

# 十進 BASIC 利用の手引き

中・上級編

はじめに

この手引きは、BASICに限らず、ある程度プログラミングに慣れている方を対象として、十進 BASIC, Full BASIC の特徴とその利用法について述べたものです。

プログラミングが初めてという方は、(仮称)十進 BASIC による JIS Full BASIC 入門 <https://decimalbasic.web.fc2.com/tutorial/contents.htm> から始めることを推奨します。

必要に応じて、十進 BASIC help <https://decimalbasic.web.fc2.com/BASICHelp/BASICHelp.html> を参照してください。

十進 BASIC は、JIS Full BASIC 規格に準拠して作成されています。言語仕様の詳細は、<https://kikakurui.com/x3/X3003-1993-01.html> JIS X 3003-1993 を参照してください。規格との相違は、十進 BASIC help 言語仕様の詳細 JIS との相違 にあります。

2026年2月18日

2026年3月3日 修正

白石和夫

# 目次

1	入出力	3
	1.1	コンソール入出力 3
	1.2	ファイル入出力 6
2	数と文字列	9
	2.1	十進モード 9
	2.2	2進モード 11
	2.3	十進 1000 桁モード 11
	2.4	有理数モード 12
	2.5	複素数 12
	2.6	桁あふれの例外処理 13
	2.7	文字列変数 14
	2.8	文字コード 14
3	配列	17
	3.1	DIM 文 17
	3.2	行列の計算 19
4	制御構造	21
	4.1	制御構造 21
	4.2	関数 23
	4.3	副プログラム 27
	4.4	モジュール 29
5	グラフィックス	30
	5.1	問題座標 30
	5.2	絵定義 34
	5.3	ピクセル座標 39

# 1 入出力

## 1.1 コンソール入出力

### 1.1.1 PRINT 文

PRINT 文には、セミコロン(;) またはコンマ(,)で区切って、数値式、文字列式を書く。セミコロンのみで区切ると、出力項目間を詰めて出力する。コンマで区切ると、`zonewidth`として指定されたタブ位置まで出力位置を進める。末尾にセミコロン、または、コンマを書くと、次に実行される PRINT 文の出力が同じ行に続けて出力される。末尾にセミコロン、コンマがないと、改行する。PRINT に続けて何も書かない PRINT 文は、改行のみを実行する。

### 1.1.2 書式指定

#### PRINT USING 文

PRINT USING 文を利用して出力の書式を指定することができる。PRINT USING に続けて書式を書き、出力項目との間はコロン(:)で区切る。書式は、数字 1 桁を#で表す。末尾にセミコロンを書けば同じ行に続けて出力される。

#### 例 1

```
10 FOR i=1 TO 9
20   FOR j=1 TO 9
30     PRINT USING "####": i*j;
40   NEXT j
50   PRINT
60 NEXT i
70 END
```

#### 実行結果

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

#### 複数項目に書式を適用する

出力項目の分だけの書式を用意しておく。出力項目はコンマ(,)で区切る。

小数を出力するとき、小数点位置をピリオド(.)で指定する。

#### 例 2

```
10 FOR x=-4 TO 4
20   PRINT USING "##  ###. #####": x, SIN(x)
30 NEXT x
40 END
```

#### 実行結果

```

-4    .756802
-3   -.141120
-2   -.909297
-1   -.841471
 0    .000000
 1    .841471
 2    .909297
 3    .141120
 4   -.756802

```

数字位置として#を書くとき、小数点の直前の0は省いて表示される。

### 小数点直前の0を省かない

小数点直前の0を表示させたいときは、書式に”-----%.#####”の形式を利用する。

#### 例 3

```

10 FOR x=-4 TO 4
20   PRINT USING "##  --%.#####": x, SIN(x)
30 NEXT x
40 END

```

実行結果

```

-4    0.756802
-3   -0.141120
-2   -0.909297
-1   -0.841471
 0    0.000000
 1    0.841471
 2    0.909297
 3    0.141120
 4   -0.756802

```

小数点の左（整数部）に書く書式文字 +, -, % は、それぞれ、次の意味を持っている。

+ 数字, 符号(+), 空白に置換される。

- 数字, 負号(-), 空白に置換される。

% 数字に置換される。0を抑止しない。

小数部は#を用いて桁数を示す。このとき、末尾の0が抑止されることはない。

### 指数部ありの形式

指数部を^^^で指示する。指数部には、Eと符号の+, -も含まれる。

#### 例 4

```

10 FOR x=-1.5 TO 1.5 STEP 0.25
20   PRINT USING "+%.###  +%.#####^^^": x, TAN(x)
30 NEXT x
40 END

```

実行結果

```

-1.500  -1.41014199E+01
-1.250  -3.00956967E+00
-1.000  -1.55740772E+00
-0.750  -9.31596460E-01

```

```

-0.500  -5.46302490E-01
-0.250  -2.55341921E-01
+0.000  +0.00000000E+00
+0.250  +2.55341921E-01
+0.500  +5.46302490E-01
+0.750  +9.31596460E-01
+1.000  +1.55740772E+00
+1.250  +3.00956967E+00
+1.500  +1.41014199E+01

```

### 桁区切りのコンマを挿入する

正の整数に 3 桁ごとにコンマを挿入したいときは、”###,###,###”の形の書式を指定する。

#### 例 5

```

10 FOR i=8 TO 32 STEP 2
20   PRINT USING "## ###,###,###,###":i,2^i
30 NEXT i
40 END

```

実行結果

```

 8           256
10          1,024
12          4,096
14         16,384
16         65,536
18        262,144
20       1,048,576
22      4,194,304
24     16,777,216
26     67,108,864
28    268,435,456
30   1,073,741,824
32  4,294,967,296

```

### JIS 互換にしたいとき

上述の動作は、一部、JIS 非互換を含む。オプションメニューの「互換性-動作」で、「書式文字に#を用いたとき、先行する空白列を生成する (JIS)」を選ぶと、負号を生成する余地がなくなる。JIS 互換にしたいときは、負数を扱う可能性があるとき、”-----%.#####”の形式を使う。

### 1.1.3 INPUT 文

Full BASIC の INPUT 文には他の言語ではあまり見ない特徴がある。

INPUT 文にいくつかの変数を書いたとき、実行時には、それらに代入したい数値か文字列をコンマ(,)で区切って入力する。そのとき、入力の最後にコンマを書けば、次に続くことを意味する。つまり、1 回の INPUT 文に対応する入力を複数回に分けて実行できる。

他の多くの言語では、コンマに続けて何もなければ、対応する変数に null が代入される。他の言語との連携を考えると、この特性の違いに注意が必要だ。

## 1.2 ファイル入出力

### 1.2.1 テキストファイル出力

十進 BASIC では、通常、計算結果はテキスト出力ウィンドウに出力される。テキスト出力ウィンドウのファイルメニューからテキスト形式で保存できるが、プログラムから直接テキストファイルに出力することもできる。大量の数値を出力したいときは、テキスト出力ウィンドウを経由せずに、直接、テキストファイルに書き出すほうが速い（テキスト出力ウィンドウへの書き出しは遅い、特に Ver. 7）。

まず、はじめに決めることは、ファイルをどこに生成するか？

Windows の場合、エクスプローラに表示される「ドキュメント」や「デスクトップ」などのフォルダ名をプログラムで直接指定できない。エクスプローラで、ドキュメントフォルダ内のファイルを右クリックして、「全般」タブの「場所」に書かれたディレクトリ名を確認する。個人所有の PC では、おそらく「C:\Users\〇〇〇〇\Documents」のようにになっている。〇〇〇〇は、ログイン名。ただし、ネットワーク上にホームディレクトリを置く PC では、¥で始まるネットワークサーバのアドレスになっているかも知れない。

ここでは、ドキュメントフォルダに TEST.txt という名前でファイルを作ることにする。

プログラムの始めと END 行の直前に次のような OPEN 文と CLOSE 文を追加する。

```
OPEN #1:NAME "C:\Users\〇〇〇〇\Documents\TEST.txt"
```

```
.....
```

```
CLOSE #1
```

さらに、「PRINT」を「PRINT #1:」に書き換える。これで、テキスト出力ウィンドウを保存したのと同じ結果が得られる。

なお、そのようにして作成したファイルを書き換えるときは、OPEN 文に続けて「ERASE #1」を実行する。ERASE 文を空のファイルに適用してもエラーにはならないから、将来、書き換えを予定するプログラムでは最初から ERASE 文を書いておいてかまわない。

一方で、ログファイルのように、実行の度にテキストを追加していきたいときは、次の形の OPEN 文を書く。

```
OPEN #1:NAME "C:\Users\〇〇〇〇\Documents\TEST.txt", ACCESS OUTPUT
```

### 1.2.2 経路番号

#に続く数字の部分を経路番号という。経路番号には、1, 2, 3 など、正の整数を用いるが、数値変数を介して数値式で指定することもできる。経路番号の有効範囲はプログラム単位である。外部副プログラムや外部絵定義の引数に経路番号を書いて経路を異なるプログラム単位間で共有することができる。プログラム単位ごとに異なる番号にもできる。

### 1.2.3 内部形式ファイルと CSV

テキスト形式で書き出したファイルは、INPUT 文で読み込むのに適した形式ではない。BASIC の INPUT 文で読み込むためには、項目間にコンマを書きこまなければならない。テキスト形式のファイルを作るとき、そのようにプログラムを作り変えておくこともできるが、BASIC プログラムで読み込むことを前提にして用意されたファイル形式を内部形式ファイルという。

内部形式ファイルには次の形式の OPEN 文を書く。

```
OPEN #経路番号 NAME ファイル名 ,RECTYPE INTERNAL
```

```
OPEN #経路番号 NAME ファイル名 ,ACCESS OUTIN ,RECTYPE INTERNAL
OPEN #経路番号 NAME ファイル名 ,ACCESS INPUT ,RECTYPE INTERNAL
OPEN #経路番号 NAME ファイル名 ,ACCESS OUTPUT ,RECTYPE INTERNAL
```

ACCESS OUTIN は入出力両用である。ACCESS 指定を省いたときは、ACCESS OUTIN を指定したものともみなす。ACCESS INPUT は、入力専用である。ACCESS OUTPUT は、追記書き出し用である。既存のファイルの内容を書き換えたいときは、ACCESS OUTIN で開き、ERASE 文を実行して既存の内容を消去してから書き出す。

内部形式ファイルに書き出すとき、PRINT 文の代わりに WRITE 文を用いる。WRITE 文は、  
WRITE #経路番号: 出力項目, ……., 出力項目  
の形に書く。項目の区切りとして書くのはコンマ(,)である。

ファイルからの入力には、INPUT 文の代わりに READ 文を用いる。READ 文は、  
READ #経路番号: 変数, 変数, ……., 変数  
の形に書く。変数は数値変数、文字列変数のいずれでもよく、添字付き変数でもよい。

#### 例 6

```
10 OPEN #1:NAME "C:¥users¥○○○○¥documents¥TEST.CSV", RECTYPE INTERNAL
20 ERASE #1
30 FOR x=0 TO 1 STEP 0.1
40   WRITE #1: x, SQR(x)
50 NEXT x
60 CLOSE #1
70 END
```

このプログラムを実行すると、ドキュメントフォルダに TEST.CSV が生成される。メモ帳で内容を読むことができる。それを転記すると、以下のようになっている。

```
0 , 0
.1 , .316227766016838
.2 , .447213595499958
.3 , .547722557505166
.4 , .632455532033676
.5 , .707106781186548
.6 , .774596669241483
.7 , .836660026534076
.8 , .894427190999916
.9 , .948683298050514
1 , 1
```

このファイルを読み込むためのプログラムは、

#### 例 7

```
10 OPEN #1:NAME "C:¥users¥○○○○¥documents¥TEST.CSV", RECTYPE INTERNAL, ACCESS INPUT
20 DO
30   READ #1, IF MISSING THEN EXIT DO: x, y
40   PRINT x, y
50 LOOP
60 CLOSE #1
70 END
```

READ 文に 2 個の変数を指定して、2 つの項目を読み込んでいる。ファイル全体を読むために、

READ 文に、IF MISSING THEN EXIT DO を書いて、読み込む要素がなくなったら EXIT DO を実行する。

なお、ファイルを読み込むとき、ACCESS INPUT の指定を省いても動作するけれども、ファイル名の指定を間違えるとその名前のファイルを作ってしまうので、ACCESS INPUT を付加しておくことが望ましい。ACCESS INPUT を指定すると、指定されたファイルが存在しないとき、extype7102 の例外状態になる。

## CSV

十進 BASIC は、上に示した出力結果から分かるように、内部形式として CSV (コンマ区切り) を採用している。JIS 規格は、内部形式の具体的な中身を規定していない。だから、内部形式での出力ファイルについて他の Full BASIC 処理系との互換性はない。

十進 BASIC の内部形式は CSV なので、Excel などとのデータ連携に使える。内部形式ファイルを作るときに拡張子を CSV にしておけば、Excelなどで読み込むのは容易。逆に、Excel のデータを十進 BASIC で処理したいときは、データを必要な範囲に絞って CSV 形式で保存する。うまく読めないときは、CSV ファイルを、一旦、メモ帳などで読み込んで、保存形態を確認し、BASIC の READ 文をそれに適合するように修正する。1 行目、2 行目だけ別の READ 文を用意して空読みする (読み飛ばす) などの処理が必要になるかも知れない。

### 1.2.4 バイナリファイル

#### バイナリファイルを読む

十進 BASIC の単文字入力 (CHARACTER INPUT) では、行末文字を未定義としている。これは、行末文字も通常の文字と同等に扱うことを意味する。十進 BASIC のデフォルトは漢字などの多バイト文字を扱う設定になっているが、文字をバイトと同一に扱う設定にすれば、バイトファイルを CHARACTER INPUT 文を用いて読むことが可能になる。そのために必要なことは、独自拡張命令の OPTION CHARACTER BYTE をプログラムのはじめに書くこと。

#### 例 8

```
10 OPTION CHARACTER BYTE
20 FILE GetOpenName fn$
30 OPEN #1:NAME fn$, ACCESS INPUT
40 DO
50   CHARACTER INPUT #1, IF MISSING THEN EXIT DO: s$
60   PRINT ORD(s$)
70 LOOP
80 CLOSE #1
90 END
```

#### 説明

20 行の FILE GetOpenName は、ファイル名入力ダイアログを表示してファイルシステムからファイル名を取得する独自拡張命令。文字列変数 fn\$ に結果が代入される。

50 行の CHARACTER INPUT 文で 1 文字 (1 バイト) ごとに文字列変数 s\$ に読み込んでいる。

60 行の ORD 関数は、s\$ が 1 文字のみであるときにその文字コード (ordinal, 順序数) を返す関数。読み込んだバイトを数として表したものを意味する。バイナリファイルを処理するプログラムは、この値を対象に書けばよい。

### バイナリファイルを作る

文字コード  $n$  に対応する文字が  $\text{CHR}\$(n)$  で求まる。バイナリファイルを作成するには、バイナリデータを  $\text{CHR}\$$  関数で文字に変換して PRINT 文で書き込んでいけばよい。そのとき、PRINT 文の末尾にセミコロン(;)を書いて、余分な空白や改行を出力しないようにする。

#### 例 9 (ドキュメントフォルダに TEST.bin を生成する)

```
100 OPTION CHARACTER byte
110 DATA 41, 59, 187, 72
120 OPEN #1: NAME "C:\users\〇〇〇〇\documents\TEST.bin"
130 ERASE #1
140 DO
150   READ IF MISSING THEN EXIT DO: n
160   PRINT #1: CHR$(n);
170 LOOP
180 CLOSE #1
190 END
```

## 2 数と文字列

### 2.1 十進モード

#### 2.1.1 数値の精度

十進モードにおいて、加減乗除は  $10^9$  を基底として計算する。十進数として 19~27 桁の精度を持つ。

#### 例 10 加減乗除の計算精度

```
10 FOR i=0 TO 10
20   PRINT USING "#####.#####":10^i/7
30 NEXT i
40 END
```

実行結果

```
. 1428571428571428571428571428571428570
1. 4285714285714285714285714280000000000
14. 2857142857142857142857142850000000000
142. 85714285714285714285714285700000000000
1428. 57142857142857142857142857100000000000
14285. 71428571428571428571428571400000000000
142857. 14285714285714285714285714200000000000
1428571. 42857142857142857142857142800000000000
14285714. 28571428571428571428571428500000000000
142857142. 857142857142857142857142857000000000000
1428571428. 57142857142857142800000000000000000000
```

Intel x86 系 CPU の 80 ビット浮動小数点演算が利用可能な環境向けのバージョン (Ver.7, ver.8) では、十進モードでの数値変数の精度は 15 桁である。数値式は 16 桁を超える精度を持ち、組込みの無理関数は、17 桁目が偶数に丸められる。一方、ハードウェアでの 2 進浮動小数点演算が 64

ビット（通常、倍精度と呼ばれる）までしか対応しない機種で実行可能なバージョン（Ver. 0.9）では、十進演算での数値変数の精度は 12 桁で、組込みの無理関数は 14 桁目を偶数に丸めている。

#### 例 11 SQR 関数の計算精度と数値変数の精度

```
10 FOR x=2 TO 12
20 LET y=SQR(x)
30 PRINT USING "## ##.#####" "##.#####":x, SQR(x), y
40 NEXT x
50 END
```

実行結果 (Ver. 7.8.7.1)

2	1.4142135623730950	1.4142135623731000
3	1.7320508075688772	1.7320508075688800
4	2.0000000000000000	2.0000000000000000
5	2.2360679774997896	2.2360679774997900
6	2.4494897427831780	2.4494897427831800
7	2.6457513110645906	2.6457513110645900
8	2.8284271247461900	2.8284271247461900
9	3.0000000000000000	3.0000000000000000
10	3.1622776601683794	3.1622776601683800
11	3.3166247903553998	3.3166247903554000
12	3.4641016151377546	3.4641016151377500

実行結果 (Ver.0.9.1.7)

2	1.4142135623730000	1.4142135623700000
3	1.7320508075688000	1.7320508075700000
4	2.0000000000000000	2.0000000000000000
5	2.2360679774998000	2.2360679775000000
6	2.4494897427832000	2.4494897427800000
7	2.6457513110646000	2.6457513110600000
8	2.8284271247462000	2.8284271247500000
9	3.0000000000000000	3.0000000000000000
10	3.1622776601684000	3.1622776601700000
11	3.3166247903554000	3.3166247903600000
12	3.4641016151378000	3.4641016151400000

### 2.1.2 桁落ち

$h=1, 0.1, 0.01, 0.001, 0.0001, \dots$  に対して  $\frac{f(a+h)-f(a)}{h}$  を計算してみたい。 $f(x)$  で定義される関数と定数  $a$  を変えて計算してみる。次の例では、 $f(x)=\sqrt{x}$ 、 $a=2$  としている。

#### 例 12

```
10 DEF f(x)=SQR(x)
20 LET a=2
30 FOR i=1 TO 17
40 LET h=10^(-i)
50 PRINT h, (f(a+h)-f(a))/h
60 NEXT i
70 END
```





と一致する。虚数単位を表す定数を用意していないので、

```
LET i=SQR(-1)
```

を実行して変数  $i$  に虚数単位を割り当てて使うこともできる。

べき乗は、底が正の数るとき、指数を虚数にしてよい。底が虚数るとき、指数は整数に限る。

複素数  $z$  の偏角は  $\text{ARG}(z)$  で求める。角度の単位は  $\text{OPTION ANGLE}$  での設定に従う。

$\text{EXP}(z)$  と  $\text{LOG}(z)$  は  $\text{OPTION ANGLE}$  の設定に影響を受けない。 $\text{LOG}(z)$  の実部は  $\log|z|$ 、虚部は  $\arg z$  (単位はラジアンで  $-\pi \sim \pi$ )。  $\text{ANGLE RADIANS}$  のとき、 $\text{LOG}(z) = \log|z| + \text{ARG}(z)i$  である。

複素数モード専用の組込関数  $\text{RE}(z)$ 、 $\text{IM}(z)$  は、それぞれ、 $z$  の実部、虚部を表す。 $z = \text{RE}(z) + \text{IM}(z)i$  である。また、組込関数  $\text{COMPLEX}(x, y)$  は、 $x, y$  が実数るとき、 $x + yi$  を与える。

複素数値は、実部、虚部の2数の前後を括弧 ( ) で括って出力される。

#### 例 16

```
10 OPTION ARITHMETIC COMPLEX
20 LET i=SQR(-1)
30 PRINT 2^i
40 END
```

実行結果

```
(.769238901363972 .638961276313635)
```

この表記法は、 $\text{INPUT}$  文、 $\text{READ}$  文に対応している。実部、虚部を括弧 ( ) で括るけれど、実部と虚部の間は空白文字で区切ることに注意。

#### 例 17

```
10 OPTION ARITHMETIC COMPLEX
20 DATA (1 2), (2 -3), 4
30 DO
40   READ IF MISSING THEN EXIT DO: z
50   PRINT z
60 LOOP
70 END
```

複素数値を  $a+bi$  の形式で表示したいときは、次のプログラムの 20 行に示すような関数  $\text{CSTR\$}$  を定義しておくといよい。

#### 例 18

```
10 OPTION ARITHMETIC COMPLEX
20 DEF CSTR$(z)=STR$(RE(z)) & " + " & STR$(IM(z)) & " i"
30 LET i=SQR(-1)
40 PRINT CSTR$(2^i)
50 END
```

実行結果

```
.769238901363972 + .638961276313635 i
```

## 2.6 桁あふれの例外処理

Full BASIC は、非数の存在を認めない。下位桁あふれは 0 に丸められるが、絶対値が所定の値 ( $\text{MAXNUM}$ ) より大きいとき、エラーになって、プログラムの進行が止まる。この種類のエラーを例外と呼ぶ。ゼロ除算や桁あふれなどの例外が発生したとき、プログラムを続行させるために、

例外状態処理を用いる。

例外発生が想定される文(複数可)を WHEN EXCEPTION IN 行と USE 行で囲み, USE 行と END WHEN 行の間に例外発生時に実行する文(複数可)を書く。

#### 例 19 例外状態処理

```
10 FOR x=-4 TO 4
20   WHEN EXCEPTION IN
30     PRINT x, 1/x
40   USE
50     PRINT x, "Error"
60   END WHEN
70 NEXT x
80 END
```

## 2.7 文字列変数

### 2.7.1 部分文字列

文字列変数 s\$ の  $i$  番目から  $j$  番目までの部分文字列を  $s$(i:j)$  で表す。

s\$ の  $i$  番目の文字は  $s$(i:i)$  である。

部分文字列への代入を利用して、文字列の置換や挿入、削除ができる。

#### 例 20

```
10 LET s$="123456789"
20 LET t$=s$
30 LET s$(5:4)="abc"
40 LET t$(5:7)=""
50 PRINT s$
60 PRINT t$
70 END
```

実行結果

1234abc56789

123489

## 2.8 文字コード

### 2.8.1 入出力と PRINT TAB, PRINT USING

十進 BASIC Ver.7 は、入出力に Shift\_JIS を用い、内部でも Shift\_JIS で文字を保持している。

十進 BASIC Ver.8 は、入出力に UTF-8 を用い、内部でも UTF-8 で文字を保持している。

PRINT TAB と PRINT USING は内部表現を対象にバイトを単位に動作する。

Shift\_JIS を採用する Ver.7 では半角文字が 1 バイト、全角文字が 2 バイトなので、全角文字と、半角文字を混ぜてもきれいに表示される。

UTF-8 を採用する Ver.8 で Ver.7 と同様の結果を得るためには、全角文字に PRINT TAB を適用する際、BLEN 関数を用いてバイト長を調べ、調整する必要がある。BLEN は、文字列のバイト長を返す独自拡張関数。

### 例 21

```
10 DATA "谷", 78
20 DATA "山本", 92
30 DATA "東海林", 68
40 DO
50   READ IF MISSING THEN EXIT DO: s$, x
60   PRINT s$; TAB(8+BLEN(s$)-2*LEN(s$)); x
70 LOOP
80 END
```

実行結果

```
谷      78
山本    92
東海林  68
```

この例は全角文字のみであることを前提としている。全角と半角が混じるときは、各文字の幅を知る必要がある。とりあえず、英数字と半角カナのみが半角だとすると、次のようになる。

### 例 22

```
10 DECLARE EXTERNAL FUNCTION extra
20 DATA "アイダ", 78
30 DATA "山本", 92
40 DATA "NAKADA", 68
50 DO
60   READ IF MISSING THEN EXIT DO: s$, x
70   PRINT s$; TAB(8+extra(s$)); x
80 LOOP
90 END
100 EXTERNAL FUNCTION extra(s$)
110 LET ex=0
120 FOR i=1 TO LEN(s$)
130   LET t$=s$(i:i)
140   SELECT CASE t$
150     CASE " " TO "~", "." TO "°" ! 英数字, 半角カナ
160       LET ex=ex+BLEN(t$)-1
170     CASE ELSE
180       LET ex=ex+BLEN(t$)-2
190   END SELECT
200   LET extra=ex
210 NEXT i
220 END FUNCTION
```

実行結果

```
アイダ  78
山本    92
NAKADA  68
```

## 2.8.2 OPTION CHARACTER byte

OPTION CHARACTER byte を書いたプログラム単位では、文字列処理の単位をバイトに変更する。保持する文字列そのものに変更はない。

### 例 23

```
10 DECLARE EXTERNAL SUB test
20 LET s$="荒川"
30 FOR i=1 TO LEN(s$)
40   PRINT ORD(s$(i:i));
50 NEXT i
60 PRINT
70 CALL test(s$)
80 END

100 EXTERNAL SUB test(s$)
110 OPTION CHARACTER byte
120 FOR i=1 TO LEN(s$)
130   PRINT ORD(s$(i:i));
140 NEXT i
150 PRINT
160 END SUB
```

実行結果 (Ver. 7)

14675 16494

141 114 144 236

実行結果 (Ver. 8)

33618 24029

232 141 146 229 183 157

実行結果の 1 行目は `OPTION CHARACTER byte` を書かないときの実行結果で、`s$` の 1 文字目と 2 文字目の JIS コードまたはユニコードである。2 行目は `s$` の内部で保持されているバイト列そのもの (すなわち、`shift_JIS` または `UTF-8`) である。

逆に、`OPTION CHARACTER byte` を書いたプログラム単位では、`shift-JIS` あるいは、`UTF-8` から文字列を生成できる。

### 例 24 (ver. 8.1.3.7)

```
10 OPTION CHARACTER byte
20 DATA 232, 141, 146, 229, 183, 157
30 LET s$=""
40 DO
50   READ IF MISSING THEN EXIT DO: n
60   LET s$ = s$ & CHR$(n)
70 LOOP
80 PRINT s$
90 END
```

## 3 配列

### 3.1 DIM 文

#### 3.1.1 実行時に配列の大きさを定める

次のプログラムは、エラトステネスの篩によって  $n$  以下の素数を求める。

```
100 INPUT n
110 DIM s(n)
120 MAT s=ZER
130 FOR i=2 TO n
140   IF s(i)=0 THEN
150     PRINT i
160     FOR j=i^2 TO n STEP i
170       LET s(j)=1
180     NEXT j
190   END IF
200 NEXT i
210 END
```

Full BASIC では配列の大きさは翻訳時に定まり、DIM 文で配列の大きさを変数で指定することができない。110 行の DIM 文は十進 BASIC 独自拡張である。

十進 BASIC でこの形の DIM 文を書くと、翻訳時には  $s$  を配列名として認識する。そして、実行時には、大きさを 0 にする指定を素通りし、正の大きさが指定されたときメモリーを確保する。ただし、一旦、配列の大きさが定まると、それ以後、配列の大きさを拡張することはできない。たとえば、

```
10 FOR n=0 TO 2
20   DIM a(n)
30   PRINT n, SIZE(a)
40 NEXT n
50 END
```

を実行すると、 $n=0$  で 20 行を実行してもエラーにならず、 $n=1$  で 20 行を実行したとき配列  $a$  の大きさを 1 に設定する。そして、 $n=2$  のとき 20 行を実行するとエラーになる。

#### 3.1.2 MAT REDIM

MAT REDIM は Full BASIC 規格外だけれど、Full BASIC の原型となった True BASIC にある命令である。MAT REDIM 文を用いると、最初に定めた大きさを超えない範囲で配列の上限と下限を変更することができる。このとき、配列の要素は変更されない。たとえば、

```
10 DATA 1, 2, 3, 4
20 DIM a(4)
30 MAT READ a
40 MAT PRINT a
50 MAT REDIM a(0 to 2)
60 MAT PRINT a
70 MAT REDIM a(4)
80 MAT PRINT a
```

90 END

では、30 行の MAT READ 文の実行によって

$a(1)=1, a(2)=2, a(3)=3, a(4)=4$

となり、50 行の MAT REDIM 文の実行で

$a(0)=1, a(1)=2, a(2)=3$

となる。この状態で、配列 a の 4 個目の要素 4 は宙に浮いた状態になる。そして、70 行の MAT REDIM 文の実行で元にもどされ、

$a(1)=1, a(2)=2, a(3)=3, a(4)=4$

となる。

### 3.1.3 2次元配列の入力

MAT PRINT 文による 2 次元配列の出力は、1 行出力ごとに改行が挿入されるが、MAT INPUT 文、MAT READ 文による 2 次元配列への入力では行方向への入力を行うが、次行への継続に特別な仕組みがなく、単に、連続的な値の列で行う。たとえば、次の 3 つのプログラムはまったく同じ効果を持つ。

10 DATA 1, 2	10 DATA 1, 2, 3	10 DATA 1, 2, 3, 4, 5, 6
20 DATA 3, 4	20 DATA 4, 5, 6	
30 DATA 4, 5		
40 DIM a(2, 3)	40 DIM a(2, 3)	40 DIM a(2, 3)
50 MAT READ a	50 MAT READ a	50 MAT READ a
60 MAT PRINT a	60 MAT PRINT a	60 MAT PRINT a
70 END	70 END	70 END

出力は、いずれも

1	2	3
4	5	6

であって、これは、

$a(1,1)=1, a(1,2)=2, a(1,3)=3, a(2,1)=4, a(2,2)=5, a(2,3)=6$

を意味する。

なお、DATA 文では行末にコンマを書かないけれども、同様の MAT INPUT 文で、入力の途中で改行する場合は、末尾にコンマが必要になる（最終行を除く）。

### 3.1.4 MAT REDIM (2次元)

上の例で、50 行の前、または、後で MAT REDIM a(3, 2) を実行すると、出力は

1	2
3	4
5	6

になる。この結果は、

$a(1,1)=1, a(1,2)=2, a(2,1)=3, a(2,2)=4, a(3,1)=5, a(3,2)=6$

と入力されることを意味する。

なお、配列全体の要素数を増やさないかぎり MAT REDIM 文で各次元の大きさを変えることができるけれども、1 次元配列を 2 次元配列に変えるなど、次元自体を変えることはできない。

## 3.2 行列の計算

十進 BASIC の最新版では、MAT 文に行列演算の数式を書くことができる。行列の加減乗除と整数べきの他、TRN 関数（転置行列）、INV 関数（逆行列）が利用できる。

ただし、行列の計算式が書けるのは、MAT 文と、DET 関数（行列式）の引数としてのみである。

たとえば、

```
DIM a(3,3),b(3,3)
```

```
MAT b=trn(a)*a-a*trn(a)
```

として、**b** が零行列かどうかを調べれば **a** が直交行列であるかどうかを判定できる。

けれども、IF 文などで条件として配列全体の比較を書くことができない。そこで、ある 2 次元配列が零行列かどうかを判定する外部関数定義を用意する。

```
EXTERNAL FUNCTION IsZer(a(,))
```

```
LET t=1
```

```
FOR i=LBOUND(a,1) TO UBOUND(a,1)
```

```
  FOR j= LBOUND(a,2) TO UBOUND(a,2)
```

```
    IF a(i,j)<>0 THEN LET t=0
```

```
  NEXT j
```

```
NEXT i
```

```
LET IsZer=t
```

```
END FUNCTION
```

上掲の外部関数 IsZer(a(,))は、引数として指定した 2 次元配列 **a** が零行列であるとき 1、そうでないとき 0 を返す。この関数を用いると、

```
DECLARE EXTERNAL FUNCTION IsZer
```

```
IF IsZer(b)=1 THEN PRINT "直交"
```

のようにしてある行列が直交行列かどうか判定できる。

なお、Full BASIC では、関数を取る値は数値または文字列に限られ、真偽値や配列値と取る関数を定義することができない。

### 3.2.1 列の取り出しと行への代入

Full BASIC では、2 次元配列から 1 行のみを取り出したり、1 行のみを入れ替えるような操作が簡単にはできない。十進 BASIC は独自拡張として、行列の行を操作する命令を用意している。

行の抽出（独自拡張）

**A** を 2 次元数値配列（計算式は不可）、**m,n** を数値式とする。

行列の計算式で以下の関数を用いることができる。

ROW(**A,n**)            **A** の第 **n** 行を抽出した 1 次元配列

ROW(**A,m:n**)        **A** の第 **m** 行から第 **n** 行までを抽出した 2 次元配列

行、または、列への部分代入（独自拡張）

**A** を 2 次元数値配列、**m,n** を数値式とする。

MAT ROW(**A,n**)=行列の計算式（1 次元）    **A** の第 **n** 行を右辺の値で置き換える。

MAT ROW(**A,m:n**)=行列の計算式（2 次元）   **A** の第 **m** 行から第 **n** 行を右辺の値で置き換える。

どちらの場合も、列数が一致していなければならない。

Note .

MAT ROW(**A,m:n**) で **n-m+1** が 右辺の計算結果の行数と異なるとき、**A** の行数は増減する。

MAT ROW(**A,n+1:n**)=行列の計算式(2次元)    を実行すると、第 **n** 行の直後に右辺の計算結果の行

列が挿入される。A の行数が増加する MAT 文を実行するためには、A のサイズを大きく宣言しておき、MAT ZER 文などで縮小してから使う。

例

```
DIM A(4, 2), B(2, 2)
MAT A=ZER(2, 2)
MAT ROW(A, 3:2)=B
```

MAT ROW(A,m:n) =B において B は行数 0 でもよいので、行削除に使える。このとき、B の列数は A と一致させておく。

例

```
DIM A(3, 3), B(3, 3)
MAT B=ZER(0, 3)
MAT ROW(A, 2:2)=B
```

を実行すると、A の第 2 行が削除される。

例 行削除, 列削除

2 次元数値配列 a の第 i 行を削除する外部副プログラムと、第 j 列を削除する副プログラムが次のように書ける。

```
EXTERNAL SUB RowDelete(a(,), i)
DIM b(0, ubound(a, 2))
MAT ROW(a, i:i)=b
END SUB
EXTERNAL SUB ColumnDelete(a(,), j)
DIM b(ubound(a, 1), 0)
MAT Column(a, j:j)=b
END SUB
```

行の交換

次のような副プログラムを用意すると、行の入れ替え（交換）ができる。

```
DIM A(3, 4)
CALL SwapRow(A, 2, 3)
END
EXTERNAL SUB SwapRow(A(,), i, j) ! A の i 行と j 行を交換する
DIM u(lbound(A, 2) to ubound(A, 2))
DIM v(lbound(A, 2) to ubound(A, 2))
MAT u=row(A, i)
MAT v=row(A, j)
MAT row(A, i)=v
MAT row(A, j)=u
END SUB
```

### 3.2.2 列の操作

MAT COLUMN(A,m:n) = 行列の計算式 (2 次元)

A の第 m 列から第 n 列を右辺の値で置き換える。

MAT COLUMN(A,n) = 行列の計算式 (1次元)                    A の第 n 列を右辺の値で置き換える。  
 COLUMN(A,n)    A の第 n 列を抽出した 1次元配列  
 COLUMN(A,m:n)    A の第 m 列から第 n 列までを抽出した 2次元配列

注意.

COLUMN(A,n) で得られる 1次元配列は意味的には列ベクトルであるが、文法的は通常の 1次元ベクトルと区別されない。

列への代入のとき、行数が一致していなければならない。

## 4 制御構造

### 4.1 制御構造

#### 4.1.1 IF~ELSEIF~ELSE~END IF

例 25    2次方程式  $ax^2+bx+c=0$  の解を求める。

```
100 INPUT a, b, c
110 LET D=b^2-4*a*c
120 IF D>0 THEN
130   PRINT (-b+SQR(D))/(2*a), (-b-SQR(D))/(2*a)
140 ELSEIF D=0 THEN
150   PRINT -b/(2*a)
160 ELSE
170   PRINT "解なし"
180 END IF
190 END
```

IF~END IF では、IF 行、ELSEIF 行、ELSE 行、END IF 行がそれぞれ独立した一行に書かれる。

IF~END IF に ELSEIF を複数行書いてもよい。一度、条件を満たすと以後は実行されない。たとえば、上の例で 140 行を

```
140 ELSEIF D>=0 THEN
```

と変えても、 $D>0$  のとき、150 行が実行されることはない。

#### 4.1.2 IF 文

IF ... THEN に続けて文を書くと、IF 文になる。その場合、END IF を書かない。

IF 文は

IF 条件 THEN 単純実行文

または

IF 条件 THEN 単純実行文 ELSE 単純実行文

の形で一行で書かれる。THEN, および ELSE に続けて書くことのできる文は一つに限る。

単純実行文は、構造文と IF 文以外の実行文である。IF 文を単純実行文から除外するのは、

IF ... THEN IF --- THEN ~ ELSE ~

の形の IF 文が生成されることがないようにするためである。

#### 4.1.3 DO~LOOP

DO 行で始まり LOOP 行で終わる部分が繰り返し実行される。繰り返しを終える条件は DO 行

にも LOOP 行にも書ける。両方に書いてもよい。DO 行に条件を書いたとき、繰り返しの条件に合致しなければ LOOP 行の次に分岐する。LOOP 行に書いたときは、繰り返しの条件に合致すると DO 行に戻る。

繰り返し条件は、次のいずれかの形に書く。

UNTIL 条件

WHILE 条件

UNTIL 条件 を書いたとき、条件が成立すると繰り返しを抜ける。

WHILE 条件 を書くと、条件が成立しなくなると繰り返しを抜ける。

#### 4.1.4 EXIT DO

DO 行と LOOP 行の間に EXIT DO 文を書くことができる。EXIT DO を実行すると、LOOP 行の次に分岐する。DO~LOOP が多重に入れ子になっているときは、もっとも内側（近い側）のループから抜ける。EXIT DO 文は単純実行文なので、  
IF 条件 THEN EXIT DO  
の形に書くことができる。

#### 例 26 2 数 $a$ , $b$ の最大公約数を求めるユークリッド互除法

互除法は、次のように説明されることが多い。

- ①  $a$  を  $b$  で割って余り  $r$  を求める。
- ② 余り  $r$  が 0 でないとき、除数を被除数、余りを除数として①に戻る。
- ③  $r=0$  のときの除数  $b$  が最大公約数。

これを素直に記述したいとき、EXIT DO が必要になる。

```
10 INPUT a,b
20 DO
30 LET r=MOD(a,b)
40 IF r=0 THEN EXIT DO
50 LET a=b
60 LET b=r
70 LOOP
80 PRINT b
90 END
```

#### 4.1.5 論理式

Full BASIC で実行可能な論理演算は、NOT, AND, OR のみで、exclusive OR はない。

NOT は、否定したい論理演算の直前に書く。論理演算の適用順は括弧 ( ) で指定できる。括弧を省いたときの優先順位は、NOT, AND, OR の順。たとえば、NOT  $p$  OR  $q$  は、(NOT  $p$ ) OR  $q$  を意味する。

Full BASIC の AND, OR は短絡評価を行う。「 $p$  AND  $q$ 」で、 $p$  が偽であると、 $q$  を評価しない。また、「 $p$  OR  $q$ 」で  $p$  が真であれば  $q$  を評価しない。たとえば、 $x=0$  のとき、  
IF  $x \leq 0$  OR LOG( $x$ ) < 1 THEN .....  
を実行しても例外状態にならない。

#### 4.1.6 CAUSE EXCEPTION

Full BASIC では、例外状態処理を制御構造の一部として利用できる。CAUSE EXCEPTION 文で

独自の例外を生成することができる。利用者が定める例外には、1～999の例外番号を用いる。例外番号は、USEブロックでEXTYPE関数で参照できる。

繰り返しが途中で抜けたときと最後まで実行した場合で異なる処理が必要になるとき、例外を利用すると書きやすくなることがある。

#### 例 27 素数／合成数の判定

```
100 INPUT n
110 WHEN EXCEPTION IN
120   FOR i=2 TO SQR(n)
130     IF MOD(n,i)=0 THEN CAUSE EXCEPTION 999
140   NEXT i
150   PRINT n;"は素数"
160 USE
170   PRINT n;"は合成数",i;"が因数"
180 END WHEN
190 END
```

USEブロックで、EXTYPE関数で生成された例外の種別を知ることができる。通常のエラーが起りえる状況では、EXTYPEの値によって処理を分けることができる。上の例だと、

```
170   IF EXTYPE=999 THEN
171     PRINT n;"は合成数",i;"が因数"
172   ELSE
173     CAUSE EXCEPTION EXTYPE
174   END IF
```

のようにUSEブロックを書くと、999以外の例外が発生したときは通常の例外として処理される。

## 4.2 関数

### 4.2.1 <sup>ひきすう</sup>引数

関数定義や副プログラムで関数名・副プログラム名に続く括弧内に書かれる変数を仮引数という。一方、それらを利用するときに関数名・副プログラムに続く括弧内に書かれる値を実引数という。両者をまとめて<sup>ひきすう</sup>引数という。

例

```
10 DEF f(x)=a*x^2+b*x+c   ← xは仮引数
20 PRINT f(1)             ← 1が実引数
30 END
```

例

```
10 SUB s(a)               ← aは仮引数
20   LET a=10
30 END SUB
40 CALL s(b)             ← bは実引数
50 END
```

仮引数は変数であるが、実引数は数値式である。実引数は変数のことも数値定数のこともある。

関数定義や副プログラム内で仮引数は外部に同名の変数があったとしても別の変数である（数値の記憶場所として異なる）。

実引数として変数を書いたとき、関数定義と副プログラムとではその挙動が異なる。

関数定義の実引数に変数を書くと、実行時、その値が仮引数に代入される。

副プログラムの実引数に変数を書くと、仮引数は実引数に書かれた変数と数値の記憶場所を共有する。したがって、

関数定義で実引数として書いた変数が変更されることはない。

副プログラムに実引数として変数を書くと、変数の値が変更されることがある。

#### 4.2.2 DEF 文

DEF 文はプログラム単位（主プログラム、外部関数定義、外部副プログラム、外部絵定義）内  
にのみ書ける。引数は DEF 文内でのみ有効な局所変数である。ただし、引数以外の変数はプログラム単位内で共通。DEF 文で定義した関数をプログラム単位外から利用することはできない。

##### 例 28

```
10 DEF f(x)=SIN(x)+2*COS(x)
20 LET x=10
30 PRINT f(2)
40 PRINT x
50 END
```

実行結果

```
7.70037537313969E-2
10
```

10 行の x と 20~40 行の x は別の変数。だから、30 行で f(x) の x に 2 を代入しても 20~40 行の x の値は変化しない。

#### 4.2.3 内部関数定義

内部関数定義は、プログラム単位の内部に書かれる関数定義で、引数は関数定義内でのみ有効な局所変数。それ以外の変数はプログラム単位と共有する。関数値は、関数名に対する LET 文で決定する。関数定義中でその関数名は変数として機能しない。たとえば、例 22 では、関数値を計算するのに関数名 extra と異なる変数名 ex を用いている。

##### 例 29 (好ましくない使い方)

```
100 FUNCTION GCD(a, b)
110   DO UNTIL b=0
120     LET r=MOD(a, b)
130     LET a=b
140     LET b=r
150   LOOP
160   LET GCD=a
170 END FUNCTION
180 LET r=10
190 PRINT GCD(45, 12)
200 PRINT r
210 END
```

実行結果

```
3
0
```

120~140 行の r は主プログラム共通。だから、190 行で GCD 関数を呼び出すと、120 行を実行し

たとき主プログラムの変数  $r$  の値が変化してしまう。関数定義内で作業用の変数が必要な場合は、外部関数定義を用いるのが基本。ただし、プログラム単位の変数の参照も必要だけれど局所変数も必要だという場合には、後述のモジュールの機能を使うのが正攻法だけれども、小規模なプログラムであれば、True BASIC 互換のために用意した独自拡張の LOCAL 文を使うこともできる。

#### 例 30 (LOCAL)

```
100 FUNCTION GCD(a, b)
105   LOCAL r
110   DO UNTIL b=0
120     LET r=MOD(a, b)
130     LET a=b
140     LET b=r
150   LOOP
160   LET GCD=a
170 END FUNCTION
180 LET r=10
190 PRINT GCD(45, 12)
200 PRINT r
210 END
```

実行結果

```
3
10
```

#### 4.2.4 関数の再帰的定義

関数定義において定義しようとする関数を用いてよい。そのとき、内部関数定義では引数と LOCAL 文で宣言した変数は、呼び出しごとに新たなメモリーに割り当てられ、実行後、元に戻る。

#### 例 31 最大公約数

```
10 FUNCTION GCD(a, b)
20   IF b=0 THEN
30     LET GCD=a
40   ELSE
50     LET GCD=GCD(b, MOD(a, b))
60   END IF
70 END FUNCTION
80 PRINT GCD(45, 12)
90 END
```

#### 4.2.5 外部関数定義

外部関数定義は、主プログラムの最後の行 (END 行) 以降に書かれる。外部関数定義では、すべての変数は局所変数である。主プログラムと変数を共有することはない。また、変数は、呼び出しごとに確保され、実行が終わると消滅する。

#### 例 32 最大公約数 (外部関数定義)

```
10 DECLARE EXTERNAL FUNCTION GCD
20 PRINT GCD(45, 12)
30 END
100 EXTERNAL FUNCTION GCD(a, b)
```

```

110 LET r=MOD(a, b)
120 IF r=0 THEN
130   LET GCD=b
140 ELSE
150   LET GCD=GCD(b, r)
160 END IF
170 END FUNCTION

```

この例のように、引数にない変数を関数定義内で用いるときは、外部関数定義にする。

外部関数を利用するプログラム単位には、10行に示すような宣言文を書く。関数名のみで引数部を書かない。組込関数名と一致する関数を定義したとき、それが外部関数であることを明示的に指示する。たとえば、十進 BASIC の有理数モードで GCD は組込関数なので、次のプログラムを実行すると外部関数の GCD は実行されない。10行の！を除去したときとで、結果を比較してみると DECLARE EXTERNAL FUNCTION 文の必要性が理解できるだろう。

### 例 33 DECLARE EXTERNAL FUNCTION 文の必要性

```

5 OPTION ARITHMETIC RATIONAL
10 ! DECLARE EXTERNAL FUNCTION GCD
20 PRINT GCD(45, 12)
30 END
100 EXTERNAL FUNCTION GCD(a, b)
105 PRINT a, b
110 LET r=MOD(a, b)
120 IF r=0 THEN
130   LET GCD=b
140 ELSE
150   LET GCD=GCD(b, r)
160 END IF
170 END FUNCTION

```

## 4.2.6 値渡し

関数定義において、仮引数は他の変数と異なる存在である。たとえば、関数 square が何回利用された知りたいと考えて、以下のようなプログラムを書いても目的は果たせない。

```

100 FUNCTION square(x, n)
110   LET square=x^2
120   LET n=n+1
130 END FUNCTION
140 LET n=0
150 FOR x=0 TO 10
160   PRINT square(x, n)
170 NEXT x
180 PRINT n
190 END

```

180行の実行結果は0となる。どこから呼ばれたかを考えなくてもよいのであれば、100行でnを引数から除外すればよいが、どこから呼ばれたかを調べたいときには、副プログラムを利用する。

## 4.3 副プログラム

### 4.3.1 参照渡し

関数定義では、引数はかならず局所変数であるが、副プログラムと絵定義において、実引数に変数を書いたとき、仮引数と実引数は数値の記憶場所を共有する。これを参照渡しという。

#### 例 34 変数渡し

```
10 SUB increm(a)
20   LET a=a+1
30 END SUB
40 LET x=1
50 LET y=10
60 CALL increm(x)
70 CALL increm(y)
80 PRINT x,y
90 END 副プログラム increm は、引数にした変数の値に 1 を加算する。
```

#### 例 35 引数は局所変数

副プログラム定義で引数リスト ( ) 内に書かれた変数は、副プログラム定義内で、主プログラム中の同名の変数と区別される。たとえば、次のプログラムで 10 行と 20 行の a は同じ変数を指すが、40 行以降の a は 10 行の a とは異なる変数。

```
10 SUB increm(a)
20   LET a=a+1
30 END SUB
40 LET a=1
50 LET b=10
60 CALL increm(a)
70 CALL increm(b)
80 PRINT a,b
90 END
```

#### 例 36 Alias(別名)

副プログラムの実引数が副プログラムのなかでも有効になっているとき、副プログラム内で見かけ上異なる 2 つの変数が実際には同一変数だということが起こる。これを alias(別名)という。

次のプログラムで 20 行、30 行の変数 a, x は同じ変数である。

```
10 SUB s(a)
20   LET a=a+1
30   LET x=x+1
40 END SUB
50 LET x=0
60 CALL s(x)
70 PRINT x
80 END
出力結果 2
```

#### 例 37

実引数を書くとき、変数名を括弧で括ると値渡しになる。

```
10 LET a=0
20 CALL s(a, (a))
```

```

30 SUB s(b, c)
40   LET a=1
50   LET b=2
60   LET c=3
70   PRINT a;b;c
80 END SUB
90 END
実行結果
  2  2  3

```

BASIC では、複数の値を持つ関数を定義できない。なので、複数の値を同時に得たい場合には、副プログラムの変数に値を代入して返すことで実現する。

**例 38 割り算の商と余りを求める。**

```

10 SUB divide(a, b, q, r)
20   LET q=INT(a/b)
30   LET r=MOD(a, b)
40 END SUB
50 CALL divide(18, 5, q, r)
60 PRINT q, r
70 END

```

#### 4.3.2 外部副プログラム

内部副プログラムでは、引数と LOCAL 文で宣言した変数以外はプログラム単位で保持される。これは、再帰アルゴリズムを記述するとき障壁になる。外部副プログラムでは、CALL 文で変数を与えられた引数以外の変数はすべて再帰呼び出しに対応する局所変数である。

**例 39 一次不定方程式**

整数  $a, b$  を入力すると  $ax+by=1$  となる整数  $x, y$  を求める副プログラムを作る。

$b=0$  のとき、 $a=1$  なら  $x=1, y$  は任意 が解で、 $a \neq 1$  のときは解がない。

以後、 $b \neq 0$  とする。

$a$  を  $b$  で割った商を  $q$ , 余りを  $r$  とすると、 $a=bq+r, 0 \leq r < b$  なので、求める方程式を、

$(bq+r)x+by=1$ , すなわち、 $b(qx+y)+rx=1$  と変形できる。

$u=qx+y, v=x$  とおくと、 $bu+rv=1$ 。

$bu+rv=1$  となる  $u, v$  を求めれば  $x=v, y=u-qv$  から  $x, y$  が定まる。

互除法の割算を繰り返すといつかは割り切れて  $r=0$  になるので、上の操作は有限回で終了する。

この操作を副プログラムで表す。このプログラムは、解のうちの一組を求める。

```

10 DECLARE EXTERNAL SUB solve
20 INPUT a, b
30 CALL solve(a, b, x, y)
40 PRINT x, y
50 END
100 EXTERNAL SUB solve(a, b, x, y)
110 IF b=0 THEN
120   IF a=1 THEN
130     LET x=1

```

```

140     LET y=0    ! 他の数を指定してもよい
150 ELSE
160     PRINT "解なし"
170 END IF
180 ELSE
190     LET q=INT(a/b)
200     LET r=MOD(a, b)
210     CALL solve(b, r, u, v)
220     LET x=v
230     LET y=u-q*v
240 END IF
250 END SUB

```

## 4.4 モジュール

### 4.4.1 モジュールとは

外部関数定義や外部副プログラムでは、副プログラムで変数引数にした場合を除いて、手続き定義内の変数は、手続きの終了後には消えてしまう。これは、前回実行時の情報をもとに動作する手続きを作るのに具合が悪い。外部関数定義や外部副プログラムなどの外部手続きに実行後も残る変数を使えるようにする枠組みがモジュールである。

#### 例 40 相対グラフィックス（タートルグラフィックス）

相対グラフィックス（タートルグラフィックス）とは、進行方向を現在の向きを基準に右、あるいは、左に何度向きを変えるか、そして、その向きにどれだけ進むかを指示して図形を描く描画システムのことである。相対グラフィックスを実現するためには、現在の進行方向と現在位置が記憶されることが不可欠である。

```

100 DECLARE EXTERNAL SUB Turtle.RightTurn, Turtle.LeftTurn
110 DECLARE EXTERNAL SUB Turtle.Forward, Turtle.PenUp, Turtle.PenDown
120 SET WINDOW -1, 3, -2, 2
130 FOR i=1 TO 8
140     CALL PenDown
150     CALL Forward(1)
160     CALL RightTurn(45)
170 NEXT i
180 END
1000 MODULE turtle
1010 MODULE OPTION ANGLE DEGREES
1020 PUBLIC SUB RightTurn, LeftTurn, Forward, PenUp, PenDown
1030 SHARE NUMERIC CurPosX, CurPosY, Pen, theta
1040 REM 初期値を設定
1050 LET CurPosX=0
1060 LET CurPosY=0
1070 LET Pen=1    ! Pen Down
1080 LET theta=90 ! 上向き
1090 EXTERNAL SUB RightTurn(a)
1100     LET theta=theta-a
1110 END SUB

```

```

1120 EXTERNAL SUB LeftTurn(a)
1130   LET theta=theta+a
1140 END SUB
1150 EXTERNAL SUB PenUp
1160   LET Pen=0
1170 END SUB
1180 EXTERNAL SUB PenDown
1190   LET Pen=1
1200 END SUB
1210 EXTERNAL SUB Forward(r)
1220   LET NewPosX=CurPosX+r*COS(theta)
1230   LET NewPosY=CurPosY+r*SIN(theta)
1240   IF Pen=1 THEN PLOT LINES:CurPosX, CurPosY; NewPosX, NewPosY
1250   LET CurPosX=NewPosX
1260   LET CurPosY=NewPosY
1270 END SUB
1280 END MODULE

```

進行方向の初期値は上向きである。**RightTurn(*t*)**を呼び出すと右に  $t^\circ$  向きを変え、**Forward(*r*)**を呼び出すとその方向に *r* だけ進む。ペンを降ろした状態で動くと軌跡が描かれる。

モジュールは **MODULE** 行で始まり **END MODULE** 行で終わる。**MODULE** 行にはモジュール名を書く。モジュールには、外部から参照可能な変数、関数定義、副プログラムなどと、モジュール内でのみ有効な変数、関数定義、副プログラムなどからなる。外部から参照可能な変数、関数、副プログラムなどは 1020 行に示すような **PUBLIC** 文を書く。一方、モジュール内で利用可能な変数や関数には、1030 行のような **SHARE** 文を書く。モジュール内だけでも、どの外部手続きにも属さない部分、上の例で 1010~1080 行をモジュール本体という。モジュール本体は、プログラム開始後、主プログラムよりも先に実行される。

モジュール内の関数、副プログラムは、外部関数、外部副プログラムとして記述する。**PUBLIC** 変数や **SHARE** 変数はモジュール全体を通して有効である。上の例では、**CurPosX**, **CurPosY** は現在地、**Pen** はペンの状態、**theta** は進む向き（角度）を保存している。

1010 行に示すように、モジュール全体で有効な **OPTION** 文を書くことができる。上の例では、モジュール全体で角の大きさの単位が度(degrees)になる。

モジュールで定義された変数や手続きを利用するプログラム単位には、100 行、110 行にあるような **DECLARE EXTERNAL** 文を書く。**DECLARE EXTERNAL** 文で宣言された手続きを利用するとき、140~160 行にあるように、モジュール名. を省いて書くことができる。

## 5 グラフィックス

### 5.1 問題座標

#### 5.1.1 SET WINDOW

Full BASIC では、描画領域の初期値は正方形であり、左下隅が(0,0)、右上隅が(1,1)の座標系が設定されている。座標系の設定を変えるのに **SET WINDOW** 文を用いる。

**SET WINDOW** 文には 4 つのパラメータをコンマで区切って指定する。それらは、順に、左端 *x*

座標, 右端  $x$  座標, 下端  $y$  座標, 上端  $y$  座標を意味する。要するに, はじめの 2 個が  $x$  座標の範囲, 続く 2 個が  $y$  座標の範囲である。

### 5.1.2 PLOT POINTS

PLOT POINTS 文は, 指定した座標に点を描く。座標は, PLOT POINTS: に続けて,  $x$  座標,  $y$  座標をコンマで区切って書く。

描かれる点の形を SET POINT STYLE 文で設定する。点の形状は以下の数値で指定する。

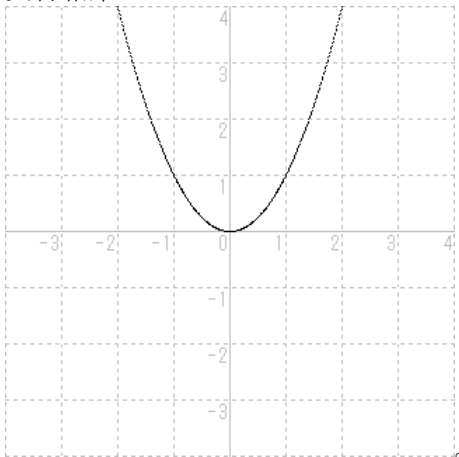
1 · 2 + 3 \* 4 ○ 5 × 6 ■ 7 ●

プログラム実行開始時の点の形状は 3 \* である。細かく点を取ってグラフを描くときは, 1 番の点を用いるように SET POINT STYLE 1 を実行しておく必要がある。

#### 例 41 関数のグラフを描く

```
10 DEF f(x)=x^2
20 SET WINDOW -4, 4, -4, 4
30 DRAW grid
40 SET POINT STYLE 1
50 FOR x=-4 TO 4 STEP 0.01
60 PLOT POINTS: x, f(x)
70 NEXT x
80 END
```

実行結果



### 5.1.3 PLOT LINES

PLOT LINES 文を実行すると, 描点が移動する。PLOT LINES 文は以下の形に書く。

PLOT LINES:  $x$  座標,  $y$  座標

PLOT LINES:  $x$  座標,  $y$  座標;

末尾にセミコロンを書かない PLOT LINES 文を実行すると描点が OFF になり, 末尾にセミコロンを書いた PLOT LINES 文を実行すると描点が ON になる。描点が ON の状態で描点を移動すると, 軌跡が描かれる。プログラム実行開始時の描点の状態は OFF なので, 最初に実行する PLOT LINES 文は線を描かない。

#### 例 42

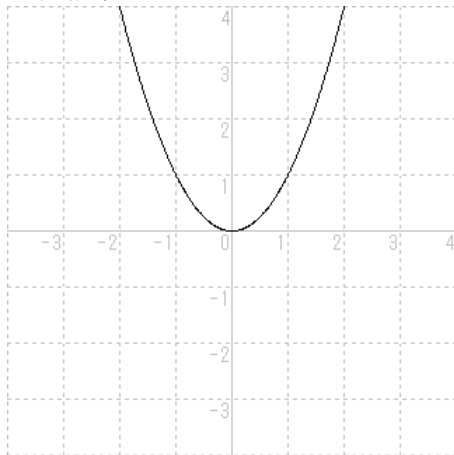
```
10 DEF f(x)=x^2
20 SET WINDOW -4, 4, -4, 4
30 DRAW grid
```

```

40 FOR x=-4 TO 4 STEP 0.01
50   PLOT LINES: x, f(x);
60 NEXT x
70 END

```

実行結果



#### 5.1.4 LINE STYLE

SET LINE STYLE 文を利用して描かれる線の形状を変えることができる。次の番号で指定する。

1 実線 2 破線 3 点線 4 一点鎖線 5 二点鎖線

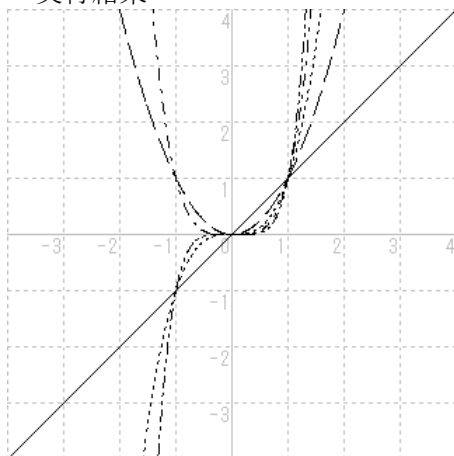
例 43

```

100 SET WINDOW -4, 4, -4, 4
110 DRAW grid
120 FOR n=1 TO 5
130   SET LINE STYLE n
140   FOR x=-4 TO 4 STEP 0.01
150     PLOT LINES: x, x^n;
160   NEXT x
170   PLOT LINES
180 NEXT n
190 END

```

実行結果



170 行にあるように、座標を指定しない PLOT LINES 文は、描点を OFF にするために用いられる。

#### 例 44 POINT COLOR と LINE COLOR

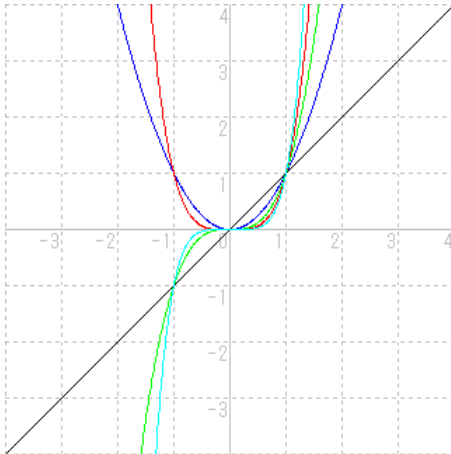
SET POINT COLOR, SET LINE COLOR で点や線の色を変えることができる。プログラム実行開始時の色指標は 1。色指標に対応する色は、プログラムで変えることもできるが、プログラム実行開始時には次のようになっている。

0 白, 1 黒, 2 青, 3 緑, 4 赤, 5 水色, 6 黄色, 7 赤紫,  
8 灰色, 9 濃い青, 10 濃い緑, 11 青緑, 12 えび茶, 13 オリーブ色, 14 濃い紫, 15 銀色, …

#### 例 45

```
100 SET WINDOW -4, 4, -4, 4
110 DRAW grid
120 FOR n=1 TO 5
130   SET LINE COLOR n
140   FOR x=-4 TO 4 STEP 0.01
150     PLOT LINES: x, x^n;
160   NEXT x
170   PLOT LINES
180 NEXT n
190 END
```

実行結果

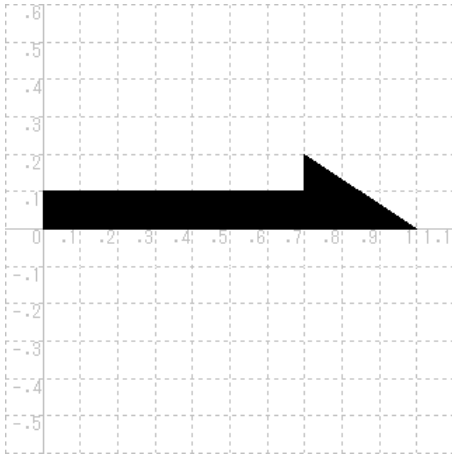


#### 5.1.5 PLOT AREA

PLOT AREA 文は、指定した点を順に結んでできる図形の内側を塗りつぶす。点は  $x$  座標,  $y$  座標をコンマ(,)で区切って書き, 点と点はセミコロンで区切って書く。最後に指定した点と最初に指定した点とを結ぶ線は自動的に追加される。

```
10 DECLARE EXTERNAL PICTURE arrow
20 SET WINDOW -0.1, 1.1, -0.6, 0.6
30 DRAW grid(0.1, 0.1)
40 DRAW arrow
50 END
100 EXTERNAL PICTURE arrow
110 PLOT AREA : 0, 0; 0, 0.1; .7, 0.1; .7, 0.2; 1, 0
120 END PICTURE
```

実行結果



## 5.2 絵定義

### 5.2.1 絵定義

絵定義は、副プログラムとほぼものである。PICTURE ~ END PICTURE で動作を定義し、DRAW 文で呼び出す。DRAW 文に WITH 句を書いて変形を指示できることが副プログラムと異なる。

#### 例 46

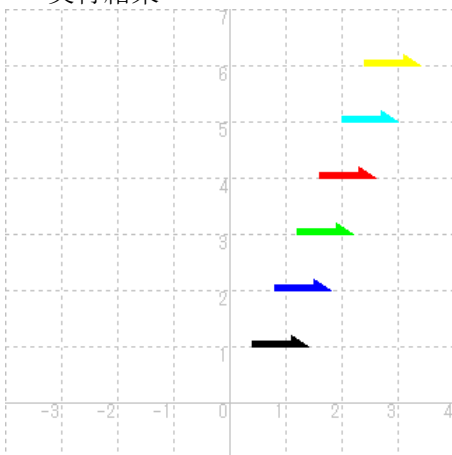
```

10 DECLARE EXTERNAL PICTURE arrow
20 SET WINDOW -4, 4, -1, 7
30 DRAW grid
40 FOR i=1 TO 6
50   SET AREA COLOR i
60   DRAW arrow WITH SHIFT(0.4*i, i)
70 NEXT i
80 END

100 EXTERNAL PICTURE arrow
110 PLOT AREA : 0, 0; 0, 0.1; .7, 0.1; .7, 0.2; 1, 0
120 END PICTURE

```

実行結果



### 5.2.2 絵定義の変形

DRAW 文の WITH 句には、SHIFT, ROTATE, SCALE, SHAER による変形を指示できる。

SHIFT( $a, b$ )は,  $x$  軸方向に  $a$ ,  $y$  軸方向に  $b$  の平行移動,

ROTATE( $\theta$ )は, 原点を中心とする  $\theta$  の回転, 角の単位は OPTION ANGLE の指定に従う,

SCALE( $r$ )は, 原点を中心とする  $r$  倍の拡大,

SCALE( $a, b$ )は, 原点を中心とする  $x$  軸方向に  $a$  倍,  $y$  軸方向に  $b$  倍の拡大。

SHEAR( $a$ )は, 剪断  $(x, y) \mapsto (x+y \tan a, y)$

SCALE( $a, b$ )は,  $(x, y) \mapsto (ax, by)$  で定義される。 $a, b$  に負数を指定してもよいので,  $x$  軸や  $y$  軸に関する対称移動も書ける。

#### 例 47

```
10 DECLARE EXTERNAL PICTURE arrow
20 SET WINDOW -2, 2, -2, 2
30 DRAW grid
40 DRAW arrow
50 SET AREA COLOR 3
60 DRAW arrow WITH SCALE(1, -1)
70 END
100 EXTERNAL PICTURE arrow
110 PLOT AREA : 0, 0; 0, 0.1; .7, 0.1; .7, 0.2; 1, 0
120 END PICTURE
```

実行結果



### 5.2.3 変換の合成

次の例は, 平行移動の後に回転することを指示している。

#### 例 48

```
10 DECLARE EXTERNAL PICTURE arrow
20 SET WINDOW -2, 2, -2, 2
30 DRAW grid
40 DRAW arrow WITH ROTATE(PI/8) * SHIFT(0, 1)
50 END
100 EXTERNAL PICTURE arrow
110 PLOT AREA : 0, 0; 0, 0.1; .7, 0.1; .7, 0.2; 1, 0
120 END PICTURE
```

実行結果



変換の合成を表す演算子は\*である。左から右の順に、つまり、書かれた順に実行される。

## 5.2.4 合同変換・相似変換

### 任意の点を中心とする回転

点 $(a, b)$ を中心とする $\theta$ の回転は、点 $(a, b)$ が原点に移るように平行移動して $\theta$ だけ回転し、原点が点 $(a, b)$ に移るように平行移動すれば実現できる。

#### 例 49

```

10 DECLARE EXTERNAL PICTURE arrow
20 SET WINDOW -2, 2, -2, 2
30 DRAW grid
40 LET a=0.5
50 LET b=0
60 LET theta=pi/3
70 DRAW arrow WITH SHIFT(-a, -b)*ROTATE(theta) * SHIFT(a, b)
80 END
100 EXTERNAL PICTURE arrow
110 PLOT AREA : 0, 0; 0, 0.1; .7, 0.1; .7, 0.2; 1, 0
120 END PICTURE

```

実行結果



同様の手法で、ある点を中心とする拡大・縮小や、ある直線に関する対称移動などができる。

## 5.2.5 $4 \times 4$ 行列による変換

DRAW 文の WITH 句に、 $4 \times 4$  行列を書くことができる。

a を DIM a(4,4) で宣言された 2 次元配列とするとき、DRAW □ with a を実行すると、点(x,y)は次式の (x', y') に変換される。ただし、a(i, j)を a<sub>ij</sub> で表す。

$$x' = \frac{a_{11}x + a_{21}y + a_{41}}{a_{14}x + a_{24}y + a_{44}}, \quad y' = \frac{a_{12}x + a_{22}y + a_{42}}{a_{14}x + a_{24}y + a_{44}}$$

### 透視投影

視点の高さを 1.5 m として、0.5 m 先に地平面に垂直に置かれたスクリーンに見えたとおりに写しとったとすると、原点をスクリーン中央の地平面との接点に置くと、地平面上の点 (x, y) は、次式で定まるスクリーン上の点 (x', y') に写る。

$$x' = \frac{0.5x}{0.5 + y}, \quad y' = \frac{1.5y}{0.5 + y}$$

x', y' ともに分母、分子は x, y の一次式であって、分母は共通である。だから、この変換は DRAW WITH を用いて表せる。

上半平面に両軸に平行な格子を描く絵定義を変換してみた。

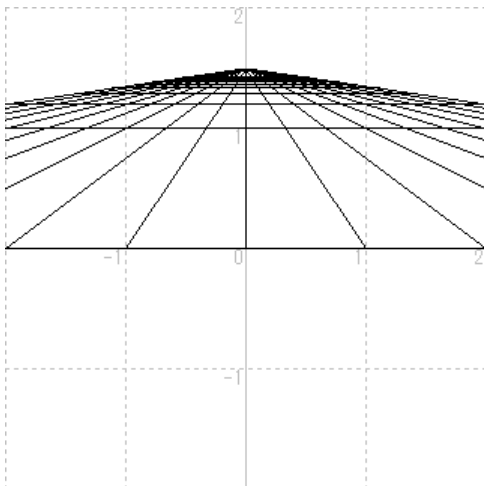
### 例 50

```

100 DECLARE EXTERNAL PICTURE p
110 DATA 0.5, 0, 0, 0
120 DATA 0, 1.5, 0, 1
130 DATA 0, 0, 0, 0
140 DATA 0, 0, 0, 0.5
150 DIM a(4,4)
160 MAT READ a
170 SET WINDOW -2, 2, -2, 2
180 DRAW grid
190 DRAW p WITH a
200 END

500 EXTERNAL PICTURE p
510   FOR x=-10 TO 10
520     PLOT LINES : x, 0; x, 100
530   NEXT x
540   FOR y=0 TO 10
550     PLOT LINES:-10, y; 10, y
560   NEXT y
570 END PICTURE

```



## 5.2.6 変換項と行列

SHIFT, SCALE, ROTATE などの変換項は、行列で定義されている。たとえば、

$$\text{SCALE}(a,b) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & 0 & 0 \end{pmatrix}$$

実際、それらは、MAT 文で配列値の関数として認識される。(正式には変形指示 MAT 文という)

### 例 51

```
DIM a(4,4)
MAT a=SHIFT(2,3)
MAT PRINT a;
END
```

実行結果

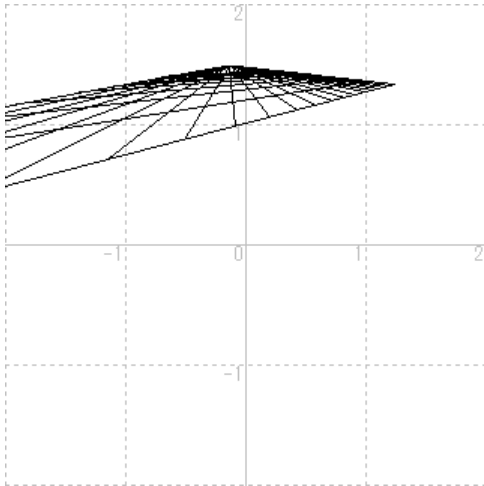
```
1 0 0 0
0 1 0 0
0 0 1 0
2 3 0 1
```

SHIFT, SCALE などの変形関数は、 $4 \times 4$  行列であり、WITH 文に配列と変形関数を混ぜて書くことができる。

### 例 52

```
100 DECLARE EXTERNAL PICTURE p
110 DATA 0.5, 0, 0, 0
120 DATA 0, 1.5, 0, 1
130 DATA 0, 0, 0, 0
140 DATA 0, 0, 0, 0.5
150 DIM a(4,4)
160 MAT READ a
170 SET WINDOW -2, 2, -2, 2
180 DRAW grid
190 DRAW p WITH SHIFT(2,1)*ROTATE(PI/12)*a
200 END
500 EXTERNAL PICTURE p
510 FOR x=-10 TO 10
520 PLOT LINES : x, 0; x, 100
530 NEXT x
540 FOR y=0 TO 10
550 PLOT LINES:-10, y; 10, y
560 NEXT y
570 END PICTURE
```

実行結果



### 5.2.7 絵定義中で DRAW 文を実行する

絵定義内に WITH 句を書いた DRAW 文を書くことができ、その絵定義を WITH 句を書いた DRAW 文で呼び出してよい。そのとき、変換行列の積が計算されて描画に適用される。

変換行列は、3次元の変換に対応している。xy 平面上の描画を空間内の別の平面への描画に変換することができる。それをさらに xy 平面に射影することで、3D グラフィックスが実現する。ただし、できるのは、空間内の平面への描画に限られる。具体例は、サンプルプログラムの SAMPLE¥3DVIEW.BAS にある。

また、絵定義中で自身を再帰的に呼び出してよい。それを利用すると、自己相似図形を簡単に描ける。具体例は、FRACTAL フォルダにある CRAB.BAS, DRAGON.BAS, KOCH.BAS, LEVY.BAS などである。

### 5.2.8 DRAW 文による変換対象の命令

PLOT POINTS, PLOT LINES, PLOT AREA など、PLOT, または、MAT PLOT で始まる描画文は、指定した座標が変換される。ただし、描かれる点や線、文字の字形などは影響を受けない。GET POINT, MAT GET POINT と独自拡張命令の MOUSE POLL は変換された座標系のもとで動作する。

## 5.3 ピクセル座標

### 5.3.1 ピクセル座標

コンピュータの画面は長方形に配置された画素からできている。画素を対象に操作したときは、次の手段を利用する。

```
ASK PIXEL SIZE (x1, y1 ; x2, y2) a, b
```

2点 $(x_1, y_1)$ ,  $(x_2, y_2)$ を対角の頂点とする長方形に含まれる画素数の水平方向の値を変数  $a$  に、垂直方向の値を変数  $b$  に代入する。長方形の頂点または辺上に位置する画素も数える。

SET WINDOW l, r, b, t を実行したとき、

```
ASK PIXEL SIZE (l, b ; r, t) a, b
```

```
FOR x=l TO r STEP (a-1)*(1-r)
```

```
FOR y=b TO t STEP (b-1)*(t-b)
```

```
.....
```

NEXT y

NEXT x

のようなプログラムを書くと、無駄なくすべてのピクセルに対する処理ができる。

### 5.3.2 独自拡張関数 PIXELX, PIXELY, PROBLEMX, PROBLEMY

2進演算のとき STEP  $(a-1)*(1-r)$  はうまく働かないかもしれない。ピクセル座標で点を指定し、その点の問題座標を求めて計算する手法も可能。ピクセル座標から問題座標を求める関数を定義することは簡単にできるが、十進 BASIC の独自拡張として用意された PLOBLEMX, PROBLEMY 関数と、PIXELX, PIXELY 関数を使うと記述が簡単になる。

SET WINDOW 文で導入される座標系を問題座標という。PIXELX(x), PIXELY(y) は、問題座標の x, y からピクセル座標を求める。PLOBLEMX(a), PROBLEMY(b) はピクセル座標 a, b から問題座標を求める。Full BASIC では、ピクセル座標は左下が原点(0,0)で、右上隅が座標が最大である。座標系の正の向きは、伝統的な PC のハードウェア座標系と異なるかも知れない。

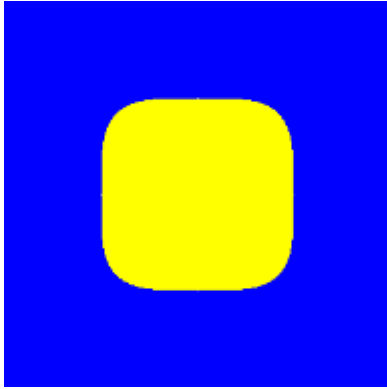
#### 関数 $f(x, y)$ の正領域と負領域

曲線  $f(x, y)=0$  を描くのは代数的手法や解析的手法を用いないと難しいが、正領域と負領域を求めれば、その境界として曲線を浮かび上がらせることができるかも知れない。対象とする領域の各点に対し関数値を計算し、正領域は 2 番、負領域は 6 番の色で塗りつぶすことにしてみる。

#### 例 53

```
100 DEF f(x,y)=x^4+y^4-1
110 LET left=-2
120 LET right=2
130 LET bottom=-2
140 LET top=2
150 SET WINDOW left, right, bottom, top
160 SET POINT STYLE 1
170 DRAW grid
180 FOR u=pixelx(left) TO pixelx(right)
190   FOR v=pixely(bottom) TO pixely(top)
200     LET x=problemx(u)
210     LET y=problemy(v)
220     LET z=f(x,y)
230     IF z>0 THEN
240       SET POINT COLOR 2
250     ELSEIF z<0 THEN
260       SET POINT COLOR 6
270     ELSE
280       SET POINT COLOR 0
290     END IF
300     PLOT points:x,y
310   NEXT v
320 NEXT u
330 END
```

実行結果



### マンデルブローの $\mu$ -map

$\mu$  を複素数とする。  $z_0=0, z_{n+1}=z_n^2+\mu$  で定義される複素数列  $\{z_n\}$  が有界となる  $\mu$  の全体をマンデルブロー  $\mu$  集合という。たとえば、  $\mu=0$  はマンデルブロー  $\mu$  集合に属することは容易にわかる。点  $\mu$  が  $\mu$  集合の点であるかどうか調べるためには、  $z$  の初期値を  $0$  として、  $z \leftarrow z^2 + \mu$  の反復を行う。無限に繰り返すことはできないので、適当な回数繰り返して有限の値にとどまれば有界と判断することにする。また、桁あふれのエラーになったら有限ではないと判断することにする。

実部が  $-2 \sim 1$ 、虚部が  $-1.5 \sim 1.5$  の範囲で調べるために 130 行の SET WINDOW 文を実行すると、ピクセル  $x$  座標は PIXELX(-2) ~ PIXELX(1)、ピクセル  $y$  座標は PIXELY(-2) ~ PIXELY(1) となる。この範囲の各ピクセル  $(u, v)$  に対し、問題座標 PROBLEMX  $(u)$ 、PROBLEMY  $(v)$  を  $a, b$  として、  $\mu = a + bi$  が  $\mu$  集合の点かどうか調べる。1000 回繰り返して桁あふれにならなければ有界と判断して、PLOT POINT a,b を実行する。

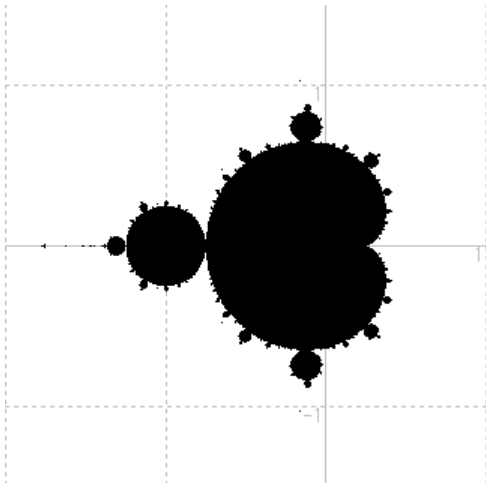
#### 例 54

```

100 OPTION ARITHMETIC COMPLEX
110 DEF f(z)=z^2 + mu
120 LET i=SQR(-1)
130 SET WINDOW -2, 1, -1.5, 1.5
140 DRAW grid
150 SET POINT STYLE 1
160 FOR u=PIXELX(-2) TO PIXELX(1)
170   FOR v=PIXELY(-1.5) TO PIXELY(1.5)
180     LET a=PROBLEMX(u)
190     LET b=PROBLEMY(v)
200     LET mu=a+i*b
210     LET z=0
220     WHEN EXCEPTION IN
230       FOR k=1 TO 1000
240         LET z=f(z)
250       NEXT k
260       PLOT POINTS : a,b
270     USE
280     END WHEN
290   NEXT v
300 NEXT u
310 END

```

実行結果



---

## 参考書

JIS Full BASIC 規格の全文

「JIS 電子計算機プログラム言語 Full BASIC」, JIS X 3003, 日本規格協会