7 実数データの扱い

7–1 実数計算

例題 7. 1 (円錐の体積 ; float 型, double 型, マクロ名)　**2** つの実数データ **r, h** を読み込み、

底面の半径が **r**、高さが **h** の円錐の体積

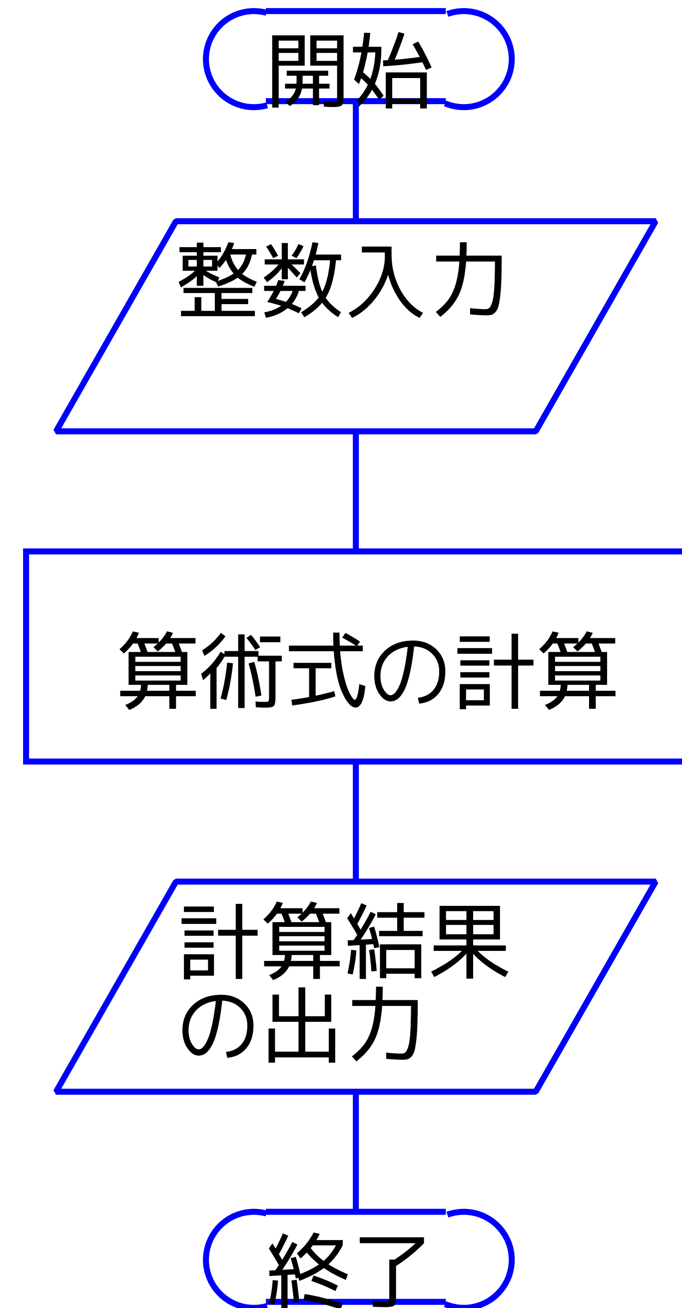
を出力する **C** プログラムを作成せよ。

(考え方)
処理の流れは第5節で考えた右図と同じである。

違いは、
計算対象が整数データではなく実数データであるということと、
計算式が

$$\text{体積} = \frac{\pi \mathbf{r}^2 \mathbf{h}}{3}$$

ということだけである。



(プログラミング)

実数データを表すためのデータ型として、C言語では

`float`型,
`double`型,
`long double`型

} (浮動小数点数型)

の3つが用意されている。

このうち、良く使われるのは `float`型 と `double`型 の2つ

⇨ `float`型 と `double`型で処理したプログラムをそれぞれ例示する。

double型で処理するプログラム：

[motoki@x205a]$ nl volume-of-cone-double.c Enter

```
 1 /* 2つの実数データ r と h を読み込み、                  */
 2 /*      底面の半径が r、高さが h の円錐の体積    */
 3 /* を出力するCプログラム                               */
 4 /*   ---double型で計算する版---                        */

 5 #include <stdio.h>
 6 #define  PI    (3.141592653589793 2)   /* 円周率 */

 7 int main(void)
 8 {
 9   double r, h;

10   scanf("%lf%lf", &r, &h);
11   printf("底面の半径が %f, 高さが %f の円錐の体積\n"
           "         = %f\n",
```

```
    12              r, h, PI*r*r*h/3.0);
    13    return 0;
    14 }
[motoki@x205a]$  gcc volume-of-cone-double.c  Enter
[motoki@x205a]$  ./a.out  Enter
2.0 5.0  Enter
底面の半径が 2.000000, 高さが 5.000000 の円錐の体積
     = 20.943951
[motoki@x205a]$
```

float型で処理するプログラム：

```
[motoki@x205a]$  nl volume-of-cone-float.c  Enter
     1  /* 2つの実数データ r と h を読み込み、          */
     2  /*      底面の半径が r、高さが h の円錐の体積   */
     3  /* を出力するCプログラム                        */
     4  /*   ---float型で計算する版---                  */

     5  #include <stdio.h>
     6  #define  PI    (3.1415926f)   /* 円周率 */

     7  int main(void)
     8  {
     9    float r, h;

    10    scanf("%f%f", &r, &h);
    11    printf("底面の半径が %f, 高さが %f の円錐の体積\n"
                 "        = %f\n",
```

```
12              r, h, PI*r*r*h/3.0f);
13     return 0;
14 }
[motoki@x205a]$ gcc volume-of-cone-float.c  Enter
[motoki@x205a]$ ./a.out  Enter
2.0 5.0  Enter
底面の半径が 2.000000, 高さが 5.000000 の円錐の体積
     = 20.943950
[motoki@x205a]$
```

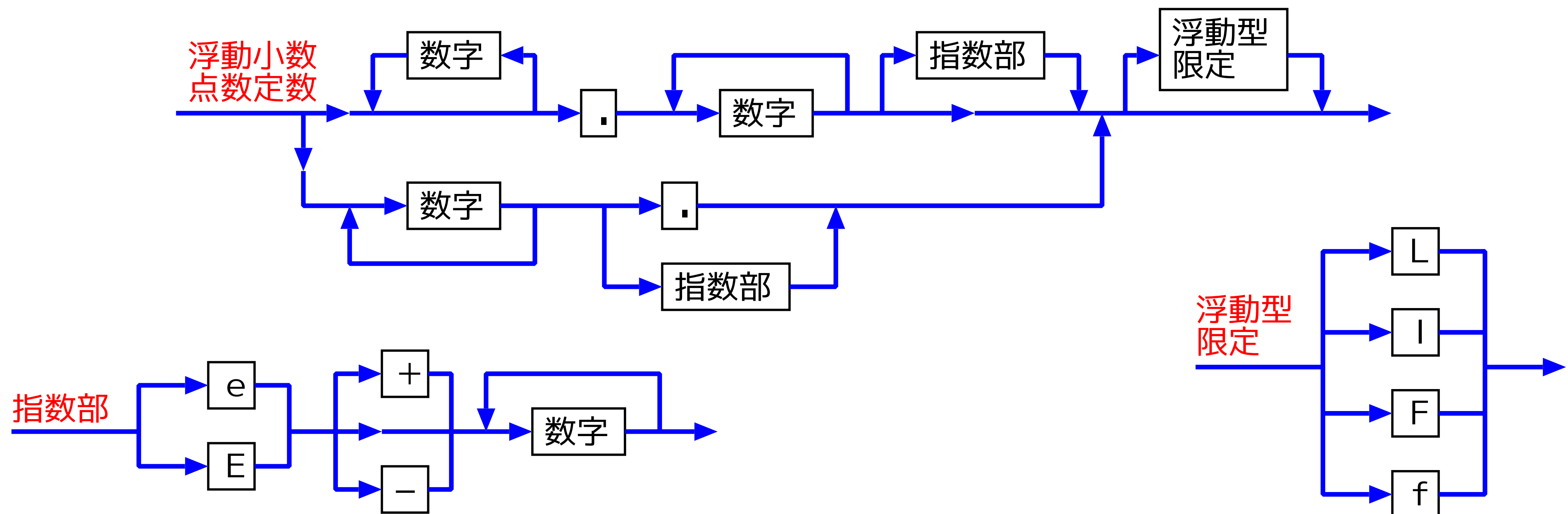
実数データ間の算術演算：

実数データ間では剰余演算子 % が使えないことを除いて、整数データの場合と同じである。 すなわち、次の算術演算子が使える。

算術演算子	機能
+	加算。但し、単項演算の場合は恒等変換を表す。
−	減算。但し、単項演算の場合は符号反転を表す。
*	乗算。
/	除算。

浮動小数点定数：

- 123.4, 123., .4, 123.4e5, .4E+5, 123e-5, ... といった書き方が出来る。 これらはdouble型の定数で、それぞれ **123.4, 123.0, 0.4, $\mathbf{123.4 \times 10^{5}}$, $\mathbf{0.4 \times 10^{5}}$, $\mathbf{123 \times 10^{-5}}$, ...** を表す。
- 定数をfloat型にしたければ、最後に f または F という接尾語を付ける。 例えば、123.4f, .4E+5F, ... 。
- 定数をlong double型にしたければ、最後に l または L という接尾語を付ける。 例えば、123.4l, .4E+5L, ... 。



精度と指数部の表現可能な範囲：

- ごく普通の計算機の場合、`float`型, `double`型データは、各々4バイト, 8バイトの領域を占め、10進で各々約6桁, 約15桁の精度を持つ。また、指数部の表現可能な範囲は、およそ、各々 -38~ +38, -308~ +308となる。[指数部の表現可能な範囲は計算機のアーキテクチャに大きく依存する。]

7–2 整数と実数の混合演算，キャスト演算

コンピュータは

- 同じデータ型同士の四則演算回路　は持っているが、
- 違ったデータ型の間の四則演算回路 は持っていない。

しかし、算術式の書き方にこの様な制限を設けるとプログラムが書きにくくなる。

⇨ C 言語では算術式の中に
違ったデータ型同士の四則演算が書けるようになっている。

補足：
コンピュータ内部では演算回路に流し入れるデータの型はやはり揃ってなければならないので、コンパイラが演算対象のデータ型を適宜変換して型を揃える操作を補う。

例題7．4 (違ったデータ型同士の四則演算) 違ったデータ型同士の四則演算を算術式の中に書いた場合、プログラム実行においては演算の前に適当な型変換によってデータ型が揃えられる。 int型, float型 または double型のデータの組に対して四則演算を行う際、演算の前に実際にどの様な型変換が行われるのかを調べよ。

(考え方) 除算の場合は、被除数a=1, 除数b=3 として a/b の演算結果の精度を見ることによってどういう型変換が行われるかを推察する。

(プログラミング) 被除数 a=1 と除数 b=3 のデータ型を int, float, double の範囲で色々と変えてみて、それらの計算結果を整数と仮定して出力したり実数と仮定して出力したりする。

補足：
この課題では計算結果のデータ型を調べようとしているので、それらの結果を使ってプログラムを書く訳にもいかない。
⇨ 出てきた結果を整数と見て出力することも
実数と見て出力することも行った。

```
[motoki@x205a]$ nl division-between-different-data-type.c  Enter
     1  /* int型, float型 または double型のデータの組に対して */
     2  /* 四則演算を行う際、演算の前に実際にどの様な型変換が */
     3  /* 行われるのかを調べるためのCプログラム                */

     4  #include <stdio.h>

     5  int main(void)
     6  {
     7    printf("a/bの結果を %%13d で表示:\n"
     8           "                    b=3(int)    b=3.0f(float)  b=3.0(double)\n
     9           "                 -------------  -------------  -------------\n
    10    printf("     a=1(int)  %13d", 1/3);
    11    printf(              "  %13d", 1/3.0f);
    12    printf(                     "  %13d\n", 1/3.0);
    13    printf("a=1.0f(float)  %13d", 1.0f/3);
    14    printf(              "  %13d", 1.0f/3.0f);
    15    printf(                     "  %13d\n", 1.0f/3.0);
    16    printf("a=1.0(double)  %13d", 1.0/3);
    17    printf(              "  %13d", 1.0/3.0f);
```

```
18    printf(                              "  %13d\n\n", 1.0/3.0);

19    printf("a/bの結果を %%13.11f で表示:\n"
20           "                   b=3(int)    b=3.0f(float)  b=3.0(double)\n
21           "                 -------------  -------------  -------------\n
22    printf("     a=1(int)  %13.11f", 1/3);
23    printf(                          "  %13.11f", 1/3.0f);
24    printf(                                    "  %13.11f\n", 1/3.0);
25    printf("a=1.0f(float)  %13.11f", 1.0f/3);
26    printf(                          "  %13.11f", 1.0f/3.0f);
27    printf(                                    "  %13.11f\n", 1.0f/3.0);
28    printf("a=1.0(double)  %13.11f", 1.0/3);
29    printf(                          "  %13.11f", 1.0/3.0f);
30    printf(                                    "  %13.11f\n", 1.0/3.0);
31    return 0;
32 }
[motoki@x205a]$
```

```
[motoki@x205a]$ gcc division-between-different-data-type.c
[motoki@x205a]$ ./a.out Enter
a/bの結果を %13d で表示:
                b=3(int)     b=3.0f(float)  b=3.0(double)
              -------------  -------------  -------------
     a=1(int)              0     1610612736     1431655765
a=1.0f(float)     1610612736     1610612736     1431655765
a=1.0(double)     1431655765     1431655765     1431655765

a/bの結果を %13.11f で表示:
                b=3(int)     b=3.0f(float)  b=3.0(double)
              -------------  -------------  -------------
     a=1(int)            nan  0.33333334327  0.33333333333
a=1.0f(float)  0.33333334327  0.33333334327  0.33333333333
a=1.0(double)  0.33333333333  0.33333333333  0.33333333333
[motoki@x205a]$
```

(実行結果の考察)　a/bの演算結果は3種類に分類できる。

```
[motoki@x205a]$ ./a.out [Enter]
a/bの結果を %13d で表示:
                   b=3(int)    b=3.0f(float)  b=3.0(double)
               -------------  -------------  -------------
     a=1(int)              0     1610612736     1431655765
a=1.0f(float)     1610612736     1610612736     1431655765
a=1.0(double)     1431655765     1431655765     1431655765

a/bの結果を %13.11f で表示:
                   b=3(int)    b=3.0f(float)  b=3.0(double)
               -------------  -------------  -------------
     a=1(int)            nan  0.33333334327  0.33333333333
a=1.0f(float)  0.33333334327  0.33333334327  0.33333333333
a=1.0(double)  0.33333333333  0.33333333333  0.33333333333
[motoki@x205a]$
```

この結果から、演算 **a/b** は次の様に行われることが分かる。

	b (int)	**b (float)**	**b (double)**
a (int)	そのまま演算	**float** に揃えて演算	**double** に揃えて...
a (float)	**float** に揃えて演算	そのまま演算	**double** に揃えて...
a (double)	**double** に揃えて...	**double** に揃えて...	そのまま演算

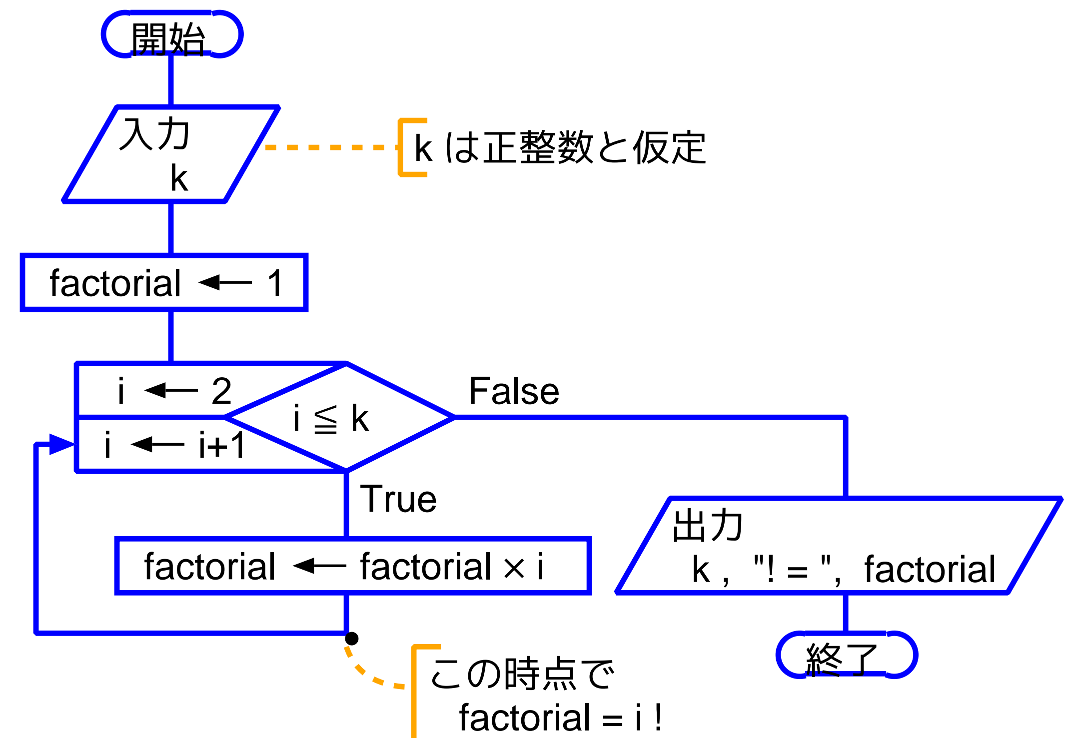
補足：

他の演算 **a+b**, **a-b**, **a*b** についても、
これと全く同様の型変換が行われる。

データ型の変換は、キャスト演算を使ってプログラムの中で指定することも出来る。

例題7. 6 (階乗;キャスト演算)　正整数データ k を読み込み、その階乗値 $k!=1\times 2\times 3\times \cdots \times k$ を double型実数として 求めて出力するCプログラムを作成せよ。

(考え方)　計算の流れは例題 6.6 と同じである。例題 6.6のプログラムとの違いは、**1!, 2!, 3!, …** の値を保持するための変数を int型ではなくて double型とするという点だけである。



(プログラミング)

[motoki@x205a]$ nl factorial-double.c Enter

```
     1  /* 正整数データを読み込み、その階乗値を          */
     2  /* double型実数として求めて出力するCプログラム */

     3  #include <stdio.h>

     4  int main(void)
     5  {
     6    int    k, i;
     7    double factorial;

     8    printf("何の階乗を求めますか?: ");
     9    scanf("%d", &k);
```

```
10    factorial = 1.0;
11    for (i=2; i<=k; ++i){
12      factorial *= (double) i;  /*この時点でfactorial=i!*/
13    }

14    printf("%d! = %21.16g\n", k, factorial);
15    return 0;
16 }
[motoki@x205a]$  gcc factorial-double.c  Enter
[motoki@x205a]$  ./a.out  Enter
何の階乗を求めますか?:   53  Enter
53! = 4.274883284060025e+69
[motoki@x205a]$
```

補足：

不注意による間違いを出来るだけ避けるために、行う予定の型変換はこの12行目の様に明示するのが好ましい。

算術計算の際の自動型変換：

- `int`型,`float`型,`double`型の間では、四則演算 a+b, a-b, a*b, a/b はどれも次の様に行われる。

	b (int)	b (float)	b (double)
a (int)	そのまま演算	floatに揃えて演算	doubleに揃えて...
a (float)	floatに揃えて演算	そのまま演算	doubleに揃えて...
a (double)	doubleに揃えて...	doubleに揃えて...	そのまま演算

- 実数→整数 間の型変換が実際にどう行われるかについては計算機に依存する。 [切捨て、切り上げ、四捨五入のいずれか。]

代入の際の自動型変換：

- 代入 変数等 = 式 において両辺の型が違えば、式 の値は 変数等 の型に強制的に変換される。

キャスト演算子：

- 明示的に型変換を行うことが出来る。

- 式の値をデータ型という型に変換したければ、次の様に書く。

  （ データ型 ） 式

- キャストは単項演算子。

- 他の単項演算子(e.g.符号反転の-,++)と同じ優先順位、結合性(右から左)を持つ。

例7. 8 (キャスト演算の優先順位)

`(float) i+3` は `((float)i) + 3` と同等である。

7–3 実数データの入出力 — %f，%e，%g —

例題7. 9 (実数データの出力書式の比較)　指数関数 $f(x)=3.14\times10^{x}$ の $x=-5, -4, -3, -2, \ldots, 7, 8$ に対する値が`printf()`に用意されている3つの変換指定子 e，f，g によって実際にどの様に出力されるのかを調べよ。

(考え方)　それぞれの $f(x)$ の値が、4つの変換指定

`%12.5e`，`%12.5g`，`%#12.5g`，`%12.5f`

によって実際にどのように出力されるのかを比較すれば良い。

(プログラミング)　$x=-5, -4, -3, -2, \ldots, 7, 8$ に対して $f(x)=3.14\times10^{x}$ の値を順に出力するだけの単純な繰り返しである。

(プログラミング)　$\mathbf{x=-5,\ -4,\ -3,\ -2,\ldots,\ 7,\ 8}$ に対して $\mathbf{f(x)=3.14\times 10^{x}}$ の値を順に出力するだけの単純な繰り返しである。

ただ、数学関数のライブラリから冪乗関数(`pow( , )`)をわざわざ呼び出すこともない。

⇨ ここでは $\mathbf{x=-5,\ -4,\ -3,\ -2,\ \ldots,\ 7,\ 8}$ に対する $\mathbf{f(x)}$ の値を次のように計算する。

$\mathbf{f(-5)}$ ＝ $\mathbf{C}$言語上で`double`型定数 `3.14e-5` に相当

$$\mathbf{f(-4) = f(-5)\times 10.0}$$
$$\mathbf{f(-3) = f(-4)\times 10.0}$$
$$\mathbf{f(-2) = f(-3)\times 10.0}$$

$\cdots\cdots\cdots\cdots\cdots\cdots$

```
[motoki@x205a]$ nl printf-e-f-g-conversion.c  Enter
     1 #include <stdio.h>

     2 int main(void)
     3 {
     4   int    x;
     5   double fx;

     6   printf("-------------------------------------------------------\
     7          "関数 f(x)=3.14*10^x の x=-5,-4,-3, ..., 7,8 に対する値が
\n"
     8          "e,f,g変換記述子によって実際にどの様に出力されるかを見る。\n"
     9          "-------------------------------------------------------\
    10          " x     %%12.5e          %%12.5g          %%#12.5g          %%12.
    11          "--   ------------   ------------   ------------   -----------
```

```
  12    fx=3.14e-5;
  13    for (x=-5; x<=8; x++) {
  14      printf("%2d  %12.5e  %12.5g  %#12.5g  %12.5f\n",
                 x, fx, fx, fx, fx);
  15      fx *= 10.0;
  16    }
  17    return 0;
  18 }
[motoki@x205a]$
```

```
[motoki@x205a]$ gcc printf-e-f-g-conversion.c Enter
[motoki@x205a]$ ./a.out Enter
----------------------------------------------------------
関数 f(x)=3.14*10^x の x=-5,-4,-3, ..., 7,8 に対する値が
e,f,g変換記述子によって実際にどの様に出力されるかを見る。
----------------------------------------------------------
 x      %12.5e        %12.5g       %#12.5g        %12.5f
--   ------------  ------------  ------------  ------------
-5    3.14000e-05     3.14e-05   3.1400e-05        0.00003
-4    3.14000e-04     0.000314   0.00031400        0.00031
-3    3.14000e-03      0.00314    0.0031400        0.00314
-2    3.14000e-02       0.0314     0.031400        0.03140
-1    3.14000e-01        0.314      0.31400        0.31400
 0    3.14000e+00         3.14       3.1400        3.14000
 1    3.14000e+01         31.4       31.400       31.40000
 2    3.14000e+02          314       314.00      314.00000
 3    3.14000e+03         3140       3140.0     3140.00000
```

```
 4   3.14000e+04          31400          31400.   31400.00000
 5   3.14000e+05       3.14e+05      3.1400e+05  314000.00000
 6   3.14000e+06       3.14e+06      3.1400e+06  3140000.00000
 7   3.14000e+07       3.14e+07      3.1400e+07  31400000.00000
 8   3.14000e+08       3.14e+08      3.1400e+08  314000000.00000
[motoki@x205a]$
```

x	%12.5e	%12.5g	%#12.5g	%12.5f
-5	3.14000e-05	3.14e-05	3.1400e-05	0.00003
-4	3.14000e-04	0.000314	0.00031400	0.00031
-3	3.14000e-03	0.00314	0.0031400	0.00314
-2	3.14000e-02	0.0314	0.031400	0.03140
-1	3.14000e-01	0.314	0.31400	0.31400
0	3.14000e+00	3.14	3.1400	3.14000
1	3.14000e+01	31.4	31.400	31.40000
2	3.14000e+02	314	314.00	314.00000
3	3.14000e+03	3140	3140.0	3140.00000
4	3.14000e+04	31400	31400.	31400.00000
5	3.14000e+05	3.14e+05	3.1400e+05	314000.00000
6	3.14000e+06	3.14e+06	3.1400e+06	3140000.00000
7	3.14000e+07	3.14e+07	3.1400e+07	31400000.00000
8	3.14000e+08	3.14e+08	3.1400e+08	314000000.00000

%12.5f ： 小数点以下5桁の精度で右詰めに出力

%12.5e ： [−] 0以外の数字 . 数字列 e ± 2桁以上の数字列 (数字列: 5桁)

%#12.5g ： 有効桁5桁(仮定)までe変換, f変換を行った時の短い方

%12.5g ： %#12.5gとほぼ同じ。小数部末尾の 0, 小数点 は省略。

7–4 数学関数の利用

- C言語では、三角関数,対数関数,指数関数,冪乗関数,平方根関数, ... 等の数学的関数は標準ライブラリの中で提供されている。

  ⇨ 数学的関数を使いたければ 、

  ◇ Cプログラムの最初に `#include <math.h>` の宣言をする。

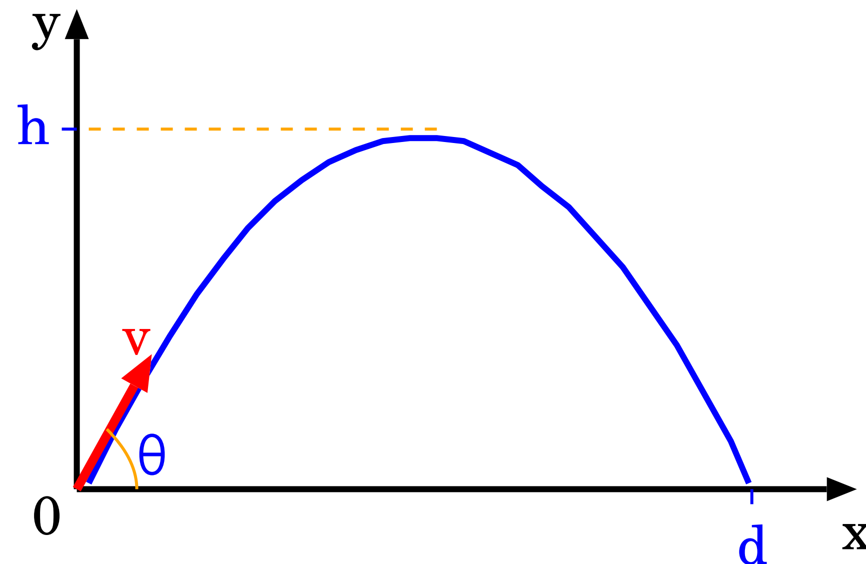
  ◇ コンパイル時には `-lm` オプションを(最後に)付ける。

- 数学的関数の引数、関数値はほとんどが `double` 型である。

例題7. 10 (ボール投げ;三角関数)　初速度 $\mathbf{v}$ $\mathbf{m/sec}$ で地面に対して θ の角度でボールを投げた時、ボールの最高点の高さ $\mathbf{h}$, 届く距離 $\mathbf{d}$, 地面に落ちるまでの時間 τ は、重力加速度 $\mathbf{g{=}9.8m/sec^2}$ を使って

$$\mathbf{h}=\frac{\mathbf{v^2 sin^2}\theta}{\mathbf{2g}},\quad \mathbf{d}=\frac{\mathbf{v^2 sin\ 2}\theta}{\mathbf{g}},\quad \tau=\frac{\mathbf{2v\ sin}\theta}{\mathbf{g}}$$

という風に求めることが出来る。　初速度 $\mathbf{v}$ を読み込み、その $\mathbf{v}$ に対して角度を $\theta = \mathbf{5}^\circ,\ \mathbf{10}^\circ,\ \mathbf{15}^\circ,\ \ldots,\ \mathbf{90}^\circ$ と変えた時に最高点の高さ $\mathbf{h}$, 届く距離 $\mathbf{d}$, 落ちるまでの時間 τ がどの様に変わるかを表の形に見易く表示するCプログラムを作成せよ。



補足(ボール投げの物理学)：

時刻 t にボールがある座標を (x,y) とすると、微分方程式

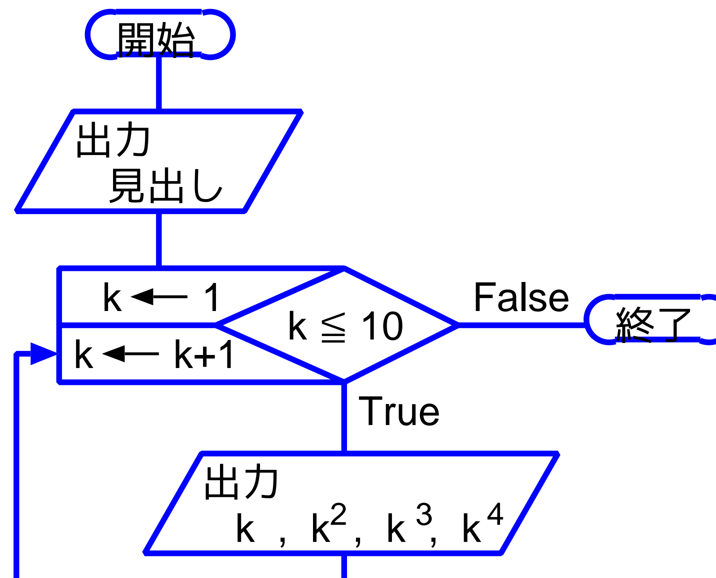
$$\frac{d^2 x}{dt^2}=0, \quad \frac{d^2 y}{dt^2}=-g$$

・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・

(考え方)　計算手順自体は $\theta = 5°, 10°, 15°, \ldots, 90°$ のそれぞれに対して

$$h=\frac{v^2 \sin^2\theta}{2g}, \quad d=\frac{v^2 \sin 2\theta}{g}, \quad \tau=\frac{2v \sin\theta}{g}$$

を計算して出力するだけの単純な繰り返しであるので、処理の構造は例題6.4と同じである。



(プログラミング)

```
[motoki@x205a]$  nl throw-a-ball.c  Enter
              (注釈部は省略)

 6 #include <stdio.h>
 7 #include <math.h>

 8 #define  PI  (3.14159265358979 32)  /* 円周率 */
 9 #define  G   (9.8)                  /* 重力加速度 */

10 int main(void)
11 {
12   int    degree;
13   double v, radian, sin_radian;

14   printf("Input the velocity: ");
15   scanf("%lf", &v);
```

```
16    printf("\nWhen velocity = %g m/sec, ...\n\n"
17           "degree    height     distance      time\n"
18           "------   ----------  ----------  ----------\n",
19           v);

20    for (degree=5; degree<=90; degree+=5) {
21      radian = (double)degree*PI/180.0;      ラジアンに変換
22      sin_radian = sin(radian);
23      printf("%4d    %10.3f  %10.3f  %10.3f\n",
24        degree,
25        v*v*sin_radian*sin_radian/(2.0*G),/*最高点の高さ*/
26        v*v*sin(2.0*radian)/G,          /*届く距離    */
27        2.0*v*sin_radian/G);            /*落ちるまでの時間*/
28    }
29    return 0;
30 }
```

```
[motoki@x205a]$
```

```
[motoki@x205a]$  gcc throw-a-ball.c -lm  Enter
```

-lmオプションを付けないと：

⟹ 次の様にコンパイルエラーになる。

```
[motoki@x205a]$ gcc throw-a-ball.c
/tmp/cc4TDjRQ.o: In function ‘main’:
/tmp/cc4TDjRQ.o(.text+0x83): undefined reference to ‘sin’
/tmp/cc4TDjRQ.o(.text+0xc9): undefined reference to ‘sin’
collect2: ld returned 1 exit status
[motoki@x205a]$
```

```
[motoki@x205a]$  gcc throw-a-ball.c -lm  Enter
[motoki@x205a]$  ./a.out  Enter
Input the velocity:  35.0  Enter

When velocity = 35 m/sec, ...

degree    height     distance      time
------  ----------  ----------  ----------
   5         0.475      21.706       0.623
  10         1.885      42.753       1.240
  15         4.187      62.500       1.849
  20         7.311      80.348       2.443
  25        11.163      95.756       3.019
  30        15.625     108.253       3.571
  35        20.562     117.462       4.097
  40        25.823     123.101       4.591
  45        31.250     125.000       5.051
```

```
  50          36.677     123.101          5.472
  55          41.938     117.462          5.851
  60          46.875     108.253          6.186
  65          51.337      95.756          6.474
  70          55.189      80.348          6.712
  75          58.313      62.500          6.899
  80          60.615      42.753          7.034
  85          62.025      21.706          7.116
  90          62.500       0.000          7.143
[motoki@x205a]$
```

注目点：

- 数学的関数を使う場合、

  #include <math.h> という行は数学的関数を呼び出す部分を間違いなく翻訳するために必要となり、

  ccコマンドの-lmオプションは数学的関数の翻訳コードも取り込んで完全な実行コードを作るために必要となる。

C言語においては、次のような数学的関数が標準ライブラリに用意されている。

機能	関数名（引数の並び）	引数の型	関数値の型	説明
切捨て	floor(a)	double	double	$\lfloor a \rfloor$
切上げ	ceil(a)	double	double	$\lceil a \rceil$
剰余	fmod(a, b)	double	double	$a \geq 0$ の時は $a - \lvert b \rvert \times \lfloor a/\lvert b \rvert \rfloor$
				$a < 0$ の時は $a - \lvert b \rvert \times \lceil a/\lvert b \rvert \rceil$
絶対値	fabs(a)	double	double	$\lvert a \rvert$
平方根	sqrt(a)	double	double	$\sqrt{a}$
べき乗	pow(a, b)	double	double	a^b
	ldexp(a, n)	doubleとint	double	$a \times 2^n$
指数	exp(a)	double	double	e^a
自然対数	log(a)	double	double	$\log_e a$
常用対数	log10(a)	double	double	$\log_{10} a$

機能	関数名（引数の並び）	引数の型	関数値の型	説明
正弦	sin(a)	double	double	sin a, 但し aはラジアン
余弦	cos(a)	double	double	cos a, 但し aはラジアン
正接	tan(a)	double	double	tan a, 但し aはラジアン
逆正弦	asin(a)	double	double	$\sin^{-1} a \in [-\pi/2, \pi/2]$
逆余弦	acos(a)	double	double	$\cos^{-1} a \in [0, \pi]$
逆正接	atan(a)	double	double	$\tan^{-1} a \in [-\pi/2, \pi/2]$
	atan2(a, b)	double	double	$\tan^{-1}(a/b) \in [-\pi/2, \pi/2]$
双曲線正弦	sinh(a)	double	double	sinh a
双曲線余弦	cosh(a)	double	double	cosh a
双曲線正接	tanh(a)	double	double	tanh a
整数部と小数部に分離	modf(a, ptr)	doubleと(double *)	double	aの小数部(符号はaと同じ)を返し、aの整数部を ptrの指す領域に格納
仮数部と指数部に分離	frexp(a, ptr)	doubleと(int *)	double	関数呼び出し直後は $a = (関数値) \times 2^{(ptr の指す int 型の値)}$

補足：

C 言語では、数学的関数には分類されていないが次のような関数も標準ライブラリに用意されている。

機能	関数名 (引数の並び)	引数の型	関数値の型	説明
乱数	rand()	なし	int	[0, RAND_MAX) の間の疑似乱数
	srand()	unsigned	なし	乱数発生器の状態を初期化
絶対値	abs(a)	int	int	$\lvert a \rvert$
	labs(a)	long	long	$\lvert a \rvert$
商と剰余	div(a,b)	int	div_t	a を b で割った時の商と剰余の組
	ldiv(a,b)	long	ldiv_t	a を b で割った時の商と剰余の組

ここで、RAND_MAX は /usr/include/stdlib.h の中で定義されたマクロ名、div_t と ldiv_t は /usr/include/stdlib.h の中で定義された「構造体」の名前である。構造体についてはこの講義ノートの第 11.6 節を参照して下さい。

7–5 コンパイルはどの様に進むか?

この講義ノートの付録A節でも触れられている様に、ccコマンド／gccコマンドによるCプログラムの翻訳作業は実際には次の順に行われる。

① 前処理 (`#include`や`#define`で始まる行の処理等、すなわちヘッダファイルの取り込み, マクロの展開, 注釈の除去など。)

② コンパイル (各々の関数定義を機械語に翻訳, 場合によっては最適化も行う。)

③ リンク (各々の関数の翻訳コードを繋げて1つの実行コードを作る。)

これらの作業の様子を図示すると図1の様になる。

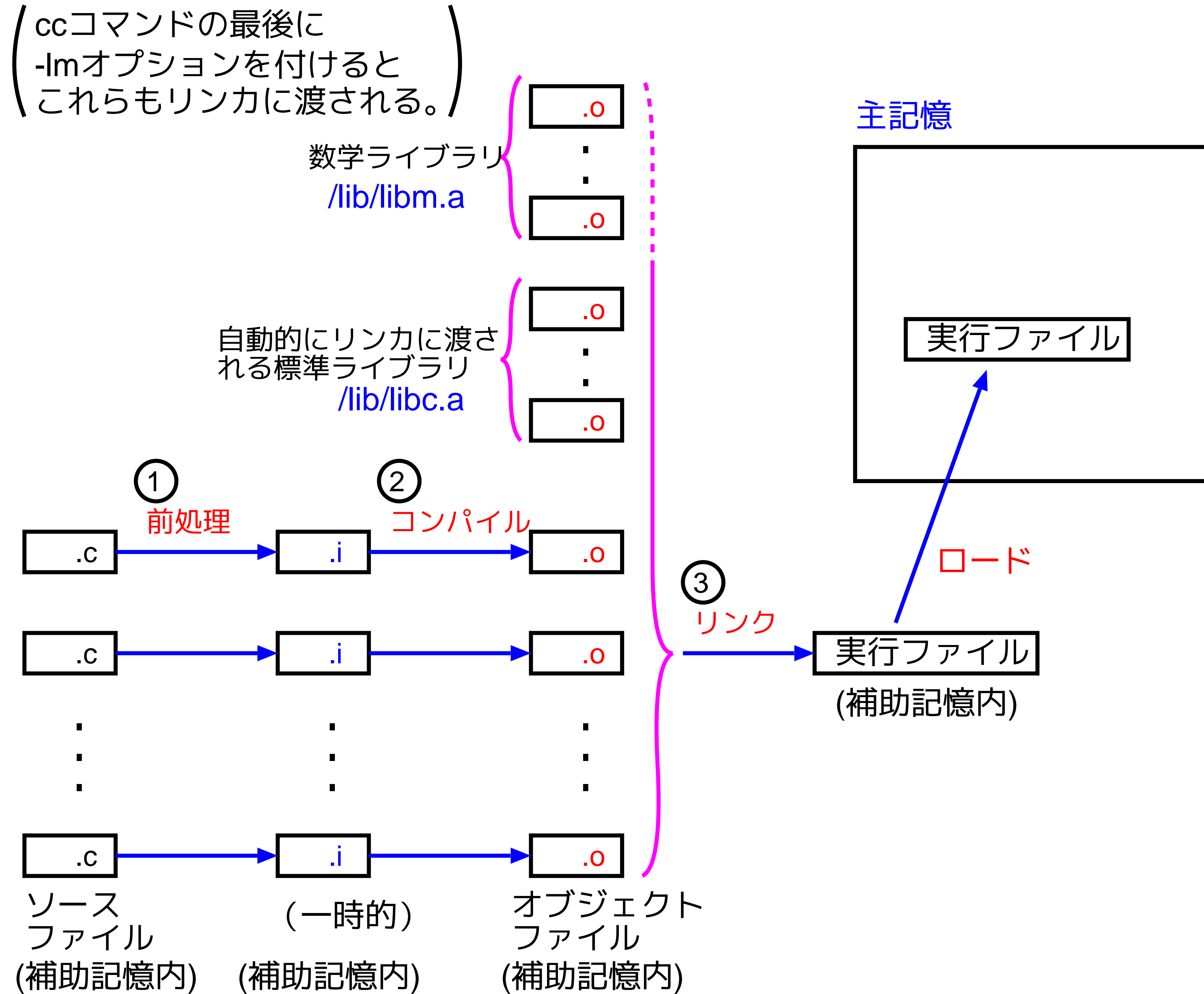


図 1: コンパイル作業の流れ

7–6 誤差の発生とその対策

この節では、実数計算のアルゴリズムを設計する際に注意すべきことを指摘しておく。そのために、まず(数学的に見ると)奇妙な実行結果を幾つか示し、それらの起こる原因について考えてみる。

例題 7. 12 (メモリの有限性, 2 進-10 進変換による誤差) $\mathbf{x=0.1}$ とした時、3 つの計算式

(1) $(10^{16}+1)-10^{16}$,

(2) $x\times10-1$,

(3) $x\times100010-10000$

の値は、数学的には各々 1, 0, 1 になる。 これらの式をコンピュータで計算すると、実際にどういう計算結果が得られるか調べよ。

(プログラミング)

```
[motoki@x205a]$ nl error-by-finiteness-of-memory.c  Enter
     1 /* メモリの有限性、2進-10進変換に起因する計算誤差 */
```

```
 2 #include <stdio.h>

 3 int main(void)
 4 {
 5   double x=0.1;

 6   printf("(1) %21.16g\n", (1e16+1)-1e16);
 7   printf("(2) %21.16g\n", x*10.0-1.0);
 8   printf("(3) %21.16g\n", x*100010-10000);
 9   return 0;
10 }
[motoki@x205a]$ gcc error-by-finiteness-of-memory.c  Enter
[motoki@x205a]$ ./a.out  Enter
(1)                     0   ...... 正解 1
(2) 5.551115123125783e-17   ...... 正解 0
(3)      1.00000000000555   ...... 正解 1
```

```
[motoki@x205a]$
```

(誤差発生の原因について)

いずれの計算結果にも計算誤差が生じている。

これらの誤差は、基本的にはどれも

数値を記憶する領域が有限であり、そのため
記憶された実数データが本来の値の近似似すぎない

ことに起因する。

特に(1)の計算結果からは、実数データ $\mathbf{10^{16}+1}$ を最も良く表す(i.e. 近似する)内部表現が $\mathbf{10^{16}}$ の内部表現と同一になっていることが分かる。

```
[motoki@x205a]$ nl error-by-finiteness-of-memory.c  Enter
          ....................
     6    printf("(1) %21.16g\n", (1e16+1)-1e16);
          ....................
[motoki@x205a]$ ./a.out  Enter
(1)                    0  .......正解 1
          ....................
```

```
5    double x=0.1;
7    printf("(2) %21.16g\n", x*10.0-1.0);
8    printf("(3) %21.16g\n", x*100010-10000);
[motoki@x205a]$  ./a.out  Enter
(2) 5.551115123125783e-17   ....... 正解 0
(3)       1.000000000000555   ....... 正解 1
```

また、(2),(3) の計算結果は、

> コンピュータの数値の記憶方式が 10 進ではなく 2 進であり、
> 10 進で桁数の小さな小数も 2 進では循環 (従って無限) 小数になり得る

ことにも起因する。　実際、10 進小数の 0.1 を 2 進に基数変換すると、$0.0\dot{0}01\dot{1}$ という循環小数になる。これを有限ビットで表そうとすると、必然的にこの無限小数のある桁以降は切捨てられてしまう。

> 補足：
> (2),(3) の場合は、基数変換による誤差が発生した後、(程度の差はあるがいずれも) 次の例題 7. 13 で説明される「桁落ち」によって誤差がクローズアップされてしまった。

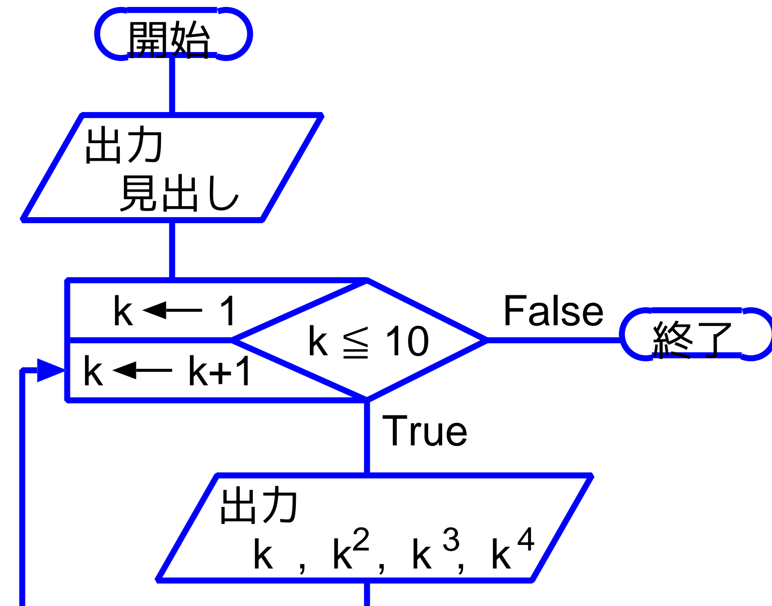
例題7. 13 (桁落ち) 数学的に等価な2つの式

(1) $\frac{1}{x} - \frac{1}{x+1}$,

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

の値を $x=10^0, 10^3, 10^6, \ldots, 10^{21}$ に対してコンピュータで計算して比較してみよ。

(考え方) 計算手順自体は $x=10^0, 10^3, 10^6, \ldots, 10^{21}$ のそれぞれに対して $\frac{1}{x} - \frac{1}{x+1}$ と $\frac{1}{x(x+1)}$ を計算して出力するだけの単純な繰り返しであるので、処理の構造は例題6.4と同じである。



(プログラミング)

[motoki@x205a]$ nl error-cancel-of-sig-digits.c Enter

(注釈は省略)

```
 4 #include <stdio.h>
 5 int main(void)
 6 {
 7   double x;

 8   printf("   x          (1) 1/x-1/(x+1)       (2) 1/(x*(x+
 9         "-------  ---------------------  -------------

10   for (x=1.0; x<1e22; x*=1000.0)
11     printf("%7.2g  %21.16g  %21.16g\n",
12            x,
13            1.0/x-1.0/(x+1.0),
14            1.0/(x*(x+1.0)));
15   return 0;
```

```
16 }

[motoki@x205a]$ gcc error-cancel-of-sig-digits.c  Enter
[motoki@x205a]$ ./a.out  Enter
   x          (1) 1/x-1/(x+1)        (2) 1/(x*(x+1))
-------  ---------------------  ---------------------
      1                    0.5                    0.5
  1e+03  9.9000990009991e-07  9.9000990009991e-07
  1e+06  9.9999000009847e-13  9.9999000010001e-13
  1e+09  9.9999998961678e1e-19          9.99999999e-19
  1e+12  1.0000001954148e1e-24      9.99999999999e-25
  1e+15  1.00003912362307e2e-30   9.9999999999999e-31
  1e+18     9.4039548065783e-37  9.9999999999999999e-37
  1e+21                      0                  1e-42
[motoki@x205a]$
```

```
   x          (1) 1/x-1/(x+1)        (2) 1/(x*(x+1))
-------  ---------------------  ---------------------
      1                    0.5                    0.5
  1e+03  9.900009990009991e-07  9.900009990009991e-07
  1e+06  9.999990000009847e-13  9.999990000010001e-13
         ..............................
  1e+15  1.000039123623072e-30   9.999999999999999e-31
  1e+18    9.403954806578 3e-37  9.999999999999999e-37
  1e+21                      0                  1e-42
[motoki@x205a]$
```

(実験結果の考察)　絶対値の十分小さなϵに対して

$\frac{1}{1+\epsilon} \approx 1 - \epsilon$ と近似でき、$\left|\frac{1}{1+\epsilon} - (1\text{-}\epsilon)\right| \approx \epsilon^2$ である。

⇨ $\frac{1}{x(x+1)} = x^{-2}\frac{1}{1+x^{-1}} \approx x^{-2}\,(1-x^{-1})$　(誤差が x^{-4} 程度)

⇨ 2つの計算結果のうち (2) はほぼ正しい計算値になっている。

```
   x           (1) 1/x-1/(x+1)        (2) 1/(x*(x+1))
-------   ---------------------   ---------------------
      1                     0.5                     0.5
  1e+03   9.9900099900099910e-07   9.9900099900099910e-07
```

x	(1) 1/x-1/(x+1)	(2) 1/(x*(x+1))
1	0.5	0.5
1e+03	9.990009990009991e-07	9.990009990009991e-07
1e+06	9.99999000009847e-13	9.99999000001001e-13
1e+09	9.99999989616781e-19	9.99999999e-19
1e+12	1.00000001954148e-24	9.99999999999e-25
1e+15	1.00003912362307e-30	9.99999999999999e-31
1e+18	9.40395480657830e-37	9.999999999999999e-37
1e+21	0	1e-42

従って、(1) の方は x の値が大きくなるにつれて徐々に有効桁が失われていることになる。この原因としては、(1) では、ほぼ同じ大きさの数の間の減算になり、上位の有効桁が打ち消し合ったことが挙げられる。

この様に、上位の有効桁が打ち消し合って計算結果の有効桁が一挙に失われる現象を桁落ちと呼んでいる。桁落ちは、次の様に手計算の際にも起きる。

$$\begin{array}{rl} \mathbf{1.234567} & \cdots\ 7\text{桁の有効数字} \\ -\ \mathbf{1.234566} & \cdots\ 7\text{桁の有効数字} \\ \hline \mathbf{0.000001} & \cdots\ 1\text{桁の有効数字} \end{array}$$

例題7. 15 (情報落ち) $\log_e 2 = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} = 0.69314718\cdots$ の近似式

$$a = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{1}{99999} - \frac{1}{100000}$$

の値を次の 2 通りの順序で計算して、それらの結果を真値 $a=0.693142180\cdots$ と比較してみよ。

(1) $\frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{1}{99999} - \frac{1}{100000}$ (定義通りに累算)

(2) $-\frac{1}{100000} + \frac{1}{99999} - \cdots - \frac{1}{4} + \frac{1}{3} - \frac{1}{2} + \frac{1}{1}$ (定義の逆順に累算)

(考え方)

誤差が打ち消しあって偶然真値に近い値が出るということもあるので、`double`型だけではなく`float`型でも計算することにする。

また、真値を知り計算値と比較するために(2)の計算を`long double`型でも行うことにする。

処理の組み立てに関しては、
何を繰り返すかに注意する必要がある。

例えば (1) の計算で 、各々の項の加減算を繰り返しの単位にすると、加算と減算を毎回切替える必要があるので面倒になる。

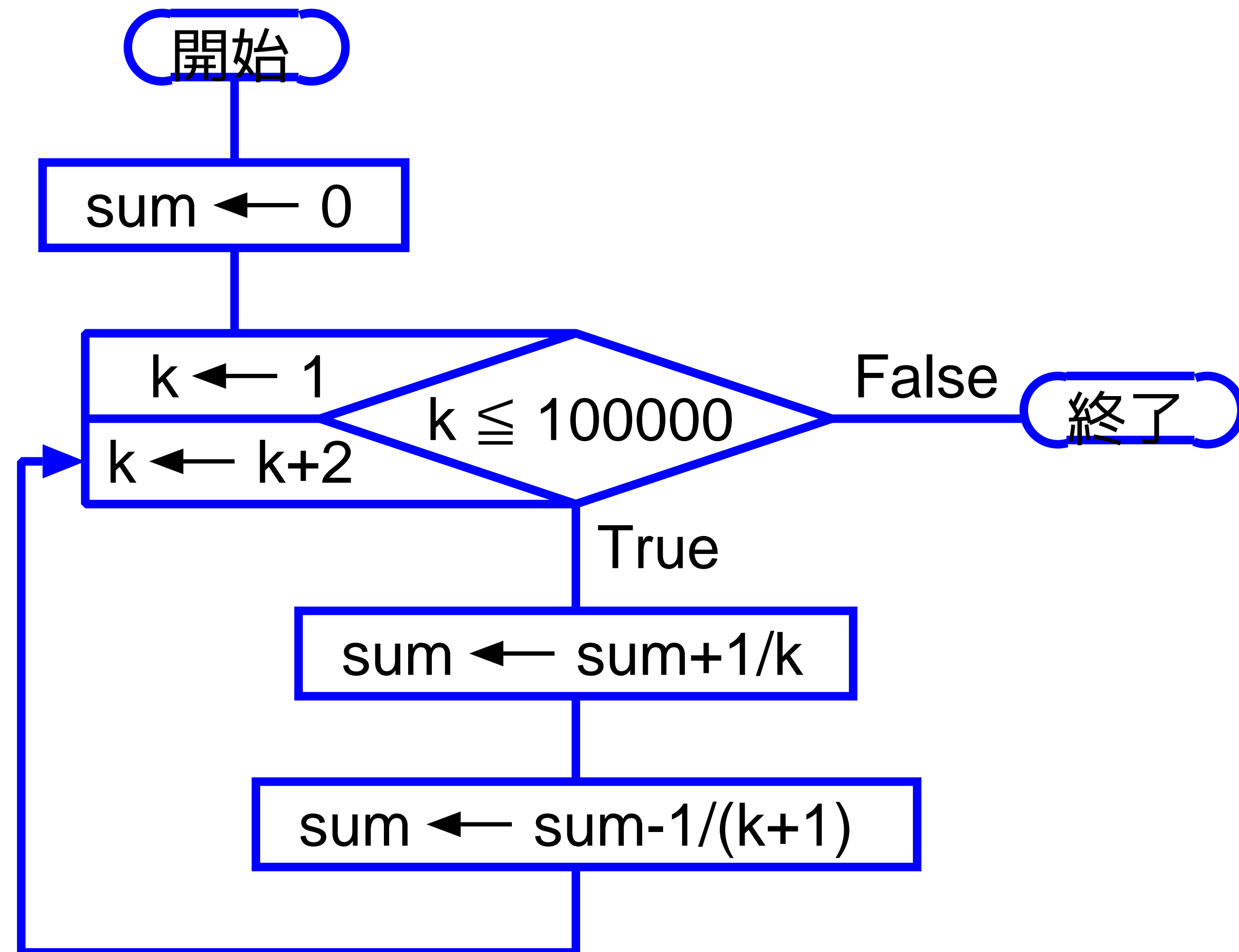
⇨ 次の様に計算すると、繰り返し処理を簡単に書き表すことが出来る。

$$\underbrace{0+\frac{1}{1}-\frac{1}{2}}_{\text{繰り返し}}\underbrace{+\frac{1}{3}-\frac{1}{4}}_{\text{繰り返し}}+\cdots\underbrace{+\frac{1}{99999}-\frac{1}{100000}}_{\text{繰り返し}}$$

(2) の計算では

$$0\underbrace{-\frac{1}{100000}+\frac{1}{99999}}_{\text{繰り返し}}-\cdots\underbrace{-\frac{1}{4}+\frac{1}{3}}_{\text{繰り返し}}\underbrace{-\frac{1}{2}+\frac{1}{1}}_{\text{繰り返し}}$$

(プログラミング)
例えば(1)の順序の累算は右図の様に行えば良い。逆の順序の累算も同程度の規則的な繰り返しで書ける。



[motoki@x205a]$ nl error-loss-of-trailing-digits.c Enter

```
1 /* 累算の順序によって計算結果が変わる例 */

2 #include <stdio.h>

3 int main(void)
```

```
 4 {
 5   int         k;
 6   float       sum_float;
 7   double      sum_double;
 8   long double sum_long_double;

 9   printf("    floatで計算          doubleで計算\n"
10         "    -----------    --------------------\n");

11   /* (1)の計算 */
12   sum_float=0.0f;
13   sum_double=0.0;
14   for (k=1; k<100000; k+=2) {
15     sum_float  += 1.0f/(float)k;
16     sum_float  -= 1.0f/(float)(k+1);
17     sum_double += 1.0/(double)k;
18     sum_double -= 1.0/(double)(k+1);
19   }
20   printf("(1) %11.9f    %20.18f\n", sum_float, sum_doubl
```

```
21    /* (2)の計算 */
22    sum_float =0.0f;
23    sum_double=0.0;
24    sum_long_double=0.0L;
25    for (k=100000; k>1; k-=2) {
26      sum_float -= 1.0f/(float)k;
27      sum_float += 1.0f/(float)(k-1);
28      sum_double -= 1.0/(double)k;
29      sum_double += 1.0/(double)(k-1);
30      sum_long_double -= 1.0L/(long double)k;
31      sum_long_double += 1.0L/(long double)(k-1);
32    }
33    printf("(2) %11.9f    %20.18f\n", sum_float, sum_double
34    printf("    %13.11Lf  %30.28Lf  (long_doubleで計算:真値)
35           sum_long_double, sum_long_double);
36    return 0;
37  }
```

```
[motoki@x205a]$  gcc error-loss-of-trailing-digits.c  Enter
[motoki@x205a]$  ./a.out  Enter
    floatで計算          doubleで計算
    -----------    --------------------
(1) 0.693133771    0.693142180584982004    下線部は真値と違う箇所
(2) 0.693142176    0.693142180584945367
    0.69314218058  0.6931421805849453094241011120  (long_doubl
[motoki@x205a]$
```

```
[motoki@x205a]$  ./a.out  Enter
    floatで計算          doubleで計算
    -----------     --------------------
(1) 0.69313377l     0.69314218058482004     下線部は真値と違う箇所
(2) 0.69314217б     0.69314218058494536ｇ
    0.69314218058   0.69314218058494530942410l1120  (long_doubl
[motoki@x205a]$
```

(実験結果の考察)
long double型で計算した結果と比較することにより、
(1) の順序では float 型で **4** 桁, double 型で **13** 桁の精度が得られ、
(2) の順序では float 型で **7** 桁, double 型で **16** 桁の精度が得られていることが分かる。

(2) の順序で累算すると各々のデータ型で最高に近い精度が得られている一方で、(1) では (2) より **3** 桁程精度が落ちている。

$$(1)\ \frac{1}{1}-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\cdots+\frac{1}{99999}-\frac{1}{100000} \qquad \text{(定義通りに累算)}$$

$$(2)\ -\frac{1}{100000}+\frac{1}{99999}-\cdots-\frac{1}{4}+\frac{1}{3}-\frac{1}{2}+\frac{1}{1} \qquad \text{(定義の逆順に累算)}$$

---

(2) の順序で累算すると各々のデータ型で最高に近い精度が得られている一方で、(1) では (2) より 3 桁程精度が落ちている。

この違いは、同じ有効桁の数でも絶対値の大きさが桁違いに違うと、それらの 2 数を加減算すると小さい方の下位の有効桁が失われてしまうことによる。

$$\begin{array}{r} 1.234567 \\ +\ 0.04321098 \\ \hline 1.27777798 \end{array}$$

この様な誤差発生の現象を情報落ち (loss of trailing digits) または情報埋没 (swamp) と呼んでいる。

加算 $\mathbf{a+b}$ (または減算 $\mathbf{a-b}$) で情報落ちが起こる場合、発生する誤差の上限はほぼ $\mathbf{max\{|a|, |b|\}}$ に比例する。

⇨ 加減算を繰り返す場合、
| 絶対値の小さな数同士の加減算の割合が増える様に
| 加減算の順序を工夫
した方が、誤差の累積は小さく抑えられる。

例えば 、IEEE規格754の単精度で実数データを表す場合を考える。この場合、仮数部は実質24ビットで表されるので、$\mathbf{a+b}$ という加算の際に発生する誤差の上限は

$$|\text{加算の際に発生する誤差}| < \mathbf{max\{|a|, |b|\} \times 2^{-23}}$$

と見積もることが出来る。 ここで、

- (1) の順序で累算する場合は 、...
- (2) の順序で累算する場合は 、...

例えばIEEE規格754の単精度で実数データを表す場合、

$$|\,\mathbf{a+b}\text{ という加算の際に発生する誤差}| < \mathbf{max}\{|\mathbf{a}|, |\mathbf{b}|\}\times \mathbf{2^{-23}}$$

- (1) の順序で累算する場合は、どの加減算においても

$$\mathbf{0.5} \le \mathbf{max}\{|\text{被演算数}|, |\text{演算数}|\} \le \mathbf{1}$$

であるから、

$$|(1)\text{ の累算で発生する誤差}| < \mathbf{2^{-23}}\times(\mathbf{100000}-\mathbf{1}) \approx \mathbf{1.2}\times\mathbf{10^{-2}}$$

補足：

プログラムの実行結果ではこの上限の $\frac{1}{100}$ 程度の誤差しか発生していないが、これは加算による誤差と減算による誤差が打ち消し合ったためと考えられる。

例えばIEEE規格754の単精度で実数データを表す場合、

$$|\,\mathbf{a+b}\ \text{という加算の際に発生する誤差}\,| < \max\{|\mathbf{a}|,\ |\mathbf{b}|\}\times 2^{-23}$$

- (1) の順序で累算する場合は

$$|\text{(1) の累算で発生する誤差}| < 2^{-23}\times(100000-1) \approx 1.2\times 10^{-2}$$

- (2) の順序で累算する場合は、各々の加減算において

$$\max\{|\text{被演算数}|,\ |\text{演算数}|\} \leq \text{演算数}$$

であるから、

$$\begin{aligned}|\text{(2) の累算で発生する誤差}| &< \left(\frac{1}{99999}+\frac{1}{99998}+\cdots+\frac{1}{2}+\frac{1}{1}\right)\times 2^{-23}\\ &< \left(\int_1^{99999}\frac{1}{x}dx+\frac{1}{1}\right)\times 2^{-23}\\ &= (\log 99999+1)\times 2^{-23}\\ &\approx 1.56\times 10^{-6}\end{aligned}$$

$\mathbf{max}\{|$被演算数$|, |$演算数$|\} \leq$ 演算数 について：

◇ 減算の場合、非負の数に関しては 相乗平均≤相加平均 であるから、

$$\left|-\frac{1}{100000}+\frac{1}{99999}-\cdots-\frac{1}{k+2}+\frac{1}{k+1}\right|$$
$$=\frac{1}{100000\times 99999}+\cdots+\frac{1}{(k+2)\times(k+1)}$$
$$\leq\frac{1}{2}\left(\frac{1}{100000^2}+\frac{1}{99999^2}+\cdots+\frac{1}{(k+2)^2}+\frac{1}{(k+1)^2}\right)$$
$$\leq\frac{1}{2}\int_k^{100000}\frac{1}{x^2}dx$$
$$\leq\frac{1}{k}$$

◇ 加算の場合、

$$\left|-\frac{1}{100000}+\frac{1}{99999}-\cdots-\frac{1}{k+3}+\frac{1}{k+2}-\frac{1}{k+1}\right|$$
$$=\left|\frac{1}{100000\times 99999}+\cdots+\frac{1}{(k+3)\times(k+2)}-\frac{1}{k+1}\right|$$
$$<\frac{1}{k+1}$$

(減算の場合の結果より $\frac{1}{100000\times 99999}+\cdots+\frac{1}{(k+3)\times(k+2)}<\frac{1}{k+1}$ だから)

$$<\frac{1}{k}$$

誤差の発生と対策(まとめ)：

計算機を用いて数値計算を行う際、次の様な誤差／現象が起こります。[この内、計算機特有のものは①基数変換に伴う誤差だけであり、残りの3種は手計算の際も起こる。]

① 基数変換 (i.e.2進↔10進変換) に伴う丸め：

例えば、10進小数 0.1 は2進法では $0.0\dot{0}01\dot{1}$ という循環小数になる。それゆえ、各数値を2進有限固定長(普通32ビット)で記憶する計算機としては、表し切れない下位の桁を 丸め (四捨五入,切り捨て,または切り上げ)ることになり、10進小数 0.1 を正確に記憶することはできない。従って、計算機内部で 0.1×10.0 の計算をしても結果は 1 にはならない。

一方、10進数 2^{-20} は計算機内部では実数データとして正確に記憶されるが、この値を10進表記(i.e.2進→10進変換)すると $9.5367431640625\times10^{-7}$ ということになる。この数値は有限小数には違いないが、これを10進7桁の精度で出力すると8桁目以降は捨てられ誤差が発生する。

② 演算に伴う丸め：

例えば、有効桁3桁同士の乗算 1.23×4.56 を行うと

$$1.23 \times 4.56 = 5.60\ 88$$

となり、10^{-3} の位以下が丸め(四捨五入,切り捨て,または切り上げ)られる。この種の誤差に対処するには、式の簡素化などにより演算回数をできるだけ少なくするしかない。

③ 情報落ち (情報埋没) :

これは②の誤差の一種である。絶対値の大きさが桁違いに違う2数を加減算すると小さい方の下位の桁が失われてしまう。例えば、次の加算では下線部が失われる。

$$\begin{array}{r l} & \mathbf{1.234567} \\ + & \mathbf{0.04321098} \\ \hline & \mathbf{1.277777}\underline{\mathbf{98}} \end{array}$$

数回の加減算では大した誤差は累積しないが、大量の実数値データを累算する場合は誤差が大きく累積することもある。

この情報落ち誤差が大きく累積しない様にするためには、

多数の実数データの累算は絶対値の小さいものから順に行う

様に心掛ける。

[実数データを累算する毎に誤差が少しずつ溜まってゆくので、次の④桁落ちほど気を付ける必要はない。]

—

④ 桁落ち：

大きさのほぼ等しい2数を減算する時，あるいはそれと同等の加算をする時、有効桁が大きく失われてしまう。 例えば、次の通り。

$$\begin{array}{rl} \mathbf{1.234567} & \cdots\ \mathbf{7}\text{桁の有効数字} \\ -\ \mathbf{1.234566} & \cdots\ \mathbf{7}\text{桁の有効数字} \\ \hline \mathbf{0.000001} & \cdots\ \mathbf{1}\text{桁の有効数字} \end{array}$$

実数計算においては精度が重要であるから、最終結果に影響を及ぼす様な桁落ちは絶対に避けなければならない。

[厳密に言うと、桁落ちにおいては新たな(絶対)誤差が発生する訳ではない。上位の桁が失われてしまうために、それまで累積していた誤差部分の(以後の計算における)影響度／注目度が大きくなるのである。]