には，ver 1.00 で導入された「オリジナルモード」と，ver 2.00 で導入された「隠蔽モード」の 2 つのモードがあります[*1]。

「オリジナルモード」は TEX ソース中に，生のままの METAPOST ソースを書ける領域を提供します。具体的には，MPpic 環境という picture 環境を拡張したものが用意されており，その中の ¥openMP と ¥closeMP ではさまれた領域を，METAPOST ソースへの抽出対象とします。

```
¥begin{MPpic}<1cm,2cm>(5,5)(0,0)
¥openMP
( この部分に，生のままの METAPOST ソースを書けます。)
¥closeMP[*2]
( この部分には picture 環境類似のマクロが使えます。)
¥end{MPpic}
```

生のままの METAPOST ソースを書けるので，META 言語の勉強にはこちらの方が向いていると思います。

一方,「隠蔽モード」は，生のままの METAPOST ソースを極力隠蔽し，METAPOST の細かい文法をとくに意識することなく図を作成するモードです。具体的には，(MPpic 環境用に作られた) TEX マクロを MEPOTEX が METAPOST ソースに展開し，それを METAPOST に渡して処理させています。円や直線など簡単な図形については，META 言語の文法をほとんど知らなくても描けるという強みがあります。また，文字列の図へのはめ込みなど，TEX ソースとの融和性も強くなっています。もちろん，複雑な処理をさせるために，生のままの METAPOST ソースを織り交ぜることも可能です[*3]。「隠蔽モード」は METAPOST を隠蔽するという仕様ですので，META 言語の勉強には向かないかもしれませんが，初心

[*1] ver 2.00 以降では両方使えます。

[*2] ¥closeMP は，単独で行頭から書かねばなりません。

[*3] ¥sendMP{〜} とすれば 〜 の部分が「そのまま」METAPOST に送られます。したがって，生のままの METAPOST ソースを転送したいときは，ここに書き込みます。ただし，# など TEX と METAPOST で意味の違う特殊文字の一部は使えません。これらを使うには,「オリジナルモード」でなければなりません。

者には負担が少ないという利点があり，上級者にとっても，複雑な図を描く上では，ある程度単純な図形をマクロ化してある方が便利ということから，結構重宝すると思います。

2 MEPOTEX のサンプル

METAFONT ソース，METAPOST ソースに続いて，MEPOTEX 用のソースのサンプルも用意しました。この本に同梱の test.tex というのがそれです。これを，通常の TEX ソースとして pLATEX 2ε で処理して下さい。生成された DVI ファイルを，EPS 画像貼り付けに対応したビューア (dviout など) でご覧になれば，画像が確認できると思います。うまくいけば，おおよそ下のような図が得られると思います[*4]。

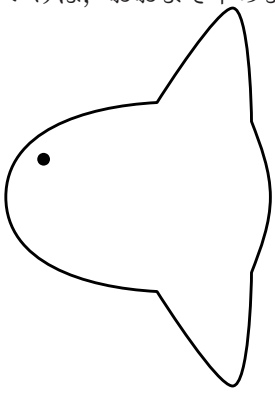


図 1.6.1　左向きマンボウ

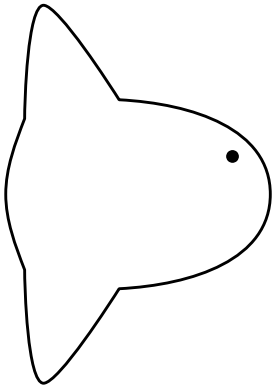


図 1.6.2　右向きマンボウ

参考までに，上図左のソースを挙げておきます。図の定義部分 (¥openMP と ¥closeMP ではさまれた部分) は，前講の METAPOST 用サンプルやその前の METAFONT 用サンプルとほぼ同じです[*5]。

```
¥begin{MPpic}<35mm,50mm>(1,1|0.15)(0,0)% オリジナルモード
¥openMP
  numeric d; d=0cm;
  w-x0=x4=0;    y0=y4=0.5[-d,h];
  x1=x7=13/14w; y1-y0=y0-y7=0.2(d+h);
  x2=x6=6/7w;   y2-y0=y0-y6=0.5(d+h);
  x3=x5=4/7w;   y3-y0=y0-y5=0.25(d+h); z8=(1/7w,0.1(d+h)+y0);
  pickup pencircle scaled 1.2pt;
  draw z1{up}..tension3..{left}z2..tension3..{curl0}z3..z4..z5{curl0}..
       tension3..z6{right}..tension3..{up}z7{curl0}..z0..{curl0}cycle;
  pickup pencircle scaled 4.8pt;
  drawdot z8;
¥closeMP
¥mptZu(0,0)[tl]<0mm,-1mm>{左向きマンボウ}{サンプル 1}
¥end{MPpic}
```

[*4]ただし，サンプルと紙面の幅が違うため，図の間隔や図番号は違います。

[*5]ただし図の大きさは，METAPOST 用サンプルが上図の 2 倍，METAFONT 用サンプルが約 0.077 倍です。— METAFONT の方は「フォント」なので小さいです。

第 7 講

必要最小限の文法 (1)

本講からしばらく，METAFONT 言語ソースを読む上で必要な文法を，できるだけ内容を絞って紹介したいと思います。ただし「必要最小限」とは書いてあっても,「これを知らなきゃ先に進めない」という意味ではありません。文法をある程度系統的に解説すると，どうしてもリファレンス的になりますので,「使いながら覚える」主義の人には退屈きわまりないものでしょう。そういう場合は，項目だけ確認しておき，本文は流し読みしておき，次の**第2部**（実例をもとにした解説）に入ってから，必要な際に戻ってくるという形でもいいと思います[*1]。

1 文

METAFONT 言語が TEX と根本的に違う点は，〈文〉という単位からなっているという点です。ここでいう〈文〉とは「文章」の意味ではなく，1 つのコマンド（と引数）または数式だけからなる基本単位のことです。TEX は，文字列（正確にはトークン）の並びを処理するため，文字や命令をどんどんつなげて記述しますが，METAFONT 言語では，1 つの〈文〉にひとつながりの命令または数式しか書けません。METAFONT 言語の〈文〉は ;（またはファイル先頭）から ;（またはファイル末尾）までの部分のことで,

```
s=3; t=4;
```

はたとえ 1 行に書かれていても 2〈文〉

```
pickup pencircle
       scaled 1pt;
```

は 2 行にわたっていても 1〈文〉と見なされます。METAFONT 言語を使い始めたときは，; のつけ忘れが多いと思いますので，ご注意下さい。

2 マクロ

ワープロに比べて TEX が優れているところといえば，強力で柔軟なマクロの存在でしょう。METAFONT 言語もマクロを組めますので，一旦マクロを作ったら，複雑な式や図形も，次からはマクロの呼び出しだけですませることができます[*2]。

マクロに対して，METAFONT 言語にもとから備わっている命令をプリミティブといます。マクロはプリミティブを組み合わせて，あるいは，別のマクロを組み合わせて作られます。（TEX と同じく）普段目にする METAFONT 言語の命令の多くはマクロです。

マクロは，それを使うソースファイル内で定義してもかまいませんが，よく使うマクロや多量のマクロ定義は，通常別ファイルにし，`input` 命令でソースファイルに読み込みます。マクロを保存するファイル名にはとくに制約はありませんが，METAFONT なら拡張子

[*1] **第2部** も，そのつもりで，かなり詳しく文法を解説しています。なお，より詳しい文法については，**第3部** を参照して下さい。

[*2] もっと言うと，誰かにマクロを作ってもらえば，労せずしていろんな機能を METAFONT 言語に付加できます。

を .mf に，METAPOST なら拡張子を .mp にしておくと，input のとき拡張子を省略できます[*3]。たとえば，MyMacro.mp というファイルを METAPOST ソースに読み込むときは，

```
input MyMacro;
```

とします。なお，MEPOTEX では METAPOST の標準的なマクロに加え，さらにいくつかのマクロファイルを自動で読み込むようになっています。

マクロの作り方や使い方については，おいおい具体例とともに説明していきますので，ここではマクロというものが存在するということを覚えておいて下さい。

3 変数のタイプと宣言

次に，META 言語で扱える変数の種類を説明します。METAFONT , METAPOST 共通で使えるのは，**numeric**（**数値**）, **pair**（**点** または **ベクトル**）, **path**（**経路**）, **pen**（**ペン先**）, **picture**（**画像**）, **boolean**（**論理値**）, **transform**（**変形**）, **string**（**文字列**）の 8 つのタイプで，METAPOST にはさらに **color**（**色**）というタイプもあります。

上記の各タイプの変数を使うには，まずタイプの「宣言」を行わなければなりません。たとえば，a を **pair** 変数として使うためには，予め，

```
pair a;
```

と宣言しておかねばなりません。宣言方法はこの例からもわかると思いますが，

⟨ タイプ名 ⟩⟨ 変数名 ⟩;

です。同タイプの変数を複数同時に宣言したいときは，

```
pair a,b,c;
```

のように，コンマで区切って列記することもできます。特に宣言せずに ⟨ 変数名 ⟩ を用いると **数値** タイプと解釈されます。

⟨ 変数名 ⟩ には，a1 , a2 のように **数値** の ⟨ 添字 ⟩ を付けることもできます。また，a1b , a2b のように **数値** と文字の ⟨ サフィックス ⟩ を付けることもできます。文字だけの ⟨ サフィックス ⟩ を付けるときは a.b のように，ピリオドで区切ります。

さて，**数値** の ⟨ 添字 ⟩ がない変数を宣言するときは，上の例と同様，

```
pair a.b;
```

などと宣言すればよいのですが，**数値** の ⟨ 添字 ⟩ の付いた変数を使いたいときは，

```
pair a[] , a[]b;
```

のように，**数値** の部分を [] に代えて宣言しなければなりません。これは，「**数値** の ⟨ 添字 ⟩ だけ異なる変数は，通常同一タイプの変数列として使う」ことや，「その個数は宣言時に特定できないことがある」ことを考えれば妥当な仕様です。上の例に挙げた宣言で，a1 , a2 , a3 , ... や a1b , a2b , a3b , ... がすべて宣言できたことになるからです。

数値 の ⟨ 添字 ⟩ は [⟨ 数値式 ⟩] でもかまいません。たとえば，a[3+2] は a5 と，a[1] は a1 と同じ意味です。

ちなみに z[n] と zn と z.n は違います。前者は z に ⟨ 数値式 ⟩ の ⟨ 添字 ⟩ n を付けたもの，中央のものは zn というひとかたまりの変数，後者は z に ⟨ サフィックス ⟩ として文字 n を付けたものという意味になります。z1a と z1.a と z[1]a と z[1].a は同じ意味です。z.1 は z[0.1] であり，z1 ではありません。

[*3] LaTeX と違い，マクロファイルとソースファイルの拡張子にとくに区別をつけていません。

第 8 講

必要最小限の文法 (2)

本講と次講でそれぞれの変数の意味を説明します。本講ではまず，もっともよく使われる **numeric** , **pair** , **color** , **path** について説明し，残りは次講にまわします。

1 numeric, pair, color, path の意味

○ **numeric** は数値データを扱うタイプです。この本では **数値** と太字で表すこともあります。META 言語で扱える **数値** は案外桁数が少なく，絶対値が 4096 未満の実数値 (正確にはそれを $\frac{1}{65536}$ の整数倍の値に丸めたもの) に限定されます。

○ **pair** は（**数値**, **数値**）の値をもつ変数タイプです。平面上の点やベクトルを表すのに使う形式です。たとえば，**pair** 値変数 a に対して，`a:=(1,2);` とすれば，a は座標 (1 , 2) をもつ点，または，成分 (1 , 2) をもつベクトルとして使えます。この本では，使用目的に応じて **点** または **ベクトル** とよび分けることもあります。

○ **color** は（**数値**, **数値**, **数値**）の値をもつ変数タイプです。rgb 形式で色を表したときの赤，緑，青の各成分をそれぞれ 0 〜 1 の数値で表し，この順に並べたもの，というのが元来の意味で，METAPOST で色を指定するのに使われます。したがって，この本では **色** と太字で表すこともあります。

color は，形式上 3 次元のベクトルであり，各成分も (色を表すだけならば 0 〜 1 だけでいいはずですが) **数値** と同じ範囲の数を使えますので，空間ベクトルとして転用することも可能です。ただし，本来の使い方ではありませんので，デフォルトでは加減とスカラー倍以外ほとんど演算が定義されていません。

○ **path** は，**点** を曲線または線分でつないだもので，線を引いたり領域を塗りつぶしたりする際の基本単位となります。

たとえば，座標 (1 , 2) , (3 , 1) , (5 , 4) の 3 **点** を曲線で結ぶなら，

```
(1,2)..(3,1)..(5,4)
```

線分 (折れ線) で結ぶなら

```
(1,2)--(3,1)--(5,4)
```

と指定します。もちろん,「通過点」には **点** つまり **pair** 値の変数も使えます。

なお,「塗りつぶし」を行うには，閉じた **path** が必要です。閉じた **path** を，METAFONT 言語ではとくに **cyclic path** とよんでいます。**cyclic path** を作るには，

```
(1,2)..(3,1)..(5,4)..cycle
```

のように，最後を `cycle` で閉めます。この本では，**path** を **経路**，**cyclic path** を **巡回経路** とよぶこともあります。

2 定義済みの numeric, pair, color, path 値定数

METAFONT 言語では，いくつかの定数を定義してあります。ここでは **numeric** , **pair** , **color** ,

path について，よく使われるものを挙げます。

○ **numeric** の定数としては，METAFONT 言語で扱える最小の正数 $\frac{1}{65536}$ を表す `epsilon` , 最大の正数 $4096-\frac{1}{65536}$ を表す `infinity` , `epsilon` の 32 倍を表す `eps` などが挙げられます。あと，意外かもしれませんが，METAFONT 言語で使える `mm` などの単位も **numeric** の定数です。これについては **第10講 3項** で述べます。

METAFONT では，文字の定義の際 (`beginchar` と `endchar` の間) 文字の幅・高さ・深さを，`w` , `h` , `d` に設定します。METAPOST にはこれらの定数は設定されません。MEPOTEX では METAFONT と METAPOST の橋渡しのつもりで，`w` と `h` に (`MPpic` 環境に入るときに指定する) 横単位長 , 縦単位長 を設定します。

○ **pair** の定数としては，`(1,0)` を表す `right` , `(-1,0)` を表す `left` , `(0,1)` を表す `up` , `(0,-1)` を表す `down` , `(0,0)` を表す `origin` などが挙げられます。

○ **color** の定数としては，`(0,0,0)` を表す `black` , `(1,1,1)` を表す `white` , `(1,0,0)` を表す `red` , `(0,1,0)` を表す `green` , `(0,0,1)` を表す `blue` などが挙げられます。なお，MEPOTEX ではこのほかに，LATEX 2_ε の標準パッケージの `color.sty` (`usenames` オプション指定時) と同じ 68 色を使えるよう，定数を定義しています。

○ **path** の定数としては，`(0,0)--(1,0)--(1,1)--(0,1)--cycle` に相当する 1 辺の長さ 1 の正方形 `unitsquare` や，原点中心で直径 1 の円 `fullcircle` などが挙げられます。

3 関　数

関数という言い方は正確ではないかもしれません。正しくは単項演算子と二項演算子ということになるのでしょうが，いわゆる初等関数の METAFONT 言語上での表現ですので，このようなまとめ方にしました。ここでは，METAFONT 言語で使える「関数」のうち非常によく使うものを挙げておきます。以下の表で，x , y は **数値** , z は **ベクトル** (**pair**) です。このほかにも使える関数は多数ありますが，それらについてはたとえば，METAFONT ブックなどを参照して下さい[*1]。

関　数	意　味	関　数	意　味
`+ - * /`	順に和・差・積・商	`x**y`	x^y
`x++y`	$\sqrt{x^2+y^2}$	`x+-+y`	$\sqrt{x^2-y^2}$
`sqrt(x)`	$\sqrt{x}$	`abs(x)`	**数値** x の絶対値
`abs(z)`	**ベクトル** z の長さ	`length(z)`	**ベクトル** z の長さ
`sind(x)`	$\sin x$ (x は角度)	`cosd(x)`	$\cos x$ (x は角度)
`dir(x)`	x 度方向の単位**ベクトル**	`angle(z)`	**ベクトル** z の方向角
`xpart(z)`	**ベクトル** z の x 成分	`ypart(z)`	**ベクトル** z の y 成分
`mlog(x)`	$256\ln x$	`mexp(x)`	$e^{x/256}$

[*1] 具体的な数値と文字の積，例えば `1.2*h` の場合，`*` を省略して `1.2h` と書くことができます。(実はこれも演算子の一種です。) また関数の後ろの ( ) を省略して，`dir40` のような書き方もできますので，より「自然な」数式が書けると思います。省略した際，どの演算が先に行われるかは，METAFONT 言語の演算の「優先順位」にしたがって判断されます。このあたりについては，**第3部 第2講 1項** で詳しく説明します。

第 9 講

必要最小限の文法 (3)

1 pen, picture, boolean, transform, string の意味

○ **pen** は線を引く際のペン先の形状を格納するための変数です。この本では，**ペン先** とよぶこともあります。**pen** 値変数には，ユーザーが **path** タイプの値をもとに自由な形状を設定できますが，実際には，METAFONT 言語のプリミティブか基本マクロで定義されている **pen** 変数を，そのまま，あるいは変形して使うのが普通です。なお，現在使用中の **ペン先** は currentpen という変数に格納されています。ペン先を変更する際は，

```
pickup ペン先;
```

のようにマクロ pickup を使う方が，currentpen に直接値を設定するより一般的です。

○ **picture** は，画像を保存する変数ですので，この本では，**画像** とよぶこともあります。よく使うのは，現在描画中の **画像** を保存している currentpicture です。なお，**画像** の文法的な位置づけは METAFONT と METAPOST で同じなのですが，出力形式の違いから，**画像** に適用できる操作に違いがあります。

○ **boolean** は true か false のいずれかの値をとる変数タイプで，if 文や for 文の制御によく使われます。ただし，変数として使われるより，**boolean** 値をとる式のままで使うことが多いと思います。たとえば，x+3>5 という式は，x=1 なら false , x=3 なら true という値をとります。この本では，**論理値** とよぶこともあります。

○ **transform** は，数学で言う affine 変換，つまり，1 次変換 + 平行移動を値にもつ変数タイプです。affine 変換というのは

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} p \\ q \end{pmatrix}$$

という変換で，a , b , c , d , p , q という 6 つの **数値** で特徴づけられます。…… といわれても，数学が苦手な人にはピンと来ないかもしれませんが，たとえば，拡大縮小や回転や平行移動といった処理を，**点** や **経路** や **ペン先** や **画像** に行うものです。この本では，**変形** とよぶこともあります。

変形 を使う際は，上記の 6 つの **数値** を設定して使うより，METAFONT 言語のプリミティブか基本マクロで定義されている ⟨ 変形演算子 ⟩ を，合成等して使うのが普通です。

○ **string** は文字列を格納する変数です。この本では，**文字列** とよぶこともあります。外部ファイルに **文字列** を書き出したり，そのファイル名を管理したり，METAPOST で画像にフォントを埋め込んだりするのに使います。

2 定義済みの pen, picture, boolean, string 値定数・プリミティブ

○ **pen** の定数，つまり，予め用意されている **ペン先** の形状としては，直径 1 の円である pencircle , 1 辺の長さ 1 の正方形である pensquare , 長さ 1 の横向きの線分 (厚み

0) である penrazor などが挙げられます[*1]。実際にはこれらを **変形** して使います。

○ **picture** のプリミティブとしては，空っぽの画像 nullpicture などがあります。

○ **boolean** は，普通 ⟨ 式 ⟩ の形で使うことが多いので，定数・プリミティブは少なく，プリミティブの true , false くらいしかありません。

○ **string** の定数・プリミティブもあまりなく，プリミティブの jobname が現在のソースファイルのベースネームに設定されている程度です。

3 定義済みの ⟨ 変形演算子 ⟩

⟨ 変形演算子 ⟩ は **pair** , **path** , **pen** , **picture** に作用する変換で，META 言語のプリミティブや基本マクロには以下のようなものがあります[*2]。

ソース	意　味
Obj scaled t	*Obj* を原**点**中心に t 倍に拡大縮小。
Obj xscaled t	*Obj* を (y 軸を中心に) x 軸方向に t 倍に拡大縮小。
Obj yscaled t	*Obj* を (x 軸を中心に) y 軸方向に t 倍に拡大縮小。
Obj zscaled z	*Obj* に z を複素数と見てかけ算したもの。
Obj shifted z	*Obj* を **ベクトル** z だけ平行移動。
Obj rotated t	*Obj* を原**点**中心に t 度回転。
Obj rotatedaround (z,t)	*Obj* を **点** z のまわりに t 度回転。
Obj reflectedabout (z,w)	*Obj* を 2 **点** z , w を結ぶ直線に関して対称移動。

通常はこれらの合成でこと足りますので，ここではこれ以上深入りしませんが，これをさらに一般化したものが **transform** 値ということになります。**transform** 値の扱いについては **第3部 第7講** を参照して下さい。

4 補　間

META 言語では，**numeric** , **pair** , **color** , **path** の「補間」ができます。

まず，**pair** の場合。これは分点といった方がわかりやすいと思います。たとえば，2 **点** A , B (対応する **pair** 値 A , B) に対し，線分 AB を $t:(1-t)$ に分ける **点** (分点) は

```
t[A,B]
```

で表せます。別の言い方をすれば，この分点は，A を原点，B を座標 1 とする座標軸に対して，座標 t の **点** を表します。t は 1 より大きい値や負の値も使えますので，実数 t を変化させることによって直線 AB 上のすべての **点** を分点で表現できます。

同様の表現は，A , B がともに **numeric** のときや，ともに **color** のときにも使えます。前者は **数値** を数直線上の点と，後者は **色** を 3 次元空間の点と見たときの分点といえます。

path の場合は，interpath というマクロを使って，次のように表現します。

```
interpath(t,p1,p2)
```

この例は，2 つの **経路** p1 , p2 を構成する **点** を対応させ，p1 上と p2 上の対応する 2 **点** を $t:(1-t)$ に分ける **点** を結んで，新たに作った **経路** を得る式です。

[*1] 正確には pencircle はプリミティブで，pensquare と penrazor は **pen** 値定数です。

[*2] 表において，z , w は **pair** , t は **numeric** , *Obj* は変形を受けるもの (**pair** , **path** , **pen** , **picture**) です。ただし，METAFONT では **画像** がビットマップなので，**画像** の **変形** には一部制限があります。

第 10 講

必要最小限の文法 (4)

1 値の設定 — 代入と等置

META 言語では変数に値を設定するとき，代入と等置という 2 つの方法が使えます。

まず代入ですが，これは普通のプログラム言語と同様の設定方法で，何らかの式を計算して得られる値，あるいは具体的な値を変数に格納します。META 言語では := を使って代入を表します。すなわち，

代入先の変数 := **式や値**;

のように表記します。たとえば，**数値** 変数 a , b , c があり，a に b+c の計算結果を代入するには，

```
a:=b+c;
```

とします。代入の際には右辺に左辺の変数名を使うことも可能です。たとえば，

```
a:=a+1;
```

は,「a の古い値に 1 を加え，それを a の新しい値に設定せよ」という意味になります。

次に等置ですが，これは **第5講 3項** でも見たように連立方程式の形で条件を与え，それによって値を設定させる方法です。これは図形を描く上で非常に有効な設定方法です。META 言語では = を使って等置を表します。たとえば，**ベクトル** z0 を，**ベクトル** z.x , z.y の 2 方向に分解したいとき (**図 1.10.1** 参照) — すなわち，z.x 方向の **ベクトル** z0x と z.y 方向の **ベクトル** z0y で z0=z0x+z0y をみたすものを得たいとき —

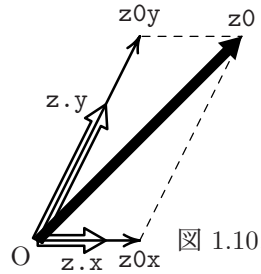


図 1.10.1

```
z0x+z0y=z0; z0x=s*z.x; z0y=t*z.y;
```

の 3 つの式で z0x と z0y を設定することができます。なお，上で s と t は一時的に使っただけで，いわば「使い捨て」の **数値** なのですが，これはもったいないので，普通は

```
z0x+z0y=z0; z0x=whatever*z.x; z0y=whatever*z.y;
```

とします。whatever は毎回異なる「使い捨ての」定数を生成してくれます。

等置も代入と同様，両辺に同じ変数が入っていてもかまいませんが，等置はいわゆる普通の方程式と同じですので，a=a+1 は「解なし」のエラーになります。また，META 言語が解けるのは連立 1 次方程式だけですので，未知数の高次式や分数式が出ないように注意する必要があります。これらが等置に関する制限です。

一方，代入についての制限としては,「マクロによって生成される値**に**何かを代入することはできない」[*1]というものがあります。細かい話になりますが，**点** を表すのに META 言

[*1] マクロによって生成される値を等置したり，他に代入したりはできます。

語でよく使う z0 , z1 , etc は，**pair** 値変数ではなく，**pair** 値を返す「マクロ」[*2]です。したがって，これらに値を設定するには等置を用いねばなりません。

2 for と if

METAFONT 言語では，一般のプログラム言語と同様，for 文や if 文を使えます。

少し例を挙げますと，

```
for n=0 step 5  until 360: z[n]=dir(n); endfor
```

とすれば，n=0 から始まって，n の値を 5 ずつ増やしながら，360 を越える直前まで z[n]=dir(n); が実行されます。したがって，

```
z0=dir(0); z5=dir(5); ..... ; z360=dir(360);
```

が実行されたのと同じ結果を得ることができます。

```
if pair c       : z0=c;
elseif numeric c: z0=c*right;
else            : z0=(0,0);
fi
```

この例では c が **pair** 値変数のときはそれをそのまま，**numeric** 値変数のときは right つまり (1,0) をかけてから z0 に設定しています。それ以外のときは z0 に (0,0) を設定しています。

METAFONT 言語で特徴的なのは，for や if で繰り返し・分岐するのは，完結した〈文〉でなくてもよい，ということです。次の例を見て下さい。

```
draw z0 for n=1 upto 4: ..z[n] endfor;
```

この例では n が 1 から 4 まで 1 ずつ増える間，..z[n] が繰り返されて，その結果，

```
z0..z1..z2..z3..z4;
```

という **経路** が得られます[*3]。

3 単位について

METAFONT 言語では pt とか mm の，おなじみの単位が使えますが，実はこれは単位に見えても単位ではありません。(TEX もそうですが) これらは実は **数値** 定数です。試みに，

```
a := mm;
```

などとすると，変数 a に mm のもっている数値が代入されます。

この pt や mm のもっている **数値** が何を表しているかといいますと，METAFONT 言語それぞれがもっている「デフォルトの単位」との比率です。デフォルトの単位は METAFONT で

[*2] z〈サフィックス〉は (x〈サフィックス〉,y〈サフィックス〉) に展開されるマクロです。この定義のおかげで「z1 の代わりに x1 と y1 を設定しても同じこと」になりますので，より柔軟な点の指定が可能になります。代入ができないのはその副作用と思って割り切って下さい。もし，どうしても代入したいときは，z1 に代入するのではなく，**numeric** 値変数の x1 , y1 に個別に代入することになります。

[*3] セミコロンの位置に注意して下さい。上の方の例では各繰り返し毎に〈文〉が終わるので，for 〜 endfor の中にセミコロンが，下の方の例では各繰り返しが完了して初めてひとつの〈文〉になるので，for 〜 endfor の後にセミコロンがつきます。

はピクセル[*4]，METAPOST では bp[*5] です。

たとえば，METAPOST で draw (0,0)--(36,0); とすれば，36 = 36bp = 0.5in なので，原点 (0,0) から右に 0.5 インチの線分を描きます。1 インチは 72bp ですので，in には 72 という数値が格納されており，0.5in は $0.5 \times 72 = 36$ となり，36 と同じ意味をもちます。もちろん 36 より 36bp や 0.5in の方が，人間にとっては意味がはっきりして見やすいソースになります。

METAFONT ではピクセルが解像度依存なので，上記と同じ線分を描きたいときは

300 dpi のフォントを作るときは draw (0,0)--(150,0);

600 dpi のフォントを作るときは draw (0,0)--(300,0);

となります。これも draw (0,0)--(0.5in,0); と書けば，見やすく，しかも解像度に依存しない (依存を吸収した) 書き方になります[*6]。

また，METAFONT には pt# のように # つきの「単位」もあります。これは，フォントサイズなど解像度に依存してはいけない長さに使います。その正体は，1 ポイント (1pt#) をデフォルトの単位としたときの比率です。(したがって METAPOST の単位は METAFONT の # つき単位に近い意味をもつことになります。) 「単位」だけでなく変数にも # つきと # なしがあり，前者が解像度に依存しないもの，後者が解像度に依存するもの，という使い分けをします。ただし，pt# などの「単位」は自動的に pt に対応づけられますが，自分で定義した # つき変数を # なし変数に対応づけるには，define_pixels など[*7]の命令 (マクロ) を使う必要があります。

(例) define_pixels(u,v,w) — u#,v#,w# から u,v,w を作る。

なお，上の説明からわかると思いますが，1pt# あたりのピクセル数を hppp[*8] とすれば，

(# なし変数) = (# つき変数) × hppp

となります。define_pixels などのマクロでは，#なし変数に hppp をかけて，これを整数に丸めています。

[*4] プリンタの解像度が 300 dpi というのは，1 インチ あたり 300 個の小さい四角が (縦横に) 並び，それを白または黒に塗って文字や図形を表現すると言うことを意味します。この小さい四角のことを本来ピクセルとよぶのですが,「この四角が何個並んでいるか」という意味で，長さの単位にも使います。たとえば 300 dpi のとき，450 ピクセル = 1.5 インチ になります。

[*5] 72 bp = 1 in です。in はインチです。

[*6] なお，METAFONT では，最初に dpi の違いを各定数に反映させる必要があります。これをやってくれるマクロが mode_setup です。

[*7] 端数処理の仕方によって何種類かあります。

[*8] これも mode_setup マクロで設定されます。

第 2 部 実践編(初級〜中級)

ここでは実際に META 言語でソースファイルを作成し,
画像を表示させてみましょう。
必要な文法知識は適宜補足していきますので,
とくに細かい知識は必要ではありません。
必要ならば **第1部** にさかのぼって,
関連する内容を見て下さい。

第 1 講 ソースファイルの書き方

1 ソースファイルの共通部分

本講はまず簡単な例ということで,「天気記号：雪」を描いてみます。ご存知の方も多いと思いますが，**図 2.1.1** のような記号です。まずはこの図をじっくりご覧下さい。METAFONT 言語で図を描くには，

- **点** を設定する。
- **点** を直線か曲線で結んだ **経路** を設定する。
- **経路** を (draw などの命令で) 描く。

といったステップを踏むことになります。この部分は METAFONT , METAPOST , MEPOTEX で全く共通なので，先に説明しておきます。

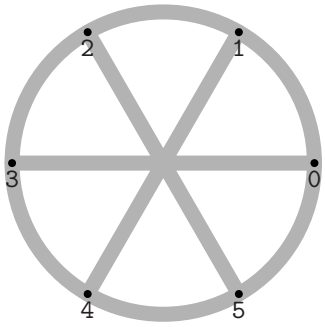


図 2.1.1

まずは，**点** から。**図 2.1.1** に示した 6 個の **点** (0 〜 5) を z0 〜 z5 とし，これらの座標を定めます。これらは同一円周上にありますので，まず半径と中心を設定しておくと便利でしょう。半径と中心は次のように定めます[*1]。

```
numeric radius; radius=0.5w;
pair ctr; ctr=(0.5w,0.5h);
```

半径 (radius) は **数値** ですので，単に値を設定するだけでもかまいません。しかし，中心 (ctr) は **点** (**pair**) ですので，必ず宣言 (pair ctr;) が要ります。

次に，z0 〜 z5 の設定です。図を見れば分かりますように，z[n] の位置は，中心から 60n° の方向にあります。したがって

```
z[n] = ctr + radius*dir(60n)
```

がその定義になります[*2]。これを for 文で n=0 から 5 まで繰り返せば，一気に 6 つとも設定できます。

```
for n=0 upto 5: z[n] = ctr + radius*dir(60n); endfor
```

さて，これで **点** が準備できましたので，**経路** の設定と描画に移りましょう。まずは描画に使う **ペン先** の指定です。今回は単純に丸ペンを用い，サイズは w の 5% とします。

```
pickup pencircle scaled 0.05w;
```

経路 は，**点** と **点** を .. で結べば曲線に，-- で結べば線分 (折れ線) になります。特に **巡回経路** は末尾を cycle で閉じます。したがって，外周の円は[*3]

[*1] w は横単位長あるいは文字の横幅，h は縦単位長あるいは文字の高さです。

[*2] dir は指定された方向角の単位ベクトルを生成するマクロです。

[*3] 同一円周上の等間隔な 4 つ以上の点を .. で結べば，ほぼ完全な円 (目で見る限り円と区別がつかない曲線) が得られます (実際には 3 点でも十分円に見えます)。そうでない曲線がほしい場合 — 細かい曲線の制御 — については次講で扱いますので，そちらも参照して下さい。

```
z0..z1..z2..z3..z4..z5..cycle
```

内部を走る線分は順に

```
z0--z3 , z1--z4 , z2--z5
```

となります[*4]。これらを draw すれば完成です。

2 ソースファイル全文

では，実際のソースに移りましょう。今回は最初ですので，METAFONT , METAPOST , MEPOTEX それぞれのソースファイルの全文を掲載します。ただし，前項で書いた「図形」の部分は全く同じで，違いといえば「絵を描く準備」に相当する部分だけです。

METAFONT は以下のようなソースになります。

```
mode_setup;

beginchar("A",50pt#,50pt#,0pt#);
numeric radius; radius=0.5w;
pair ctr; ctr=(0.5w,0.5h);
for n=0 upto 5: z[n] = ctr + radius*dir(60n); endfor
pickup pencircle scaled 0.05w;
draw z0..z1..z2..z3..z4..z5..cycle;
for n=0 upto 2: draw z[n] -- z[n+3]; endfor
endchar;

end.
```

beginchar(**フォント番号**, **幅**, **高さ**, **深さ**); と endchar; の間に前項で説明したソースが入っています。**幅**, **高さ**, **深さ** は # つきの単位や変数で指定します。これらの値は，指定したモードに対するピクセル数に換算され，w,h,d に格納されます。今回は幅・高さが 50pt# , 深さが 0pt# で「文字」を作ってみました。最初に mode_setup; , 最後に end. を入れるのを忘れないようにして下さい。

METAPOST は以下のようなソースになります。

```
beginfig(0);
numeric w,h,radius; w=4cm; h=4cm; radius=0.5w;
pair ctr; ctr=(0.5w,0.5h);
for n=0 upto 5: z[n] = ctr + radius*dir(60n); endfor
pickup pencircle scaled 0.05w;
draw z0..z1..z2..z3..z4..z5..cycle;
for n=0 upto 2: draw z[n] -- z[n+3]; endfor
endfig;
```

[*4] この 3 つも for 文を使うと楽でしょう。

```
end.
```

beginfig(**図番号**); と endfig; の間に前項のソースが入っています。また，METAFONT と違い w と h は自動では設定されませんので，最初に設定してあります。後で気が変わって図形のサイズを変えたくなることもあるでしょうから，この例のように最初に単位長を設定しておき，図の定義はその単位長を使った相対的な式で書いておけば便利でしょう。こうしておけば，図のサイズを変えるとき w=4cm; h=4cm; の部分を書き換えるだけで済みます。最後に end. を入れるのを忘れないようにして下さい。

MEPOTEX は以下のようなソースになります。

```
¥begin{MPpic}<50pt>(1,1)
¥openMP
numeric radius; radius=0.5w;
pair ctr; ctr=(0.5w,0.5h);
for n=0 upto 5: z[n] = ctr + radius*dir(60n); endfor
pickup pencircle scaled 0.05w;
draw z0..z1..z2..z3..z4..z5..cycle;
for n=0 upto 2: draw z[n] -- z[n+3]; endfor
¥closeMP
¥end{MPpic}
```

¥openMP と ¥closeMP の間に前項のソースが入っています。w , h は，MPpic 環境に入るときに設定した横単位長と縦単位長で，今回は 50pt に設定されています。MEPOTEX では METAPOST ソースの図形描画部分のみを TEX ソースに挿入できるように作られていますので，end. などは要りません。

3 beginchar, beginfig について

この項は少し込み入った話になりますので，興味のない方は読みとばして下さって結構です。

さて，説明に入る前に予備知識ですが，beginchar 〜 endchar や beginfig 〜 endfig では，内部をグルーピングするため beginchar (beginfig) の初めに begingroup を，endchar (endfig) の最後に endgroup を入れています。META 言語 のグルーピングは TEX と少し違い，基本的に変数などの変更はすべてグローバルで，特に指定されたものだけがローカルになるという仕様です。変数などをローカルにする — というより，それまでのその変数の意味を保存し，現在のグルーピングから抜けたときにその意味を復活させる — には，save を使います。もう少し具体的に言うと，グルーピング内でたとえば save x; とすれば，そのグルーピング内では，以後 x がローカル変数のように使え，現在のグルーピングから抜けたときもとの x が復活します。このことを念頭に置いて以下読み進めて下さい。

まずは，beginchar から。ここでは，上記グルーピングを開始した後，**フォント番号**[*5]

[*5] "A" のように文字で指定することも可能です。この場合 **フォント番号** は "A" の文字コードになります。

を charcode に保存します[*6]。そして，**幅**，**高さ**，**深さ**（いずれも # つきの寸法で指定します）を，charwd，charht，chardp に保存し，さらにこれらを # なし寸法に変換したものを w，h，d に設定します[*7]。そして，charic（イタリック補正）を一旦 0 にし，x，y の現在の意味を save しローカル変数に使えるようにします。また，currentpicture，currentpen をクリアします。そして最後に extra_beginchar という文字列（命令列）が定義されていれば，それも読み込みます。

endchar では，extra_endchar という文字列（命令列）が定義されていればそれを読み込み，モードに応じた形式でフォントを出力し，グルーピングを閉じます。

蛇足ですが，beginchar の引数の寸法は，複数の文字サイズに対応するため，変数で与えることが多いのですが，そのときも # つき変数を使うことを忘れないようにして下さい。また，METAFONT では「#つき」「#なし」2 つの「寸法」を対応づけるため，最初に解像度等を設定する必要があります。他にもプリンタに応じた補正なども設定する必要があります。これらを行うマクロが mode_setup; で，これを最初の beginchar の前に入れる必要があります。

beginfig，endfig で行われていることは，基本的に beginchar，endchar で行われていることと同じですが，若干 METAPOST 用に特化されています。

まず beginfig はグルーピングを開始し，画像番号を（METAFONT と同じく）変数 charcode に保存し，x，y の現在の意味を save しローカル変数に使えるようにします。また，currentpicture，currentpen をクリアします。そして，defaultpen を pickup し，drawoptions をクリア — drawoptions() — します。そして最後に extra_beginfig という文字列（命令列）が定義されていれば，それも読み込みます。

endfig では，extra_endfig という文字列（命令列）が定義されていればそれを読み込み，画像を出力し，グルーピングを閉じます。

あと，上記の定義を見ると，x1 や y1 がローカル変数として使えるのはわかりますが，z1 は初期化されておりません。にもかかわらずこれがローカル変数のように使えるのは，p.*21* にも書いたように，z⟨サフィックス⟩が実はマクロで

(x⟨サフィックス⟩,y⟨サフィックス⟩)

に置き換えられるためです。

Exercise 2.1　右図のみぞれマークを作ってみましょう。答は巻末に。

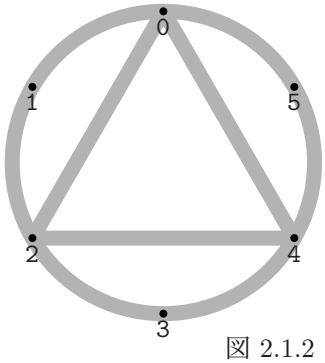


図 2.1.2

[*6] ただし，この時点で **フォント番号** が未知変数なら charcode:=0 になります。
[*7] 「#つき」「#なし」については **第1部 第10講 3項** を参照して下さい。

第 2 講

曲線の制御

1 直観的な曲線の制御

本講のメインテーマはマンボウ — ではなくて曲線の制御です。

さて，**図 2.2.1** を見て下さい。これが本講の目標の「マンボウの輪郭」です。この図を描くために座標を設定した点は，実は図に書き込んだ 8 点だけです。しかし，この 8 点を単純に曲線で結んだだけでは  のように訳の分からない形になってしまいます。では，どうするか。方法は 2 つ考えられます。1 つは，点をもっと細かく取って望ましい形にすること。これは数学的に定まった関数のグラフを描くときの手法としては有効ですが (また，コンピュータの画像データが最終的に落ち着く形でもありますが)，人間にとって受け入れやすい表記ではありません。たとえば,「ここをもうちょっと膨らませたい」というとき，細かく点を取った場合は，その周囲の点も含めておびただしい修正が待っているわけです。手間を考えると「ちょっと」なんてとても言えない雰囲気です。もう 1 つの方法は — これが METAFONT 言語最大の特長なんですが — 膨らます・絞る・すっと・くるっと，といった直観的な曲線の表現を理解するようなインターフェイスを用意して，それらで指定する方法です。これなら，絵を描く人は，自分のもつ直観を素直に式に翻訳するだけで望ましい形が得られ，しかも直観に従った修正も容易です。

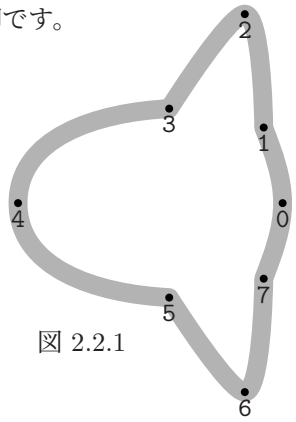


図 2.2.1

2 端点での指定

では実際に上の図の書き方を説明しましょう。マンボウの輪郭線は，z1 〜 z3 の部分，z3 〜 z5 の部分，z5 〜 z7 の部分，z7 〜 z1 の部分に分解されます。これらは「連続」につながっていますが，つなぎ目では「滑らか」ではありません。METAFONT 言語では，特に指定なしに .. で **点** を結べば,「すべての **点** をなるべく滑らかに」つないでしまいますので，z1 , z3 , z5 , z7 では滑らかさを断ち切る命令を入れます。この目的に使われるのは **{ベクトル}** と {curl **数値**} で，{(1,2)}z1{(3,4)} や，{up}z3{curl 0}[*1] のように，滑らかさを断ち切りたい **点** の両側にこれらの命令を配置します。

{ベクトル} はその **点** における曲線の接線方向を指示し，ソース上 **点** の前 (上の例では z1 の前の {(1,2)}) が **点** に入るとき，ソース上 **点** の後 (上の例では z1 の後の {(3,4)}) が **点** から出るときの接線方向です。これらのベクトルを異なるものにすれば，

[*1] up はベクトル (0,1) の省略形です。同様に down は (0,-1) の，right は (1,0) の，left は (-1,0) の省略形です。

この **点** で曲線は滑らかでなくなります。もちろん，両側のベクトルを同じにすれば，曲線は滑らかなままで，その **点** での接線方向を指定することになります。

{**ベクトル**} がその **点** での曲線の方向を客観的に与えるのに対し，{curl **数値**} は「主観的」な指定です[*2]。これは曲線の滑らかさを指定した **点** で断ち切り，同時にその **点** での曲線の「巻き」具合を指定します。デフォルトは {curl 1} で，**数値** を大きくすると巻き上がり，小さくすると直線的になります。

なお，**点** の両側で同じ {**ベクトル**} または {curl **数値**} を指定するときは，片方を省略できます。

少し例を挙げますと，z0=(0,0); z1=(0.5w,h); z2=(w,0); のとき，

図 2.2.2：単純に z0..z1..z2 としたもの

図 2.2.3：z0..{(1,1)}z1{(2,-1)}..z2 としたもの

図 2.2.4：z0..z1{(1,1)}..{down}z2 としたもの

図 2.2.5：z0{curl 0}..z1..{curl 0}z2 としたもの

図 2.2.6：z0{curl 10}..z1..{curl 10}z2 としたもの

です。

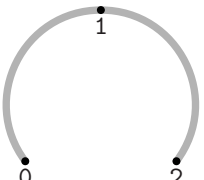


図 2.2.2

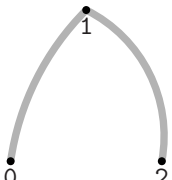


図 2.2.3

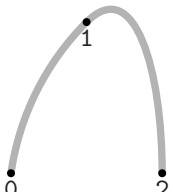


図 2.2.4

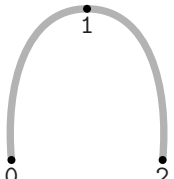


図 2.2.5

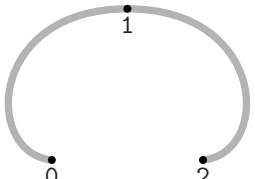


図 2.2.6

余談ですが，2 **点** を線分で結ぶ -- は，実は {curl 1}..{curl 1} の省略形 (マクロ) です。2 **点** の間の曲線を (滑らかさにおいて) 他から切り離して，滑らかに (というより素直に) 結べば線分となる，という理屈です。

3 曲線の張り具合

さて，次は曲線の張り具合です。これは tension で指定します。z0..z1 とすべきところを，たとえば z0..tension 3..z1 などととすれば，線の張り具合が変わります。**数値** は 3/4 以上の値で指定します。デフォルト (ただの ..) は tension 1 に相当します。始 **点** 側と終 **点** 側で張り具合を変えることもできます。たとえば z0..tension 3 and 4..z1 とすれば，z0 に近い側の tension を 3 に，z1 に近い側の tension を 4 にできます。これも実例を見てもらった方がわかりやすいと思います。先程と同じ **点** を用いて，

図 2.2.7：単純に z0..z1..z2 としたもの

[*2] もちろんプログラムされたものである以上，厳密には主観的ではあり得ないんですが。

図 2.2.8：z0..tension 3/4..z1..tension 3/4..z2 としたもの
図 2.2.9：z0..tension 2..z1..tension 2..z2 としたもの
図 2.2.10：z0..tension 5 and 1..z1..tension 5 and 1..z2 としたもの
図 2.2.11：z0---z1---z2 としたもの (--- は ..tension infinity..)
です。

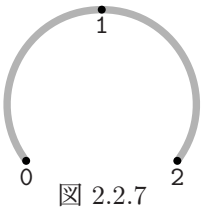


図 2.2.7

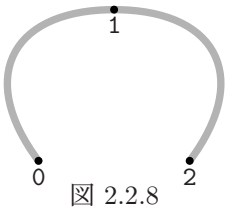


図 2.2.8

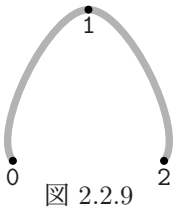


図 2.2.9

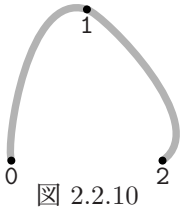


図 2.2.10

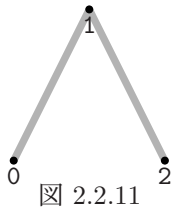


図 2.2.11

--- は図で見るとほとんど線分ですが，-- が曲線の他の部分と滑らかさにおいて切り離されているのに対し，--- は切り離されていません。したがって，線分 (--) と曲線を単純につなぐと，つなぎ目が滑らかになりませんが，--- を使うと楽に滑らかにつなげられます。下の 2 つを見比べて下さい。

図 2.2.12

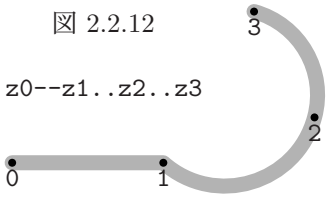
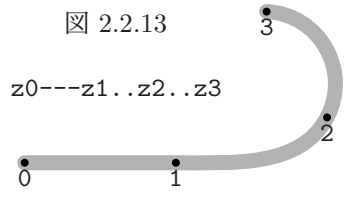


図 2.2.13



4 実　例

さて，ではマンボウの輪郭の説明に入りましょう。

z1..z2..z3 の部分 (背鰭) は少し張りつめた感じを出したいので，tension 3 を指定します。また z1 では正確に上向きに曲線を打ち出したいので，{up} を指定します。また，z2 がきっちり最高 **点** となるように接線方向を左方向 {left}に，z3 では一旦滑らかさを切って直線的に終わりたいので {curl 0} を指定します。したがって，

```
z1{up}..tension3..{left}z2..tension3..{curl 0}z3
```

のようになります。

z3..z4..z5 の部分 (頭) は，端 **点** で滑らかさを断ち切り直線的にしたいので，

```
z3{curl 0}..z4..{curl 0}z5
```

のようになります。

z5..z6..z7 の部分 (腹鰭?) は，向きが逆なことを除いて z1..z2..z3 の部分と同様です。

z7..z0..cycle の部分（尾鰭）は，端 **点** で滑らかさを断ち切り直線的にしたいので，

```
z7{curl 0}..z0..{curl 0}cycle
```

のようになります。cycle は出発 **点** z1 であると同時に，**経路** を **巡回経路** にします。

以上をまとめて，省略できるものを省略[*3]すると，以下のようになります。

```
z1{up}..tension3..{left}z2..tension3..{curl 0}z3
    ..z4..z5{curl 0}..tension3..z6{right}..tension3..{up}z7
    {curl 0}..z0..{curl 0}cycle
```

5 Bézier 3 次曲線

METAFONT 言語において，曲線は「Bézier 3 次曲線」で描かれています。Bézier 3 次曲線とは，2 点 z_1 , z_2 を結ぶとき，2 つのコントロールポイント（ z_{1a} , z_{2a} とします ）を使って，

$$(1-t)^3 z_1 + 3t(1-t)^2 z_{1a} + 3t^2(1-t) z_{2a} + t^3 z_2 \quad (\ 0 \le t \le 1\ )$$

で与えられる点を結んでいくというものです[*4]。

しかし，METAFONT 言語での z1..z2 という表現には，コントロールポイントは明示的に指定されていません。これはどうなっているかというと，METAFONT 言語では，与えられた指定や前後の曲線の具合から，自動的にコントロールポイントを求めてくれるためです。Bézier 曲線を描くときに，コントロールポイントをうまく見つけるのは結構難しいので，この仕様は図形のデザイン上すごくありがたいものといえます。ちなみにコントロールポイントを明示的に指定した書き方は

```
z1..controls z1a and z2a..z2
```

になります[*5]。

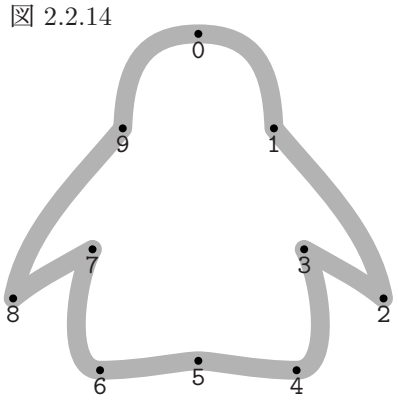
なお，tension の指定は，Bézier 曲線を書くときの 2 つのコントロールポイント（上の例では z_{1a} , z_{2a} ）が，z_1 や z_2 とどれくらい近いかを指定することに相当します[*6]。

Exercise 2.2　右図（ペンギンの輪郭線）を作ってみましょう。ただし，**点** z0 〜 z9 はすでに与えられているものとします。たとえば，

```
z0 = (0.5w,0.985h);  z1 = (0.7w,0.735h);
z2 = (0.99w,0.285h); z3 = (0.78w,0.415h);
z4 = (0.76w,0.095h); z5 = (0.5w,0.12h);
z6 = (0.24w,0.095h); z7 = (0.22w,0.415h);
z8 = (0.01w,0.285h); z9 = (0.3w,0.735h);
```

ちなみにマンボウやペンギンのような，対称形の場合は，対称軸を x 軸か y 軸にして **点** を設定しておいてから，あとで平行移動なり回転移動なりをした方が楽です。これについては **第4講** でまとめて話したいと思います。

図 2.2.14



[*3] たとえば，{curl 0}z3{curl 0} は {curl 0}z3 でかまいません

[*4] ちなみに z_1 における曲線の接線は z_1 から z_{1a} に向かう方向，z_2 における曲線の接線は z_{2a} から z_2 に向かう方向になります。

[*5] ..controls z1a.. は ..controls z1a and z1a.. の省略形になります。

[*6] tension c and d の c が z_1 と z_{1a} の近さ，d が z_2 と z_{2a} の近さに対応します。なお，詳しい規則は **第3部 第4講** で説明します。

第 3 講

塗りつぶし (1)

1 経路変数・塗りつぶし

本講のサンプルには，前講で作った「マンボウ」の輪郭を使います。これまでは，**点** をつないで **経路** を作って，それをそのまま draw に渡していましたが，今回同じ **経路** を何度も使いますので，**経路** を変数に格納しておきます。変数に格納するのは簡単で，

変数名 := **経路の定義式**;

とすれば OK です。ただし，変数名はタイプを予め宣言 (path **変数名**;) しておく必要がありますので，実際には以下のような形になります。

```
z0=(0.85w,0.5h);  z1=(0.8w,0.7h);  z2=(0.75w,h);
z3=(0.55w,0.75h); z4=(0.15w,0.5h); z5=(0.55w,0.25h);
z6=(0.75w,0);     z7=(0.8w,0.3h);
path p;
p := z1{up}..tension3..{left}z2..tension3..{curl 0}z3
      ..z4..z5{curl 0}..tension3..z6{right}..tension3..
      {up}z7{curl 0}..z0..{curl 0}cycle;
```

さて，今までは

```
draw p;
```

のように，**経路** を draw に渡し，曲線を描画していたのですが，これを

```
fill p;
```

のように fill に渡せば領域の塗りつぶしになります。ただし，fill に渡せる **経路** は，**巡回経路** に限ります。注意すべきは

```
z1--z2--z3--z1
```

は，たとえ始点と終点が同じであっても **巡回経路** ではないということです。**巡回経路** は

```
z1--z2--z3--cycle
```

のように，必ず cycle を使って記述しなければなりません。

2 色

META 言語のうち，METAPOST (したがって MEPOTEX も) はカラーに対応しています。色を使うには

draw (または fill) **経路** withcolor **色**;

のようにします。**色** は本来の METAPOST では rgb 方式の 3 次元ベクトル

(**赤**, **緑**, **青**)

なのですが，MEPOTEX では

hsb(**色相**, **彩度**, **輝度**) や cmyk(**シアン**, **マゼンタ**, **黄**, **黒**)

(いずれも各パラメータは 0 〜 1) の他，color.sty の named オプションで使える 68 色も，(マクロとしての実装で) 使えるようになっています。

少しサンプルを挙げましょう。

図 2.3.1：単純に `fill p;` としたもの

図 2.3.2：`fill p withcolor (0,1,1);` (シアンで塗りつぶし)

図 2.3.3：`fill p withcolor Cyan; draw p withcolor red;`
(シアンで塗りつぶした後赤で輪郭)

です。

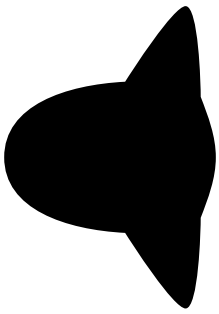


図 2.3.1

図 2.3.2

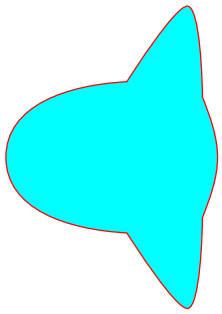


図 2.3.3

3 重ね塗りの処理

塗りつぶしについての表面的な説明は以上なのですが，実際の動作において注意するべきことがもう 1 つあります。それは,「重ね塗り」の際の動作です。例えば，重なりをもつ 2 つの領域 (**巡回経路** の内部) A と B を，この順に塗りつぶしたとします。このとき A と B の重なった部分は何色になるでしょうか。この動作が METAPOST と METAFONT で異なりますので，以下，それを説明します。

まず METAPOST の場合。これは一言でいえば「置き換え型」です。つまり A を赤 (1,0,0) で塗った後 B を青 (0,0,1) で塗ると，重なった領域は，後から塗った「青」になり，決して紫などにはなりません。これは「上書きされると下の図形が消えてしまう」ということを意味しますので，複数の図を重ねて描くときには，その順序が問題になってきます[*1]。この仕様は「METAPOST の」というより METAPOST の生成するデータ，つまり PostScript の仕様のようですので，マクロでどうにかなるものではありません。「描いたはずの図が描画されない」というときは，上のようなことも疑ってみて下さい。

次に METAFONT の場合。これは「加算型」です。METAFONT はフォントを生成するプログラムですので，最終的には白黒 2 値 (白 = 0 , 黒 = 1) になるのですが，描画途中のデータは整数値で保持しています。そして `fill` は「領域内の各点の色データを 1 加算する」という機能をもっています。したがって，A を塗りつぶした後 B を塗りつぶすと，重なった領域の色データは 2 になります。最終的にフォントにするときに，1 以上は黒 (1)，0 以下は白 (0) に統合されるのですが，描画途中で，1 以上を一旦 1 に統合する (2 などに統合することも可能) ようなコマンドもいくつかあります。これらをうまく

[*1] 簡単にいえば大きいものから描くということになるんですが，実際にはそう「簡単」ではないでしょう。

活用すれば，ある領域の外をクリップするとか，重なった領域のみを塗りつぶすなどの操作ができます[*2]。

以上のような仕様の違いがありますので，fill の反対の動作をする unfill の動作も，METAPOST と METAFONT とでは違います。METAPOST ではこれは background (通常白 (1,1,1)) での塗りつぶしになり，塗りつぶした領域には「何も残らない」状態になりますが，METAFONT では「その領域の色データを 1 減算する」というコマンドになり，たとえば，その領域内に色データ 2 の部分があれば，そこは黒のまま残ります。

Exercise 2.3 以下のソースを実行し比べてみましょう。(実行の前に結果を予想しましょう。) shifted は次講で説明する **変形** ですが，ここでは単に「平行移動」させるコマンドと思って下さい。3 つの円を順に「塗りつぶし」「塗りつぶし」「塗り消し」したものです。

MEPOTEX ソース

```
¥begin{MPpic}<5cm>(1,0)(0,1)
¥openMP
numeric r; r=w/3;
for n = 0 upto 3:
  z[n] = (r,r) + r*dir(90n);
endfor
path p[];
p1:=z0..z1..z2..z3..cycle;
p2:=p1 shifted (r*dir0);
p3:=p1 shifted (r*dir60);
fill p1 withcolor red;
fill p2 withcolor blue;
unfill p3;
¥closeMP
¥end{MPpic}
```

METAPOST ソース

```
beginfig(1);
numeric w,r; w=5cm; r=w/3;
for n = 0 upto 3:
  z[n] = (r,r) + r*dir(90n);
endfor
path p[];
p1:=z0..z1..z2..z3..cycle;
p2:=p1 shifted (r*dir0);
p3:=p1 shifted (r*dir60);
```

[*2] これらについては **第6講**，**第7講** で詳しく説明します。

```
fill p1 withcolor red;
fill p2 withcolor blue;
unfill p3;
endfig;
end.
```

METAFONT ソース

```
mode_setup;

beginchar(0,10pt#,10pt#,0pt#);
numeric r; r=w/3;
for n = 0 upto 3:
  z[n] = (r,r) + r*dir(90n);
endfor
path p[];
p1:=z0..z1..z2..z3..cycle;
p2:=p1 shifted (r*dir0);
p3:=p1 shifted (r*dir60);
fill p1;
fill p2;
unfill p3;
endchar;
end.
```

第 4 講

変　形

1 変　形

変形[*1]— 理系の人には変換と言った方がなじみが深いでしょうか。META 言語では，数学でいう affine 変換が使えます。これは，変換式が 1 次式であるような変換なのですが，もう少し具体的な効果で説明すると，

- 平行移動
- ある点のまわりの回転移動
- ある直線を中心とした特定の方向のみへの拡大縮小
- 正方形を平行四辺形に「つぶす」変換 — 直交座標から斜交座標へ

と，それらの合成をひっくるめたものです[*2]。

この **変形** は，**点** (**pair**) にも **経路** (**path**) にも **ペン先** (**pen**) にも **画像** (**picture**) にも使えますが，**画像** までもっていくと，METAPOST と METAFONT でだいぶ (実行可能な**変形**の制約が) 違うため，ここでは主に **点** と **経路** について扱うことにします。なお，**点** に「拡大縮小を施す」とは，中心となる直線や点からの距離が拡がる・狭まることを意味します。

本講のサンプルは，**図 2.4.1** の「工場マーク (歯車)」です。円の部分の書き方は以前説明しました。問題は「歯」の部分で，これはよく見ると台形になっています。この台形を従来通り 1 つ 1 つ定義すると，$4 \times 8 = 32$ 個の点を定義せねばならず，大変です。そこで台形の **経路** を 1 つ定義しておき，それを回転移動・平行移動して 8 つの「歯」にすることを考えます。また，1 つの「歯」は等脚台形，つまり，線対称な図形ですので，2 点を定義すれば，残りの点は対称移動で設定できます。

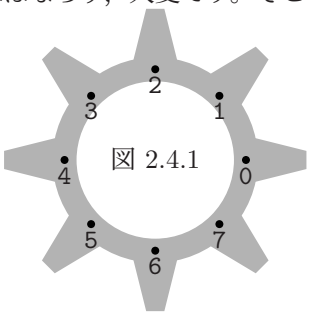
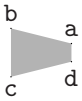


図 2.4.1

ではまず台形の定義から。

```
z.a = (0.2w,0.03w);   z.b = (0,0.08w);
z.c = z.b yscaled -1; z.d = z.a yscaled -1;
path p; p:=z.a--z.b--z.c--z.d--cycle;
```

これで描かれる台形は 

のようになります。z.a , z.b は直接設定しましたが，z.c ,

[*1]狭義には **transform** 値を指しますが，ここでは **transform** を作用させる演算およびそれと同等の作用をもつプリミティブ，マクロを合わせたもの，すなわち〈 変形演算子 〉の意味にとって下さい。なお，**第2部** ではプリミティブで与えられる〈 変形演算子 〉しか扱いません。**transform** 値の扱いについては **第3部 第7講** を参照して下さい。

[*2]なお,「ある直線に関する対称移動」は「直線を中心とした拡大縮小」の倍率が −1 のもの,「ある点を中心とした拡大縮小」は，直交する 2 直線について「直線を中心とした拡大縮小」を等しい倍率で行ったもの — 合成したもの — といえますので，こういった変換も affine 変換に含まれます。

z.d は z.b , z.a を y 軸方向に -1 倍[*3]したものとして定義しています。yscaled は y 軸方向の拡大率を指定する **変形** ですが，拡大率を負にすると，「y 軸方向に反転して拡大縮小」となります。予想はつくと思いますが，x 軸方向の拡大縮小は xscaled です。xscaled と yscaled に同時に同じ値を指定しますと，原点中心の拡大縮小（拡大率が負のときは原点中心の対称移動を行って拡大縮小）になります。これは scaled と表記することができます。つまり，

sacled t = xscaled t yscaled t

です。数式で書くとこんな感じです。

$$\texttt{xscaled}\, t \iff \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} t & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\texttt{yscaled}\, t \iff \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & t \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\texttt{scaled}\, t \iff \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} t & 0 \\ 0 & t \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

こうやって定義した点を用いて台形の **経路** p を定義できましたので，次はこれを適当な向きに回転させ，平行移動して配置します。（原点中心の）回転移動は rotated **角度** で，平行移動は shifted **ベクトル** で指定します。数式で書くとこんな感じです。

$$\texttt{rotated}\, t \iff \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\texttt{shifted}\, (s,t) \iff \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} s \\ t \end{pmatrix}$$

今回の場合 n=0〜7 に対し，**経路** p を 45n° 回転し，（予め設定した）z[n] まで平行移動すればいいので

```
p rotated(45n) shifted z[n]
```

のようになります。

以上をふまえて，**図 2.4.1** の歯車のソースは以下のようになります。

```
¥begin{MPpic}<4cm>(1,1)
¥openMP
z.a = (0.2w,0.03w);   z.b = (0,0.08w);
z.c = z.b yscaled -1; z.d = z.a yscaled -1;
path p; p:=z.a--z.b--z.c--z.d--cycle;
for n=0 upto 7:
  z[n] = 0.3w*dir(45n) + (0.5w,0.5h);
  fill (p rotated(45n) shifted z[n]);
endfor
pickup pencircle scaled 0.08w;
draw for n=0 upto 7: z[n].. endfor cycle;% 円の描画
¥closeMP
¥end{MPpic}
```

変形 には，本講で紹介したもの以外に，任意の点の周りの回転や，任意の直線に関する

[*3] y 座標を -1 倍すること，つまり x 軸に関する対称移動を意味します。

対称移動などもあるのですが，それらは **第3部 第7講** で説明します。また，本講で取り上げたような定義済みの特殊な変換でなく，一般的な affine 変換の指定法も **第3部 第7講** で説明します。

2 影付けとグラデーション

さて，本講のテーマの **変形** の応用例として，ちょっと面白い「絵」を作ってみます。以下，カラーを使いますので，METAFONT は関係なしで，METAPOST (と MEPOTEX) の話です。

よく,「ホームページ作成法」などというものを見ると，画像 (ボタンなど) を立体的に浮き立たせる方法が載ってますが，あれをやってみたいと思います。本項で試してみるのは,「影を付ける」ことと「ハイライト (一種のグラデーション)」です。

まずは,「影つけ」から。例によってマンボウの輪郭を使います。影の **経路** は，マンボウの輪郭の **経路** を **変形** させて作ります。今回採用する **変形** は「平行四辺形につぶす」**変形** の一種 slanted

slanted $t \iff \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & t \\ 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$

と，yscaled , shifted です。slanted は，もともとはその名の通り斜体字を作るための機能で，これを施した **経路** は，高さはそのままに横方向に傾斜した感じになります。したがって，slanted で少し傾斜させて，yscaled で少し縦方向に縮めて，shifted で若干ずらせば,「影」っぽくみえる **経路** が得られます。結果は**図 2.4.2** のようになります。この図は以下のソースで生成されています。(p はマンボウの輪郭の **経路** です。)

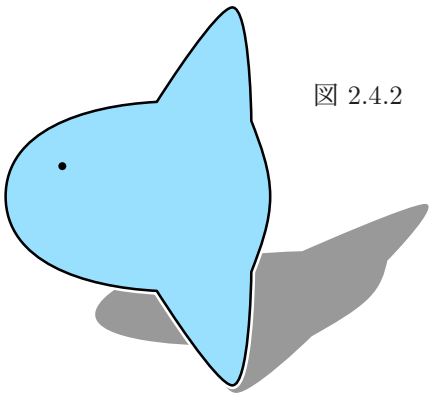


図 2.4.2

```
¥begin{MPpic}<5cm>(1,0)(0.2,0.98)
¥openMP
path p;
z0=(0.85w,0.5h);  z1=(0.8w,0.7h);  z2=(0.75w,h);
z3=(0.55w,0.75h); z4=(0.15w,0.5h); z5=(0.55w,0.25h);
z6=(0.75w,0);     z7=(0.8w,0.3h);  z8=(0.3w,0.58h);
p := z1{up}..tension3..{left}z2..tension3..{curl 0}z3
     ..z4..z5{curl 0}..tension3..z6{right}..tension3..{up}z7
     {curl 0}..z0..{curl 0}cycle;
fill (p slanted0.5 yscaled0.5 shifted(0.01w,-0.02h))
                                        withcolor 0.6white;% 影
pickup pencircle scaled 3pt;
draw p withcolor white;%                輪郭のまわりを少し消す（白太線）
fill p withcolor hsb(0.55,0.4,1);%      本体塗りつぶし
drawdot z8;%                            目玉
pickup pencircle scaled 1pt; draw p;% 改めて輪郭を黒で
¥closeMP
```

```
¥end{MPpic}
```

次はグラデーションを使ったハイライトです。残念ながら METAPOST には「グラデーションで塗る」というコマンドはありません[*4]ので，ちまちまと塗り重ねていきます。METAPOST の「塗り」は前回説明したとおり「上書き」型式ですので，大きい部分から小さい部分へと塗り重ねれば色が徐々に変化する図が得られます。元になる **経路** から，各色の領域の **経路** をつくるのには，**変形** を利用しています — **経路** の中心を原点に移動し，原点中心に縦横に適当に縮小し，再び中心を元の位置に戻しています[*5]。結果は **図 2.4.3** のようになります。この図は以下のソースで生成されています。

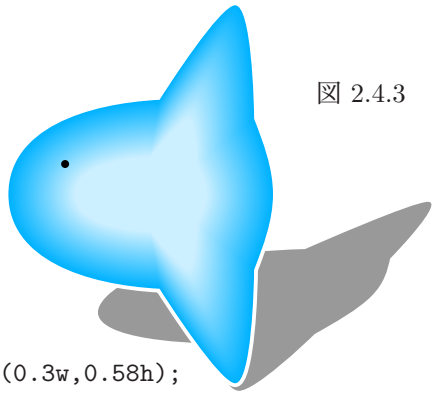


図 2.4.3

```
¥begin{MPpic}<5cm>(1,0)(0.2,0.96)
¥openMP
path p,q;
z0=(0.85w,0.5h);  z1=(0.8w,0.7h);
z2=(0.75w,h);     z3=(0.55w,0.75h);
z4=(0.15w,0.5h);  z5=(0.55w,0.25h);
z6=(0.75w,0);     z7=(0.8w,0.3h);  z8=(0.3w,0.58h);
p := z1{up}..tension3..{left}z2..tension3..{curl 0}z3
     ..z4..z5{curl 0}..tension3..z6{right}..tension3..{up}z7
     {curl 0}..z0..{curl 0}cycle;
fill (p slanted0.5 yscaled0.5 shifted(0.01w,-0.02h))
                                        withcolor 0.6white;% 影
pickup pencircle scaled 3pt;
draw p withcolor white;%                輪郭のまわりを少し消す（白太線）
for t=0 step 0.02 until 1.01:% ここからの 5 行がグラデーション部分
  q := p shifted (-0.5w,-0.5h) xscaled (1-0.5t) yscaled (1-0.6t)
                                        shifted (0.5w,0.5h);
  fill q withcolor hsb(0.55,1-0.8t,1);
endfor
drawdot z8;% 目玉
¥closeMP
¥end{MPpic}
```

Exercise 2.4　右の記号 (ガス田マーク) を作ってみましょう。ただし，まず 1 本棒を作り，それを回転等して記号を生成するものとします。

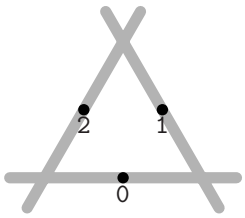


図 2.4.4

[*4] ver3.00 以降の MEPOTEX にはありますが，これはここで述べたものと同じアルゴリズムで実現しています。
[*5] 次講で扱う interpath というマクロを使えば，もう少し楽に必要な **経路** を生成できます。

第 5 講

経路の加工

経路 は定義したものを直接使うことが多いのですが，適当に加工して図に変化をもたせたり，一部を取り出して別の図の構成要素としたりすることもあります。こういった加工の 1 つの例は前講の **変形** ですが，ここではそれ以外の例 — subpath と interpath を取り上げてみます。

1 subpath , point 〜 of

第1講 で説明しましたように，METAFONT 言語では曲線 (直線・折れ線を含む) は **経路** 変数の形で扱われ，**経路** 変数は複数の **点** を接続して作られます。この **経路** の一部分を抜き出して，新たな **経路** を作るコマンドが subpath です。

1 つ例を挙げましょう。

```
path pth;
pth := z.a--z.b--z.c--z.d;
```

z.a , z.b , z.c , z.d の 4 点を順に線分で結んで，**経路** 変数 pth に代入するという式ですが，このとき，

```
z.a が point 0 of pth
z.b が point 1 of pth
z.c が point 2 of pth
z.d が point 3 of pth
```

で取得できます[*1]。そして，例えば

```
subpath(0,2) of pth
```

で，pth のうち 0 番の **点** から 2 番の **点** の部分，すなわち，z.a--z.b--z.c が取り出せます。また，

```
subpath(2,0) of pth
```

で，pth のうち 2 番の **点** から 0 番の **点** の部分，すなわち，z.c--z.b--z.a という上と逆順の **経路** が取り出せます。

さて，ここで質問です。point 〜 of や subpath (〜,〜) of の 〜 に，対応する点のない数値を入れたらどうなるでしょう。— これは **巡回経路** とそうでない **経路** で対応が違います。例えば，上の例では (**巡回経路** ではない) pth を構成する **点** は 4 つですので，番号は 0 番から 3 番までしかありません。ここで 3 以上の番号を指定するとそれはすべて z.d の意味になり，0 以下の番号を指定するとそれはすべて z.a の意味になります。一方，

[*1] 番号は 0 番から始まることに注意して下さい。よく間違いますが，

```
p1 := z1--z2--z3--z4;
```

の場合，(〈 添字 〉 とは無関係に) z1 が point 0 of p1 になります。
ちなみに **経路** の「最後の」番号 — **巡回経路** の場合は cycle に相当する点の番号 — は length **経路** で取得できます。

巡回経路，例えば，

```
q := z.a--z.b--z.c--z.d--cycle;
```

のときは，整数 k に対して $4k$ 番が z.a , $4k+1$ 番が z.b , $4k+2$ 番が z.c , $4k+3$ 番が z.d というように，4 を法とした剰余類で考えることになります。また，**巡回経路**・そうでない **経路** とも，小数（分数）の番号を指定した場合，前後の整数番号の **点** から適宜 **経路** 上の **点** が補間されます[*2]。

この規則によれば，上の q は z.a を出発して z.b , z.c , z.d を経由して z.a に戻る **巡回経路** ですが，subpath(1,5) of q は z.b を出発して z.c , z.d , z.a を経由して z.b に戻る **経路**[*3] , subpath(5,1) of q は z.b を出発して z.a , z.d , z.c を経由して z.b に戻る **経路** というように，**経路** の始点や向きを変えることができます。なお，**経路** 全体の向きを単純に逆転する reverse というプリミティブもあります。

2 経路の補間 〜 interpath 〜

interpath は，2 つの **経路** を補間[*4]するマクロで，

interpath(**パラメータ**, **経路**a, **経路**b)

の型式で使います。パラメータは通常 0〜1 の値を設定し，0 で **経路**a , 1 で **経路**b に一致します。たとえば，円と正方形を補間しますと，こんな風になります。

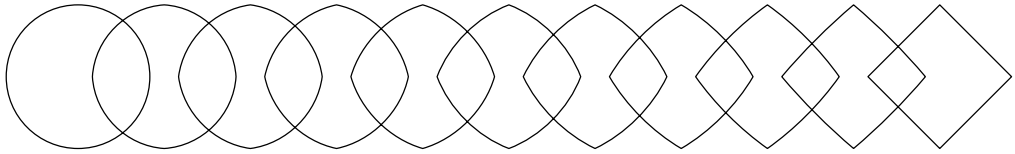


図 2.5.1

この図は以下のソースで生成されています。

```
path p,q;
for n=0 upto 3: z[n]=(0.5w,0.5h)+0.5w*dir(90n); endfor
p:=z0..z1..z2..z3..cycle;%                          左端の円
q:=(z0--z1--z2--z3--cycle) shifted (6w*right);%  右端の正方形
for n=0 upto 10: draw interpath(n/10,p,q); endfor
```

interpath は，**経路** を構成する **点** ごとに補間し，これらを結んでいますので，2 つの経路を構成する **点** の数は，揃えておく方がきれいに補間できます。

ちょっと立ち入った話になりますが，**点** の数が異なったときの補間は次のようになっています[*5]。

- interpath(t,p,q) の場合，p の **点** の数が，生成される **経路** の **点** の数になります。
- 巡回でない **経路** のとき，p の始 **点** は q の始 **点** に，終 **点** は終 **点** に対応します。間の **点** については，番号の小さい順に対応をつけていき，p の **点** が余ったときは，余っ

[*2] 詳しくは **第3部第4講** を参照して下さい。

[*3] subpath で「切り取る」と **巡回経路** でなくなってしまいますので注意して下さい。なお，**巡回経路** にしたいときは subpath(1,5) of q -- cycle とします。

[*4] ドローソフトで「ブレンド」などとよばれている機能です。

[*5] 興味のない人は読みとばして下さい。

た **点** をすべて q の終 **点** に対応させ，q の **点** が余ったときは，余った **点** をとばします。

○ **巡回経路** のとき，p の始 **点** は q の始 **点** に対応します。(終 **点** は cycle で閉じますのでとくに合わす必要はありません。) **巡回経路** のときは，**点** の番号が巡回的に付けられますので，q の **点** の数が少ないときは，2 周，3 周と必要なだけ点をとります。q の **点** が余ったときは，余った **点** をとばします。

以下の例では，2 つの **経路** を構成する **点** の数と中心からの方向を，わざと変えてあります。場合によってはこうして独特の雰囲気を演出することも考えられます。

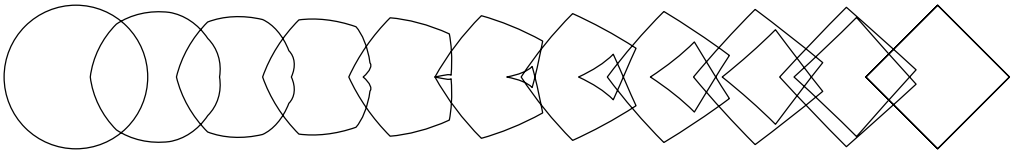


図 2.5.2

この図は以下のソースで生成されています。このソースで p は 8 点からなる円，q は 4 点からなる正方形です。

```
path p,q;
for n=0 upto 7: z[n]=(0.5w,0.5h)+0.5w*dir(45n); endfor
p:=z2..z3..z4..z5..z6..z7..z0..z1..cycle;
q:=(z0--z2--z4--z6--cycle) shifted (6w*right);
for n=0 upto 10: draw interpath(n/10,p,q); endfor
```

3 グラデーション再説

前講では，グラデーションを実現する際に，各色の領域を **変形** で作っていましたが，ハイライト部分がもとの形の「相似形」に固定されてしまいます。ハイライト部分をもっと異なる形状にしたいなら，2 つの **経路** を補間できる interpath が便利です。これを使った例が **図 2.5.3** です。この図のソースは以下のようになっています[*6]。

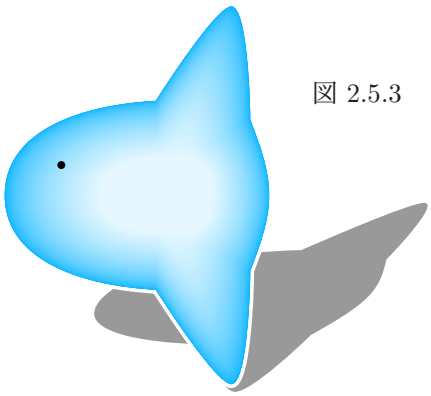


図 2.5.3

```
¥begin{MPpic}<5cm>(1,0)(0.2,0.96)
¥openMP
path p,q;
z0=(0.85w,0.5h);  z1=(0.8w,0.7h);  z2=(0.75w,h);
z3=(0.55w,0.75h); z4=(0.15w,0.5h); z5=(0.55w,0.25h);
z6=(0.75w,0);     z7=(0.8w,0.3h);  z8=(0.3w,0.58h);
p := z1{up}..tension3..{left}z2..tension3..{curl 0}z3
     ..z4..z5{curl 0}..tension3..z6{right}..tension3..{up}z7
     {curl 0}..z0..{curl 0}cycle;
```

[*6] p はマンボウの輪郭，q は中央のハイライト部分の **経路** です。p , q を構成する **点** の数と中心からの方向をそろえておくと，きれいなグラデーションができます。

```
q := (0.65w,0.6h)..(0.6w,0.6h)..(0.55w,0.6h)..(0.4w,0.5h)..
     (0.55w,0.4h)..(0.6w,0.4h)..(0.65w,0.4h)..(0.7w,0.5h)..cycle;
fill (p slanted0.5 yscaled0.5 shifted(0.01w,-0.02h))
                                        withcolor 0.6white;% 影
pickup pencircle scaled 3pt;
draw p withcolor white;%              輪郭のまわりを少し消す（白太線）
for t=0 step 0.02 until 1.01:% ここからの 3 行がグラデーション部分
  fill interpath(t*t,p,q) withcolor hsb(0.55,1-0.9t,1);
endfor
drawdot z8;% 目玉
¥closeMP
¥end{MPpic}
```

ちなみに，各講のタイトルに使われている枠の背景のグラデーションも，同様のテクニックで作られています。以下にそのグラデーション部分のソースを書きます[*7]。

```
path p[]; numeric r; r=5mm;
z0=z4=(0,-0.5h); z1=(w,-0.5h); z2=(w,0.5h); z3=(0,0.5h);
for n=0 upto 3:
  z.a[n]   - z[n]   = r * unitvector(z[n+1] - z[n]);
  z.b[n+1] - z[n+1] = r * unitvector(z[n]   - z[n+1]);
endfor
p0:= z.a0--z.b1{right}..{up}z.a1--z.b2{up}..{left}z.a2--
           z.b3{left}..{down}z.a3--z.b4{down}..{right}cycle;
p1:=subpath (3,6) of p0;
p2:=subpath (-1,2) of p0;
for t = 0 step 0.02 until 1.01:
  fill p1--(interpath(t,p2,reverse p1))--cycle
                                withcolor hsb(2/6-t/6,0.2,1);
endfor
```

Exercise 2.5　右の星を作ってみましょう。ただし，星形の **経路** を生成するマクロ

StarShape(**中心**, **半径**, **方向角**)

は，すでに与えられているものとします。**色** は MEPOTEX の hsb マクロなどを用いて下さって結構です。

図 2.5.4

[*7] 実際にはマクロ化されています。

第 6 講

塗りつぶし (2)

1 buildcycle

「4 つの曲線 p , q , r , s で囲まれた部分の面積を求めよ」などという問題を説明するとき,「 p , q , r , s で囲まれた部分」を図示して斜線や塗りつぶしを施すことがよくありますが,「塗りつぶし」を実行するには，その囲まれた部分を表す **巡回経路** を p , q , r , s から作らねばなりません。このようなとき便利に使えるのが METAPOST マクロの buildcycle で,

```
buildcycle(p,q,r,s)
```

とすれば,「p , q , r , s で囲まれた部分」の **巡回経路** が得られます。ところがここに 1 つ問題があります。p , q , r , s の囲む部分が 2 つ以上あったら，あるいは，曲線どうしの交 **点** が 1 つでなかったら，一体どの部分が返ってくるのでしょうか。— 結論を言いますと,

s と p の交 **点** のうち，p 上の番号が最も若い **点**

を出発 **点** とし,

p と q の交 **点** のうち，q 上の番号が最も若い **点**
q と r の交 **点** のうち，r 上の番号が最も若い **点**
r と s の交 **点** のうち，s 上の番号が最も若い **点**

を順に経由し，元に戻るという **巡回経路** になります[*1]。しかし,「囲む部分」が 2 つ以上あるとき，こうして作られた **巡回経路** が，希望するものである保証はありません。このときどうすればいいかというと — 前講の内容をふまえれば明らかだと思いますが — subpath を使って **経路**の始 **点** を変えて,「最も若い **点** 」が望ましい **点** になるよう細工するわけです。というわけで，本講の目標は**図 2.6.1** です。

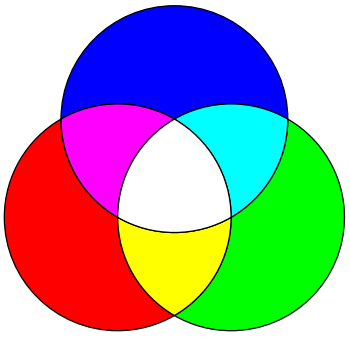


図 2.6.1

一見簡単そうですが，METAPOST は塗りつぶしが「上書き」ですので，単に塗り重ねても，色は合成されません。**図 2.6.1** は，ちまちまと各領域を 1 つずつ塗りつぶしています[*2]。

3 つの円の名前は，左下が p , 右下が q , 中央上が r であるとします。また，3 つの円はそれぞれ 4 **点** から構成され，その 4 **点** の **経路** 上の位置関係は，**図 2.6.2** のようになっ

[*1] ただし，p 上の番号が最も若い交 **点** と q 上の番号が最も若い交 **点**が違っていて，なおかつ，それらの交点が近接している — p , q を構成する隣り合う描画点 (キーポイント) の間に 2 交点ともに入る — ときは，この原則からはずれることがあります。(詳しくは **第3部 第4講 8項** を参照して下さい。) したがって，うまく領域が取得できないときは，以下に示すように，p と q の交 **点** が，p 上でも q 上でも最も若い交 **点** となるように，**経路** の向きや始 **点** を変更する (場合によっては **経路** の一部のみを取り出す) 必要があります。

[*2] 「上書き」であることを活用して，もう少し手抜きはできますが，今回は領域の周の取得が 1 つの目標ですので，地道にやることにします。

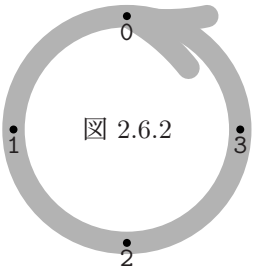


図 2.6.2

ているものとします。今回は subpath(a,a+b) of pth の形をよく使いますが，これは長いので，短縮したマクロ SP(a,b,pth) が定義されているものとします[*3]。今回の円は上記のように 4 **点** で構成されているため，b に 4 を入れると円を順方向に 1 周，−4 を入れると逆方向に 1 周となります。

さて，目標とする領域は 7 つあるのですが，要領は同じですので，そのうちの 1 つ，**図 2.6.3** の赤く塗った領域を取得してみましょう。以下の作業は，頭で理解するのは難しいですが，紙の上で実際に作業してみると，体で覚えられます。是非お試し下さい。— ではまず，**図 2.6.3** のように，求める領域の境界が，もとの 3 円 p,q,r のどの部分からできているか確認して下さい。buildcycle(p,q,r) の形で使うときは，まず，r と p の交 **点** (図の a) を確認します。この a が出発 **点** になり，図に書き込んだ矢印のように，円 p の順方向に進み，p と q の交 **点** (図の b) に到ります。ここで円 q に乗り換えて，その逆方向に進み，q と r の交 **点** (図の c) に到ります。ここで円 r に乗り換えて，その逆方向に進み，もとの a に到ります。ここまで確認したら，次はこれらの交 **点** が，各円 (**経路**) 上で「最も早く現れる交 **点**」になるように，**経路** の始 **点** をずらします。まず，a が p 上で最も早く (r と p のもう 1 つの交 **点** より早く) 現れる交 **点** になるようにするのですが，p のもともとの始 **点** が a のすぐ手前にあるため，何もしなくてかまいません。(あえて SP を使うなら SP(0,4,p) になります。) 次に q ですが，これは逆方向にまわることを考慮して，b の「手前」(順方向なら直後) にある **点** に始 **点** をずらすことになります。たとえば，SP(2,-4,q) でいいでしょう。r も同様に考えて SP(2,-4,r) とでもしておけばいいでしょう。以上を総合して，求める **経路** pth は，以下のようになります。

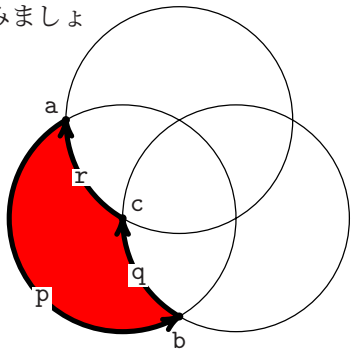


図 2.6.3

```
pth:=buildcycle(SP(0,4,p),SP(2,-4,q),SP(2,-4,r));
```

もう 1 つやってみましょう。要領はさっきと同じです。もうちょっと作業っぽい書き方にしますと，

(1) まず円 r と p の 2 交 **点** に a と書く。
(2) 同じく円 p と q の 2 交 **点** に b と書く。
(3) 同じく円 q と r の 2 交 **点** に c と書く。
(4) 欲しい領域の境界を太く塗って，要らない交 **点** に × を打つ。
(5) 残った交 **点** を a , b , c の順に矢印で結ぶ。
(6) a が p 上 a より早く現れるように，また (5) で確認した向きにも注意して，p の始 **点**・向きを変える。
(7) q 上の b , r 上の c についても同様。

という感じになります。

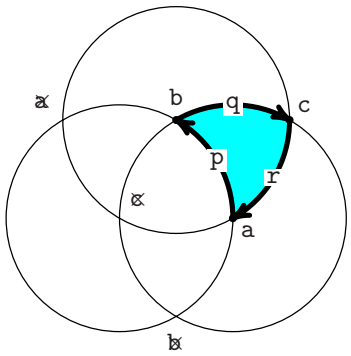


図 2.6.4

[*3] マクロの定義については，**第10講** で扱います。

Exercise 2.6.1 残りの領域も作ってみて，**図 2.6.1** を完成させましょう。

2 cull

前項で取りあげた buildcycle は METAPOST に標準で備わっているマクロですが，METAFONT にはなぜかありません。まあ，フォントを作るときはあんまり使わない機能かも知れませんし，METAPOST と METAFONT の **経路** レベルのマクロはほぼ互換ですので，どうしても欲しいときには plain.mp からもってくればいいのですが，本項では METAFONT について前項とは別のアプローチで，同様の「機能」を実現してみたいと思います[*4]。

さて，まずは少しおさらいを。METAFONT では METAPOST と違い「塗り」が加算式です。最終的には白黒 2 値になるのですが，途中のデータ (**picture** 変数) の段階では，各点の「色」を整数値で管理しています。そして,「塗 (draw , fill)」ればその整数が 1 加算され,「消 (undraw , unfill)」せば 1 減算されます。通常はこうしてできた**画像**を，1 以上の「色」をもつ部分は黒に，0 以下の「色」をもつ部分は白にしてフォントを作るわけですが，これは，別の見方をすれば，1 以上の「色」を 1 に，0 以下の「色」を 0 に統合したともいえます。このような「色の統合」を，画像を作っている最中に行うのが cull で，これは単に上記のような統合をするだけでなく，白黒の境い目の整数をずらしたり，特定の値の「色」だけ抜き出したりできます。

では具体的な説明に入りましょう。サンプルは前項と同じ 3 つの円 (**図 2.6.5**) です。便宜上，左下の円を p , 右下の円を q , 中央上の円を r とします。まず，右図の黄色い部分を塗りつぶすにはどうすればいいでしょう。— これは特にコマンドを使う必要はありません。

```
fill p; unfill q; unfill r;
```

で黄色い部分だけ，正の「色」をもつようになりますので，最終的にここだけが黒塗りされます。なお，右図での色分けは，fill p; unfill q; unfill r; としたときの，各領域の「色」の値を反映しています。

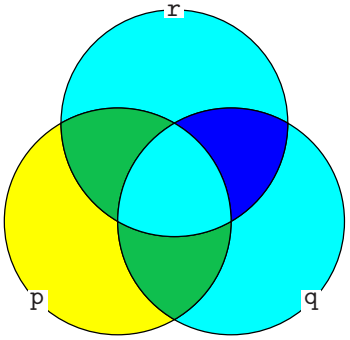


図 2.6.5

青：−2 , 水：−1 , 緑：0 , 黄：+1 , 橙：+2 , 赤：+3 のつもりです。以下もこの色分けでいきます。

次は，3 つの円の重なった中央の部分だけを塗ることを考えましょう。ここは，

```
fill p; fill q; fill r;
```

としたとき,「色」が最大の 3 になる領域ですので，白黒の境い目をずらせば楽に塗れます。このために cull を使います。使い方は，

```
cull 画像 keeping （黒にする範囲）;
```

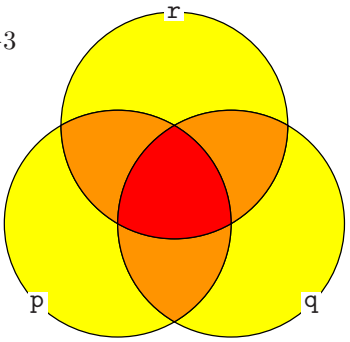


図 2.6.6

[*4]なお，本項の図は，METAPOST で METAFONT の動作の「結果」を模倣しているだけですので，ソース — 本文中に ¥verb で書き込んでいるソースではなく，この文章そのもののソース — を見ても余り役には立たないでしょう。

または

　　cull **画像** dropping (**白にする範囲**);

です。**画像** は **picture** 変数で，今回の場合は currentpicture になります。**黒 (白) にする範囲** は (3,5) のように **ベクトル** で表現します。

　　keeping (3,5) とすれば 3 , 4 , 5 が黒 (1) で他は白 (0) に

　　dropping (-3,1) とすれば -3 , -2 , -1 , 0 , 1 が白 (0) で他は黒 (1) に

なります。今回の場合，3 以上を黒にすればいいので，

　　cull currentpicture keeping (3,infinity);

でいいでしょう。

なお，cull を使うとき 1 つ注意すべきことがあります。METAFONT では一応各フォントでサイズを指定しますが，あくまでこれは TEX が認識するサイズであって，描画領域はこのサイズに限定されません[*5]。このことは，画像中で「色」が 0 の点は無限にあるということを意味します。そこで，METAFONT は「色」が 0 以外の点のみ管理するという形で，この問題をクリアしています。したがって，METAFONT は「色」が 0 の点すべてを一度に他の「色」には変えられない，という制約をもちます。より具体的に言えば，

　　keeping では 0 を含めてはいけない

　　dropping では 0 を含めなければならない

という制約があります。ご注意下さい。

ついでに，cull は通常「色」を 0 と 1 に統合しますが，0 と w に統合することもできます。そのためには

　　cull **画像** keeping (w **にする範囲**) withweight w;

または

　　cull **画像** dropping (0 **にする範囲**) withweight w;

のように withweight w を末尾につけます。

では最後の例。p と q の交わりで，r の外部にある部分を塗るにはどうすればいいでしょう。— 正解例は

　　fill p; fill q; unfill r;

として (**図 2.6.7**) 2 以上の部分を黒にすればいいので，

　　cull currentpicture keeping (2,infinity);

となります。

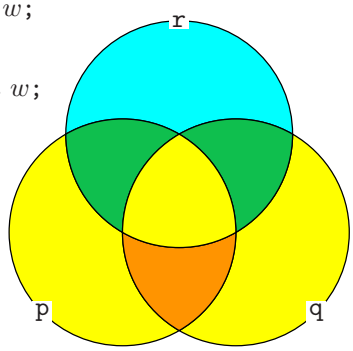


図 2.6.7

Exercise 2.6.2　最後の例で,「色」が 0 以外 (1,2,-1 がある) の領域を黒にするにはどうすればいいでしょうか。

[*5] たとえば，LATEX の picture 環境で使う円のためのフォントなどは，わざとフォントのサイズに指定した領域から大きくはみ出させてあります。

第 7 講

クリップ

1 クリップ

本講では「ある領域の外を消す」テクニックを取り上げます。これは METAPOST と METAFONT でやり方が違います。まずは，本講のサンプル (**図 2.7.1**) をご覧下さい。x 軸と y 軸はそれぞれ $-2.5 \leq x \leq 2.5$, $-3 \leq y \leq 3$ の範囲で描いてあります。ここに $y = x^3 - 3x$ のグラフを描きます。ちょっと計算すればわかると思いますが，3 次関数は増加速度が速いので，$x = 2.5$ に達する前に，y の方が 3 を越えてしまいます。これを $-3 \leq y \leq 3$ の範囲に収めるには，$y = x^3 - 3x = \pm 3$ となる x を求めて，そこまでの描画にすればいいのですが，3 次方程式を解くのはさほど容易ではありませんし，もっと複雑な関数なら事実上お手上げということも考えられます。そこで，$-2.5 \leq x \leq 2.5$, $-3 \leq y \leq 3$ の長方形領域の外をクリップする (= 刈り込む) ことを考えます。なお，本講では座標軸の書き方や文字の入れ方については触れません。こういうものはマクロ化しておかないと面倒ですし，マクロ化したものはすでに MEPOTEX にありますので，次講で，関連するトピックスいくつかとまとめて説明したいと思います。

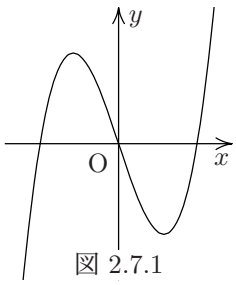


図 2.7.1

以下簡単のため，座標軸などは描かず，**図 2.7.2** のように 3 次関数のグラフと，それをクリップするときの境界線のみを考えることにします。3 次関数のグラフは，次のように **経路** 変数 pth に格納されているものとします。

```
pth := ( for t=-2.5 step 0.1 until 2.5:
           (t,t*(t*t-3)) --
         endfor (2.5,2.5*(2.5*2.5-3)) ) scaled w;
```

この書き方は METAPOST , METAFONT でほぼ共通です[*1]。また，クリップするときの境界線も，次のように **経路** 変数 edg に格納されているものとします。

```
edg := (-2.5w,-3h)--(2.5w,-3h)--
                    (2.5w,3h)--(-2.5w,3h)--cycle;
```

これらを

```
draw pth; draw edg;
```

として描いたものが **図 2.7.2** です。3 次関数のグラフが大きくはみ出しているのがわかると思います。

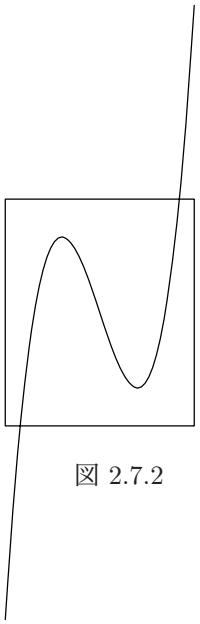


図 2.7.2

[*1] MEPOTEX なら専用のマクロがありますので，もっと楽に書けます。それについても，次講を参照して下さい。

2 METAPOST でのクリップ

METAPOST にはそのものズバリ clip というプリミティブが備わっています。使い方は,

```
clip 画像 to 境界線
```

です。今回の場合は，draw pth; で currentpicture に 3 次関数のグラフを書き込んだ後で,

```
clip currentpicture to edg;
```

とすれば，目的は達せられます。

なお，METAPOST で作成した PostScript ファイルを画像として表示させて確認する場合，GSview などでは原点を紙面の左下に合わせますので,「グラフの第 1 象限しか見えない」ということが起こります。これがまずいようでしたら，作成した画像を最後に shifted で平行移動することになります。以下のソースはそれも行った METAPOST 用のソースです。

```
beginfig(0);
numeric w,h; w=h=5mm;
path pth, edg;
pth := ( for t=-2.5 step 0.1 until 2.5:
         (t,t*(t*t-3)) -- endfor (2.5,2.5*(2.5*2.5-3)) ) scaled w;
edg := (-2.5w,-3h)--(2.5w,-3h)--(2.5w,3h)--(-2.5w,3h)--cycle;
draw pth;
clip currentpicture to edg;
currentpicture := currentpicture shifted (2.5w,3h);
endfig;

end.
```

3 METAFONT でのクリップ

METAFONT には残念ながら clip というプリミティブはありません。しかし，前講で説明したテクニックを使えば，簡単に「クリップ」できます。まず,

```
draw pth; fill edg
```

とします。これにより，3 次関数のグラフでクリップ領域の内側に入った部分の「色」が 2 , 他の部分が 1 になりますので,

```
cull currentpicture keeping (2,infinity);
```

で，その部分だけ残せるわけです。

なお，METAFONT は通常 0<x<w , -d<x<h でフォントを作りますので，proof モードや GFtoDVI で作られる「ゲラ」で仕上がりを確認したいときは,「文字」の幅・高さ・深さに適当な値を設定した上で，上記の範囲に画像 (「文字」) が収まるようにしなければいけません。(さもないとはみ出してビューアで確認しづらくなります。) したがって，前項のソースは，以下のように修正すべきです。

◦ x の変域の幅 5 をフォントの幅 w に合わせ，y の変域の幅 6 をフォントの高さと深さの和 h+d に合わせるため，pth の定義の最後の scaled w の部分を

```
xscaled (w/5) yscaled ((h+d)/6)
```

に変えます。

◦ さらに 図の中心をフォントの中心 0.5[(0,-d),(w,h)] に合わせるため，

```
shifted (0.5[(0,-d),(w,h)])
```

を行います。

◦ クリップの境界の定義は

```
edg := (0,-d)--(w,-d)--(w,h)--(0,h)--cycle;
```

にします。

◦ あと **ペン先** の指定 (pickup pencircle scaled 0.4pt; など) も忘れないようにします。METAFONT のデフォルトペンは大きさが 0 で目に見えないため，描いたはずの図が見えなくてあわてることがよくあります。

これらの修正を加えたソースは以下の通りです。

```
mode_setup;

beginchar(0,10pt#,8pt#,2pt#);
path pth, edg;
pth := ( for t=-2.5 step 0.1 until 2.5:
           (t,t*(t*t-3)) -- endfor (2.5,2.5*(2.5*2.5-3)) )
       xscaled (w/5) yscaled ((h+d)/6) shifted (0.5[(0,-d),(w,h)]);
edg := (0,-d)--(w,-d)--(w,h)--(0,h)--cycle;
pickup pencircle scaled 0.4pt;
draw pth; fill edg;
cull currentpicture keeping (2,infinity);
endchar;

end.
```

4 網掛け

クリップの応用として網掛けについて考えてみたいと思います。ここでいう「網掛け」とは，**図 2.7.3** のように特定の領域に網をかけることを指します。これは，特定の領域に向きの異なる斜線を引けば実現できるため,「斜線かけ」ともいえます。

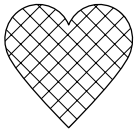


図 2.7.3

残念ながら，plain な状態の METAFONT や METAPOST には，網掛けの機能はありません。しかし，以下のようにすれば網掛けを実現できます。

◦ まず，これまでに描画した画像を **画像** 変数 (仮に pct と名付けたとします) に待避 (代入) しておきます。その上で，一旦 currentpicture に nullpicture を代入[*2]しておきます。

[*2]つまり，currentpicture を一旦クリアするわけです。

- 斜線を引きたい領域を含む少し大きな領域に，等間隔に斜線を引きます。
- 斜線を引きたい領域の境界線を **経路** 変数に代入し，METAPOST なら

```
clip currentpicture to 境界線
```

METAFONT なら

```
cullit; [*3]
fill 境界線
cull currentpicture keeping (2,infinity);
```

とします。

- 境界線を描画します。
- 待避させておいた画像を合成します。

```
addto pct also currentpicture; [*4]
currentpicture:=pct;
```

「斜線を引きたい領域を含む少し大きな領域」は，METAFONT の場合「フォントの幅・高さ・深さいっぱい」にとればいいでしょう。METAPOST の場合，

```
llcorner(画像または経路)
```

で **画像** または **経路**を含む (座標軸方向の辺をもつ) 矩形の左下の **点** が取得でき，同様に lrcorner で右下，ulcorner で左上，urcorner で右上の **点**が取得できますので，それを活用すれば，一般の領域に対し上記の操作をマクロ化できるでしょう。実際 MEPOTEX には領域に網掛けを施すマクロが定義されており，そのアルゴリズムはおおよそ上のようになっています。これについても，次講で使い方を説明したいと思います。

Exercise 2.7　今までの内容を駆使して **図 2.7.4** を書いてみましょう。斜線の向きは 45° の方向で，横方向の間隔は 2mm です。

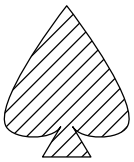


図 2.7.4

[*3] cullit は cull currentpicture dropping (-infinity,0) の省略形です。

[*4] METAPOST で addto を使う場合，図の上下に注意して下さい。

```
addto A also B ;
```

で，A の **上に** B を描きます。

第 8 講

MEPOTEX の便利なマクロ

1 本講で扱う MEPOTEX マクロ一覧

本講で扱う便利な MEPOTEX マクロのおおよその書式と主要なオプション引数を説明します。「TEX マクロ」と記されたマクロは，MEPOTEX の隠蔽モード (参照：**第1部 第6講**) で使うマクロ,「 METAPOST マクロ」と記されたマクロは，MEPOTEX のオリジナルモードの METAPOST 領域，または，隠蔽モードの `¥sendMP{}` 内など METAPOST 式を要求する部分で使うマクロです。

○ `¥mptLabel{`**配置点**`}[`**ラベル位置**`]<`**横補正**`,`**縦補正**`>{`**ラベル文字列**`}` ···· TEX マクロ

図中に文字列を入れます。デフォルトで文字列の背景に当たる部分の画像を消去してくれます。[**ラベル位置**] と <**横補正**, **縦補正**> は省略可です。より細かくオプションを指定する方法もありますが，それについては MEPOTEX マニュアルをご覧下さい。

配置点 には METAPOST の文法で **点** (**pair**) に相当するものなら何を入れてもかまいません。(**pair** 値を取る「式」でもかまいません。)

ラベル位置 はラベルの文字列のどの位置を **配置点** に合わせるかを指定します。`t` , `b` , `B` , `l` , `r` の 1 つまたは 2 つ，あるいは `c` を指定します。`t` はラベル上部，`b` は下部，`B` はベースライン，`l` は左部，`r` は右部，`c` は中央，2 つ組み合わせると例えば `bl` で左下部になります。デフォルトは `c` です。

横補正 と **縦補正** はラベルの文字列を配置する **点** を微調整します。つまり，ここに指定した寸法だけラベルを横方向または縦方向に移動します。

このマクロでは文字列を METAPOST ではなく TEX で処理しています。したがって，POSTSCRIPT フォントの知識や特殊 (?) な設定は必要ありません。より細かくいえば，まず文字列を TEX で処理して，大きさを測っておき，その大きさを (背景削除用に) METAPOST へ転送すると同時に，文字列が適切な位置に配置されるよう計算された座標を，TEX に戻させるような命令も METAPOST に転送しています。

○ `¥mptDrawPath[key=val` **列**`]{`**経路**`}` ····· TEX マクロ

その名の通り **経路** を描画し，(**巡回経路** に対しては) 塗りつぶしもできます。

`key=val` **列** に指定できる主要なキーの種類と意味は次の通りです。

- `linetype` — 描画する **経路** の線のタイプです。`none` を指定すれば **経路** を描きません。(塗りつぶしのみ行うときに便利です。) それ以外の「値」については，METAPOST の文で `draw` **経路** の後に続くものなら何でもかまいません。
- `linecolor` — 描画する **経路** の線の色です。デフォルトは「描画色」(通常黒) です。
- `pensize` — **経路** を描く丸ペンの直径です。デフォルトは `0.6pt` です。
- `arrowtype` — **経路** の両端に矢印をつけるかどうかを指定します。`->` なら終点のみ,

<- なら始点のみ，<-> なら両端，指定がなければ矢印をつけません。他にも，|-> や <-< なども指定できます。(形はだいたい想像できると思います。)

• filltype — 塗りつぶしタイプです。solid(全面塗りつぶし) と amikake(網掛け) の 2 種類に加え，ver3.00 からは grad(グラデーション) があります。指定しないときは塗りつぶしはしません。ただし，次の fillcolor キーに none 以外の値が設定されていて，filltype が指定されていない場合は，filltype を solid と解釈します。**経路** が **巡回経路** でないときは無視されます。

• fillcolor — 塗りつぶしの色です。デフォルトは「描画色」(通常黒) です。**経路** が **巡回経路** でないときは無視されます。

• amidir — 網掛け時の「網」の向きです。デフォルトは 45 (45 度方向の斜線) です。コンマで区切っていくつでも指定できます。たとえば，amidir={45,135} のようにすれば直交する網になります。**経路** が **巡回経路** でないときは無視されます。

○ ¥mptXaxis , ¥mptYaxis ……TEX マクロ

横軸 (x 軸) , 縦軸 (y 軸) を描き，その矢印の先端に x や y などを表示するためのマクロです[*1]。次の型式で使います。

¥mptXaxis|**始点の** x **座標**=y **座標**=**終点の** x **座標**>
　　[**ラベル位置**]<**横補正**, **縦補正**>{**ラベル文字列**}
¥mptYaxis|**始点の** y **座標**=x **座標**=**終点の** y **座標**>
　　[**ラベル位置**]<**横補正**, **縦補正**>{**ラベル文字列**}

ラベル文字列 以外の引数は省略可能です。

以下，¥mptXaxis について引数の意味と，省略したときのデフォルト値を説明します[*2]。

始点の x **座標**,y **座標**, **終点の** x **座標** は矢印の始 **点** ・終 **点** を指定します。横軸ですので，y 座標は始 **点** と終 **点** で共通のはずです[*3]。これらを省略した場合，**始点の** x **座標**, **終点の** x **座標** は MPpic 環境の ¥begin{MPpic} の引数から割り出した値を使います。y **座標** はデフォルトで 0 です。

ラベル文字列 は矢印の先端に配置する文字列です。配置の仕方は [**ラベル位置**] および <**横補正**, **縦補正**> で指定します。これらの意味は ¥mptLabel と同様です。

○ ¥mptClip[key=val **列**] ……TEX マクロ

ある長方形領域外にはみ出した **画像** をクリップするマクロです。なお，クリップされるのは，¥mptClip を指定するまでに書かれた **画像** に限りますので，クリップを指定する順序にご注意下さい。また，¥mptLabel などで配置したラベルは，TEX 側で面倒を見ているのでクリップされません。

key=val **列** に指定できる主要なキーは xmin , xmax , ymin , ymax で，順にクリップ領域の左下の x 座標，右上の x 座標，左下の y 座標，右上の y 座標を表します。省略したときは，MPpic 環境の引数から割り出した値を使います。

[*1] 単に矢印を引くなら ¥mptDrawPath で arrowtype キーを指定した方が早いでしょう。

[*2] ¥mptYaxis については x と y の役割を交換したものとお考え下さい。

[*3] ちなみに上の凡例ではわかりにくいですが，この部分のパターンマッチングは |==> という矢印をイメージしています。

○ kansu **シリーズ** ·····METAPOST マクロ

マクロの書式は,

kansu（**関数式**）(**始**, **終**, **分割数**)
kansuP(**関数式**)(**始**, **終**, **分割数**)
kansuR(**関数式**)(**始**, **終**, **分割数**)

です。関数のグラフ (**経路**) を返します。引数の意味は以下の通りです。

関数式 — グラフを描きたい関数の式を t を変数とした式で書きます。具体的には

kansu — 陽関数 $y = f(t)$ の $f(t)$ の部分
kansuP — パラメータ表示関数 $(x, y) = (f(t), g(t))$ の $f(t)$, $g(t)$ の部分をコンマで区切って
kansuR — 極方程式 $r = f(t)$ の $f(t)$ の部分

始 — t の定義域の始点 (下限)

終 — t の定義域の終点 (上限)

分割数 — グラフは線分で近似します。その分割数です。

○ ついでに — この本のサンプルでは，GFtoDVI の出力をまねて，灰色で描かれた図に描画点 (キーポイント) を打ったものがよく出てきます (たとえば p.*24* の **図 2.1.1**)。これは，線を灰色 (0.7white くらい) で描き，ここに dotLabels というマクロで点とラベルを書き添えることにより実現しています。

dotLabels(**ラベル位置**)(〈 サフィックス 〉**列**) ·····METAPOST マクロ

指定された **点** (例えば z1 , z2a , z3 の 3 点) に現在の **ペン先** で点 (dot) を描き，そのそばに 〈 サフィックス 〉(先の例では 1 , 2a , 3) を書き添えます。添える位置は **ラベル位置** で制御します。使用方法は例えば

```
dotLabels("[t]")(1,2a,3)
```

のようになります。〈 サフィックス 〉列には，コンマで区切っていくらでも **点** の 〈 サフィックス 〉を指定できます。

2 実例

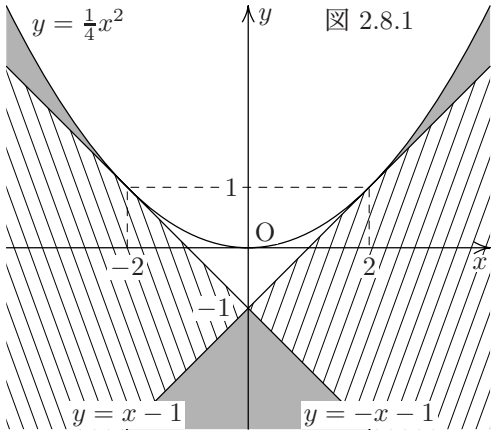


図 2.8.1

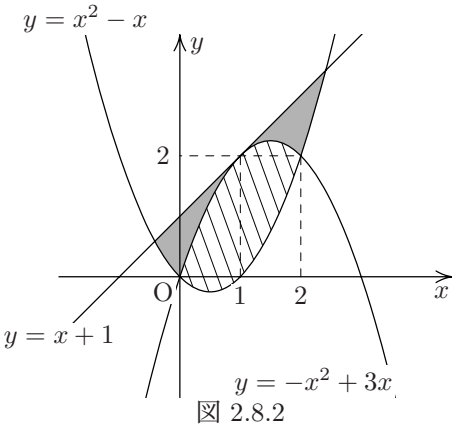


図 2.8.2

以下は **図 2.8.1** のソースです。難しいところは buildcycle を使った領域の取得でしょうか。これについては，**第6講 1項** の説明を再度参照して下さい。

```
¥begin{MPpic}<8mm>(8,0)(-4,4)
¥sendMP{path pth.p, pth.l[], pth.f;
  pth.p  = kansu(1/4t*t)(-4,4,50);%          pth.p は 放物線
  pth.l1 = kansu(-t-1)(-4,4,1);%             pth.l1 は右下がりの直線
  pth.l2 = kansu(t-1)(-4,4,1);%              pth.l2 は右上がりの直線
  pth.f=(0,4h)--(-4w,4h)--(-4w,-3h)--(4w,-3h)--(4w,4h)--cycle;% 枠
}
¥mptDrawPath[fillcolor=0.7white,linetype=none]%       影付き領域 (1)
  {buildcycle(pth.p, reverse pth.l1, subpath(2,0) of pth.f)}
¥mptDrawPath[fillcolor=0.7white,linetype=none]%       影付き領域 (2)
  {buildcycle(reverse pth.p, pth.l2, subpath(3,5) of pth.f)}
¥mptDrawPath[fillcolor=0.7white,linetype=none]%       影付き領域 (3)
  {buildcycle(pth.l1, subpath(3,2) of pth.f, pth.l2)}
¥mptDrawPath[filltype=amikake,amidir=110,linetype=none]%斜線領域 (1)
  {buildcycle(pth.l1, reverse pth.l2, subpath(2.5,0) of pth.f)}
¥mptDrawPath[filltype=amikake,amidir=70,linetype=none]% 斜線領域 (2)
  {buildcycle(pth.l1, subpath(2.5,5)of pth.f, reverse pth.l2)}
¥mptDrawPath[pensize=0.4pt,linetype={dashed evenly}]%          点線
  {(-2w,0)--(-2w,h)--(2w,h)--(2w,0)}
¥sendMP{forsuffixes t=p,l1,l2: draw pth.t; endfor}%      グラフ描画
¥mptXaxis[tr]<0mm,-1mm>{$x$}%                                 x 軸
¥mptYaxis|-3h==4h>[tl]<1mm,0mm>{$y$}%                         y 軸
¥mptLabel{(0,0)}[bl]<0.5mm,0.5mm>{O}%                         原点
¥mptLabel{(0,h)}[r]<-1mm,0mm>{1}%                             座標
¥mptLabel{(-2w,0)}[t]<0mm,-1mm>{$-2$}%                        座標
¥mptLabel{(2w,0)}[t]<0mm,-1mm>{2}%                            座標
¥mptLabel{(0,-h)}[r]<-2mm,0mm>{$-1$}%                         座標
¥mptLabel{pth.p intersectionpoint pth.f}
  [tl]<3mm,0mm>{$y=¥frac{1}{4}x^2$}%                          関数式
¥mptLabel{pth.l1 intersectionpoint (subpath(2,3) of pth.f)}
  [b]{$y=-x-1$}%                                              関数式
¥mptLabel{pth.l2 intersectionpoint (subpath(2,3) of pth.f)}
  [b]{$y=x-1$}%                                               関数式
¥mptClip[ymin=-3h,ymax=4h]%                          はみ出した部分を消去
¥end{MPpic}
```

Exercise 2.8　**図 2.8.2** のソースも考えてみましょう。

第 9 講

ペン先の形状

本講では「**ペン先** の変更」および「曲線の端点・屈曲点の形状の制御」について説明します。たとえば右の例を見て下さい。**図 2.9.1** は角が丸く，**図 2.9.2** は角が鋭くなってます。このような描き分けのためには，METAFONT , METAPOST 共通の方法として，**ペン先** を変更する方法，METAPOST だけの方法として，端点・屈曲点を制御する内部変数を変更する方法があります[*1]。

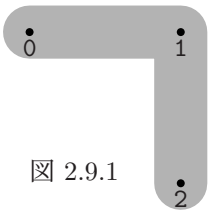


図 2.9.1

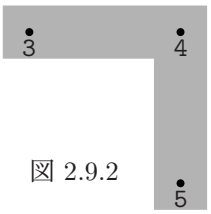


図 2.9.2

1 ペン先の変更

これまでのサンプルは，draw の際に (明示的・暗黙の違いはあれ) 主に

```
pickup pencircle scaled 〜
```

という形で **ペン先** を指定してきました。これは,「直径 1[*2] の円形の **ペン先** を 〜 倍して使え」という命令で，**ペン先** は当然「丸ペン」になります。これで直角に曲がる経路を描けば，**図 2.9.1** のような形状になります。しかし，丸ペンでは都合が悪いとき，あるいは，丸ペンにはない特徴を図形にもたせたいとき[*3]，**ペン先** を変更するという手が使えます。

METAFONT , METAPOST でデフォルトで定義されているのは[*4]

- `pencircle` — 丸ペン：上述のように直径 1 の円
- `pensquare` — 角ペン：1 辺の長さ 1 の正方形
- `penrazor` — カミソリペン：幅 1 厚み 0 の横向き線分

の 3 つです。もちろんこれらは `scaled` , `xscaled` , `yscaled` , `rotated` などで **変形** できますので，楕円，長方形，斜め向きの線分などの形状をもった **ペン先** に変換できます。たいていはこれで事足りると思いますが，必要なら凸多角形の **巡回経路** をマクロ `makepen` に渡して，新たに **ペン先** として登録することもできます。たとえば `penrazor` は

```
penrazor = makepen((-.5,0)--(.5,0)--cycle);
```

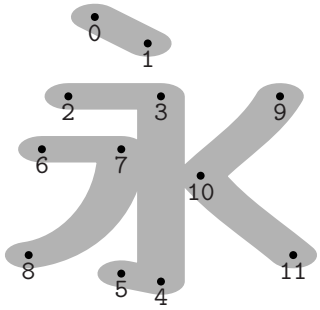
と定義されています[*5]。

ペン先 変更の実例ですが，**図 2.9.2** は

```
pickup pensquare scaled 20pt;
```

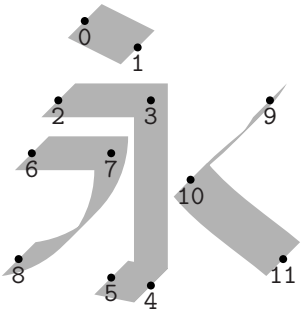
[*1] 端点の形状については METAFONT の方にも若干変更手段があります。
[*2] 単位は METAFONT ならピクセル，METAPOST なら bp です。
[*3] 装飾文字などを想像して下さい。
[*4] これらのうち `pencircle` はプリミティブです。
[*5] この定義は METAPOST のものです。METAFONT の方は，〈 先物ペン 〉というものにするため，定義に少し細工をしてありますが，本質は同じです。なお，〈 先物ペン 〉については，**第3部 第5講** を参照して下さい。

で出力したものです。また，以下の例では，それぞれ図の下に記した **ペン先** を使っています[*6]。



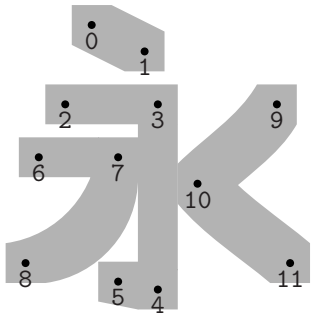

```
pencircle xscaled 18pt
          yscaled 10pt
```

図 2.9.3




```
penrazor rotated 45
         scaled 18pt
```

図 2.9.4




```
pensquare scaled 15pt
```

図 2.9.5

ご覧のように，丸ペン以外のものは，線の方向とペンの向きをうまく調整しないと，非常にみっともないことになりかねないのですが，残念なことに METAFONT や METAPOST では線を描いている途中でペンの向きを変更する命令はありません。しかし，それに似た機能を実現するマクロは備わっています。penpos と penstroke というのがそれで，

```
penpos1(5pt,45)
```

とすれば，見かけ上 **点** z1 での **ペン先** (カミソリペン) の太さが 5pt , 向きが 45° であるかのように振る舞い，

```
penstroke z0e..z1e..z2e
```

などとしたときに，その **ペン先** の大きさ・向きが反映されます。このマクロは，実は **ペン先** を変更しているのではなく，penpos1(5pt,45) で z1 の「両側 (指定の向き) 」に z1l , z1r という **点** を定義し，penstroke で引数の経路の e の部分を l や r に読み替えて，**巡回経路** を作り，それを塗りつぶしているにすぎません。これを使って「永」の字を作り直したのが下図です。(点の位置も若干変えました。) まだまだ,「美しい」文字からはほど遠いものですが，それでも「筆」っぽい線の強弱 (?) は出ていると思います。

```
¥begin{MPpic}<40mm>(1.3,0)(-0.1,1)
¥openMP
z0=(0.25w,h);        z1=z0+(0.2w,-0.1h);
z4=(0.5w,0)=z4a;     z3=z4+(0,0.7h);
z2=z3-(0.35w,0);     z5=z4+(-0.25w,0.06h);
z7=(0.35w,0.5h);     z6=z7-(0.3w,0);
z8=z7-(0.35w,0.4h); z10=(0.6w,0.45h);
z9=z10+(0.3w,0.3h); z11=z10+(0.3w,-0.35h);
z12=z11+(0.2w,-0.05h);
penpos0(3pt,45);  penpos1(22pt,50);  penpos2(13pt,120);
penpos3(23pt,30); penpos4(20pt,0);   penpos4a(23pt,30);
```

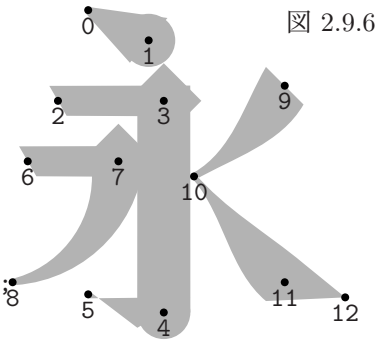


図 2.9.6

[*6] ここでの例が，何で「永」の字かというと，ペン先の変更 → 運筆 → 永字八法 という連想です。— 永字八法 :「永」の字に書道の基本の 8 つの運筆がすべて含まれる — しかし，今回固定ペンなので,「運筆」とよべるものは何もないので，これは全く意味がないですね。

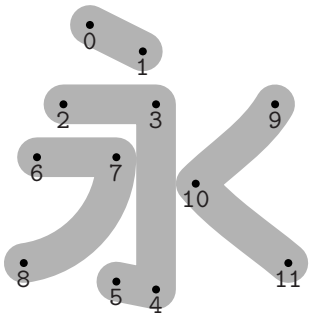
```
penpos5(3pt,-90); penpos6(13pt,120); penpos7(23pt,30);
penpos8(3pt,-45); penpos9(20pt,135); penpos10(3pt,90);
penpos11(20pt,45);
penstroke z0e--z1e;      penstroke z5e--z4a.e;
penstroke z2e--z3e--z4e; penstroke z6e--z7e{down}..z8e;
penstroke z9e{dir-120}..{dir-150}z10e{dir-50}..{dir-40}z11e--z12;
pickup pencircle scaled 20pt; drawdot z1; drawdot z4;
pickup pensquare scaled 20pt rotated 45; drawdot z3; drawdot z7;
¥closeMP
¥end{MPpic}
```

2 内部変数の変更

さて，次は「端点・屈曲点を制御する内部変数を変更」する方法を試してみましょう。PostScript では，曲線の端点や屈曲点の形状について，いくつかのタイプがあります。これらの指定は METAPOST でも有効ですので，それを利用したのが下の例です。下の例ではすべて **ペン先** を

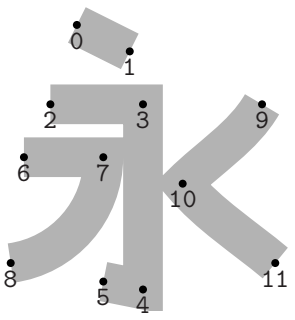
```
pencircle scaled 15pt
```

で描いています。



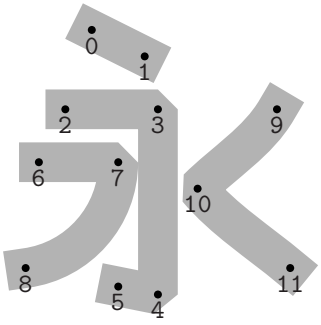

```
linecap:=rounded;
linejoin:=rounded;
```

図 2.9.7




```
linecap:=butt;
linejoin:=mitered;
```

図 2.9.8




```
linecap:=squared;
linejoin:=beveled;
```

図 2.9.9

まず，端点の形状から。これは内部変数 linecap で制御できます。デフォルトは rounded で，端点が **ペン先** の形状にはみ出します。これを butt に変えると端点がばっさりと切られた形になります。また，squared に変えると端点が **ペン先** によらず四角くなります。

次に，屈曲点の形状。これは内部変数 linejoin で制御できます。デフォルトは rounded で，屈曲点が **ペン先** の形状に依存します。これを mitered に変えると屈曲点でくっきりと曲がります。また，beveled に変えると「面取り (?)」したような形状になります。

実際の形状は上の例を見て下さい。なお，あまり鋭角的な屈曲点に mitered を使うと，「角」の部分が大きくなりすぎるので，miterlimit という上限 (デフォルトは 10.0) が設けられています。線の幅と角の部分の長さの比がこれを超えると，mitered を指定していても beveled になってしまいます。

3 METAFONT における端点の形状

METAFONT の方には屈曲点の形状を制御する内部変数はないのですが，端点については butt と似た結果を生むマクロがあります。cutoff というマクロがそれで，たとえば，cutoff(z2,210) とすれば **点** z2 で $(210-90)^\circ$ から $(210+90)^\circ$ に対応するペン先の図形を消去してくれます。また，このマクロを使って定義された cutdraw というマクロもあって，draw した後で線の両端を butt のように進行方向と垂直に切ってくれます。cutdraw を使って描いたのが **図 2.9.10** です。

ソースは以下の通りです。

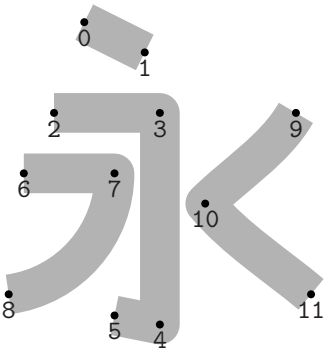


図 2.9.10

```
¥begin{MPpic}<40mm>(1.3,0)(-0.1,1)
¥openMP
  z0=(0.25w,h);        z1=z0+(0.2w,-0.1h);
  z4=(0.5w,0);         z3=z4+(0,0.7h);
  z2=z3-(0.35w,0);     z5=z4+(-0.15w,0.03h);
  z7=(0.35w,0.5h);     z6=z7-(0.3w,0);
  z8=z7-(0.35w,0.4h); z10=(0.65w,0.4h);
  z9=z10+(0.3w,0.3h); z11=z10+(0.35w,-0.3h);
  pickup pencircle scaled 15pt;
  cutdraw z0--z1;
  cutdraw z2--z3--z4--z5;
  cutdraw z6--z7{down}..z8;
  cutdraw z9{dir-120}..{dir-135}z10{dir-50}..{dir-40}z11;
¥closeMP
¥end{MPpic}¥vspace{-¥baselineskip}
```

Exercise 2.9　右図を書いてみましょう。屈曲点の形状や端点の形状に注意して下さい。

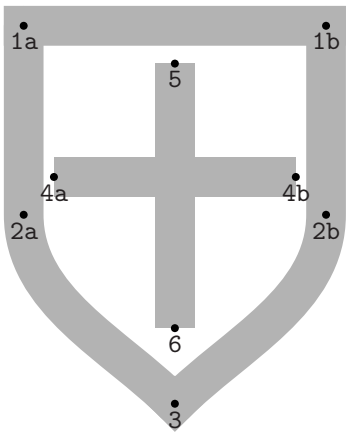


図 2.9.11

第 10 講

マクロ定義

1 マクロ定義

本講では METAFONT 言語のマクロ定義について説明します。さて，TEX がトークン列を扱い，マクロもトークン列の置き換えが中心なのに対し，METAFONT 言語は複数種類のデータを扱い，それらの間の演算を考えたりもしますので，マクロの形式にバリエーションがあります。誤解を怖れずきわめておおざっぱな言い方で，これを分類すると，以下のようになります[*1]。

def	一連の操作をひとまとめにします。TEX でのマクロ定義に近いものです。Pascal でいえば procedure , FORTRAN でいえば SUBROUTINE に近いものです。
vardef	値を返すマクロを定義します。Pascal でいえば function , FORTRAN でいえば FUNCTION に近いものです。引数が 1 個のときは，単項演算子 (dir などがその例) を定めることになります。
primarydef secondarydef tertiarydef	二項演算子 (+ などがその例) を定義できます。primarydef で定義したものは * や / と同格，secondarydef で定義したものは + や - と同格，tertiarydef で定義したものは > や & と同格になります。

二項演算子について少し補足しておきますと，METAFONT 言語では，計算の優先順位 (通常の数学におけるものと同様) というものがあり，たとえば 3+2*5 と書けば，2*5=10 が先に計算され，その後で 3+10=13 が行われる，というように，かけ算は足し算より優先順位が高くなっています。したがって，二項演算子を新たに定義するときは，その優先順位をどの程度にするか決めてから定義しなければなりません。二項演算子の 3 つの定義法はそれぞれ表に示した優先順位の演算子を定義します。

def と vardef の違いは，ちょっとピンと来ないかもしれません。後者は「値を返す」と表には書いてありますが，def でも値を返すマクロを作れますし，vardef でも値を返さないマクロを作れます。そうなった場合 def と vardef の違いはどこにあるのか，ということになりますが，当座はそのことに目をつぶっていいと思います。この 2 者の差異については，**第3部 第1講 1項** および **第9講** を参照して下さい。

2 マクロの引数タイプ

さて，これらマクロ定義は，TEX や他の言語と同じように引数をとれます。引数のタイプは次の 3 つです。ただし，ここでいうタイプとは，変数の種類 (**数値** とか **経路**) **ではな**

[*1] これらの正確な意味については，**第3部 第9講** で扱います。

く，引数に指定されたものの扱い方の種類といった方がいいかもしれません。実際，expr タイプの引数には，**数値** でも **経路** でも，(マクロの定義内容と矛盾しない限り) 何でも代入できます。

expr	通常の引数です。引数を「値」として渡します。expr タイプの引数に未整理の数式を書き込んだ場合，その計算結果が引数として使われます。変数だけでなく，具体的な値を指定できます。
suffix	引数を ⟨ サフィックス ⟩ として扱います。expr と異なり，未整理の計算式など ⟨ サフィックス ⟩ に使えないものは書けません。ただし，マクロ定義内部で，この引数を ⟨ 変数 ⟩ 名に転用することは可能です。要するに引数が値としてではなく ⟨ トークン ⟩ として必要な場合に使います。
text	⟨ トークン ⟩ (命令や値) の並びを指定できます。マクロが呼び出された時点では text 型の引数は評価されず，マクロ内で実際に使われるとき初めて評価されますので，draw のような命令でも，for 文の繰り返し文字列 — たとえば，for n=1,2a,3a: の 1,2a,3a の部分 — など，何でも指定できます。

少し補足します。

```
vardef test(expr c) = c enddef;
```

のように引数をそのまま返すマクロを考えますと，a := test(3+4) とすれば，a には 7 が代入されます。しかし，このマクロの定義の引数部分を

```
vardef test(suffix c) = c enddef;
```

のように変えてしまいますと，もはや c には 3+4 のような未整理の数値は使用できなくなります[*2]。

3 マクロの実例 (1)

さて，実際の定義ですが，たとえば次の例のようになります。

```
def drawTyusen(expr a,b,c) =
  draw a--b--c--cycle;
  draw a--(0.5[b,c]);
  draw b--(0.5[c,a]);
  draw c--(0.5[a,b]);
enddef;
```

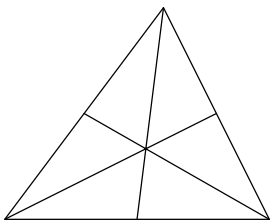


図 2.10.1

だいたい想像はつくと思いますが，これは与えられた 3 点を元にして，三角形とその 3 本の中線を描くマクロです。マクロはこのように，

⟨def など ⟩⟨ マクロ名 ⟩(**引数の列**) =

で始まり[*3]，

```
enddef;
```

[*2] 逆に，suffix タイプの引数 — 引数を ⟨ トークン ⟩ として扱えるのこと — のメリットについては以下の例を参照して下さい。また，**第3部 第9講 6項** でも，引数のタイプの違いについて詳しく説明します。

[*3] ( ) をつけない引数の指定方法もあります。これについては **第3部 第9講** を参照して下さい。なお，def の行にある引数は，正確には ⟨ 仮引数 ⟩ とよばれるものです。

で終わります。これで定義された drawTyusen は，たとえば

```
drawTyusen(z1,z2,z3);
```

のように使います。つまりコンマで区切られた z1 , z2 , z3 が，それぞれ a , b , c に渡されます。META 言語では)(は , と同じであると解釈されますので,

```
drawTyusen(z1)(z2,z3);  や  drawTyusen(z1)(z2)(z3);
```

も同じ意味になります。例外は text タイプの引数で，この引数は上記のようにコンマ付き文字列全体も引数とできますので，コンマではなく ( ) で区切る必要があります。

上の drawTyusen の引数は expr タイプですので，具体的な座標を指定することもできます。たとえば，上図は

```
drawTyusen((0,0),(5w,0),(3w,4w));
```

として描画させたものです。

suffix タイプの引数の活用例も挙げましょう。

```
enws(z0,1cm);
```

とすれば，**点** z0 の右・上・左・下の指定された距離 (上記の例なら 1cm) の位置に z0e , z0n , z0w , z0s という **点** を定義すると同時に，これらを結んで正方形を描くマクロを定義してみます。この場合，引数の z0 というトークン (この場合 **点** の名前) をもとに，新たな **点** の名前を作るので，z0 の値だけ渡しても意味がありません。こういうときに suffix タイプを使います。もちろん，マクロ定義内で z0 の実際の値を使うこともできます。実際の値は，マクロ定義内で z0 の値を使うとき初めて評価されます。下が定義，**図 2.10.2** が使用例です。

```
def enws(suffix zz)(expr len) =
  zz.e = zz + len*right; zz.w = zz + len*left;
  zz.n = zz + len*up;    zz.s = zz + len*down;
  draw zz.e--zz.n--zz.w--zz.s--cycle;
enddef;
```

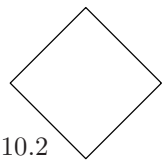


図 2.10.2

次は，text タイプの引数をもつマクロを考えてみましょう。上の enws の発展形で，新しく設定する **点** の方向は，方向角をコンマで区切って指定することにしましょう。(新しく設定する **点** の名前は，z0v1 , z0v2 , ... としましょう。) この場合，コンマで区切られた方向角の列を**そのまま** for 文に渡したいので，text タイプの引数の出番となります。下が定義，**図 2.10.3** が使用例です。この図は,

```
SUPERenws(z0,1cm)(0,60,120,270);
```

で出力しています。

```
def SUPERenws(suffix zz)(expr len)(text drc) =
begingroup
  save k; numeric k; k:=0;
  for n=drc: k:=k+1; zz.v[k] = zz + len*dir(n); endfor
  draw for n=1 upto k: zz.v[n] -- endfor cycle;
endgroup
enddef;
```

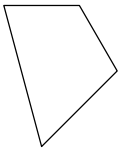


図 2.10.3

Exercise 2.10.1　x_1 , x_2 , x_3 （ $x_1 < x_2 < x_3$ ）に対する y の値 y_1 , y_2 , y_3 を与えて，これらを満たす二次関数 $y = ax^2 + bx + c$ のグラフを $x_1 \leq x \leq x_3$ の範囲で描くマクロを定義してみましょう。簡単のため $x_1 \leq x \leq x_3$ の区間を 50 等分し，対応する **点** を -- で結び，scaled w でサイズ調整することにします。座標軸などはいりません。

4 引数のスコープ

前項は引数のタイプを中心に扱いましたが，引数の有効範囲（スコープ）については説明していませんでした。そこで，次のような（あまり意味のない）マクロを考えます。

```
def addone(expr a) =
begingroup
  save b;
  b:=a; b:=b+1; c:=c+1;
endgroup
enddef;
```

やっていることはわかると思いますが，引数の a を b に代入してこれに 1 を加算，さらに変数 c にも 1 を加算しています。このマクロを次のように使った場合，show の時点で a , b , c の値はいくらでしょうか。

```
a:=100; b:=4; c:=5;
addone(9);
show a,b,c;
```

答は，a=100 , b=4 , c=6 です。マクロの定義には a が引数として使われていますが，これはマクロ定義内だけのローカルな変数で，マクロの外の a とは無関係ですし，そもそもマクロ内で expr タイプの引数に値を代入することは許されていませんので，a の値は変更を受けません。また，b は，グルーピング内で save されていますので，ローカル変数（プライベート変数）のごとく働きます。つまり，save を指定した時点での b の値 4 が一旦保存され，グルーピング（begingroup 〜 endgroup）を抜けたときその値が復活しますので，グルーピング内の変更の影響は受けません。これらに対し，c は引数でもなく save もされていませんので，もろに値が変更されます。なお，plain.mf や plain.mp では，いちいち save を使うと非効率なので，末尾が _ で終わる変数はユーザーが使わないという仮定の下でこの形式の変数（k_ など）をプライベート変数として使ってます。「ユーザーが使わない」「plain.mf や plain.mp 内で衝突がない」の 2 つの条件が満たされていれば，いちいち save しなくていいという理屈です。したがって，一般のユーザーは，末尾が _ で終わる変数を使わない方が得策でしょう。ちなみに拙作の MEPOTEX は先頭が _ の変数をプライベート変数として使っていますが，こちらはきちんと save していますので，衝突の心配はありません。（たぶん（^_^;））

5 マクロの実例 (2)

次は，vardef の使用例を挙げましょう。前々項で，指定した点を中心として，指定した方向の，指定した長さの位置に点を取っていき，それらを結んで多角形を描くというマクロ（SUPERenws）を作りましたが，これを改造して，多角形の **経路** を返すようにしてみ

ましょう。このようにしておけば，単に多角形を描く (draw) だけでなく，多角形を塗りつぶす (fill SUPERenws(z0,1cm)(0,60,120,270); などとする) といった，臨機応変な使い方ができてより便利です。以下が改造した定義です。改造前の定義 (p.**62**) と見比べてみて下さい。

```
vardef SUPERenws(suffix zz)(expr len)(text drc) =
  save k,pth; path pth; k:=0;
  for n=drc: k:=k+1; zz.v[k] = zz + len*dir(n); endfor
  pth:= for n=1 upto k: zz.v[n] -- endfor cycle;
  pth
enddef;
```

経路 変数 pth を導入したことは本質的ではありません。改造前と後の違いは，def が vardef になったこと，begingroup と endgroup がなくなっていること，定義の最後に ; のない行があること，の 3 点です。begingroup と endgroup がなくなっているのは，vardef では定義の最初と最後に begingroup と endgroup が暗黙のうちに入るため，これらを明示的に入れる必要がないからです。METAFONT 言語では，

begingroup ⟨ 文 a⟩; ⟨ 文 b⟩;……; ⟨ 文 c⟩; ⟨ 文 d⟩ endgroup

とすると，(⟨ 文 a⟩; ⟨ 文 b⟩;……; ⟨ 文 c⟩; を順に実行した上で，) あたかもグルーピング内に ⟨ 文 d⟩ (; がない行) のみがあるかのようにふるまいます。したがってこの形式をマクロの定義に入れれば,「値を返すマクロ」ができるわけです。

最後に，primarydef の使用例を挙げましょう。METAPOST では **色** が 3 次元ベクトルで表されます。これを，数学で扱う通常の 3 次元ベクトルと見て，ベクトルの外積 gaiseki を定義してみます[*4]。定義は以下のようになります。これは実際 MEPOTEX で採用されている定義です。

```
primarydef _w gaiseki _z =
  (greenpart _w * bluepart  _z - bluepart  _w * greenpart _z ,
   bluepart  _w * redpart   _z - redpart   _w * bluepart  _z ,
   redpart   _w * greenpart _z - greenpart _w * redpart   _z)
enddef;
```

外積は優先順位でかけ算と同格ですので，primarydef を使いました。

マクロ定義に関してのおおよその話はこれで終わりです。マクロ定義に関しては，まだまだ深い内容があるのですが，これは **第3部 第9講**で扱うことにします。

Exercise 2.10.2 3 点を与えて，その 3 点で形成される三角形の外心を返すマクロを作ってみましょう。ヒントは，whatever (使うたびに新しく使い捨ての **数値** 変数を生成するマクロ) と dir (引数に指定された方向の単位ベクトルを返すマクロ) および angle (引数のベクトルの方向角を返すプリミティブ) をうまく使うこと，でしょうか。

[*4] 外積を知らない人のために説明しておきますと，
$(a,b,c)\times(x,y,z)=(bz-cy,cx-az,ay-bx)$
という演算です。なお，**色** 変数の各成分は順に redpart , greenpart , bluepart でアクセスできます。

第 3 部 文法編(中級〜上級)

これまでにも，必要な文法はこまめに書いてきましたが，
さらなる高みに登るには，やはりもっと精密な文法を知るべき，
と言うことで，
第1部，**第2部** よりは立ち入ったところまで踏み込もうと思っています。
なお，話の流れの関係で，以前の話と重複する部分も出るでしょうが，
重複しつつも内容は深くなっていますので，その点ご了承下さい。

第 1 講 トークンと変数名，宣言

1 トークン・スパーク・タグと変数名

METAFONT 言語のソースを見ると 2/3x1a2 みたいに，どこからどこまでが変数か，初心者にはわかりづらい形式を見かけます[*1]。こういった形式を理解するには，METAFONT 言語がソースを読み込む規則をある程度理解しておく必要があります。

ご存知の通り TEX ソースは，文章を構成する文字と，それを処理する命令などからなり，これら文字，命令などをひっくるめて「トークン」とよんでいます。同様に，METAFONT 言語のソースも「トークン」とよばれる構成要素からなっています。ただし TEX と違い，1 文字ずつ〈トークン〉になるとは限りません。

METAFONT 言語の〈トークン〉は以下のように分類されます。

- トークン token
  - 文字列トークン string token — " " で囲まれた文字列 (例) "Hello"
  - 数値トークン numeric token — 数値 (詳しい規則は次項で) (例) 3.14
  - 記号トークン symbolic token
    - スパーク — プリミティブとマクロ (例外あり)
    - タグ — その他の記号トークン

プリミティブというのは METAFONT 言語に組み込まれている基本命令のことで，end や angle などがその例です。マクロはそれらを組み合わせて作った命令のことで，fill や draw などがその例です。これらは上の分類でいけば〈スパーク〉になるのですが，例外として「vardef で定義されたマクロは〈タグ〉になる」という規則があります。よく使う例では dir などがそうです[*2]。1 つ例を挙げましょう。

```
beginchar("A",10pt#,5pt#,5pt#);
␣␣for␣n␣=␣0␣upto␣3:
␣␣␣␣z[n]␣=␣(r,r)␣+␣r*dir(90n);
␣␣endfor
```

を，文字列トークン，数値トークン，スパーク，タグ に分類すると，次のようになります。

```
beginchar ( "A" , 10 pt # , 5 pt # , 5 pt # ) ;
  for n = 0 upto 3 :
    z [ n ] = ( r , r ) + r * dir ( 90 n ) ;
```

[*1] ちなみにこの例は x1a2 が〈変数名〉でそれの 2/3 倍という意味です。

[*2] **第2部 第10講** で保留した「def と vardef の違い」ですが，端的に言えば，〈スパーク〉を定義するのが def で，〈タグ〉を定義するのが vardef ということになります。(**第9講** でより厳密な説明を与えます。)

endfor

上で空白 ␣ が無視されていることに注意して下さい。META 言語では，空白および「後ろに 0 〜 9 やピリオドが続かない」ピリオドは，⟨ トークン ⟩ の区切りとして機能し，読み込みの際削除されます。ただし削除されるといっても，その前後の ⟨ トークン ⟩ が 1 つになるわけではありません。x.a や x␣a は x a という 2 つの ⟨ タグ ⟩ からなる ⟨ 変数名 ⟩，xa は xa という 1 つの ⟨ タグ ⟩ からなる ⟨ 変数名 ⟩ です。

さて，ご覧のように，⟨ 変数名 ⟩ (r や z[n] など) は 1 つまたは複数の ⟨ トークン ⟩ からできているわけですが，その基幹となるもの (先頭の ⟨ トークン ⟩) は ⟨ タグ ⟩ です。⟨ 変数名 ⟩ の付け方の正確な規則は以下の通りです[*3]。

⟨ 変数名 ⟩ ⟶ ⟨ タグ ⟩⟨ サフィックス ⟩
⟨ サフィックス ⟩ ⟶ ⟨ 空 ⟩ | ⟨ サフィックス ⟩⟨ 添字 ⟩ | ⟨ サフィックス ⟩⟨ タグ ⟩
⟨ 添字 ⟩ ⟶ ⟨ 数値トークン ⟩ | [⟨ 数値式 ⟩]

…… って，なんのことかわかりづらいと思いますが。

たとえば，先頭の ⟨ タグ ⟩ に xa を使ったとして上の規則をなぞると，

⟨ 変数名 ⟩ ⟶ xa⟨ サフィックス ⟩⟶ xa⟨ 空 ⟩⟶ xa
⟨ 変数名 ⟩ ⟶ xa⟨ サフィックス ⟩⟶ xa⟨ サフィックス ⟩⟨ 添字 ⟩⟶ xa⟨ 空 ⟩2 ⟶ xa2
⟨ 変数名 ⟩ ⟶ xa⟨ サフィックス ⟩⟶ xa⟨ サフィックス ⟩⟨ タグ ⟩⟶ xa⟨ 空 ⟩r ⟶ xa.r

のように，許される ⟨ 変数名 ⟩ の形式が構成できます。⟨ サフィックス ⟩ の定義は再帰的[*4]ですので，いくらでも長い ⟨ 変数名 ⟩ を定義できます。たとえば，xa2a55pq3iii9 とか — まあ，こんな ⟨ 変数名 ⟩ を使う人はいないでしょうが。あと，上述の規則によれば，xa␣r は xa.r と同じ意味になりますが，xa␣r ではコンピュータには理解できても人間には見づらい形ですので，やめた方がいいでしょう。

2 数値トークン

0 〜 9 からなる (空白を含まない) 数字の列を ⟨ 数字列 ⟩ とよぶことにすれば，⟨ 数値トークン ⟩ は以下のように表現できます。

⟨ 数値トークン ⟩ ⟶ ⟨ 数字列 ⟩ | .⟨ 数字列 ⟩ | ⟨ 数字列 ⟩.⟨ 数字列 ⟩

要するに，123 とか .01 とか 3.14 とかのことです。これらの意味はおおよそ通常の数値と同じで，順に 123 , 0.01 , 3.14 の意味です。「おおよそ」といったのは，META 言語で扱える数値が 4096 以下で $\frac{1}{65536}$ の整数倍の値だけだからです。そのため 0.01 と 0.00999 は，ともに最も近い $\frac{1}{65536}$ の倍数である $\frac{655}{65536}$ に解釈されます[*5]。

なお，意外かもしれませんが，負の数は ⟨ 数値トークン ⟩ ではありません。たとえば，-3.14 は，⟨ 数値トークン ⟩ 3.14 に ⟨ スパーク ⟩ の - がついたものです。この場合の ⟨ スパーク ⟩ - の意味は，後ろの数値の -1 倍を作る，という意味ですので，意味は -3.14 と同じなのですが，もはや ⟨ 数値トークン ⟩ ではなく ⟨ 数値式 ⟩[*6] ですので，たとえば [ ]

[*3] ⟨ 添字 ⟩ と ⟨ サフィックス ⟩ の違いにご注意下さい。META 言語では ⟨ 添字 ⟩ は数値または数値式の ⟨ サフィックス ⟩ です。
[*4] ⟨ サフィックス ⟩ の定義の中にまた ⟨ サフィックス ⟩ があることを指します。
[*5] したがって 0.01=0.00999 は真です。
[*6] 詳しくは **第2講 2項** を参照して下さい。

なしでは〈添字〉に使えません。つまり x[-3.14] では $x_{-3.14}$ の意味になりますが，[ ] を省略した x␣-3.14 では $x-3.14$ という違う意味になってしまいます。

3 ソース読み込み規則

では，ここで METAFONT 言語がソースファイルを読み込むとき，どのように〈トークン〉の切り分けが行われているかについて見ていきましょう。なお，METAFONT 言語の文法では，プログラムの 1〈文〉は；で区切られた部分 (ソース上何行にわたっていても，；が現れるまでは同一の〈文〉) で，この〈文〉のことを「行」と混用することもありますが，この項で言う「行」は，METAFONT 言語がソースの意味をとる前に〈トークン〉を取り込む段階での話ですので，正味ソースファイルの 1 行のことです。

あと，以下の内容は METAFONT ブックの内容をもとに自分なりにまとめ直したものです。ちょっとオリジナルの説明と順序が違ってますが，本質的には同じだと思います。

(1) まず，" が現れたら，同一行で次に現れる " までを〈文字列トークン〉とします。もし同一行に対応する " がなければ，

```
! Incomplete string token has been flushed.
```

というエラーが出ます。

(2) 上記を除いて，つまり対になっている 2 つの " の間にある場合を除いて，% とその行のそれ以降の文字は無視します。要するに，TEX と同じく % 以降はコメントになるわけです。

(3) 以降は上記 2 つの場合 — 対になった 2 つの " の間と % の後ろ — を除いての話です。

(i)「空白」や「『後ろに 0 〜 9 もピリオドも続かない』ピリオド」は，〈トークン〉の切れ目として作用し，読み込みテキストからは除かれます。

(ii) (と) と , と ; は 1 文字だけで〈記号トークン〉(より絞った言い方にすれば〈スパーク〉) になります。

(iii) 0 〜 9 や,「『後ろに 0 〜 9 が続く』ピリオド」は，〈数値トークン〉を形成します。〈数値トークン〉の規則については前項で説明したとおりです。METAFONT 言語は，行の初めから読んでいって，なるべく長く〈数値トークン〉をとります。

(iv) 上記以外は，〈記号トークン〉(〈スパーク〉と〈タグ〉) を形成します。METAFONT 言語は，行の初めから読んでいって，以下の規則に従って，なるべく長く〈記号トークン〉をとります。

- ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz のうちの 1 個または複数個からなる文字列。
- <=>:| のうちの 1 個または複数個からなる文字列。
  (たとえば < や <= という〈スパーク〉があります。)
- `' のうちの 1 個または複数個からなる文字列。
- +- のうちの 1 個または複数個からなる文字列。
  (たとえば + や ++ という〈スパーク〉があります。)
- /*¥ のうちの 1 個または複数個からなる文字列。
- !? のうちの 1 個または複数個からなる文字列。
- #&@$ のうちの 1 個または複数個からなる文字列。

- ~~ のうちの 1 個または複数個からなる文字列。
- [1 個または複数個からなる文字列。
-] 1 個または複数個からなる文字列。
- {} のうちの 1 個または複数個からなる文字列。
- 複数個のピリオド (.) の並び。

この規則からわかると思いますが，x.a という ⟨ 変数名 ⟩ に倣って x.1 としても x1 の意味にはなりません。x.1 では，ピリオドの後ろに数字がありますので，このピリオドは ⟨ トークン ⟩ の区切りではなく小数点になってしまいます。つまり，x.1 は，x[0.1] や x0.1 と同じ意味になってしまいます。

Exercise 3.1.1　次の式を，⟨ 数値トークン ⟩，⟨ スパーク ⟩，⟨ タグ ⟩ に切り分けてみましょう。

```
xa:=1.23;
xb=-2.34;
xc=sind30␣+␣(xa+xb)*cosd60;
```

Exercise 3.1.2　次の「意味のない」文字の羅列を META 言語に与えたら，どのように切り分けて読み取るでしょうか。

```
xa100.3b.m#?+--<[[^]//m␣mn....{dir30};
```

Exercise 3.1.3　⟨ 変数名 ⟩ xa2a55pq3iii9 を定義にしたがって ⟨ トークン ⟩ に分解していくと，次のようになります。

⟨ 変数名 ⟩ xa2a55pq3iii9 ⟶ 「⟨ タグ ⟩ xa」＋ 「⟨ サフィックス ⟩ 2a55pq3iii9」
⟨ サフィックス ⟩ 2a55pq3iii9 ⟶ 「⟨ サフィックス ⟩ 2a55pq3iii」＋ 「⟨ 添字 ⟩ 9」
⟨ サフィックス ⟩ 2a55pq3iii ⟶ 「⟨ サフィックス ⟩ 2a55pq3」＋ 「⟨ タグ ⟩ iii」
⟨ サフィックス ⟩ 2a55pq3 ⟶ 「⟨ サフィックス ⟩ 2a55pq」＋ 「⟨ 添字 ⟩ 3」
⟨ サフィックス ⟩ 2a55pq ⟶ 「⟨ サフィックス ⟩ 2a55」＋ 「⟨ タグ ⟩ pq」
⟨ サフィックス ⟩ 2a55 ⟶ 「⟨ サフィックス ⟩ 2a」＋ 「⟨ 添字 ⟩ 55」
⟨ サフィックス ⟩ 2a55pq ⟶ 「⟨ サフィックス ⟩ 2」＋ 「⟨ タグ ⟩ a」
⟨ サフィックス ⟩ 2 ⟶ 「⟨ 空 ⟩」＋ 「⟨ 添字 ⟩ 2」

これにならって，次の ⟨ 変数名 ⟩ を分解してみましょう。ym.n25.31pq

Exercise 3.1.4　(通常の状態で) a.+ という名前の ⟨ 変数名 ⟩ を使えるでしょうか。もしだめならその理由は何でしょうか。

4 変数の種類

METAFONT には，**numeric** , **pair** , **path** , **pen** , **picture** , **boolean** , **transform** , **string** の 8 つのタイプがあり，METAPOST にはこれに加えて **color** というタイプがあります。本当はもう 1 つ **空** というタイプがあるんですが，これは文字通り「空」という 1 つの値しかとらないので，とくに説明すべきことはありません。

上記タイプは，これまで状況に合わせて適当に日本語に翻訳してきました[*7]が，**第3部**以降では表現を厳密にするため，本来の名前で統一したいと思います。

なお，それぞれのタイプの意味はもうおわかりとは思いますが，念のため一覧表を挙げておきます。

- **numeric** ···· 数値
- **pair** ····· 点または 2 次元ベクトル
- **color** ····· 色または 3 次元ベクトル
- **path** ····· 経路，曲線
- **pen** ····· ペン先
- **picture** ····· 画像
- **boolean** ····· 論理式 (真か偽)
- **transform** ·· 変形 (affine 変換)
- **string** ····· 文字 (列)

これらの変数の使い方などについては，次講以降で個別に扱っていきます。

5 変数の宣言 (1)

META 言語で変数を使うには，まずその〈変数名〉を宣言しなければなりません。〈変数名〉の宣言のために使うコマンドは，変数のタイプ名そのもので，たとえば，a , b , c の 3 つを **pair** 値変数として宣言したいならば，以下のようにします。

```
pair a,b,c;
```

例外は **numeric** 値変数で，これは宣言してもしなくても使えます。逆に言えば，宣言を忘れた変数はすべて **numeric** と見なされてしまう，ということですので

```
! Equation cannot be performed (numeric=pair).
```

などというエラーが出たら，宣言忘れを疑って下さい。

なお，変数の宣言をすれば，もしその変数がこれまで何かの意味をもっていたとしても，その意味は捨て去られて初期化されてしまいます[*8]。

META 言語では，値の宣言 (初期化) や代入などによる変更は，**とくに断りのない限りグローバル**ですので，特定の範囲のみに宣言や代入を限定したいときは，その範囲を begingroup と endgroup で囲んで，その中で変更を限定したい (グルーピング外での意味を保存したい) 変数を save してから使うことになります。たとえば，先述の a , b , c の 3 つをローカルに **pair** 値変数として宣言するには

```
begingroup
  save a,b,c;
  pair a,b,c;
    ( a,b,c を使う各種操作 )
endgroup
```

などとします。

6 変数の宣言 (2)

前項では pair a,b,c; で a , b , c の 3 つの変数を **pair** 値変数にできると言いましたが，この宣言を行っても，a に〈サフィックス〉がついた a1 や a.r などは宣言できたこ

[*7] たとえば **pair** は **点**や **ベクトル** と表現してきました。

[*8] したがって numeric は，変数の宣言という意味ではあってもなくても同じなのですが，変数の初期化という意味では欠くことのできないコマンドです。

とになりません。ここではこの部分について補足したいと思います。

まず，おさらいです。「サフィックス」と「添字」は，数学などではほぼ同じ意味に使われるのですが，METAFONT 言語の言語規則においては，後者は前者の特殊なものに限定されます。以下，⟨ 添字 ⟩ は **第1講 1項** の意味，すなわち，⟨ 数値トークン ⟩ もしくは [⟨ 数値式 ⟩] の形のサフィックスととって下さい。⟨ サフィックス ⟩ は ⟨ 添字 ⟩ も含めた一般のサフィックスです。

さて，METAFONT 言語では，⟨ サフィックス ⟩ の異なる ⟨ 変数名 ⟩ を，別々のタイプの変数にできます。たとえば，a は **pair** , a.n は **numeric** , a.p は **path** といった具合です。ただし例外がありまして，「⟨ 添字 ⟩ 以外が同じ ⟨ 変数名 ⟩ は同じタイプでなければならない」ということになっています。たとえば，a1 が **pair** なら，a2 も a3 も **pair** でなければなりません。a1p が **path** なら，a2p も a3p も **path** でなければなりません。これは制限ととらえると窮屈に思うかもしれませんが，実際のところ，数値の ⟨ 添字 ⟩ は，ある特定の型の変数の配列に使うのが普通ですので，「(個数も不明なことが多い) 配列の宣言を一括ですませるため」と考えれば，妥当な規則といえます。

というわけで，⟨ サフィックス ⟩ つきの ⟨ 変数名 ⟩ の宣言は，次の例のようになります。

(例) a は **pair** , a.n は **numeric** , a1 も **numeric** , a1p は **path** としたいとき

```
pair a;
numeric a.n , a[];
path a[]p;
```

上記のように，⟨ 添字 ⟩ 部分は [] に代えて宣言します[*9]。この宣言で，a2 , a5 などは同時に **numeric** に，a[-1]p , a3p などは同時に **path** になります。

配列を 2 次元にすることもできます[*10]。宣言の仕方はだいたい想像がつくと思いますが，

```
numeric a[][];
```

とすれば，二次元配列 a[1][1] , a[1][5] , a[3][2] , etc が **numeric** 値変数として使えるようになります。

なお，a[1][5] で [] を省略するにはどうすればいいでしょう。a15 では a[15] の意味になりますし，a1.5 では a[1.5] の意味に，a.1.5 では a[0.1][0.5] の意味になります。正解は a1␣5 のように，「小数点と間違われない ⟨ トークン ⟩ の区切り」である空白を入れることです。ただ，これは見にくいので多用しない方がいいかもしれません。他に a1[5] や a[1]5 や a1¥5 も a[1][5] の意味になります。最後の例に使った ¥ はプリミティブで，「空」に展開されますので，⟨ トークン ⟩ の区切りとして使えます。

Exercise 3.1.5　a=5 , b=3 , c15=2 , c[5][3]=6 であるとき，次のそれぞれは，「値の定まった変数」「値の定まっていない変数」「エラーが出る形式」のいずれでしょうか。また，「値の定まった変数」のときはその値を，「エラーが出る形式」のときはどういうエラーが出るかを考えてみましょう。

c[a][b] , c[a*b] , c[ab] , c.a.b , c␣a␣b , c(a)(b)

[*9] numeric a1; は

```
! Illegal suffix of declared variable will be flushed.
```

のエラーになってしまいます。

[*10] 2 次元配列は，たとえば行列の成分を表現するときに使います。

第 2 講

計算の優先順位，numeric

1 計算式の優先順位

数学では普通，演算に優先順位が定められていて，それにしたがって計算順序を決定しています。たとえば，

$$2+3x>(-x+2)\times 6\sin 30^\circ$$

という数式において，左辺では $2+3$ より $3x$ (x の 3 倍) の方が先に計算され，右辺では ( ) で囲われた部分や $\sin 30^\circ$ の部分がかけ算 (6 と sin の間の暗黙の積) より先に計算されます。(もっとも，$6\times\sin$ では計算しようがないんですけど。) そして，左辺と右辺の計算が終わってから，不等号 (というより不等式) の真偽が判定されます。たとえば，$x=1$ ならこの式は真，$x=-1$ ならこの式は偽です。

通常のプログラム言語では，関数の引数は ( ) で囲わなければなりませんし，$3x$ という「暗黙のかけ算」は許されていませんので，上記の数式は，たとえば次のように記述しなければなりません。

```
2+3*x>(-x+2)*6*sind(30)
```

しかし，METAPOST 言語ではこのあたりがかなり緩和されていて，以下のような通常の数式に近い書き方が許されています。

```
2+3x>(-x+2)*6sind30
```

この式を例に，METAPOST 言語の計算における優先順位を見ていきましょう。

METAPOST 言語において，式は優先順位の高い順に ⟨1 位式⟩ , ⟨2 位式⟩ , ⟨3 位式⟩ , ⟨ 式 ⟩ (4 位) に分類されます。

⟨1 位式⟩ は ( ) で囲まれた部分，および，sind30 のような ⟨ 数値演算子 ⟩ とその ⟨ 引数 ⟩ の部分を合わせたものなどです。⟨2 位式⟩ は * , / のような積・商およびそれと同等と定義されている (primarydef で定義されている) 演算で結ばれた式，⟨3 位式⟩ は + , - のような和・差およびそれと同等と定義されている (secondarydef で定義されている) 演算で結ばれた式，⟨ 式 ⟩ は < , > のような関係演算子およびそれと同等と定義されている (tertiarydef で定義されている) 演算で結ばれた式です。このことから，⟨ 数値演算子 ⟩ とその ⟨ 引数 ⟩ の結合が優先順位が最も高い「演算」で，以下，積・商の類が優先順位 2 番目の「演算」，和・差の類が優先順位 3 番目の「演算」，不等式評価などが優先順位 4 番目の「演算」という言い方もできます。

なお，上位 (位の小さい式) はそのまま下位の式としても使えます[*1]。また，式はさらにその計算結果の種類によって，⟨ 数値式 ⟩ , ⟨ ペア式 ⟩ 等に分類されます。ちなみに今回

[*1] つまり上位の式の集合は下位の式の集合に包含されます。最も広い集合が ⟨ 式 ⟩ です。

の例の場合，⟨3 位式 ⟩ のレベルまでは ⟨ 数値式 ⟩ (計算結果が **numeric**) ですが，式全体では ⟨ 論理式 ⟩ (計算結果が **boolean**) です。

さて，ここでいくつか注意すべき点を列記します。

○ 3x と 3*x は，数値としてはどちらも $3x$ に等しく，(-x+2) の部分の -x と -1*x は，数値としてはどちらも $-x$ に等しいのですが，計算の優先順位は異なります。3x や -x のように，数値変数などに直接くっつけて書かれた ⟨ 数値トークン ⟩ や + , - は sind などと同じ ⟨ 数値演算子 ⟩ となり，最高の優先順位となります。なお，a␣x は a.x つまり a にサフィックスの x がついたもの，ax は，ax で 1 つの ⟨ 変数名 ⟩ となり，いずれも a と x の積の意味にはならないことに注意して下さい。あくまで 3 などの ⟨ 数値トークン ⟩ や符号が，前にかかっているときだけです。あと，2␣2 は 2×2 にはなりません。⟨ 数値トークン ⟩ を直接かけられるのは,「後ろ」が ⟨ 数値トークン ⟩ **でない**場合と定められています。2(2) は (2) が ⟨ 数値トークン ⟩ ではなく ⟨ 数値式 ⟩[*2] ですので 2×2 です。

○ ⟨ 数値演算子 ⟩ の ⟨ 引数 ⟩ は ⟨ 数値 1 位式 ⟩ が要求され，⟨ 数値演算子 ⟩ の後に続く式を ⟨1 位式 ⟩ の範囲内で最も長く ⟨ 引数 ⟩ として採用します。したがって，6sind2x では 2x 全体を sind の ⟨ 引数 ⟩ として採用し，sind2x 全体を 6 (暗黙の積) の ⟨ 引数 ⟩ として採用します。したがって，計算順序としては 2x (2 と x の間の積) が先に行われた後，その計算結果が sind に渡され，さらにそれが 6 倍されることになります[*3]。

○ 優先順位が同じ二項演算は，左から順に行われます。これはたとえば，⟨3 位式 ⟩ を作る ⟨ 二項演算子 ⟩ + が前後の式[*4]に

⟨3 位式 ⟩ + ⟨2 位式 ⟩

という位数を要求する仕様になっているからです[*5]。つまり，a+b+c において左側の + の ⟨ 引数 ⟩ は a と b , 右側の + の ⟨ 引数 ⟩ は a+b と c であるため，右側の + より左側の + が先に行われることになります。同様に ⟨2 位式 ⟩ を作る演算 * や ⟨ 式 ⟩(4 位) を作る演算 < は

⟨2 位式 ⟩ * ⟨1 位式 ⟩

⟨ 式 ⟩ < ⟨3 位式 ⟩

という位数を要求します。

○ 同じ記号でも出現位置によって意味や優先順位が違うことがあります。とくに注意しなければならないのは + と - と / です (以下で説明します)。

○ + と - には上記のように ⟨ 数値演算子 ⟩ として最高の優先順位をもつ場合と，通常の足し算・引き算として 3 番目の優先順位をもつ場合があります。たとえば (-x+2) の - は ⟨ 数値演算子 ⟩ で，(2-x) の - は引き算です。要は「符号」か「二項演算子」かの違いですので，こちらは意味が取りやすいと思います。

○ 問題は / です。これは通常割り算で，優先順位は 2 番目なのですが，1 つ例外があり

[*2] ⟨ 数値式 ⟩ については次項を参照して下さい。

[*3] なお，sind のようなプリミティブの ⟨ 数値演算子 ⟩ 以外に，vardef で定義された単項演算子などの「マクロ」も ⟨ 引数 ⟩ をとります。この ⟨ 引数 ⟩ に何位の式を要求するかは，マクロの定義の際に指定できます。これについては **第9講 6項** を参照して下さい。

[*4] ⟨ 二項演算子 ⟩ + の ⟨ 引数 ⟩ ともいえます

[*5] ⟨ 数値演算子 ⟩ の ⟨ 引数 ⟩ と同様，指定の位数の範囲内で最も長い式を ⟨ 引数 ⟩ として採用します。

ます。それは,

「/ の前後が ⟨ 数値トークン ⟩ のときのみ最高位の優先順位の計算になる」[*6]

というものです。これは 1/4x で $\frac{1}{4}x$ を表すための規則です。これは，x が **numeric** 以外の **pair** などの場合，決して分母にくることはなく，$\frac{1}{4x}$ より $\frac{1}{4}x$ の方が圧倒的に使用頻度が高いことを考えれば妥当な仕様といえます。注意すべきは，あくまで / の前後が ⟨ 数値トークン ⟩ の場合に限るということで，a/4x では $\frac{a}{4x}$ の意味になってしまいます。

2 numeric と ⟨ 数値式 ⟩

⟨ 数値式 ⟩ は結果が **numeric** タイプになる式，すなわち，**numeric** 値をとる式のことです。これの正式な定義は METAFONT ブックの第 25 章にあります。— **第1講** で，⟨ 変数名 ⟩ の定義に出てきた「あの」形式で定義されています。— しかし，プログラム的に厳密な定義が人間にとって理解しやすい形式かというと，必ずしもそうとはいえない (というより，プログラムに精通していない人にはかえってわかりにくい) と思いますので，詳細は省略します[*7]。

ここではおおよその感じのみを説明することにしますと，⟨ 数値トークン ⟩ や ( ) で囲まれた部分や，優先順位最高の演算[*8]で結ばれたものが ⟨ 数値 1 位式 ⟩，優先順位 2 番目の演算 (積や商) で結ばれたものが ⟨ 数値 2 位式 ⟩，優先順位 3 番目の演算 (和や差) で結ばれたものが ⟨ 数値 3 位式 ⟩，優先順位 4 番目の演算で結ばれたものが ⟨ 数値式 ⟩ となります。ただし，デフォルトでは「結果が **numeric** 値である優先順位 4 番目の演算」はありませんので，⟨ 数値式 ⟩ は実質 ⟨ 数値 3 位式 ⟩ と同義になります[*9]。

numeric を値にとる演算は，通常の数学で使う演算が中心ですので，あまり細かい解説は要らないと思います。したがって，ここではいくつかの演算をトピックス的に取り上げるだけにします。

- 初等的な演算は一通りそろっています。

  加減乗除は + - * / で，$\sqrt{x}$ は sqrt x で，$|x|$ は abs x で，x^p は x**p で表せます。

  また，三角関数は角度法の数値を ⟨ 引数 ⟩ にする形式で，$\sin x$, $\cos x$ に対応する sind x , cosd x が用意されています。

  指数・対数関数は，META 言語で扱える数値の範囲内でなるべく精度がよくなるように，$\mathrm{mlog}\, x = 256 \log_e x$, $\mathrm{mexp}\, x = e^{x/256}$ (e は自然対数の底) を使っています。(それぞれ mlog x , mexp x と記述します。)
- 小数値から整数値への切り上げ，切り捨て，四捨五入には，それぞれ ceiling , floor , round が用意されています。数学の記号で書けば，x の切り上げは $\lceil x \rceil$, 切り捨ては $\lfloor x \rfloor$ です[*10]。四捨五入は $\lfloor x + 0.5 \rfloor$ で定義されています。

[*6] 文法的には ⟨ 数値トークン ⟩/⟨ 数値トークン ⟩ が特別扱いされ，このまとまりで ⟨ 数値演算子 ⟩ として機能するからです。

[*7] 興味のある人は，METAFONT ブックの第 25 章をご覧下さい。

[*8] ⟨ 数値トークン ⟩ どうしの商や ⟨ 数値演算子 ⟩ とその ⟨ 引数 ⟩ — sind などとその ⟨ 引数 ⟩ や暗黙の積 — を指します。

[*9] 不等式や等式の結果は **boolaen** です。

[*10] $\lceil -3.1 \rceil = -3$, $\lfloor -3.1 \rfloor = -4$ です。「切り上げ」とは数直線上，その数値かそれより右にある最小の整数値,「切り捨て」とは数直線上，その数値かそれより左にある最大の整数値のことを指します。

割り算 x/y の商を整数値で求めるには，`floor(x/y)` とします。余りは数式で

$$x - y\lfloor x/y \rfloor$$

と表せますが，これは `x mod y` でも求められます (優先順位は `*` などと同じです)。

- 複数の数値の中で最も大きい数[*11]，小さい数[*12]を求める `max` , `min` という関数もあります。たとえば，`a` , `b` , `c` のうちで最大の数値を求めるには

```
max(a,b,c)
```

とします。

- ちょっと変わったところで言えば，ピタゴラス和，ピタゴラス差というものがプリミティブで用意されています。これはそれぞれ数式で $\sqrt{x^2+y^2}$, $\sqrt{x^2-y^2}$ と表現されるもので，2 点間の距離を求める際などに利用されます。もちろん `sqrt(x*x+y*y)` でも $\sqrt{x^2+y^2}$ は求まるのですが，計算の途中の数値が結構大きくなり，META 言語で扱える数値を超えてしまうおそれがあり，誤差もばかにならないため，プリミティブで用意されているわけです。$\sqrt{x^2+y^2}$ は `x++y` で，$\sqrt{x^2-y^2}$ は `x+-+y` で求められます。ちなみに優先順位は `+` や `-` と同じです。
- このほかにも，**pair** 値を ⟨ 引数 ⟩ にとって **numeric** 値を返す関数や演算もあるのですが，それらは **第3講** で **pair** を説明するときにあわせて説明することにします。

Exercise 3.2.1　`2x++2y` や `3++␣-7` は，とくに ( ) をつけなくても，それぞれ

$$\sqrt{(2x)^2+(2y)^2} \text{ や } \sqrt{3^2+(-7)^2}$$

の意味になりますが，それはなぜでしょう。また，`3++␣-7` において，空白 ␣ を取り去るとどうなるでしょうか。

Exercise 3.2.2　`1/256/256` は $\dfrac{1}{\frac{256}{256}}(=1)$ か $\dfrac{\frac{1}{256}}{256}\left(=\dfrac{1}{256\times 256}\right)$ いずれの意味になるでしょうか。

Exercise 3.2.3　`x=2` のとき，`x/2/2` は `(x/2)/2` , `x/(2/2)` のいずれの意味になるでしょう。またそれはどうしてでしょう。

Exercise 3.2.4　`1*256/256` では `*` と `/` のいずれが先に行われても，結果は 1 となりますが，実際のところ，どちらが先に行われているのでしょう。また，この計算順序の違いが，何か問題を生むことがあるでしょうか。

Exercise 3.2.5　`1/2[2,4]` , `(1/2)[2,4]` , `-1/2[2,4]` , `(-1/2)[2,4]` の値はいくらになるでしょう。またそれはどうしてでしょう。なお，`t[a,b]` は **第1部 第9講 4項** でも出てきましたように，$(1-t)a+tb$ を表します。この演算は `t` の部分に ⟨ 数値アトム ⟩ — **numeric** 値変数 , ⟨ 数値トークン ⟩ , ⟨ 数値トークン ⟩ どうしの商，( ) で囲われた ⟨ 数値式 ⟩ のいずれか — を要求し，`t[a,b]` 全体で ⟨ 数値 1 位式 ⟩ を構成します。

[*11] 等しい数がある場合のことを考えて，正確には「小さくない数」と表現されます。

[*12] こちらも正確には「大きくない数」と表現されます。

第 3 講

pair と color

1 pair と〈ペア式〉の意味

pair は (〈数値式〉,〈数値式〉) という形式で表され，平面上の点の座標や平面ベクトルに使われます。META 言語では，この **pair** 値を (〈数値式〉,〈数値式〉) という形式だけではなく，ひとかたまりの値として演算を行ったり変数に代入したりできるようになっており，これが点の位置を決める上で非常に有効に機能します[*1]。

〈ペア式〉は結果が **pair** タイプになる式，すなわち，**pair** 値をとる式のことです。これの正確な定義は，例によって METAFONT ブック第 25 章を参照して下さい。ここではおおよその感じだけ説明することにします。

まず，(〈数値式〉,〈数値式〉) や 計算の優先部位を表す ( ) で囲まれた部分や，優先順位最高の演算で結ばれたものが〈ペア 1 位式〉になります。「優先順位最高の演算」とは，たとえば，a が **pair** 値変数のとき，これを 2 倍するという意味の 2a という形の「暗黙の積」がそうです。この 2 は〈スカラー乗法演算子〉と見なされ，2*a (優先順位 2 番目の積) より優先順位が高くなります[*2]。

次に，優先順位 2 番目の演算 (積や商) で結ばれたものが〈ペア 2 位式〉，優先順位 3 番目の演算 (和や差) で結ばれたものが〈ペア 3 位式〉，優先順位 4 番目の演算で結ばれたものが〈ペア式〉となります。ただし，デフォルトでは「結果が **pair** 値である優先順位 4 番目の演算」はありませんので，〈ペア式〉は実質〈ペア 3 位式〉と同義になります。

優先順位に関しては，演算の種類とともに示した方がわかりやすいと思いますので，詳しくは **第3講 3項** を参照して下さい。

2 color と〈カラー式〉の意味

color は (〈数値式〉,〈数値式〉,〈数値式〉) という形式で表され，本来「色」を RGB 値で表すためのものですが，空間内の点の座標や空間ベクトルにも流用できます[*3]。なお，「色」を表すときには，各成分が 0 以上 1 以下の **numeric** 値に限定されます。すなわち，0 以下の値は 0 に，1 以上の値は 1 に解釈されます[*4]。

〈カラー式〉は結果が **color** タイプになる式，すなわち，**color** 値をとる式のことです。その文法的構造は〈ペア式〉に準じる思えば間違いないです。こちらについても，優先順位に関しては，演算の種類とともに示した方がわかりやすいと思いますので，**第3講 3項** を参照して下さい。

[*1] この実例については，**第3講 5項** で説明します。

[*2] この辺りは，**第2講 2項** の〈数値式〉と同様の考え方に則っています。

[*3] ただし，空間ベクトルとしての演算はほとんど定義されていないため，空間ベクトルに流用するなら各種演算を定義しなければなりません。しかし，それはさほど難しいことではないでしょう。

[*4] もちろん,「色」以外の意味に使う場合は，このような同一視は行われません。

3 numeric , pair , color 値から pair , color 値を導く演算

⟨ 数値式 ⟩ に比べて ⟨ ペア式 ⟩ 関連の演算は若干複雑なため，優先順位別に整理していきたいと思います。なお，**path** 値が絡むものについては **path** の解説のときにまとめて行います。

3.1 優先順位最高 — ⟨ ペア 1 位式 ⟩ , ⟨ カラー 1 位式 ⟩ を生成するもの

先に挙げた，(⟨ 数値式 ⟩,⟨ 数値式 ⟩) は ⟨ ペア 1 位式 ⟩ です。また，⟨ ペア式 ⟩ を ( ) でくくったものや，分点を表す t[⟨ ペア式 ⟩,⟨ ペア式 ⟩] (t は ⟨ 数値アトム ⟩) も ⟨ ペア 1 位式 ⟩ と見なされます。

それから，a を **pair** 値とするとき，2a のような ⟨ ペア 1 位式 ⟩ のスカラー倍 (暗黙の積) も ⟨ ペア 1 位式 ⟩ になります[*5]。

⟨ ペア 1 位式 ⟩ を生成するマクロとしては dir⟨ 数値 1 位式 ⟩ が有名です。他にも vardef で定義される **pair** 値のマクロも ⟨ ペア 1 位式 ⟩ を生成します。意外と見落としがちですが，z⟨ サフィックス ⟩ も ⟨ ペア 1 位式 ⟩ を生成するマクロです。

⟨ カラー 1 位式 ⟩ についても同様で，(⟨ 数値式 ⟩,⟨ 数値式 ⟩,⟨ 数値式 ⟩) や，⟨ カラー式 ⟩ を ( ) でくくったものや，分点を表す t[⟨ カラー式 ⟩,⟨ カラー式 ⟩]，⟨ カラー 1 位式 ⟩ のスカラー倍 (暗黙の積) などが ⟨ カラー 1 位式 ⟩ になります。

⟨ カラー 1 位式 ⟩ を生成するマクロはデフォルトではとくにありません。MEPOTEX では「色」を RGB 形式，CYMK 形式，HSB 形式で扱うためのマクロ

```
rgb (⟨ 数値式 ⟩,⟨ 数値式 ⟩,⟨ 数値式 ⟩)
cymk(⟨ 数値式 ⟩,⟨ 数値式 ⟩,⟨ 数値式 ⟩,⟨ 数値式 ⟩)
hsb (⟨ 数値式 ⟩,⟨ 数値式 ⟩,⟨ 数値式 ⟩)
```

が定義されています。

マクロではありませんが，定数としては以下のようなものも定義されています。

- METAFONT , METAPOST で定義されている **pair** 値定数 (⟨ ペア 1 位式 ⟩)
  right , left , up , down , origin [*6]
- METAPOST で定義されている **color** 値定数 (⟨ カラー 1 位式 ⟩)
  black , white , red , green , blue , background [*7]
- MEPOTEX で定義されている **color** 値定数 (⟨ カラー 1 位式 ⟩)
  LaTeX 2_ε の color.sty の usenames オプション指定時に使える 68 色の色名

3.2 優先順位 2 位 — ⟨ ペア 2 位式 ⟩ , ⟨ カラー 2 位式 ⟩ を生成するもの

METAFONT では，**pair** 値に **numeric** 値をかけたり割ったりでき，これが ⟨ ペア 2 位式 ⟩ となります。より具体的には，以下の形式になります。

⟨ ペア 2 位式 ⟩ * ⟨ 数値 1 位式 ⟩
⟨ ペア 2 位式 ⟩ / ⟨ 数値 1 位式 ⟩

[*5] 先に説明したとおり 2a と 2*a は，演算結果が一緒であっても，演算の優先順位は異なるので要注意です。
[*6] 念のために復習しておきますと，right=(1,0) , left=(-1,0) , up=(0,1) , down=(0,-1) , origin=(0,0) です。
[*7] black=(0,0,0) , white=(1,1,1) , red=(1,0,0) , green=(0,1,0) , blue=(0,0,1) です。background は通常 white と等置されています。

〈数値 2 位式〉 * 〈ペア 1 位式〉

積・商の記号 (* , /) の前が 〈2 位式〉, 後ろが 〈1 位式〉 となっているのは，前講で説明したとおり，かけ算，割り算が左から順に行われるようにするためです。

METAPOST では，上記に加え，**color** 値に **numeric** をかけたり割ったりでき，これが 〈カラー 2 位式〉 となります。

MEPOTEX にはさらに **color** を 3 次元ベクトルと見たときのベクトル積 (外積)

〈カラー 2 位式〉 gaiseki 〈カラー 1 位式〉

も定義されています。

〈カラー 2 位式〉 についてはこれだけですが，〈ペア 2 位式〉 にはさらに

〈ペア 2 位式〉〈変形演算子〉

という形式もあります。〈変形演算子〉 とは scaled や shifted のような，いわゆる **変形** を行う演算子 (にその 〈引数〉 を付けたもの) です[*8]。なお，scaled や shifted などは 〈引数〉 として 〈1 位式〉 を要求するので注意が必要です。たとえば，

scaled は 〈ペア 2 位式〉 scaled 〈数値 1 位式〉

shifted は 〈ペア 2 位式〉 shifted 〈ペア 1 位式〉

の形で使います[*9]。

〈ペア 2 位式〉, 〈カラー 2 位式〉 を生成する 〈二項演算子〉 は，積・商と同格になりますので，primarydef を用いて定義することになります。

3.3 優先順位 3 位 — 〈ペア 3 位式〉, 〈カラー 3 位式〉 を生成するもの

METAFONT では，**pair** 値どうしを足したり引いたりできます。これが 〈ペア 3 位式〉 となります。

METAPOST では，これに加えて **color** 値どうしを足したり引いたりできます。これが 〈カラー 3 位式〉 となります。

〈ペア 3 位式〉, 〈カラー 3 位式〉 を生成する 〈二項演算子〉 は，和・差と同格になりますので，secondarydef を用いて定義することになります。

3.4 補足

一般的に演算やコマンドの 〈引数〉 に何位の式が要求されるかについては，おおよそ次のように考えればいいでしょう。

- 単項演算子タイプのプリミティブの 〈引数〉 は通常 〈1 位式〉 です[*10]。

(例) length〈ペア 1 位式〉

単項演算子タイプのマクロの 〈引数〉 は定義次第ですが，それでも 〈1 位式〉 を要求するものが多く，次いで 〈式〉 (4 位) を要求するものが多いようです。

[*8] 〈変形演算子〉 については **第7講** で詳しく説明します。

[*9] これは別の見方をすれば，scaled や shifted などが，優先順位 2 位の 〈二項演算子〉 として前に 〈2 位式〉，後に 〈1 位式〉 を要求しているとも言えます。

[*10] 値を返す単項演算子は，〈引数〉 に 〈1 位式〉 を要求します。値を返さないもの — そういうものを「単項演算子」というかどうかは微妙ですが — では 〈式〉 (4 位) を要求するものもあります。たとえば，**第8講** で取り上げる errmessage などがそうです。

○ ⟨ コマンド ⟩ 〜 of 〜 の形の演算子の ⟨ 引数 ⟩ は，of の前が ⟨ 式 ⟩ (4 位) , 後が ⟨1 位式 ⟩ になります。
(例) point⟨ 数値式 ⟩of⟨ パス 1 位式 ⟩[*11]
○ 二項演算子は定義次第ですが，優先順位を n 位とすると，以下の形で使われます。
⟨n 位式 ⟩ ⟨ 二項演算子 ⟩ ⟨$n-1$ 位式 ⟩ ($n=2$, 3 , 4)
結果は ⟨n 位式 ⟩ です。マクロとして定義する場合，$n=2$ のものは primarydef で，$n=3$ のものは secondarydef で，$n=4$ のものは tertiarydef で定義します[*12]。

Exercise 3.3.1　次の 4 つのうち，エラーが出るものはどれでしょう。また，どのようなエラーが出るでしょう。

```
z1 = right shifted 2*up;   show z1;
z2 = right shifted 2dir90; show z2;
z3 = right shifted (0,2);  show z3;
z4 = right shifted 2(0,1)  show z4;
```

Exercise 3.3.2　dir の定義は以下のようになっています。

```
vardef dir primary d = right rotated d enddef;
```

dir が ⟨ 引数 ⟩ に ⟨(数値)1 位式 ⟩ を要求するのは，⟨ 区切りなし仮引数 ⟩[*13]の「位数」を指定する primary が付加されているためです。この primary (⟨1 位式 ⟩ を要求する) を，secondary (⟨2 位式 ⟩ を要求する) や tertiary (⟨3 位式 ⟩ を要求する) に変えた場合，以下の式の結果 — エラーが出るかどうかと結果の **pair** 値の成分表示 — はどうなるでしょうか。

```
z1 = dir 90*2;
z2 = dir 90+90;
```

4 pair , color 値から numeric 値を導く演算

pair , **color** 値を ⟨ 引数 ⟩ にとり，**numeric** 値をかえすマクロまたはプリミティブの中で，有名なものを挙げます。

まず，**pair** 値の平面ベクトルとしての x 成分，y 成分は，それぞれ次のようにして得られます。

xpart⟨ ペア 1 位式 ⟩ , ypart⟨ ペア 1 位式 ⟩

また，**color** 値の空間ベクトルとしての x 成分，y 成分，z 成分は，それぞれ次のようにして得られます。

redpart⟨ カラー 1 位式 ⟩ , greenpart⟨ カラー 1 位式 ⟩ , bluepart⟨ カラー 1 位式 ⟩

なぜ，x , y , z 成分が redpart , greenpart , bluepart なのかといいますと，もともと **color** 値は色を RGB 値で表すためのもので，x , y , z 成分が順に赤，緑，青の値に対応するためです。

[*11] point 〜 of 〜 については **第4講 7項** を参照して下さい。
[*12] primary , secondary , tertiary のもともとの意味は順に 1 位，2 位，3 位ですが，これは演算子の後ろに要求する ⟨ 式 ⟩ の位数を指していると言えます。
[*13] 詳しくは **第9講** を参照して下さい。

次にベクトルの長さですが，**pair** 値は

`length`⟨ ペア 1 位式 ⟩ または `abs`⟨ ペア 1 位式 ⟩

で長さを得ることができます。ところが，デフォルトでは **color** 値の長さを得るコマンドがありません[*14]。

ベクトルの方向角は，**pair** 値に対しては

`angle`⟨ ペア 1 位式 ⟩

という形式で得ることができます。こちらもデフォルトでは **color** 値に対応するコマンドがありません[*15]。

以上はいずれも ⟨ 数値 1 位式 ⟩ を生成するコマンドだったのですが，⟨ 数値 2 位式 ⟩ を生成するマクロとしてはスカラー積 (内積) があります。**pair** 値に対しては

⟨ ペア 2 位式 ⟩ `dotprod` ⟨ ペア 1 位式 ⟩

がスカラー積 (内積) を表します。**color** 値のスカラー積は METAPOST には定義されていませんが，MEPOTEX には `naiseki` という名前で定義されています。

Exercise 3.3.3 空間内部の点は，原点からの距離 R , 緯度 θ , 経度 φ を用いて

($R\cos\theta\cos\varphi$, $R\cos\theta\sin\varphi$, $R\sin\theta$)

と表せます。これを参考にして，**color** 値 — 空間内の点の座標と見る — の緯度と経度を返すマクロを，それぞれ作ってみましょう。

5 平面上の点の表し方

META 言語では **pair** 値を直接扱えるため，様々な方法で点の位置を与えることができます。ここでは **pair** 値の変数 (正確にはマクロ) `z1` に，色々な方法で点の「位置」を設定してみましょう。以下で `a` , `b` , `r` , `t` は **numeric** , `u` , `v` は **pair** であるとします。

(1) 直交座標 — 点の位置を表すのに x 座標と y 座標を用いる方法です。(**図 3.3.1**)

(例) 直交座標 (a , b) の点を設定するには

```
z1 = (a,b);
```

とします。

(2) 極座標 — 点の位置を表すのに，原点からの距離 r と基準の方向からの角度 t を用いる方法です。(**図 3.3.2**)

(例) 極座標 (r , t) の点を設定するには

```
z1 = r*dir t;
```

とします。

(3) 位置ベクトル — ある基準点からの移動方向と移動量を，ベクトルで表現したものです。

(例) 直交座標 (a , b) の点から，方向角 t の向きに距離 r だけ進んだ点を設定するには

[*14] もっとも作ろうと思えば容易に作れます。たとえばこうです。

```
vardef nagasa primary c =
   redpart c ++ greenpart c ++ bluepart c
enddef;
```

これで `nagasa`⟨ カラー 1 位式 ⟩ という形式で長さを取得できます。これは MEPOTEX ではすでに実装されています。

[*15] MEPOTEX ではすでに実装されています。定義例については，直後の演習問題を参照して下さい。

```
z1 = (a,b) + r*dir t;
```

とします。

(4) 斜交座標 — 直交座標と異なり，直交しない座標軸をもつ座標系です。(**図 3.3.3**)

直交座標 $(a, b) = a*(1, 0) + b*(0, 1)$

において，基本ベクトル $(1, 0), (0, 1)$ が他のベクトルに置き換わったものが斜交座標と解釈できます。空間図形を平面上に射影するときなどに活躍します。

(例) **pair** 値 `u` , `v` を基本ベクトルとする斜交座標で (a, b) と表される点を設定するには

```
z1 = a*u + b*v;
```

とします。

(5) 複素数平面 — **pair** 値 `(a,b)` は複素数 $a + bi$ と解釈することもできます。このとき，和・差・実数倍はベクトルの和・差・実数倍と同じ意味になります。複素数どうしの積 — たとえば **pair** 値 `u` , `v` の積を設定するには

```
z1 = u zscaled v;
```

とします。

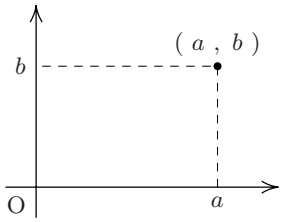


図 3.3.1　直交座標

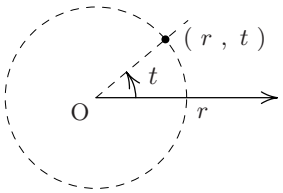


図 3.3.2　極座標

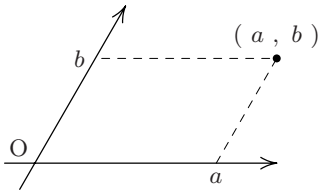


図 3.3.3　斜交座標

Exercise 3.3.4　点 z_1 は直交座標で $(3, 2)$ の点，z_2 は z_1 から 45° の方向に 3 進んだ点，原点と点 z_1 , z_2 , z_3 はこの順で平行四辺形を作るとします。z_1 , z_2 , z_3 に対応する **pair** 値 `z1` , `z2` , `z3` を設定して下さい。ただし，z_2 , z_3 は座標を直接与えないものとします。

第 4 講

path

1 3 次 Bézier 曲線

METAFONT 言語の **path** は，METAFONT や METAPOST が内部的に処理し，最終的に 3 次の Bézier 曲線とよばれるものに解釈されます。これは，2 点 A , B を結ぶ曲線を描く際に，コントロールポイントとよばれる 2 点 C , D [*1]を与えて，以下の式で表される点 P の軌跡を採用するというものです。

$$\overrightarrow{\mathrm{OP}} = (1-t)^3\overrightarrow{\mathrm{OA}} + 3t(1-t)^2\overrightarrow{\mathrm{OC}} + 3t^2(1-t)\overrightarrow{\mathrm{OD}} + t^3\overrightarrow{\mathrm{OB}} \quad (\ 0 \le t \le 1\ ) \cdots\cdots (*)$$

「内部的に」と書きましたが，もちろん直接コントロールポイントを指定することもできます。A , B , C , D に対応する **pair** 値を順に `a` , `b` , `c` , `d` とすると，

```
a..controls c and d ..b
```

が，(∗) で表される曲線と同じものになります。

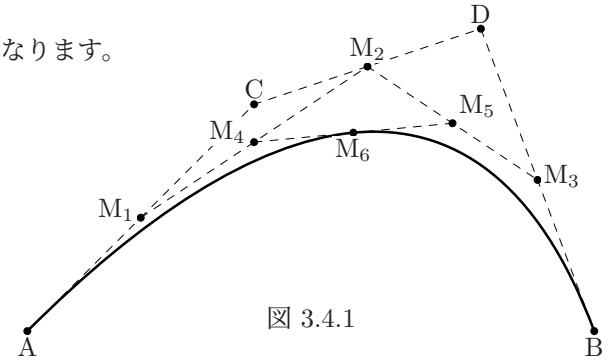


図 3.4.1

さて，ではこの 3 次 Bézier 曲線の性質を見ていきましょう。まず，右図を見て下さい。太実線が先程 (∗) で与えた曲線です。図を見ればわかるように，この曲線は直線 AC および DB に接しています (性質 1)。また，線分 AC , CD , DB の中点を順に M_1 , M_2 , M_3 とし，さらに線分 $\mathrm{M}_1\mathrm{M}_2$, $\mathrm{M}_2\mathrm{M}_3$ の中点を順に M_4 , M_5 とすると，線分 $\mathrm{M}_4\mathrm{M}_5$ の中点 M_6 が曲線上に乗っています (性質 2)。これらは，数式上でも以下のようにして証明できます。まず (∗) を t で微分します。

$$\frac{d}{dt}\overrightarrow{\mathrm{OP}} = -3(1-t)^2\overrightarrow{\mathrm{OA}} + 3\{(1-t)^2 - 2t(1-t)\}\overrightarrow{\mathrm{OC}} + 3\{2t(1-t) - t^2\}\overrightarrow{\mathrm{OD}} + 3t^2\overrightarrow{\mathrm{OB}}$$

この式に $t = 0$ を代入すると，$3(\overrightarrow{\mathrm{OC}} - \overrightarrow{\mathrm{OA}}) = 3\overrightarrow{\mathrm{AC}}$ となり，始点 A での接線の方向が $\overrightarrow{\mathrm{AC}}$ の向きであることがわかります。また，$t = 1$ を代入すると，$3(\overrightarrow{\mathrm{OB}} - \overrightarrow{\mathrm{OD}}) = 3\overrightarrow{\mathrm{DB}}$ となり，終点 B での接線の方向が $\overrightarrow{\mathrm{DB}}$ の向きであることもわかります。また，

$$\overrightarrow{\mathrm{OM_1}} = \frac{1}{2}(\overrightarrow{\mathrm{OA}} + \overrightarrow{\mathrm{OC}})\ ,\quad \overrightarrow{\mathrm{OM_2}} = \frac{1}{2}(\overrightarrow{\mathrm{OC}} + \overrightarrow{\mathrm{OD}})\ ,\quad \overrightarrow{\mathrm{OM_3}} = \frac{1}{2}(\overrightarrow{\mathrm{OD}} + \overrightarrow{\mathrm{OB}})$$

$$\overrightarrow{\mathrm{OM_4}} = \frac{1}{2}(\overrightarrow{\mathrm{OM_1}} + \overrightarrow{\mathrm{OM_2}}) = \frac{1}{4}(\overrightarrow{\mathrm{OA}} + 2\overrightarrow{\mathrm{OC}} + \overrightarrow{\mathrm{OD}})$$

$$\overrightarrow{\mathrm{OM_5}} = \frac{1}{2}(\overrightarrow{\mathrm{OM_2}} + \overrightarrow{\mathrm{OM_3}}) = \frac{1}{4}(\overrightarrow{\mathrm{OC}} + 2\overrightarrow{\mathrm{OD}} + \overrightarrow{\mathrm{OB}})$$

[*1]これに対して A , B をこの本ではキーポイントとよぶことにします。

$$\overrightarrow{\mathrm{OM}_6}=\frac{1}{2}(\overrightarrow{\mathrm{OM}_4}+\overrightarrow{\mathrm{OM}_5})=\frac{1}{8}(\overrightarrow{\mathrm{OA}}+3\overrightarrow{\mathrm{OC}}+3\overrightarrow{\mathrm{OD}}+\overrightarrow{\mathrm{OB}})$$

であり，これは $(*)$ に $t=\frac{1}{2}$ を代入したものに相当することもわかります。

さらに上において，$\mathrm{A}=\mathrm{A}'$, $\mathrm{M}_6=\mathrm{B}'$, $\mathrm{M}_1=\mathrm{C}'$, $\mathrm{M}_4=\mathrm{D}'$ とおいて，A' , B' , C' , D' について 3 次の Bézier 曲線をつくると，これは $(*)$ で表される曲線の左半分，すなわち $0\le t\le\frac{1}{2}$ に対応する部分に一致します (性質 3)。これは次のように証明されます。

$$\begin{aligned}\overrightarrow{\mathrm{OP}}&=(1-s)^3\overrightarrow{\mathrm{OA}'}+3s(1-s)^2\overrightarrow{\mathrm{OC}'}+3s^2(1-s)\overrightarrow{\mathrm{OD}'}+s^3\overrightarrow{\mathrm{OB}'}\\&=(1-s)^3\overrightarrow{\mathrm{OA}}+\frac{3}{2}s(1-s)^2(\overrightarrow{\mathrm{OA}}+\overrightarrow{\mathrm{OC}})\\&\quad+\frac{3}{4}s^2(1-s)(\overrightarrow{\mathrm{OA}}+2\overrightarrow{\mathrm{OC}}+\overrightarrow{\mathrm{OD}})+\frac{1}{8}s^3(\overrightarrow{\mathrm{OA}}+3\overrightarrow{\mathrm{OC}}+3\overrightarrow{\mathrm{OD}}+\overrightarrow{\mathrm{OB}})\\&=\Big(1-\frac{s}{2}\Big)^3\overrightarrow{\mathrm{OA}}+3\Big(\frac{s}{2}\Big)\Big(1-\frac{s}{2}\Big)^2\overrightarrow{\mathrm{OC}}+3\Big(\frac{s}{2}\Big)^2\Big(1-\frac{s}{2}\Big)\overrightarrow{\mathrm{OD}}+\Big(\frac{s}{2}\Big)^3\overrightarrow{\mathrm{OB}}\end{aligned}$$

$t=\frac{s}{2}$ とおけば，$0\le s\le 1$ の部分が $(*)$ の $0\le t\le\frac{1}{2}$ の部分になることがわかると思います。

このことから，3 次 Bézier 曲線の作図的な構成ができます。すなわち，与えられた 4 点 A , B , C , D から上記のように中点 M_1 〜 M_6 を構成します。次に「左半分」について A , M_6 , M_1 , M_4 を A' , B' , C' , D' として，ふたたび中点を構成していきます。「右半分」についても同様です。以下この操作を繰り返し，各操作で得られた M_6 をつなげていけば，目的の曲線が得られます。

Exercise 3.4.1　M_6 , B , M_5 , M_3 を A'' , B'' , C'' , D'' として，3 次 Bézier 曲線を作ると，もとの Bézier 曲線の右半分 ($\frac{1}{2}\le t\le 1$) に一致することを示してみて下さい。

さて，次に 性質 2 , 3 の一般化を考えてみたいのですが，Bézier 曲線の式 $(*)$ は少々煩雑ですので，少し簡略化した記号を導入しておきます。以下では点の位置を表現するとき，複素数表示を使います[*2]。そして，META 言語で `t[z1,z2]` が `(1-t)*z1+t*z2` を表すのにならって，複素数の演算として，$t[z_1,z_2]$ を，さらにその発展として $t[z_1,z_2,z_3]$ および $t[z_1,z_2,z_3,z_4]$ を以下のように定義します。

$$\begin{aligned}&t[z_1,z_2]=(1-t)z_1+tz_2\\&t[z_1,z_2,z_3]=t[\,t[z_1,z_2],t[z_2,z_3]\,]\\&t[z_1,z_2,z_3,z_4]=t[\,t[z_1,z_2,z_3],t[z_2,z_3,z_4]\,]\end{aligned}$$

このようにすれば，Bézier 曲線上の点 $(1-t)^3z_1+3(1-t)^2tz_2+3(1-t)t^2z_3+t^3z_4$ が $t[z_1,z_2,z_3,z_4]$ で表せます。このことから 性質 2 を一般化した次のことが言えます。線分 AC , CD , DB を $t:(1-t)$ に内分する点を順に M_1 , M_2 , M_3 とし，さらに線分 $\mathrm{M}_1\mathrm{M}_2$, $\mathrm{M}_2\mathrm{M}_3$ を $t:(1-t)$ に内分する点を順に M_4 , M_5 とすると，線分 $\mathrm{M}_4\mathrm{M}_5$ を $t:(1-t)$ に内分する点 M_6 が曲線上に乗っています (性質 $2'$)。

Exercise 3.4.2　$t[z_1,z_2,z_3,z_4]=(1-t)^3z_1+3(1-t)^2tz_2+3(1-t)t^2z_3+t^3z_4$ を証明してみましょう。

[*2] といってもたいしたことではなく，要は **pair** 値 **z1** に対応する「点」を「複素数」と見て z_1 などと表すだけの話です。

前述の記号を使って，z_1' , z_2' , z_3' , z_4' および z_1'' , z_2'' , z_3'' , z_4'' を以下のように定めます。

$$z_1' = z_1 \ , \ z_2' = t_0[z_1, z_2] \ , \ z_3' = t_0[z_1, z_2, z_3] \ , \ z_4' = t_0[z_1, z_2, z_3, z_4]$$
$$z_1'' = t_0[z_1, z_2, z_3, z_4] \ , \ z_2'' = t_0[z_2, z_3, z_4] \ , \ z_3'' = t_0[z_3, z_4] \ , \ z_4'' = z_4$$

このとき，Bézier 曲線 $t[z_1, z_2, z_3, z_4]$ ($0 \le t \le 1$) の $0 \le t \le t_0$ の部分は $s[z_1', z_2', z_3', z_4']$ ($0 \le s \le 1$) に，$t_0 \le t \le 1$ の部分は $s[z_1'', z_2'', z_3'', z_4'']$ ($0 \le s \le 1$) に，それぞれ一致します (性質 3′)[*3]。つまり，Bézier 曲線は，その一部分を取り出してもやはり Bézier 曲線になるわけです。

Exercise 3.4.3 Bézier 曲線 $t[z_1, z_2, z_3, z_4]$ ($0 \le t \le 1$) の $0 \le t \le t_0$ の部分が $s[z_1', z_2', z_3', z_4']$ ($0 \le s \le 1$) と一致することを証明してみましょう。(z_1' , z_2' , z_3' , z_4' の意味は上記の通りとします。)

2 パス結合

path は **pair** どうしを ⟨ パス結合 ⟩ でつないで構成されます。もちろん **path** どうし，あるいは，**path** と **pair** を ⟨ パス結合 ⟩ でつないでより大きな **path** にすることもできます[*4]。⟨ パス結合 ⟩ については，これまでに何度となく出てきているのですが，ここで一覧しておきましょう。

まず，⟨ 基本パス結合 ⟩ ですがこれは以下の 4 種類があります。

(1) 単純な接続 `&`

`&` の前後の **path** を単純につなぎます。`&` の前の **path** の末端の点と，`&` の後の **path** の開始点が一致していないといけません。

(2) 自由曲線の `..`

最もよく使われるもので，点どうしを「なめらか」につなぎます。`..` 自体はプリミティブなのですが，意味は `..tension 1 and 1..` と同じになります。

(3) 張り具合を指定した `..tension` α `and` β `..`

上と同じく点どうしを曲線で結ぶのですが，その曲線の張り具合を指定できます。α と β は，ともに $\frac{3}{4}$ 以上の値をもつ ⟨ 数値 1 位式 ⟩ で，必要ならその前に `atleast` をつけられます。`atleast` をつけると，曲線のこの結合の部分が，端点を結ぶ線分と端点における 2 つの接線とで作られる三角形からはみ出さないように，張り具合が補正されます[*5]。なお，`..tension` α `..` は `..tension` α `and` α `..` の省略形です。

(4) コントロールポイントを指定した `..controls` α `and` β `..`

Bézier 曲線のコントロールポイントを直接指定する形式です。α と β は，ともに ⟨ ペア 1 位式 ⟩ です。なお，`..controls` α `..` は `..controls` α `and` α `..` の省略形です。

これら ⟨ 基本パス結合 ⟩ および，その一端または両端に ⟨ 方向指示子 ⟩ をつけたものが ⟨ パス結合 ⟩ です。方向指定子は端点での方向を指定するもので

`{curl`⟨ 数値式 ⟩`}` または `{`⟨ ペア式 ⟩`}` または `{`⟨ 数値式 ⟩`,`⟨ 数値式 ⟩`}`

[*3] 性質 2,3 は 性質 2′,3′ の $t_0 = \frac{1}{2}$ に対する特殊ケースだったわけです。

[*4] 文法的に正確に言えば，**path** と **pair** , **pair** と **pair** の結合は，**pair** を 1 点からなる **path** と見なすことにより，**path** と **path** の結合と統一的に解釈します。

[*5] この項の最後の例を参照して下さい。

の形式で指定します[*6] 。なお，**path** と **path** , **path** と **pair** , **pair** と **pair** を ⟨ パス結合 ⟩ で結びつけて作ったものが ⟨ パス部分式 ⟩ で，⟨ パス部分式 ⟩，または，⟨ パス部分式 ⟩ に ⟨ 方向指示子 ⟩ をつけたもの，または，⟨ パス部分式 ⟩ に ⟨ パス結合 ⟩ と `cycle` をつけたものが ⟨ パス式 ⟩ です。とくに最後のものは ⟨ 巡回パス ⟩ を生じます。

⟨ パス結合 ⟩ には上記の他に，マクロとして定義されたものもあり，よく使うものでは以下のものがあります。

`--` は `{curl 1}..{curl 1}`

`---` は `.. tension infinity ..`

`...` は `.. tension atleast 1 ..`

最後に `atleast` による `tension` の補正効果を見てもらうために，1 組の例を挙げましょう。

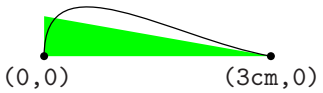


図 3.4.2

図 3.4.2 は `draw (0,0){dir90}..{dir-10}(3cm,0)`

図 3.4.3 は `draw (0,0){dir90}...{dir-10}(3cm,0)` です。`...` は `..tension atleast 1..` であることを思い出して下さい。

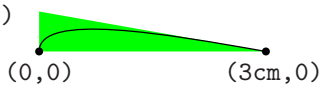


図 3.4.3

3 3 点からなる path の Bézier 曲線への変換

本項では **path** がどのように Bézier 曲線に変換されるかを説明したいと思います。もちろん，ここで扱うものは 2 点以上から構成される **path** です[*7]。ただし，2 点からなる **path** は単純すぎて，かえって本質がわかりにくくなるため，3 点からなる **path** を基本ケースとして説明したいと思います。

まず，**図 3.4.4** を見て下さい。以下において，点 Z_0 , C_0 , ····· に対応する **pair** 値を `z0` , `c0` , ····· , 複素数平面で考えたときの対応する複素数を z_0 , c_0 , ····· で表すことにします。**図 3.4.4** の太実線は

```
draw z0..z1..z2;
```

で描いています。これだけで曲線弧 Z_0Z_1 , Z_1Z_2 をそれぞれ Bézier 曲線で描くために必要なコントロールポイント C_0 , D_1 , C_1 , D_2 が自動的に算出され，Z_0 , Z_1 , Z_2 がなめらかにつながります。

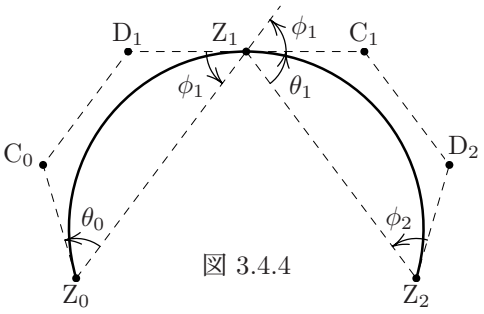


図 3.4.4

さて，コントロールポイントの計算方法ですが，既知のキーポイント Z_0 , Z_1 , Z_2 からコントロールポイント C_0 , D_1 , C_1 , D_2 を求めるには，$\overrightarrow{Z_0C_0}$, $\overrightarrow{D_1Z_1}$, $\overrightarrow{Z_1C_1}$, $\overrightarrow{D_2Z_2}$ の 4 つのベクトルを求めればよいことがわかると思います。META 言語ではこれらのベクトルの方向角をまず求め，その後長さを求めるという 2 段階の計算を行います。

まず方向角についてですが，たとえば $\overrightarrow{Z_0C_0}$ の方向角は，$\overrightarrow{Z_0Z_1}$ の方向角 (既知) に **図 3.4.4** の角度 θ_0 を加えたものになります。同様に，$\overrightarrow{D_1Z_1}$, $\overrightarrow{Z_1C_1}$, $\overrightarrow{D_2Z_2}$ の方向角は，順

[*6] `{1,9}` は `{(1,9)}` と同じ意味です。

[*7] 1 点からなる **path** は文字通り「点」ですので。

に $\overrightarrow{\mathrm{Z}_0\mathrm{Z}_1}$, $\overrightarrow{\mathrm{Z}_1\mathrm{Z}_2}$, $\overrightarrow{\mathrm{Z}_1\mathrm{Z}_2}$ の方向角 (既知) に $-\phi_1$, θ_1 , $-\phi_2$ を加えたものになります[*8]。したがって，方向角を求めることは θ_0 , ϕ_1 , θ_1 , ϕ_2 を求めることと等価です。なお，方向角を求めるのは，上記の例のように方向角が指定されなかったときの話で，指定されていればそちらが優先されます。たとえば，

```
z0{dir120}..{dir10}z1{dir-20}..{dir-60}z2
```

の場合は $\overrightarrow{\mathrm{Z}_0\mathrm{C}_0}$, $\overrightarrow{\mathrm{D}_1\mathrm{Z}_1}$, $\overrightarrow{\mathrm{Z}_1\mathrm{C}_1}$, $\overrightarrow{\mathrm{D}_2\mathrm{Z}_2}$ の方向角は順に $120°$, $10°$, $-20°$, $-60°$ になります。また，

```
z0{dir120}..{dir10}z1..{dir-60}z2
```

のように `z1` の片側のみ 〈 方向指定子 〉 があるときは，それが `z1` の両側に複製され

```
z0{dir120}..{dir10}z1{dir10}..{dir-60}z2
```

と解釈されます。

さて，方向角が指定されなかったときの方向角の求め方ですが，これは θ_0 , ϕ_1 , θ_1 , ϕ_2 の連立方程式を解くことになります。**path** の端点の 〈 方向指定子 〉 が省略されているときは `{curl 1}` が補われ，〈 基本パス結合 〉 の `..` は `..tension 1 and 1..` と等価であることも考慮して，

`z0{curl` γ_0`}..tension` α_0 `and` β_1 `..z1..tension` α_1 `and` β_2 `..{curl` γ_2`}z2`

の場合を説明します。以下の連立方程式は John Hobby 氏の考えたアルゴリズムに依るらしいのですが，その意味するところはちょっとわかりません (^_^; まぁ，こういった計算が内部で行われているという雰囲気が伝われば，ぐらいに思って書いていますので，そういうつもりで流し読みして下さい。なお，以下では見やすいように既知の値を細字で，未知の値を太字で表します。また，

$$l_1 = |z_1 - z_0| = (\overrightarrow{\mathrm{Z}_0\mathrm{Z}_1}\text{の長さ}),\quad l_2 = |z_2 - z_1| = (\overrightarrow{\mathrm{Z}_1\mathrm{Z}_2}\text{の長さ})$$

$$\psi_1 = \arg\frac{z_2 - z_1}{z_1 - z_0} = (\mathrm{Z}_1\text{での回転角})$$

を導入します[*9]。l_1 , l_2 , ψ_1 はすべて既知です。

$$\boldsymbol{\theta_1} + \boldsymbol{\phi_1} = -\psi_1$$

$$\frac{1}{l_1}{\beta_1}^2\Big(\frac{1}{\alpha_0}(\boldsymbol{\theta_0} + \boldsymbol{\phi_1}) - 3\boldsymbol{\phi_1}\Big) = \frac{1}{l_2}{\alpha_1}^2\Big(\frac{1}{\beta_2}(\boldsymbol{\theta_1} + \boldsymbol{\phi_2}) - 3\boldsymbol{\theta_1}\Big)$$

$${\alpha_0}^2\Big(\frac{1}{\beta_1}(\boldsymbol{\theta_0} + \boldsymbol{\phi_1}) - 3\boldsymbol{\theta_0}\Big) = \gamma_0{\beta_1}^2\Big(\frac{1}{\alpha_0}(\boldsymbol{\theta_0} + \boldsymbol{\phi_1}) - 3\boldsymbol{\phi_1}\Big)$$

$${\beta_2}^2\Big(\frac{1}{\alpha_1}(\boldsymbol{\theta_1} + \boldsymbol{\phi_2}) - 3\boldsymbol{\phi_2}\Big) = \gamma_2{\alpha_1}^2\Big(\frac{1}{\beta_2}(\boldsymbol{\theta_1} + \boldsymbol{\phi_2}) - 3\boldsymbol{\theta_1}\Big)$$

未知数 4 つに連立方程式 4 つなので，(方程式が退化しない限り) 方向角がすべて求まることがわかると思います。なお，式は複雑ですが，未知数についての次数は 1 次ですので，コンピュータで機械的に解けます。

次は，$\overrightarrow{\mathrm{Z}_0\mathrm{C}_0}$, $\overrightarrow{\mathrm{D}_1\mathrm{Z}_1}$, $\overrightarrow{\mathrm{Z}_1\mathrm{C}_1}$, $\overrightarrow{\mathrm{D}_2\mathrm{Z}_2}$ の長さですが，$\overrightarrow{\mathrm{Z}_1\mathrm{C}_1}$, $\overrightarrow{\mathrm{D}_2\mathrm{Z}_2}$ の長さの求め方は $\overrightarrow{\mathrm{Z}_0\mathrm{C}_0}$,

---

[*8] 次の表現は等価です。
「$\overrightarrow{\mathrm{D}_1\mathrm{Z}_1}$ の方向角は $\overrightarrow{\mathrm{Z}_0\mathrm{Z}_1}$ の方向角に $-\phi_1$ を加えたもの。」
「$\overrightarrow{\mathrm{Z}_1\mathrm{D}_1}$ の方向角は $\overrightarrow{\mathrm{Z}_1\mathrm{Z}_0}$ の方向角に $-\phi_1$ を加えたもの。」
後者の方が図との対応が取りやすいと思いますが，〈 方向指定子 〉 の表現が前者に則っていますので本文でも前者で表現しました。

[*9] (Z_1 での回転角) ψ_1 は $\overrightarrow{\mathrm{Z}_0\mathrm{Z}_1}$ を $\overrightarrow{\mathrm{Z}_1\mathrm{Z}_2}$ の向きに向けるための回転角です。

$\overrightarrow{D_1Z_1}$ の長さの求め方と同様ですので，こちらのみ説明します。$\overrightarrow{Z_0C_0}$, $\overrightarrow{D_1Z_1}$ の長さは $\overrightarrow{Z_0Z_1}$ の長さにそれぞれ $\dfrac{f(\theta_0,\phi_1)}{\alpha_0}$, $\dfrac{f(\phi_1,\theta_0)}{\beta_1}$ をかけて得られます。ここに $f(\theta,\phi)$ はすでに求めた方向角のみで決定される値で，以下の式で与えられます[*10]。

$$f(\theta,\phi)=\frac{2+\sqrt{2}(\sin\theta-\frac{1}{16}\sin\phi)(\sin\phi-\frac{1}{16}\sin\theta)(\cos\theta-\cos\phi)}{3\{1+\frac{1}{2}(\sqrt{5}-1)\cos\theta+\frac{1}{2}(3-\sqrt{5})\cos\phi\}}$$

非常に複雑な式ですが，とくに $\theta=\phi$ のときは，あらかた項が消えてしまい，

$$f(\theta,\theta)=\frac{2}{3(1+\cos\theta)}$$

となります。

4 一般の path の Bézier 曲線への変換

一般の **path** についても 3 点の場合と同様に，曲線上のキーポイントからコントロールポイントまでのベクトルを，方向角，長さの順に求めます。長さの方は 3 点の場合と全く同様です。方向角も 3 点の場合と同様に連立方程式で求めます。この場合，〈パス結合〉が `&` のとき，あるいは〈方向指定子〉がついているときはそこで「連立」がとぎれますので，`z0{`d_0`}..tension` α_0 `and` β_1 `..z1..`(続く)`..z[n-1]..tension` α_{n-1} `and` β_n `..{`d_n`}z[n]` のケースについて考えれば十分なことがわかります[*11]。未知数は θ_0 , θ_1 , $\cdots\cdots$, θ_{n-1} , ϕ_1 , ϕ_2 , $\cdots\cdots$, ϕ_n の $2n$ 個です。まず $k=1$, 2 , $\cdots\cdots$, $n-1$ について以下の $(n-1)$ 組，つまり $2(n-1)$ 個の連立方程式を考えます。

$$\boldsymbol{\theta_k}+\boldsymbol{\phi_k}=-\psi_k$$

$$\frac{1}{l_k}{\beta_k}^2\Big(\frac{1}{\alpha_{k-1}}(\boldsymbol{\theta_{k-1}}+\boldsymbol{\phi_k})-3\boldsymbol{\phi_k}\Big)=\frac{1}{l_{k+1}}{\alpha_k}^2\Big(\frac{1}{\beta_{k+1}}(\boldsymbol{\theta_k}+\boldsymbol{\phi_{k+1}})-3\boldsymbol{\theta_k}\Big)$$

端点の〈方向指定子〉`{`d_0`}` がベクトルの場合はその方向角から θ_0 が求まります。`{`d_0`}` が `{curl` γ_0`}` のときは

$${\alpha_0}^2\Big(\frac{1}{\beta_1}(\boldsymbol{\theta_0}+\boldsymbol{\phi_1})-3\boldsymbol{\theta_0}\Big)=\gamma_0{\beta_1}^2\Big(\frac{1}{\alpha_0}(\boldsymbol{\theta_0}+\boldsymbol{\phi_1})-3\boldsymbol{\phi_1}\Big)$$

を追加します。また，〈方向指定子〉`{`d_n`}` がベクトルの場合はその方向角から ϕ_n が求まります。`{`d_n`}` が `{curl` γ_n`}` のときは

$${\beta_n}^2\Big(\frac{1}{\alpha_{n-1}}(\boldsymbol{\theta_{n-1}}+\boldsymbol{\phi_n})-3\boldsymbol{\phi_n}\Big)=\gamma_n{\alpha_{n-1}}^2\Big(\frac{1}{\beta_n}(\boldsymbol{\theta_{n-1}}+\boldsymbol{\phi_n})-3\boldsymbol{\theta_{n-1}}\Big)$$

を追加します。これで未知数 $2n$ 個に連立方程式 $2n$ 個[*12]で未知数がすべて求まることがわかります。

Exercise 3.4.4　`z0{curl 1}..tension 1 and 1..{curl 2}z1` は Z_1 と Z_2 を結ぶ線分になります。つまり $\theta_0=\phi_1=0$ となります。これを連立方程式で確認して下さい。

[*10] これも John Hobby 氏に依るそうです。

[*11] 記号の使い方は 3 点の場合に準じて読んで下さい。

[*12] $\boldsymbol{\theta_k}+\boldsymbol{\phi_k}=-\psi_k$ を除いた残りの式は，

$${\alpha_k}^2\Big(\frac{1}{\beta_{k+1}}(\boldsymbol{\theta_k}+\boldsymbol{\phi_{k+1}})-3\boldsymbol{\theta_k}\Big)\quad(\ k=0\ ,\ 1\ ,\ \cdots\cdots\ ,\ n-1\ )$$

$${\beta_k}^2\Big(\frac{1}{\alpha_{k-1}}(\boldsymbol{\theta_{k-1}}+\boldsymbol{\phi_k})-3\boldsymbol{\phi_k}\Big)\quad(\ k=1\ ,\ 2\ ,\ \cdots\cdots\ ,\ n\ )$$

の $2n$ 個をひとかたまりで見た方がわかりやすいかも知れません。

5 曲線上の時刻

METAFONT 言語では **path** 値変数 p に対して p 上の「時刻」というものを考えます。直観的にいえばこれは p を動点の軌跡ととらえ，動点の移動を「時間」の経過で表現するものです。具体的に言えば，p を構成する Bézier 曲線弧[*13]を

$$t[z_{k-1},u_{k-1},v_k,z_k]\ (\ 0\le t<1\ ,\ k=1\ ,\ \cdots\cdots\ ,\ n\ )\cdots\cdots(**)$$

とするとき，動点が点 $t_0[z_N,u_N,v_{N+1},z_{N+1}]$ を「時刻」$N+t_0$ に通過すると解釈します。とくに t が 0 以上 n 以下の整数のときは，時刻 t での通過点が p を構成するキーポイント z_t そのものに一致します。n より大きい値や負の値の「時刻」は，〈 巡回パス 〉の場合とそうでないときとで解釈が異なります。まず，〈 巡回パス 〉でないときですが，この場合，n 以上の「時刻」は終点 z_n に，0 以下の「時刻」は始点 z_0 に対応すると解釈されます。〈 巡回パス 〉のときは,「時刻」が周期 n でサイクリックに解釈されます。たとえば，n が 6 のときは,「時刻」 $\cdots\cdots$, -5 , 1 , 7 , 13 , $\cdots\cdots$ がすべて同じ点に対応します。

さて，以下ではこの「時刻」というパラメータを前提として，**path** 値変数 p からその一部を切り出したり，p を構成するキーポイントやコントロールポイントを取り出すプリミティブについて説明します[*14]。

6 subpath〜of

subpath は **path** 値からその一部を切り出すプリミティブで，

subpath 〈 ペア式 〉 of 〈 パス 1 位式 〉

の形で使います。実際の使用例については **第2部 第5講** および **第6講** で挙げましたので，ここではその正確な機能のみ説明することにします。p を **path** 値変数とし，その表示式を前項の (∗∗) 式で与えるものとします。このとき，

subpath (t_1,t_2) of p

(ここで整数 M_1 , M_2 に対して $M_1\le t_1<M_1+1$, $M_2\le t_2<M_2+1$)

は，次のような **path** 値をとります。

- $M_1<M_2$ のとき,「時刻」0 に p の「時刻」t_1 の点,「時刻」1 に p の「時刻」M_1+1 の点,「時刻」2 に p の「時刻」M_1+2 の点，$\cdots\cdots$,「時刻」M_2-M_1 に p の「時刻」M_2 の点,「時刻」M_2-M_1+1 に p の「時刻」t_2 の点で，これが終点[*15]。
- $M_1>M_2$ のとき,「時刻」0 に p の「時刻」t_1 の点,「時刻」1 に p の「時刻」M_1 の点[*16],「時刻」2 に p の「時刻」M_1-1 の点，$\cdots\cdots$,「時刻」M_1-M_2 に p の「時刻」M_2+1 の点,「時刻」M_1-M_2+1 に p の「時刻」t_2 の点で，これが終点。
- $M_1=M_2$ のとき,「時刻」0 に p の「時刻」t_1 の点,「時刻」1 に p の「時刻」t_2 の点で，これが終点。

$M_1<M_2$ のときと $M_1>M_2$ のときでは，最初と最後の曲線弧の「時刻」がもとの p に比べて引き延ばされていることに注意して下さい。この「引き延ばし」は前項で説明し

[*13] 全部で n 個あるとします。前項で説明しましたように，p を構成する **pair** と **pair** の間は，それぞれが Bézier 曲線になっています。なお，〈 巡回パス 〉のときは $z_n=z_0$ と解釈します。

[*14] p の表示式は上の (∗∗) を以下でも使うものとします。

[*15] t_2 が整数 ($t_2=M_2$) のときは「時刻」M_2-M_1 までです。

[*16] t_1 が整数 ($t_1=M_1$) のときは,「時刻」0 に p の「時刻」$M_1=t_1$ の点となりますので，以後「時刻」が 1 ずつ小さくなります。

た性質：「Bézier 曲線を途中で切ってもやはり Bézier 曲線になる」(性質 3′) を利用しています。$M_1 = M_2$ のときは，Bézier 曲線の切断を 2 回行うことになります。簡単のため $0 < t_1 < t_2 < 1$ の場合で説明すると，

`subpath` (t_1,t_2) `of p` = `subpath` $(t_1/t_2,1)$ `of (subpath` $(0,t_2)$ `of p)`

ということです。

7 path を構成する pair の抽出

次に，**path** 値変数 p を構成する **pair** の取り出し方ですが，たとえば p の「時刻」t に対応する **pair** を取り出すには `point` t `of p` とします。このプリミティブの正式な書式は

`point` ⟨ 数値式 ⟩ `of` ⟨ パス 1 位式 ⟩

で，結果は ⟨ ペア 1 位式 ⟩ になります。とくに t が 0 以上 n 以下 (n は曲線弧の数) の整数のときはキーポイントを取り出すことになります。キーポイントではなくコントロールポイントを取り出したいときは，`precontrol` , `postcontrol` を使います。使い方は `point ～ of ～` と同様で

`precontrol` ⟨ 数値式 ⟩ `of` ⟨ パス 1 位式 ⟩

および

`postcontrol` ⟨ 数値式 ⟩ `of` ⟨ パス 1 位式 ⟩

であり，たとえば `precontrol` t `of p` で「時刻」t に対応する点の直前のコントロールポイントを，`postcontrol` t `of p` で「時刻」t に対応する点の直後のコントロールポイントを，⟨ ペア 1 位式 ⟩ として得られます。

`precontrol` , `postcontrol` に，p を構成する曲線弧の数 n より大きい「時刻」や負の「時刻」を指定した場合，⟨ 巡回パス ⟩ であれば「時刻」がサイクリックに解釈されますので問題ないですが，そうでない場合は以下のように解釈されます。まず，`precontrol` ですが，これに 0 以下の「時刻」を指定すると z_0 と同じ意味になります。また，n 以上の「時刻」を指定すると，z_n の直前のコントロールポイントが得られます[*17]。`postcontrol` についても同様で，0 以下の「時刻」を指定すると z_0 の直後のコントロールポイントが，n 以上の「時刻」を指定すると，z_n がそれぞれ得られます[*18]。

8 path 値から numeric , pair 値を導く演算

path 値から **numeric** 値または **pair** 値を導く演算のうち，METAFONT , METAPOST で共通なのは `length` , `point` , `precontrol` , `postcontrol` , `directiontime` , `intersectiontimes` などです。METAPOST には他にも `arctime` , `arclength` , `center` , `llcorner` , `lrcorner` , `ulcorner` , `urcorner` などがあります。

これらのうち `point` , `precontrol` , `postcontrol` については前項で説明しましたので，ここでは残り (およびこれらから派生するいくつかのマクロ) について解説します。以下 p および q は **path** 値，a は **numeric** 値，z は **pair** 値とします。

[*17] 何か解釈が対称的でないような印象を受けるかも知れませんが，巡回でないパスでは 0 以下の「時刻」がすべて始点 z_0 , n 以上の「時刻」がすべて終点 z_n に対応すること，および，始点には直後のコントロールポイントはあっても直前のコントロールポイントはないことを考え合わせれば理解できると思います。

[*18] 終点には直前のコントロールポイントはあっても直後のコントロールポイントがないからです。

まず，length について説明します。length p は 直訳すると「p の長さ」ですが，ここで言う「長さ」は弧長ではなく，p を構成する曲線弧の個数[*19]を〈数値 1 位式〉で返します。普通の意味での弧長を求めるコマンドは METAFONT にはありませんが，METAPOST には arclength というものがあり，arclength p で p の弧長が得られます。また，METAPOST ではさらに，arctime a of p で始点からの弧長が a となる「時刻」を得られます。

次に directiontime ですが，これは directiontime z of p の形で用い，p 上はじめて接線が z と同じ向きになる「時刻」を〈数値 1 位式〉で返します。ただし，指定された向きにならないときは -1 を返します。なお，このプリミティブから派生したマクロとして directionpoint というものがあります。使い方は directiontime と同様で,「時刻」directiontime z of p に対応する点を〈ペア 1 位式〉で返します。また，directiontime とは逆に，ある「時刻」における方向を返すマクロ direction もあります。使い方は direction a of p で,「時刻」a における方向[*20]を〈ペア 1 位式〉で返します。

intersectiontimes は p intersectiontimes q の形[*21]で用い，2 つの **path** p と q の共有点の「時刻」を返します。もちろん，点としては同じであっても，p 上と q 上では異なる「時刻」に対応するでしょうから，返ってくる「時刻」は p 上の「時刻」t と q 上の「時刻」u で作られた **pair** 値 (t,u) になります[*22]。ただし，p と q に共有点がないときは $t=u=-1$ になります。また，複数の共有点があるときは，以下のような探し方で最初に見つかった共有点を採用します。

- p を構成する (始点側から見て) 最初の曲線弧に対して，q を構成する最初の曲線弧，2 番目の曲線弧，$\cdots$ の順番に共有点があるかどうか探していく。
- 次に，p を構成する 2 番目の曲線弧に対して，q を構成する最初の曲線弧，2 番目の曲線弧，$\cdots$ の順番に共有点があるかどうか探していく。
- 以下同様に，p を構成する曲線弧について次々に調べていく。

これは表現を変えると次のようになります。

- p 上で最も早く共有点が現れる曲線弧に共有点が 1 個だけならそれを採用。
- p 上で最も早く共有点が現れる曲線弧に共有点が複数個あるなら，それらが q 上で出現する曲線弧を調べる。q 上で最も早くこれらの共有点が現れる曲線弧に共有点が 1 個だけならそれを採用。

問題はこれでも 1 つに絞り込めないとき，すなわち p 上で最も早く共有点が現れる曲線弧に共有点が複数個あり，q 上でそれらが最も早く現れる曲線弧にやはり複数個の共有点があるケースですが，このとき META 言語では「シャッフル二分木検索」で初めて見つかった共有点を採用します。シャッフル二分木検索について簡単に説明しますと，まず p 側の曲線弧を 2 等分します。前半部と後半部両方に共有点があるときは，曲線弧の後半部を捨てます。前半部あるいは後半部の一方にしか共有点がないときは，共有点をもたない方を捨てます。これで残った部分にある共有点について，q 側の曲線弧を 2 等分し，上と同様

[*19]〈巡回パス〉なら p を構成する点の個数に，そうでないなら p を構成する点の 個数 -1 に等しくなります。
[*20]precontrol a of p から postcontrol a of p へのベクトルで定義されています
[*21]〈パス 3 位式〉intersectiontimes〈パス 2 位式〉の形で用い，結果は〈ペア 3 位式〉です。
[*22]t , u それぞれを取り出すには xpart , ypart を使うか，あるいは (a,b) = p intersectiontimes q; のように等値して a や b を使うことになります。

に共有点のあるなしでその一方を捨てます。以下，p , q 交互に曲線弧を次々に半分にしていきます。曲線弧が短くなっていくにつれ，共有点も絞り込まれていきますので，いつかは 1 つの共有点に絞り込めますから，そうなったらそれを採用します。これは数式的にいうと，共有点に対応する「時刻」の組 t および u の小数部分を二進法で表したものを順に $0.t_1t_2\cdots$, $0.u_1u_2\cdots$ とするとき，数 $0.t_1u_1t_2u_2\cdots$ が最も小さいものを採用することに相当します。したがって，この最後のケースで求まる共通点は，必ずしも t あるいは u が最小の共有点ではないことに注意して下さい。

なお，共有点はもちろん近似計算によって求められていますので，求まった時刻の組 (t,u) に対し，p 上の時刻 t の点と q 上の時刻 u の点はわずかにずれる可能性があります。intersectiontimes から派生してできたマクロ intersectionpoint[*23] では，これら 2 点の中点を返すようになっています。

最後に METAPOST にある center , llcorner , lrcorner , ulcorner , urcorner ですが，これらは center p のように用いて[*24]，順に p をぎりぎり囲む矩形[*25]の中心，下左，下右，上左，上右の点を表す〈 ペア 1 位式 〉を返します。

9 path に関する優先順位

path についても **numeric** や **pair** と同じように，演算に優先順位が定められています。

まず〈 パス 1 位式 〉ですが，これは〈 ペア 1 位式 〉などと同じく，〈 パス式 〉を ( ) でくくったものや〈 パス変数 〉などになります。

〈 パス 1 位式 〉を生成するプリミティブとしては，

reverse〈 パス 1 位式 〉[*26]　や　subpath〈 ペア式 〉of〈 パス 1 位式 〉

があります。

〈 パス 1 位式 〉を生成するマクロとしては，interpath や METAPOST の buildcycle などが有名です。これらについては，**第2部 第5講** および **第6講** で説明しましたので省略します。

〈 パス 2 位式 〉は〈 パス 1 位式 〉または〈 パス 2 位式 〉〈 変形演算子 〉という形式になります。〈 変形演算子 〉は〈 ペア式 〉のときにも出てきた scaled などの **変形** の意味をもつ演算子です[*27]。〈 パス 3 位式 〉は〈 パス 2 位式 〉と同義です[*28]。

そして **path** と **path** , **path** と **pair** , **pair** と **pair** を結びつけてあたらしい **path** をつくる .. や -- のいわゆる〈 パス結合 〉は 4 位の優先順位をもちます。

Exercise 3.4.5　(3,3)--(4,2) scaled 2; において，scaled 2 は **pair** である (4,2) にかかるのでしょうか，それとも **path** 全体 ((3,3)--(4,2)) にかかるのでしょうか。

Exercise 3.4.6　**path** 値 p , q が，以下のように定められているとします。

```
p:=(2,5)--(2,-5); q:=(2,0)..(1,1)..(0,0)..(1,-1)..cycle;
```

このとき，p intersectionpoint q scaled 2 はいくらになるでしょうか。

[*23] 使い方は intersectiontimes と同様です。p intersectionpoint q で p と q の共有点を返します。
[*24] center〈 パス 1 位式 〉の形で用います。
[*25] 紙面の 4 辺に平行な辺をもつ矩形です。
[*26]〈 パス 1 位式 〉の逆順の **path** を作ります。
[*27]〈 変形演算子 〉については **第7講** を参照して下さい。
[*28]〈 パス 3 位式 〉であって〈 パス 2 位式 〉でない式はデフォルトでは存在しないということです。

第 5 講

pen

1 makepen

pen 値は，プリミティブで用意されたもののほか **path** 値を変換しても作ることができます[*1]。逆に，**pen** 値を **path** 値に変換することもできます。**path** 値から **pen** 値への変換は makepen で，**pen** 値から **path** 値への変換は makepath で行います。いずれも〈引数〉は〈1 位式〉で，返値も〈1 位式〉です。

pen 値から **path** 値への変換はとくに問題ないのですが，**path** 値から **pen** 値への変換には，いくつかの制約があります。

まず，**makepen** に与える **path** 値はキーポイントのみ抽出して使用されます。わかりやすくいえば，曲線を与えても，キーポイントを単純につないでできる多角形にむりやり変換されます[*2]。

また，**pen** 値に変換する **path** 値は，〈巡回パス〉でなければなりません。これは当然といえば当然の仕様ですが，〈巡回パス〉でない場合，METAPOST では強制的に〈巡回パス〉にされてしまい，METAFONT では

```
! Pen path must be a cycle.
```

のエラーが出ます。

さらに，**pen** 値に変換する **path** 値は，凸図形でなければなりません。凸図形でない場合，METAPOST では強制的に与えられた図形を含む凸図形（凸包）にされてしまい，METAFONT では

```
! Pen cycle must be convex.
```

のエラーが出ます。なお，METAFONT では凸かどうかを，外周の多角形の頂点において，移動方向が左に曲がるか右に曲がるかで判断しますので，すべての頂点で左に曲がる「左回りの多角形」のみを凸と認識します。三角形以上ではこの点にも注意して下さい。

2 pen 値の扱いと優先順位

ここでは，**pen** 値の扱いについて説明しますが，その前にまず，〈先物ペン〉という概念について説明します。これは METAFONT においての話であって，METAPOST には関係ありません。

METAFONT では出力がビットマップである関係上，たとえ「丸ペン」であってもその外周を多角形化して処理します。この「多角形や円・楕円形の **path**」から「多角形の **pen**」

[*1] ただし，通常ペン先の形状としてはプリミティブの丸ペン（pencircle），**pen** 値変数として定義された角ペン（pensquare），カミソリペン（penrazor）ぐらいで十分ですので，**path** 値を **pen** 値に変換する機会は少ないかと思います。

[*2] 「丸ペン」の pencircle が makepen で定義されるのではなくプリミティブで与えられているのは，このような事情に依ります。

の変換処理はかなり入り組んでいるそうですので，METAFONT ではこの処理を減らしてスピードアップするために，変換処理する前のペン先のパス（＝〈先物ペン〉）と変換した後のペン（＝〈ペン〉）をきちんと区別するようになっています。もう少し具体的に言えば，**pen** 値にも **path** 値と同様〈変形演算子〉を作用させることができるのですが,「多角形の **pen**」に変換されてしまうとこの「変形」がかなり面倒になるため,〈変形演算子〉を作用させる演算 — つまり優先順位 2 位の演算 — までは〈先物ペン〉と〈ペン〉を区別し,「変形」が必要な場合はなるべく〈先物ペン〉を活用するようにし，これ以上「変形」が行われなくなってから改めて〈ペン〉に変換することを推奨しています。

また，多角形への変換にはもうひとつ深刻な問題があります。makepen で作ったペン先はもとから多角形ですので，処理スピードを問題にしなければ〈先物ペン〉と〈ペン〉の違いはさほど問題ではないとも言えるのですが，pencircle（丸ペン）という〈先物ペン〉は,「変形」後の大きさに応じてベストな多角形化[*3]を行うようできていますので,「変形」する前に多角形化して〈先物ペン〉でなくしてしまうととんでもない形になってしまいます。具体例を挙げますと，METAFONT で

```
pickup (pencircle xscaled 1pt) yscaled 1pt;
```

とすれば，ペン先は円ではなく正方形に近い六角形になってしまいます。これは後述のように ( ) のせいで (pencircle xscaled 1pt) の部分が〈ペン 1 位式〉になり，この時点で多角形化が行われ，(座標が半整数値に限定される関係上) 非常に細長いだ円が非常に細長い六角形になってしまい，そのあと yscaled が行われるためです[*4]。

さて，ここで〈ペン式〉の優先順位についてまとめておきましょう。METAPOST では〈先物ペン〉と〈ペン〉の区別はありませんので，適宜読み替えて下さい。

〈ペン 1 位式〉は〈ペン式〉を ( ) でくくったものや〈ペン変数〉, プリミティブの nullpen です。nullpen は 1 点 (0 , 0) からなり，塗りつぶしの際の境界線などに使われる，見えない線を引くペンです。〈先物ペン 1 位式〉は，makepen〈パス 1 位式〉および，プリミティブの pencircle です。

〈ペン 2 位式〉は〈ペン 1 位式〉と同義,〈先物ペン 2 位式〉は,〈先物ペン 1 位式〉および，これに〈変形演算子〉をいくつかつけたものです。〈ペン 1 位式〉に〈変形演算子〉をつけたものも文法上〈先物ペン 2 位式〉になりますが，こうして作ったものは先程説明しましたように，期待した形とは大きく異なってしまうかも知れません。

〈ペン 3 位式〉は〈ペン 2 位式〉または〈先物ペン 2 位式〉で，ここで〈ペン〉と〈先物ペン〉が合流します。4 位の〈ペン式〉は〈ペン 3 位式〉と同義です。

少しだけ補足をしておきますと,〈ペン式〉を ( ) でくくったものや〈ペン変数〉が,〈ペン 1 位式〉であって〈先物ペン 1 位式〉でないことは要注意です。このため，METAFONT では penrazor などの定義を

```
penrazor = makepen((-.5,0)--(.5,0)--cycle);
```

ではなく,

[*3] 頂点の座標が半整数 — 整数または整数に $\frac{1}{2}$ を加えた値 — である多角形に変換されます。

[*4] これは解像度の低い文字を単純に拡大すると文字のギザギザが目立つのと同根です。ちなみに，正しく「円」に拡大したいなら，( ) を使わずに,

```
pickup pencircle xscaled 1pt yscaled 1pt;
```

とするか，あるいは素直に scaled を使うべきです。

```
capsule_def(penrazor) makepen((-.5,0)--(.5,0)--cycle);
```

で行っています。前者では penrazor が〈ペン〉になってしまうからです。capsule_def は,「カプセル」(マクロの〈引数〉が一時保存される変数) が〈先物ペン〉になり得ることを利用して,〈先物ペン〉などを定義するマクロです。METAPOST ではそもそも〈ペン〉と〈先物ペン〉の区別がありませんので,penrazor の定義は前者を採用しています。

3 pen の使い方 — pickup と savepen

さて,次に **pen** 値の使い方ですが,これは currentpen という **pen** 値変数に代入して使います。currentpen 自体は特別な変数ではないのですが,draw や fill などの「基本描画命令」が currentpen を使うように定義されています[*5]ので,表面上特別な変数としてふるまいます。

ただし,currentpen へ新しい **pen** 値を代入したならば,rt などのマクロ[*6](に必要なパラメータ) も再定義しなければこれらのマクロが使い物にならなくなるため,これらをまとめてやってくれるマクロ pickup を通常用い,currentpen へ直接代入することはあまり行いません。このときの pickup の使い方は

pickup〈ペン 2 位式〉または pickup〈先物ペン 2 位式〉

のようになります。

pickup にはもうひとつ,savepen と組み合わせて使う方法があります。savepen は,よく使うペン先を保存しておく命令で,いくつ目に定義されたかの番号が返値として得られます。この番号を pickup に渡せば,保存したペンを使えるようになります。具体的には以下のような使い方をします。

```
pickup pencircle scaled 1pt;% 直径 1pt のペン
Mypen:=savepen;             % numeric 値変数 Mypen に番号を保存
       ⋮
pickup Mypen;               % 保存したペンを復活。
```

1 行目と異なり最後の行の pickup の〈引数〉が **numeric** 値であることに注意して下さい。savepen を使えば **pen** 値の呼び出しが早くなるのですが,当然記憶領域を消費しますので,むやみに使うのはあまりよくありません。保存したペンを一掃するには clear_pen_memory というマクロを使います。

なお,**pen** 値変数そのものをクリアするには nullpen を代入します。とくに currentpen に対するクリア[*7]は clearpen というマクロにまとめられています。なお,METAFONT の beginchar や METAPOST の beginfig では clearpen を自動で行ってくれます。METAPOST ではその後自動で defaultpen (直径 0.5 pt の丸ペン) を pickup してくれるのですが,METAFONT ではその後特に pickup しませんので,METAFONT ではユーザーの方で各文字毎に pickup をしなければ,draw などで描いた線が見えないことに注意して下さい。

[*5] **第6講** の **picture** 値の説明の際に述べます。
[*6] これについては次項を参照して下さい。
[*7] やはり rt などに必要なパラメータもクリアする必要があります。

4 penoffset

これはとくにフォントを作るときに問題になることですが，フォントサイズの矩形の端から端まで線を draw した場合，ペンの太さの半分ずつ描画部分が矩形の両側にはみ出します。あるいは，矩形の縁に沿って線を draw すると，線の太さの半分は矩形の外にはみ出します。このような「フォント」を並べると，隣り合う文字がくっついて見苦しくなりますので，フォントを作成する際にはペンの太さの分だけ点の位置を補正する必要があります。この補正値は，

penoffset 〈ペア式〉 of 〈ペン 1 位式〉

の形で得られます。詳しく説明しますと，〈ペア式〉で指定されるベクトルを $\overrightarrow{v}$, ペンを原点に置いたときのペン先の多角形の頂点を，左回りに A_0 , A_1 , $\cdots\cdots$, A_{n-1} , $\mathrm{A}_n\,(=\mathrm{A}_0)$ とします。$\overrightarrow{\mathrm{A}_{k-1}\mathrm{A}_k}$ の向きは，k が 1 から n まで増加する間に左回りに 1 回転します。したがってどこかで $\overrightarrow{v}$ の向きを追い越すことになります。$\overrightarrow{v}$ の向きが $\overrightarrow{\mathrm{A}_{k-1}\mathrm{A}_k}$ の向きと $\overrightarrow{\mathrm{A}_k\mathrm{A}_{k+1}}$ の向きの間のとき，penoffset $\overrightarrow{v}$ of 〈ペン 1 位式〉は頂点 A_k を返します[*8]。これを現在描画しようと思っている点に加えれば，その点にペンを置いたとき，$\overrightarrow{v}$ 方向がペンの外周の「接線[*9]」となるようなペンの「頂点」が取得できます。とくによく使われるのは $\overrightarrow{v}$ が up , down , left , right で〈ペン 1 位式〉が currentpen の場合で，これを現在描画中の点に加える計算はマクロ化されています。rt , lft , top , bot というのがそれで，たとえば，rt z1[*10] で currentpen を z1 においたとき，ペンの外周の「接線」が上を向く頂点 — 丸ペンならペン先の『右端』— が得られます[*11]。これらは〈引数〉に〈数値式〉を取ることもでき，たとえば rt x1 の場合，z1 の x 座標 x1 に「ペンの中央から『右端』への符号付き移動量」を加算することになります[*12]。

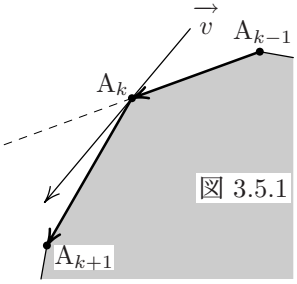


図 3.5.1

Exercise 3.5　**図 3.5.2** のソースを考えてみて下さい。なお，図を描くためのおおよその手順は以下の通りとします。まず，直径 1 の円に内接する三角形のペン先をもった **pen** 値 (pentriangle という名前にします) を定義します。これを直径 8mm に拡大したものを savepen を使って MyTri の名前で保存します。また，(w=1cm として) drawdot (3w,0); で三角形の「点」を描きます。次に，z0=(0,0) , z1=(6w,0) , z2=(6w,w) , z3=(0,w) として，直径 1pt の丸ペンで枠を描きます (draw z0--z1--z2--z3--cycle;)。最後に，保存していた MyTri を呼び出して，penoffset をうまく使って z10 , z11 , z12 を定めて draw z10..z11..z12; で曲線を描きます。

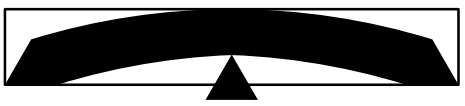


図 3.5.2

[*8] ただし，$\overrightarrow{v}$ の向きが $\overrightarrow{\mathrm{A}_{k-1}\mathrm{A}_k}$ の向きと一致してしまったときは，A_{k-1} と A_k のうち「計算しやすい方」が返ってきます。

[*9] ここで言う接線は，解析的な意味ではなく，頂点のみで接触する直線を指します。

[*10] このマクロの場合，なぜか rt.z1 のように〈トークン〉の区切りにピリオドを使うことが多いようです。

[*11] おわかりとは思いますが，(丸ペンに対して) lft は『左端』，top は『上端』，bot は『下端』を与えます。

[*12] つまり，rt x1 は z1 に currentpen をおいたときのペン先の『右端』の x 座標を与えることになります。しかし文法上はどんな〈ペア 1 位式〉,〈数値 1 位式〉でも〈引数〉として受け入れてしまうので，たとえば rt や lft に y 座標を与えたり，top や bot に x 座標を与るという意味のないことをしても，(ナンセンスな) 値が返ってきます。

第 6 講

picture

1 picture 値の扱いと優先順位

META 言語にはデフォルトで nullpicture と currentpicture という **picture** 値変数があります。前者はプリミティブで，文字通り空っぽの **picture** 値を表します[*1]。後者も読んで字のごとく,「現在描画中の絵」を表します。こちらは plain.mf や plain.mp で宣言された変数で，一般の **pictutre** 値変数と変わりないのですが，draw や fill などの「基本描画命令」が currentpicture への描画として定義されていますので，結果として特別扱いとなっています。

通常「絵」を描くには，currentpicture 1 枚だけで十分なのですが，場合によっては複数の「画像」を使い分けたいこともあるでしょう。そこで，**picture** 値変数に関する演算をまとめておきましょう。

まず **picture** 値変数への図形の書き込みは，コマンド addto で行われます。この addto には大きく分けて 3 つの使い方があります。

(1) addto 〈ピクチャー変数〉 also 〈ピクチャー式〉

これは画像の合成に使います。たとえば

```
addto pic1 also pic2
```

とすれば，pic1 に pic2 を書き足します[*2]。

(2) addto 〈ピクチャー変数〉 doublepath 〈パス式〉〈付帯リスト〉 [*3]

これは〈パス式〉の描画を行います。たぶんおわかりと思いますが，マクロ draw , undraw はこれを用いて定義されています。たとえば draw〈パス式〉の定義は以下のようになります[*4]。

```
addto currentpicture doublepath 〈パス式〉 withpen currentpen
```

通常，currentpicture への描画だけであれば draw などで事足りますので，この形式を直接使うことは少ないと思いますが，currentpicture 以外への描画をするにはこの形式を直接使うか，あるいは — 頻繁に使うのであれば — draw と同様のマクロを予め定義しておけばいいでしょう。

[*1] つまり，nullpicture を代入すれば，その **picture** 値変数の内容は消去されます。なお，currentpicture に nullpicture を代入する操作は clearit という名前でマクロ化されており，METAFONT の beginchar や METAPOST の beginfig では clearit を自動で行ってくれます。

[*2] **第2部 第3講** も参照して下さい。とくに，METAPOST では pic2 の「上書き」になることに注意して下さい。つまり，2 つの画像の色を塗った部分が重なったとき，下側の pic1 のその部分は見えなくなってしまいます。

[*3] 〈付帯リスト〉については次項を参照して下さい。

[*4] METAPOST では draw〈ピクチャー式〉で

```
addto currentpicture also〈ピクチャー式〉
```

の意味になるよう定義が拡張されています。また，METAFONT , METAPOST とも，実際にはもう少し「飾り」— METAPOST では drawoptions に指定されたもの，METAFONT では currenttransform — が付きます。

(3) addto⟨ピクチャー変数⟩ contour ⟨パス式⟩⟨付帯リスト⟩

これは ⟨パス式⟩ が ⟨巡回パス⟩ のときその内部を塗りつぶす命令で，マクロ fill , unfill , filldraw , unfilldraw , drawdot , undrawdot はこれを用いて定義されています。たとえば fill⟨パス式⟩ , filldraw⟨パス式⟩ , drawdot⟨ペア式⟩ の定義は，順に

```
addto currentpicture contour ⟨パス式⟩
addto currentpicture contour ⟨パス式⟩ withpen currentpen
addto currentpicture contour makepath currentpen shifted ⟨ペア式⟩
```

のようになります[*5]。こちらも通常は fill などで事足りるのですが，currentpicture 以外への描画 (あるいはそのような描画をするマクロの作成) にはこの形式が必要になるでしょう。

その他の演算についてですが，METAFONT では描画が加算方式ですので，+ , - で **picture** 値どうしの加算・減算もできます。しかし，上記の addto に比べると効率が悪いので，あえて使う必要はないでしょう。**picture** から **pair** などへの演算としては，**第4講 7項** で紹介した center , llcorner , lrcorner , ulcorner , urcorner[*6] が，⟨ピクチャー 1 位式⟩ に対しても適用できます。その他にもいくつかコマンドがあるのですが，データチェック的な意味合いのものがほとんどで,「絵を描く」というさしあたっての目的にはあまり必要がないと思われますのでこれらは割愛します。

最後に例によって優先順位についてまとめておきましょう。⟨ピクチャー 1 位式⟩ は ⟨ピクチャー式⟩ を ( ) でくくったものや ⟨ピクチャー変数⟩ などになります。これに shifted などの ⟨変形演算子⟩[*7]をつけたものが ⟨ピクチャー 2 位式⟩ です。METAFONT には優先順位 3 位の加算・減算がありますが，上記のようにさほど使うものではありませんので，それを除くと，⟨ピクチャー 3 位式⟩ , ⟨ピクチャー式⟩(つまり 4 位) は ⟨ピクチャー 2 位式⟩ と同義になります。

Exercise 3.6.1　**picture** 値変数 BGpicture を用意し，これに描画するマクロを一通り作ってみましょう。

2 ⟨付帯リスト⟩

前項で挙げた addto は末尾に ⟨付帯リスト⟩ をつけられると書きましたが，この ⟨付帯リスト⟩ について少し補足しておきます。

⟨付帯リスト⟩ には METAFONT , METAPOST 共通のものとしては「withpen⟨ペン式⟩」，METAFONT だけのものとしては「withweight⟨数値式⟩」，METAPOST だけのものとしては「withcolor⟨カラー式⟩」,「dashed⟨ピクチャー式⟩」があり，これらを自由に組み合わせて記述できます。もちろん，これらを使わなくてもかまいません。そのときにはそれぞれのデフォルト値が適用されます。また，同種のものを複数指定した場合は，最後に

[*5] これらは METAPOST の定義に準じています。METAFONT では drawdot の定義に withpen penspeck — penspeck は **pen** 値変数で，1 辺の長さ eps のごく小さい正方形のペン先です — をつけているなど，細かいところでいくつか違いが見られますが，本質は同じです。

[*6] **path** 値のときと同じく，METAPOST でのみ定義されています。

[*7] ⟨変形演算子⟩ については **第7講** を参照して下さい。

指定されたものが優先されますので，draw などのように定義中にすでに〈付帯リスト〉をもっているものに対しても，その設定を上書きできます。

さて，まず withpen〈ペン式〉ですが，これは文字通り描画に使うペン先の形状を指定します。もう少し具体的に言うと，draw タイプ（つまり addto 〜 doublepath 〜 に付加した場合）では，描画の線の「太さなど」を変化させ，filldraw タイプ（つまり addto 〜 contour 〜 に付加した場合）では，ペンの「太さ」の半分だけ塗りつぶす領域を広げます。省略した場合のデフォルト値は nullpen ですが，draw や filldraw などでは withpen currentpen が定義に盛り込まれていますので，これらのマクロに対する実質のデフォルトは currentpen です。したがって，実際には currentpen に値を設定する方が多いと思いますので，withpen〈ペン式〉を直接使うことは少ないと思います。

次に，withweight〈数値式〉と withcolor〈カラー式〉ですが，これらも文字通り描画の際のウェイト，色を指定します。**第2部 第3講** および **第6講** で説明しましたように，METAFONT では各ピクセルの「色（＝ウェイト）」は整数値で管理され，「塗れ」ば値が加算され，「消せ」ば値が減算されます。withweight は描画する図形に含まれるピクセルの「ウェイト」をいくつずつ増加（負のときは減少）させるかを指定します。METAPOST では「色」は文字通りの色なので，説明は不用でしょう。

withcolor は，通常 fill や draw（addto〜contour〜 や addto〜doublepath〜）で使うものですが，

```
addto pic1 also pic2 withcolor 〜
```

のように使うこともできます。ただしこのとき，pic2 内の図形の色はすべて withcolor で指定した色に統合されてしまいますのでご注意下さい。

withweight の〈引数〉は -3 , -2 , -1 , +1 , +2 , +3 もしくは整数に丸めたときこれらのいずれかになる数です。もっと大きなウェイトにしたいときには複数回「塗る」必要があります。デフォルト値は +1 です。withcolor の〈引数〉の **color** 値は各成分が 0 から 1 に限定 — 0 以下は 0 に，1 以上は 1 に統合 — されます。デフォルト値は black すなわち (0,0,0) です。

なお，unfill などの「消し」を指定する命令は，METAFONT では withweight -1 を，METAPOST では withcolor background（通常 background = white = (1,1,1)）を付加したものとして定義されています。

最後に dashed〈ピクチャー式〉ですが，これは，METAPOST で draw タイプ（つまり addto 〜 doublepath 〜 由来の命令）に作用し[*8]，線種を破線や点線に変更するのに使われます。〈ピクチャー式〉の部分は，dashed の〈引数〉としてふさわしい **picture** 値を生み出す命令 — dashpattern , evenly , withdots など — で生成するのが普通です。dashpattern は

```
dashpattern(on 2pt off 2pt on 1pt off 2pt on 2pt)
```

[*9]

のような形で使い，on の後に線の長さ，off の後に空白の大きさを指定します。evenly

[*8] その他のタイプでは無視されます

[*9] ( ) 内のパターンが繰り返されますので，両端の 2pt はつながって 4pt の線ができあがることに注意して下さい。したがってこの例は一点鎖線となります。なお，単位が省略されたときはデフォルトの bp となります。

は dashpattern(on 3 off 3) で定義されていますので，破線を生じます。withdots は dashpattern(off 2.5 on 0 off 2.5) で定義されています[*10]ので，点線を生じます。

dashpattern , evenly , withdots などで生成されるものは通常の ⟨ ピクチャー式 ⟩ ですので，scaled などの ⟨ 変形演算子 ⟩ を作用させられます。ただし，描画の際に解釈されるのはその x 軸方向の射影だけです。線の太さは withpen で指定した **pen** 値で決まりますのでご注意下さい。

なお，METAPOST では複数の描画命令に対して同じ色・線種などを指定するために drawoptions という命令が定義されています。これは

```
drawoptins(withcolor red dashed evenly);
```

のように使い，たとえばこの例ではこれ以降の描画をすべて赤で行い，線を引くときはすべて破線にすることを意味します。この命令の仕組みは単純で，⟨ 引数 ⟩ に与えられたものを _op_ というマクロに保存し，一方で，fill などの定義の末尾に，

```
def fill expr c = addto currentpicture contour c _op_ enddef;
```

のように _op_ を潜り込ませているだけです。drawoptins の設定をリセットするには drawoptins(); とします。beginfig の定義には drawoptins(); が含まれていますので，いちいち図を描き終わる毎にリセットする必要はありません。

Exercise 3.6.2　**図 3.6.1** の線分は，それぞれどのような dashpattern で生成されているか考えてみて下さい。なお，線分を構成する「断片」および「空白」のうち，長いものは 6bp , 中くらいのものは 3bp , 短いものは 1.5bp とし,「点」は on 0 で生成するものとします。

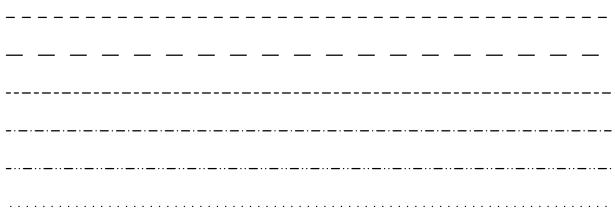


図 3.6.1

[*10] withdots は METAPOST の ver0.50 以降で導入されました。on 0 が点を生むことに注意して下さい。

第 7 講

transform

1 transform と〈変形演算子〉

まず復習ですが，**transform** 値とは数学でいう affine 変換

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} p \\ q \end{pmatrix} \cdots\cdots (*)$$

を表現するためのデータ型です。これは本来平面上の点すなわち METAFONT 言語で言うところの **pair** 値を変換するものなのですが，**pair** 値から形成される **path** 値，および，**path** 値から形成される **pen** 値，**picture** 値にも適用できます[*1]。METAFONT 言語では，**pair** 値，**path** 値，**pen** 値，**picture** 値に affine 変換を作用させる演算を〈変形演算子〉とよびますので，**transform** 値は〈変形演算子〉を表現するパラメータを保持するデータ型とも言えます。実際，`t` を **transform** 値の変数とするとき，これを `transformed t` の形で〈変形演算子〉として使えます[*2]。〈変形演算子〉のうち，回転（`rotated`）や拡大縮小（`scaled`）といった基本的な変形は，すでにプリミティブに備わっているのですが，**transform** 値を使えばこれらの合成をひとまとめにしたり，さらに一般的な変形を定義したりできます。

さて，METAFONT 言語では $(*)$ 式の変換を，その 6 つの成分（p , q , a , b , c , d）で認識しています。実際 `t` を **transform** 値とするとき，`show t;` としたときのログはこのような形式で表示されます。しかし，だからといって

```
t:=(p,q,a,b,c,d);
```

といった代入は受け付けてもらえません。もし，成分で値を指定したいなら，

```
xpart  t = p; ypart  t = q;
xxpart t = a; xypart t = b; yxpart t = c; yypart t = d;
```

のように，各成分を 1 つ 1 つ等置[*3]させていかねばなりません。この設定方法は非常に不便なのであまり使うことはないと思います。では普通はどうやって設定するのかといいますと，以下の 2 つの方法のいずれかを使うことが多いようです。

まず 1 つ目。METAFONT 言語は変数の値を設定する際に（連立）方程式を使えることを思い出して下さい。affine 変換は同一直線上にない 3 点の像によって決まりますので，これらとその像から **transform** 値を設定できます。たとえば，上記の変換では，`origin` , `right` ,

[*1] ただし，METAFONT では出力がビットマップである関係上，**picture** 値への変形には一部制限があります。具体的に言えば，各成分は整数で，affine 変換の行列部分が $\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$ または $\begin{pmatrix} 0 & a \\ b & 0 \end{pmatrix}$ の形のもの — 要するにビットマップのピクセルを壊さないもの — に限られます。METAPOST にはこのような制限はありません。

[*2] なお，これまでにたびたび出てきていますが，〈変形演算子〉とその直前の値（変形を受けるもの）の結合は優先順位 2 位です。すなわち，「〈変形を受けるもの〉〈変形演算子〉」で〈2 位式〉を構成します。

[*3] 代入ではありません。**transform** 値に限らず成分をもつデータの 1 つの成分だけを代入することはできません。

up つまり (0 , 0) , (1 , 0) , (0 , 1) の像が (p , q) , ($p+a$, $q+c$) , ($p+b$, $q+d$) ですので,

```
transform t;
origin transformed t = (p,q);
right  transformed t = (p+a,q+c);
up     transformed t = (p+c,q+d);
```

で t を設定できます。直線に関する対称変換など，図形的な意味づけがはっきりした変換の場合は，affine 変換の各パラメータよりも特徴的な点の像の方が求めやすい[*4]と思いますので，実用上これは便利な方法です。なお，plain.mf や plain.mp で定義されている identity (恒等変換 — すべての点を動かさない変換) は次のように定義されています。

```
for z=origin,right,up: z transformed identity = z; endfor
```

もう 1 つの方法は，既存の **transform** 値に ⟨ 変形演算子 ⟩ を作用させ，別の **transform** 値を作るものです[*5]。たとえば，up の方向に 3mm 移動し，原点中心に 45° 回転し，x 軸方向に 3 倍に拡大する変換は

```
shifted (3mm*up) rotated 45 xscaled 3
```

と表せますが，これを **transform** 値 t に保存し，以降 transformed t だけで済むようにするには,

```
transform t;
t := identity shifted (3mm*up) rotated 45 xscaled 3;
```

のようにします[*6]。実際に「絵」を描くときに必要なのは，平行移動，回転移動，対称移動，特定方向への拡大縮小などがほとんどで，これらは METAFONT 言語のプリミティブやその組合せで簡単に実現できますから，このような形で **transform** 値を設定する方が一般的でしょう。

Exercise 3.7.1　それでもやっぱり成分で指定したいという人のために，成分で **transform** 値を設定できるマクロを作ってみて下さい。マクロ名は setTransform とし，使い方は

```
transform t;
t:=setTransform(p,q,a,b,c,d);
```

のような形式とします。

2 定義済みの ⟨ 変形演算子 ⟩ , transform 関連の演算

次に，デフォルトで与えられている ⟨ 変形演算子 ⟩，および，⟨ 変形演算子 ⟩ や **transform** 値に関係するマクロや演算を紹介します。

まず，プリミティブとして与えられている ⟨ 変形演算子 ⟩ は

- shifted ⟨ ペア 1 位式 ⟩
- xscaled ⟨ 数値 1 位式 ⟩
- zscaled ⟨ ペア 1 位式 ⟩
- rotated ⟨ 数値 1 位式 ⟩
- scaled ⟨ 数値 1 位式 ⟩
- yscaled⟨ 数値 1 位式 ⟩
- slanted⟨ 数値 1 位式 ⟩

[*4] 演習問題の reflectedabout の定義を参照して下さい。

[*5] 演習問題の rotatedaround の定義を参照して下さい。

[*6] 恒等変換 (identity) は，この例のように，合成によって新たな **transform** 値を設定するときの出発点として使われます。

で，これらのうち zscaled 以外はすでに **第2部 第4講** で実例とともに説明しました。zscaled は **第3講 5項** で少し取り上げましたが，点を複素数とみたときの積

$$(a+bi)(c+di) = (ac-bd)+(ad+bc)i$$

を定義するもので，

$$(\ a\ ,\ b\ )\ \texttt{zscaled}\ (\ c\ ,\ d\ ) = (\ ac-bd\ ,\ ad+bc\ )$$

となります。

マクロとして定義されている〈変形演算子〉としては

reflectedabout(w,z) ····· 2 点 w , z を通る直線に関する対称移動

rotatedaround (z,d) ····· 点 z を中心とする角度 d の回転移動

があります[*7]。

transform 値の定数として定義されているものには，前項の identity があります。

transform から **numeric** へのコマンドとしては，**transform** 値の各成分を与える xpart , ypart , xxpart , xypart , yxpart , yypart があります。これらも前項で現れていますので参照して下さい。

transform から **transform** へのコマンドとしては，逆変換を求める inverse があります。これはたとえば，**transform** 値 t1 の逆変換を t2 に設定したいとき[*8]，

```
t2 = inverse t1;
```

のようにして使います。

なお，**transform** 値の演算の優先順位についてですが，**transform** 値の変数や ( ) で囲われた部分が〈変形 1 位式〉, これに〈変形演算子〉をいくつか作用させたものが〈変形 2 位式〉で，〈変形 3 位式〉,〈変形式〉(4 位) は〈変形 2 位式〉と同義です。

Exercise 3.7.2　右図において，点 A(z1) , B(z2) はすでに与えられているものとします。このとき C(z3) の位置 (z3 の値) を設定する方法を考えてみて下さい。

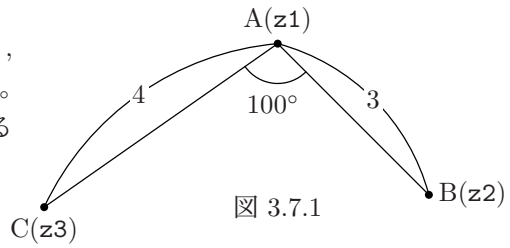


図 3.7.1

Exercise 3.7.3　reflectedabout , rotatedaround , inverse の定義は以下の通りです。定義を解析してみましょう。

```
def reflectedabout(expr w,z) =
  transformed
  begingroup transform T_;
  w transformed T_ = w;  z transformed T_ = z;
  xxpart T_ = -yypart T_; xypart T_ = yxpart T_;
  T_ endgroup enddef;
```

[*7] rotatedabout というものも定義されていますが，これは rotatedaround と同じ — rotatedaround の別名登録 — です。

[*8] transformed t2 としたあと transformed t1 とすると元に戻ることを意味します。

```
def rotatedaround(expr z, d) =
  shifted -z rotated d shifted z enddef;

vardef inverse primary T =
  transform T_; T_ transformed T = identity; T_ enddef;
```

Exercise 3.7.4　数学的な話[*9]になりますが，**transform** 値 `t2` が `t1` の逆変換のとき，すなわち，

```
transformed t2 transformed t1
```

が恒等変換のとき，(丸め誤差を除いて)

```
transformed t1 transformed t2
```

も恒等変換となることを示してみて下さい。ただし，行列の成分計算，和・積に関する結合法則，和に関する交換法則，分配法則など，基本的な知識は仮定します。

Exercise 3.7.5　3 次元の affine 変換は 3×3 の行列と 3 次元ベクトルの合計 12 個のパラメータが必要なのですが，xy 平面内などの座標平面 (と平行な平面) 内に移動を限定すれば通常の **transform** 値で表現できます[*10]。そこで，

⟨ カラー 2 位式 ⟩ `XYtransformed` ⟨ 変形 1 位式 ⟩
⟨ カラー 2 位式 ⟩ `YZtransformed` ⟨ 変形 1 位式 ⟩
⟨ カラー 2 位式 ⟩ `ZXtransformed` ⟨ 変形 1 位式 ⟩

でそれぞれ xy 平面，yz 平面，zx 平面 (に平行な平面) 内の affine 変換を実現する二項演算子 `XYtransformed` , `YZtransformed` , `ZXtransformed` を定義してみて下さい[*11]。

[*9] 数学に興味のない人はパスして下さい。
[*10] 一般の 3 次元 affine 変換については，**第4部 第9講** を参照して下さい。
[*11] これらは MEPOTEX ですでに実現されています。

第 8 講

string

1 string 値と文字列トークン

string 値は，〈文字列トークン〉[*1]を格納するデータ型です。図形を記述する METAFONT 言語では文字列を使うことは少ないだろうと思われるかも知れませんが，使い方次第ではマクロに非常に高度な機能をもたせることができます。本講では，この **string** 値の文法的扱いと簡単な使い方を説明します[*2]。

さて，とりあえず本格的な解説に入る前に，例によって演算の優先順位やデフォルトで設定されている定数をまとめておきましょう。

まず演算の優先順位ですが，〈文字列トークン〉や **string** 値変数，〈文字列式〉を ( ) で囲ったものは，これまでと同様〈文字列 1 位式〉です。デフォルトで定義されている単項演算子で **string** 値の返値をもつものもすべて〈文字列 1 位式〉を返します。それ以外の演算は文字列の結合を行う二項演算子 & だけで，これは優先順位 4 位です。したがって，デフォルトでは〈文字列 2 位式〉と〈文字列 3 位式〉は〈文字列 1 位式〉と同義になっています。〈文字列式〉(4 位) は，これらおよびこれらを & でつないだものになります。

次に定数ですが，これはデフォルトではあまりなく，METAFONT , METAPOST 共通で定義されているものは，現在処理中のソースファイル名 (拡張子部分を除く) を格納している jobname と，二重引用符 " を意味する ditto くらいです。なぜ， ditto という定数が定義されているかというと，二重引用符は〈文字列トークン〉の区切りに使われているため，文字列中に含めることができないからです。したがって，たとえば abc "def" ghi という文字列を **string** 値変数 s に格納したいのなら，

```
s:="abc␣" & ditto & "def" & ditto & "␣ghi";
```

とする必要があります。METAPOST ではファイルの入出力や PostScript フォントの描画を実装している関係上，このほかに ASCII コード 0 番の「制御文字」を意味する EOF やデフォルトフォントのフォント名 defaultfont が定義されています。

2 文字列関連の演算

ここでは **string** 値関連の単項演算子をまとめてみます。

まず，〈文字列 1 位式〉を返す演算から挙げていきます。

○ str〈サフィックス〉

〈サフィックス〉を文字列にします。ここで言う〈サフィックス〉とは，「文法上〈サフィックス〉としての要請を満たしているもの」という意味ですので，実質〈タグ〉と〈添字〉の任意の並びとなります。もちろん，〈変数名〉なども〈サフィックス〉として

[*1] " " で囲まれた文字列です。詳しくは **第3部 第1講** を参照して下さい。

[*2] 実例については **第4部** の **第3講** , **第4講** , **第5講** などを参照して下さい。

の要請を満たしますので，〈 変数名 〉を 〈 引数 〉に渡すこともできます。たとえば，

```
string␣s;
s:=␣for␣n=1␣upto␣3:␣str␣x[n]␣&␣"+"␣&␣endfor
    forsuffixes␣t=a,b:␣str␣x␣t␣&␣"+"␣&␣endfor␣str␣x.e;
```

とすれば，**string** 値変数 s に "x1+x2+x3+x.a+x.b+x.e" が設定されます。x.a と x.e のように，〈 トークン 〉の区切りの空白とピリオドは，ピリオドに統一されることにも注意して下さい。

- char〈 数値 1 位式 〉
  〈 引数 〉の 〈 数値 1 位式 〉の値に対応する ASCII コードをもつ文字を返します[*3]。
- decimal〈 数値 1 位式 〉
  〈 引数 〉の 〈 数値 1 位式 〉の値を文字列にします。小数値はそのまま小数で，整数値は小数点以下をつけない形で返します。
- substring〈 ペア式 〉of〈 文字列 1 位式 〉
  部分文字列を返します。右のように文字列を構成する文字と文字の「間」に座標を設定し，その座標で取り出す部分を指示します。たとえば substring (2,4) of "ABCDE" は "CD" を意味します。

次に，〈 文字列式 〉を 〈 引数 〉に取る演算を挙げます。

- scantokens〈 文字列 1 位式 〉
  〈 文字列 1 位式 〉の内容を通常の META 言語ソースとして解釈します。
- ASCII〈 文字列 1 位式 〉
  〈 文字列 1 位式 〉の先頭の 1 文字の ASCII コードを返します。
- oct〈 文字列 1 位式 〉と hex〈 文字列 1 位式 〉
  順に 〈 文字列 1 位式 〉を 8 進数値，16 進数値と解釈しその値を返す演算です。不適切な文字[*4] が含まれていた場合は

```
! String contains illegal digits.
```

META 言語で扱える数値を超えた場合は

```
! Number too large (数値).
```

のエラーとなります。なお，〈 文字列 1 位式 〉を 10 進数値と解釈するには scantokens が使えますので専用の命令はありません。

Exercise 3.8.1　scantokens を使えば，TEX の ¥csname 〜 ¥endcsname のように，「文字列」をマクロ名に変えることもできます。ここではその簡単な例として，

```
UniteDef("abc")("xyz")(定義内容);
```

とすれば，

```
def abcxyz = 定義内容 enddef;
```

と展開されるようなマクロ UniteDef を定義してみて下さい。

[*3] 〈 数値 1 位式 〉の値は 0 〜 127 の整数値が期待されます。そうでないときは整数に丸めた上で，128 で割った余りを採用します。

[*4] 8 進数値としては 0 〜 7 , 16 進数値としては 0 〜 9 , a 〜 f , A 〜 F のみが許されています。

3 ユーザーとの対話

たとえば正 n 角形を描画するマクロを作ったとします。ユーザーが n に不正な値 — 負数や小数 — を代入したときは条件分岐で描画しないことにしたとしましょう。このとき，単に描画せずに終わったのではユーザーの目には「訳もわからず終了した」ように見えるでしょう。ユーザーに「不正な入力があった」旨を伝えるには，故意にエラー状態を起こし，エラーメッセージを表示する必要があります。また，ユーザーに対話式にデータ入力を促すときも，どのようなデータが求められているかメッセージを表示する必要があるでしょう。ここではこういった「ユーザーとの対話」に必要な命令をまとめておきます[*5]。

- `message`〈 文字列式 〉
  実行画面とログファイルに (改行してから) 〈 文字列式 〉の内容を表示します。処理は止まりません。
- `errmessage`〈 文字列式 〉
  エラーを起こさせます。つまりエラーが起きたときと同様に `errmessage` のある行で処理が止まり，改行後 〈 文字列式 〉の前に `!`, 後ろに `.` を補ったメッセージを表示し，エラーの起きた行すなわち `errmessage` のある行が表示されます。
- `errhelp`〈 文字列式 〉
  `errmessage` で「起こさせた」エラーでユーザーがヘルプを求めてきたときに表示するメッセージを設定します。`errmessage` を使う前に予め設定しておく必要があります[*6]。
- `readstring`
  実行画面を止めてユーザーにキーボードからの入力を要求します。Enter キーを押すまでの文字列が取得され，それを 〈 文字列 1 位式 〉として返します。

Exercise 3.8.2 実際に「正 n 角形を描画するマクロ」を作ってみましょう。定義は以下の通りとします[*7]。エラー処理の部分を考えて，定義を完成させてみて下さい。

```
def drawNkaku(expr N) =
  if numeric N:
    if (N - floor N = 0) and (N>=3):
      draw (for m=1 upto N: 1cm*dir(90+360m/N)-- endfor cycle)
                                                shifted (1cm,1cm);
    else:
      (エラー処理)
    fi
  else:
```

[*5] 実例は，以下の演習問題の他，**第4部 第4講 1項** でも扱います。

[*6] 設定していないと，以下のように「ヘルプが設定されていないので，自力でがんばってね。」といったメッセージが表示されます。

```
This error message was generated by an ‘errmessage’
command, so I can’t give any explicit help.
Pretend that you’re Miss Marple:  Examine all clues,
and deduce the truth by inspired guesses.
```

ところで，このメッセージを意訳するとすれば Miss Marple は誰にしたらいいんでしょうか。

[*7] 簡単のため，向き，大きさ，位置は固定とします。

```
      （エラー処理）
    fi
  enddef;
```

4 ファイルへの入出力・文字列からの命令読み取り

METAPOST では，METAFONT に比べてファイルの入出力が強化されています[*8]。まず，

write ⟨ 文字列式 ⟩ to ⟨ ファイル名 ⟩;

は ⟨ 文字列式 ⟩ の内容を ⟨ ファイル名 ⟩ に指定されたファイルに書き込みます[*9]。⟨ ファイル名 ⟩ も ⟨ 文字列 1 位式 ⟩ であることに注意して下さい。METAPOST の処理が終了したときファイルは自動で閉じられますが，処理の途中で閉じたいとき[*10]は，**string** 値定数 EOF を書き込みます[*11]。次に，

readfrom ⟨ ファイル名 ⟩;

これは，⟨ ファイル名 ⟩ に指定されたファイルから 1 行を読み取り，⟨ 文字列 1 位式 ⟩ として値を返します。⟨ ファイル名 ⟩ は ⟨ 文字列 1 位式 ⟩ であることに注意して下さい。ファイルが読み込めない，あるいは，ファイル末端まで読み込んだ場合は，ASCII コード 0 番の文字 (定数 EOF に格納されているものと同じ) が返ってきます。ASCII コード 0 番の文字が返ってきた後で同じファイルに読み込みにいった場合は，再び先頭から読み込みます。

なお，scantokens は ⟨ 引数 ⟩ の ⟨ 文字列式 ⟩ を「1 行からなるファイルからの入力」のように扱っている，という見方もできます。⟨ 文字列式 ⟩ は scantokens されるまで，その内容は全く解釈されませんので，使い方次第ではかなり重宝します。

5 METAFONT，METAPOST 固有の使い方

METAFONT では ⟨ 文字列式 ⟩ 単独で ⟨ 文 ⟩ を作れます。これは「タイトル」とよばれ，proof モードでソースを処理して GFtoDVI で表示したとき，各ページ上部に，前ページの文字が表示されてからそのページの文字が表示されるまでに記述された「タイトル」すべてが，文字通りタイトルとして表示されます。「タイトル」は METAPOST では無視されます。

METAFONT では beginchar の第 1⟨ 引数 ⟩ にも ⟨ 文字列式 ⟩ を使えますが，これは文字コードの代用ですのでさほど面白いものではありません。

METAPOST では PostScript フォントを画像として表示するための infont というプリミティブがあります。これは

⟨ 文字列 2 位式 ⟩ infont ⟨ 文字列 1 位式 ⟩

の形で使い，⟨ 文字列 2 位式 ⟩ のほうに表示したい文字列，⟨ 文字列 1 位式 ⟩ にフォント名を指定します。返値は ⟨ ピクチャー 2 位式 ⟩ ですので，各種変形を施せます。infont を利用した各種マクロもあるのですが，これについては **第4部 第3講** で扱います。

[*8] 実例は，**第4部 第5講 1項** および **第6講** を参照して下さい。

[*9] TEX ではファイル書き出し (¥write) の前にファイルを開く (¥openout) のですが，METAPOST ではいきなり write します。なお，TEX の ¥closeout 〜 は，write EOF to 〜 に対応します (〜 に渡すものは違いますが。)。

[*10] METAPOST が同時に書き出しに使えるファイル数には上限がありますので，多数のファイルを書き出しに使うときは，使い終わったものから順に閉じていく必要があります。

[*11] EOF には ASCII コード 0 番の文字が格納されています。

第 9 講

マクロ再説

1 マクロ定義の形式

マクロ定義はたとえば以下のような形で行われます。

```
def draw expr p =
  addto currentpicture
    if picture p: also p
    else        : doublepath p withpen currentpen
    fi _op_
enddef;
```

上の例で def から = の手前までが〈定義ヘッダ〉とよばれる部分，= より後 enddef; の手前までが〈置き換えテキスト〉です。一般的に，マクロ定義は以下のようになります。

〈定義ヘッダ〉〈イコール〉〈置き換えテキスト〉enddef

このうち enddef はすべてのマクロ定義について共通ですので，とくに説明するべきことはありません。〈イコール〉は等値や代入の意味ではなく，単に〈定義ヘッダ〉と〈置き換えテキスト〉の区切りに入っている形式的なものですので，= と := いずれでも受け付けてもらえます。〈定義ヘッダ〉はさらに〈二項演算子定義ヘッダ〉[*1]，〈スパーク定義ヘッダ〉[*2]，〈変数定義ヘッダ〉の 3 つに大別されます。定義の種類は〈定義ヘッダ〉で，内容は〈置き換えテキスト〉で決まります。本節ではこの〈定義ヘッダ〉の正確な形式とその意味，および，〈置き換えテキスト〉に関するいくつかのトピックスを扱います。

2 用語の整理

さて，本題に入る前に，少し用語の整理をしておきましょう。

まずは復習から。META 言語に限らずプログラム言語は，解釈の最小単位〈トークン〉をもち，個々のプログラムは〈トークン〉を連ねて記述されます。META 言語の場合，〈トークン〉は〈文字列トークン〉，〈数値トークン〉，〈記号トークン〉の 3 つに大別されます。〈文字列トークン〉は " " で囲まれたいわゆる「文字列」，〈数値トークン〉はいわゆる「数値」です。〈記号トークン〉は，さらに〈スパーク〉と〈タグ〉に分類されます。〈スパーク〉は，通常のプログラム言語で言う「予約語」に近い概念で，特定の機能をもたされているため，〈変数名〉などには使えない[*3]などの制約をもちます。〈タグ〉は〈スパー

[*1] METAFONT ブックでは〈レベル定義ヘッダ〉とよばれています。

[*2] METAFONT ブックではとくに名前は与えていません。なお，二項演算子も（前後に 1 つずつ引数を要求する特殊な）〈スパーク〉ですので，正確には〈「〈引数〉がないか，あるいは，後置するタイプのスパーク」のための定義ヘッダ〉ということになります。

[*3] META 言語の場合，〈変数名〉の**一部にさえ**使えないという強い制約をもちます。

ク〉以外の〈記号トークン〉で，実用上は〈変数名〉に使われることになります。実際〈変数名〉は〈タグ〉もしくは〈タグ〉〈サフィックス〉の形をしています。

METAFONT 言語では，〈数値トークン〉または [〈数値式〉] を〈添字〉とよび，〈添字〉と〈タグ〉の任意の並びを〈サフィックス〉よびます。これらは，もともとこの形式が〈変数名〉のサフィックス（添字）に使われる形であることから付いた名前なのですが，〈変数名〉そのものも（先頭が〈タグ〉という制約がつくだけで）〈サフィックス〉の条件を満たしていますので，〈変数名〉も含めた広い意味で使われることもしばしばあります。とくに「サフィックス」と「添字」は，数学では同義語として使われることが多いのですが，METAFONT 言語では区別される[*4]ことに注意して下さい。

次に，プログラム言語でのマクロ[*5]定義における「引数」について。たとえば今回の最初に上げた例の〈定義ヘッド〉では draw というマクロが p という引数をもつものとして定義されていますが，これは実際に draw というマクロを使うとき，p という名前の変数を使うという意味ではありません。この場合の p は，draw が使われるまでの仮の引数で，たとえば，draw origin--up とすれば origin--up という本当の引数が p という仮の引数に自動的に代入されて使われます。この意味で，マクロ定義中の引数は〈仮引数〉とよばれ，マクロ使用時の本当の〈引数〉とは区別されます。METAFONT ブックでは〈仮引数〉を〈パラメータ〉，本当の〈引数〉を〈引数〉（argument）と記述していますが，若干紛らわしいので，以下では〈仮引数〉，〈引数〉とよび分けることにします。

METAFONT 言語のマクロには，draw のように〈引数〉を（）で囲わなくてよいものと，rotatedaround のように〈引数〉を（）で囲わなければならないものがあります。そこで，（）で囲わなくてよい引数を〈区切りなし引数〉，（）で囲わなければならない引数を〈区切り付き引数〉とよびます[*6]。もちろん，〈区切りなし引数〉は区切りを「つけなくてよい」だけで，つけても間違いではありません[*7]。また，〈区切りなし引数〉は，たとえば

```
draw := origin--up;
```

のように，手前に = や := をつけてもよいことになっています。〈区切り付き引数〉の方は「区切り」をつけなくてはいけないのですが，（**text** タイプ以外[*8]の）〈区切り付き引数〉が複数あるときは，これらをコンマで区切ることも可能です。たとえば，3 つの〈区切り付き引数〉をもつ MyMacro というマクロに対しては，

MyMacro(a)(b)(c)，MyMacro(a,b)(c)，MyMacro(a)(b,c)，MyMacro(a,b,c)

のいずれの記述も同じ意味をもちます。

マクロが〈区切りなし引数〉，〈区切り付き引数〉のどちらを要求するかは，〈定義ヘッダ〉に〈区切りなし仮引数〉，〈区切り付き仮引数〉のどちらを指定したかに依ります。もちろん，両方を要求するように定義することも可能です[*9]。

用語の解説はこのくらいにして，以下〈定義ヘッダ〉をタイプ別に見ていきましょう。

[*4]〈添字〉は〈サフィックス〉の特殊な 1 形式ということです。

[*5]言語によって，「関数」「手続き」など色々な名前でよばれていますが，要するにそういったものです。

[*6]区切りとは（デフォルトでは）（）のことです。

[*7]逆に，〈区切り付き引数〉の場合は，（）をつけないと引数と認識されず，次のようなエラーが出ます。

```
! Missing argument to マクロ名.
```

[*8]**第2部 第10講**でも説明しましたが，**text** タイプの引数はコンマ自体を引数に含められるので，（）で区切られなければいけません。

[*9]しかし，実際には〈区切り付き仮引数〉と〈区切りなし仮引数〉の両方をもつマクロは少ないようです。

3 〈二項演算子定義ヘッダ〉

〈二項演算子定義ヘッダ〉は，読んで字のごとく二項演算子型のマクロを定義するための〈定義ヘッダ〉で，具体的には以下のような形をしています。

〈レベル定義〉〈仮引数〉〈二項演算子名〉〈仮引数〉

〈仮引数〉と〈二項演算子名〉はいずれも単一の〈記号トークン〉でなければなりません。そして，〈レベル定義〉というのは，以下の 3 つです。

`primarydef` , `secondarydef` , `tartiarydef`

これまで扱ってきたように〈二項演算子〉は〈n 位式〉〈二項演算子〉〈$n-1$ 位式〉の形式で使われます。ここで，$n=2$ に相当する〈二項演算子〉を定義するのが `primarydef` , $n=3$ に相当するのが `secondarydef` , $n=4$ に相当するのが `tartiarydef` です。

1 つ具体例を挙げますと

```
primarydef x mod y = (x-y*floor(x/y)) enddef;
```

のような使い方をします。`primarydef x mod y` が〈二項演算子定義ヘッダ〉で，`x` と `y` が〈仮引数〉, `mod` がここで定義される〈二項演算子名〉です。

4 〈スパーク定義ヘッダ〉

〈スパーク定義ヘッダ〉は，〈二項演算子〉でない〈スパーク〉を定義します。〈引数〉はもたないか，あるいは，定義された〈スパーク〉の後に置かれます。〈二項演算子〉とは異なり，複数個の〈引数〉を取ることもできます。具体的には次の形をしています。

`def`〈スパーク名〉〈仮引数ヘッド〉

〈スパーク名〉はここで定義しようとしている〈スパーク〉の名前で，当然ながら単一の〈記号トークン〉でなければなりません。〈仮引数ヘッド〉は〈仮引数〉を列記する部分で，以下のような形になります。

〈区切り付き仮引数〉……〈区切り付き仮引数〉〈区切りなし仮引数〉

つまり，最後の 1 つの引数のみ区切り — つまり ( ) — をつけずに記述できる形式が使えるということです。もちろん〈区切り付き仮引数〉も〈区切りなし仮引数〉も〈空〉でもかまいませんので，〈区切り付き引数〉のみ，あるいは〈区切りなし引数〉のみ，あるいは〈引数〉を全くもたないマクロも定義できます。

〈空〉でない〈区切り付き仮引数〉には以下の 3 つのタイプがあります[*10]。

`(expr`〈仮引数〉の列`)` , `(suffix`〈仮引数〉の列`)` , `(text`〈仮引数〉の列`)`

なお，連続する同じタイプの仮引数は，コンマで区切って列記できる点に注意して下さい。1 つ具体例を挙げますと

```
def rotatedaround(expr z, d) =
 shifted -z rotated d shifted z
enddef;
```

`def rotatedaround(expr z, d)` が〈スパーク定義ヘッダ〉で，`rotatedaround` がここで定義される〈スパーク名〉です。そしてこの〈スパーク〉(マクロ) は **expr** タイプの

[*10] それぞれの意味については，**第2部 第10講** でも説明しましたが，後でもう少し厳密な説明にまとめます。

⟨ 区切り付き引数 ⟩ を 2 つもつことが，⟨ 区切り付き仮引数 ⟩(expr z, d) によって指定されています。

一方，⟨ 空 ⟩ でない ⟨ 区切りなし仮引数 ⟩ には以下の 7 つのタイプがあります[*11]。

- primary　⟨ 仮引数 ⟩
- secondary ⟨ 仮引数 ⟩
- tertiary　⟨ 仮引数 ⟩
- expr　⟨ 仮引数 ⟩
- expr　⟨ 仮引数 ⟩ of ⟨ 仮引数 ⟩
- suffix ⟨ 仮引数 ⟩
- text　⟨ 仮引数 ⟩

1 つ具体例を挙げましょう。

```
def undrawdot expr z = drawdot z withcolor background enddef;
```

def undrawdot expr z が ⟨ スパーク定義ヘッダ ⟩ で，undrawdot がここで定義される ⟨ スパーク名 ⟩ です。そしてこの ⟨ スパーク ⟩ (マクロ) は **expr** タイプの ⟨ 区切りなし引数 ⟩ を 1 つもつことが，⟨ 区切りなし仮引数 ⟩ expr z によって指定されています。

5 ⟨ 変数定義ヘッダ ⟩

⟨ 変数定義ヘッダ ⟩ は，⟨ 変数名 ⟩ と同じ形式，すなわち，⟨ タグ ⟩ から始まり，⟨ タグ ⟩ と ⟨ 添字 ⟩ の任意の並びからなる「名前」をもつマクロを作ります[*12]。具体的には次のいずれかの形をしています。

```
vardef⟨ 変数型マクロ名 ⟩⟨ 仮引数ヘッド ⟩
vardef⟨ 変数型マクロ名 ⟩@#⟨ 仮引数ヘッド ⟩
```

⟨ 変数型マクロ名 ⟩ はここで定義しようとしている ⟨ 変数型マクロ ⟩ の名前で，その形式は ⟨ 変数名 ⟩ の宣言のときに使う形式に準じます。すなわち，⟨ 添字 ⟩ 以外の ⟨ サフィックス ⟩ はそのまま，⟨ 添字 ⟩ は [] に代えて使います。たとえば ab1cd という名前でマクロを定義したいときの ⟨ 定義ヘッダ ⟩ は

```
vardef ab[]cd⟨ 仮引数ヘッド ⟩
```

のようになります[*13]。なお，変数の宣言と同じく，ab[]cd がすでにある型の変数として宣言されていたとしても，上の定義を行った時点で，その宣言は上書きされてしまいます。さらに vardef による定義の場合はそれだけにとどまらず，「ab[]cd⟨ サフィックス ⟩」の形の変数宣言すべてを取り消してしまいますので注意が必要です[*14]。⟨ 仮引数ヘッド ⟩ については前項の ⟨ スパーク定義ヘッダ ⟩ と同じだけタイプがありますので，そちらをご覧下さい。ちょっと特殊な ⟨ 仮引数 ⟩ 指定法が @# で，これは ⟨ 変数型マクロ名 ⟩ に続く ⟨ トークン ⟩ が ⟨ サフィックス ⟩ の条件を満たす場合，それを ⟨ 引数 ⟩ として渡すというものです[*15]。わかりやすくいえば，**suffix** タイプの ⟨ 区切りなし仮引数 ⟩ を 1 つだけ ⟨ 仮引数 ⟩ の先頭にも置くことができるということです。

@# を含まない例と含む例も 1 つずつ挙げておきます。

[*11] ここで現れた ⟨ 引数 ⟩ のタイプについても後でまとめます。

[*12] したがって，このタイプのみ複数個の ⟨ トークン ⟩ からなる ⟨ マクロ名 ⟩ を定義できます。

[*13] このとき，ab2cd なども同時に定義されたことになり，これらは同じ ⟨ 置き換えテキスト ⟩ をもちます。

[*14] z⟨ サフィックス ⟩ がすべてマクロとして機能し，宣言し直さない限り他の意味に使えないのはこのような事情に依ります。したがって，「a」などあまり単純な名前の ⟨ 変数型マクロ ⟩ をむやみに定義すると顰蹙かも知れません。

[*15] そういうものがなく，いきなり ⟨ 区切り付き引数 ⟩ が続いた場合は，⟨ 置き換えテキスト ⟩ 中の @# には ⟨ 空 ⟩ が代入されます。なお，@# の ⟨ 置き換えテキスト ⟩ 中での使い方も，後でまとめます。

```
vardef dir primary d = right rotated d enddef;
vardef z@#=(x@#,y@#) enddef;
```

def dir primary d や vardef z@# が〈変数定義ヘッダ〉で，dir や z がここで定義される〈変数型マクロ名〉です。

6 引数のタイプ

META 言語マクロの〈引数〉のタイプは大きく分けて 3 つあり，これらはそのマクロの定義における〈仮引数〉にどのようなタイプを指定したかによって決まります。

まず 1 つ目は **text** タイプ。これはマクロに添付した〈引数〉を，加工せずそのまま〈置き換えテキスト〉中の〈仮引数〉にはめ込みます。〈定義ヘッダ〉において **text** タイプの〈仮引数〉は，text〈仮引数〉の形で宣言します。

2 つ目は **suffix** タイプです。これは〈引数〉が〈サフィックス〉としての要請を満たしている必要があり，その中の〈添字〉は評価され〈数値トークン〉に置き換えられてから，〈置き換えテキスト〉中の〈仮引数〉にはめ込まれます[*16]。〈定義ヘッダ〉において **suffix** タイプの〈仮引数〉は，suffix〈仮引数〉の形で宣言します。

なお，**text** タイプと **suffix** タイプは〈二項演算子定義ヘッダ〉では使えません。

3 つ目は **式 (expression)** タイプです。これは〈引数〉に指定されたものを〈式〉として完全に評価してから，〈置き換えテキスト〉中の〈仮引数〉にはめ込むものです。より正確には，〈式〉評価によって得られた値が「カプセル」とよばれる特殊な変数に保存され，これが〈仮引数〉にはめ込まれます[*17]。二項演算子型のマクロにおいては，その性質上〈引数〉は必ず**式 (expression)** タイプになります。また，その位数も〈二項演算子定義ヘッダ〉の〈レベル定義〉により明らかですので，〈二項演算子定義ヘッダ〉の〈仮引数〉にはとくにタイプを識別する宣言はつきません。〈スパーク定義ヘッダ〉および〈変数定義ヘッダ〉の〈区切り付き仮引数〉において，**式 (expression)** タイプに対応する〈仮引数〉は expr〈仮引数〉の形で宣言します。一方，〈区切りなし仮引数〉においては，どこからどこまでが〈引数〉かを位数で判定する必要上，位数を込めた形で宣言する必要があります。具体的には，

〈1 位式〉を〈引数〉にしたいとき，対応する〈仮引数〉は primary〈仮引数〉
〈2 位式〉を〈引数〉にしたいとき，対応する〈仮引数〉は secondary〈仮引数〉
〈3 位式〉を〈引数〉にしたいとき，対応する〈仮引数〉は tertiary〈仮引数〉
〈式〉(4 位) を〈引数〉にしたいとき，対応する〈仮引数〉は expr〈仮引数〉

という形でそれぞれ宣言します。少し変わっているのは，

expr〈仮引数〉of〈仮引数〉

という宣言法で，これは 2 つの〈区切りなし仮引数〉をとり，of の前が〈式〉(4 位)，of の後ろが〈1 位式〉であるマクロを定義します。

Exercise 3.9.1 次のサンプルを実行すると，ログ画面 (エラーメッセージ) に何が表示されるでしょうか。また，マクロ定義中の suffix を text にするとどうなるでしょう。

[*16] たとえば MyMacro というマクロが **suffix** タイプの〈区切り付き引数〉を 1 つとるとき，MyMacro(a[3+2]) の [3+2] の部分は評価され，MyMacro(a5) が指定されたものと解釈されます。

[*17] なお,「カプセル」は代入を受け付けませんので，**式 (expression)** タイプに対応する〈仮引数〉も，〈置き換えテキスト〉中で**代入を受け付けません**。

```
def MyMacro(suffix ss) =
  a:=1; s3:=ss; errmessage s3;
enddef;

string s[]; s1:="Good Morning(^_^)/";s2:="Bad Morning(-"&¥ditto&"-)";
a:=2; MyMacro(s[a]);
end.
```

Exercise 3.9.2 `def MyMacro(expr a) = enddef;` で定義されたマクロ MyMacro を考えます。これは〈引数〉を 1 つだけ取り〈置き換えテキスト〉は〈空〉です。これを，以下の形で使ったとき，それぞれの〈引数〉はエラーなく受け入れられるでしょうか。また，〈仮引数〉のタイプ宣言を expr から suffix や text に代えるとどうなるでしょうか。

`MyMacro(1+2);` , `MyMacro(z1);` , `MyMacro("abc");` , `MyMacro(a,b,c);`

7 〈置き換えテキスト〉について

META 言語では，マクロ定義の〈定義ヘッダ〉がよみこまれると，それとマッチする enddef までの部分，すなわちそのマクロの〈置き換えテキスト〉を，解釈せずに実際に使うときまで保存します。ただし，〈定義ヘッダ〉以降を全く解釈しなければ，そもそも〈置き換えテキスト〉の終端の enddef も認識できなくなりますので，〈定義ヘッダ〉とそれにマッチする enddef および〈仮引数〉名と同じ〈トークン〉には反応します。〈置き換えテキスト〉に含まれる〈仮引数〉名と同じ〈トークン〉は，特別な「内部パラメータトークン」に置き換えられます。これは，実際にマクロが使われたとき，対応する〈引数〉を代入する「変数」に相当します。

以上のような仕様のため，〈置き換えテキスト〉中でさらに別のマクロの定義を行いたいときは，〈定義ヘッダ〉とマッチする enddef も入れなければなりません[*18]。また，〈仮引数〉に t を使うと，〈置き換えテキスト〉中のすべての t というトークンが〈仮引数〉と解釈されますので，たとえば，

```
def MyMacro(suffix t)=  a.t=5;  enddef;
```

と定義した後，このマクロを `MyMacro(b);` の形で使えば，値 5 を代入されるのは a.t ではなく a.b になってしまいます。

したがって，〈置き換えテキスト〉の中にマッチしない〈定義ヘッダ〉や enddef を入れたいときや，〈仮引数〉と同じ名前の〈トークン〉で仮引数と解釈されたくないものがあるとき[*19]は，定義読み込み時の「特別な解釈」を抑制する必要があります。これを行うプリミティブが quote で，定義読み込みの際，quote の直後のトークンは「特別な解釈」が禁止され，quote 自身は消滅します。たとえば先程の例では，

```
def MyMacro(suffix t)=  a.quote t=5;  enddef;
```

とすれば a.t に 5 が代入されるようになります。

[*18] さもなければ，もともとのマクロの〈置き換えテキスト〉の末端を，〈置き換えテキスト〉中の定義の末端と誤認してしまいます。

[*19] もっとも，後者は〈仮引数〉名を工夫することで防げますので，本来そうすべきでしょう。

〈置き換えテキスト〉中で特別な意味をもつ〈トークン〉としては，@ , #@ , @# があります。これらは〈変数定義ヘッダ〉で始まる〈変数型マクロ〉の〈置き換えテキスト〉でのみ使え[*20]，順に〈変数型マクロ名〉の最後の〈トークン〉, 最後の〈トークン〉より前すべて，最後の〈トークン〉より後ろに続く〈サフィックス〉[*21] を表します。たとえば，

```
vardef MyMacro[]@# =〈置き換えテキスト〉enddef;
```

と定義されたマクロを MyMacro3ab の形で使えば，〈変数型マクロ名〉は MyMacro3 であり，〈置き換えテキスト〉中の @ は 3 に，#@ は MyMacro に，@# は ab になります。

Exercise 3.9.3 〈定義ヘッダ〉が vardef a@# suffix b であるマクロ a を a sss ttt; の形で使うと，〈置き換えテキスト〉中で @# と b は何を表すでしょう。

Exercise 3.9.4 SetDir_z2bot(30) とすれば〈添字〉2 と〈サフィックス〉bot と本来の〈引数〉30 から bot z2=dir30 を構成するマクロ SetDir_z を定義してみて下さい。

8 グルーピング

グルーピングについては，以前から何度か触れていますが，ここでもう一度触れておきます。

通常プログラム言語では，変数の型宣言や値の変更をローカル (局所的) なものにするために，グルーピング — これらの変更を封じ込めるために「グループ」(というより範囲) を区切ること — を行います。多くの言語では,「グループ」内で加えた変更はとくに指定しない限りそのグループ内限定のローカルなものになり，グローバル (大域的) 指定のあるもののみがグループ外でも通用する全体的な変更となるのですが，METAFONT 言語の場合は逆で，とくに指定しない限り変更はグローバル，指定したもののみ変更がローカルとなります。

METAFONT 言語の場合，グルーピングは begingroup と endgroup で囲むことによって行われます。そして，変更をそのグルーピング内に限定するための指定 — 別の言い方をすれば，それまでの宣言・値などを保存して，グルーピングを抜けたときに値を復活させるよう予約 — するには，グルーピング内で「指定」をつけたい変数を save します[*22]。たとえば，グルーピング内だけで，変数 a と b をそれぞれ (1,1) と (1,-1) を意味する **pair** 値としたい場合は，

```
begingroup
  save a,b; pair a,b; a:=(1,1); b:=(1,-1);
  (各種操作)
endgroup
```

とします[*23]。

[*20] これら 3 つのうち，@# を使うには，〈定義ヘッダ〉中の〈変数型マクロ名〉の直後に @# が指定されている必要があります。

[*21]〈サフィックス〉の条件を満たす限りなるべく長くとります。

[*22] なお，save した時点で，その変数は初期化され，宣言すらされていない状態になってしまいます。これは save されるものが〈スパーク〉でもそうなりますのでご注意下さい。

[*23] なお，〈内部変数〉という特殊な変数を save し，グルーピング内で一時的に値を変更するにはするには，

```
interim〈内部変数〉:= 変更したい値;
```

とします。

ここまでは以前説明したことの復習ですが，METAFONT 言語のグルーピングには，もうひとつの使い方 — ⟨ 文 ⟩ の隠蔽 — があります。たとえば，与えられたデータ (a とします) からある種の計算規則に従って計算を行い，その値 (v とします) を最終的にある変数 (x とします) に代入することを考えます。通常なら計算を行った後，x:=v; とすればいいのですが，これをマクロ化して x:=MyMacro(a); のように使いたいとすると，マクロの ⟨ 置き換えテキスト ⟩ が

```
(計算処理で a から v を求める);
v
```

ではうまく機能しません。

```
x:=(計算処理で a から v を求める);v
```

となって，x と代入すべき v が離れてしまうためです。この場合，(**計算処理で a から v を求める**); の部分を隠蔽して，表面上ソースから消し去らねばなりません。METAFONT 言語のグルーピングにはこのような機能があり，

```
begingroup
  ⟨文 1⟩;
  ⟨文 2⟩;
    ⋮
  ⟨文 n − 1⟩;
  ⟨文 n⟩
endgroup
```

とすると，⟨ 文 1⟩; 〜 ⟨ 文 $n-1$⟩; の内容は反映した上で，なおかつこれらを隠蔽してソース上にあたかも ⟨ 文 n⟩ のみ[*24]があるように機能します。したがって，先の例では

```
begingroup
  (計算処理で a から v を求める);
  v
endgroup
```

で望みの結果が得られます。なお，begingroup 〜 endgroup で囲まれた部分は，( ) で囲まれた場合と同じように ⟨1 位式 ⟩ になります。

ちなみに，vardef を使ったマクロ定義，すなわち ⟨ 変数型マクロ ⟩ では，その名が示す通り本来値を返す変数を定義するのが目的ですので，⟨ 置き換えテキスト ⟩ 全体が，とくに指定しなくても自動的に begingroup 〜 endgroup で囲まれます。

Exercise 3.9.5　実際にやると影響が予測できなくて危険ですが，可能性の問題として，⟨ スパーク ⟩ の + などを ⟨ タグ ⟩ にするにはどうすればいいでしょう。

[*24] グルーピング内の最後の ; の後に続く ⟨ 文 ⟩ のみということです。したがって，最後の ⟨ 文 n⟩ にも ; をつけてしまうと,「最後の ; の後に続く ⟨ 文 ⟩」は ⟨ 空 ⟩ と解釈されてしまいますのでご注意下さい。

第 10 講

boolean と条件式・ループ

1 boolean 値に関する演算と優先順位

まずは，〈 論理式 〉(**boolean** 値をとる 〈 式 〉) に関連する定数や演算，優先順位についてまとめておきましょう。

まず，プリミティブで与えられている変数は `true` と `false`[*1] です。これらと 〈 論理式 〉 を ( ) でくくったものや 〈 論理変数 〉 は 〈 論理 1 位式 〉 です。

プリミティブの単項演算子で **boolean** 値の返値をもつものは，すべて 〈 論理 1 位式 〉 を返します。

boolean 値を返すプリミティブの単項演算子は，以下の表の通りです[*2]。

演算子と引数	真となる条件
`known` 〈1 位式 〉	〈1 位式 〉 の値が既知のとき
`unknown`〈1 位式 〉	〈1 位式 〉 の値が未知のとき
`numeric`〈1 位式 〉	〈1 位式 〉 が **numeric** のとき
`pair` 〈1 位式 〉	〈1 位式 〉 が **pair** のとき
`color` 〈1 位式 〉	〈1 位式 〉 が **color** のとき
`path` 〈1 位式 〉	〈1 位式 〉 が **path** のとき
`pen` 〈1 位式 〉	〈1 位式 〉 が **pen** のとき
`picture`〈1 位式 〉	〈1 位式 〉 が **picture** のとき
`transform`〈1 位式 〉	〈1 位式 〉 が **transform** のとき
`string` 〈1 位式 〉	〈1 位式 〉 が **string** のとき
`boolean`〈1 位式 〉	〈1 位式 〉 が **boolean** のとき
`cycle` 〈1 位式 〉	〈1 位式 〉 が 〈 巡回パス 〉 のとき
`odd`〈 数値 1 位式 〉	〈 数値 1 位式 〉 が奇数のとき
`not`〈 論理 1 位式 〉	〈 論理 1 位式 〉 が偽のとき
`charexists`〈 数値 1 位式 〉	〈 数値 1 位式 〉 番の文字 (画像) が出力済みのとき

なお，`numeric` などは 〈 変数宣言 〉 にも使われますし，`cycle` は 〈 巡回パス 〉 を作るときにも使われます。これをどう使い分けているかというと，

- 〈 文 〉 の先頭に `numeric` などがあれば 〈 変数宣言 〉 , そうでなければ上記の単項演算子
- 〈 パス結合 〉 の後に `cycle` があれば 〈 巡回パス 〉 を作るもの，そうでなければ上記の単項演算子

[*1] 意味はいうまでもないと思いますが，順に「真」と「偽」 です。

[*2] `color` はもちろん METAPOST のみで使えます。また，`odd` は引数を整数に丸めてから，真偽を判定します。`charexists` は — `beginchar` や `beginfig` の引数の「文字 (画像) 番号」が守備範囲ですので — 引数を整数に丸めて (METAFONT の場合はさらに 256 で割った余りについて) 真偽を判定します。

という判断です。したがって，a が **numeric** 値変数かどうかの判定結果を，**boolean** 値変数 b に等置するには，

```
b = numeric a; または (numeric a) = b;
```

としなければなりません。`numeric a = b;` では a の〈宣言〉(に不正な〈サフィックス〉がついたもの) になってしまいます。

プリミティブの二項演算子には `and` と `or` があります。

`and` は優先順位 2 位，つまり通常の積と同じで，

〈論理 2 位式〉`and`〈論理 1 位式〉

の形で使い，〈論理 2 位式〉を返します。結果は `and` の前後の〈論理式〉がともに真のときに限り真になります。

`or` は優先順位 3 位，つまり通常の和と同じで，

〈論理 3 位式〉`or`〈論理 2 位式〉

の形で使い，〈論理 3 位式〉を返します。結果は `or` の前後の〈論理式〉の少なくとも一方が真のときに真になります。

〈論理式〉を返す演算にはもうひとつ，〈関係演算子〉というものがあります。これは簡単に言ってしまえば「大小比較」で，実際プリミティブの〈関係演算子〉は以下の等号，不等号で表されます[*3]。

```
<，<=，>，>=，=，<>
```

優先順位は最低の 4 位，つまり

〈式〉〈関係演算子〉〈3 位式〉

の形で使われ〈論理式〉(4 位) を返します。もちろん，〈関係演算子〉の両側は同じタイプの〈式〉でなければならず，使えるタイプは **numeric** , **pair** , **color**[*4] , **transform** , **string** , **boolean** のみです。それぞれのタイプに対する「大小」の意味ですが，**numeric** 値については通常の数の大小，**string** 値については辞書式順序，**boolean** 値については `true` が `false` より「大」です。残りについては成分を辞書式に見ていきます。たとえば，**pair** 値については x 成分が等しくないときはそれで大小判定し，y 成分は検査しません。x 成分が等しいときに限って y 成分を検査し，最終的な判断を下します。また，未知数に対して大小判定させようとすると，

```
! Unknown relation will be considered false.
```

のエラーが出るのですが，成分をもつものについては — たとえば 2 つの **pair** 値について，x 成分が既知で異なり，y 成分が未知のとき，上記のように x 成分だけで判定が出てしまい y 成分は検査されませんので，エラーは出ません。

なお，`=` は方程式 (等置) の意味にも使われます。これの使い分けは，`=` の左辺が「〈文〉の先頭か，別の方程式の右辺の先頭[*5]」であれば等置記号，そうでなければ〈関係演算子〉という判断ですので，たとえば，a1 と a2 が等しいかどうかの判定結果を，**boolean** 値変数 b に等置するには，

[*3] 意味は明らかと思いますが，順に，記号の左が右より小さい，以下，より大きい，以上，等しい，等しくないです。

[*4] **color** 値は METAPOST のみで使えます。

[*5] たとえば，`x=y=z` において，左の `=` の左辺 `x` は〈文〉の先頭なので等置記号，右の `=` の左辺 `y` は 左の等式の右辺の先頭なのでやはり等置記号です。

```
b = (a1 = a2); または (a1 = a2) = b;
```

のように ( ) でくくって意味を明確にする必要があります。また，関係演算子は優先順位最低ですので，2 つの不等式 a<b と c>d を and などでつなぐときも，

```
(a<b) and (c>d)
```

のように ( ) でくくって優先順位を指定する必要があります。

さて，以上のような ⟨ 論理式 ⟩ を何に使うかといいますと，それは条件によって処理を分岐したり繰り返したりの制御に使います。以下ではそれらについてまとめてみたいと思います。

2 条件分岐

条件分岐は，以下の形で使います[*6]。

```
if     ⟨ 論理式 1⟩ : ⟨ テキスト 1⟩
elseif ⟨ 論理式 2⟩ : ⟨ テキスト 2⟩
elseif ⟨ 論理式 3⟩ : ⟨ テキスト 3⟩
       ⋮
elseif ⟨ 論理式 n⟩ : ⟨ テキスト n⟩
else               : ⟨ テキスト n+1⟩
fi
```

META 言語は ⟨ 論理式 1⟩ から順番に真偽を判定していって，初めて真になった ⟨ 論理式 ⟩ に対応する ⟨ テキスト ⟩ のみを残して後は捨て去ります[*7]。たとえば初めて真になった ⟨ 論理式 ⟩ が ⟨ 論理式 k⟩ とすると，あたかも初めからそこには ⟨ テキスト k⟩ のみがあったかのようにふるまいます。⟨ 論理式 1⟩ から ⟨ 論理式 n⟩ がすべて偽の場合は，else が真と判断され，⟨ テキスト $n+1$⟩ のみが残ります。上で，elseif はいくつあってもかまいませんし，全くなくてもかまいません。else: ⟨ テキスト $n+1$⟩ も省略可能で，このときは else: ⟨ 空 ⟩ が指定されたと解釈されます。

仕組みは以上の通りで，これだけを見れば通常のプログラム言語と同様なのですが，META 言語の条件分岐が他のプログラム言語の条件分岐と異質なのは，⟨ テキスト ⟩ 部分が **完結した ⟨ 文 ⟩ を単位としなくてもいい** という点です。たとえば，

```
c = (if pair c: 2 elseif color c: 1,1 fi , 0);
```

とすれば，c が **pair** 値のときは c=(2,0) , **color** 値のときは c=(1,1,0) の意味になります。つまり，META 言語の条件分岐はむしろマクロに近く — マクロが展開されてプリミティブになって初めてその意味を「解釈」されるのと同じように — 条件分岐が「展開」されるのは意味が「解釈」される前なわけです。そのためかなり柔軟な使い方ができるのですが，反面，普通のプログラム言語に慣れた人から見ると，ギョッとするような文が現れるわけです。

[*6] ⟨ 論理式 ⟩ や else の後のコロンを忘れないようにして下さい。

[*7] もう少し詳しくいうと，if に出会ったら，最初のコロンまでを条件文として真偽を判断。真ならコロンの後から次の elseif (か else) までを残して残り (fi まで) を捨て去り，偽ならば次の elseif (か else) までを捨て去り次の条件文の真偽判断，という操作を繰り返します。

3 繰り返しの制御

まず初めに，META 言語の繰り返し制御 (ループ文) は 2 つの意味でマクロに似ています。1 つは，条件分岐のところで説明しましたように，繰り返す 〈 テキスト 〉 が **完結した 〈 文 〉 を単位としなくてもいい** — やはり意味が「解釈」される前に「展開」される — という意味。もう 1 つは，たとえば，

```
for n=1,2,3: z[n] = dir 90n; endfor
```

としたとき，これが

```
z1 = dir 90;  z2 = dir 180;  z3 = dir 270;
```

と展開される様が，n を 〈 仮引数 〉 とするマクロに 〈 引数 〉 1 , 2 , 3 を順番に入れたように見えることを指します。その意味で，繰り返し制御のパラメータ (上の例では n) を，本節では 〈 仮引数 〉 とよぶことにします。

META 言語の繰り返し構造には「刻み幅指定タイプ」「**expr** 型 〈 引数 〉 列記タイプ」「**suffix** 型 〈 引数 〉 列記タイプ」「無限循環タイプ」の 4 タイプがあり[*8]，これとループ中断命令 exitif を合わせたものが「繰り返し制御」となります。

まず，ループ中断命令について先に片付けておきます。META 言語は

```
exitif〈 論理式 〉;
```

に出会うと，〈 論理式 〉 の真偽を判定します。そして，真なら (一番内側の) ループを即座に終わらせます。偽ならば何事もなかったかのように，ループが続行されます。ループ中断命令は以下で説明する 4 つの構造のどれにも使えるのですが，とくに「無限循環タイプ」と組み合わせて使うことが多いようです[*9]。

さて，では前述の 4 つの構造を順に見ていきましょう。

(1) 刻み幅指定タイプ — 通常のプログラム言語の for 文に近い書式で，以下のような形で使います[*10]。

```
for 〈 仮引数 〉〈 イコール 〉〈 初期値 〉 step 〈 刻み幅 〉 until 〈 制限値 〉:
  〈 繰り返しのテキスト 〉
endfor
```

〈 初期値 〉 , 〈 刻み幅 〉 , 〈 制限値 〉 はすべて **既知** の 〈 数値式 〉 です。〈 イコール 〉 は = または := です[*11]。

動作は，まず 〈 仮引数 〉 に 〈 初期値 〉 を代入して 〈 繰り返しのテキスト 〉，次に 〈 仮引数 〉 に 〈 刻み幅 〉 を加算して[*12]〈 繰り返しのテキスト 〉，以下これを繰り返し，〈 仮引数 〉 の値が 〈 制限値 〉 を「越えた」ら[*13]そこで終了です。「制御構造」はすべて消え，〈 繰り返しテキスト 〉 が繰り返された分だけ後に残されます。

[*8] マクロと違い 〈 引数 〉 はすぐに使われるので，当座の解釈を保留する「**text** 型 〈 引数 〉」を使う繰り返し構造はありません。

[*9] というより,「無限循環タイプ」はループ中断命令がないと，文字通り無限ループになってしまいます。

[*10] 〈 制限値 〉 の後のコロンを忘れないようにして下さい。以下のタイプでも同様です。

[*11] 他のタイプに現れる 〈 イコール 〉 も，= と := のいずれでもかまいません。

[*12] 〈 刻み幅 〉 が負のときは当然値は減ります。

[*13] 〈 刻み幅 〉 が負のときは「下回った」らです。なお，〈 初期値 〉 , 〈 刻み幅 〉 , 〈 制限値 〉 が整数でないときは，丸め誤差の影響も考えて，〈 制限値 〉 を少し大きい目に (〈 刻み幅 〉 が負のときは小さい目に) とる方がいいでしょう。

〈刻み幅〉が正で〈初期値〉が〈制限値〉より大きいとき，〈刻み幅〉が負で〈初期値〉が〈制限値〉より小さいときは，何も残さずにすべて消えてしまいます。

なお，upto は step 1 until の，downto は step -1 until の略です。

(2) **expr** 型〈引数〉列記タイプ — 以下のような形で使います。

```
for 〈仮引数〉〈イコール〉〈引数〉の列:
  〈繰り返しのテキスト〉
endfor
```

これは for n = 1, 2, 3: のように〈仮引数〉に渡す〈引数〉を，コンマで区切って列記して与えるものです。〈引数〉は〈仮引数〉に渡す前に，マクロの **expr** タイプの引数と同じように評価されます。つまり，各〈引数〉は〈式〉として評価できなければなりません。ただし，**既知か未知かは問いません**。なお，〈引数〉に〈空〉を混ぜた場合[*14]，〈空〉に相当する回は**無視されます**。

(3) **suffix** 型〈引数〉列記タイプ — 以下のような形で使います。

```
forsuffixes 〈仮引数〉〈イコール〉〈引数〉の列:
  〈繰り返しのテキスト〉
endfor
```

これは，〈引数〉が〈仮引数〉に渡される前に，マクロの **suffix** タイプの引数と同じように評価される[*15]点以外は，「**expr** 型〈引数〉列記タイプ」と同様です。なお，〈空〉は〈サフィックス〉の要請を満たしますので，こちらのタイプでは，〈引数〉に〈空〉を混ぜた場合，〈空〉に相当する回は**無視されません**。

(4) 無限循環タイプ — 以下のような形で使います。

```
forever:
  〈繰り返しのテキスト〉
endfor
```

これは〈繰り返しのテキスト〉を無限に繰り返します。ただし，これでは暴走と変わりないので，実際には前述の exitif を併用し，条件を満たしたらループを抜けるようにします。これは，C 言語や Pascal でいう while 文と似た動作です。

Exercise 3.10.1 BASIC 言語の

```
FOR N=1 TO 11 STEP 2
  S=S+N
NEXT N
```

は，META 言語ではどのように表現されるでしょうか。

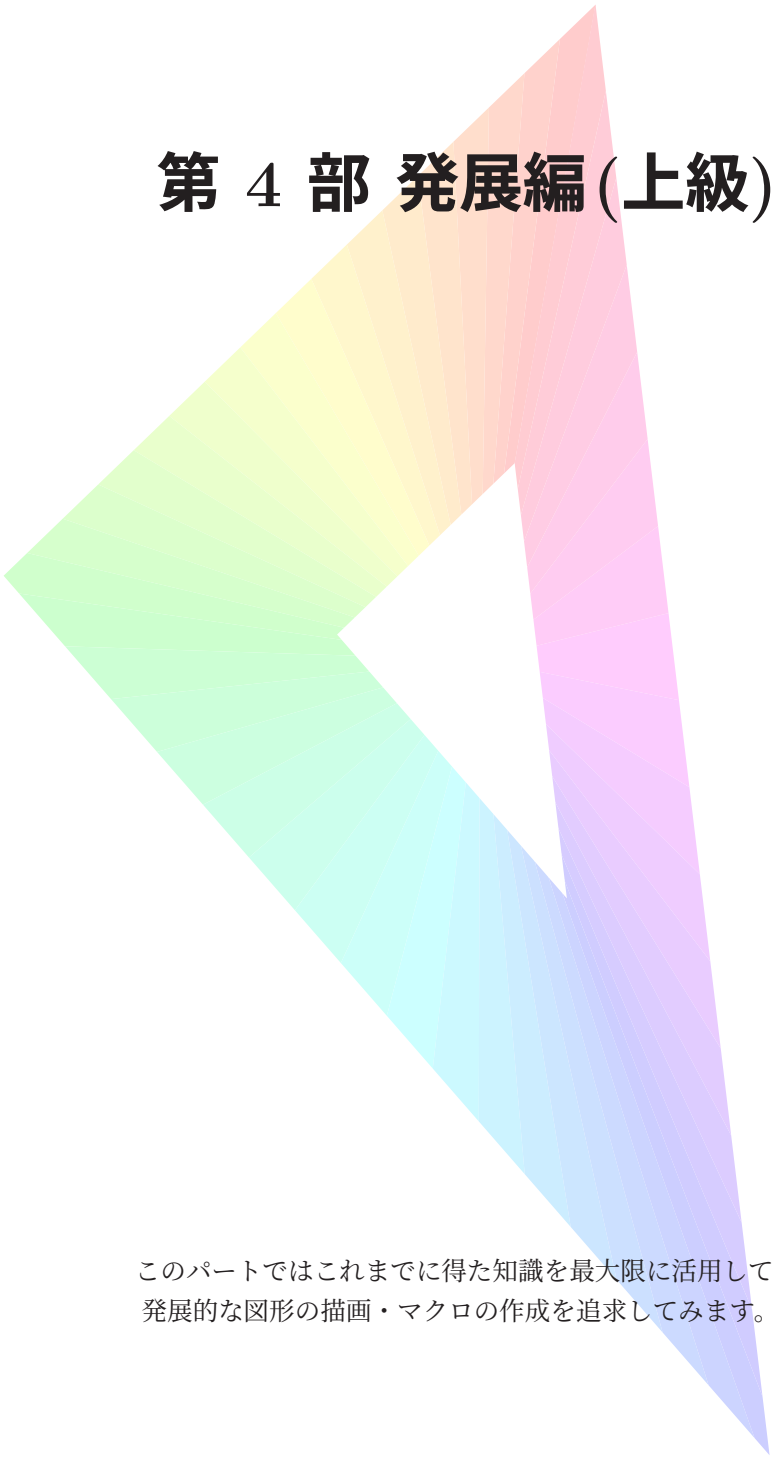
Exercise 3.10.2 Pascal 言語の repeat S:=S+N; until S>50; は，META 言語ではどのように表現されるでしょうか。

Exercise 3.10.3 C 言語の while(S<100) S = S+N; は，META 言語ではどのように表現されるでしょうか。

[*14]たとえば，n = 1, 2, , 3: とすれば 3 つ目の引数は〈空〉となります。

[*15]つまり各〈引数〉は〈タグ〉と〈添字〉の任意の並びでなければなく，〈添字〉の部分のみ評価され〈数値トークン〉におきかえられます。

第 4 部 発展編(上級)



このパートではこれまでに得た知識を最大限に活用して
発展的な図形の描画・マクロの作成を追求してみます。

第 1 講

図の配置方法

本講では，比較的地味ですが，**図 4.1.1**[*1] を描くことを目的にします。ただし，単に描くだけでは面白くないので，少し制約をつけます。

- あとで文字を入れることも考えて，図全体を拡大縮小できるようにする。
- 図全体だけでなく，部分的に隙間を広げたりもできるようにする。
- そのときも，3 つの電解槽の配置は左右対称に。電解槽や極板の形，相対位置も変化しないようにする。

とくに最後の項目が簡単に実現できるところが，METAFONT 言語の特長といえるでしょう。

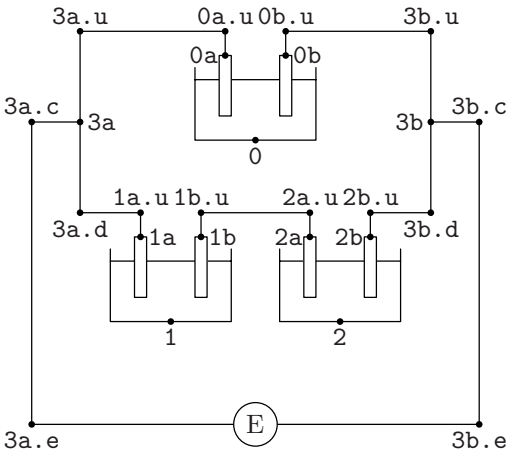


図 4.1.1

ではまず,「図全体の拡大縮小」から。これは簡単で,

「**絶対単位**を用いず**相対単位**を用いる」

ことで実現できます。**絶対単位** や **相対単位** は，今作った言葉です。— 要するに，cm や mm といった長さが固定された単位で点の座標を指定するのではなく，ソースの先頭で，たとえば

```
w=1cm; h=1cm;
```

などと定義しておき，この w や h をあたかも単位のように用いるわけです。こうしておけば，ソースの先頭の w , h の値を変更するだけで，図全体の拡大縮小ができます。なお，MEPOTEX では，MPpic 環境に入るときに与える横・縦の単位長がそれぞれ w と h に自動的に設定されます。METAFONT でも beginchar に与えたフォントの幅・高さ・深さがそれぞれ w , h , d に自動的に設定されます。METAPOST にはこのような設定はないので，自分で設定する必要があります。

次に,「図の相対関係を保ったまま，部分的に配置を変える」にはどうすればいいかです。このある意味矛盾するような要求にうってつけなのが，METAFONT 言語最大の特長

連立方程式で値を指定できる

です。

まずは **図 4.1.1** を見て下さい。この例では，電解槽や極板などは同じ大きさにするため，それぞれ 1 つの **path** を平行移動して貼り付けています。電解槽 (と水面) の参照点は，z0 , z1 , z2 (図中で 0 , 1 , 2 と表現)，極板の参照点は z0a , z1a , z2a , z0b , z1b ,

[*1] 電気分解の実験装置のつもりです。

z2b (同様に，図中で 0a , 1a , 2a , 0b , 1b , 2b と表現) です。移動前の **path** はこれらの参照点を原点とするように定義すればいいので，以下のような定義になります。

```
path denkaiso, suimen, kyokuban;
denkaiso := (-5w,6h)--(-5w,0)--(5w,0)--(5w,6h);
suimen   := (-5w,5h)--(5w,5h);
kyokuban := (-0.5w,-5h)--(0.5w,-5h)--(0.5w,0)--(-0.5w,0)--cycle;
```

このように定義した **path** を，たとえば

```
draw kyokuban shifted z1a;
```

などとすれば，望みの位置に部品を配置できます。

したがって，問題は**点の設定方法**に集約されます。点を設定するに関して，まず「どのように配置したいか」を数式化しやすいようにまとめておきましょう。

(1) z0 , z1 , z2 の重心が原点になるように配置。

(2) z2 は z0 から右に 7 単位 (7w) , 下に 15 単位 (-15h) 移動した点。
(移動量は暫定値)

(3) z1 と z2 は z0 を通る縦線に関して対称。

(4) z0a は z0 から左に 2.5 単位 (-2.5w) , 上に 7 単位 (7h) 移動した点。
(移動量は暫定値)

(5) z0a と z0b は z0 を通る縦線に関して対称。

(6) z1 , z1a , z1b の相対位置，および，z2 , z2a , z2b の相対位置は，z0 , z0a , z0b の相対位置に同じ。

ここまでのところを数式化すると，以下のようになります。複素数平面の知識がないと少しわかりにくいかも知れませんが，点 A , B を表す **pair** が a , b だとすると，

$$\overrightarrow{\mathrm{AB}} = \overrightarrow{\mathrm{OB}} - \overrightarrow{\mathrm{OA}} = \mathrm{b} - \mathrm{a}$$

となるとお考え下さい。

(1) z0+z1+z2=(0,0);

(2) z2=z0+(7w,-15h);

(3) z2-z0=(z1-z0) xscaled -1;

(4) z0a=z0+(-2.5w,7h);

(5) z0b-z0=(z0a-z0) xscaled -1;

(6) forsuffixes t=a,b: for n=1,2: z[n]t-z[n]=z0t-z0; endfor endfor

その他，配線に関しても同様の考え方で点を設定し，線分で結んでいきます。

(7) z0a.u など極板から上に伸びる線分の端点は，すべて極板の参照点 (z0a など) から上に 2 単位 (2h) 移動した点。 (移動量は暫定値)

(8) z3a , z3a.u , z3a.d は x 座標が等しく，z0a.u より 12 単位 (12w) 左で，z3a.c , z3a.e はそれよりさらに 4 単位 (4w) 左。 (移動量は暫定値)

(9) z3a , z3a.c は y 座標が等しく，z0a.u と z1a.u のちょうど真ん中の高さ。z3a.u は z0a.u と，z3a.d は z1a.u と y 座標が等しく，z3a.e は z3a.c より 25 単位 (25h) 下側の点。 (移動量は暫定値)

(10) z3b などは z3a などと z0 を通る縦線に関して左右対称。

これを数式化すると，以下のようになります。なお，z⟨サフィックス⟩はマクロで，

(x⟨サフィックス⟩,y⟨サフィックス⟩)

と同じですので，たとえば以下のように z3a を設定する代わりに x3a , y3a を別々に設定することもできます。

(7) for n=0,1,2: forsuffixes t=a,b: z[n]t.u=z[n]t+2h*up; endfor endfor

(8) x3a=x3a.u=x3a.d=x3a.c+4w=x3a.e+4w=x0a.u-12w;

(9) y3a=y3a.c=0.5[y0a.u,y1a.u];y3a.u=y0a.u;y3a.d=y1a.u;y3a.e=y3a.c-25h;

(10) forsuffixes t=,u,d,c,e: z3b.t-z0=(z3a.t-z0) xscaled -1; endfor

このようにして作ったソースなら，一部の「暫定値」を変えても，部品間のその他の「位置関係」は保たれます。色々お試し下さい。完成版のソースは以下のようになります。(点やコメントは省いてあります。)

```
¥begin{MPpic}<1.6mm>(45,0)(-20,20)
  ¥sendMP{path denkaiso, suimen, kyokuban;
    denkaiso :=(-5w,6h)--(-5w,0)--(5w,0)--(5w,6h);
    suimen   :=(-5w,5h)--(5w,5h);
    kyokuban :=(-0.5w,-5h)--(0.5w,-5h)--(0.5w,0)--(-0.5w,0)--cycle;}
  ¥sendMP{z0+z1+z2=(0,0); z2=z0+(7w,-15h); z2-z0=(z1-z0) xscaled -1;
    z0a=z0+(-2.5w,7h)   ; z0b-z0=(z0a-z0) xscaled -1;
    forsuffixes t=a,b: for n=1,2: z[n]t-z[n]=z0t-z0; endfor endfor}
  ¥sendMP{for n=0,1,2:
            draw denkaiso shifted z[n];
            draw suimen shifted z[n];
            forsuffixes t=a,b:
              unfill kyokuban shifted z[n]t;
              draw kyokuban shifted z[n]t;
              z[n]t.u=z[n]t+2h*up;
            endfor
          endfor}
  ¥sendMP{x3a=x3a.u=x3a.d=x3a.c+4w=x3a.e+4w=x0a.u-12w;
          y3a=y3a.c=0.5[y0a.u,y1a.u];
          y3a.u=y0a.u; y3a.d=y1a.u; y3a.e=y3a.c-25h;
          forsuffixes t=,u,d,c,e:
            z3b.t-z0=(z3a.t-z0) xscaled -1;
          endfor}
  ¥mptDrawPath{z0a--z0a.u--z3a.u--z3a.d--z1a.u--z1a}
  ¥mptDrawPath{z0b--z0b.u--z3b.u--z3b.d--z2b.u--z2b}
  ¥mptDrawPath{z1b--z1b.u--z2a.u--z2a}
  ¥mptDrawPath{z3a--z3a.c--z3a.e--z3b.e--z3b.c--z3b}
  ¥mptDrawPath[fillcolor=background]{circle(0.5[z3a.e,z3b.e],1.8w)}
  ¥mptLabel[fillcolor=none]{0.5[z3a.e,z3b.e]}{E}
```

```
¥end{MPpic}
```

Exercise 4.1.1　ソースの 1 箇所だけを変更して，**図 4.1.2** のように 3 つの電解槽の相対位置を上下逆に変えて下さい。また，ソースの 2 箇所だけを変更して，**図 4.1.3** のように上下の電解槽の間隔を広げ，なおかつ，すべての極板の間隔を狭めて下さい。

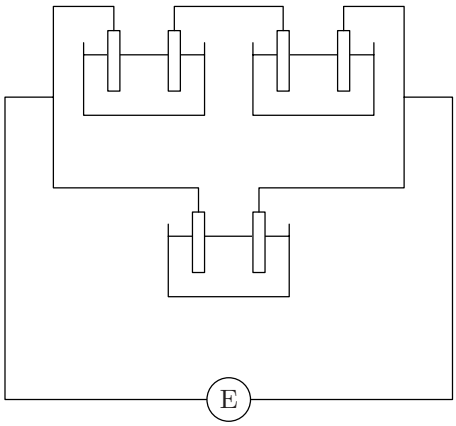


図 4.1.2

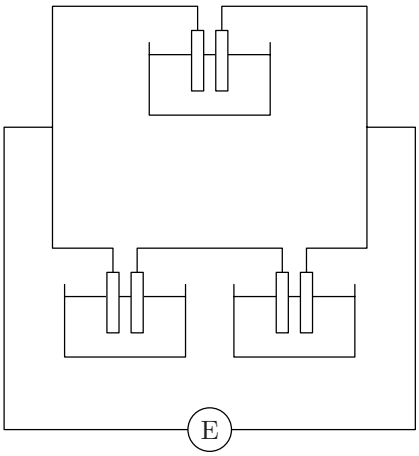


図 4.1.3

Exercise 4.1.2　z1 ～ z5 は，原点を中心とするある円周上に等間隔に並んでいます。このような点 z1 ～ z5 を設定してみましょう。ただし，中心から z1 までの距離と向きは後で調整するつもりですので，z1 を変更したら他の点も自動で変化するように設定して下さい。

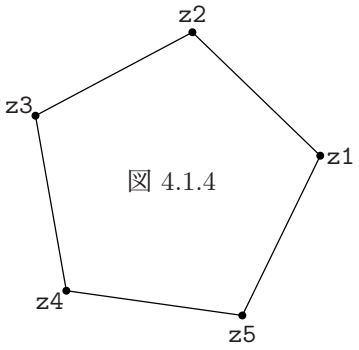


図 4.1.4

Exercise 4.1.3　**図 4.1.5** の 3 つの点 z1 , z2 , z3 は，重心が原点，頂角が 120° , 底辺が水平な二等辺三角形の頂点になっています。このような点 z1 , z2 , z3 を設定してみましょう。ただし，点の間隔は後で調整するつもりですので，z1 と z2 の間隔を変更するだけで 3 点とも調整できるように設定して下さい。

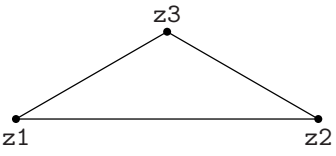


図 4.1.5

第 2 講

円グラフ

本講のテーマは「円グラフ」です。それも，単に扇形を並べるだけではおもしろくありませんので，与えられたデータから扇形の中心角や位置を計算し，扇形を配置し，ついでにコメントを入れやすいように扇形の「中央」付近の点を指定した変数に設定してくれるようなマクロを作ってみましょう。

まず，マクロの仕様を考えましょう。たとえば，

- 数値と扇形の色[*1]を 38 , Red , 38 , Blue , 28 , Yellow , ····· のように交互に指定できるようにしたい。
- 与えられた数値の合計から，各数値の割合を求め，扇形の大きさと位置を，自動的に計算したい。
- 円の中心と半径は別個に ⟨ 引数 ⟩ で与えることにする。
- 扇形の「中央」は対称軸となる半径を中心側から見て 2 : 1 に内分する点[*2]とし，この点を求める。
- 上記の「中央」点の名前を ⟨ 引数 ⟩ に指定できるようにしたい。たとえば ⟨ 引数 ⟩ に comment と指定すれば comment1 , comment2 , comment3 , ····· で上記の点にアクセスできるようにしたい。

⟨ 引数 ⟩ のうち「数値と扇形の色」は総数が不明ですので，**text** タイプの ⟨ 引数 ⟩ として取得し，for 文で処理することになります。「中心と半径」は値そのものを使いますので，**expr** タイプの ⟨ 引数 ⟩ です。「点の名前」は値ではなく文字列を ⟨ タグ ⟩ として使うので，**suffix** タイプの ⟨ 引数 ⟩ です。

以上をふまえて，

```
EnGraph(中心, 半径)(点名)(数値と色)
```

という形式でマクロを使うなら，

```
def EnGraph(expr ctr,r)(suffix nm)(text tt)
```

が ⟨ 定義ヘッダ ⟩ になります。

さて，定義の ⟨ 置き換えテキスト ⟩ ですが，まず円を **path** ⟨ 変数 ⟩ pth に取得しましょう。円周上の等間隔な 4 点以上を .. で結べばほぼ完全な円が得られますので，これで (上端を始点とする) 円をつくり，必要な扇形の弧は subpath で抜き出してくることにします。なお，今回は 4 点からなる巡回パスをつくりますので subpath (0,4) of pth で全円周です。とりあえず，円を構成する部分は次の通りです。

```
path pth;
```

[*1] 色名は MEPOSTEX には color.sty と同じ 68 色がすでに定義済みですので，MEPOSTEX を使うならとくに定義する必要はありません。METAPOST を直に使う人は MEPOSTEX から色の定義をもってきて下さい。

[*2] 中心角が 0 に近づいたとき，扇形の重心は「対称軸となる半径を中心側から見て 2 : 1 に内分する点」に近づきます。

```
pth := (up..right..down..left..cycle) scaled r shifted ctr;
```

数値と扇形の色は，それぞれ v1 , v2 , ····· と c1 , c2 , ····· という配列に取得することにしましょう。ただし，扇形の位置を決定するのに必要なのは，数値と言うより数値の累積値ですので，v1 , etc には累積値を取得するものとします。また，データの組の総数を n に取得することにしましょう[*3]。その部分は以下のようになります。

```
numeric n,v[]; n:=0; v0:=0; color c[];
for t=tt:
  n:=n+1;
  if odd n: v[(n+1)/2]:=v[(n-1)/2]+t;
  else    : c[n/2]:=t;
  fi
endfor
n:=n/2;
```

上で取得した値をもとに，どの部分の subpath をとってくるかを決め，扇形を描画する部分は，次のようになります。

```
path ptha;
for m=1 upto n:
  ptha := (subpath 4(v[m-1],v[m])/v[n] of pth)--ctr--cycle;
  fill ptha withcolor c[m];
  draw ptha;
endfor
```

さらに，上の for 文の中に，扇形の「中央」の点を求める部分を入れます。— 予め pair nm[]; とした上で

```
nm[m] := 2/3[ctr,point 2(v[m-1]+v[m])/v[n] of pth];
```

となります。

以上をまとめて，定義は次のようになります[*4]。

```
def EnGraph(expr ctr,r)(suffix nm)(text tt) =
begingroup
save pth, ptha, n, v, c;
  path pth, ptha; numeric n,v[]; color c[]; pair nm[];
  pth := (up..right..down..left..cycle) scaled r shifted ctr;
  n:=0; v0=0;
  for t=tt:
    n:=n+1;
    if odd n: v[(n+1)/2]:=v[(n-1)/2]+t;
    else    : c[n/2]:=t;
    fi
  endfor
```

[*3] したがって，数値の総計は v[n] になります。

[*4] 〈 変数 〉 をローカルにするための begingroup , save , endgroup を入れておく方がいいでしょう。

```
    n:=n/2;
    for m=1 upto n:
      ptha := (subpath 4(v[m-1],v[m])/v[n] of pth)--ctr--cycle;
      fill ptha withcolor c[m];
      draw ptha;
      nm[m]:=2/3[ctr,point 2(v[m-1]+v[m])/v[n] of pth];
    endfor
  endgroup
  enddef;
```

図 4.2.1 は，以下のようにして出力しています[*5]。コメント文字は面倒なので，MEPOTEX の隠蔽モードマクロで出力しています。¥mptLabel , ¥mptDrawPath などの使い方に関しては **第2部 第8講** をご覧下さい。

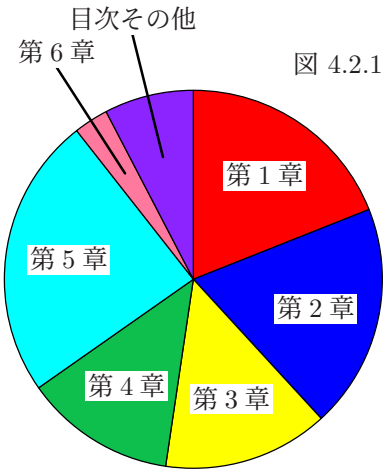


図 4.2.1 ある本の章構成

```
  ¥begin{MPpic}<5mm>(10,0)(-5,-5)
    ¥sendMP{
      EnGraph((0,0),5w)(comment)
        (38,Red,38,Blue,28,Yellow,
         26,PineGreen,48,Cyan,
         6,Salmon,15,Purple);
    }
    ¥mptDrawPath[pensize=1pt]{comment6--(2comment6)}
    ¥mptDrawPath[pensize=1pt]{comment7--(2comment7)}
    ¥mptLabel{comment1}{第 1 章}
    ¥mptLabel{comment2}{第 2 章}
    ¥mptLabel{comment3}{第 3 章}
    ¥mptLabel{comment4}{第 4 章}
    ¥mptLabel{comment5}{第 5 章}
    ¥mptLabel{2comment6}[b]{第 6 章}
    ¥mptLabel{2comment7}[b]{目次その他}
    ¥mptLabel{5w*down}[t]<0mm,-2mm>{ある本の章構成}
  ¥end{MPpic}
```

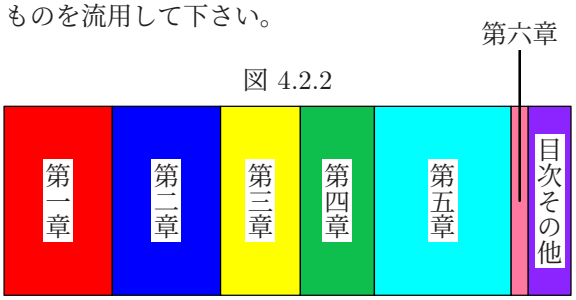
[*5] マクロ定義部分は省略しました。

Exercise 4.2.1　同様にして **図 4.2.2** のような帯グラフを作ってみましょう。なお，〈定義ヘッダ〉は

```
def ObiGraph(expr lefttop,ww,hh)(suffix nm)(text tt)
```

とします。〈仮引数〉の意味は，lefttop がグラフの左上の位置，ww が幅，hh が高さ，nm がコメントを入れる点を設定する〈変数名〉の先頭の〈タグ〉, tt が数値と色の列です。数値と色は **図 4.2.1** ものを流用して下さい。

図 4.2.2



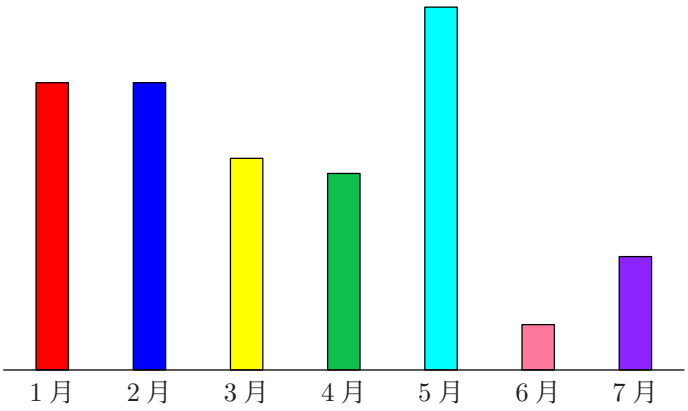
ある本の章構成

Exercise 4.2.2　**図 4.2.3** のような棒グラフを作ってみましょう。なお，〈定義ヘッダ〉は

```
def BoGraph(expr leftBline,ww,unit)(suffix nm)(text tt)
```

とします。〈仮引数〉の意味は，leftBline がグラフのベースライン左端の位置，ww が幅，unit が数値の 1 に対応する高さ，nm がコメントを入れる点を設定する〈変数名〉の先頭の〈タグ〉, tt が数値と色の列です。数値と色は **図 4.2.1** ものを流用して下さい。なお，棒の幅はグラフの幅から妥当なものを計算で設定する (たとえば，全体の幅 ÷ 数値の個数 ÷ 3) ものとします。

図 4.2.3



売　上　高

第 3 講

フォントの「描画」

本節のテーマはフォントの「描画」です。METAFONT はフォントを作成するプログラムなので，ちょっと紛らわしいですが，ここで言っているフォントの「描画」はそういう意味ではなく，既存のフォントを **picture** 値として貼り付けるという意味です[*1]。さて，サンプルは **図 4.3.1** の通りです。

図 4.3.1

ここで使っているフォントは ptmb8r というものです。世間で知られている名前でいえば Times-Bold になります。METAPOST (および MEPOTEX) でフォントを画像として扱うにはいくつか条件があります。まず，METAPOST では TEX と同じく TFM ファイルを使ってフォントサイズを取得するようですので，TFM ファイルをもつフォントしか使えません[*2]。逆に言えば TFM ファイルさえあれば METAPOST は「通る」のですが，さ

[*1] この機能をもつのは，METAPOST と，それを利用している MEPOTEX で，METAFONT にはそのような機能はありません — まぁ，METAFONT はフォントを利用する側ではなく作る側のプログラムなので，これは当たり前ですが。

[*2] ファイル検索は TEX と共通ですので，TEX が探せる TFM ファイルなら METAPOST も探し出せます。また，最近のシステムでは，TFM ファイルがないときもいきなりエラーになるのではなく，TFM ファイルの自動作成機能が働いて，必要な METAFONT ソースなどを検索し，それすらなかったときに初めてエラーが出るようになっています。

らにこれをビューアなどで見ようと思いますと，そちらでもしかるべき設定が必要になります。METAPOST の出力は PostScript ファイルですので，画像として入れたフォントの扱いは TEX の流儀ではなく PostScript の流儀に従うことになります。したがって，PostScript の流儀でビューアなどが適正にそのフォントを表示できる環境にないと，せっかく作ったファイルを見ることすらできないかも知れません[*3]。**図 4.3.1** において，TEX で普通に使われる cmr10 を使わずに ptmb8r を使ったのはそのような[*4]事情によるものです。

ちなみに，これでは本文のフォントと図に挿入するフォントをそろえるのがかなり面倒です。さらに数式など複雑なものを図に挿入するなら — METAPOST には一応 TEX を呼び出して数式などを組んで画像として取り込む機能はあるのですが — フォントの問題に加えて，TEX のスタイルファイルなどの設定を METAPOST に教える段取りが増えて，とても気軽には使えない感じになってしまいます。MEPOTEX ではその点を考慮して — そもそも MEPOTEX では TEX から METAPOST を呼び出しているのですから — TEX 側で文字を組んで，METAPOST には座標だけ計算させて，TEX 側でその文字を貼り付ける，という仕様にしています。たとえばグラフに座標や数式を書き込むといった「図形として簡単」な — でも本文とフォントがそろっていないといけないような — 文字を挿入するなら，MEPOTEX のマクロを使うべきと思います[*5]。

ずいぶん否定的なことを書いてしまいましたが，逆にたとえば「タイトル」などに使う目的で，本文とは切り離された部分でフォントを図形的に扱うのであれば，METAPOST のこの機能はかなり面白いと思いますので，以下使い方を説明してみます。

METAPOST で文字列を図形化するには infont という演算を用います。これは二項演算子の形で使い，左側に表示したい文字列を〈文字列 2 位式〉で，右側に使用フォント名を〈文字列 1 位式〉で与えます。演算結果（返値）は〈ピクチャー 2 位式〉です[*6]。たとえば初めのサンプルは以下のようにして出力されています。

```
¥begin{MPpic}<50mm>(2,2|0.1)(-1,-1)
¥openMP
numeric r; picture p; string s;
r:=1; s:="abcdefghijklmnopqrstuvwxyz";
for m:=0 upto 4: for n:=0 upto 25:
  p:=substring(n,n+1)of s infont "ptmb8r" scaled 5r; r:=0.99r;
  draw p shifted (w*r*down-center p) withcolor hsb(n/26,1,1);
  currentpicture:=currentpicture rotated -360/26;
endfor endfor
```

[*3] 見られても代替フォントが使われているかも知れません。

[*4] 通常の TEX 配布セットに TFM ファイルがあって，なおかつ，デフォルトの設定で PostScript のビューア等で見られそうなフォントという意味です。

[*5] 使い方は **第2部 第8講** で説明していますので，参照して下さい。

[*6] 文法上，通常の〈ピクチャー 2 位式〉と何ら変わりありませんので，各種〈変形演算子〉を作用させられます。たとえば，デフォルトでは文字列のベースライン左端が原点に配置されるのですが，文字列の中央を原点に配置したいのであれば，返値の〈ピクチャー式〉を **picture** 値変数 p に格納して，

```
draw p shifted (-centrer p);
```

のように平行移動で配置位置を変更できます。

```
¥closeMP
¥end{MPpic}
```

METAPOST には infont を使ったマクロがいくつか定義されています。もっとも基本的なのが

thelabel⟨サフィックス⟩(⟨文字列式またはピクチャー式⟩,⟨ペア式⟩)

で，⟨ペア式⟩に指定された位置に⟨文字列式またはピクチャー式⟩の内容を配置してできた⟨ピクチャー 1 位式⟩を返します。⟨サフィックス⟩は配置位置指定で，以下のものが使えます。

⟨空⟩ , lft , rt , top , bot , ulft , urt , llft , lrt

意味は順に，描きたいものの中央を配置点に合わせて，描きたいものを配置点の左に，右に，上に，下に，上左に，上右に，下左に，下右に配置する[*7]というものです。なお，thelabel の定義内で infont 以下は

infont defaultfont scaled defaultscale

のようになっています。defaultfont はデフォルトでは "cmr10" , defaultscale はデフォルトでは 1 となっていますので，必要ならこれらを変更してから thelabel などを使います。

thelabel の返す⟨ピクチャー 1 位式⟩を直接 currentpicture に貼り付けるよう定義されたのが label で，⟨サフィックス⟩や⟨引数⟩は thelabel と同じです。また，label の機能に加えて図の配置点に「点」を打つ機能を追加したのが dotlabel です[*8]。なお，「点」は直径 dotlabeldiam (デフォルトは 3bp) の丸ペンで描かれます。

labels , dotlabels , penlabels (語尾の s に注意) は⟨引数⟩が

(⟨文字列式またはピクチャー式⟩,⟨ペア式⟩)

ではなく

(⟨サフィックス⟩列)

となるところがこれまでと違います。これらは METAPOST で図を (試行錯誤で) 描くとき，キーポイントを明示して微調整をやりやすくするための機能で[*9]，⟨引数⟩の⟨サフィックス⟩列 に指定された⟨サフィックス⟩をもつ点に，その⟨サフィックス⟩を文字列として表示します。たとえば，labels.lft(1,2a,4); は

label.lft("1",z1); label.lft("2a",z2a); label.lft("4",z4);

が指定されたのと同等の効果をもち，点 z1 , z2a , z4 の左に，それぞれ 1 , 2a , 4 という文字列が貼り付けられます。なお，labels が label を呼び出すのに対し，dotlabels は dotlabel を呼び出すため，文字列だけではなく「点」が打たれます。penlabels は，⟨引数⟩に (1,2a,4) と指定すれば，それを (1.l,2a.l,4.l,1,2a,4,1.r,2a.r,4.r) に

[*7] それぞれ設定にしたがって配置点と図の間に空白も入れます。空白量は labeloffset (デフォルトで 3bp) に⟨サフィックス⟩に応じた倍率をかけた値になります。なお，ここでの指定法は LaTeX の ¥makebox(0,0) などを使った上下左右の指定法と逆になりますので注意して下さい。たとえば LaTeX の ¥makebox(0,0) で [r](右) を指定すると，**配置したいテキストの右端**を現在の位置に合わせるため，テキストは**配置点の左側** に配置されますが，METAPOST の label で rt(右) を指定しますと，テキストは **配置点の右側** に配置されます。

[*8] 他にも makelabel があり，これは (METAFONT との互換性のための) dotlabel の別名定義です。

[*9] METAFONT にも同様の機能をもつマクロ labels , penlabels があります — というより METAPOST の方が METAFONT の動作をエミュレートしているという方が正確でしょう。なお，METAFONT の labels は METAPOST の dotlabels に対応します。

変更して dotlabels に渡すのに相当します。これは penstroke[*10]を使うとき便利なように配慮されたマクロです。

なお，MEPOTEX では dotlabels の機能を，METAPOST の infont を使わずに TEX 側で貼り付ける文字の面倒を見るマクロ dotLabels を用意しています[*11]。MEPOTEX を使うならこちらの方が便利だと思います。

Exercise 4.3.1　長方形を描き，その頂点の座標をコメントとして入れてみましょう。METAPOST の dotlabel を用いたのが **図 4.3.2** , MEPOTEX の mptPoint を用いたのが **図 4.3.3** です。

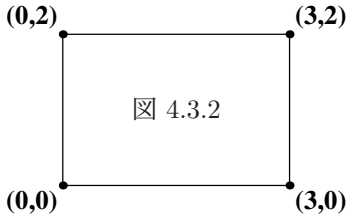


図 4.3.2

図 4.3.3

Exercise 4.3.2　曲線を描き，そのキーポイントに使った z⟨ サフィックス ⟩ の ⟨ サフィックス ⟩ をコメントとして入れてみましょう。METAPOST の dotlabels を用いたのが **図 4.3.4** , MEPOTEX の dotLabels を用いたのが **図 4.3.5** です。

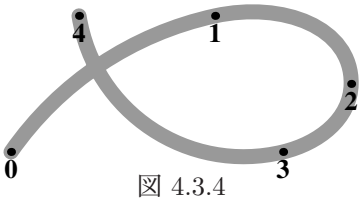


図 4.3.4

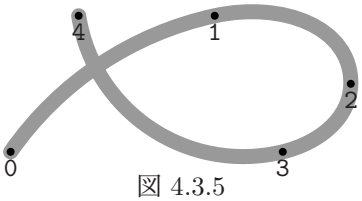


図 4.3.5

Exercise 4.3.3　**path** 値 ⟨ 変数 ⟩ p に沿ってアルファベットの 26 文字を配置してみましょう。たとえば，w=0.1mm , h=5mm として，

```
p:= origin for n=30 step 30 until 1080:..(w*n,h*sind n) endfor;
```

とすると **図 4.3.6** のような結果を期待します。

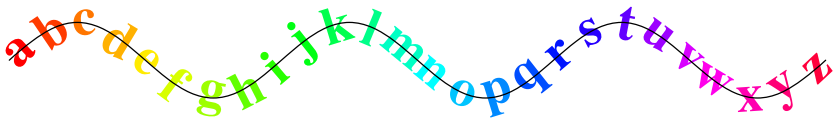


図 4.3.6

[*10] **第2部 第9講** を参照して下さい。

[*11] 使い方は，**第2部 第8講** を参照して下さい。なお，METAPOST の dotlabel とは少し違いますが，点の座標を設定すると同時に「点」を打ち，コメントもつける ¥mptPoint というマクロもあります。使い方は，演習問題を参照して下さい。

第 4 講 発展的なマクロ (1)

本講と次講では，発展的なマクロ定義や，マクロに関する発展的な話題について取り上げます。まず本講で扱うのは「ユーザーとの対話」と「情報表示」です。

1 ユーザーとの対話

ここではユーザーとの対話 — 入力を要求したり，エラーメッセージを出したり，エラーに対するヘルプを設定したり — について説明します。

この本のソースに同梱のサンプルソース `sample.tex` を LaTeX 2ε でタイプセットしてみて下さい。ログ画面で次のような「対話」を要求してくると思います。

(1)「どの形がすき？」(`Which shape do you like?`)
　選択肢が `1.` 〜 `4.` まで提示されると思います。プロンプト (`>`) に対して適正な文字 (`1` 〜 `4` の整数値) を入力[*1]すれば (3) に移ります。不正な文字を入力すれば (2) に移ります。

(2)「有効な数字は 1 , 2 , 3 , 4」(`Available numbers are 1 , 2 , 3 and 4.`) のエラーを出して止まります。ここで処理をやめるときは，通常の METAPOST のエラーの場合と同じように `x` を入力します。何も入力せず `[Enter]` キーのみを押せば続行で，(1) に戻ります。

(3)「何色がすき？」(`What color do you like?`)
　今度は選択肢番号ではなく，色名を直接入れるように要求してきます。使える色名は LaTeX 2ε の `color.sty` 準拠の 68 色です[*2]。覚えていないときは，例に挙がっている `Red` でも入力して下さい。プロンプト (`>`) に対して適正な色名を入力すれば (6) に移ります。不正な文字列を入力すれば (4) に移ります。

(4)「〜は有効な色名ではありません。」(`〜 is not an available color name.`) のエラーを出して止まります。〜 の部分には入力した文字列が入ります。ここで処理をやめるときは，通常の METAPOST のエラーの場合と同じように `x` を入力します。また，`h` を入力すれば，(5) に移ります。何も入力せず `[Enter]` キーのみを押せば続行で，(3) に戻ります。

(5) 解釈可能な 68 色の色名がずらずらっと表示されます。ここで処理をやめるときは，通常の METAPOST のエラーの場合と同じように `x` を入力します。何も入力せず `[Enter]` キーのみを押せば続行で，(3) に戻ります。

(6) 処理を終わります。
　dviout などのビューアで選んだ形・色が表示されるかどうか，確認して下さい。なお，

[*1] もちろん入力は `[Enter]` キーを押すことで確定します。以下もいちいち断りませんがその意味です。
[*2] 大文字・小文字を区別しますので注意して下さい。

dviout をご使用の方はカラー設定をお忘れなく[*3]。

サンプルの動作は以上の通りですので色々試してみて下さい。

さて，ではサンプルソースの仕組みを解析しましょう[*4]。まず — META 言語に備わっているユーザー対話関連のプリミティブは少々単純すぎますので — 今回の目的に適した機能を盛り込んだマクロをいくつか定義しておきましょう。

(1) 入力要求マクロ「Prompt(〈 文字列式 〉列)」

META 言語の実行画面で，ユーザーからの入力待ちの状態を作り，入力文字列を受け取るには readstring を使えばいいのですが，これだけではユーザーは何故実行が止まったのかわかりません。readstring の前にメッセージを表示し，何が要求されているか知らせるべきですし，入力位置にはいわゆる「プロンプト[*5]」を表示するべきでしょう。また，これらは通常のログ画面に表示されるので，通常のログとの間に空白行を入れるなりして目立つような工夫も要るでしょう。以上を加味して，Prompt の定義は以下のようになります。

```
vardef Prompt(text _t)=
  message " ";
  for _s:=_t: message _s; endfor
  message " ";
  message "> ";
  readstring
enddef;
```

message は実行するたびに (〈 引数 〉が空文字列でない限り) メッセージを表示する「前」に改行が行われますので，message " "; で空行を作れます。なお，このマクロでは，複数行のメッセージを，行毎にコンマで区切って〈 引数 〉に与えられるよう工夫されています。また，readstring で取得された文字列が返値の〈 文字列 1 位式 〉になっていることにも注意して下さい。

(2) エラーメッセージ表示マクロ「mptErr(〈 文字列式 〉)(〈 文字列式 〉列)」

エラーメッセージも複数行表示できるように工夫をしています。第 1〈 引数 〉が 1 行目，第 2〈 引数 〉が 2 行目以降です。

META 言語にもとからある errmessage は，使われた時点で実行を止めてしまい，そのときに与えた〈 引数 〉の〈 文字列式 〉だけを表示しますので，それより後ろにいくら message や errmessage を並べても (ユーザーが先に進めようとしない限り) 続きのメッセージは表示されません。もちろん，errmessage の前に message を並べれば複数行表示になるのですが，エラーメッセージの先頭に「!␣」が，末尾に「.」が追加されるという仕様から，空行以外の目的で message を使うと，少々みっともないことになり

[*3] [Option]→[Setup Parameters] メニューから表示されるダイアログの [Graphic] ページの [GIF] の項目を，[BMP(full color)] にして下さい。

[*4] errmessage などの文法は **第3部 第8講** で取り上げましたので，そちらを参照して下さい。

[*5] ここで言う「プロンプト」とは「入力要求を意味する文字」で，DOS では伝統的 (?) に「>」が使われています。

ます。加えて,「改行コード」は〈引数〉の〈文字列式〉に入れても機能しないというおまけ付きです。

仕方がないのでここでは，表示行幅 (ログ画面の横方向の文字数) を _lw という変数に設定しておき，それを用いて，複数行にしたいメッセージの各行の行末から表示画面の行末までを空白で埋めるという姑息な手を使っています[*6]。1 行目だけ「!␣」の分空白が少なくて済むことに注意して下さい。なお，本来ならば最終行も「.」が追加されることを考慮しないといけないのですが，これはあえて放置して，次の表示行にピリオド 1 個だけを送り，空行の代わりにしています。行頭にピリオド 1 個があってもたぶん気にはならないと思います。定義は以下の通りです。

```
def mptErr(expr _tt)(text _t)=
  message " ";
  errmessage _tt for _m := length _tt + 3 upto _lw: &" " endfor
             for _s := _t:
               & _s for _m := length _s + 1 upto _lw: &" " endfor
             endfor;
enddef;
```

(3) 複数行にわたるメッセージを作成するマクロ「makeStrMessage(〈サフィックス〉列)」

ここで言う〈サフィックス〉は例によって〈タグ〉と〈添字〉の任意の並びのことで，実質は〈変数名〉やメッセージを (いちいち " " で囲まず) 裸で与えられるようにしたものです[*7]。今回はヘルプメッセージで 68 個もの色名[*8]を表示しますので，色名を与えるだけでメッセージを組んでくれるマクロを用意したわけです。これも表示行幅を考えて，色名の途中で改行が起こらないよう — 追加の色名が同じ行に収まらないなら行末までを空白で埋めて次の行頭から追加の色名が始まるよう — 工夫しています。

定義は以下の通りです。このマクロは str の使用例でもありますので参考にして下さい。

```
vardef makeStrMessage(text _t)=
  save _s, _n; string _s; numeric _n; _s:=""; _n:=0;
    forsuffixes _ss := _t:
      if _n + length (str _ss & ", ") > _lw:
        _s := _s for _m := _n+1 upto _lw: &" " endfor;
        _n := 0;
      fi
      _n := _n + length (str _ss & ", ");
      _s := _s & str _ss & ", ";
    endfor
```

[*6] 変なところで改行が起こるという場合は，当方と表示行幅が異なる環境ということですので，sample.tex の _lw の値を変更して下さい。

[*7] 偶然〈スパーク〉などに合致しない限り，割とうまく働きます。

[*8] ちょっとやりすぎの気もしないではないですが，これくらいないとマクロを作る意味がありませんので。

```
          substring(0,length _s-2) of _s
                        & "." for _m := _n upto _lw: &" " endfor
        enddef;
```

以上のマクロを準備して，今回のサンプルは以下のように実現されています。

```
path p; string s; p:=origin;
forever:
  s:=Prompt("Which shape do you like?",
            "1. circle  2. square  3. triangle  4. star shape");
  if     s="1": p:=(right..up..left..down..cycle) scaled w;
  elseif s="2": p:=(dir45--dir135--dir225--dir315--cycle) scaled w;
  elseif s="3": p:=(dir90--dir210--dir330--cycle) scaled w;
  elseif s="4": z0=(dir90--dir-54) intersectionpoint (dir18--dir162);
                p:=(for n=0 upto 4: z0 rotated 72n--dir(90+72n)-- endfor
                    cycle) scaled w;
  else        : mptErr("Available numbers are 1 , 2 , 3 and 4.")
                      ("Type <Enter> to proceed.");
  fi
exitif cycle p;
endfor

forever:
  s:=Prompt("What color do you like? (ex) Red");
exitif color scantokens s;
  errhelp makeStrMessage(GreenYellow,Yellow,Goldenrod,Dandelion,Apricot,
    Peach,Melon,YellowOrange,Orange,BurntOrange,Bittersweet,RedOrange,
    Mahogany,Maroon,BrickRed,Red,OrangeRed,RubineRed,WildStrawberry,
    Salmon,CarnationPink,Magenta,VioletRed,Rhodamine,Mulberry,RedViolet,
    Fuchsia,Lavender,Thistle,Orchid,DarkOrchid,Purple,Plum,Violet,
    RoyalPurple,BlueViolet,Periwinkle,CadetBlue,CornflowerBlue,
    MidnightBlue,NavyBlue,RoyalBlue,Blue,Cerulean,Cyan,ProcessBlue,
    SkyBlue,Turquoise,TealBlue,Aquamarine,BlueGreen,Emerald,JungleGreen,
    SeaGreen,Green,ForestGreen,PineGreen,LimeGreen,YellowGreen,
    SpringGreen,OliveGreen,RawSienna,Sepia,Brown,Tan,Gray,Black,White)
    &"Type <Enter> to proceed.";
  mptErr(s&" is not an available color name.")
   ("Type <Enter> to proceed. To get available color names, type 'h'.");
endfor

fill p withcolor scantokens s;
```

最初の入力要求は，有効な番号が入力された場合のみ **path** 値変数 p が〈巡回パス〉になることを利用して，ループから抜け出します。

2 番目の入力要求は，有効な色名が入れられたときのみ **string** 値変数 s を scantokens した結果が color となることを利用してループから抜け出します。この仕様のため，色名ではなく 3 次元ベクトルを与えてもループから抜け出しますが，これは問題ないでしょう。

Exercise 4.4.1 サンプルを実際に実行して色々試してみて下さい。

2 情報表示

META 言語では，計算やマクロ定義などがうまく行われているかなどを追跡するために，変数の値やマクロの定義内容を実行画面およびログファイルに書き出す命令がいくつかあります。本項ではこのような〈情報表示命令〉について取り上げます。〈情報表示命令〉は以下の 5 つです。

〈情報表示命令〉	表示される内容
show 〈式〉の列	〈式〉の値
showvariable 〈記号トークン〉の列	変数の構造・値
showtoken 〈記号トークン〉の列	〈記号トークン〉の現在の意味
showdependencies	未知変数の依存関係
showstats	メモリ使用状況

これらのうち，showstats は見ての通りの意味ですので，以下では残りの 4 つについて，実例を挙げて説明します[*9]。

まずは show から。これはこれまでにもたびたび使っていますが，〈式〉の値を表示させる命令です。以下のような例を考えましょう。

```
showstopping:=1; path p; transform t;
z0+z1+z2=(0,0); z2=z0+(3,-6); z2-z0=(z1-z0) xscaled -1;
z3b-z3=(z3a-z3) xscaled -1;
p:=z0..z1..z2..cycle;
z1 transformed t =z2; z0 transformed t =z0; z2 transformed t =z1;
```

pair 値 z0 , z1 , z2 は決定されますが，z3 , z3a , z3b は決定されません。また，**path** 値 p と **transform** 値 t は決定されます。ここで，

```
show z0, z3, p, t;
```

とすると，実行画面に以下のように表示されます[*10]。

```
>> (0,4)
>> (0.5x3a+0.5x3b,y3)
>> path (see the transcript file)
>> (0,0,-1,0,0,1)
```

[*9] こういった実験をするときは，〈内部変数〉showstopping に正の値を設定しておくと，〈情報表示命令〉毎に実行画面が止まるため，確認が楽になります。

[*10] show を **path** , **pen** , **picture** に対して使った場合は，通常ログファイルにのみ情報が書き出されます。これを実行画面にも表示させるには，〈内部変数〉tracingonline に正の値を設定します。

順に z0 , z3 , p , t の値を指します。p はログファイルを見ると，以下のように詳しく表示されています[*11]。

```
>> (0,4)
>> (0.5x3a+0.5x3b,y3)
>> Path at line 7:
(0,4)..controls (-3.04355,4) and (-4.80011,0.51659)
 ..(-3,-2)..controls (-1.52878,-4.05676) and (1.52878,-4.05676)
 ..(3,-2)..controls (4.80013,0.51659) and (3.04355,4)
 ..cycle
>> (0,0,-1,0,0,1)
```

なお，METAFONT と METAPOST では出力形式が違うため，**pen** および **picture** に対する内部表現は大きく異なっています。たとえば，上の例で

```
fill p; show currentpicture;
```

とすると，METAFONT では

```
>> Edge structure at line 9:
row 3: 2- -2+ |
row 2: 3- -3+ |
row 1: 4- -4+ |
row 0: 4- -4+ |
row -1: 4- -4+ |
row -2: 4- -4+ |
row -3: 3- -3+ |
row -4: 2- -2+ |
```

のように，ピクセルに割り振られたウェイト値の変化する位置を示すデータが得られ，METAPOST では

```
>> Edge structure at line 7:
Filled contour :
(0,4)..controls (-3.04355,4) and (-4.80011,0.51659)
 ..(-3,-2)..controls (-1.52878,-4.05676) and (1.52878,-4.05676)
 ..(3,-2)..controls (4.80013,0.51659) and (3.04355,4)
 ..cycle
End edges
```

のような，**path** と同様の表示になります。また，

```
show pencircle scaled 4;
```

とすると，METAFONT では

```
>> Pen polygon at line 9:
(0.5,-2)
 .. (2,-1)
```

[*11] at line の部分に表示されている行番号は，ソース行でその **path** 値が現れた行です。先の例では主要部分しか挙げていないため，この行番号についてはあまり気にしないで下さい。

```
.. (2,1)
.. (0.5,2)
.. (-0.5,2)
.. (-2,1)
.. (-2,-1)
.. (-0.5,-2)
.. cycle
```

のように，pencircle が多角形化されているのがわかり，METAPOST では

```
>> Pen at line 7:
pencircle transformed (0,0,4,0,0,4)
```

のように，(少なくとも show で見る限り) 多角形化が行われていないことがわかります[*12]。

次は，showvariable — これは，与えた ⟨ 記号トークン ⟩ を先頭 ⟨ タグ ⟩ にもつ変数および ⟨ 変数型マクロ ⟩ の構造と値を表示します[*13]。たとえば先の例で

```
showvariable x,p,c,z,dir,upto,addto,dotprod;
```

とすると，以下のように表示されます。

```
x[]=numeric
x[]a=numeric
x[]b=numeric
x0=0
x1=-3
x2=3
x3=linearform
x3a=x3a
x3b=x3b
p=path
> c=tag
z@#=macro:->begingroup(x(SUFFIX2),y(SUFFIX2))endgroup
dir=macro:<primary>->begingroup.right.rotated(EXPR2)endgroup
> upto=macro:
->step1until
> addto=addto
> dotprod=primarydef'd macro:
(xpart(EXPR0)*xpart(EXPR1)+ypart(EXPR0)*ypart(EXPR1))
```

x3 は後の showdependencies を見ればわかりますが，x3a と x3b の線形結合で表されますので,「linearform」と表示されています。c はまだ全く使われていないので，⟨ タグ ⟩ になっています。⟨ 変数型マクロ ⟩ の z , dir と ⟨ スパーク ⟩ の upto , addto , ⟨ 二項演算

[*12] 多角形の **pen** について表示させると，ほぼ METAFONT と同様の表示になります。
[*13] ⟨ スパーク ⟩ に対して使用すると，次の showtokens と同じ表示になります。

子〉の dotprod の表示が微妙に違うことにもご注意下さい。addto はプリミティブですので，自分自身に等しい (= それ以上展開できない) ことを示しています。

なお，showvariable でマクロの定義を表示させたとき，〈仮引数〉が (EXPR0) や (SUFFIX2) などで表現されていることに気付かれたでしょうか。たとえば，

```
def a(expr s)(suffix t)(text u) = s+t+u; enddef;
```

によって定義されたマクロ a に対して，showvariable a とすると，以下のように表示されます。

```
> a=macro:
(EXPR0)(SUFFIX1)(TEXT2)->(EXPR0)+(SUFFIX1)+(TEXT2);
```

このように，〈引数〉はタイプと出現順につけられた番号で表現されます。とくに〈二項演算子〉の場合，左の〈引数〉が (EXPR0)，右の〈引数〉が (EXPR1) になります。〈変数型マクロ〉の場合，2 番の〈引数〉から始まっていますが，これは (SUFFIX0) が #@ に，(SUFFIX1) が @ に，暗黙のうちに割り振られているためです。〈定義ヘッダ〉に @# を使った場合は，さらに (SUFFIX2) が @# に割り振られます。

次は，showtoken — これは，与えた〈記号トークン〉の現在の意味を表示します。たとえば先の例で

```
showtoken x,p,c,z,dir,upto,addto,dotprod;
```

とすると，以下のように表示されます。

```
> x=variable
> p=variable
> c=tag
> z=variable
> dir=variable
> upto=macro:
->step1until
> addto=addto
> dotprod=primarydef'd macro:
(xpart(EXPR0)*xpart(EXPR1)+ypart(EXPR0)*ypart(EXPR1))
```

c や upto , addto , dotprod に対する表示は showvariable のときと同じですが，〈変数型マクロ〉の z , dir は variable (変数) と表示されることにご注意下さい。

最後は，showdependencies です。これは未知数の間に成り立っている関係式を表示します。たとえば先の例で

```
showtoken x,p,c,z,dir,upto,addto,dotprod;
```

とすると，以下のように表示されます。

```
x3=0.5x3a+0.5x3b
y3a=y3b
```

Exercise 4.4.2　次のように定義されたマクロ a に対して，showvariable a とすると，どのように表示されるでしょうか。

```
vardef a[]@#(suffix s)(text t)primary u = @+@#+s+t+u; enddef;
```

第 5 講

発展的なマクロ (2)

前講に引き続いて，発展的なマクロ定義について取り扱います。本講で扱うのは「ファイルの入出力」と「オプション引数をもつマクロの定義」です。

1 ファイルの入出力

ここではファイルの入出力を扱います。METAPOST では METAFONT に比べてファイルの入出力が強化されているということは，**第3部 第8講** でお話ししたと思いますが，ここでははその実例 — サンプルマクロ calc — を挙げてみたいと思います。calc の「機能」は以下のようなものを想定します。

- マクロ calc は ⟨ 引数 ⟩ に ⟨ 文字列式 ⟩ を要求し，ここにファイル名 (から拡張子を取り去ったもの) を指定します。
- 指定したフィル名に拡張子 .dat をつけたファイルを開き，そこから 1 行ずつ読み込んでいきます。このファイルには数値がコンマで区切って列記されているものとします。
- 読み込んだ数値を (行毎に) 合計し，指定ファイル名に拡張子 .ans をつけたファイルに書き込んでいきます。

要は，表計算ソフトのようなものを想像していただければいいかと思います。

拡張子 .dat のファイルの内容としては，たとえば次のようなものを考えています。

```
%% samplcal.dat
      12, 32, 56, 98, 145
      1.25, 3.15, 9.77, 6.08
      32.2, 13.2, 44.3, 345.6, 0.18 -21.0
```

各行の数値の個数は一定でなくてもかまいません。

さて，ではマクロ calc の定義を考えましょう。

まず ⟨ 定義ヘッダ ⟩ ですが，calc は calc("samplcal"); のような形で使いますので，⟨ 仮引数 ⟩ は ⟨ 式 ⟩ (⟨ 文字列式 ⟩) を受け取れるよう **expr** タイプにします[*1]。したがって，以下のような ⟨ 定義ヘッダ ⟩ となります。

```
      def calc(expr f)
```

次に読み込み部ですが，これは ⟨ 仮引数 ⟩ に与えられたファイルのベースネームに拡張子をつけ，readfrom に渡すだけですので，以下のようになります[*2]。

```
      string s;
```

[*1] **suffix** タイプや **text** タイプにすれば ⟨ 引数 ⟩ の " " が要らなくなるのに，と思われるかも知れませんが，実はうまくいきません。readfrom や write が要求するファイル名は ⟨ 文字列式 ⟩ であり，**suffix** タイプや **text** タイプの ⟨ 引数 ⟩ を str で ⟨ 文字列式 ⟩ に直すとき，数値部分が評価され，たとえば sample01 が sample1 になってしまうからです。

[*2] readfrom は，ファイルを 1 行単位で順次読み込んでいく命令です。読み込んだ内容は **string** 値で返されます。なお，読み込み結果は以下で使いますので，**string** 値変数 s に保存します。

```
s:=readfrom(f&".dat");
```

これをファイル末端まで繰り返します。ファイル末端では readfrom で EOF が返ってきますので，それまで同じ操作を繰り返すことになります。したがって，繰り返し部分は以下のようになります。

```
forever:
  s:=readfrom(f&".dat");
exitif s=EOF;
  (和を求める操作)
endfor
```

次に，和を求める操作ですが，これは scantokens を使って **string** 値変数 s を METAFONT 言語ソースに戻して，for 文の ⟨ 引数 ⟩ 列に渡すだけですので，容易でしょう。その部分を書き上げると以下のようになります。

```
sum:=0;
for n = scantokens s:
  sum:=sum+n;
endfor
```

ただし，このままでは，空行やコメント行に対しても，和 sum を出してしまいます[*3]ので，予めこれらは除外する必要があります[*4]。空行は s="" となりますので，それで検出されます。コメント行は先頭が % で始まる行としておきます。readfrom の返す値は ⟨ 文字列式 ⟩ ですので —"%" で % が無視されないように — たとえ先頭が % で始まる行であっても，s="" とはなりません。したがって，substring を用いて先頭文字を調べることになります。以上から，空行・コメント行の除外は，上記の和の計算を

```
(s<>"") and (substring(0,1) of s<>"%")
```

が真のときのみ行うことで実現できます。

最後に書き出し部ですが，これは計算で得られた和 sum を decimal で文字列化し，⟨ 仮引数 ⟩ , つまり，与えられたファイルのベースネームに拡張子をつけ，write 〜 to に渡すだけですので，以下のようになります。

```
write decimal sum to (f&".ans");
```

以上から完成版のマクロは以下のようになります。

```
def calc(expr f) =
  begingroup save s, sum; string s; numeric sum;
  forever:
    s:=readfrom(f&".dat");
  exitif s=EOF;
    if (s<>"") and (substring(0,1) of s<>"%"):
      sum:=0;
```

[*3] もちろん値は 0 です。

[*4] そういう行をそもそも samplcal.dat に含めないようにする，という選択肢も考えられますが，コメントや空行は，あとでデータを見直すときに非常に便利なものですので，全く許されないのは実用性という観点から考えて問題があります。

```
      for n = scantokens s:
        sum:=sum+n;
      endfor
      write decimal sum to (f&".ans");
    fi
  endfor
  endgroup
enddef;
```

Exercise 4.5.1 **第2講** にサンプルとして挙げた円グラフマクロ

EnGraph(**中心, 半径**)(**点名**)(**数値と色**)

を，以下の形に改造して下さい。

EnGraph(**中心, 半径**)(**点名**)(**ファイル名**)

つまり **数値と色** を別ファイルから読み込む形式とするわけです。ファイルの中味は 1 行目が数値，2 行目が対応する色とします。

```
38, 38, 28, 26, 48, 6, 15
Red,Blue,Yellow,PineGreen,Cyan,Salmon,Purple
```

改造する部分は〈定義ヘッダ〉と，数値・色を配列 v[] , c[] に獲得する部分です。

2 オプション引数つきマクロ

たとえば次のような例を考えましょう。〈巡回パス〉p に対して，

```
FillandDraw(p)(red);
```

とすれば，p の内部を赤 (red) で塗り，枠を黒で引くマクロ FillandDraw があるとします。ここで第 2〈引数〉の red を省略したら塗りつぶしはせずに枠線だけを引くものとします。FillandDraw はどのように定義したらよいでしょうか。

問題となるのは第 2〈引数〉ですので，以下これに話を絞ります。

〈仮引数〉のタイプには **式** タイプ，**suffix** タイプ，**text** タイプがありますが，このうち **式** タイプは今回の目的には不向きです。なぜなら，**式** タイプの〈仮引数〉は〈空〉でない〈式〉を〈引数〉に要求するからです[*5]。〈空〉を〈引数〉にとれるのは **suffix** タイプと **text** タイプです。したがって，〈定義ヘッダ〉は以下のようになります。

色を省略するとき FillandDraw(p)(); とするなら

FillandDraw(expr p)(suffix c) または FillandDraw(expr p)(text c)

色を省略するとき FillandDraw(p); とするなら

FillandDraw(expr p)suffix c または FillandDraw(expr p)text c

最後の 1 つの〈仮引数〉は〈区切りなし変数〉にできることも思い出して下さい[*6]。

[*5] 第 2〈仮引数〉が **式** タイプの〈区切り付き引数〉のとき，FillandDraw(p)(); とすれば，

```
! An expression can't begin with `right delimiter that matches (`.
```

のエラーが出ますし，〈区切りなし引数〉にして FillandDraw(p); としても，

```
! An expression can't begin with `;`.
```

のエラーが出てしまいます。

[*6]〈区切りなし仮引数〉は〈引数〉に区切りの ( ) を付けても付けなくてもいいというものですので，FillandDraw(p)(); と FillandDraw(p); のいずれの形も許容されます。

なお，今回の例では c に〈タグ〉としても通用する red のような形だけでなく，(1,0,0) のような〈式〉も入れることを想定しようと思います。この場合,「実際に使われるまで〈引数〉の評価をされない」**text** タイプがベストの選択になります[*7]。

さて，次に定義の置き換えテキストですが，これについてはどうでしょう。〈仮引数〉c に空が入っているとき，これを文字列化した c は "" と等価になりますので,

```
if not(str c=""): fill p withcolor c; fi
```

でいけそうな気がしますが，実はこれは少々まずいです。というのは str は〈サフィックス〉つまり〈タグ〉と〈添字〉の並びを文字列化する命令ですので，〈スパーク〉には対処できないからです。たとえば，c に (〈カラー式〉として red と等価な) (1,0,0) を入れると,

```
! Missing `)' has been inserted.
```
[*8]

のエラーが出ます。かといって,

```
if color c: fill p withcolor c; fi
```

では c に (1,0,0) を入れたときはエラーが出なくなる代わりに，c が〈空〉のとき，color c: の部分が

```
! A primary expression can't begin with `:'.
```

のエラーを生じます[*9]。ではどうすれば〈空〉か〈カラー〉かを判定できるのでしょうか。正解は以下の通りです。

```
if color begingroup c endgroup: fill p withcolor c; fi
```

begingroup 〜 endgroup の部分はこれ全体で〈1 位式〉を作りますので，たとえ c が〈空〉であっても消滅せずに〈空の 1 位式〉として扱われます。したがって，c が **color** 値のときはこの条件分岐が真になり fill が実行され，c が〈空〉のときはこの条件分岐が (エラーを出さずに) 偽になり fill は実行されません。

以上より，求めるマクロの定義は以下のようになります。

```
def FillandDraw(expr p)text c =
  if color begingroup c endgroup: fill p withcolor c; fi
  draw p;
enddef;
```

Exercise 4.5.2　上記の FillandDraw の定義の別解を考えてみましょう。

Exercise 4.5.3　上記の FillandDraw を枠の色も指定できるように改造してみましょう。

(仕様その 1) FillandDraw(〈巡回パス〉)〈塗りつぶしの色〉,〈枠の色〉;

(仕様その 2) FillandDraw〈枠の色〉(〈巡回パス〉)〈塗りつぶしの色〉;

いずれも，**color** 値には〈タグ〉を使うものとしてかまいません。(仕様その 1) では色を 1 つしか指定しなかったときは塗りつぶしの色と解釈するものとし，(仕様その 2) では塗りつぶしの色と枠の色を個別に省略できるものとします。

[*7] **color** 値に〈タグ〉である red や blue などしか使わないのであれば，**suffix** タイプでもかまいません。

[*8] あるはずのない〈スパーク〉(にぶち当たったので，括弧の閉じ忘れを疑われています。

[*9] 〈空〉が〈空の式〉とは認識されず，完全に「存在しないもの」として — あたかも if color: であるかのように — 扱われていることに注意して下さい。なお，if (color c): としても

```
! A primary expression can't begin with `right delimiter that matches ('.
```

のエラーが出ます。if color (c): でもほぼ同じエラーが出ます。

第 6 講

ふきだし

本講では MEPOTEX の内部動作の一端を解説し，METAPOST と TEX の連携について考えたいと思います。題材としては，**図 4.6.1** のように「ふきだし」の中にコメントを入れることを考えます。ふきだしの形状は，長方形およびだ円を用意することにしましょう。用意するマクロですが，

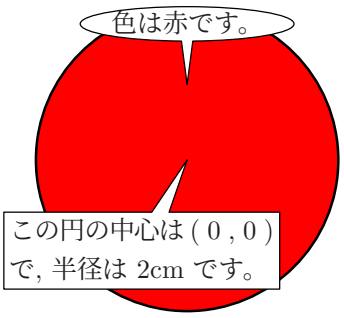


図 4.6.1

ユーザーの使うマクロ：

MEPOTEX の隠蔽モードマクロ (TEX のマクロ)

内部マクロ：

METAPOST のマクロ (上のマクロが自動生成)

の 2 つとしておきます。

MEPOTEX の隠蔽モードマクロ (TEX のマクロ) の方は

`¥Fukidasi{`**ふきだしの始点**`}{`**中心**`}{`**説明文の幅**`}{`**説明文**`}`[*1]

のように使うものとします。`¥Fukidasi` が四角いふきだし，`¥Fukidasi*` が丸いふきだしとします。また，このマクロが自動生成する METAPOST マクロは，

`Fukidasi(`**ふきだしの始点，中心，説明文の幅，高さ，深さ，本体，空白量，タイプ**`)`

の形で使うものとします。

MEPOTEX の隠蔽モードマクロは，METAPOST のソースを隠蔽するというものですが，何も難しいことをしているのではなく，単に内部で METAPOST コマンドを生成し，`¥sendMP` で METAPOST ソースに送っているだけです。`¥sendMP` には 2 つのモードがあります。1 つは引数をそのまま METAPOST ソースに送るモードで，通常ユーザーが使うのはこれです。もう 1 つは引数を (`¥edef` などの ⟨ 置き換えテキスト ⟩ のように) できる限り展開して METAPOST ソースに送るモードで，こちらは

`¥sendMP*{`**送りたい内容**`}`

のように，`*` をつけて使います。隠蔽マクロの定義によく使われるのは，後者の方です。

さて，ではマクロ定義の解説に入りたいと思うのですが，その前に TEX に関する予備知識を列記しておきます[*2]。

- グローバルに定義したいマクロは `¥gdef` で定義します[*3]。
- 引数に改段落が入る可能性があるマクロは `¥long` をつけて定義します。
- `picture` 環境 (に類似した `MPpic` 環境) 内で使うマクロは，よけいな単語間空白を生み出さないよう，定義の ⟨ 置き換えテキスト ⟩ の末尾に `¥ignorespaces` をおきます。

[*1] 「説明文の幅」があまり小さいと，`Overfull` になってふきだしからはみ出しますのでご注意下さい。

[*2] この本の目的は META 言語の解説ですので，深入りは避けて結果のみの列記となります。詳しくは TEX 関連の書籍をご覧下さい

[*3] 今回のマクロは `¥def` でもよかったのですが，`picture` 環境関連のマクロに合わせました。

- * のあるなしで処理を分けるときは，

  ¥@ifstar{ * **ありのとき** }{ * **なしのとき** }

  とします。
- 小さい幅の段落を作るには，¥vbox 内に文章を入れます。このとき，段落の幅は，¥vbox に入ってすぐに，¥hsize を希望の幅にし，¥linewidth を ¥hsize と同じ大きさにすることで指定できます。また，段落先頭の字下げを抑制 するには，¥parindent=0pt とします。
- 上記の段落の幅，高さ，深さは，¥vbox をボックスレジスタに入れれば ¥wd などの命令で取り出せます。たとえば，¥setbox0=¥vbox{〜} とすれば，¥vbox がボックスレジスタ 0 番に入り，¥wd0 , ¥ht0 , ¥dp0 で幅，高さ，深さが取得できます。これらはさらに ¥the をつけると値が文字列化し，別ファイルに書き出せる形式になります。
- ¥edef などの〈置き換えテキスト〉内では徹底的な展開が行われます[*4]が，それを回避する方法もいくつかあります。そのうちの 1 つはトークンレジスタを使う方法で，たとえばトークンレジスタ 0 番（¥toks0 ）に展開したくないものを入れておきます。このトークンレジスタの中味は，¥the¥toks0 とすれば文字列化できるのですが，重要なことは，この取り出された中味は，たとえ ¥edef などの〈置き換えテキスト〉の中にあっても，それ以上展開されることがないということです。
- レジスタを一時的に使うときは，その影響が外部に及ばないように，¥begingroup と ¥endgroup で囲います。

以上のことを知っておけば，隠蔽マクロの方はおおよそ解析できると思います。以下がその定義です。

```
¥gdef¥Fukidasi{¥@ifstar{¥@Fukidasi{"b"}{0pt}}{¥@Fukidasi{"a"}{3pt}}}
¥long¥gdef¥@Fukidasi#1#2#3#4#5#6{%
  ¥begingroup
    ¥setbox0=¥vbox{%
      ¥hsize=#5¥relax¥linewidth=¥hsize¥relax¥parindent=0pt¥relax#6}%
    ¥toks0={¥vbox{%
      ¥hsize=#5¥relax¥linewidth=¥hsize¥relax¥parindent=0pt¥relax#6}}%
    ¥sendMP*{Fukidasi((#3),(#4),¥the¥wd0,¥the¥ht0,¥the¥dp0)%
                                          ("¥the¥toks0",#2)(#1);}
  ¥endgroup
  ¥ignorespaces}
```

¥@Fukidasi の定義の引数のうち，#1 はタイプ，#2 は空白量，#3 はふきだしの始点，#4 はふきだしの中心，#5 は説明文の幅，#6 は説明文本体です。#1 は受け取り側の METAPOST マクロが〈文字列式〉を要求しますので，"a" または "b" を送るようになっています。#2 , #3 , #4 は寸法や点を表す〈式〉（ の文字列 ）をそのまま送ればいいでしょう[*5]。#5 は

[*4] もちろんここでは ¥sendMP* の引数が徹底的に展開されることも指します。

[*5] METAPOST 側のマクロが **text** タイプの〈引数〉を要求するときには，始点・中心などは（ ）で囲って〈1 位式〉にしてから送るべきです。今回の場合は **expr** タイプの〈引数〉なので（ ）をつけなくても問題ないのですが，普段からこういう気遣いをする癖をつけておく方が安全です。

¥vbox 内の段落の幅を指定するために使われています。#6 は大きさを測るためにボックスレジスタに入れ，次いで，METAPOST マクロに展開せずに渡すため，トークンレジスタにも入れています。トークンレジスタの内容は METAPOST に送って，適切な座標などを計算しそのまま TEX 側に戻してもらうつもりですので，よけいな評価が入らないよう " " で囲って **string** 値にしておきます[*6]。

さて，いよいよ本題 — METAPOST マクロです。〈定義ヘッダ〉は

```
def Fukidasi(expr zz,ctr,wd,ht,dp,labbody,labsep,typ)
```

です。zz と ctr はふきだしの始点と中心，typ はふきだしのタイプ，wd , ht , dp は説明文の幅，高さ，深さ，labsep は説明文とふきだしの輪郭との間の空白量です。labbody は説明文の本体で，これは適切な座標を添えてそのまま TEX 側に戻しますので，とくにいじりません。

定義の〈置き換えテキスト〉の内容は以下のようになります。まず，ふきだしの輪郭の基礎となる長方形またはだ円を，**path** 値変数 ptha に取得します。これは，原点を中心とする必要な大きさの長方形やだ円を作っておき，それを ctr だけ平行移動することで得られます。したがって，問題は「必要な大きさの長方形やだ円」を作ることになります。計算によりますと，幅 wd , 高さ ht + dp の長方形あるいはそれに外接するだ円[*7]を得るには，半径 1 の円あるいはそれに内接する正方形を，横方向に $\sqrt{2}\times\frac{\mathtt{wd}}{2}=\sqrt{0.5}\times\mathtt{wd}$ 倍，縦方向に $\sqrt{2}\times\frac{\mathtt{ht}+\mathtt{dp}}{2}=\sqrt{0.5}\times(\mathtt{ht}+\mathtt{dp})$ 倍すればいいので，ptha を取得する部分は以下のようになります。

```
if typ="a":  ptha:=(dir45--dir135--dir225--dir315--cycle)
else      :  ptha:=(up..left..down..right..cycle)
fi                  xscaled(sqrt0.5*wd+labsep)
                    yscaled(sqrt0.5*(ht+dp)+labsep) shifted ctr;
```

次に，これに「とげ」をつけ，ふきだしの輪郭を完成させます。それには「とげ」の部分を少し大きめに作っておき，それと上で作った ptha を buildcycle で結合するのが楽でしょう。まず「大きめのとげ」は，以下のようにして作ります。

```
pthb:=((ctr-zz) rotated 0.5agl -- origin
         -- (ctr-zz) rotated -0.5agl) shifted zz;
```

ここに agl は「とげの角度」で仮に agl:= 15; としておきます。また，ptha も subpath で始点の位置を変えて，buildcycle を使いやすくしておきます[*8]。具体的には，ふきだしの始点と中心を結ぶ線分が,「とげ」の対称軸ですので，これで ptha を切ってそこを新たな始点としておきます。以上の説明に対応するソースは以下の通りです。

```
agl := 15; % ふきだしの「とげ」の角度
t := xpart(ptha intersectiontimes (zz--ctr));
ptha := subpath(t, t + length ptha) of ptha;
```

[*6] " " で囲んだ部分を「評価しない」のはもちろん META 言語の方で，TEX は「評価」してトークンレジスタの中味を取り出します

[*7] 横 : 縦 の比は長方形と同じ比率とします。なお，実際には長方形もだ円も，ふきだしの輪郭と中味の説明文がくっつかないように，幅，高さともに labsep の 2 倍だけ大きめに取ります。

[*8] 詳しくは **第2部 第6講** を参照して下さい。

```
    pthb := ((ctr-zz) rotated 0.5agl -- origin
              -- (ctr-zz) rotated -0.5agl) shifted zz;
    ptha := buildcycle(ptha,pthb);
```

あとはこうして作ったふきだしの輪郭線 ptha を背景色で塗りつぶし，輪郭線を黒で描く (fill ptha withcolor background; draw ptha;) だけです。

さて，最後になりましたが，説明文本体を TEX に戻す部分について説明します。MEPOTEX では，オリジナルモードの METAPOST ソース部分，あるいは，隠蔽モードマクロが自動生成する METAPOST ソースを別ファイルに転送し[*9]，それに対し METAPOST を実行します。この際，METAPOST から TEX に戻したいデータは所定のファイルに書きためておき，最後にまとめて TEX に読み込ませています。この「所定のファイル」の名前は，SAVfile という **string** 値変数に格納する仕様になっていますので，TEX に戻したいデータは

```
    write 戻したいデータ to SAVfile;
```

と記述することになります[*10]。今回の場合,「戻したいデータ」は,

¥mptput(**座標**){labbody} (を 〈 文字列式 〉 にしたもの)

ですので，以下のようになります。

```
    write "¥mptput"& PointCoord(ctr) & "{" & labbody & "}" to SAVfile;
```

ここに PointCoord(〈 ペア式 〉) は，〈 ペア式 〉 の座標を TEX 側が求めている形式・単位に合わせて 〈 文字列式 〉 にするマクロです。

以上をまとめて，METAPOST マクロ Fukidasi の定義は以下のようになります。

```
def Fukidasi(expr zz,ctr,wd,ht,dp,labbody,labsep,typ) =
  begingroup save agl,t, ptha, pthb; numeric agl,t; path ptha, pthb;
    if typ="a":  ptha:=(dir45--dir135--dir225--dir315--cycle)
    else      :  ptha:=(up..left..down..right..cycle)
    fi                  xscaled(sqrt0.5*wd+labsep)
                        yscaled(sqrt0.5*(ht+dp)+labsep) shifted ctr;
    agl:= 15; % ふきだしの「とげ」の角度
    t:=xpart(ptha intersectiontimes (zz--ctr));
    ptha:=subpath(t, t + length ptha) of ptha;
    pthb:=((ctr-zz) rotated0.5agl -- origin
             -- (ctr-zz) rotated-0.5agl) shifted zz;
    ptha:=buildcycle(ptha,pthb);show ptha; show cycle ptha;
    fill ptha withcolor background; draw ptha;
    write "¥mptput"&PointCoord(ctr)& "{" & labbody & "}" to SAVfile;
  endgroup
enddef;
```

Exercise 4.6　**図 4.6.1** のソースを考えて下さい。ただし，マクロは本講のものを使うこととします。

[*9] 前述の通りこれには ¥sendMP を使います。
[*10] 「戻したいデータ」は 〈 文字列式 〉 です。

第 7 講

射影・投影

本講と次講では立方体を描いてみます。一見簡単そうに見えるかも知れませんが，立体を平面上に描くわけですから，それなりにテクニックを要するものです。たとえばまっすぐ立った立方体なら誰でも容易に描けるでしょうが，**図 4.7.1** のように頂点のひとつで立った立方体は，フリーハンドでは意外と描きにくいものです。この際要求されるテクニックは「空間座標の平面への射影・投影」と「任意の向きを向いた立方体の頂点の計算」ですが，本講のメインテーマは前者です[*1]。

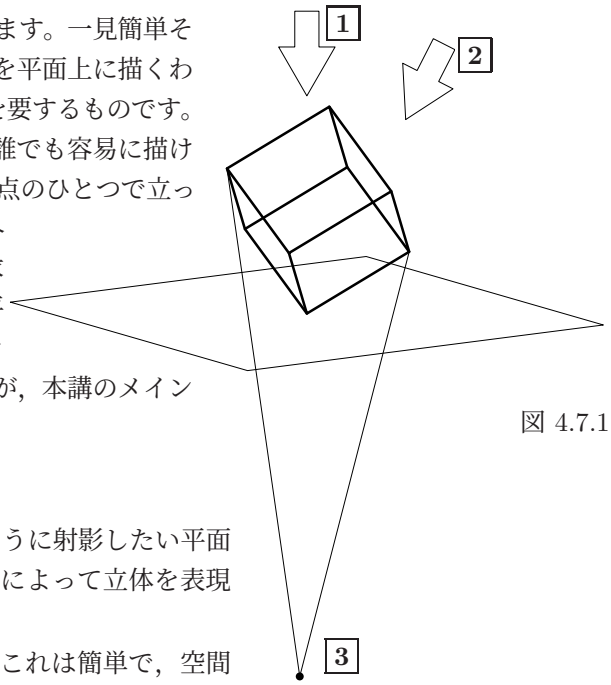


図 4.7.1

1 正射影

正射影とは，**図 4.7.1** の 1 のように射影したい平面に垂直方向の光線を当てて，その影によって立体を表現する方法です。

まずは xy 平面への正射影から。これは簡単で，空間内の点 (p, q, r) に対し，平面上の点 (p, q) を対応させるだけです。しかしこう考えたのでは，一般の平面への正射影に拡張しにくいので，少し一般的な表現に変更してみます。

まず，座標とはそもそも何かといいますと，平面あるいは空間内にとった基準となるベクトル[*2]で点の位置ベクトルを表したときの係数の組です — って，いきなり一般化されたのではわかりにくいと思いますので，最も普通に使われる直交座標について，具体的に説明します。直交座標では基底として通常 $\vec{a} = (1, 0, 0)$, $\vec{b} = (0, 1, 0)$, $\vec{c} = (0, 0, 1)$ をとります。そしてこれを用いてベクトル $\vec{v} = (p, q, r)$ を

$$\vec{v} = p\vec{a} + q\vec{b} + r\vec{c} \cdots\cdots (*)$$

と表したときの係数の組 (p, q, r) が座標というわけです。上記の $\vec{a}$, $\vec{b}$, $\vec{c}$ を基底にとったのでは，当たり前すぎてありがたみが感じられないと思いますが，この考え方は $\vec{a}$, $\vec{b}$, $\vec{c}$ が $(1, 0, 0), (0, 1, 0), (0, 0, 1)$ でなくても使えるというところに利点があります。具体的に言いますと，

$$\vec{a'} = \left(\frac{9}{25}, \frac{12}{25}, \frac{4}{5}\right),\ \vec{b'} = \left(\frac{12}{25}, \frac{16}{25}, -\frac{3}{5}\right),\ \vec{c'} = \left(-\frac{4}{5}, \frac{3}{5}, 0\right)$$

[*1] 後者については **第8講** で扱います。
[*2] 基底といいます

は直交する長さ 1 のベクトルなのですが，これにより，

$$\overrightarrow{v} = (\,p\,,\,q\,,\,r\,) = p'\overrightarrow{a'} + q'\overrightarrow{b'} + r'\overrightarrow{c'} \cdots\cdots (**)$$

と表せたとすると，$\overrightarrow{a'}$ ，$\overrightarrow{b'}$ ，$\overrightarrow{c'}$ の向きを新たな X ，Y ，Z 軸に取ったときの座標が $(\,p'\,,\,q'\,,\,r'\,)$ であるということです[*3]。

さて，このように座標を解釈したとき，正射影とは空間内の点 (ベクトル) $\overrightarrow{v}$ を基底 $\overrightarrow{a}$ ，$\overrightarrow{b}$ ，$\overrightarrow{c}$ を用いて $(*)$ のように表したとき，$\overrightarrow{c}$ の成分を無視して $(\,p\,,\,q\,)$ を座標として採用するということに他なりません。こう考えれば一般の平面への正射影は簡単かと思います。

まず，射影したい平面上に X 軸と Y 軸をとり，この軸方向の長さ 1 のベクトルを $\overrightarrow{a}$ ，$\overrightarrow{b}$ とします。また，X 軸と Y 軸の交点 — つまり平面上の原点 — の，空間内での座標を $\overrightarrow{o}$ とし，さらに平面の単位法線ベクトル (平面に垂直な長さ 1 のベクトル) を $\overrightarrow{c}$ としておきます。$\overrightarrow{a}$ ，$\overrightarrow{b}$ ，$\overrightarrow{c}$ ，$\overrightarrow{o}$ は予めわかっているものとします[*4]。このとき，空間内の任意の点 (ベクトル) $\overrightarrow{v}$ の射影平面上の座標 (X ，Y) は，方程式

$$\overrightarrow{v} = X\overrightarrow{a} + Y\overrightarrow{b} + r\overrightarrow{c} + \overrightarrow{o}$$

の解として求まります[*5]。METAFONT 言語は連立 1 次方程式を解くことができますので，正射影を計算するマクロはこの方程式をそのまま記述するだけで作れます。つまり 3 次元ベクトルを表す **color** 値変数 v を ⟨ 仮引数 ⟩ として，射影平面上での座標を **pair** 値として返すマクロ prja(v) は，以下のようになります[*6]。

```
vardef prja(expr v) =
  save X,Y;
    v = X*a + Y*b + whatever*c + o;
    (X,Y)
enddef;
```

これを用いて描いた図が **図 4.7.2** です。

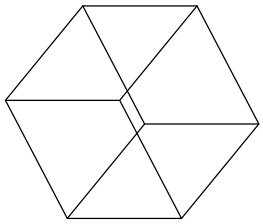


図 4.7.2

2 斜投影法，等角投影法

斜投影法は，**図 4.7.1** で 2 のように射影したい平面に斜め方向の光線を当てて，その影によって立体を表現する方法です。これを実現するためのマクロは正射影のときと全く同じで，違うところといえば，正射影の場合，基底のベクトル $\overrightarrow{a}$ ，$\overrightarrow{b}$ ，$\overrightarrow{c}$ が互いに直交するものを採用したのに対し，斜投影法では $\overrightarrow{c}$ の代わりに $\overrightarrow{a}$ ，$\overrightarrow{b}$ と直交しないベクトル $\overrightarrow{c'}$ を採用するという点です。この考え方で描いた図形が **図 4.7.3** です。

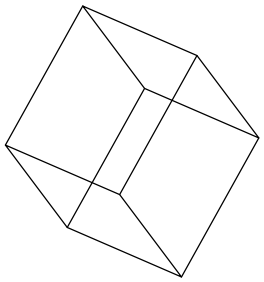


図 4.7.3

[*3] ちなみに，方程式 $(**)$ を解くと，

$$p' = \frac{9}{25}p + \frac{12}{25}q + \frac{4}{5}r\ ,\ \ q' = \frac{12}{25}p + \frac{16}{25}q - \frac{3}{5}r\ ,\ \ r' = -\frac{4}{5}p + \frac{3}{5}q$$

となります。

[*4] $\overrightarrow{a}$ ，$\overrightarrow{b}$ ，$\overrightarrow{c}$ を個別のケース毎に決定するのは案外面倒ですが，マクロ化は容易です。これは結局は「任意の向きを向いた立方体の頂点の計算」と同じになりますので，**第8講** で扱います。

[*5] x ，y ，z 成分それぞれが一致しますので，1 次方程式 3 つに分解され，未知数 X ，Y ，r がちょうど求まります。

[*6] これは METAPOST の proj マクロとほぼ同じです。

さて，一応斜投影法を実現する方法を挙げたのですが，そもそも何故 (影が歪むのに) 斜め方向の光線を考えるのでしょう。— それはまっすぐ立った立方体を「気軽に」表現できるからです。つまり，本来斜投影法を使うべき状況は，最初の図のように任意の向きを向いた立方体ではなく，まっすぐ立った立方体をその面の 1 つを含む平面に射影するという単純な状況なわけです[*7]。「気軽に」の意味を説明するため，別の角度から斜投影法を説明してみます。いま，$\overrightarrow{a}$, $\overrightarrow{b}$, $\overrightarrow{c}$ を互いに直交する長さ 1 のベクトル，$\overrightarrow{c'}$ を射影方向とし，$\overrightarrow{a}$, $\overrightarrow{b}$ は射影面上に載っているものとします。$\overrightarrow{a}$, $\overrightarrow{b}$ の射影は自分自身と一致するわけですが，区別のため $\overrightarrow{a_p}$, $\overrightarrow{b_p}$ と表すことにします[*8]。同様に $\overrightarrow{c}$ の射影を $\overrightarrow{c_p}$ と表すことにします。もちろん $\overrightarrow{c'}$ の射影は $\overrightarrow{0}$ です。

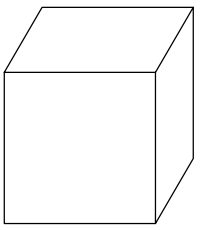


図 4.7.4

$$\overrightarrow{c'} = \alpha\overrightarrow{a} + \beta\overrightarrow{b} + \gamma\overrightarrow{c}$$

で $\overrightarrow{c'}$ が表せるとき，

$$\overrightarrow{0} = \alpha\overrightarrow{a_p} + \beta\overrightarrow{b_p} + \gamma\overrightarrow{c_p} \iff \overrightarrow{c_p} = -\frac{\alpha}{\gamma}\overrightarrow{a_p} - \frac{\beta}{\gamma}\overrightarrow{b_p}$$

が成立します[*9]。xy 平面上にまっすぐ立った立方体を表現するには，$\overrightarrow{a} = (\,1\,,\,0\,,\,0\,)$ に対して $\overrightarrow{a_p} = (\,1\,,\,0\,)$, $\overrightarrow{b} = (\,0\,,\,1\,,\,0\,)$ に対して $\overrightarrow{b_p} = (\,0\,,\,1\,)$ を用いればよく，さらに $\overrightarrow{c} = (\,0\,,\,0\,,\,1\,)$ とすれば，点 $(\,p\,,\,q\,,\,r\,) = p\overrightarrow{a} + q\overrightarrow{b} + r\overrightarrow{c}$ を射影した点が $p\overrightarrow{a_p} + q\overrightarrow{b_p} + r\overrightarrow{c_p}$ で得られることになります。つまり，空間内での座標が，そのまま射影平面上のベクトル $\overrightarrow{a_p}$, $\overrightarrow{b_p}$, $\overrightarrow{c_p}$ の係数となるわけです。**図 4.7.4** は $\overrightarrow{c_p}$ の一例として $\frac{\alpha}{\gamma} = 0.25$, $\frac{\beta}{\gamma} = 0.43$ として $\overrightarrow{c_p} = (\,-0.25\,,\,-0.43\,)$ を用いて描いた立方体の射影図です。

関連する事項として等角投影法も扱っておきましょう。これは空間内の座標 (成分) $(\,p\,,\,q\,,\,r\,)$ の点 (ベクトル) を，射影平面内のベクトル $p\overrightarrow{a_p} + q\overrightarrow{b_p} + r\overrightarrow{c_p}$ で表すという点では斜投影法と同じなのですが，$\overrightarrow{a_p}$, $\overrightarrow{b_p}$, $\overrightarrow{c_p}$ を等しい角 ($120°$) をなす長さ 1 のベクトルで表現するというところが異なります。**図 4.7.5** は先程と同じく xy 平面上にまっすぐ立った立方体 — 基底ベクトルが $\overrightarrow{a} = (\,1\,,\,0\,,\,0\,)$, $\overrightarrow{b} = (\,0\,,\,1\,,\,0\,)$, $\overrightarrow{c} = (\,0\,,\,0\,,\,1\,)$ である立方体を等角投影法で描いたものです。$\overrightarrow{a}$, $\overrightarrow{b}$, $\overrightarrow{c}$ をそれぞれ

$$\overrightarrow{a_p} = (\,\cos(-30)°\,,\ \sin(-30)°\,)$$
$$\overrightarrow{b_p} = (\,\cos 90°\,,\ \sin 90°\,)$$
$$\overrightarrow{c_p} = (\,\cos 210°\,,\ \sin 210°\,)$$

に射影しているのですが，これは別の見方をすれば，もとの立方体を $(\,1\,,\,1\,,\,1\,)$ 方向に正射影したものとも言えます。

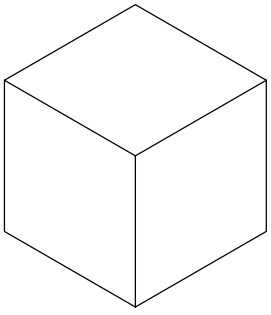


図 4.7.5

Exercise 4.7.1 **図 4.7.4** , **図 4.7.5** のソースを考えて下さい。

[*7] この状況で正射影を使うと，立方体の影は完全に正方形になってしまい，立体っぽく見えません。

[*8] 図形的には一致していますが，$\overrightarrow{a}$, $\overrightarrow{b}$ は空間ベクトル，$\overrightarrow{a_p}$, $\overrightarrow{b_p}$ は平面ベクトルですので，以下のように成分の次元が異なります。

[*9] 射影は線型変換だからです。

3 遠近法

これは人間の視界に近い表現で，近くのものは大きく，遠くのものは小さく描くことにより，立体感を出すものです。なぜ，人間の視界がこのようになっているかといいますと，「眼」という 1 点に集まってくる光が視界を構成し，同じ長さの線分でも，近いものほど視野角に占める割合が大きくなるためです。これを模式化しますと，**図 4.7.1** の 3 のように,「眼」に相当する 1 点から立体の各点を結ぶ直線と，投影したい平面との交点を投影点とすることになります[*10]。

投影したい平面上の X 軸，Y 軸方向の長さ 1 のベクトルを $\overrightarrow{a}$, $\overrightarrow{b}$ とし，X 軸と Y 軸の交点 (投影面上の原点) の空間内での座標を $\overrightarrow{o}$,「眼」の座標を $\overrightarrow{c}$ として数式化しますと，空間内の任意の点 (ベクトル) $\overrightarrow{v}$ の投影面上の座標 (X , Y) は，方程式

$$(1-t)\overrightarrow{c}+t\overrightarrow{v}=X\overrightarrow{a}+Y\overrightarrow{b}+\overrightarrow{o}$$

を解くことで得られます[*11]。左辺は $\overrightarrow{c}$ と $\overrightarrow{v}$ を結ぶ直線上の点を，右辺は投影面上の点を表しますので，これらが等しくなる点，つまり交点が求まるという寸法です。これをマクロ化したものが以下の prjb です。

```
vardef prjb(expr v) =
  save X,Y;
    whatever[c,v] = X*a + Y*b + o;
    (X,Y)
enddef;
```

これを用いて描いた図が **図 4.7.6** です[*12]。

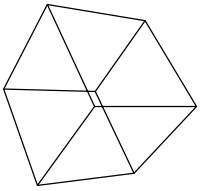


図 4.7.6

Exercise 4.7.2　(1 , 0 , 0) , (0 , 1 , 0) , (0 , 0 , 1) , (1 , 1 , 1) の 4 点は正四面体をなします。これを遠近法で xy 平面に投影して下さい。ただし,「眼」の位置は (0 , 0 , -4) とします。

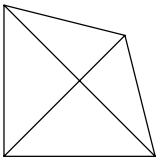


図 4.7.7

[*10] 他にも何種類か「遠近感」を出す手法が考えられており，それらの総称が「遠近法」とよばれるものです。

[*11] 未知数は t , X , Y です。

[*12] 「眼」が立体に近すぎますと,「眼」に近い辺と遠い辺の差が大きくなりすぎていびつに見え，かえって嘘っぽくなりますので注意して下さい。

第 8 講

直交化・空間における回転

本講では前講で触れられなかった話題 — 「任意の向きを向いた立方体の頂点の計算」を扱ってみたいと思います。立方体 (や直方体，一般に平行 6 面体) は，1 つの頂点を始点とする 3 つのベクトルで表現されますので，このベクトルさえ求めれば，立方体の頂点はすべて求められます[*1]。もちろん，立方体の場合この 3 つのベクトルは長さが等しく直交していますので,「任意の向きを向いた立方体の頂点」を計算することは「空間内に長さが 1 で直交する 3 つのベクトルをとる」ことに他なりません。ここでは「**直交化**」と「**空間における回転** 」の 2 通りのアプローチに沿って，頂点 (ベクトル) の計算アルゴリズムを考えることにします。

1 直交化

直交化という手法を用いて，空間内に，長さが 1 で互いに垂直な 3 つのベクトル $\vec{a}$, $\vec{b}$, $\vec{c}$ をとることを考えます。$\vec{a}$ の向き，および，$\vec{b}$ のおおよその向きはわかっているものとします。もう少し具体的に言えば，$\vec{a}$ と同じ向きのベクトル $\vec{a'}$ と，$\vec{a'}$ とは異なる向きを向いたベクトル $\vec{b'}$ が与えられ，$\vec{b}$ は $\vec{a'}$ と $\vec{b'}$ を含む平面内にとるものとします。

まず，$\vec{a}$ ですが，これは $\vec{a'}$ を自身の長さで割ればいいので問題はありません。次に $\vec{b}$ を求めます。$\vec{b}$ の向きは **図 4.8.1** の $\overrightarrow{\mathrm{OB}}$ の向きですので，

$\vec{b'} = \overrightarrow{\mathrm{OB}} + \overrightarrow{\mathrm{OH}} \iff \overrightarrow{\mathrm{OB}} = \vec{b'} - \overrightarrow{\mathrm{OH}}$

であることから，正射影ベクトル $\overrightarrow{\mathrm{OH}}$ を求めれば $\overrightarrow{\mathrm{OB}}$ も求まることがわかると思います。

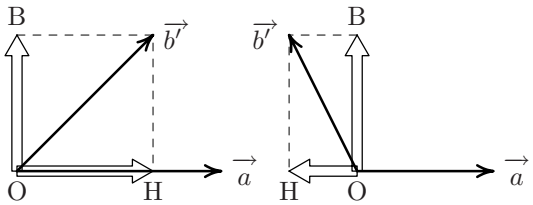


図 4.8.1

$\overrightarrow{\mathrm{OH}}$ は $\vec{a}$ と平行ですので，$\overrightarrow{\mathrm{OH}} = k\vec{a}$ とおけます。ここで，

内積 $\vec{a}\cdot\vec{b'} = \vec{a}\cdot\left(\overrightarrow{\mathrm{OH}} + \overrightarrow{\mathrm{OB}}\right) = \vec{a}\cdot\left(k\vec{a}\right) + \vec{a}\cdot\overrightarrow{\mathrm{OB}} = k\left|\vec{a}\right|^2 + 0$

であるので，$\left|\vec{a}\right| = 1$ から $k = \vec{a}\cdot\vec{b'}$ であることがわかります。このことから，

$\overrightarrow{\mathrm{OB}} = \vec{b'} - \left(\vec{a}\cdot\vec{b'}\right)\vec{a}$

となり，こうして求めた $\overrightarrow{\mathrm{OB}}$ を自身の長さで割って $\vec{b}$ を得ることができます。

次に $\vec{c}$ ですが，これは空間内で $\vec{a}$ と $\vec{b}$ に垂直なベクトルですので，外積を用いて $\vec{c} = \vec{a}\times\vec{b}$ で求められます。

以上を META 言語[*2]で表現しますと，次のようになります[*3]。なお，$\vec{a'}$, $\vec{b'}$ を表す **pair** 値を `_a` , `_b` とします。

[*1] 頂点の 1 つの位置ベクトルを $\vec{o}$, この頂点を始点とする 3 つのベクトルを $\vec{a}$, $\vec{b}$, $\vec{c}$ とすると，立方体の 8 つの頂点の位置ベクトルは $\vec{o}$, $\vec{o}+\vec{a}$, $\vec{o}+\vec{b}$, $\vec{o}+\vec{c}$, $\vec{o}+\vec{a}+\vec{b}$, $\vec{o}+\vec{b}+\vec{c}$, $\vec{o}+\vec{c}+\vec{a}$, $\vec{o}+\vec{a}+\vec{b}+\vec{c}$ となります。

[*2] **color** 値 〈 変数 〉 を使うので，METAPOST または MEPOTEX を使うことになります。

[*3] `naiseki` , `gaiseki` , `nagasa` は MEPOTEX で定義されているものと同じです。

```
% 内積の定義
  primarydef _w naiseki _z =
    (redpart _w * redpart _z + greenpart _w * greenpart _z
                             + bluepart _w * bluepart _z)
  enddef;
% 外積の定義
  primarydef _w gaiseki _z =
    (greenpart _w * bluepart  _z - bluepart  _w * greenpart _z ,
     bluepart  _w * redpart   _z - redpart   _w * bluepart  _z ,
     redpart   _w * greenpart _z - greenpart _w * redpart   _z)
  enddef;
% 3 次元ベクトルの長さを求めるマクロ
  vardef nagasa primary _z =
    (redpart _z ++ greenpart _z ++ bluepart _z)
  enddef;
% 以上のマクロを用いて a , b , c は以下のように求められます。
  a:=(_a)/nagasa(_a);
  b:=_b-(_b naiseki a)*a; b:=b/nagasa(b);
  c:=a gaiseki b;
```

Exercise 4.8.1　ベクトル $\overrightarrow{a'} = (1, 2, 2)$, $\overrightarrow{b'} = (4, 1, 3)$ をもとにして，上記の直交化を行うと，得られるベクトル $\overrightarrow{a}$, $\overrightarrow{b}$, $\overrightarrow{c}$ はいくらになるでしょうか。結果を予想した上で，METAPOST に計算させて確かめて下さい。

2 空間内の回転

空間内に互いに垂直な 3 つのベクトルをとるには，もともと互いに垂直な 3 つのベクトル — たとえば $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$ — をとり，これを任意の軸のまわりに回転して必要なベクトルを得るという手もあります[*4]。

回転を含めた「1 次変換」は空間の場合 3×3 行列すなわち 9 つのパラメータで表現されるのですが，回転に限定すれば，回転軸の向きを表す緯度 θ , 経度 φ (**図 4.8.2** 参照) および回転角 ω という 3 つのパラメータだけで表現できます。今回は

c Rotated（**緯度, 経度, 回転角**）

で **color** 値 c を,「緯度」「経度」で指定される回転軸のまわりに「回転角」だけ回転するマクロ Rotated を作ってみましょう[*5]。

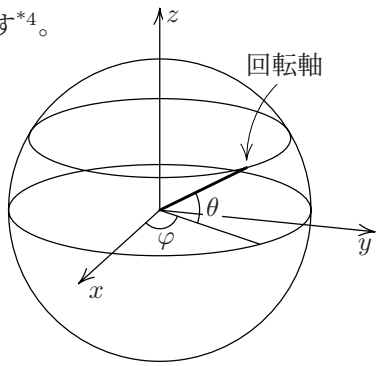


図 4.8.2

[*4] もっとも，回転軸の向き，回転角を予め決めておく必要がありますので，これらが指定しにくいときには不利な方法といえます。

[*5] (緯度, 経度, 回転角) の部分も 3 パラメータですので，**color** 値とし，
〈 カラー 2 位式 〉Rotated〈 カラー 1 位式 〉
という形式で使うものとします。この優先順位は，**pair** 値に対する〈 変形演算子 〉と同じレベルです。

まず予備知識ですが，空間内の任意の軸のまわりの回転は，2 本の軸のまわりの回転の合成として表現できます。そこで，まず座標軸まわりの回転マクロを定義しておきましょう[*6]。まずは x 軸のまわりの回転 — これは yz 平面内の回転ですので，**color** 値の greenpart(y 成分に相当) と bluepart(z 成分に相当) を抜き出して **pair** 値を作り，これに METAFONT 言語のプリミティブである rotated を作用させれば実現できます。y 軸 , z 軸まわりの回転についても同様です。これらを実現するマクロを順に XRotated , YRotated , ZRotated とすれば以下のようになります。

```
primarydef _c XRotated _d =
  begingroup save _y,_z;
    (_y,_z)=(greenpart _c, bluepart _c) rotated _d;
    (redpart _c, _y, _z)
  endgroup
enddef;
primarydef _c YRotated _d =
  begingroup save _z,_x;
    (_z,_x)=(bluepart _c, redpart _c) rotated _d;
    (_x, greenpart _c, _z)
  endgroup
enddef;
primarydef _c ZRotated _d =
  begingroup save _x,_y;
    (_x,_y)=(redpart _c, greenpart _c) rotated _d;
    (_x, _y, bluepart _c)
  endgroup
enddef;
```

一般の軸のまわりの回転は次のように実現されます。回転軸の緯度と経度を順に θ , φ とし，回転角を ω とします。まず回転軸を x 軸に一致するまで回転します。これは z 軸まわりに $-\varphi$ 回転し，そのあと y 軸まわりに θ 回転すれば実現できます[*7]。その後 x 軸まわりに ω 回転し，回転軸を元に戻します — つまり，先程とは逆に y 軸まわりに $-\theta$ 回転し，そのあと z 軸まわりに φ 回転します。以上をマクロ化すると次のようになります。

```
primarydef _c Rotated _d =
  begingroup save _cc; color _cc;
    _cc:=_c ZRotated (-greenpart _d) YRotated (redpart _d)
            XRotated (bluepart _d)
            YRotated (-redpart _d) ZRotated (greenpart _d);
    _cc
  endgroup
```

[*6] x 軸 , y 軸 , z 軸のうち 2 本について定義すればすべての回転を表し得るのですが，実際には 3 軸ともあった方が便利ですので，すべて定義しておきます。

[*7] **図 4.8.2** を参照して下さい。

```
enddef;
```

Exercise 4.8.2　次のソースの意味を考え結果を予想した上で，METAPOST で実行して確認してみて下さい[*8]。

```
color c[]; c0:=(4,0,0); c1:=(1,1,1);
c2:=c0 XRotated 30; show c2;
c2:=c0 YRotated 30; show c2;
c2:=c0 ZRotated 30; show c2;
c2:=c0 YRotated -30 ZRotated 60; show c2;
c2:=c0 Rotated (latitude c1,longitude c1,120); show c2;
```

Exercise 4.8.3　本文中の Rotated は原点を通る軸のまわりの回転でしたが，これをこれまでに作ってきたマクロと組み合わせれば，一般の軸のまわりの回転に拡張できます。実際に試してみましょう。a , b , c , d は **color** 値変数とします。空間内の 2 点 a , b を通る直線を回転軸，回転方向は a を奥，b を手前においたとき反時計回り（正の回転方向）に ω とし，この回転によって点 c が移る点を d に設定する方程式を立ててみて下さい。また，a:=(1,1,1) , b:=(3,2,2) , c:=(3,4,6) , $\omega = 60^\circ$ として，実際に d を METAPOST に計算させてみて下さい。なお，これまでに定義したマクロは自由に使って下さって結構です。

[*8] latitude と longitude は **第3部 第3講** で出てきた，緯度・経度を求めるマクロですので，コピーして使って下さい。

第 9 講

空間内における affine 変換

今回は，空間における affine 変換を扱います。空間における affine 変換というのは，

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix} \cdots\cdots (*)$$

という式で表されるもので，平面における affine 変換と同様，回転・対称移動，平行移動，拡大縮小などを包含した広い変換を表します。META 言語では，平面における affine 変換を **transform** 値で表せます。実際 `z` を平面上の点を表す **pair** 値，`t` を **transform** 値とすると，

```
z transformed t;
```

で `z` を `t` で affine 変換したものが得られます。本講では，これと同様のことを空間でも実現できないか追求してみます。

まず目標ですが，これは

⟨ カラー 2 位式 ⟩ `Transformed` ⟨affine 変換⟩[*1]

という形式の ⟨ 二項演算子 ⟩ `Transformed` を定義することです。ところがここに問題があります。⟨ 二項演算子 ⟩ の ⟨ 引数 ⟩ は **expr** (式) タイプに固定されます。一方 METAPOST の ⟨ 式 ⟩ のタイプの中で，空間の affine 変換 — 上記のように 12 個もの成分をもちます — を保持できるタイプは **path** 値しかありません。しかし，**path** 値は既知数にしか使えません。これでは META 言語の長所 — 方程式による未知数の決定 — が生かせません。

ところで，affine 変換の成分が未知数のまま方程式で扱われるのは，通常その affine 変換の成分を決定する段階です。一旦成分が決定してしまってからは，そのような形式で扱われることは少ないでしょう。

そこで，ここでは次のような方法で空間の affine 変換を実現しようと思います。

- 空間の affine 変換を仮に `t` で表すことにします。この `t` はそれ自体はとくに値をもたない ⟨ タグ ⟩ で，成分を保持するのは 4 つの **color** 値 `t._cx` , `t._cy` , `t._cz` , `t._co` と 1 つの **path** 値 `t.p` とします。これらの確保は，マクロ `newTransform` で行うものとします。このマクロの ⟨ 引数 ⟩ は ⟨ タグ ⟩ (上の例では `t`) です。
- 式 $(*)$ の表示に対して，`t._cx` は (a_1 , a_2 , a_3) を，`t._cy` は (b_1 , b_2 , b_3) を，`t._cz` は (c_1 , c_2 , c_3) を，`t._co` は (d_1 , d_2 , d_3) を表すものとします。また，`t.p` は

  (a_1,a_2)--(a_3,d_1)--(b_1,b_2)--(b_3,d_2)--(c_1,c_2)--(c_3,d_3)

  を表すものとします。

[*1] 平面の affine 変換に対応する **transform** 値は，**pair** 値だけでなく，**pair** 値から構成される **path** 値，**pen** 値，**picture** 値にも適用できますが，(**color** 値は平面に投影することなしに **path** 値などに変換できませんので，) 今回ここで考える ⟨ 二項演算子 ⟩ は左側に **color** 値がくることのみを想定していいでしょう。

○ 未知の成分を含む段階では t._cx , t._cy , t._cz , t._co を使い，すべての成分が既知になってからは t.p を使うものとします。

○ ただし，ユーザーは t._cx , t._cy , t._cz , t._co を直接意識しないで使えるよう，〈変数型マクロ〉t.s を予め定義しておきます[*2]。これは〈カラー 1 位式〉を〈引数〉に取り，〈引数〉に対して (∗) 式の変換を施した結果を〈カラー 1 位式〉で返すものとします。t._cx , t._cy , t._cz , t._co が未知であっても〈引数〉が既知ならこの計算は可能ですので，変換が未知成分を含むとき — 成分を決定する段階 — では，もっぱらこの形式を使うことになります。

○ t._cx , t._cy , t._cz , t._co から t.p へ成分をコピーするマクロは

```
realizeTransform
```

で，〈引数〉は〈タグ〉とします[*3]。

○ 当初の目標であるマクロ Transformed は，

〈カラー 2 位式〉Transformed〈パス 1 位式〉

の形式で使うものとします。もちろん，ここでいう〈パス 1 位式〉は，上述の例では t.p です。

さて，マクロ定義の解説に入る前に，使用例を見てみましょう。上の説明では何か大変そうな印象を受けるかも知れませんが，面倒なことはほとんど隠蔽されているため，ユーザーはとくに意識せずに使えるはずです。

```
newTransform(ttt);%        タグ名を ttt に
(1,2,3) = ttt.s (1,0,0);% (1,0,0) の像は (1,2,3)
(4,5,6) = ttt.s (0,1,0);% (0,1,0) の像は (4,5,6)
(7,8,9) = ttt.s (0,0,1);% (0,0,1) の像は (7,8,9)
(0,0,0) = ttt.s (0,0,0);% (0,0,0) の像は (0,0,0)
realizeTransform(ttt);%    以上で成分が決定できたので path にコピー

color a;
a:=(3,1,4) Transformed ttt.p; % (3,1,4) の像を a に
show a;
```

Exercise 4.9.1　上の例で show a; としたときの結果はどうなるでしょうか。

では，マクロ定義の解説に入りましょう。なお，以下の定義には MEPOTEX のマクロ naiseki（空間ベクトルの内積）が使われています。

まず newTransform から。これは，変数の宣言と，〈引数〉に与えた〈タグ〉に .s をつけた〈変数型マクロ〉を定義します。

```
def newTransform(suffix a) =
  color a._cx, a._cy, a._cz, a._co; path a.p;
```

[*2] これも newTransform で行うものとします。

[*3] コピーする前に各成分が既知になっているかチェックする必要があります。このチェックを t.s の定義に盛り込み，値のコピーを自動化することも可能ですが，そうすると t.s を使う度にこのチェックが入り，かえって効率が悪くなりますので，ユーザーの方で指定するようにしました。

```
    vardef a.s primary c =
      (a._cx naiseki c, a._cy naiseki c, a._cz naiseki c) + a._co
    enddef;
  enddef;
```

宣言部分は

```
    color a._cx, a._cy, a._cz, a._co; path a.p;
```

です，⟨ 仮引数 ⟩ の a が ⟨ 引数 ⟩ の ⟨ タグ ⟩ に置き換えられて宣言されることに注意して下さい。これは ⟨ 仮引数 ⟩ が **suffix** タイプ (や **text** タイプ) だからできることです。

「.s をつけた ⟨ 変数型マクロ ⟩」に関しては，

$(\ a_1\ ,\ a_2\ ,\ a_3\ ) = \overrightarrow{a}\ ,\ (\ b_1\ ,\ b_2\ ,\ b_3\ ) = \overrightarrow{b}\ ,\ (\ c_1\ ,\ c_2\ ,\ c_3\ ) = \overrightarrow{c}\ ,$
$(\ d_1\ ,\ d_2\ ,\ d_3\ ) = \overrightarrow{d}\ ,\ (\ x\ ,\ y\ ,\ z\ ) = \overrightarrow{x}$

とおくとき，

$$(*) \iff (\ x'\ ,\ y'\ ,\ z'\ ) = (\ \overrightarrow{a}\cdot\overrightarrow{x}\ ,\ \overrightarrow{b}\cdot\overrightarrow{x}\ ,\ \overrightarrow{c}\cdot\overrightarrow{x}\ ) + \overrightarrow{d} \qquad (\cdot \text{は内積})$$

となることを考え合わせれば定義の意味が理解できると思います。

次は，realizeTransform です。これは変換のすべての成分が既知かどうかを確認し，そのときに限り値を **path** 値の成分にコピーしています。

```
  def realizeTransform(suffix a) =
    if known a._cx and known a._cy and known a._cz and known a._co:
      a.p :=(redpart  a._cx, greenpart a._cx)--
            (bluepart a._cx, redpart   a._co)--
            (redpart  a._cy, greenpart a._cy)--
            (bluepart a._cy, greenpart a._co)--
            (redpart  a._cz, greenpart a._cz)--
            (bluepart a._cz, bluepart  a._co);
    else:
      errmessage str a & " is not a known transform";
    fi
  enddef;
```

最後に，Transformed です。これは (∗) に従って **color** 値を計算するだけなのですが，**path** 値から成分を取り出す部分は煩雑なので，一旦 **color** 値の一時変数 cx , cy , cz , co に値をコピーしてから計算しています。

```
  primarydef _c Transformed _p =
    begingroup save cx,cy,cz,co; color cx,cy,cz,co;
      cx:=(xpart point0of _p,ypart point0of _p,xpart point1of _p);
      cy:=(xpart point2of _p,ypart point2of _p,xpart point3of _p);
      cz:=(xpart point4of _p,ypart point4of _p,xpart point5of _p);
      co:=(ypart point1of _p,ypart point3of _p,ypart point5of _p);
      (cx naiseki _c, cy naiseki _c, cz naiseki _c) + co
    endgroup
  enddef;
```

Exercise 4.9.2　平面上では 2 点 z , w を通る直線に関する対称移動を表すマクロ

```
reflectedabout(z,w)
```

があります。これと同じように，空間内で 3 点 a , b , c を通る平面に関する対称移動を表すマクロ

```
Reflectedabout(a,b,c)
```

を作ってみましょう。なお，これまでに定義したマクロは自由に使って下さって結構です。

《ヒント》〈置き換えテキスト〉は本節で定義した Transformed を用いて，

```
Transformed〈パス式〉
```

のようになります。〈パス式〉の部分は **path** 値の一時変数を計算して入れます。〈パス式〉は newTransform , realizeTransform を用いれば計算できると思います。この計算部分は最終的にソースから隠蔽しなければならないのですが，これに関してはグルーピング（**第3部 第9講 8項** を参照）が使えます。

Exercise 4.9.3　平面上では点 z のまわり t 度の回転を表すマクロ

```
rotatedaround(z,t)
```

があります。これと同じように，空間内で 2 点 a , b を通る直線のまわり t 度の回転を表すマクロ

```
Rotatedaround(a,b,t)
```

を作ってみましょう。なお，これまでに定義したマクロは自由に使って下さって結構です。

《ヒント》 基本的に前問と同じ考え方で定義できます。〈パス式〉を求めるために必要な空間内の点の計算法は，前講の演習問題を参照して下さい。

第 10 講

結晶格子

本講のテーマは NaCl 型結晶格子 (**図 4.10.1**) です。結晶格子の中では結構基本的なものですので，高校で化学を履修した人なら見覚えはあると思います。大きな淡色の球が Cl^- イオン，小さな濃色の球が Na^+ イオンです。現実には同種イオンは電気的に反発するため，少し隙間が空くのですが，今回の図は理想化された NaCl「型」結晶で，大きな球の作る面心立方格子[*1]の隙間[*2]に小さな球がピタッと入るものです。計算によると，立方格子の立方体の 1 辺の長さ[*3]を L として，大きい方の球の半径は $\frac{\sqrt{2}}{4}L$, 小さい方の球の半径は $\frac{2-\sqrt{2}}{4}L$ となります。

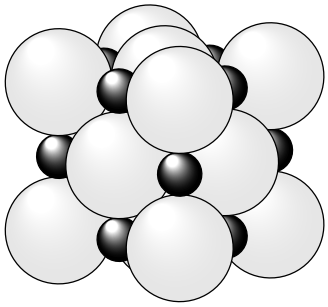


図 4.10.1

この図をどうやって描いているかですが，まず大雑把にやっていることを説明します。結晶格子を形成する球を立体っぽく見せているのは，**第2部 第5講** でやったグラデーションの手法です。今回は同じ球をたくさん描画するので，これをマクロ化しています。そして，この球を適切な位置に配置します。PostScript の仕様上，重なった部分は上書きとなりますので，裏側の球から配置していくだけで結晶格子ができます。ただ，この「裏側から」というのがくせ者で，ここを間違うと，見えているはずの球が本来隠れるはずの球で上書きされて見えなくなったり，あるいは,「だまし絵」状態になったりします。

では，ソースの解説に入りましょう。まず，球を描くマクロから。

%% グラデーションで立体的な球を描くマクロ

```
def Ball(expr _r,_ctr)(text _clr) =
  begingroup save _pp; path _pp[];
    _pp0 := (right..up..left..down..cycle) scaled _r rotated 120;
    _pp1 := _pp0 shifted _ctr;
    _pp2 := _pp0 scaled 1/4 shifted (_ctr + 2/3_r*dir120);
    for _t=0 step 0.02 until 1.01:
      fill interpath(sqrt _t,_pp1,_pp2) withcolor _clr;
    endfor
    draw _pp1;
  endgroup
enddef;
```

[*1] 面対角線方向に大きな球が接しています。

[*2] 正 8 面体間隙といいます。

[*3] 格子定数といいます。本来，格子定数とは，結晶格子の平行 6 面体を構成する 3 本の基底ベクトルの長さとなす角，都合 6 つのパラメータを指すのですが，今回のような立方格子の場合，3 本の基底ベクトルのなす角はすべて 90° で長さも等しいので，実質 1 辺の長さのみが有効な情報ととらえられます。そのため，高校化学では「1 辺の長さ」が格子定数と同義に使われています。

%% L を格子定数として NaCl 型（理想型）の 2 つの球を定義

```
numeric L; L:=2w;
def BigBall(expr _ctr) =
  Ball((sqrt2)*L/4,_ctr)(0.1(9+_t)*white);
enddef;
def SmallBall(expr _ctr) =
  Ball((2-sqrt2)*L/4,_ctr)(_t*white);
enddef;
```

大きい球（BigBall(**中心**)）も小さい球（SmallBall(**中心**)）も，半径と色が違うだけでやることは同じですので，それぞれのマクロから共通のマクロ

Ball(**半径，中心**)(**色変化**)

を呼び出す仕様になっています。

Ball でやっていることですが，グルーピング（begingroup〜endgroup）と save で一時変数を確保，確保した一時変数 _pp[] を path 値変数として宣言します。円周上の等間隔な 4 点以上を .. で結べばほぼ完全な円が得られますので，まず，中心が原点，半径が _r で，120° の方向を始点にもつ円 _pp0 を定め，それを移動して _pp1 に外側の円を，移動・縮小して _pp2 に内側の円を設定します。

グラデーションの本体は

```
for _t=0 step 0.02 until 1.01:
  fill interpath(sqrt _t,_pp1,_pp2) withcolor _clr;
endfor
```

の部分で，大きい円（_pp1）から小さい円（_pp2）までを，0.02 刻みで補間して色を塗ります。補間パラメータ _t は 0 から 1 までなのですが，小数の丸め誤差で最後の円が描かれないことがありますので，終値を少し大きく（1.01）しています。また，丸みを表現するため，補間を等間隔（_t）ではなく sqrt _t（つまり $\sqrt{t}$）にしています。色に当たる _clr は BigBall , SmallBall から送られてきます。前者は 0.1(9+_t)*white , 後者は _t*white です[*4]。このあたりは立体っぽく見えるように試行錯誤して決めて下さい[*5]。

さて，次は球の配置ですが，その前にちょっと結晶格子についてお話を。結晶格子とは，結晶構造の幾何学的表現で，1 つの頂点から出る 3 つのベクトルで張られる平行六面体（今回の場合立方体）を基本単位（単位格子）として表現されます。って，急に難しく (?) なりましたが，要は基本となる 3 つのベクトルさえ定義すれば，あとはその定数倍と和だけで球の配置場所を表現できるということです。この基本となる 3 つのベクトル — **pair** 値 — a , b , c は，次のように定めています[*6]。

%% 結晶格子の「基本ベクトル?」の定義

```
pair a,b,c;
```

[*4] _t は **text** タイプの〈引数〉ですので，〈置き換えテキスト〉の中で使われるまで評価されません。実際に使われるのは，〈置き換えテキスト〉の for 文の中ですので，_t は繰り返しのパラメータに解釈されます。

[*5] もっとも作っている筆者自身，「立体の表現」がよくわかってないので，3D の CG をやってる人から見れば，「おかしな絵」になってる可能性大ですが，その辺はご容赦を。

[*6] 実際には，結晶格子の辺の半分の長さのベクトルを a , b , c と定めました。NaCl 型の場合，こうしておくとすべての球の中心の位置ベクトルが，a , b , c の整数倍の和で表せて便利だからです。

```
normalvec((4,3,1)); upwardvec((0,0,1));
a:=proj((0.5L,0,0)); b:=proj((0,0.5L,0)); c:=proj((0,0,0.5L));
```

proj は **color** 値変数を 3 次元ベクトルと見て 2 次元に射影する MEPOTEX のマクロで，normalvec と upwardvec は，それぞれ射影面の法線ベクトル[*7]と上方向の指定[*8]のための MEPOTEX マクロです。射影面の法線ベクトルとは，別の言い方をすれば「見る方向」ですので，法線ベクトルとの内積が大きい位置ベクトルほど手前の点を表すといえます。したがって,「球を裏側から配置する」ということは「法線ベクトルとの内積の小さい順に球の中心の位置ベクトルを並べ替え，その順に球を配置する」ことに他なりません。その並べ替えをマクロ化したものを

SortColor(**位置ベクトル列の先頭タグ**, **個数**, **法線ベクトル**);

とすると，位置ベクトルの設定と並べ替えの部分は以下のようになります。

%% 位置ベクトル (color **値**) **の設定と並べ替え**

```
color cc[]; numeric n; n=0;
for k=0,1,2: for l=0,1,2: for m=0,1,2:
  n:=n+1; cc[n]:=(k,l,m);
endfor endfor endfor
SortColor(cc,n,_cn);
```

なお，並べ替えマクロの定義は，以下の通りです[*9]。

%% 位置ベクトル (color **値**) **の並べ替えマクロ**

```
def SortColor(suffix _c)(expr _num,_axis) =
  begingroup save _v; numeric _v[];
    for _n = 1 upto _num:
      _v[_n] := _c[_n] naiseki _axis;
      for _k = _n downto 2:
      exitif _v[_k]>_v[_k-1];
        _v0      := _v[_k];   _c0      := _c[_k];
        _v[_k]   := _v[_k-1]; _c[_k]   := _c[_k-1];
        _v[_k-1] := _v0;      _c[_k-1] := _c0;
      endfor
    endfor
  endgroup;
enddef;
```

アルゴリズムは単純で，端から順番に調べていって大小が逆順になっていれば入れ替える — いわゆるバブルソート (の変種？) です。

そして，いよいよ結晶格子の描画です。

[*7]法線ベクトルは，単位ベクトルにして **color** 値変数 _cn に保存されます。

[*8]法線ベクトルをもとに補正されますのでおおよそで大丈夫です。すでにお気づきかと思いますが，upwardvec の指定がおおよそでいいのは，**第8講** で説明した「直交化」のテクニックで補正を加えているためです。そして，立方格子を平面上に射影するテクニックは **第7講** で扱ったものです。

[*9]naiseki は MEPOTEX のマクロで，空間ベクトルの内積を表します。

%% 結晶格子の描画

```
for k=1 upto n:
  if odd(cc[k] naiseki (1,1,1)): SmallBall else: BigBall fi
     (redpart cc[k] * a + greenpart cc[k] * b + bluepart cc[k] * c);
endfor
```

NaCl 型結晶の場合，格子定数 L の半分を単位長とした場合，x 座標 $+\,y$ 座標 $+\,z$ 座標 が奇数になる点が Na^+ イオン (小さい濃色の球) , 偶数になる点が Cl^- イオン (大きい淡色の球) となっていますので，それを条件分岐で分けています。なお，球を配置する位置ベクトルは，この x 座標，y 座標，z 座標を a , b , c の係数として和を取ったもので与えられます。

Exercise 4.10　**図 4.10.2** の「体心立方格子」を描いてみて下さい。マクロは流用して下さって結構です。球の半径は $\frac{\sqrt{3}}{4}L$ です。

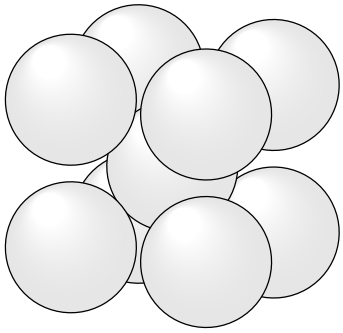


図 4.10.2

問題解答

Exercise2.1

MEPOTEX でのソースを挙げます。METAFONT や METAPOST でも同様です。なお，問題文に添えた図は，灰色で描かれ，描画 **点** に番号が打たれていますが，解答ではその部分は省略します。したがって，以下のソースで図を描くと，単純な黒線の図になります[*1]。

```
¥begin{MPpic}<4cm>(1,0.8|0.2)
¥openMP
numeric radius; radius=0.5w;
pair ctr; ctr=(0.5w,0.5h);
for n=0 upto 5: z[n] = ctr + radius*dir(60n+90); endfor
pickup pencircle scaled 0.05w;
draw z0..z1..z2..z3..z4..z5..cycle;
draw z0--z2--z4--cycle;
¥closeMP
¥end{MPpic}
```

Exercise2.2

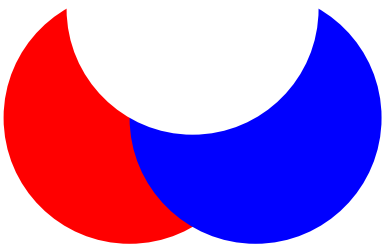
たとえば次のようになります。

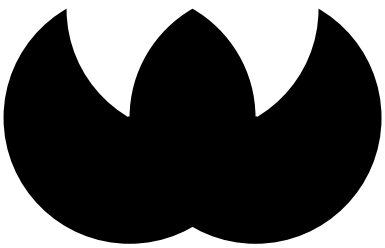
```
path pth;
pth:=z0..tension 0.8..{curl 0}z1{dir(-50)}..{dir(-80)}z2{dir140}..
     {dir150}z3{curl 0}..tension 2 and 1.2..z4{left}
     ..tension 0.8 and 2..z5..tension 2 and 0.8..
     z6{left}..tension 1.2 and 2..{curl 0}z7{dir(-150)}
     ..{dir(-140)}z8{dir80}..{dir50}z9{curl 0}..tension 0.8..cycle;
```

もちろんこれのみが解答というわけではありません。いろいろと試してみて下さい。

Exercise2.3

下図左は MEPOTEX の実行結果です。MEPOTEX と METAPOST は同じ結果のはずです。下図右は METAFONT の実行結果（を MEPOTEX で擬似的にまねたもの）です。





[*1] 今後もとくに断りのない限り同様です。この点について詳しくは p.*54* の dotLabels の説明をご覧下さい。

Exercise2.4

MEPOTEX での作成例です。

```
¥begin{MPpic}<3cm>(1,0)
¥openMP
path p; p:=(-0.5w,0)--(0.5w,0);
pickup pencircle scaled 0.05w;
for n=0,1,2:
  z[n] = (0.5w,0.5h) + 0.2w*dir(-90+120n);
  draw (p rotated(120n) shifted z[n]);
endfor
¥closeMP
¥end{MPpic}
```

Exercise2.5

MEPOTEX での作成例です。

```
¥begin{MPpic}<3cm>(1,0)
¥openMP
path p,q;
p:=StarShape((0.5w,0.5h),0.5w,0);
q:=StarShape((0.45w,0.55h),0.05w,0);
for t=0 step 0.02 until 1.01:
  fill interpath (t*t,p,q) withcolor hsb(0.15,1-0.9t,0.8+0.2t);
endfor
¥closeMP
¥end{MPpic}
```

なお，星形生成マクロは以下のように定義しました[*2]。

```
vardef StarShape(expr _zc,_r,_theta) =
save _z,_pth; pair _z; path _pth;
  _z=whatever[dir90,dir-54]; _z=whatever[dir18,dir162];
  _pth:=(for n=0 upto 4: _z rotated 72n--dir(90+72n)-- endfor
         cycle) scaled _r rotated _theta shifted _zc;
  _pth
enddef;
```

[*2]マクロ定義については **第2部 第10講** を参照して下さい。

Exercise2.6.1

以下の通りです。¥mptDrawPath は MEPOTEX のマクロです[*3]。ここでの

¥mptDrawPath[fillcolor=**色**]{**経路**}

の意味は,

fill **経路** withcolor **色**; draw **経路**;

と思って下さって結構です。(正確な意味については MEPOTEX のマニュアルを参照して下さい。)

```
¥begin{MPpic}<15mm>(3,0)(-1,-1.5)
¥sendMP{%
  for n=0 upto 3: z[n]=w*dir(90(n+1));endfor
  path p,q,r;
  p:=z0..z1..z2..z3..cycle;
  q:=p shifted (w*right); r:=p shifted (w*dir60);}
¥mptDrawPath[fillcolor=red]
    {buildcycle(SP(0,4,p), SP(2,-4,q),SP(2,-4,r))}
¥mptDrawPath[fillcolor=green]
    {buildcycle(SP(0,-4,p),SP(1,4,q), SP(0,-4,r))}
¥mptDrawPath[fillcolor=blue]
    {buildcycle(SP(1,-4,p),SP(1,-4,q),SP(2,4,r))}
¥mptDrawPath[fillcolor=Yellow]
    {buildcycle(SP(0,-4,p),SP(2,-4,q),SP(1,4,r))}
¥mptDrawPath[fillcolor=Cyan]
    {buildcycle(SP(2,4,p), SP(1,-4,q),SP(0,-4,r))}
¥mptDrawPath[fillcolor=Magenta]
    {buildcycle(SP(1,-4,p),SP(0,4,q), SP(2,-4,r))}
¥mptDrawPath[fillcolor=white]
    {buildcycle(SP(2,4,p), SP(0,4,q), SP(1,4,r))}
¥end{MPpic}
```

Exercise2.6.2

cull currentpicture dropping (0,0) — つまり 0 だけ 0 (のまま) にし, 他は 1 にします。

Exercise2.7

以下の通りです。(MEPOTEX での解答例です。)

```
¥begin{MPpic}<8mm>(3,0)(-1.5,0)
¥openMP
path spade;% ここから 4 行がスペード型の定義
spade := (0,2h){curl 0}..tension2..(w,0)
                        ..tension1.5..(0.1w,0)--(0.4w,-0.5h);
```

[*3]詳しくは p.***52*** を参照して下さい。

```
spade := spade -- reverse spade xscaled -1 --cycle;
for xx=-3w step 2mm until w:% 2 mm 間隔で少し広い範囲に斜線を引く
  draw (origin -- 4w*dir45) shifted (xx,-0.5h);
endfor
clip currentpicture to spade;% はみ出した斜線を消去
draw spade;% スペード型の輪郭を描画
¥closeMP
¥end{MPpic}
```

Exercise2.8

たとえば以下のようになります。

```
¥begin{MPpic}<8mm>(6.5,7)(-2,-3)
¥sendMP{path pth.p[], pth.l, pth.f;
  pth.p1 = kansu(-t*t+3t)(-2,4,50);%          pth.p1 は上に凸の放物線
  pth.p2 = kansu(t*t-t)(-2,4,50);%            pth.p2 は下に凸の放物線
  pth.l  = kansu(t+1)(-2,4,1);%               pth.l  は          直線
  pth.f  =(0,4h)--(-2w,4h)--(-2w,-2h)--(4w,-2h)--(4w,4h)--cycle;%枠
}
¥mptDrawPath[filltype=amikake,amidir=110,linetype=none]%   斜線領域
  {buildcycle(pth.p1, reverse pth.p2)}
¥mptDrawPath[fillcolor=0.7white,linetype=none]%        影付き領域(1)
  {buildcycle(pth.p1, reverse pth.l, pth.p2)}
¥mptDrawPath[fillcolor=0.7white,linetype=none]%        影付き領域(2)
  {buildcycle(reverse pth.p1, pth.l, reverse pth.p2)}
¥mptDrawPath[pensize=0.4pt,linetype={dashed evenly}]%          点線
  {(2w,0)--(2w,2h)--(0,2h)}%                                この次の行も
¥mptDrawPath[pensize=0.4pt,linetype={dashed evenly}]{(w,0)--(w,2h)}
¥sendMP{forsuffixes t=p1,p2,l: draw pth.t; endfor}%      グラフ描画
¥mptXaxis[tr]<0mm,-1mm>{$x$}%                                 x 軸
¥mptYaxis|-2h==4h>[tl]<1mm,0mm>{$y$}%                         y 軸
¥mptLabel{(0,0)}[tr]<-0.5mm,-0.5mm>{0}%                   原点
¥mptLabel{(w,0)}[t]<0mm,-1mm>{1}%                         座標
¥mptLabel{(2w,0)}[t]<0mm,-1mm>{2}%                        座標
¥mptLabel{(0,2h)}[r]<-1mm,0mm>{2}%                        座標
¥mptLabel{(reverse pth.p1) intersectionpoint pth.f}
  [br]{$y=-x^2+3x$}%                                      関数式
¥mptLabel{pth.p2 intersectionpoint pth.f}[b]{$y=x^2-x$}%  関数式
¥mptLabel{pth.l intersectionpoint pth.f}{$y=x+1$}%        関数式
¥mptClip[ymin=-2h,ymax=4h]%                       はみ出した部分を消去
¥end{MPpic}
```

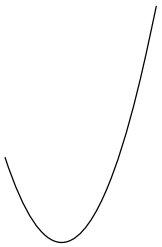
Exercise2.9

ペン先 の変更で実現する例です。

```
¥begin{MPpic}<40mm,50mm>(1.2,0|1)(-0.1,0.8)
¥openMP
  z1a = (0,h); z3 = (0.5w,0); x2a = x1a;  x3 = x5 = x6; x4a = 0.1w;
  y2a = 0.5h;  y4a= 0.6h;     y5  = 0.9h; y6 = 0.2h;
  for n=1,2,4: x[n]b = w - x[n]a; y[n]b = y[n]a; endfor
  pickup pensquare scaled 15pt; draw z1b--z1a;
  pickup pencircle scaled 15pt;
      draw z1a--z2a{dir(-90)}..
           {dir(-45)}z3{dir(45)}..{dir(90)}z2b--z1b;
  pickup pensquare scaled 15pt rotated 45; drawdot z3;
  pickup penrazor scaled 15pt rotated90;   draw z4a--z4b;
  pickup penrazor scaled 15pt;             draw z5--z6;
¥closeMP
¥end{MPpic}
```

Exercise2.10.1

右図は NIJIKANSU(1,2,3)(1,0,3); で出力しました。



```
def NIJIKANSU(expr xa,xb,xc)(expr ya,yb,yc) =
begingroup
  save a,b,c,t,zz; numeric a,b,c,t; pair zz[];
  ya = a * xa * xa + b * xa + c;% 連立方程式で
  yb = a * xb * xb + b * xb + c;% 係数 a , b , c を
  yc = a * xc * xc + b * xc + c;% 確定する。
  for n=0 upto 50:
    t:=(n/50)[xa,xc];
    zz[n]=(t , a*t*t + b*t + c);
  endfor
  draw (for n=0 upto 49: zz[n]-- endfor zz50) scaled w;
endgroup
enddef;
```

Exercise2.10.2

これは実際に MEPOTEX で使われているマクロです。

```
vardef gaisin(expr _aa,_bb,_cc) =
save _oo; pair _oo;
  _oo=0.5[_bb,_cc]+whatever*dir(angle(_cc-_bb)+90);
  _oo=0.5[_cc,_aa]+whatever*dir(angle(_aa-_cc)+90);
  _oo
enddef;
```

Exercise3.1.1

数値トークン, スパーク, タグ と色分けすると，次のようになります[*4]。

```
xa := 1.23 ;
xb = - 2.34 ;
xc = sind 30 + ( xa + xb ) * cosd 60 ;
```

⟨ トークン ⟩ の区切りの空白が除かれていることにも注意して下さい。

Exercise3.1.2

数値トークン, 記号トークン と色分けすると，次のようになります[*5]。

```
xa 100.3 b m # ? +-- < [[ ^ ] // m mn .... { dir 30 } ;
```

⟨ トークン ⟩ の区切りのピリオドや空白が除かれていることにも注意して下さい。

Exercise3.1.3

次のようになります。

⟨ 変数名 ⟩ ym.n25.31pq ⟶ 「⟨ タグ ⟩ ym」＋ 「⟨ サフィックス ⟩ n25.31pq」
⟨ サフィックス ⟩ n25.31pq ⟶ 「⟨ サフィックス ⟩ n25.31」＋ 「⟨ タグ ⟩ pq」
⟨ サフィックス ⟩ n25.31 ⟶ 「⟨ サフィックス ⟩ n」＋ 「⟨ 添字 ⟩ 25.31」
⟨ サフィックス ⟩ n ⟶ 「⟨ 空 ⟩」＋ 「⟨ タグ ⟩ n」

Exercise3.1.4

通常 + は ⟨ スパーク ⟩ であり，⟨ スパーク ⟩ は ⟨ サフィックス ⟩ に使えないことを考えれば，a.+ という ⟨ 変数名 ⟩ は使えないことがわかります。もし，この ⟨ 変数名 ⟩ が許されるなら — 直後にピリオドも ⟨ 数値トークン ⟩ も続かないピリオドは空白と同じなので — a.+ は a␣+ と同じになり，これも ⟨ 変数名 ⟩ になってしまうため，たとえば「a + b」は「a 足す b」ではなく,「a + b」で 1 つの ⟨ 変数名 ⟩ ということになり，文法が根本的に崩れてしまいます。⟨ スパーク ⟩ と ⟨ タグ ⟩ の区別は，このように非常に重要な意味をもつわけです。

Exercise3.1.5

c[a][b] は c[5][3] になりますので，値 6 をもつ変数です。

c[a*b] は a と b の積が 15 になりますので，c15 の意味になります。したがって，値 2 をもつ変数です。

c[ab] は未定義の変数 (**numeric** の未知数と解釈されます) を添字にしてしまったので，

```
! Improper subscript has been replaced by zero.
```

のエラーが出て，続行させると c0 と解釈されます。これは値の定まっていない変数です。

c.a.b はこれ全体で 1 つの ⟨ 変数名 ⟩ になりますので，値の定まっていない変数です。

c␣a␣b は c.a.b と全く同じ意味をもちますので，上と同じく値の定まっていない変数です。

[*4] ⟨ スパーク ⟩ は plain なものに限定しています。
[*5] 今回意味のない文字の羅列ですので，⟨ スパーク ⟩ と ⟨ タグ ⟩ はまとめて ⟨ 記号トークン ⟩ としました。

c(a)(b) は，マクロでもない c が引数 (のようなもの) をもっているため，(a)(b) の部分が余分な ⟨ トークン ⟩ と見なされ，

```
! Extra tokens will be flushed.
```

のエラーが出て，続行させると「余分な ⟨ トークン ⟩ 」は無視され c と解釈されます。これも値の定まっていない変数です。

Exercise3.2.1

2 と x の間の暗黙の積や，符号 - と 7 の結びつきが，優先順位最高であるため，優先順位 3 番目の ++ より先に行われるためです。

なお，3++␣-7 において，- の前の空白を除くと，METAFONT 言語は

3 ++- 7

のように ⟨ トークン ⟩ を切り分け，その結果，++- は (とくに定義されていないので) ⟨ タグ ⟩ ，7 は ++- の ⟨ 添字 ⟩，つまり，3++-7 は ⟨ 添字 ⟩ をもった変数 ++-7 の 3 倍という意味になってしまいます。

Exercise3.2.2

METAFONT 言語はソースを左から読んでいき，まず左の / の前後が数値トークンであることから $\frac{1}{256}$ を作ります。これ (1/256) は，⟨ 数値トークン ⟩ どうしの商ですので優先順位最高の演算による結合となり，⟨ 数値 1 位式 ⟩ を形成します。そして，右の / により $\frac{1}{256} \div 256 = \frac{1}{256\times 256}$ が作られます。こちらは / の前が ⟨ 数値トークン ⟩ ではなく ⟨ 数値 1 位式 ⟩ ですので，通常の商，すなわち優先順位 2 番目の演算による結合となり，⟨ 数値 2 位式 ⟩ を形成します。なお，1/256/256 は epsilon の定義にもなっています。

Exercise3.2.3

x/2/2 の 2 つの / のうち，後ろの方のみが両側に ⟨ 数値トークン ⟩ をもつので，後ろの / が優先順位最高の商，前の / が普通の (優先順位 2 番目の) 商になります。したがって，x/2/2 は x/(2/2) の意味になります。

Exercise3.2.4

* は通常の積なので，優先順位は 2 番目ですが，/ はその両側が ⟨ 数値トークン ⟩ ですので，優先順位最高の商となり，その結果 / の方が先に実行されます。計算結果は,「割り切れる」ときは積・商どちらが先でも同じですが,「割り切れないとき」は丸め誤差が変わってきます。たとえば $\frac{100}{9}$ より $10 \times \frac{10}{9}$ の方が誤差が大きく現れますので，

```
(10*10)/9 は 11.11111
10*10/9   は 11.11115
```

になります。また，

```
(1000*1000)/999
```

は，先に 1000×1000 を行おうとして

```
! Arithmetic overflow.
```

(桁あふれ) を起こしますが，

```
1000*1000/999
```

は $1000 \div 999$ が先に行われるので桁あふれしません。

Exercise3.2.5

1/2[2,4] と (1/2)[2,4] はいずれも $(1-\frac{1}{2})\times 2+\frac{1}{2}\times 4=3$ を表します。

-1/2[2,4] は [2,4] が 1/2 を ⟨ 数値アトム ⟩ として要求し，1/2[2,4] で 1 つのかたまり (⟨ 数値 1 位式 ⟩) を形成します。-1/2[2,4] は，これに - がついたものと解釈されますので，-3 という値になります。

(-1/2)[2,4] は [2,4] が要求する ⟨ 数値アトム ⟩ が，( ) で囲われた ⟨ 数値式 ⟩ である (-1/2) になっているため，$(1-\frac{-1}{2})\times 2+\frac{-1}{2}\times 4=1$ を表します。

Exercise3.3.1

最初の例が以下のエラーを出します。

```
! Improper transformation argument.
```

shifted などの ⟨ 変形演算子 ⟩ は後ろに ⟨1 位式 ⟩ を要求します。shifted (平行移動) の場合，移動量を表す ⟨ ペア 1 位式 ⟩ が要求されるのですが，2*up は ⟨ ペア 2 位式 ⟩ ですので，切り分けられて 2 のみが shifted の ⟨ 引数 ⟩ と解釈されてしまいます。しかし，2 は当然 ⟨ 数値 1 位式 ⟩ ですので,「引数が不適切」のエラーになります[*6]。

他の例の shifted の ⟨ 引数 ⟩ が，いずれも ⟨ ペア 1 位式 ⟩ になっていることも確認しておいて下さい。なお，最初の例も shifted 2up なら 2up が ⟨ ペア 1 位式 ⟩ となるため，エラーは起こりません。

Exercise3.3.2

以下のようになります。(primary が本来の dir)

	dir 90*2	dir 90+90
primary	(0,2)	エラー
secondary	(-1,0)	エラー
tertiary	(-1,0)	(-1,0)

エラーというのは次のエラーです。

```
! Not implemented: (pair)+(known numeric).
```

META 言語では，⟨ 区切りなし引数 ⟩ をもつように定めたマクロは，直後に続く式から要求された「位数」でできるだけ長いものを取り出し，それを ⟨ 引数 ⟩ にします。90*2 は ⟨ 数値 2 位式 ⟩ , 90+90 は ⟨ 数値 3 位式 ⟩ であるため，これらを ⟨ 引数 ⟩ にするには，前者なら secondary 以上，後者なら tertiary の「位数」が必要で，それより小さい「位数」(たとえば primary) では，最初の 90 のみが ⟨ 引数 ⟩ になります。そのため，dir 90*2 の方は,

```
dir 90*2 = (dir 90) * 2 = (0,1) * 2 = (0,2)
```

となります。また dir 90+90 の方は,

```
dir 90+90 = (dir 90) + 90 = (0,1) + 90
```

となり,「**pair** と **numeric** の加算は実装されていません。」のエラーが出るわけです。

[*6] このエラーは慣れた人でもよくやっちゃいます。

Exercise3.3.3

$(\ x\ ,\ y\ ,\ z\ ) = (\ R\cos\theta\cos\varphi\ ,\ R\cos\theta\sin\varphi\ ,\ R\sin\theta\ )$ とおくと,

ベクトル $(\ \sqrt{x^2+y^2}\ ,\ z\ ) = R(\ \cos\theta\ ,\ \sin\theta\ )$

ベクトル $(\ x\ ,\ y\ ) = R\cos\theta(\ \cos\varphi\ ,\ \sin\varphi\ )$

であるため,

(緯度 θ) = [$(\ \sqrt{x^2+y^2}\ ,\ z\ )$ の方向角]

(経度 φ) = [$(\ x\ ,\ y\ )$ の方向角]

となります。したがって，マクロの定義は次のようになります。

```
vardef latitude  primary c = % 緯度
  angle (redpart c ++ greenpart c , bluepart c)
enddef;
vardef longitude primary c = % 経度
  angle (redpart c , greenpart c)
enddef;
```

Exercise3.3.4

次式で望みの結果が得られます。

```
z1 = (3,2); z2 = z1 + 3dir45; z3 - origin = z2 - z1;
```

Exercise3.4.1

$$\begin{aligned}\overrightarrow{\mathrm{OP}} &= (1-s)^3\overrightarrow{\mathrm{OA''}} + 3s(1-s)^2\overrightarrow{\mathrm{OC''}} + 3s^2(1-s)\overrightarrow{\mathrm{OD''}} + s^3\overrightarrow{\mathrm{OB''}}\\ &= \frac{1}{8}(1-s)^3(\overrightarrow{\mathrm{OA}} + 3\overrightarrow{\mathrm{OC}} + 3\overrightarrow{\mathrm{OD}} + \overrightarrow{\mathrm{OB}}) + \frac{3}{4}s(1-s)^2(\overrightarrow{\mathrm{OC}} + 2\overrightarrow{\mathrm{OD}} + \overrightarrow{\mathrm{OB}})\\ &\quad + \frac{3}{2}s^2(1-s)(\overrightarrow{\mathrm{OD}} + \overrightarrow{\mathrm{OB}}) + s^3\overrightarrow{\mathrm{OB}}\\ &= \Big(\frac{1-s}{2}\Big)^3\overrightarrow{\mathrm{OA}} + 3\Big(\frac{1-s}{2}\Big)^2\Big(\frac{1+s}{2}\Big)\overrightarrow{\mathrm{OC}}\\ &\quad + 3\Big(\frac{1-s}{2}\Big)\Big(\frac{1+s}{2}\Big)^2\overrightarrow{\mathrm{OD}} + \Big(\frac{1+s}{2}\Big)^3\overrightarrow{\mathrm{OB}}\end{aligned}$$

となりますので，$t = \frac{1+s}{2}$ とおけば，$0 \le s \le 1$ の部分が $(*)$ の $\frac{1}{2} \le t \le 1$ の部分になることがわかると思います。

Exercise3.4.2

$t[z_1, z_2] = (1-t)z_1 + tz_2$ ですので,

$$\begin{aligned}t[z_1,z_2,z_3] &= t[\,t[z_1,z_2], t[z_2,z_3]\,] = (1-t)(t[z_1,z_2]) + t(t[z_2,z_3])\\ &= (1-t)\{(1-t)z_1 + tz_2\} + t\{(1-t)z_2 + tz_3\}\\ &= (1-t)^2z_1 + 2(1-t)tz_2 + t^2z_3\end{aligned}$$

$$\begin{aligned}t[z_1,z_2,z_3,z_4] &= t[\,t[z_1,z_2,z_3], t[z_2,z_3,z_4]\,] = (1-t)(t[z_1,z_2,z_3]) + t(t[z_2,z_3,z_4])\\ &= (1-t)\{(1-t)^2z_1 + 2(1-t)tz_2 + t^2z_3\}\\ &\quad + t\{(1-t)^2z_2 + 2(1-t)tz_3 + t^2z_4\}\\ &= (1-t)^3z_1 + 3(1-t)^2tz_2 + 3(1-t)t^2z_3 + t^3z_4\end{aligned}$$

となります。

Exercise3.4.3

$s[z_1', z_2', z_3', z_4']$ ($0 \leq s \leq 1$) の展開式において，z_1 ，z_2 ，z_3 ，z_4 の係数をそれぞれ求めると，

$$\begin{aligned}
z_1 \text{ の係数} &= (1-s)^3 + 3(1-s)^2 s(1-t_0) + 3(1-s)s^2(1-t_0)^2 + s^3(1-t_0)^3 \\
&= \{(1-s) + s(1-t_0)\}^3 = (1-st_0)^3 \\
z_2 \text{ の係数} &= 3(1-s)^2 st_0 + 3(1-s)s^2 \cdot 2(1-t_0)t_0 + s^3 \cdot 3(1-t_0)^2 t_0 \\
&= 3st_0\{(1-s)^2 + 2(1-s)s(1-t_0) + s^2(1-t_0)^2\}^3 \\
&= 3st_0\{(1-s) + s(1-t_0)\}^2 = 3(1-st_0)^2 st_0 \\
z_3 \text{ の係数} &= 3(1-s)s^2 {t_0}^2 + s^3 \cdot 3(1-t_0){t_0}^2 \\
&= 3s^2{t_0}^2\{(1-s) + s(1-t_0)\} = 3(1-st_0)(st_0)^2 \\
z_4 \text{ の係数} &= s^3{t_0}^3 = (st_0)^3
\end{aligned}$$

よって，$s[z_1', z_2', z_3', z_4'] = (st_0)[z_1, z_2, z_3, z_4]$ となりますので，$t = st_0$ とおけば，$t[z_1, z_2, z_3, z_4]$ の $0 \leq t \leq t_0$ の部分が $s[z_1', z_2', z_3', z_4']$ の $0 \leq s \leq 1$ の部分に一致することがわかると思います。

Exercise3.4.4

連立方程式は $\begin{cases} (\theta_0 + \phi_1) - 3\theta_0 = 1\{(\theta_0 + \phi_1) - 3\phi_1\} \\ (\theta_0 + \phi_1) - 3\phi_1 = 2\{(\theta_0 + \phi_1) - 3\theta_0\} \end{cases}$ となり，これを解くと

$(\theta_0 + \phi_1) - 3\theta_0 = (\theta_0 + \phi_1) - 3\phi_1 = 0$ すなわち $\theta_0 = \phi_1 = 0$

となります。

なお，`z0--z1` は `z0{curl 1}..tension 1 and 1..{curl 1}z1` となるのですが，この場合上記の 2 つの連立方程式は同値な式になってしまい，解が不定になります[*7]。この場合どう解釈されているのかは METAFONT ブックにも載ってないんですが，おそらく $(\theta_0 + \phi_1) - 3\theta_0 = (\theta_0 + \phi_1) - 3\phi_1 = 0$ のように $= 0$ を追加してるんじゃないかと思います。(⇐ 確かめてません (^_^;)

Exercise3.4.5

`(3,3)` と `(4,2)` の結合 `--` が優先順位 4 位，`scaled` による `(4,2)` と `2` の結合が 2 位ですので，`scaled 2` は `(4,2)` のみにかかることになります。したがって，

```
(3,3)--(4,2) scaled 2
```

は `(3,3)--(8,4)` の意味になります。

Exercise3.4.6

`intersectionpoint` が優先順位 3 位，`scaled` が 2 位ですので，`p` と `q scaled 2` の共有点，すなわち `(2,2)` が得られます。(実際は丸め誤差で `(2,1.99998)` になりますが。)

Exercise3.5

MEPOTEX の場合，以下のようになります。

```
¥begin{MPpic}<10mm>(6,0)(0,0.8)
¥openMP
  pen pentriangle;
```

[*7] 2 点からなる **path** で両端の `curl` の積が 1 のときにこうなります。

```
  pentriangle = makepen(0.5dir90--0.5dir210--0.5dir330--cycle);
  pickup pentriangle scaled 8mm;
  numeric MyTri; MyTri:=savepen;
  drawdot (3w,0);
  pickup pencircle scaled 1pt;
  z0=origin; z1=6w*right; z3=w*up; z2-z1=z3-z0;
  draw z0--z1--z2--z3--cycle;
  pickup MyTri;
  z10 + penoffset dir-60 of currentpen = z0;
  z11 + penoffset dir180 of currentpen = 0.5[z2,z3];
  z12 + penoffset dir60  of currentpen = z1;
  draw z10..z11..z12;
¥closeMP
¥end{MPpic}
```

METAPOST , METAFONT でも同様に希望の図が作れるでしょう。ただし METAFONT の場合，本格的に使うなら〈先物ペン〉として定義すべきですので，

```
pen pentriangle;
pentriangle = makepen(0.5dir90--0.5dir210--0.5dir330--cycle);
```

の代わりに

```
capsule_def(pentriangle)
  makepen(0.5dir90--0.5dir210--0.5dir330--cycle);
```

とする方がよいかも知れません。

なお，ソースで注目すべきは penoffset の〈引数〉に与える **pair** 値の向きです。具体的に言いますと，z10 には $-120° < \theta < 0°$, z11 には $120° < \theta < 240°$, z12 には $0° < \theta < 120°$ の向きを指定します。

Exercise3.6.1

METAPOST の定義に合わせて作ってみますと，以下のようになります。なお，_op_ は drawoptions の〈引数〉が設定されます。

```
def BGfill expr c = addto BGpicture contour c _op_ enddef;
def BGdraw expr p =
  addto BGpicture if picture p: also p
                  else        : doublepath p withpen currentpen
                  fi _op_
enddef;
def BGfilldraw expr c =
  addto BGpicture contour c withpen currentpen _op_
enddef;
def BGdrawdot expr z =
  addto BGpicture contour makepath currentpen shifted z _op_
```

```
enddef;
def BGunfill expr c = BGfill c withcolor background enddef;
def BGundraw expr p = BGdraw p withcolor background enddef;
def BGunfilldraw expr c = BGfilldraw c withcolor background enddef;
def BGundrawdot expr z = BGdrawdot z withcolor background enddef;
```

なお，これらのマクロを使うときは，BGpicture の初期化

```
picture BGpicture; BGpicture:=nullpicture;
```

も忘れないようにして下さい。

Exercise3.6.2

上から順に

```
dashed dashpattern(on 3 off 3)
dashed dashpattern(on 6 off 6)
dashed dashpattern(on 1.5 off 1.5 on 1.5 off 1.5 on 1.5)
dashed dashpattern(on 1.5 off 1.5 on 0 off 1.5 on 1.5)
dashed dashpattern(on 1.5 off 1.5 on 0 off 1.5 on 0 off 1.5 on 1.5)
dashed dashpattern(off 1.5 on 0 off 1.5)
```

で描かれています。dashpattern(on 3 off 3) は evenly の定義でもありますので，最初の例は dashed evenly としても同じ線を得られます。2 つ目の例は最初の例の「パターン」を 2 倍に拡大したものですので，dashed evenly scaled 2 としても同じ線を得られます。また，withdots の定義は dashpattern(off 2.5 on 0 off 2.5) ですので，最後の例は dashed withdots scaled 3/5 でも同じ線を得られます。

Exercise3.7.1

たとえば次のような定義でお望みのものが得られます。

```
vardef setTransform(expr _p,_q,_a,_b,_c,_d) =
  save _t; transform _t;
  xpart  _t = _p; ypart  _t = _q; xxpart _t = _a;
  xypart _t = _b; yxpart _t = _c; yypart _t = _d;
  _t
enddef;
```

Exercise3.7.2

$\overrightarrow{\mathrm{AC}}$ は $\overrightarrow{\mathrm{AB}}$ を $\frac{4}{3}$ 倍して時計回りに 100° つまり -100° 回転したものですので，

```
z3 - z1 = (z2 - z1) scaled 4/3 rotated -100;
```

で C(z3) を設定できます。$\overrightarrow{\mathrm{AB}} = \overrightarrow{\mathrm{OB}} - \overrightarrow{\mathrm{OA}} =$ b − a であること，および，回転の向き (回転角の正負) に注意して下さい。

なお，scaled 4/3 rotated -100 は複素数平面で，絶対値 $\frac{4}{3}$, 偏角 -100° の複素数をかけることに相当しますので，別解として

```
z3 - z1 = (z2 - z1) zscaled 4/3dir-100;
```

も可能です。

Exercise3.7.3

まず reflectedabout から。対称の軸となる直線上の 2 点は動きません。affine 変換は 3 点とその像の情報，つまり，**numeric** の方程式 6 個分の等式から決定されますので，あと 2 つ **numeric** の等式を与えれば変換は決定できます。reflectedabout の定義では，直線に関する対称移動において，affine 変換の行列部分が必ず $\begin{pmatrix} a & b \\ b & -a \end{pmatrix}$ となることを用いています。

次に rotatedaround。これは回転の中心をいったん原点に移し，角度 d だけ回転し，再び中心を元に戻しています。プリミティブの合成を使った定義例になっていることにも注意して下さい。

最後に inverse。T と合成して identity になるものというのが T の逆変換の定義です。このマクロの定義はそれを忠実に方程式で表したものです。

Exercise3.7.4

行列 $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ に対して A の行列式 $|A|$ を $|A| = ad - bc$ によって定めます。このとき，

$$A^2 - (a+d)A + |A|E = O \cdots\cdots (**)$$

が成り立つことが成分計算からわかります。ここに，$E = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $O = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$ です。また，2 つの行列 A , B に対して，$|A||B| = |AB|$ であることも成分計算からわかります。これらから，次のことが証明できます。

$AB = E$ が成立すれば，$BA = E$ も成立する。

《証明》$|A||B| = |AB| = |E| = 1$ より $|A| \neq 0$ となるので，$X = \dfrac{-1}{|A|}\{A - (a+d)E\}$ とおけば，$(**)$ から，$AX = XA = E$ が成立（逆行列 X が存在）することがわかります。したがって，

$B = EB = XAB = XE = X$（逆行列の一意性）

が言えて，$BA = XA = E$ がわかります。 《証明終》

さて，**transform** 値 t1 , t2 が順に $\overrightarrow{x'} = A\overrightarrow{x} + \overrightarrow{a}$, $\overrightarrow{x'} = B\overrightarrow{x} + \overrightarrow{b}$ で表されるとします。t2 を施した後 t1 を施すと，変換は

$\overrightarrow{x'} = A(B\overrightarrow{x} + \overrightarrow{b}) + \overrightarrow{a} = AB\overrightarrow{x} + (A\overrightarrow{b} + \overrightarrow{a})$

となります。これが恒等変換 identity つまり $\overrightarrow{x'} = E\overrightarrow{x} + \overrightarrow{0}$ に一致するとすれば，

$AB = E$ かつ $A\overrightarrow{b} + \overrightarrow{a} = \overrightarrow{0}$

となります。したがって，t1 を施した後 t2 を施したものは，

$\overrightarrow{x'} = B(A\overrightarrow{x} + \overrightarrow{a}) + \overrightarrow{b} = BA\overrightarrow{x} + B(\overrightarrow{a} + A\overrightarrow{b}) = E\overrightarrow{x} + \overrightarrow{0}$

となり，確かに恒等変換です。

Exercise3.7.5

以下のようになります。

```
primarydef _c XYtransformed _t =
  begingroup
    save _z; pair _z;
```

```
    _z = (redpart _c, greenpart _c) transformed _t;
    (xpart _z, ypart _z, bluepart _c)
  endgroup
enddef;
primarydef _c YZtransformed _t =
  begingroup
    save _z; pair _z;
    _z = (greenpart _c, bluepart _c) transformed _t;
    (redpart _c, xpart _z, ypart _z)
  endgroup
enddef;
primarydef _c ZXtransformed _t =
  begingroup
    save _z; pair _z;
    _z = (bluepart _c, redpart _c) transformed _t;
    (ypart _z, greenpart _c, xpart _z)
  endgroup
enddef;
```

Exercise3.8.1

次のような定義で望みのものが得られます。

```
def UniteDef(expr _sa,_sb)(text _tt) =
  expandafter def scantokens ( _sa & _sb ) = _tt enddef;
enddef;
```

expandafter⟨ トークン 1⟩⟨ トークン 2⟩ は，⟨ トークン 1⟩ を一旦保留して，⟨ トークン 2⟩ (それが ⟨ 引数 ⟩ をもつ命令なら ⟨ 引数 ⟩ も) を読み込んで先に展開してから，改めて，

⟨ トークン 1⟩⟨ トークン 2 (とその ⟨ 引数 ⟩) を展開したもの ⟩

を処理してくれるものです。これをつけないと，scantokens の再定義になってしまいますので注意して下さい。

Exercise3.8.2

完成版は，以下のようになります。

```
def drawNkaku(expr N) =
  if numeric N:
    if (N - floor N = 0) and (N>=3):
      draw (for m=1 upto N: 1cm*dir(90+360m/N)-- endfor cycle)
                                                shifted (1cm,1cm);
    else:
      errmessage"Argument must be an integer greater than 2";
    fi
  else:
```

```
      errmessage"Argument must be numeric";
    fi
  enddef;
```

なお，この定義で，`if numeric N:` と `if (N - floor N = 0) and (N>=3):` を分けているのは，METAFONT 言語が `and` で結ばれた〈論理式〉を判定するとき，つねにその〈論理式〉全体を評価するため，`N` が **numeric** でないとき，こちらが発行するエラーの前に `floor N` でエラーが出てしまうためです。この辺りをうまく処理して 1 つの分岐で済ます方法もないわけではありませんが，かなり複雑ですのでここでは触れません。興味のある方は，METAFONT ブックの Appendix D ダーティトリックを参照して下さい。

Exercise3.9.1

suffix タイプでは〈仮引数〉にはめ込む前に `[a]` が評価されて 2 になりますので，マクロ定義内の `a:=1;` の影響を受けませんが，**text** タイプでは〈仮引数〉にはめ込むときには `[a]` の評価を行わず `[a]` のままはめ込みますので，`a:=1;` の影響を受けます。したがって，前者では `s2` の「`Bad Morning(-"-)`」が，後者では `s1` の「`Good Morning(^_^)/`」が，それぞれ表示されます。

Exercise3.9.2

いずれの〈引数〉も〈式〉にはなっているのですが，最後の例だけ（**expr** タイプとしては）〈引数〉が 3 つあると見なされますので，

```
! Too many arguments to MyMacro;（〈引数〉多すぎ）
  Missing `)' has been inserted.
```

というエラーが出ます[*8]。他は受け入れられます。

〈仮引数〉のタイプ宣言を `suffix` に代えた場合は，最初の例では `1` のみが〈サフィックス〉で `+2` は「よけいなもの」と見なされ，

```
! Missing `)' has been inserted.（括弧の閉じ忘れ）
```

のエラー[*9]が出ます。2 つ目の例の `z1` は〈引数〉全体が〈サフィックス〉としての要請を満たしますので問題ありません。3 つ目の例では〈引数〉が〈文字列トークン〉だけのため，〈サフィックス〉とはどう転んでも解釈できず，〈引数〉は〈空〉で，よけいなもの `"abc"` が入っていると見なされます。したがって，エラーの種類は 1 つ目の例と同じになります。最後の例では，`a` , `b` , `c` それぞれは〈サフィックス〉としての要請を満たしているのですが，やはり〈引数〉が 3 つあると見なされますので，`expr` で宣言したときと同じく「〈引数〉が多すぎる」のエラーとなります。

〈仮引数〉のタイプ宣言を `text` に代えた場合は，〈置き換えテキスト〉で〈仮引数〉が現れるまで〈引数〉は解釈されませんので，いずれもエラーなく受け入れられます。とくに，コンマで区切られた複数の〈トークン〉をまとめて 1 つの〈引数〉にできるのは **text** タイプだけであることは，要注目です。

[*8] 1 つ目の〈引数〉の後に閉じ括弧が抜けていると解釈されるようです。

[*9] 括弧を閉じ忘れたからよけいなものが〈引数〉に現れたと解釈されるようです。

Exercise3.9.3

@# は〈変数型マクロ名〉に続く〈サフィックス〉をなるべく長く取ろうとしますので，a に続く〈サフィックス〉sss.ttt すべてが @# に引き取られ，b は〈空〉になってしまいます。@# に sss , b に ttt を引き取らせるには，a sss(ttt) とします[*10]。このように，@# のあとに〈区切りなし引数〉を置くと，@# と〈仮引数〉の境界が意図したところに来なくなる可能性がありますので，@# のあとに〈引数〉をつけたいなら，〈区切り付き引数〉であろうと〈区切りなし引数〉であろうと，( ) を付けて使う必要があります。使用時に ( ) を付けて使うのなら，〈定義ヘッダ〉であえて〈区切りなし仮引数〉を使う意味がないと思われるかも知れませんが，場合によっては〈区切りなし仮引数〉の方が便利なことがあります。これに関しては **第4部 第5講 2項** を参照して下さい。

Exercise3.9.4

vardef SetDir_z[]@#(expr d) = @# z.@=dir d; enddef; で，望みの結果が得られます。

Exercise3.9.5

save +; とすれば，(そのグルーピング内で) + の意味が失われ，単なる〈タグ〉になってしまいます。もちろんすべてのグルーピングの外でこれを行えば，そのソースが終わるまで + は本来の意味を失い，ただの〈タグ〉になってしまいます。

Exercise3.10.1

for N = 1 step 2 upto 11: S := S+N; endfor です。

Exercise3.10.2

forever: S := S+N; exitif (S>50); endfor です。

Exercise3.10.3

forever: exitif not(S<100); S:=S+N; endfor です。

Exercise4.1.1

図 4.1.2 はもとのソースの z2=z0+(7w,-15h); の部分を z2=z0+(7w,15h); に変えるだけです。また，**図 4.1.3** は同じ部分を z2=z0+(7w,-20h); に変え，z0a=z0+(-2.5w,7h); の部分を z0a=z0+(-w,7h); に変えるだけです。

Exercise4.1.2

次のようにします。

```
z1=2w*dir10;% 暫定値
for n=2 upto 5: z[n] = z1 rotated 72(n-1); endfor
```

Exercise4.1.3

次のようにします。

[*10]〈区切りなし引数〉は，〈定義ヘッダ〉の対応する〈仮引数〉が〈区切りなし仮引数〉である〈引数〉という意味で，「区切りをつけなくてもよい」— 逆に言えば「区切りを付けてもいい」—〈引数〉です。一方，@# は，表面上〈区切りなし仮引数〉のように扱えますが，最初に現れる「区切りまたは〈文〉末」までを〈引数〉として受け取りますので，「区切り」をつけてはいけません。それゆえ a sss(ttt) において，@# に sss ,〈区切りなし引数〉b に ttt が渡されるわけです。

```
  z1+z2+z3=(0,0);
  z2-z3=(z1-z3) rotated 120;
  z2-z1=4w*right; % 4w は暫定値
```

Exercise4.2.1

たとえば次のようになります。

```
¥begin{MPpic}<5mm>(15,8)(0,0)
¥sendMP{
def ObiGraph(expr lefttop,ww,hh)(suffix nm)(text tt) =
begingroup
save pth, n, v, c;
  path pth; numeric n,v[]; color c[]; pair nm[];
  n:=0; v0=0;
  for t=tt:
    n:=n+1;
    if odd n: v[(n+1)/2]:=v[(n-1)/2]+t;
    else    : c[n/2]:=t;
    fi
  endfor
  n:=n/2;
  for m=1 upto n:
    pth := ((v[m-1],0)--(v[m],0)--(v[m],-1)--(v[m-1],-1)--cycle)
                  xscaled (ww/v[n]) yscaled hh shifted lefttop;
    fill pth withcolor c[m];
    draw pth;
    nm[m]:=center pth;
  endfor
endgroup
enddef;
}
¥sendMP{ObiGraph((0,6h),15w,5h)(comment)
          (38,Red,38,Blue,28,Yellow,26,PineGreen,
           48,Cyan,6,Salmon,15,Purple);}
¥mptDrawPath[pensize=1pt]{comment6--(comment6+4h*up)}
¥mptLabel{comment1}{¥hbox{¥tate 第一章}}
¥mptLabel{comment2}{¥hbox{¥tate 第二章}}
¥mptLabel{comment3}{¥hbox{¥tate 第三章}}
¥mptLabel{comment4}{¥hbox{¥tate 第四章}}
¥mptLabel{comment5}{¥hbox{¥tate 第五章}}
¥mptLabel{(comment6+4h*up)}[b]{第六章}
```

```
¥mptLabel{comment7}{¥hbox{¥tate 目次その他}}
¥mptLabel{7.5w*right}[b]<0mm,0mm>{ある本の章構成}
¥end{MPpic}
```

Exercise4.2.2

たとえば次のようになります。

```
¥begin{MPpic}<10mm>(10,5.5)(0,0)
¥sendMP{
def BoGraph(expr leftBline,ww,unit)(suffix nm)(text tt) =
  begingroup
    save pth, b, n, v, c;
    path pth; numeric b,n,v[]; color c[]; pair nm[]; n:=0;
      for t=tt:
        n:=n+1;
        if odd n: v[(n+1)/2]:=t;
        else    : c[n/2]:=t;
        fi
      endfor
      n:=n/2;
    b:=ww/6n;
    for m=1 upto n:
      nm[m]:=((6m-3)*b,0) shifted leftBline;
      pth := ((-1,0)--(1,0)--(1,1)--(-1,1)--cycle)
                    xscaled b yscaled (unit*v[m]) shifted nm[m];
      fill pth withcolor c[m];
      draw pth;
    endfor
    draw ((0,0)--(ww,0)) shifted leftBline;
  endgroup
enddef;
}
¥sendMP{BoGraph(origin,9w,1mm)(comment)
          (38,Red,38,Blue,28,Yellow,26,PineGreen,
           48,Cyan,6,Salmon,15,Purple);}
¥mptLabel{comment1}[t]<0mm,-1mm>{1月}
¥mptLabel{comment2}[t]<0mm,-1mm>{2月}
¥mptLabel{comment3}[t]<0mm,-1mm>{3月}
¥mptLabel{comment4}[t]<0mm,-1mm>{4月}
¥mptLabel{comment5}[t]<0mm,-1mm>{5月}
¥mptLabel{comment6}[t]<0mm,-1mm>{6月}
```

```
¥mptLabel{comment7}[t]<0mm,-1mm>{7月}
¥mptLabel{(5w,-w)}<0mm,0mm>{売　上　高}
¥end{MPpic}
```

Exercise4.3.1

図 4.3.2 のソースは以下の通りです。defaultfont は "ptmb8r" にしました。

```
¥begin{MPpic}<10mm>(5,3)(-1,-0.5)
¥sendMP{z0=origin; z1=3w*right; z3=2w*up; z2-z1=z3-z0;
  defaultfont:="ptmb8r";
  dotlabel.llft("(0,0)",z0);
  dotlabel.lrt("(3,0)",z1);
  dotlabel.urt("(3,2)",z2);
  dotlabel.ulft("(0,2)",z3);
  draw z0--z1--z2--z3--cycle;}
¥end{MPpic}
```

また，**図 4.3.3** のソースは以下の通りです。

```
¥begin{MPpic}<10mm>(5,3)(-1,-0.5)
¥mptPoint[pensize=3pt,label={(0,0)},origin=tr]{z0}{origin}
¥mptPoint[pensize=3pt,label={(3,0)},origin=tl]{z1}{3w*right}
¥mptPoint[pensize=3pt,label={(0,2)},origin=br]{z3}{2w*up}
¥mptPoint[pensize=3pt,label={(3,2)},origin=bl]{z2}{z1+z3-z0}
¥mptDrawPath{z0--z1--z2--z3--cycle}
¥end{MPpic}
```

配置点の指定の仕方の違い（dotabel の〈サフィックス〉と ¥mptPoint の origin に指定されている文字の違い）にご注意下さい。

Exercise4.3.2

図 4.3.4 のソースは以下の通りです。

```
¥begin{MPpic}<9mm>(6,2.5)(0,0)
¥sendMP{z0=origin; z1=(3w,2w); z2=(5w,w); z3=(4w,0); z4=(w,2w);
        defaultfont:="ptmb8r";}
¥mptDrawPath[linecolor=0.6white,pensize=2mm]{z0..z1..z2..z3..z4}
¥sendMP{dotlabels.bot(0,1,2,3,4);}
¥end{MPpic}
```

また，**図 4.3.5** のソースは以下の通りです。

```
¥begin{MPpic}<9mm>(6,2.5)(0,0)
¥sendMP{z0=origin; z1=(3w,2w); z2=(5w,w); z3=(4w,0); z4=(w,2w);}
¥mptDrawPath[linecolor=0.6white,pensize=2mm]{z0..z1..z2..z3..z4}
¥sendMP[3pt]{dotLabels("[t]<0mm,-1mm>")(0,1,2,3,4);}
¥end{MPpic}
```

なお，MEPOTEX の dotLabels の「点」の太さは，currentpen に依るのですが，これは ¥sendMP のオプション引数で指定しました。

Exercise4.3.3

以下の通りです。

```
¥begin{MPpic}<0.1mm,5mm>(1080,2|1.3)(0,0)
¥openMP
  numeric act,tg; pair zp,zs; path p; picture pic; string s;
  p:= origin for n=30 step 30 until 1080:..(w*n,h*sind n) endfor;
  s:="abcdefghijklmnopqrstuvwxyz";
  for n:=0 upto 25:
    pic:=substring(n,n+1)of s infont "ptmb8r" scaled 2;
    zs:=center pic + (2n-25)/50*(lrcorner pic - llcorner pic);
    act:=arctime(n/25*arclength p) of p;
    zp:=point act of p;
    tg:=angle(direction act of p);
    draw pic shifted -zs rotated tg shifted zp
                                        withcolor hsb(n/26,1,1);
  endfor
  draw p;
¥closeMP
¥end{MPpic}
```

Exercise4.4.1

試してね。

Exercise4.4.2

以下のように表示されます。

```
a[]@#=macro:(SUFFIX3)(TEXT4)<primary>->begingroup(SUFFIX1)+(SUFFIX2) ETC.
```

〈区切りなし仮引数〉の表示形式に注意して下さい。ちなみに，〈定義ヘッダ部分〉が長かったので，〈置き換えテキスト〉は途中から省略されています。このようなことが起こるのは〈変数型マクロ〉の場合だけで，他のタイプのマクロでは〈置き換えテキスト〉部分は改行され，最後まできっちりと表示されます。

Exercise4.5.1

以下の通りです。

```
def EnGraph(expr ctr,r)(suffix nm)(expr f) =
  begingroup
    save pth, ptha, n, v, c;
    path pth, ptha; numeric n,v[]; color c[]; pair nm[];
    pth := (up..right..down..left..cycle) scaled r shifted ctr;
    n:=0; v0=0;
```

```
    for t= scantokens readfrom f:  n:=n+1; v[n]:=v[n-1]+t; endfor
    n:=0;
    for t= scantokens readfrom f:  n:=n+1; c[n]:=t;  endfor
    for m=1 upto n:
      ptha := (subpath 4(v[m-1],v[m])/v[n] of pth)--ctr--cycle;
      fill ptha withcolor c[m];
      draw ptha;
      nm[m]:=2/3[ctr,point 2(v[m-1]+v[m])/v[n] of pth];
    endfor
  endgroup
enddef;
```

このマクロに対し，たとえば (データが aaa.dat というファイルに入っているものとして)

```
  EnGraph((0,0),5w)(comment)("aaa.dat");
```

などとすれば，望みの結果が得られます。

Exercise4.5.2

`if color begingroup c endgroup: fill p withcolor c; fi` の部分を以下のように変更しても同じ効果が得られます。

```
  for cc = c: fill p withcolor cc; endfor
```

繰り返し構文のうち「**expr** 型 ⟨ 引数 ⟩ 列記型」では，⟨ 空 ⟩ は無視されることを思い出して下さい[*11]。

Exercise4.5.3

(仕様その 1) の定義例は以下のようになります。

```
def FillandDraw(expr p)text c =
  begingroup save cc,n; color cc[]; numeric n; n:=0;
  for t = c: n:=n+1; cc[n]:=t;  endfor
  if n>0: fill p withcolor cc1; fi
  draw p if n=2: withcolor cc2 fi;
  endgroup
enddef;
```

考え方は前問の解答の内容に沿っています。

(仕様その 2) の定義例は以下のようになります。

```
vardef FillandDraw@#(expr p)text c =
  if color begingroup c endgroup: fill p withcolor c; fi
  draw p if color begingroup @# endgroup: withcolor @# fi;
enddef;
```

[*11] 詳しくは **第3部 第10講 3項** を参照して下さい。

vardef を使えば，〈仮引数〉の先頭にも **suffix** タイプの〈区切りなし仮引数〉@#[*12] を使えることを思い出して下さい。また，vardef FillandDraw[]@#(expr p)text c とすれば，FillandDraw は〈添字〉を付けて使うことになり，これを〈引数〉のように使うこともできます[*13]。

なお，@# と text c の間に〈区切り付き仮引数〉(expr p) が入らない場合，つまり，〈定義ヘッダ〉が

```
vardef FillandDraw@#text c
```

の場合，

```
FillandDraw red (blue); は @# が red , c が blue
FillandDraw red;        は @# が red , c が〈空〉
FillandDraw (blue);     は @# が〈空〉, c が blue
FillandDraw;            は @# も c も〈空〉
```

となります。FillandDraw red blue; とすると @# が red.blue , c が〈空〉となってしまうことに注意して下さい。

Exercise4.6

以下のようになります (マクロ定義部分は省略しました)。

```
¥begin{MPpic}<2cm>(2,0)(-1,0)
¥sendMP{z0=(0,0); z1=(-0.5w,-0.5h); z2=(0,0.5h); z3=(0,0.9h);}
¥mptDrawPath[fillcolor=red,pensize=1pt]{circle(z0,w)}
¥Fukidasi{z0}{z1}{35mm}{この円の中心は (~0~,~0~) で，半径は 2cm です。}
¥Fukidasi*{z2}{z3}{2cm}{色は赤です。}
¥end{MPpic}
```

Exercise4.7.1

図 4.7.4 のソースは以下の通りです。

```
¥begin{MPpic}<20mm>(2.4,0)(-1.5,-1.5)
¥openMP
  pair a,b,c; a:=(1,0); b:=(0,1); c:=(-0.25,-0.43); z0=(0,0);
  z1=w*a; z2=w*b; z4=w*c; z3=z1+z2; z5=z1+z4; z6=z2+z4; z7=z1+z2+z4;
  draw for n=6,4,5,1,3,2,6,7: z[n]-- endfor z3; draw z5--z7;
¥closeMP
¥end{MPpic}
```

図 4.7.5 のソースは以下の通りです。

```
¥begin{MPpic}<20mm>(2.4,0)(-1.5,-3)
¥openMP
  pair a,b,c; a:=dir-30; b:=dir90; c:=dir210; z0=(0,0);
  z1=w*a; z2=w*b; z4=w*c; z3=z1+z2; z5=z1+z4; z6=z2+z4; z7=z1+z2+z4;
  draw for n=6,4,5,1,3,2,6,7: z[n]-- endfor z3; draw z5--z7;
```

[*12] ただし，通常の〈区切りなし引数〉と違い，( ) を付けてはいけません。
[*13] 詳しくは **第3部 第9講** を参照して下さい。

```
¥closeMP
¥end{MPpic}
```

Exercise4.7.2

図 4.7.7 のソースは以下のようになります。

```
¥begin{MPpic}<20mm>(1.1,0)(0,-0.3)
¥openMP
  color a,b,c,o,c[];
  o:=(0,0,0); a:=(1,0,0); b:=(0,1,0); c:=(0,0,-4);
  vardef prjb(expr v) =
  save X,Y;
    whatever[c,v] = X*a + Y*b + o;
    (X,Y)
  enddef;
  c1=(1,0,0); c2=(0,1,0); c3=(0,0,1); c4=(1,1,1);
  for n=1 upto 4: z[n]=prjb(c[n]) scaled w; endfor
  draw z1--z2--z3--z4--z1--z3; draw z2--z4;
¥closeMP
¥end{MPpic}
```

Exercise4.8.1

$\overrightarrow{a} = \frac{1}{3}(\,1\,,\,2\,,\,2\,)$, $\overrightarrow{b} = \frac{1}{3\sqrt{10}}(\,8\,,\,-5\,,\,1\,)$, $\overrightarrow{c} = \frac{1}{3\sqrt{10}}(\,4\,,\,5\,,\,-7\,)$ となります。METAPOST の実行結果はこれらの近似値で，順に，

```
(0.33333,0.66667,0.66667)
(0.84328,-0.52705,0.10541)
(0.42165,0.52705,-0.73787)
```

となります。

なお，前節の「正射影のために必要な基本ベクトルの設定」に関して一言補足しておきますと，射影する平面の法線ベクトルが「立体を見る方向」となりますので，これを $\overrightarrow{a'}$ として与え，平面上の X 軸のおおよその方向として $\overrightarrow{b'}$ を与えるのが現実的と思われます。このとき，平面上の X 軸，Y 軸の向きの単位ベクトルが上で計算される $\overrightarrow{b}$, $\overrightarrow{c}$, 単位法線ベクトルが $\overrightarrow{a}$ となります。$\overrightarrow{b}$ が $\overrightarrow{b'}$ を平面へ正射影したものになっていることにも注意して下さい。

Exercise4.8.2

順に $(\,4\,,\,0\,,\,0\,)$, $(\,2\sqrt{3}\,,\,0\,,\,-2\,)$, $(\,2\sqrt{3}\,,\,2\,,\,0\,)$, $(\,\sqrt{3}\,,\,3\,,\,2\,)$, $(\,0\,,\,4\,,\,0\,)$ の近似値になります。

最初の例では yz 平面内での回転を表すのですが，もとの点の y 座標，z 座標がともに 0 なので何も変化はありません。2 番目の例では zx 平面内での回転を表すため，$(\,z\,,\,x\,)=(\,0\,,\,4\,)$ が 30° 回転され $(\,-2\,,\,2\sqrt{3}\,)$ になります。この際，y 座標には変化はありません。3 番目の例では xy 平面内での回転を表すため，$(\,x\,,\,y\,)=(\,4\,,\,0\,)$ が

30° 回転され ($2\sqrt{3}$, 2) になります。この際，z 座標には変化はありません。4 番目の例では zx 平面内で，(z , x) = (0 , 4) が $-30°$ 回転され (2 , $2\sqrt{3}$) になり，さらにその後，xy 平面内で，(x , y) = ($2\sqrt{3}$, 0) が 60° 回転され ($\sqrt{3}$, 3) になり，結局，(x , y , z) = ($\sqrt{3}$, 3 , 2) が得られます。最後の例では回転軸として (1 , 1 , 1) ベクトルの方向を指定したことになります。この軸のまわりに 120° 回転すると，$x \to y \to z \to x$ の順に座標が入れ替わりますので，(4 , 0 , 0) は (0 , 4 , 0) に移ります。

Exercise4.8.3

方程式は以下のようになります。

```
d - a = (c - a) Rotated(latitude(b - a), longitude(b - a), ω);
```

仕組みですが，まず，a , b , c , d をすべて -a だけ平行移動します。こうすると，回転軸は原点を通る直線となり，先に定義した Rotated が使えるようになります。より具体的には，原点を通る方向ベクトル b-a の直線のまわりに，点 c-a を ω だけ回転したものが d-a ということになります。あとは方向ベクトル b-a の「緯度」「経度」を求めれば Rotated に必要な ⟨ 引数 ⟩ がそろうのですが，これは，既出のマクロ latitude , longitude で求められます。

なお，a:=(1,1,1) , b:=(3,2,2) , c:=(3,4,6) , $\omega = 60°$ で d を求めた結果は，以下の通りです。

```
(4.70714,0.6716,5.91423)
```

Exercise4.9.1

(35,43,51) が表示されます。

Exercise4.9.2

以下のようになります。

```
def Reflecterabout(expr a,b,c) =
  Transformed
  begingroup save _temp,d;
    color d; d:=(b-a)gaiseki(c-a);
    newTransform(_temp);
    for _c = a,b,c: _temp.s _c = _c; endfor
    _temp.s (a+d) = (a-d);
    realizeTransform(_temp);
    _temp.p
  endgroup
enddef;
```

Exercise4.9.3

以下のようになります。

```
def Rotatedaround(expr a,b,t) =
  Transformed
  begingroup save _temp;
```

```
    newTransform(_temp);
    for c = black, red, green, blue:
      _temp.s c = a + (c-a)Rotated(latitude(b-a),longitude(b-a),t);
    endfor
    realizeTransform(_temp);
    _temp.p
  endgroup
enddef;
```

Exercise4.10

以下のようになります。

```
¥begin{MPpic}<1cm>(5,0)(-2.5,0.5)
¥openMP
  numeric L; L:=2w;
  def BigBall(expr _ctr) =
    Ball(sqrt3*L/4,_ctr)(0.1(9+_t)*white);
  enddef;
  pair a,b,c;
  normalvec((4,2,1)); upwardvec((0,0,1));
  a:=proj((L,0,0)); b:=proj((0,L,0)); c:=proj((0,0,L));
  color cc[]; numeric n; n=0;
  for k=0,1: for l=0,1: for m=0,1:
    n:=n+1; cc[n]:=(k,l,m);
  endfor endfor endfor
  n:=n+1; cc[n]:=(0.5,0.5,0.5);
  SortColor(cc,n,_cn);
  for k=1 upto n:
   BigBall(redpart cc[k] * a + greenpart cc[k] * b
                             + bluepart  cc[k] * c);
  endfor
¥closeMP
¥end{MPpic}
```

METAFONT & METAPOST で楽々お絵かき　　ⓒみなも

2003 年 1 月 4 日 初版発行
2006 年 12 月 2 日 第 2 版発行
発行者 MEPOTEX 推進委員会 (嘘)
著　者 海月 水母 (嘘)
印刷所 あなたのぷりんたぁ
連絡先 http://homepage2.nifty.com/domae/

printout 2006 年 12 月 2 日