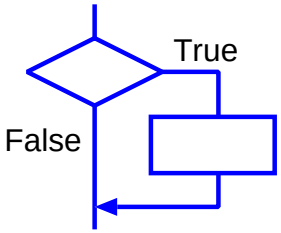
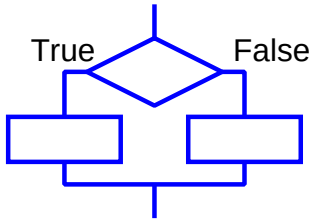
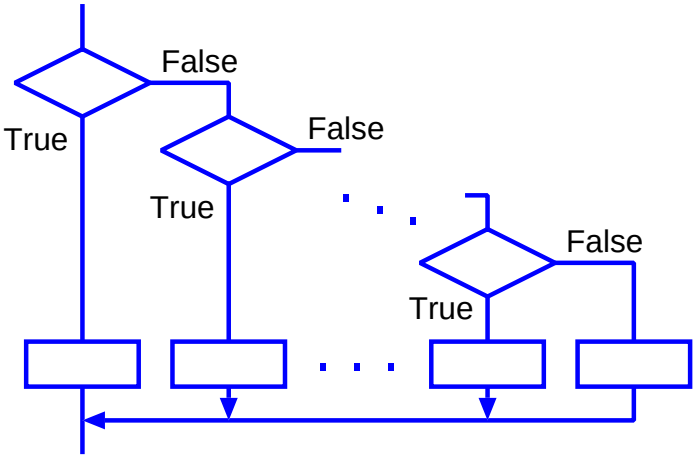
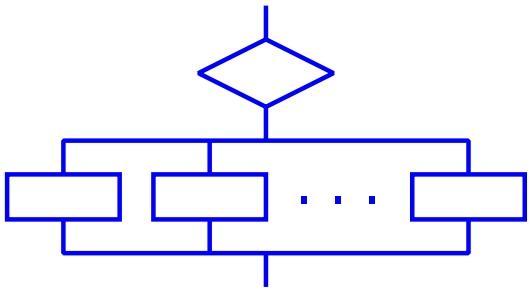
3 復習 処理の選択と繰り返し

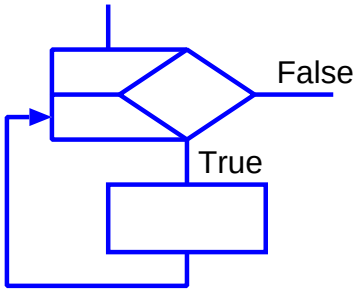
この節では、下図の形の処理の流れがc言語でどの様に記述されるのかを見る。

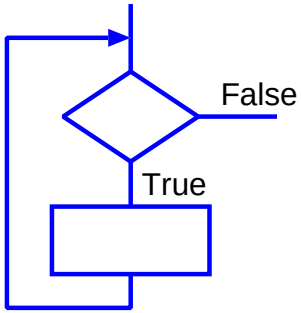


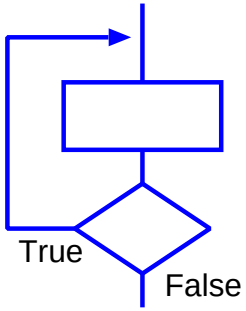


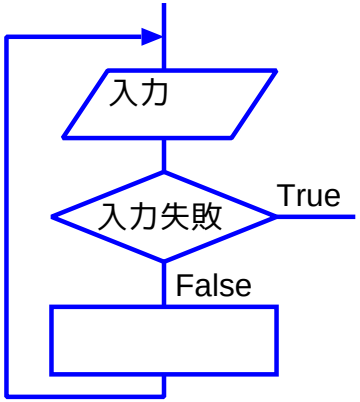





False
True

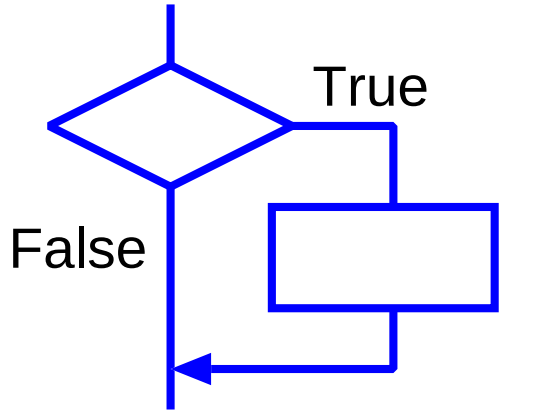

False
True

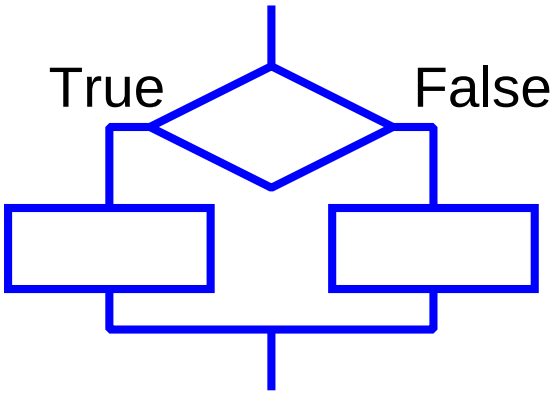

True
False

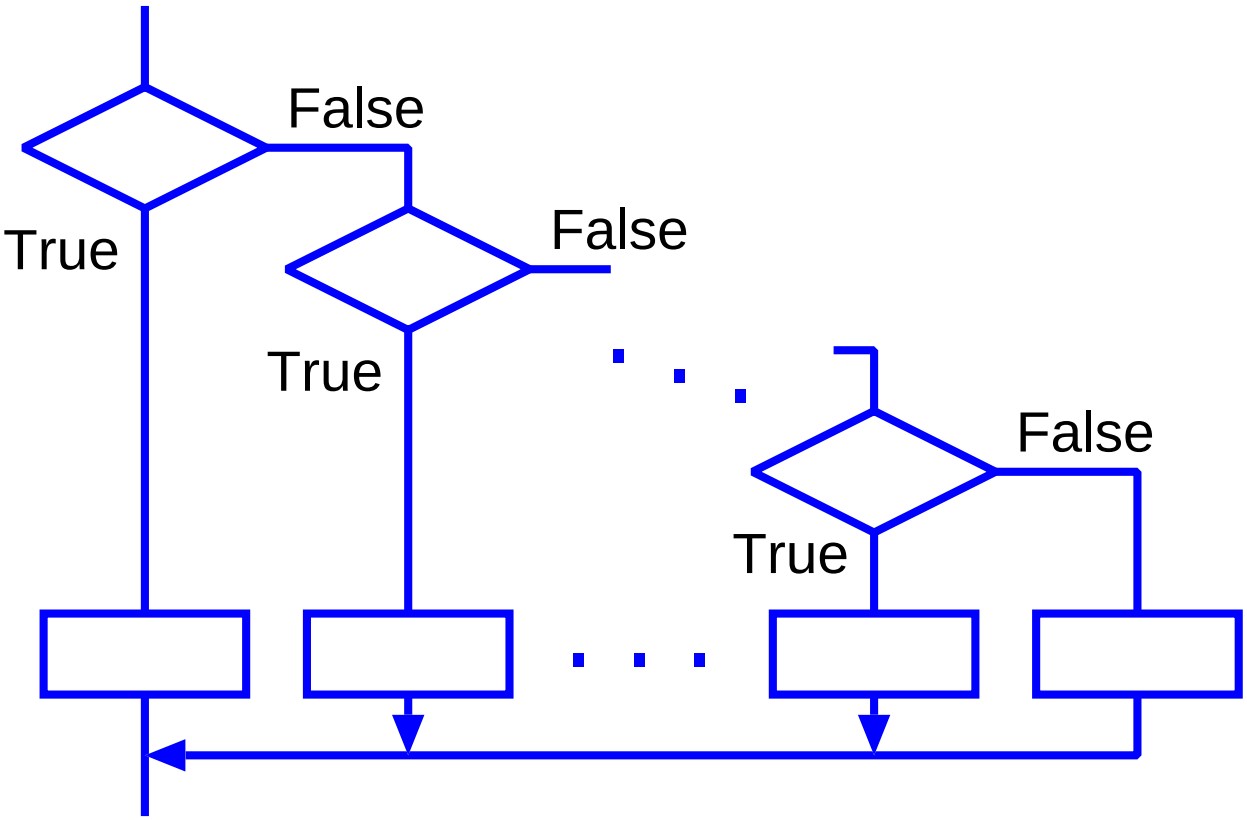

入力
入力失敗
True
False

3–1 条件判断による処理の選択

まず手始めに、
次の形の処理の流れがc言語でどの様に記述されるのかを見る。





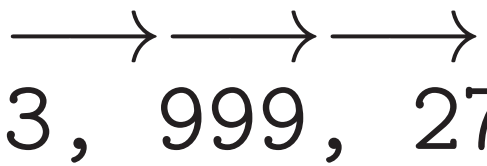


例題3．1（3つの数の最大値；if文,if-else構文,複合文）　整数を3つ読み込みその中の最大値を出力するCプログラムを作成せよ。

この、一見単純そうな問題に対しても、3つのアルゴリズムが思い浮かぶ。以下、これらのアルゴリズムを順に説明していこう。

例題3.1に対するアルゴリズム(1)：

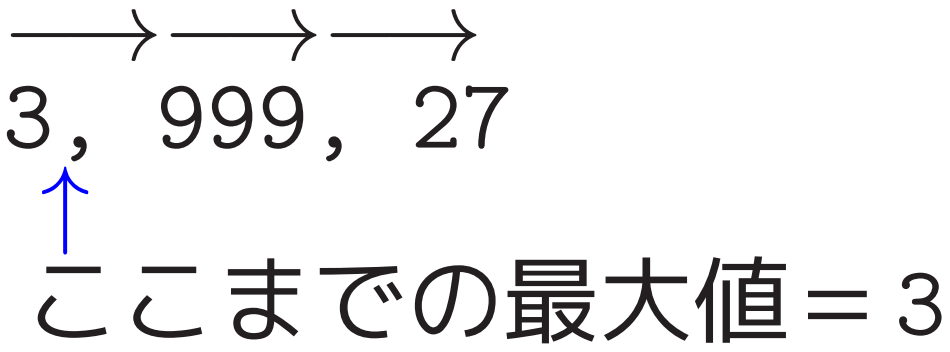
（考え方）　整数3つを読み込んだあとで、読み込んだ整数を順番に眺めていく。



例題3．1（3つの数の最大値；if文,if-else構文,複合文）　整数を3つ読み込みその中の最大値を出力するCプログラムを作成せよ。

例題3.1に対するアルゴリズム(1)：

（考え方）　整数3つを読み込んだあとで、読み込んだ整数を順番に眺めていく。

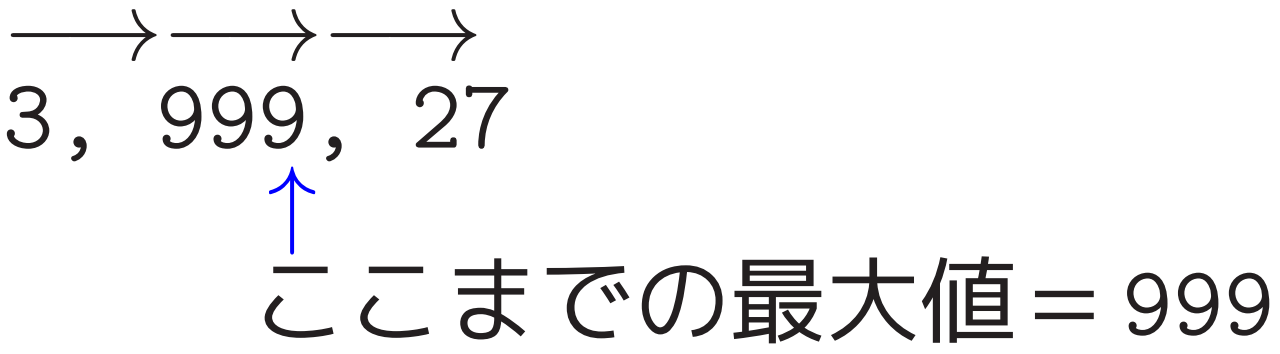


その際、常に「それまでに見た中での最大値」を保持する様にすれば、読み込んだ整数を全部眺め終った時点で、この保持データが求める最大値となっているはずである。

例題3．1（3つの数の最大値；if文,if-else構文,複合文）　整数を3つ読み込みその中の最大値を出力するCプログラムを作成せよ。

例題3.1に対するアルゴリズム(1)：

（考え方）　整数3つを読み込んだあとで、読み込んだ整数を順番に眺めていく。



その際、常に「それまでに見た中での最大値」を保持する様にすれば、読み込んだ整数を全部眺め終った時点で、この保持データが求める最大値となっているはずである。

例題3．1（3つの数の最大値；if文,if-else構文,複合文）　整数を3つ読み込みその中の最大値を出力するCプログラムを作成せよ。

例題3.1に対するアルゴリズム(1)：

（考え方）　整数3つを読み込んだあとで、読み込んだ整数を順番に眺めていく。

その際、常に「それまでに見た中での最大値」を保持する様にすれば、読み込んだ整数を全部眺め終った時点で、この保持データが求める最大値となっているはずである。

(プログラミング)

読み込んだ整数データを格納

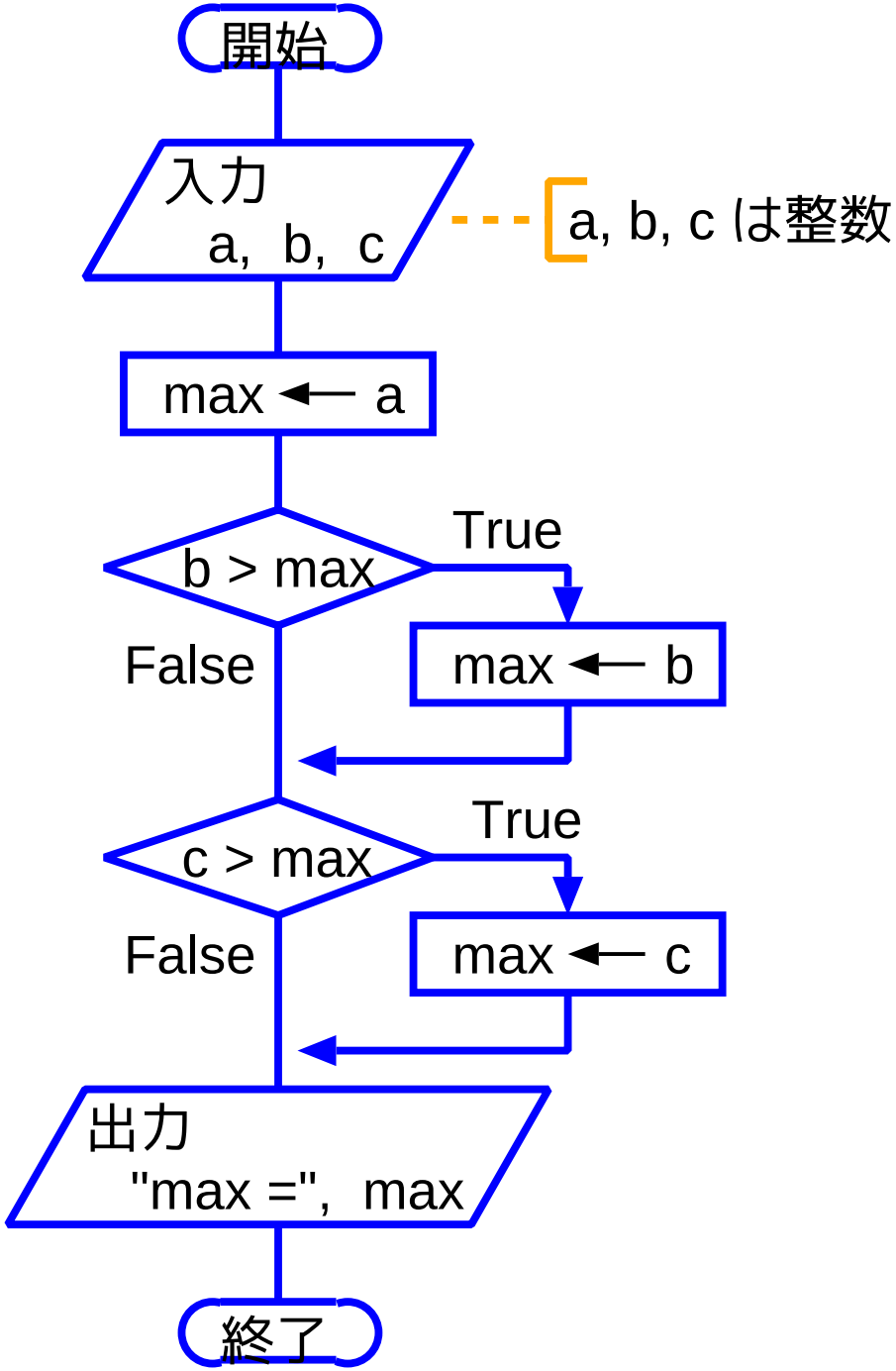
$\Longrightarrow$`a`, `b`, `c` という名前の変数

「それまでに見た中での最大値」を保持

$\Longrightarrow$`max` という名前の変数

`a`, `b`, `c` の順にデータを眺めることにすれば、

⇨ 行うべき処理は右図



```
[motoki@x205a]$  nl max-among-3-elem-no1.c   Enter
```

```
 1  /* ３つの入力データの最大値(その１) */

 2  #include <stdio.h>

 3  int main(void)
 4  {
 5    int  a, b, c, max;

 6    scanf("%d%d%d", &a, &b, &c);
 7
 8    max = a;
 9    if (max < b)
10      max = b;
11    if (max < c)
12      max = c;
```

if文

```
    13      printf("max = %d\n", max);
    14      return 0;
    15  }
[motoki@x205a]$  gcc max-among-3-elem-no1.c  Enter
[motoki@x205a]$  ./a.out  Enter
1 2 3  Enter
max = 3
[motoki@x205a]$
```

補足：

C言語においては、max < b といった条件を表す式の評価結果は true, false などの論理値ではなく整数値である。

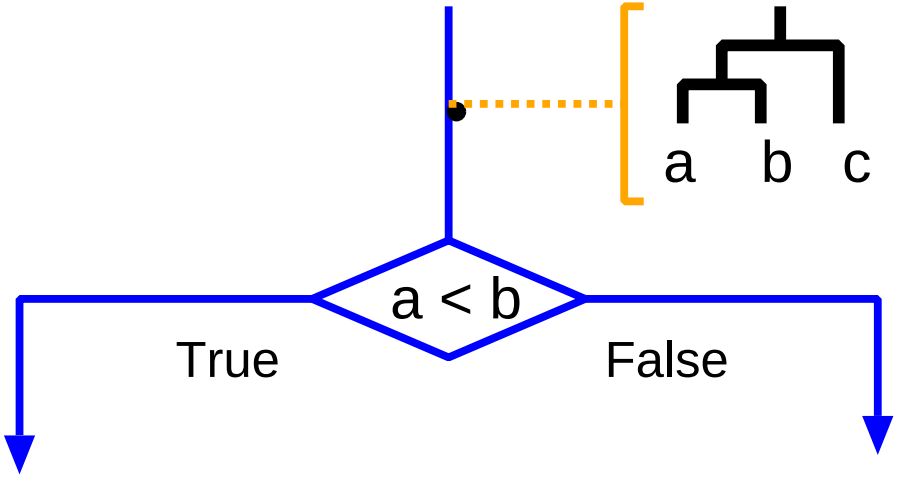
⇨ 例えば9~10行目においては、実際には、
条件 max < b が成立すればこの関係式の値は 1 と計算され、
成立しなければ 0 と計算される。

⇨ この関係式の値が 1 の時だけ10行目の代入文が実行される。

例題3.1に対するアルゴリズム(2):

(考え方)　整数3つを読み込んだあとで、最大要素が特定できるまで、読み込んだ整数 a, b, c 間の大小関係についての場合分けを重ねる。

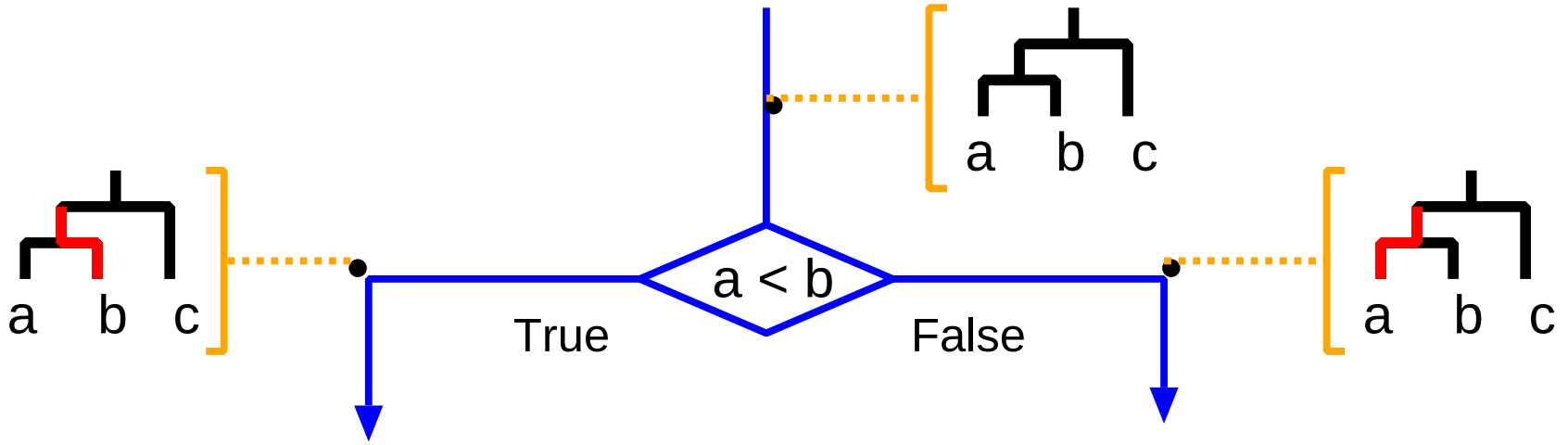
例えば 、まず a<b かどうかで場合分けし、.....



例題3.1に対するアルゴリズム(2)：

（考え方）　整数3つを読み込んだあとで、最大要素が特定できるまで、読み込んだ整数 a，b，c 間の大小関係についての場合分けを重ねる。

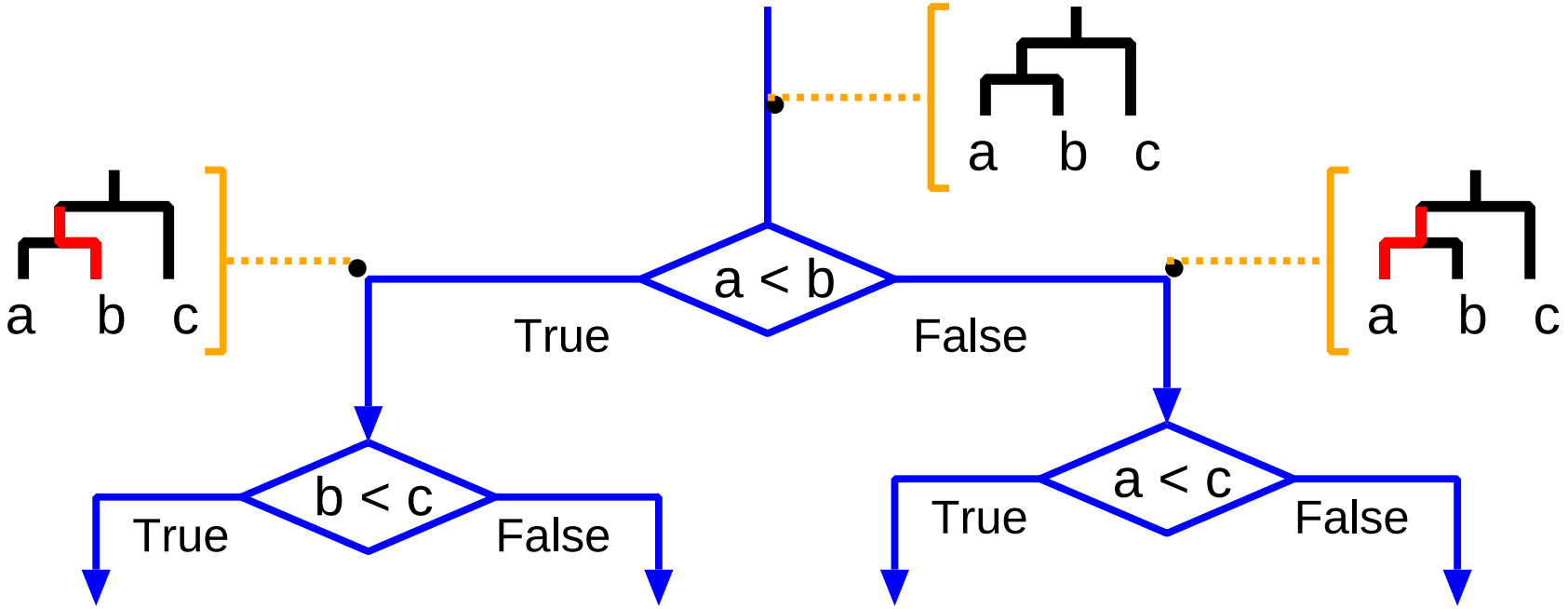
例えば 、まず a<b かどうかで場合分けし、. . . . . .



例題3.1に対するアルゴリズム(2)：

（考え方）　整数3つを読み込んだあとで、最大要素が特定できるまで、読み込んだ整数 a，b，c 間の大小関係についての場合分けを重ねる。

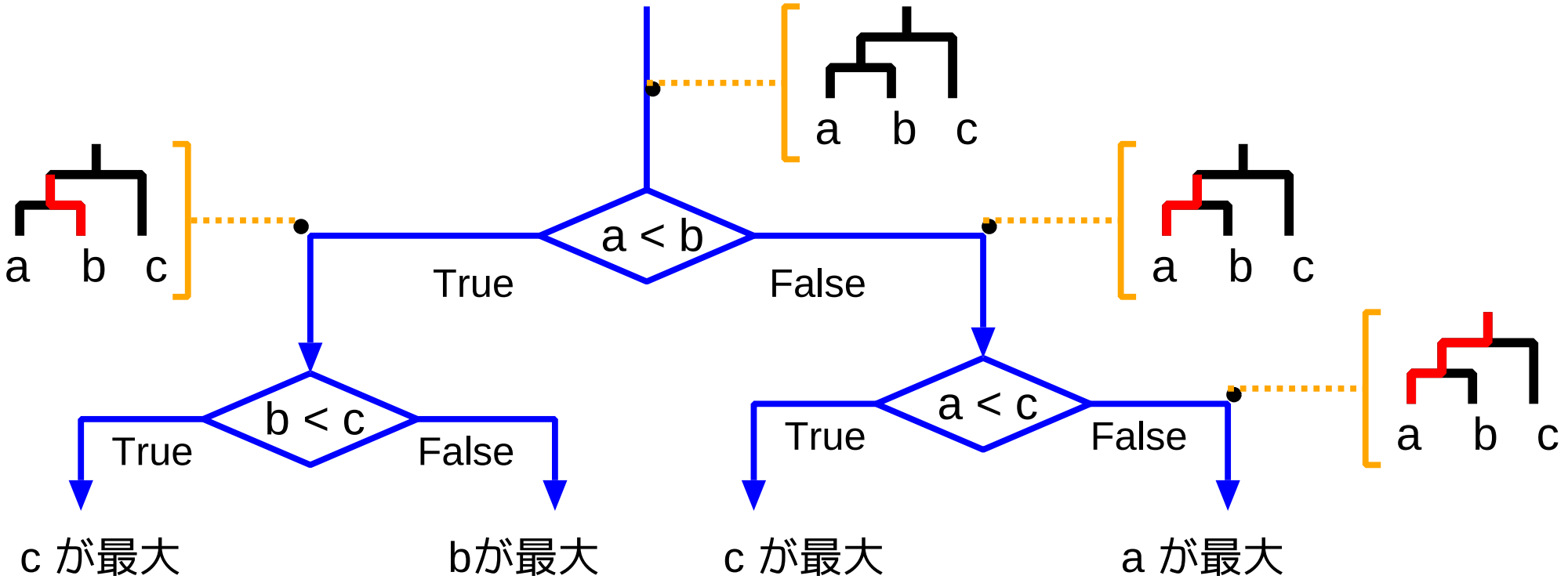
例えば 、まず a<b かどうかで場合分けし、. . . . . .



例題3.1に対するアルゴリズム(2)：

（考え方）　整数3つを読み込んだあとで、最大要素が特定できるまで、読み込んだ整数 a，b，c 間の大小関係についての場合分けを重ねる。

例えば 、まず a<b かどうかで場合分けし、. . . . . .

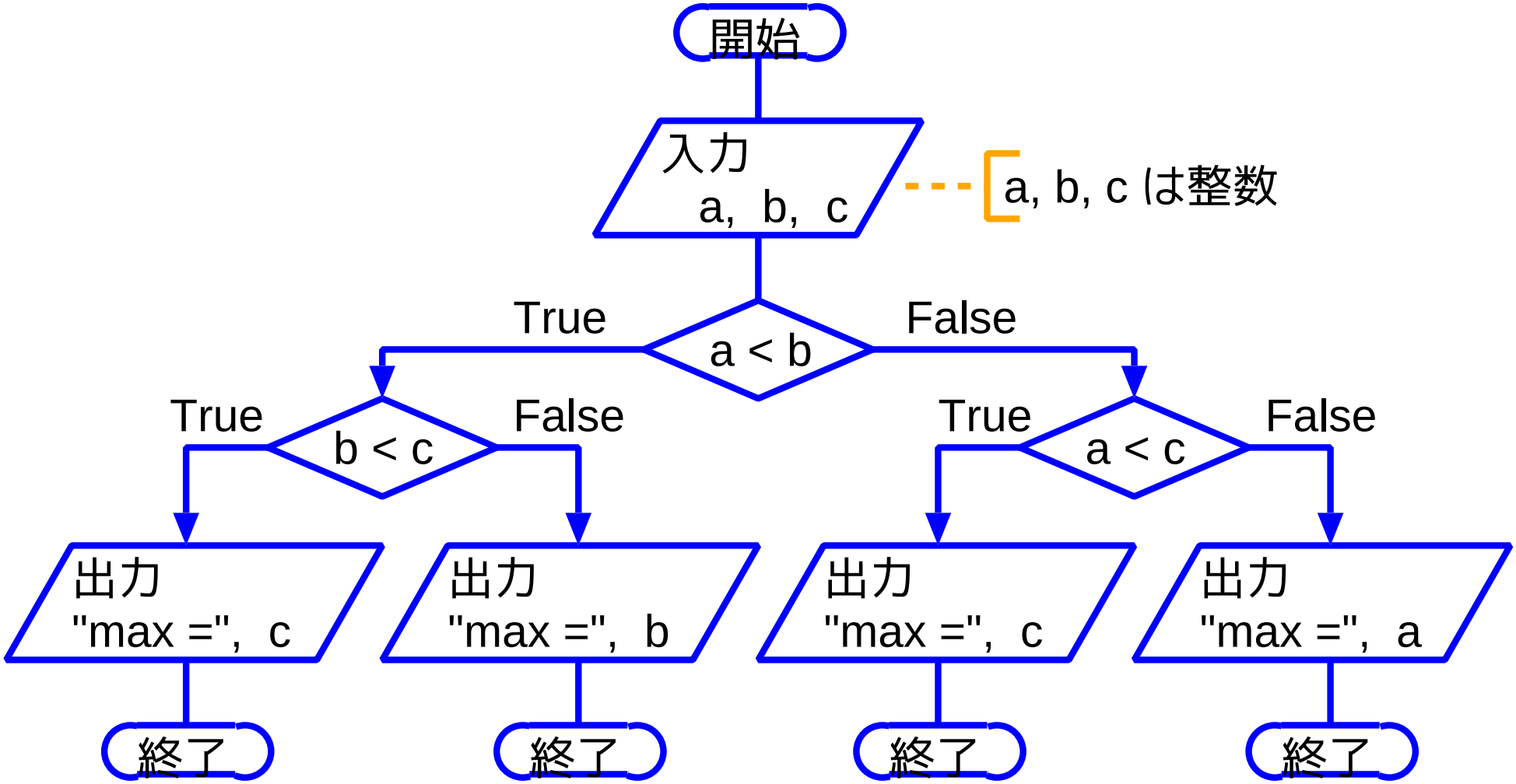


(プログラミング)

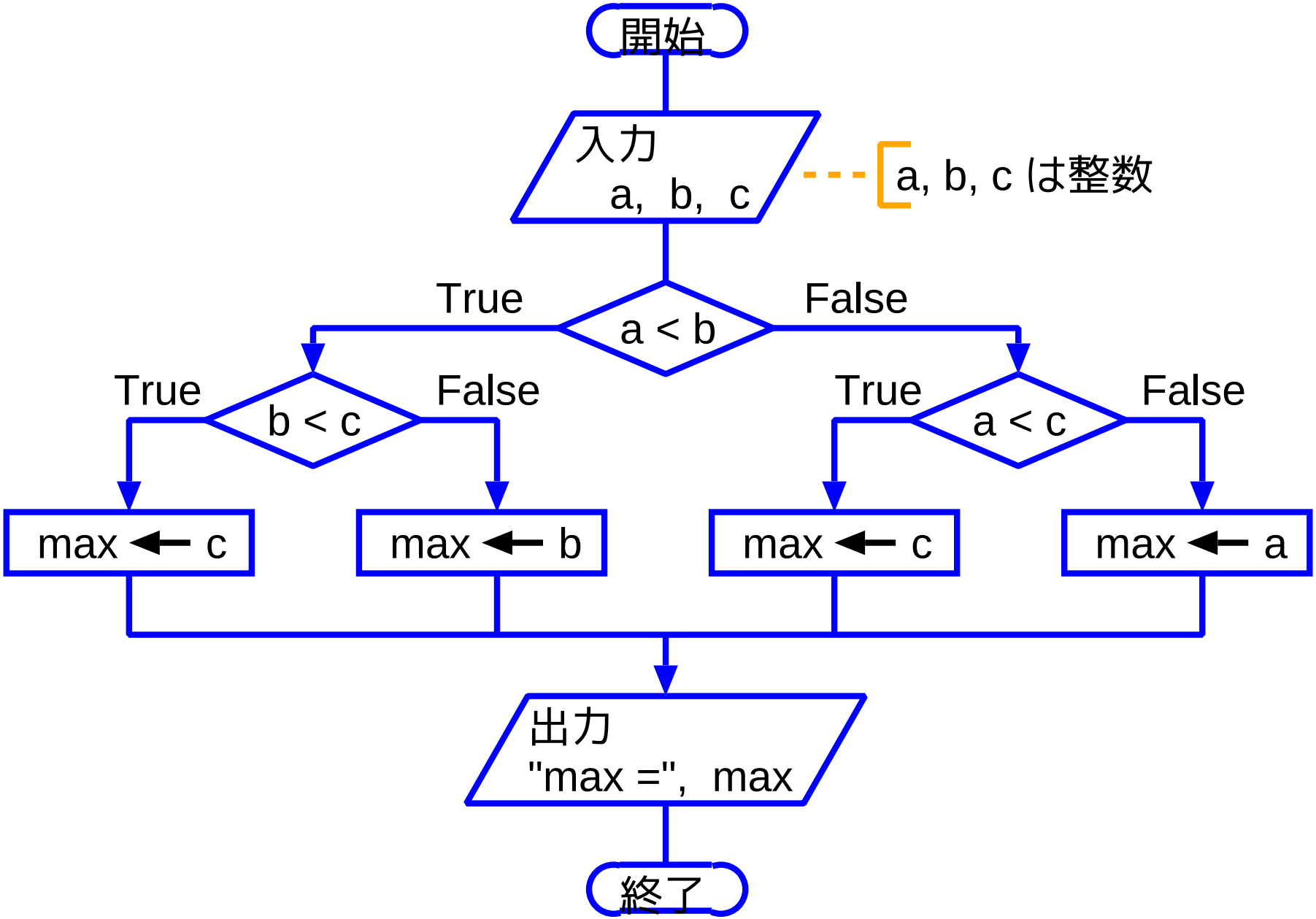
読み込んだ整数データを格納

$\Longrightarrow$ a, b, c という名前の変数を用意

⇨ 行うべき処理は



あるいは


開始
入力
a, b, c
a, b, c は整数
a < b
True
False
b < c
True
False
a < c
True
False
max ← c
max ← b
max ← c
max ← a
出力
"max =", max
終了

```
[motoki@x205a]$  nl max-among-3-elem-no2.c   Enter
     1  /* ３つの入力データの最大値(その２) */

     2  #include <stdio.h>

     3  int main(void)
     4  {
     5    int  a, b, c, max;

     6    scanf("%d%d%d", &a, &b, &c);
     7
     8    if (a < b){
     9      if (b < c)
    10        max = c;
    11      else
    12        max = b;
    13    } else{
```

複合文

if-else構文

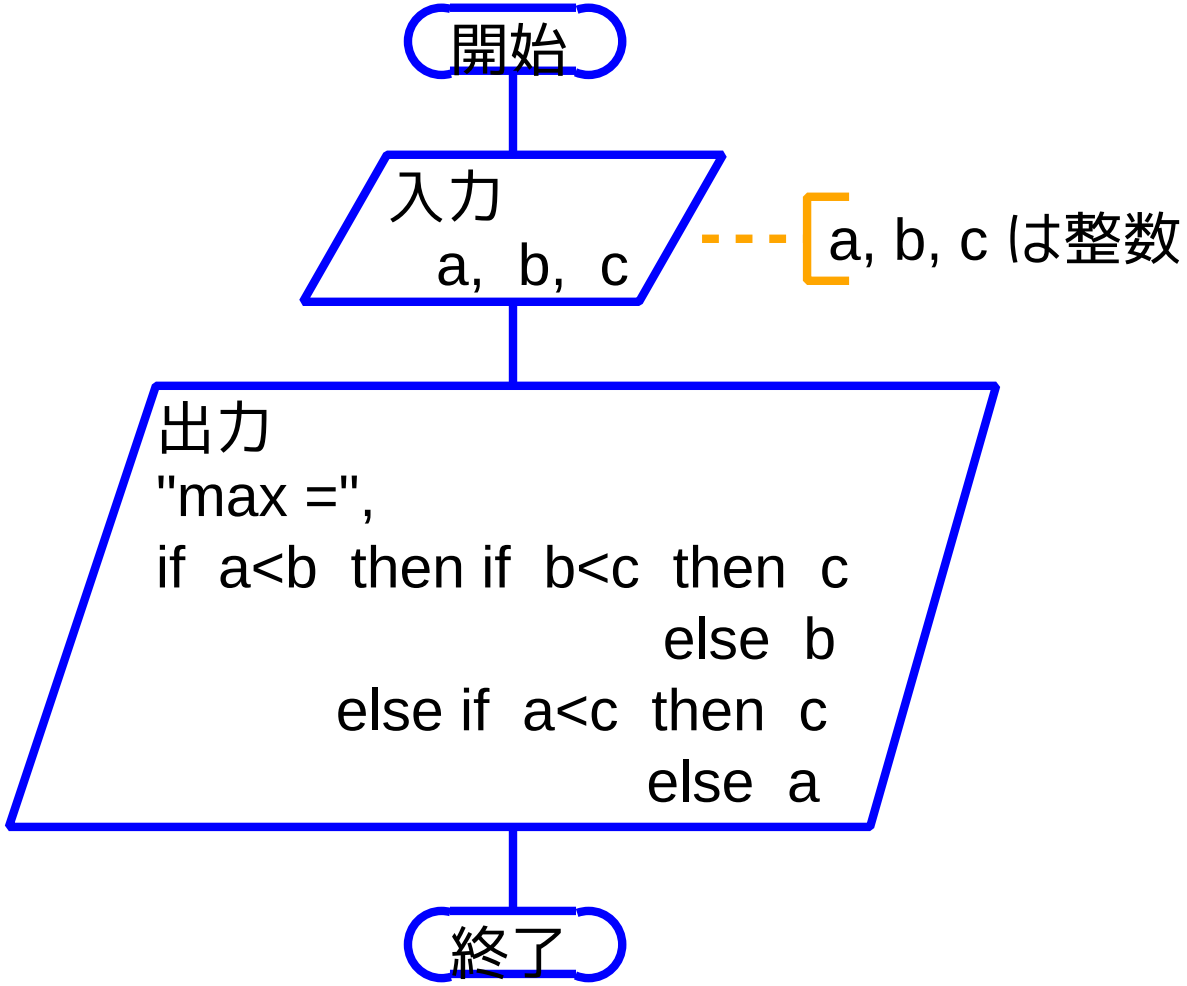
```
14        if (a < c)
15          max = c;
16        else
17          max = a;
18      }

19      printf("max = %d\n", max);
20      return 0;
21    }
```

```
[motoki@x205a]$  gcc max-among-3-elem-no2.c  Enter
[motoki@x205a]$  ./a.out  Enter
1 2 3  Enter
max = 3
[motoki@x205a]$
```

（プログラミング，別の方向）

c言語においては、計算式の中で条件分岐を表せる。

⇨ 最大要素が特定できるまで場合分けを重ねるという、
先程の処理手順はc言語で次の様に書き表すこともできる。



[motoki@x205a]$ nl max-among-3-elem-no4.c Enter

```
1  /* ３つの入力データの最大値(その４) */

2  #include <stdio.h>

3  int main(void)
4  {
5    int  a, b, c;

6    scanf("%d%d%d", &a, &b, &c);

7    printf("max = %d\n",
           (a<b) ? (b<c ? c : b) : (a<c ? c : a));
8    return 0;
9  }
```

条件演算子

式1 ? 式2 : 式3

```
[motoki@x205a]$  gcc max-among-3-elem-no4.c  Enter
[motoki@x205a]$  ./a.out  Enter
1 2 3  Enter
max = 3
[motoki@x205a]$
```

例題3.1に対するアルゴリズム(3):

（考え方） 読み込んだ整数 a, b, c 間の大小関係を調べることにより、各々の要素が最大であるかどうかを判定することができる。例えば、

aが最大 $\Longleftrightarrow$ a≥b かつ a≥c

⇨ 整数3つを読み込んだあとで、
まず a が最大かどうか で場合分けする。
その結果、

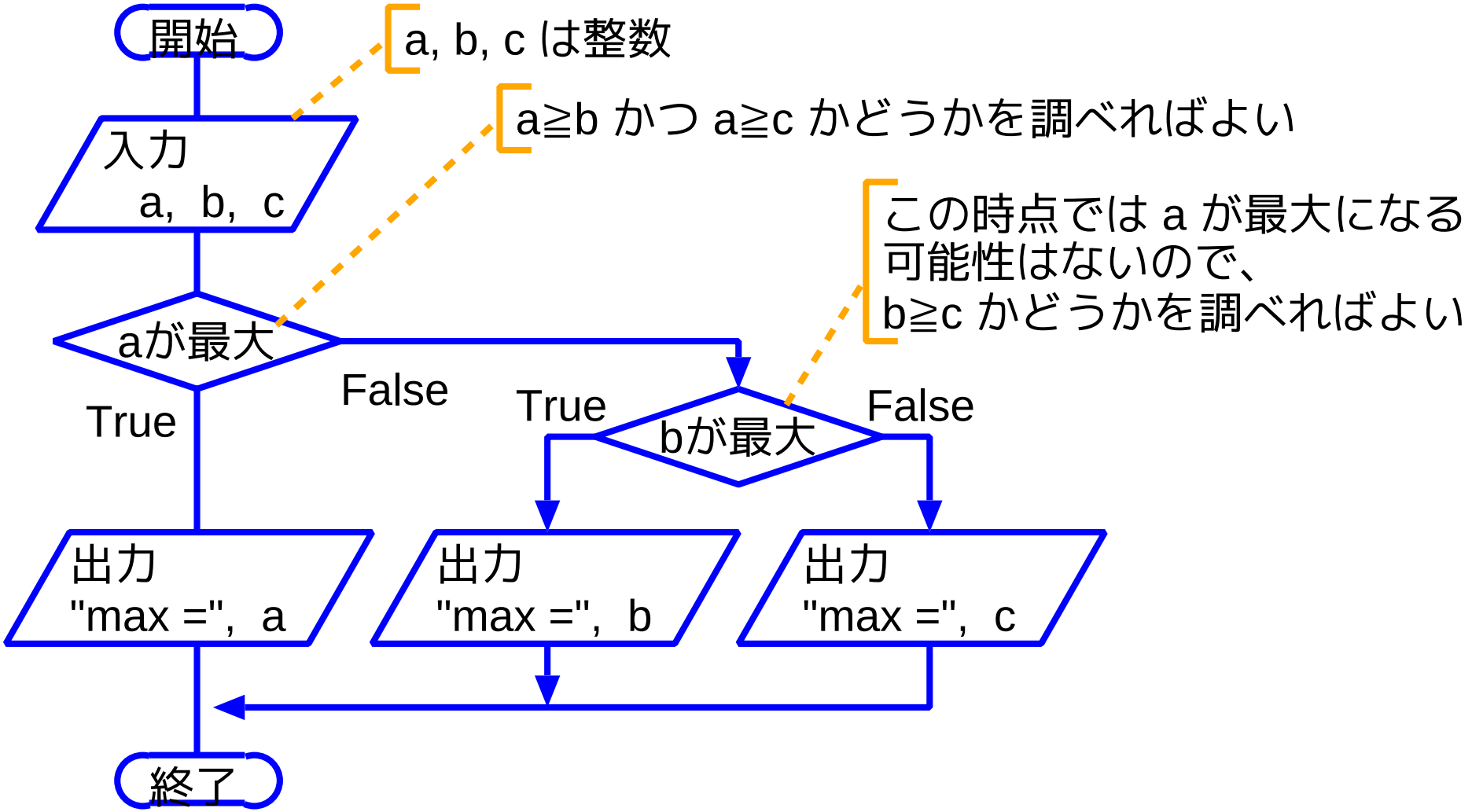
- aが最大 $\Longrightarrow$ aの値を出力
- そうでない $\Longrightarrow$ 残ったb,cの間で bが最大かどうか で場合分け
  その結果、
  - bが最大 $\Longrightarrow$ bの値を出力
  - そうでない $\Longrightarrow$ cの値を出力

(プログラミング)

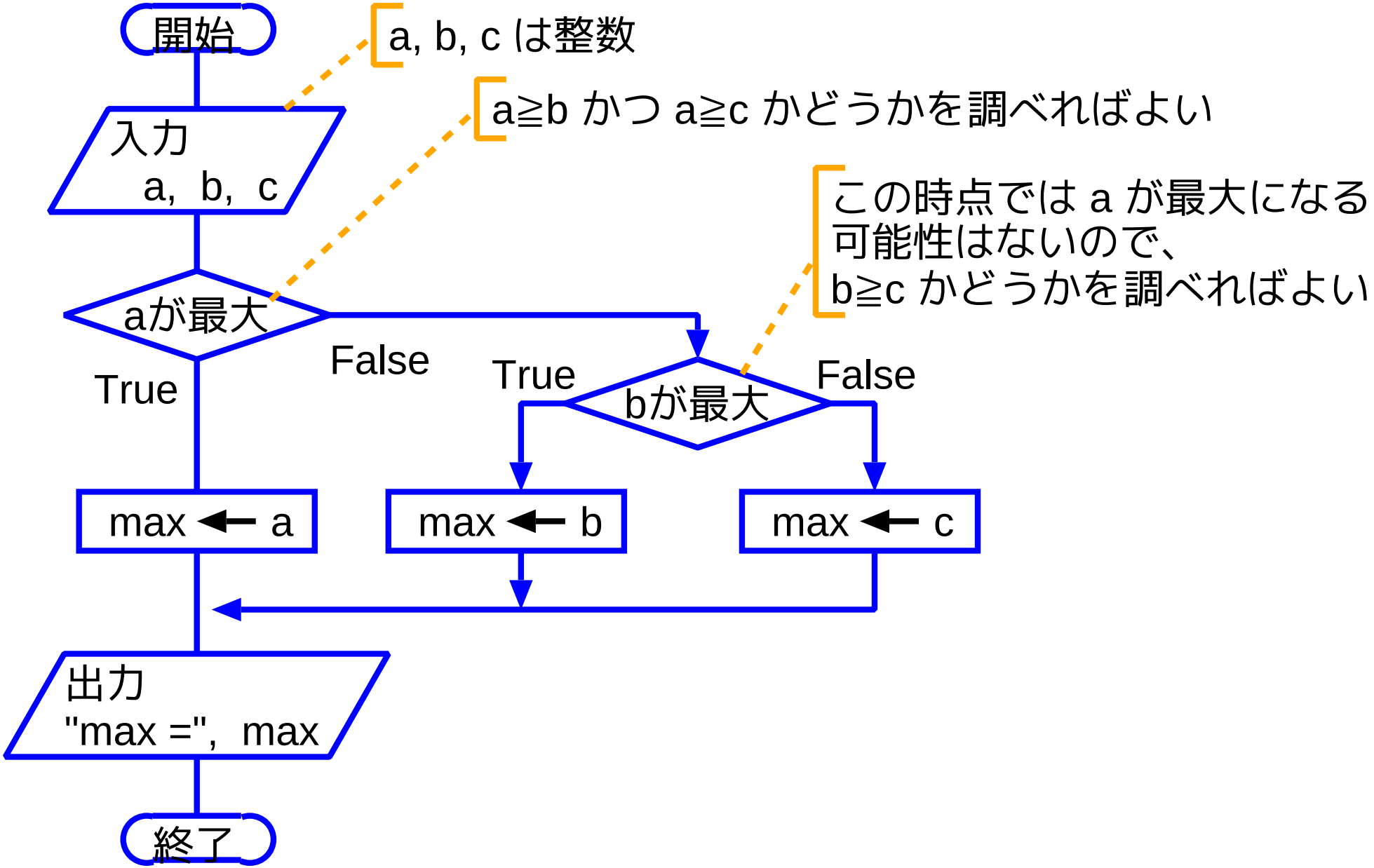
読み込んだ整数データを格納

$\Longrightarrow$ a, b, c という名前の変数を用意

⇨ 行うべき処理は次の図の様に書き表すことができる。



あるいは


開始
入力
a, b, c
aが最大
True
False
max ← a
bが最大
True
False
max ← b
max ← c
出力
"max =", max
終了
a, b, c は整数
a≧b かつ a≧c かどうかを調べればよい
この時点では a が最大になる
可能性はないので、
b≧c かどうかを調べればよい

```
[motoki@x205a]$  nl max-among-3-elem-no3.c   Enter
     1  /* ３つの入力データの最大値(その３) */

     2  #include <stdio.h>

     3  int main(void)
     4  {
     5    int  a, b, c, max;

     6    scanf("%d%d%d", &a, &b, &c);
     7
     8    if (b<=a && c<=a)
     9      max = a;
    10    else if (c<=b)
    11      max = b;
    12    else
    13      max = c;
```

論理積

```
    14      printf("max = %d\n", max);
    15      return 0;
    16  }
[motoki@x205a]$  gcc max-among-3-elem-no3.c  Enter
[motoki@x205a]$  ./a.out  Enter
1 2 3  Enter
max = 3
[motoki@x205a]$
```

C言語における論理式の扱い(概略):　　(⟹ 3.7.1節を参照)

- C言語には真理値(真と偽)を表すためのデータ型は用意されていない。

  ⟹ int型で代用。

  - 真 ··· 0以外(標準は1)
  - 偽 ··· 0

- 関係演算子として使えるのは <, <=, >, >=, ==, != の6つ。

  例えば　b*b-4*a*c>=0　や　x==0　といった関係式を...

- 論理演算子として使えるのは &&, ||, ! の3つで、それぞれ AND, OR, NOT を表す。

  例えば 、論理式

  ```
  a>0 && b>0 && c>0 && a+b>c && b+c>a && c+a>b
  ```

  は

  a>0 かつ b>0 かつ c>0 かつ a+b>c かつ b+c>a かつ c+a>b

  という意味であり、論理式

  ```
  !(a<=0 || b<=0 || c<=0)
  ```

  は

  (a<=0 または b<=0 または c<=0) でない

  という意味である。

- 式 p && q は左の条件から順に評価され、p の条件が不成立なら q の評価を行うことなく、p && q は不成立と判定される。

  同様に、p || q も左から順に評価され、pの条件が成立すればq の評価を行うことなく、p || q は成立と判定される。

  この様に式全体の評価値が確定した時点で評価を終える方式を短絡評価と言う。

論理式の値が整数として処理されるために、
本来は文法エラーとなるべき記述もチェックされない。例えば、

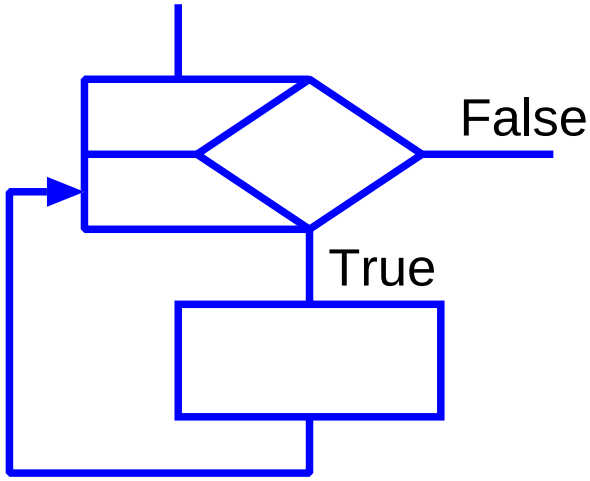
- 誤った記述例： `if (a=1) ...`　（正しくは　`if (a==1) ...`）
  条件部の「`a=1`」が代入式であるために、.....

- 誤った記述例： `if (-127<E<128) ...`
  （正しくは　`if (-127<E && E<128) ...`）
  条件部の「`-127<E<128`」が
  `-127<E<128` $\Longrightarrow$（式　`-127<E`　の評価結果）<128
  $\Longrightarrow$（0 または 1）$<$ 128
  $\Longrightarrow$ 1（真を表す）
  という風に式変形/計算されるので、.....

⇨ 上記のような誤りは見つけにくいので特に気を付けること。

3–2 処理の規則的な繰り返し

右図の形の繰り返しを制御する変数は加法的に変化させる場合が多いが、それ以外の規則的な変化のさせ方も可能である。 次にその例を示す。



例題3．2（べき乗；for文, コンマ演算子） 指数部が非負整数のべき乗を計算する際は、等式

$$x^y = \begin{cases} 1 & \text{if } y=0 \\ \left(x^2\right)^{\lfloor y/2 \rfloor} & \text{if } y \text{が} 2 \text{以上の偶数} \\ \left(x^2\right)^{\lfloor y/2 \rfloor} \times x & \text{if } y \text{が奇数} \end{cases}$$

に注目して左辺を右辺のように変形して計算する作業を繰り返せば、乗算の回数が少くて済む。実数データ x と 非負整数データ y を読み込み、この方法でべき乗 x^y を計算して出力するCプログラムを作成せよ。

（考え方）　与えられた等式に基づけば、我々は 2^{27} の計算を次のように進めることができる。

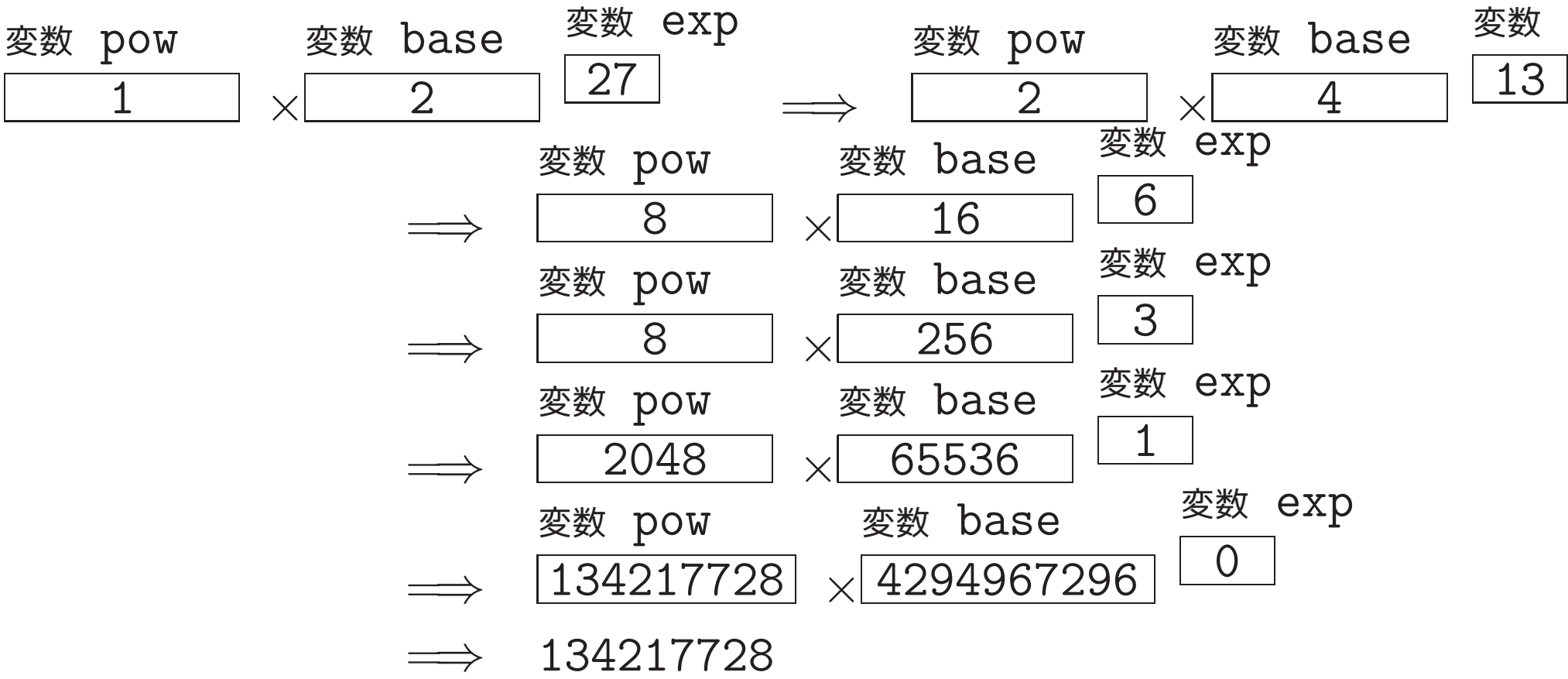
$$
\begin{aligned}
2^{27} &= 2\times 4^{13} && (y\text{が奇数なら } x^y=\left(x^2\right)^{\lfloor y/2\rfloor}\times x \text{ だから})\\
&= 2\times 4\times 16^{6} && (y\text{が奇数なら } x^y=\left(x^2\right)^{\lfloor y/2\rfloor}\times x \text{ だから})\\
&= 8\times 16^{6}\\
&= 8\times 256^{3} && (y\text{が偶数なら } x^y=\left(x^2\right)^{\lfloor y/2\rfloor} \text{ だから})\\
&= 8\times 256\times 65536^{1} && (y\text{が奇数なら } x^y=\left(x^2\right)^{\lfloor y/2\rfloor}\times x \text{ だから})\\
&= 2048\times 65536^{1}\\
&= 2048\times 65536\times 4294967296^{0}\\
& && (y\text{が奇数なら } x^y=\left(x^2\right)^{\lfloor y/2\rfloor}\times x \text{ だから})\\
&= 134217728\times 4294967296^{0}\\
&= 134217728\times 1 && (y=0\text{なら } x^y=1 \text{ だから})\\
&= 134217728
\end{aligned}
$$

では、この場合どんな変数を用意すれば良いのか? この計算は結局は

$$1\times 2^{27} \Longrightarrow 2\times 4^{13} \Longrightarrow 8\times 16^{6} \Longrightarrow 8\times 256^{3} \Longrightarrow \cdots\cdots$$

という式変形を行っているだけである。

式変形の現在の状態 $\boxed{\cdots}\times\boxed{\cdots}\ \boxed{\cdots}$ を認識するために3つの $\boxed{\cdots}$ の数値を記憶する変数を用意し、各々 pow, base, exp という名前を付けることにすれば、先の計算例における変数の更新は次の様に進む。



これらの変数値更新のために実際にどういう処理を行えば良いのか?

1つの式変形の状態から次の状態への更新は、ほとんどの場合次の様に進む。

変数 pow: a × 変数 base: b 変数 exp: c

$$\Longrightarrow$$ 変数 pow: if odd(c) then a×b else a × 変数 base: b^2 変数 exp: $\lfloor c/2 \rfloor$

この更新を行うには、

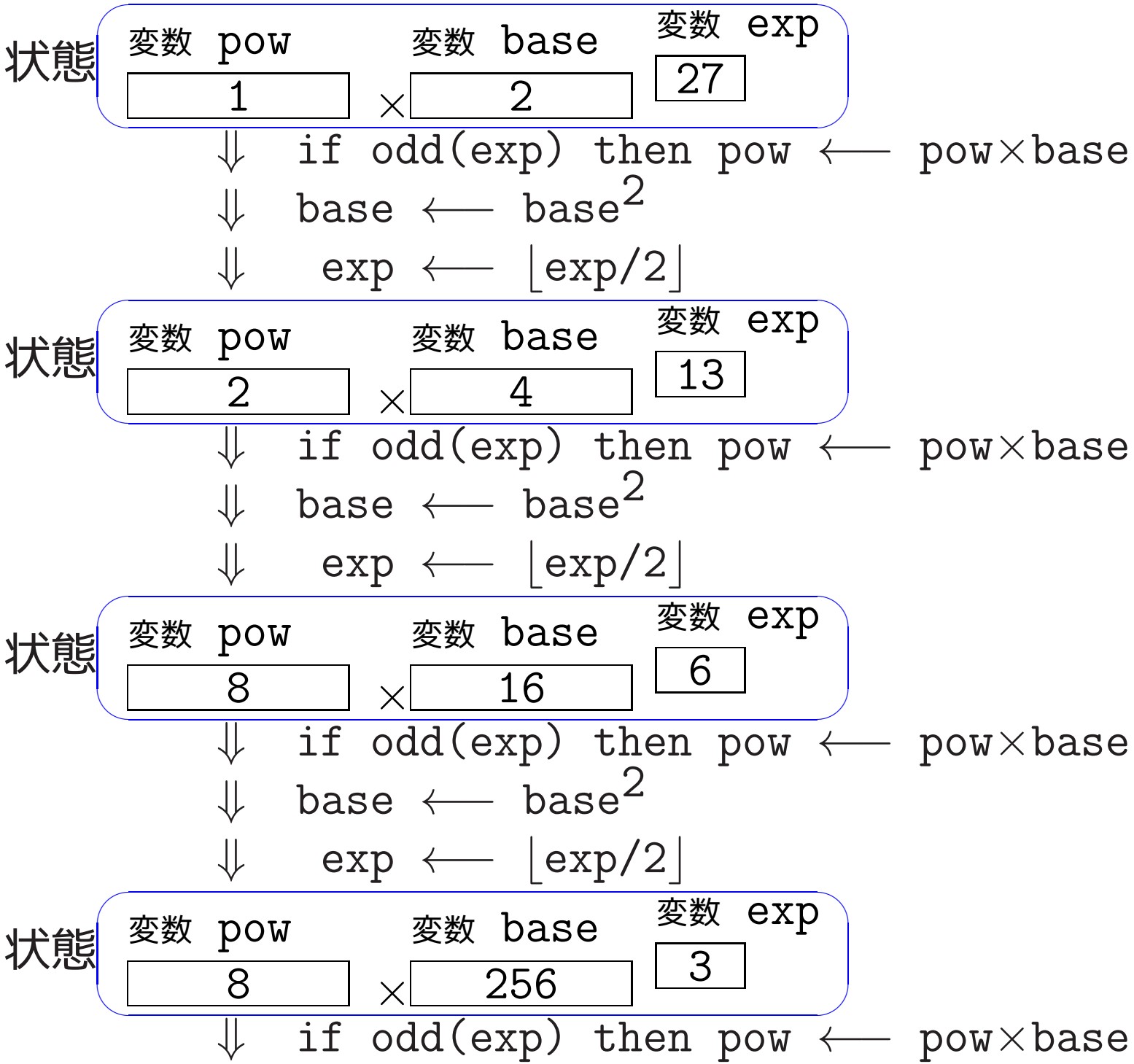
```
if odd(exp) then pow ⟵ pow×base
base ⟵ base^2
exp ⟵ ⌊exp/2⌋
```

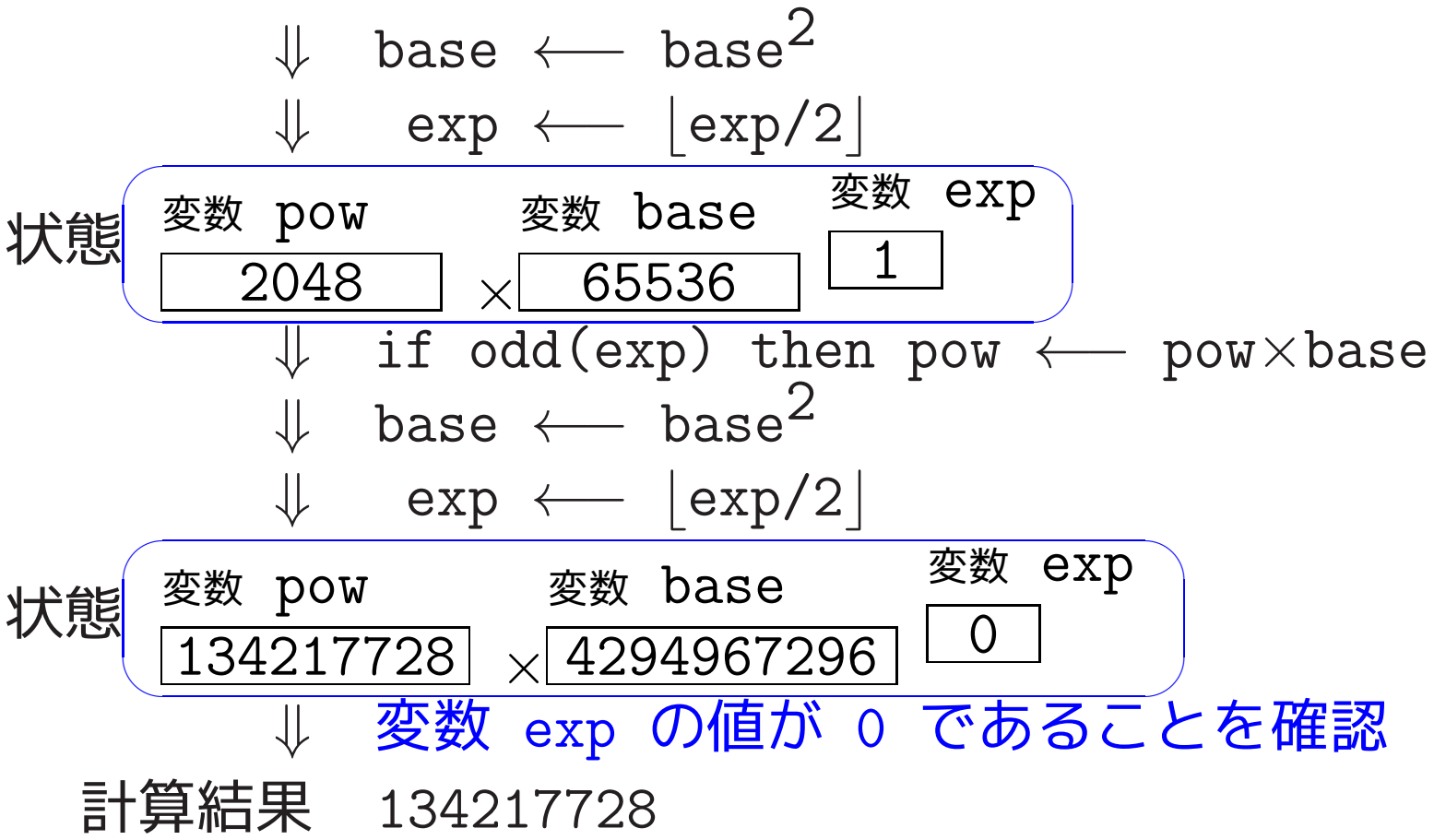
とすれば良い。　また、最後の

変数 pow: 134217728 × 変数 base: 4294967296 変数 exp: 0 $\Longrightarrow$ 134217728

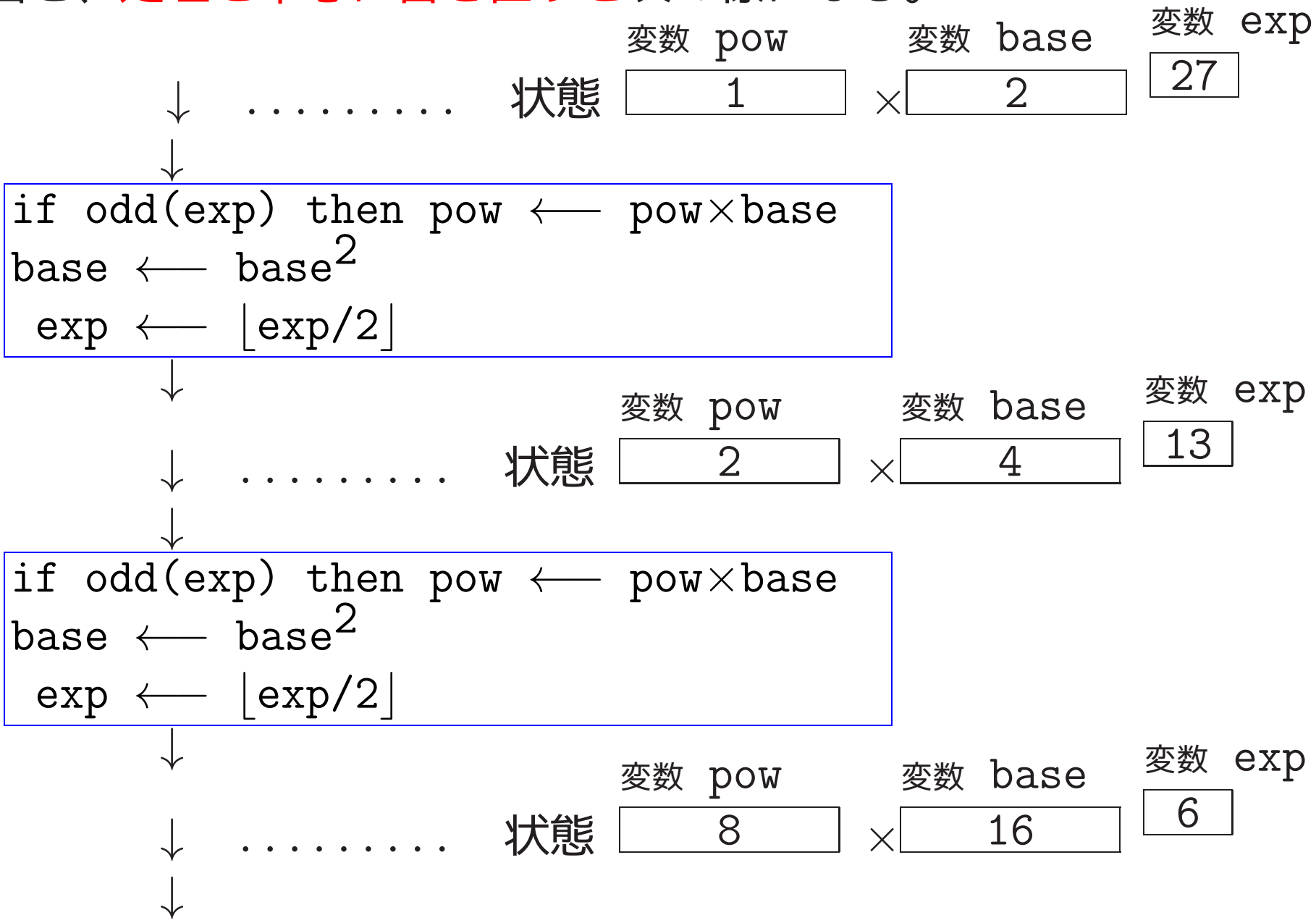
という式変形は、変数 exp の値が 0 なので起こっていると考えられる。

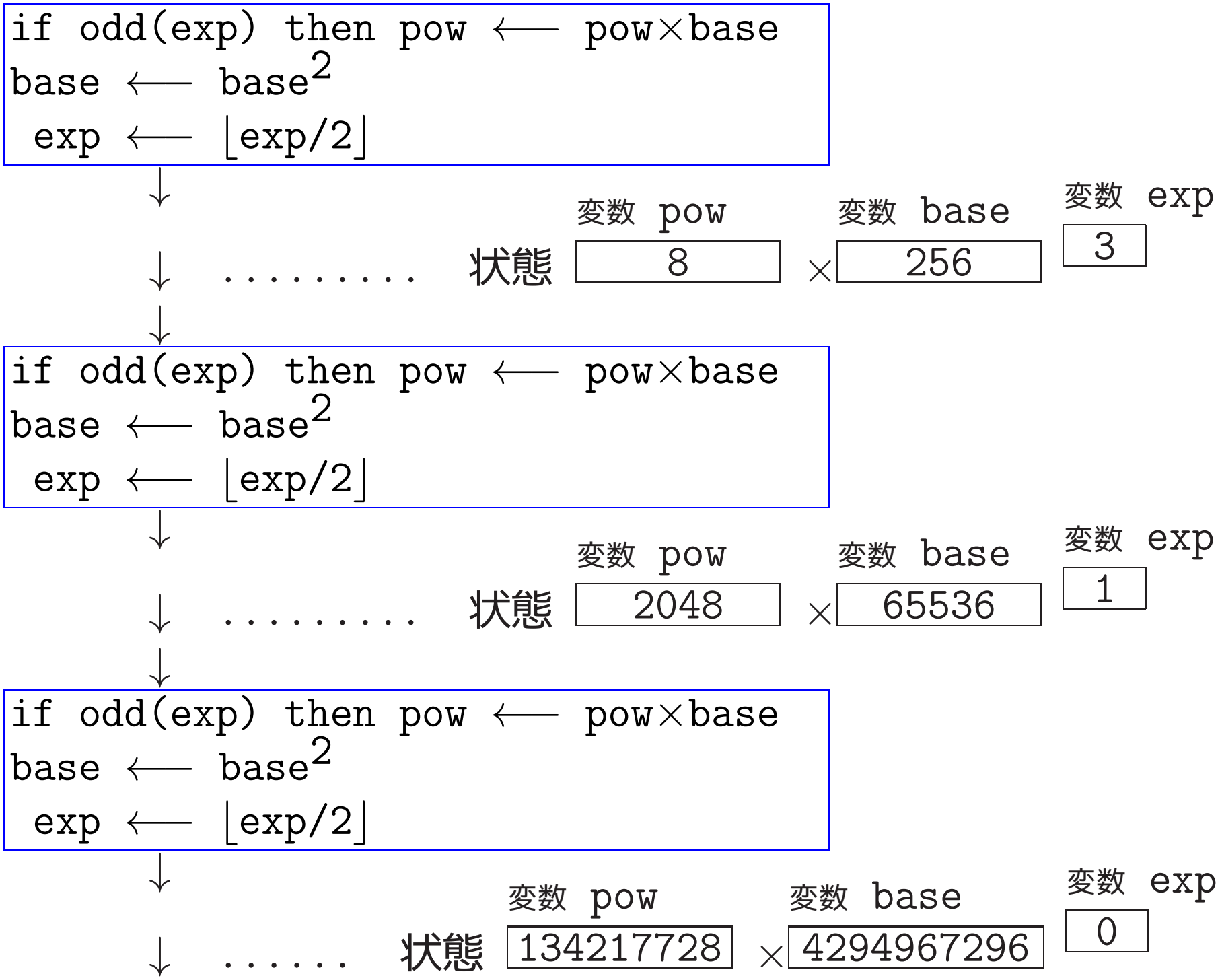
結局、2^{27} の計算の場合に、どういう処理によってどういう風に状態が変わっていくかを具体的に明示すると次の様になる。

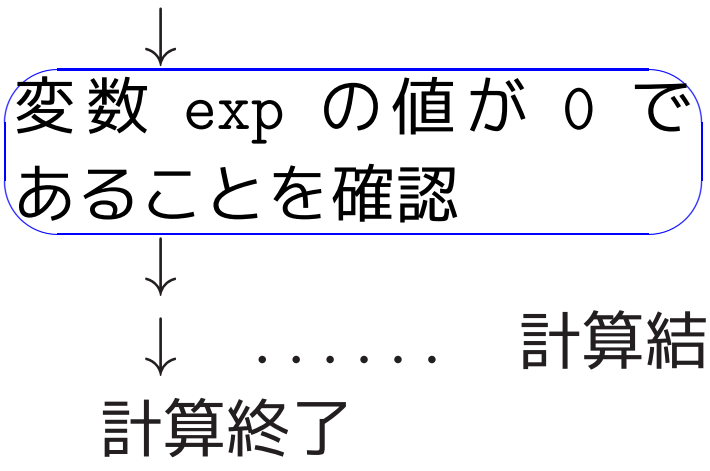

状態
変数 pow
1
×
変数 base
2
変数 exp
27
⇓ if odd(exp) then pow ⟵ pow×base
⇓ base ⟵ base^2
⇓ exp ⟵ ⌊exp/2⌋
状態
変数 pow
2
×
変数 base
4
変数 exp
13
⇓ if odd(exp) then pow ⟵ pow×base
⇓ base ⟵ base^2
⇓ exp ⟵ ⌊exp/2⌋
状態
変数 pow
8
×
変数 base
16
変数 exp
6
⇓ if odd(exp) then pow ⟵ pow×base
⇓ base ⟵ base^2
⇓ exp ⟵ ⌊exp/2⌋
状態
変数 pow
8
×
変数 base
256
変数 exp
3
⇓ if odd(exp) then pow ⟵ pow×base


⇓ base ⟵ base^2
⇓ exp ⟵ ⌊exp/2⌋
状態
変数 pow
2048
×
変数 base
65536
変数 exp
1
⇓ if odd(exp) then pow ⟵ pow×base
⇓ base ⟵ base^2
⇓ exp ⟵ ⌊exp/2⌋
状態
変数 pow
134217728
×
変数 base
4294967296
変数 exp
0
⇓ 変数 exp の値が 0 であることを確認
計算結果 134217728

(プログラミング)　上述の、状態とそれらの間の遷移を引き起こす処理の関係図を、処理を中心に書き直すと次の様になる。




if odd(exp) then pow ⟵ pow×base
base ⟵ base²
exp ⟵ ⌊exp/2⌋
状態
変数 pow
8
×
変数 base
256
変数 exp
3
if odd(exp) then pow ⟵ pow×base
base ⟵ base²
exp ⟵ ⌊exp/2⌋
状態
変数 pow
2048
×
変数 base
65536
変数 exp
1
if odd(exp) then pow ⟵ pow×base
base ⟵ base²
exp ⟵ ⌊exp/2⌋
状態
変数 pow
134217728
×
変数 base
4294967296
変数 exp
0



要するに、exp=0 となるまで

```
if odd(exp) then pow ⟵ pow×base
base ⟵ base^2
 exp ⟵ ⌊exp/2⌋
```

という固定的な処理を繰り返すだけである。

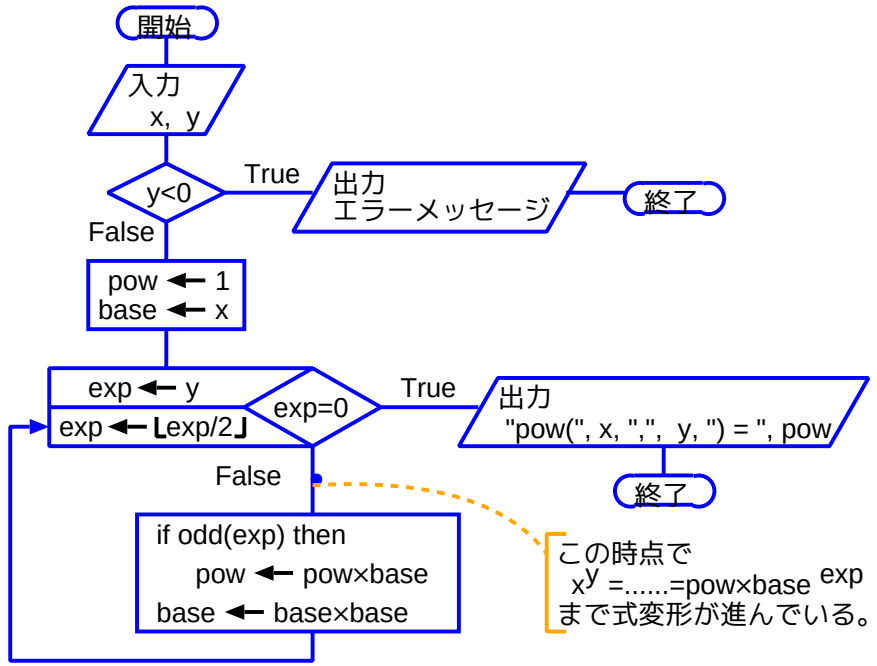
変数 exp の値が 0 になった時点では、
計算結果は変数 pow に保持されている。

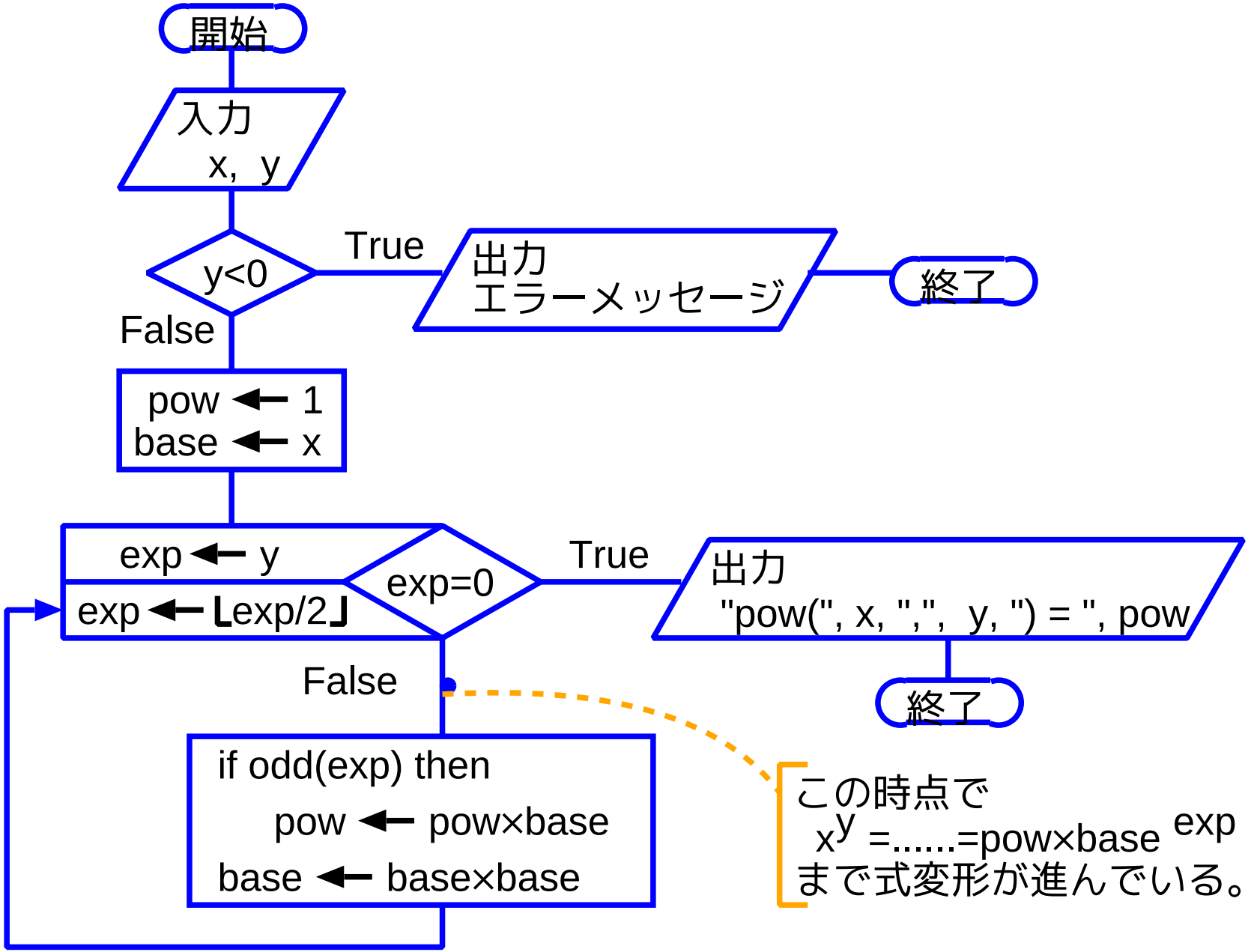
一般に、べき乗 x^y を計算する場合 、変数 exp の値は

$$y \longrightarrow \lfloor y/2 \rfloor \longrightarrow \lfloor y/2^2 \rfloor \longrightarrow \lfloor y/2^3 \rfloor \longrightarrow \cdots \longrightarrow 1 \longrightarrow 0$$

と確定的に変わり、この値が0になれば繰り返しを終了するので、この変数 exp は繰り返しを制御する変数として働く。

それゆえ、読み込んだ実数データ,非負整数データを格納するために各々 x, y という名前の変数を、 式変形の際の $\boxed{\cdots} \times \boxed{\cdots}^{\boxed{\cdots}}$ の 1番目,2番目,3番目の $\boxed{\cdots}$ の数値 を保持するためにするために各々 pow base, exp という名前の変数を用意することにすれば、...




開始
入力
x, y
y<0
True
出力
エラーメッセージ
終了
False
pow ← 1
base ← x
exp ← y
exp ← ⌊exp/2⌋
exp=0
True
出力
"pow(", x, ",", y, ") = ", pow
終了
False
if odd(exp) then
pow ← pow×base
base ← base×base
この時点で
x^y =.....=pow×base^exp
まで式変形が進んでいる。

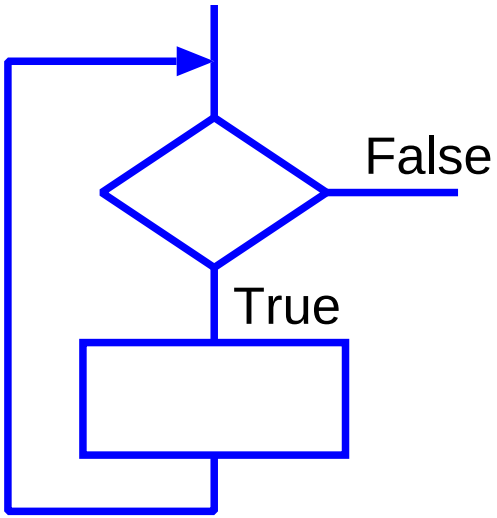
```
[motoki@x205a]$  nl power-function.c   Enter
     1 /* 実数 x と非負整数 y を読み込み、                    */
     2 /* べき乗値 x^y を計算・出力するCプログラム */

     3 #include <stdio.h>
     4 #include <stdlib.h>

     5 int main(void)
     6 {
     7   double x, pow, base;
     8   int    y, exp;

     9   printf("x の y 乗を計算をします。\n"
    10       "実数 x と非負整数 y をこの順に入力して下さい:  ");
    11   scanf("%lf %d", &x, &y);
    12   if (y < 0) {
    13     printf("Input Error!\n");
```

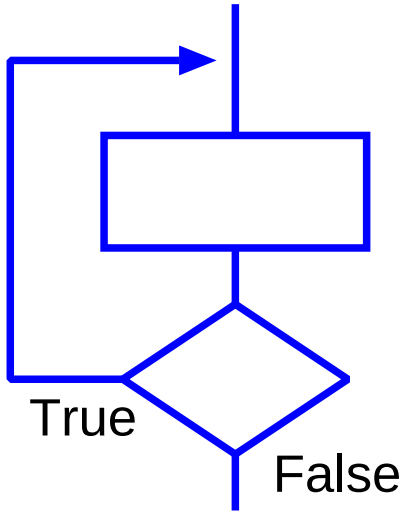
```
14      exit(EXIT_FAILURE);
15    }

16    pow  = 1.0;
17    base = x;

18    for (exp=y; exp>0; exp/=2) {
19                              /* この時点で x^y=...=pow*base^exp
20      if (exp%2 == 1)         /* まで式変形が進んでいる。          *
21        pow *= base;
22      base *= base;
23    }

24    printf("pow(%g, %d) = %.16g\n", x, y, pow);
25    return 0;
26  }
```

[motoki@x205a]$ gcc power-function.c Enter

```
[motoki@x205a]$  ./a.out  Enter
x の y 乗を計算をします。
実数 x と非負整数 y をこの順に入力して下さい:   2.0  27  Enter
pow(2, 27) = 134217728
[motoki@x205a]$  ./a.out  Enter
x の y 乗を計算をします。
実数 x と非負整数 y をこの順に入力して下さい:   8.5  50  Enter
pow(8.5, 50) = 2.957646637126993e+46
[motoki@x205a]$  ./a.out  Enter
x の y 乗を計算をします。
実数 x と非負整数 y をこの順に入力して下さい:   8.5  -5  Enter
Input Error!
[motoki@x205a]$
```

3–3 条件判断による処理の繰り返し





例題3．3（最大公約数，ユークリッドの互除法；while文，do-while文）
2つの正整数を読み込み、それらの最大公約数を出力するCプログラムを作成せよ。

（考え方）　ユークリッドの互除法（あるいはユークリッドのアルゴリズム）と呼ばれるものが有名である。

このアルゴリズムは次の事実に基づいて計算を進める。

> 命題3．4　　2つの正整数 a,b の最大公約数を gcd(a,b)、整数 b を整数 a で割った時の余りを mod(b,a) と表すことにすれば、
> (1) a<b なら gcd(a,b) = gcd(a,b−a)
> (2) gcd(a,b) = gcd(mod(b,a), a)

この命題に基づけば 、我々は 1596 と 308 の最大公約数 gcd(1596,308) の計算を次のように進めることができる。

命題3．4　**2**つの正整数 **a,b** の最大公約数を **gcd(a,b)**、整数 **b** を整数 **a** で割った時の余りを **mod(b,a)** と表すことにすれば、
(1) a<b なら **gcd(a,b) = gcd(a,b-a)**
(2) gcd(a,b) = gcd(mod(b,a), a)

```
gcd(1596, 308) = gcd(mod(308,1596), 1596)  命題6.13(2)より
               = gcd(308, 1596)
               = gcd(mod(1596,308), 308)   命題6.13(2)より
               = gcd(56, 308)
               = gcd(mod(308,56), 56)      命題6.13(2)より
               = gcd(28, 56)
               = gcd(mod(56,28), 28)       命題6.13(2)より
               = gcd(0, 28)
               = 28
```

では、この場合どんな変数を用意すれば良いのか?

この計算は、結局は

gcd(1596, 308) $\Longrightarrow$ gcd(308, 1596) $\Longrightarrow$ gcd(56, 308) $\Longrightarrow$ ..

という式変形を行っているだけである。

式変形の現在の状態 gcd(..., ...) を認識するために gcd( ) の第1引数，第2引数を記憶する変数を用意し、各々 x, y という名前を付ける...

変数 x 変数 y

gcd(1596 , 308) $\Longrightarrow$ gcd(308 , 1596)

変数 x 変数 y

$\Longrightarrow$ gcd(56 , 308)

変数 x 変数 y

$\Longrightarrow$ gcd(28 , 56)

変数 x 変数 y

$\Longrightarrow$ gcd(0 , 28)

$\Longrightarrow$ 28

これらの変数値更新のために実際にどういう処理を行えば良いのか?

1つの式変形の状態から次の状態への更新は、ほとんどの場合次の様に進む。

変数 x　変数 y

gcd(a , b) $\Longrightarrow$ gcd(mod(b,a) , a)

変数 x　変数 y

この更新を行うには、例えば next_x という名前の変数を用意して

```
next_x ⟵ mod(y,x)
     y ⟵ x
     x ⟵ next_x
```

とすれば良い。 [変数への代入は一般には同時に行えないので、...]

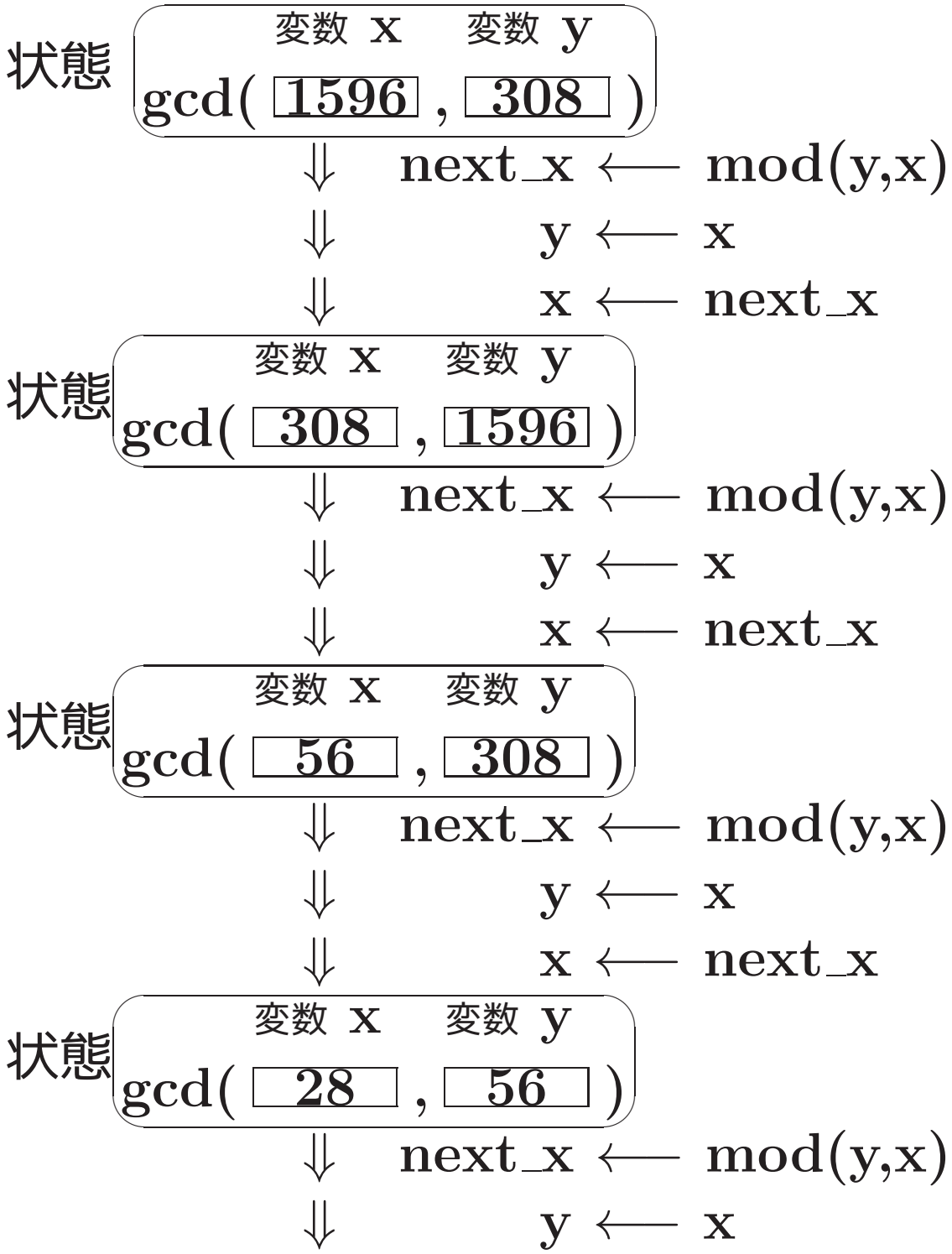
また、最後の

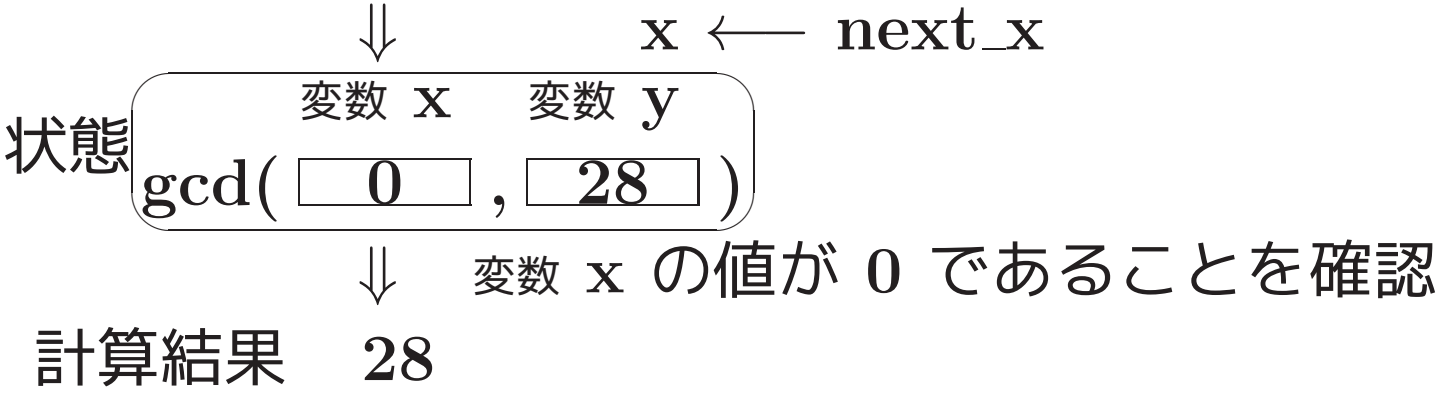
変数 x　変数 y

gcd(0 , 28) $\Longrightarrow$ 28

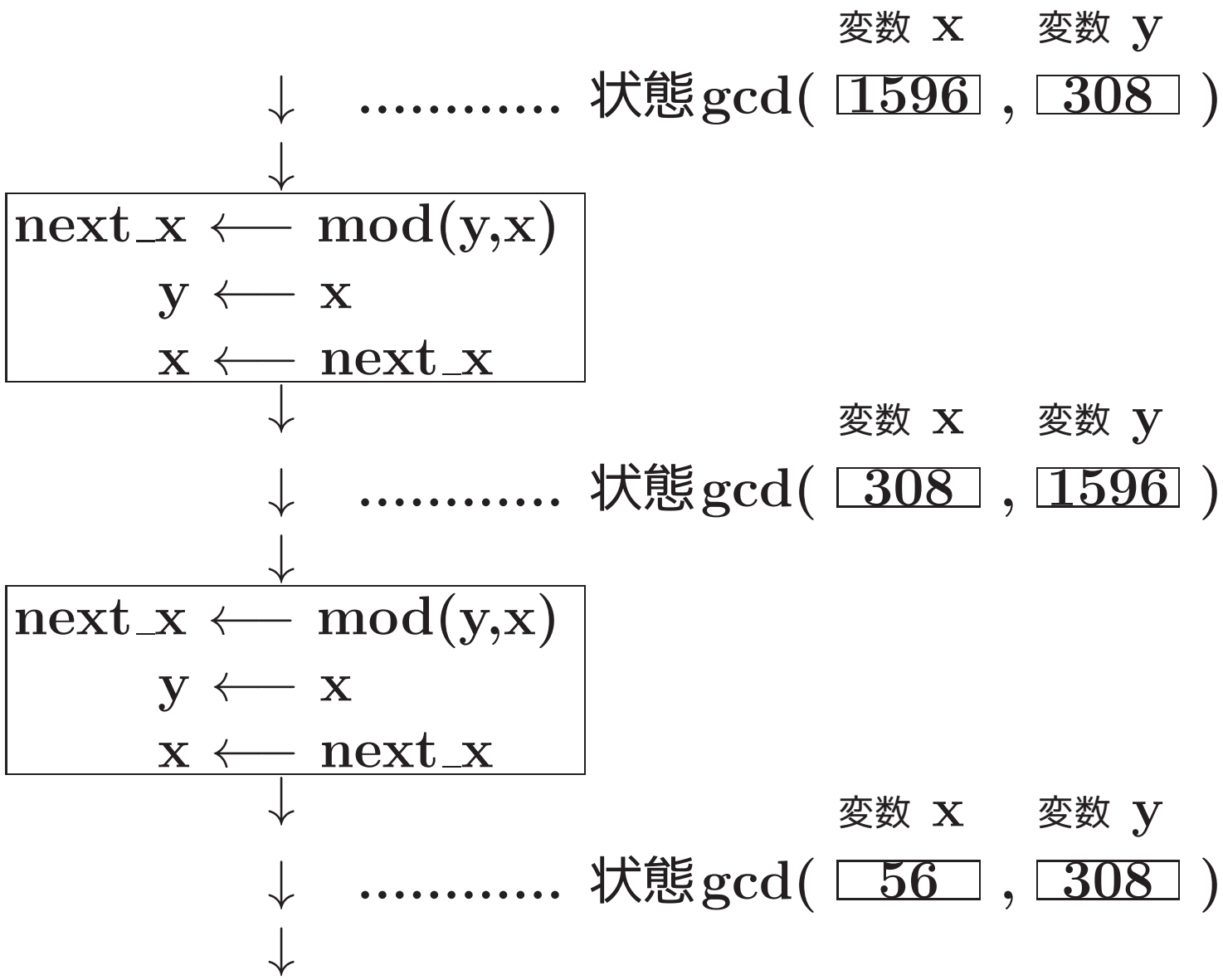
という式変形は、変数 x の値が 0 なので起こっていると考えられる。

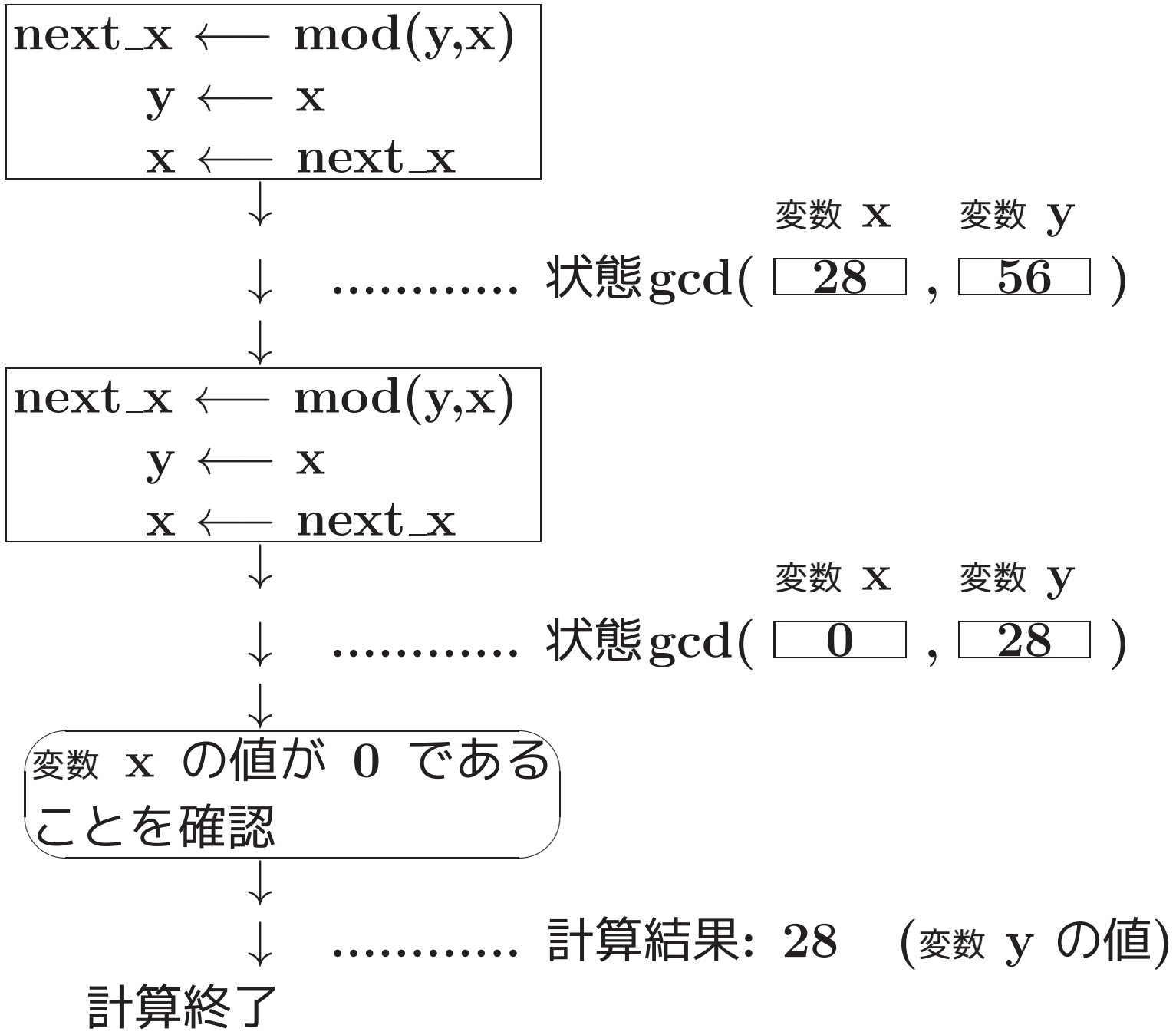
結局、gcd(1596,308) の計算の場合に、どういう処理によってどういう風に状態が変わっていくかを具体的に明示すると次の様になる。


状態
変数 x
変数 y
gcd(1596 , 308)
⇓ next_x ⟵ mod(y,x)
⇓ y ⟵ x
⇓ x ⟵ next_x
状態
変数 x
変数 y
gcd(308 , 1596)
⇓ next_x ⟵ mod(y,x)
⇓ y ⟵ x
⇓ x ⟵ next_x
状態
変数 x
変数 y
gcd(56 , 308)
⇓ next_x ⟵ mod(y,x)
⇓ y ⟵ x
⇓ x ⟵ next_x
状態
変数 x
変数 y
gcd(28 , 56)
⇓ next_x ⟵ mod(y,x)
⇓ y ⟵ x


⇓
x ⟵ next_x
変数 x
変数 y
状態
gcd(0 , 28)
⇓
変数 x の値が 0 であることを確認
計算結果 28

（プログラミング）　上述の、状態とそれらの間の遷移を引き起こす処理の関係図を、処理を中心に書き直すと次の様になる。



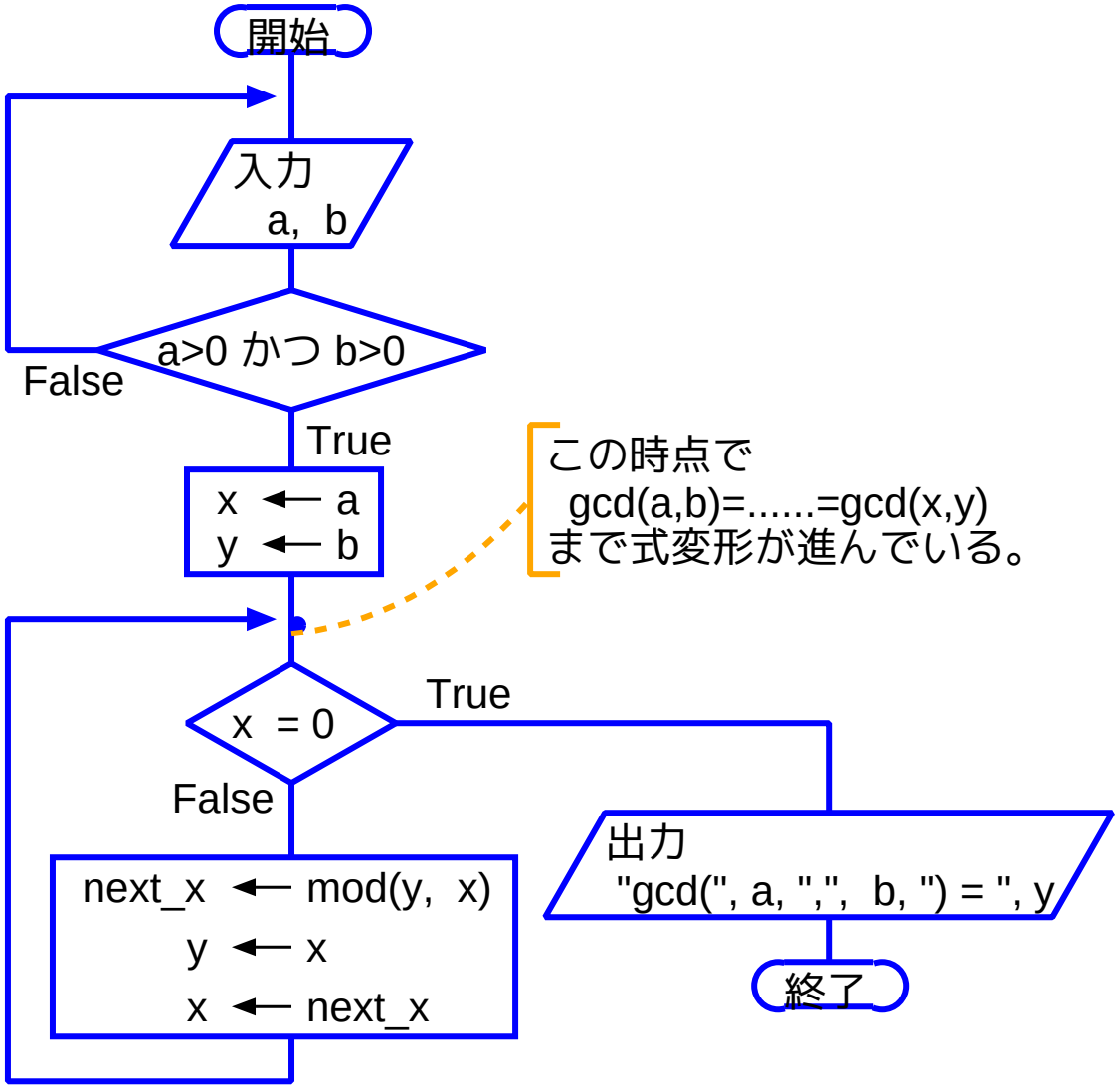

next_x ⟵ mod(y,x)
y ⟵ x
x ⟵ next_x
変数 x
変数 y
..... 状態gcd(28 , 56)
next_x ⟵ mod(y,x)
y ⟵ x
x ⟵ next_x
変数 x
変数 y
..... 状態gcd(0 , 28)
変数 x の値が 0 であることを確認
..... 計算結果: 28 (変数 y の値)
計算終了

要するに、x=0 となるまで

```
next_x ⟵ mod(y,x)
y ⟵ x
x ⟵ next_x
```

という固定的な処理を繰り返すだけである。

変数 x の値が 0 になった時点では、計算結果は変数 y に保持されている。それゆえ、読み込んだ整数データを格納するために a, b という名前の変数を、式変形の際の gcd(..., ...) の第1引数, 第2引数 を保持するために各々 x, y という名前の変数を、...



```
[motoki@x205a]$  nl gcd-euclid-algorithm.c   Enter
     1 /* 2つの正整数を読み込み、それらの最大公約数を出力 */
     2 /* するCプログラム  (Euclidのアルゴリズム)          */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   int  a, b, x, y, next_x;

     7   do {      do-while構文
     8     printf("最大公約数を計算します。"
                  "正整数を2つ入力して下さい: ");
     9     scanf("%d%d", &a, &b);
    10   }while (!(a>0 && b>0));
```

```
11    x = a;
12    y = b;
```

while構文

```
13    while (x != 0) {  /* この時点で gcd(a,b)=...=gcd(x,y)
14                      /* まで式変形が進んでいる。          *
15      next_x = y%x;
16      y      = x;
17      x      = next_x;
18    }

19    printf("gcd(%d, %d) = %d\n", a, b, y);
20    return 0;
21 }
```

計算状態を表す注釈

```
[motoki@x205a]$  gcc gcd-euclid-algorithm.c   Enter
[motoki@x205a]$  ./a.out   Enter
最大公約数を計算します。正整数を2つ入力して下さい:  1596 308   Ent
gcd(1596, 308) = 28
```

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

例題3．5（素数；while文,break文）　2以上の整数を読み込み、それが素数かどうかを判定して答えるCプログラムを作成せよ。

（考え方）　2以上の整数 k が与えられたとき、

k が素数 $\Longleftrightarrow$ k は 2~k−1 の整数で割り切れない（定義より）

$\Longleftrightarrow$ k は 2~$\sqrt{k}$ の整数で割り切れない

（i が k の約数なら k/i も k の約数でi と k/i のどちらかは $\sqrt{k}$ 以下となるから）

である。 従って、与えられた2以上の整数 k が素数であるかどうかを判定するためには、単に

2 が k を割り切るか,

3 が k を割り切るか,

．．．．．．．．．．．．．．．．．．．．．．．．

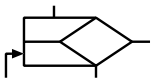
$\lfloor\sqrt{k}\rfloor$ が k を割り切るか,

ということを順に調べて、途中で「割り切る」という結果になったら即座に「素数でない」と判定を与え、途中で全然「割り切る」という結果にならなければ「素数だ」と判定を与えれば良い。

（プログラミング）　2以上の整数 k が素数かどうかの判定は、基本的には

i が k を割り切るかどうかを調べ ...

という処理を i=2，3，...，$\lfloor\sqrt{k}\rfloor$ に対して（すなわち i=2 から始め条件 $i^2 \leq k$ を満たす間、刻み幅 +1 で）順に行えば良いだけである。

流れ図においては繰り返しの箱  を用いるだけである。

ただ、この繰り返しは次の2点において通常の繰り返しと違っている。

(1) 繰り返しは途中で中止する可能性もある。

(2) 繰り返し後は判定結果を出すだけの状態になっているので、この繰り返し処理は2つの出口を持つ。

実際、

◇ 繰り返しの途中で「割り切る」という結果になったら、即座に繰り返しを終了して 「素数でない」と判定を下し、また、

◇ 繰り返しの途中で全然「割り切る」という結果に結果にならなければ、繰り返し後に「素数だ」と判定を下したい。

一般に、予め処理手順をC言語向きに構成しておかないと、実際にCプログラムを書く際に困ったことになる。 そこで、ここでは、上記(1)~(2)の特異点に対して次の様に対処する。

上記(1)に対する方策:

C言語では、現在実行中の場所から見て最も内側の繰り返し(またはswitch文)から脱出するために、break文と呼ばれるものが用意されている。プログラムを書く際は、それを使って繰り返しを途中で中止させる。

上記(2)に対する方策:

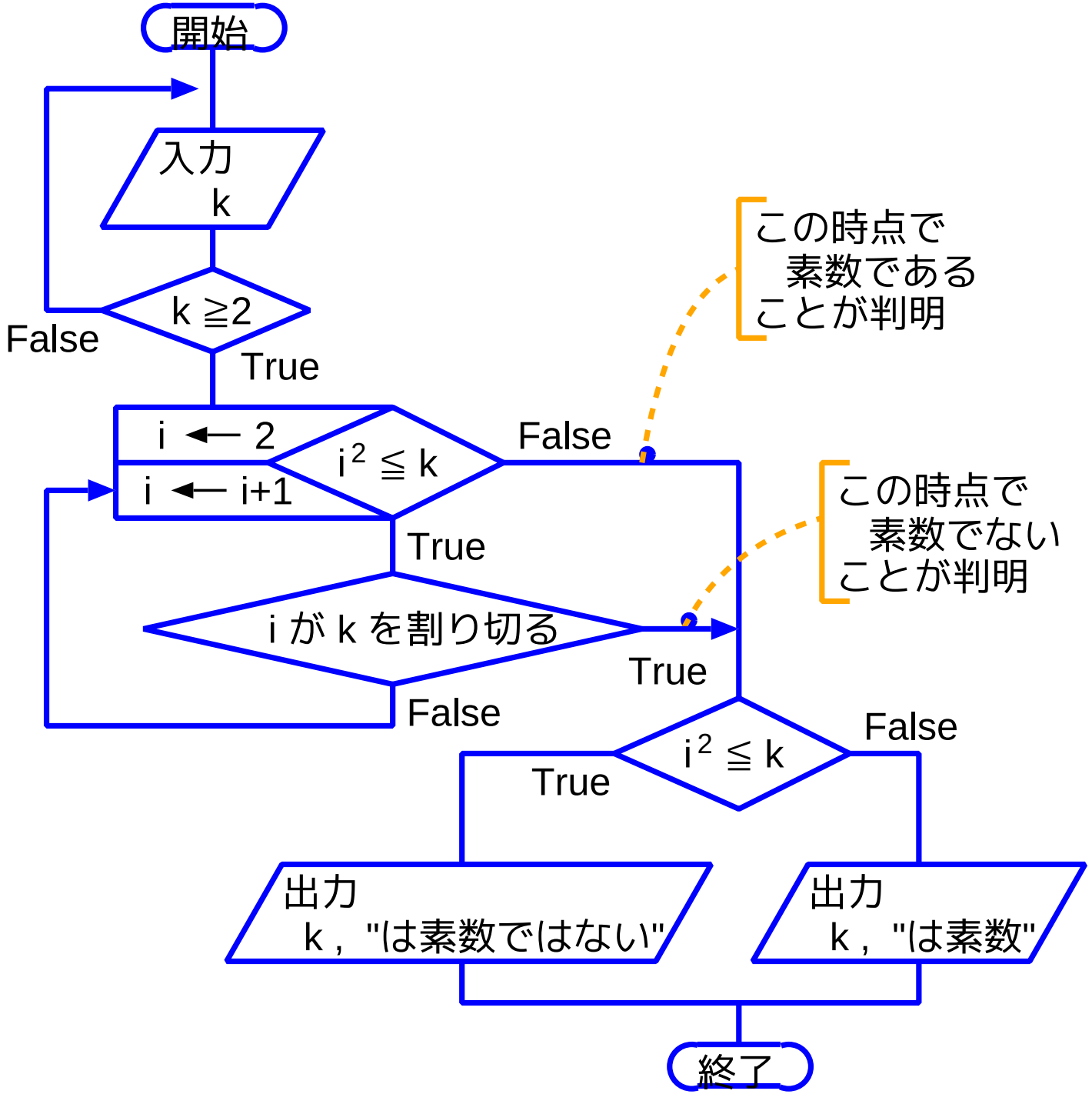
C言語の繰り返しの構文はどれも出口が1箇所であるので、繰り返しを途中で中止する場合と最後まで行った場合を区別せずに、繰り返し終了直後の処理を共通に用意しなければならない。

しかし、一旦合流したとしても、合流直後の繰り返しの変数 i の値を調べて、

もし $i^2 \leq k$ なら 繰り返しを途中で中止した,

もし $i^2 > k$ なら 繰り返しを最後まで行った

と判断することができるので、...


開始
入力
k
k ≧2
False
True
i ← 2
i ← i+1
i² ≦ k
False
True
この時点で
素数である
ことが判明
この時点で
素数でない
ことが判明
i が k を割り切る
True
False
i² ≦ k
True
False
出力
k , "は素数ではない"
出力
k , "は素数"
終了

```
[motoki@x205a]$  nl prime-number.c   Enter
     1 /* 2以上の整数を読み込み、それが素数かどうかを */
     2 /* 判定して答えるCプログラム                   */
     3 #include <stdio.h>

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

     7   do {
     8     printf("素数かどうかの判定をします。"
                  "2以上の整数を1つ入力して下さい: ");
     9     scanf("%d", &k);
    10   }while (!(k >= 2));
```

```
11    for (i=2; i*i<=k; i++) {
12      if (k%i == 0)  /* k%i==0 <==> i が k を割り切る */
13        break;       /* この時点でkが素数でないことが判明 *
14    }
```

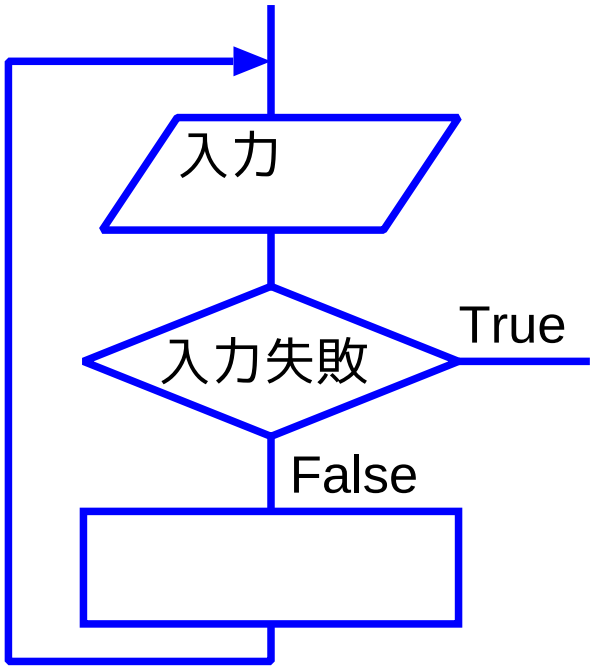
break文

```
15    if (i*i<=k)
16      printf("%dは素数ではない。\n", k);
17    else
18      printf("%dは素数です。\n", k);
19    return 0;
20  }
```

```
[motoki@x205a]$  gcc prime-number.c  Enter
[motoki@x205a]$  ./a.out  Enter
素数かどうかの判定をします。2以上の整数を1つ入力して下さい: 24
24は素数ではない。
[motoki@x205a]$  ./a.out  Enter
素数かどうかの判定をします。2以上の整数を1つ入力して下さい: 31
```

31は素数です。

3–4 入力データが無くなるまで繰り返し



例題3．6（不定個の入力データの合計）　データが無くなるまで次々と整数データを読み込み、それらの数値の合計を求めて出力するCプログラムを作成せよ。

（考え方）　例えば入力データが 2，5，10，33，77，. . .　の時、我々が手で合計を出すとしたら、次の様に入力順に計算を進める。

(step1) 1番目のデータまでの合計 = 2

(step2) 2番目のデータまでの合計 = 1番目のデータまでの合計 + 5 = 2 + 5 = 7

(step3) 3番目のデータまでの合計 = 2番目のデータまでの合計 + 10 = 7 + 10 = 17

(step4) 4番目のデータまでの合計 = 3番目のデータまでの合計 + 33 = 17 + 33 = 50

(step5) 5番目のデータまでの合計 = 4番目のデータまでの合計 + 77 = 50 + 77 = 127

. . . . . .

これに相当する計算を一般的にコンピュータに行わせれば良い。

では、

この場合、どんな変数を用意すれば良いのだろうか?

データの個数が予め分かってないので、読み込むデータ毎に別々の記憶領域を用意する という訳にもいかない。

⇨ 読み込んだデータを保持する変数を 1 個だけ用意し、

- そこへのデータ読み込みと、
- 読み込んだデータに対する処理

を交互に繰り返すことにする。

注目点：
過去に読み込んだデータは保存されないので、
次のデータを読む前に「読み込んだデータに対する処理」を十分に行わなければならない。

⇨ 具体的には、次のように処理を進める。

(step 1.a) 整数データ1個を 入力データを保持する変数 に読み込む。

(step 1.b) 1番目のデータまでの合計を保持する変数 ←− 入力データを保持する変数

(step 2.a) 整数データ1個を 入力データを保持する変数 に読み込む。

(step 2.b) 2番目のデータまでの合計を保持する変数

←− 1番目のデータまでの合計を保持する変数 ＋ 入力データを保持する変数

(step 3.a) 整数データ1個を 入力データを保持する変数 に読み込む。

(step 3.b) 3番目のデータまでの合計を保持する変数

←− 2番目のデータまでの合計を保持する変数 ＋ 入力データを保持する変数

. . . . . .

(step k.a) 整数データ1個を 入力データを保持する変数 に読み込む。

(step k.b) k番目のデータまでの合計を保持する変数

←− $(k-1)$番目のデータまでの合計を保持する変数 ＋ 入力データを保持する変

(final step) 読み込むデータが無くなったら、

k番目のデータまでの合計を保持する変数 の値を出力して終了。

これだと、それまでに読み込んだデータの合計を保持するために際限のない個数の変数が必要になる様に見えるが、. . . . . .

実際には、

> i番目のデータまでの合計を計算する時点では、
> 計算に必要な値は（i−1）番目までの合計と i番目のデータ値だけであり、それ以外の合計の結果はそれ以降も必要ない

から、例題6.6の場合と同様に考えて、

それまでに読み込んだデータの合計

を保持するために共通のデータ格納領域を1つだけ用意すれば良いことが分かる。

⇨ 共通のデータ格納領域として[それまでの合計を保持する変数]を用意して、次の手順でコンピュータに計算させれば良い。

(step 1.a) 整数データ1個を 入力データを保持する変数 に読み込む。

(step 1.b) それまでの合計を保持する変数 ⟵ 入力データを保持する変数

(step 2.a) 整数データ1個を 入力データを保持する変数 に読み込む。

(step 2.b) それまでの合計を保持する変数

⟵ それまでの合計を保持する変数 ＋ 入力データを保持する変数

(step 3.a) 整数データ1個を 入力データを保持する変数 に読み込む。

(step 3.b) それまでの合計を保持する変数

⟵ それまでの合計を保持する変数 ＋ 入力データを保持する変数

. . . . . .

(step k.a) 整数データ1個を 入力データを保持する変数 に読み込む。

(step k.b) それまでの合計を保持する変数

⟵ それまでの合計を保持する変数 ＋ 入力データを保持する変数

(final step) 読み込むデータが無くなったら、

それまでの合計を保持する変数 の値を出力して終了。

（プログラミング）

上記の手順(step 2.a)～(step k.b)は、

（a） 整数データ1個を 入力データを保持する変数 に読み込む。

（b） それまでの合計を保持する変数

$\longleftarrow$ それまでの合計を保持する変数 ＋ 入力データを保持する変数

という処理を 入力データが無くなるまで繰り返しているだけである。

手順(step 1.a)～(step 1.b)も、この共通の繰り返しパターンを使って

（step 0） それまでの合計を保持する変数 $\longleftarrow$ 0

（a） 整数データ1個を 入力データを保持する変数 に読み込む。

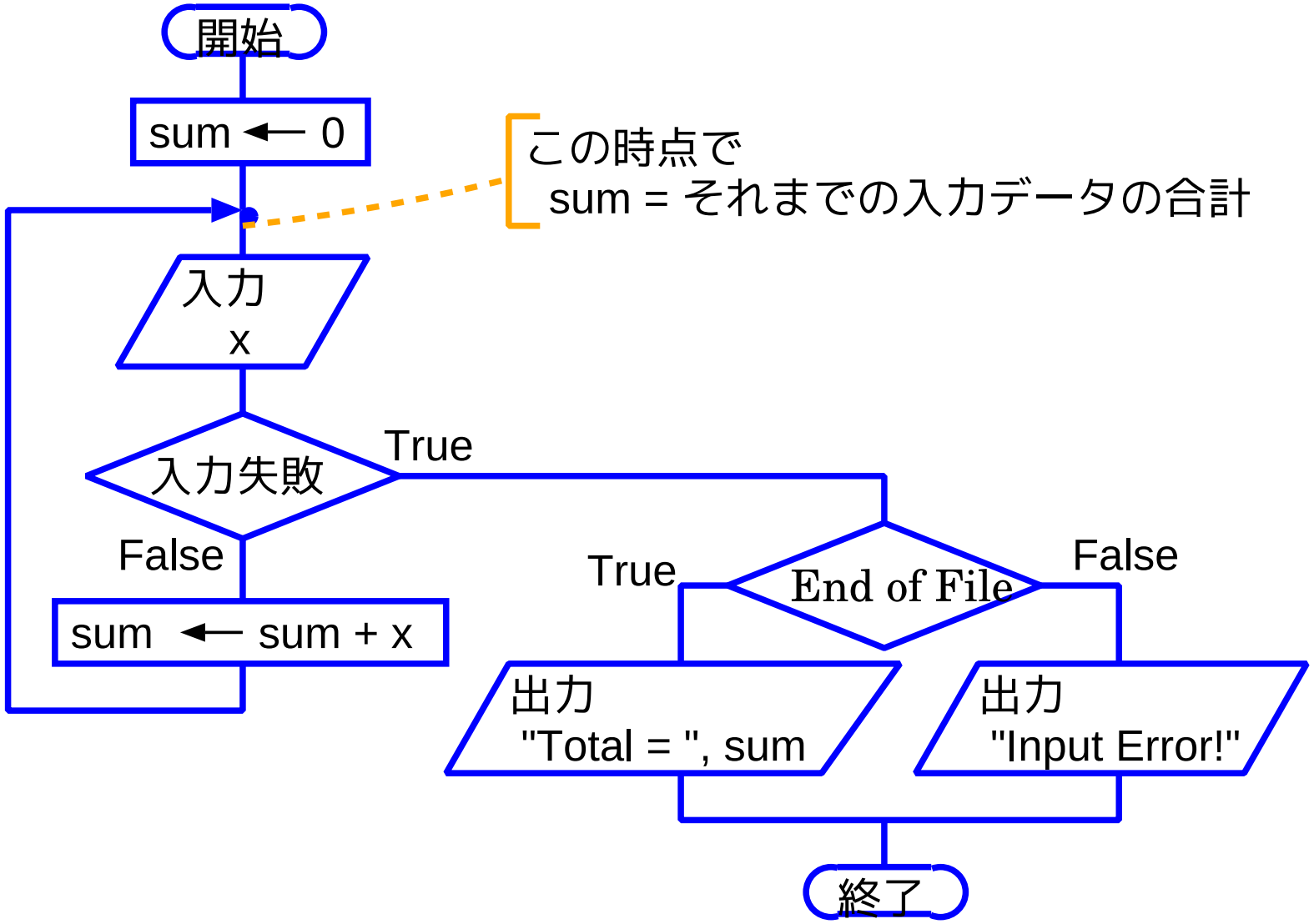
（b） それまでの合計を保持する変数

$\longleftarrow$ それまでの合計を保持する変数 ＋ 入力データを保持する変数

} 共通の繰り返しパターン

と書き換えることができる。

C言語では入力データ側に異常があった場合でも即座に実行時のエラーとはならないので、データ入力の失敗が起こった時その原因に応じた処置が必要である。　それゆえ、...

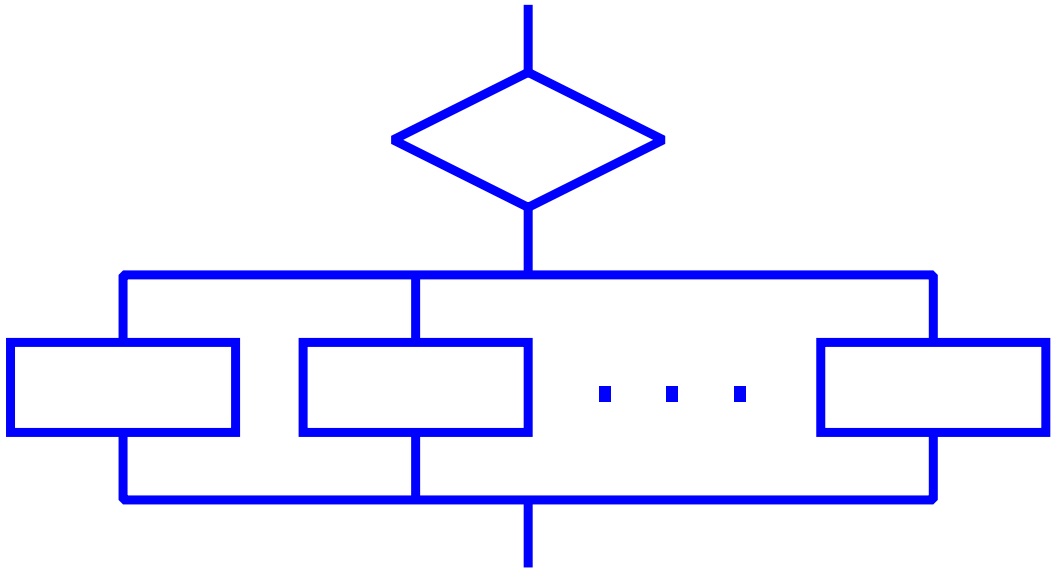


```
[motoki@x205a]$  nl sum-input-until-eof.c   Enter
```

```
 1 /*不定個の整数入力データの合計を求めて出力するCプログ...*/

 2 #include <stdio.h>

 3 int main(void)
 4 {
 5   int  x, sum, scanf_val;

 6   sum = 0;                    scanfの関数値
 7   while ((scanf_val=scanf("%d", &x))==1){
 8     sum += x;          /* sum = それまでの入力の合計 */
 9   }

10   if(scanf_val == EOF)
11     printf("Total = %d\n", sum);
12   else
```

```
   13      printf("Input Error!\n");
   14    return 0;
   15 }
[motoki@x205a]$  gcc sum-input-until-eof.c  Enter
[motoki@x205a]$  ./a.out  Enter
1 2 3 4  Enter
5 6 7 8 9 10  Enter
Ctrl-d
Total = 55
[motoki@x205a]$  ./a.out  Enter
1 2 w 4 5  Enter
Input Error!
[motoki@x205a]$
```

3–5 式の値に基づいた処理の選択



例題3．7（元号表記→西暦表記）　元号を表す文字(M，m，T，t，S，s，H，または h) と年数を読み込み、その元号表記の年を西暦表記に変換して出力するCプログラムを作成せよ。

（考え方）　元号を表す文字と年数を読み込んだ後、元号を表す文字が何であるかによって場合分けするだけである。

読み込んだ年数に誤りがないとすると、具体的には、

元号文字が M または m の場合：　明治元年が西暦1868年だから、
西暦の年数　読み込んだ年数 +1867　を出力すればよい。

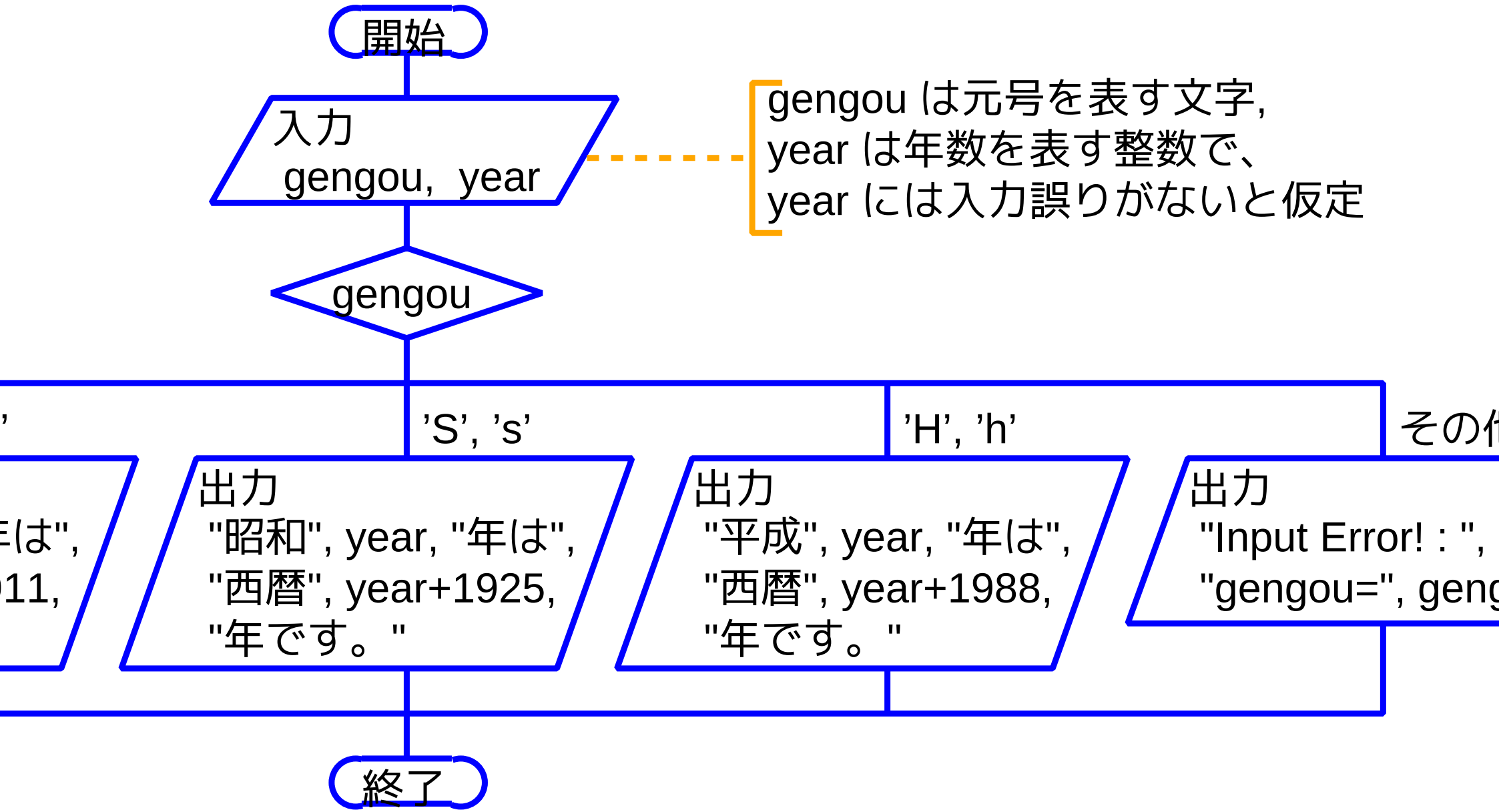
元号文字が T または t の場合：　大正元年が西暦1912年だから、
西暦の年数　読み込んだ年数 +1911　を出力すればよい。

元号文字が S または s の場合：　昭和元年が西暦1926年だから、
西暦の年数　読み込んだ年数 +1925　を出力すればよい。

元号文字が H または h の場合：　平成元年が西暦1989年だから、
西暦の年数　読み込んだ年数 +1988　を出力すればよい。

元号文字が M,m,T,t,S,s,H,h 以外の場合：
入力データの誤りを指摘すればよい。

（プログラミング）


開始
入力
gengou, year
gengou は元号を表す文字,
year は年数を表す整数で、
year には入力誤りがないと仮定
gengou
’S’, ’s’
’H’, ’h’
その他
出力
"昭和", year, "年は",
"西暦", year+1925,
"年です。"
出力
"平成", year, "年は",
"西暦", year+1988,
"年です。"
出力
"Input Error! : ",
"gengou=", geng
終了

[motoki@x205a]$ nl trans-gengou-year-to-Gregorian-year.c

```
 1 /*元号を表す文字(M,m,T,t,S,s,H,またはh)と年数を読み込み   */
 2 /*その元号表記の年を西暦表記に変換して出力するCプログラム*/

 3 #include <stdio.h>

 4 int main(void)
 5 {
 6   char  gengou;
 7   int   year;

 8   scanf("%c%d", &gengou, &year);
```

char型

switch文

```
 9   switch (gengou) {
10   case 'M': case 'm':
11     printf("明治%d年は西暦%d年です。\n",
12             year, year+1867);
13     break;
14   case 'T': case 't':
15     printf("大正%d年は西暦%d年です。\n",
16             year, year+1911);
17     break;
18   case 'S': case 's':
19     printf("昭和%d年は西暦%d年です。\n",
20             year, year+1925);
21     break;
22   case 'H': case 'h':
23     printf("平成%d年は西暦%d年です。\n",
24             year, year+1988);
25     break;
```

```
  26    default:
  27      printf("Input Error!: gengou='%c'\n", gengou);
  28    }
  29    return 0;
  30 }
[motoki@x205a]$  gcc trans-gengou-year-to-Gregorian-year.c
[motoki@x205a]$  ./a.out
H15
平成15年は西暦2003年です。
[motoki@x205a]$  ./a.out
G15
Input Error!: gengou='G'
[motoki@x205a]$
```

3–6 プログラムを組み立てられない時は ...

慣れて来ると与えられた問題を見ていきなりプログラムを書き下ろすということも出来る様になるであろうが、これは、過去の経験に基づき、

(1) 処理アルゴリズムを十分に理解した上で、
(2) 処理手順を計算機向きに構成し、更に
(3) 計算機向きに表された処理手順をC言語で表す、

ということを頭の中で全て行っているからに他ならない。

⇨ プログラムを組み立てられない時は 、
まず、プログラム作成のどの段階でつまづいているかを見定めた上で、つまづき段階に応じた適切な作業に入らなければならない。

⇨ 以下では 、
つまづき段階を特定するための簡単な質問と、各々のつまづき段階に応じた対処例を示す。 質問(1)から順番に試してみて下さい。

質問(1)： 必要なデータが全て与えられた時、コンピュータに代わって、紙の上で計算/処理を行えますか?

Yes ⟹① 実際に行ってみて下さい。すなわち、
例題1.3の「考え方」で示した様に、紙の上で計算/処理を行ってみる。入力データに応じて計算の方向が代わって来る場合は、例題3.3や例題3.6の「考え方」で行った様に、具体的な入力データを幾つか考え、それらに対して紙の上で計算/処理を行ってみる。

② 質問(2)へjump。

No ⟹ 自分の手で出来ない計算を、コンピュータに行わせられるはずもありません。

⟹ ⓪ 与えられた問題を人間の手で解くために、関連した文献を調べる。 そして、

① 「Yes」の場合の①~ ②を順に行う。

質問(2)： どういう変数を用意すれば良いか分かりますか?

Yes $\Longrightarrow$ ① 変数を全て列挙し、保持するデータにふさわしい名前を各々に付けてみて下さい。

② 質問(1)の所で行った紙の上での計算/処理の主要な時点において、それぞれ、①の変数の値がどうなっているかを明記した図/表を作り、計算/処理の時間順に並べてみて下さい。 各々の変数の値がどうなっているかを表す図/表は、計算/処理の状態を表す。

③ 前ステップ②で注目した状態(i.e. 各変数の値がどうなっているかを表す表)に関して、時間的に隣り合った状態を線で結び、それらの遷移を引き起こすために行った式計算/代入の列を線の脇に書き込むことによって、p.64やp.69に示したものと同様の状態遷移図(i.e. 変数値の変遷の様子を表した図)を構成して下さい。

④ 前ステップ③で作った状態遷移図を、p.62やp.67の様に処理を中心に書き直して下さい。

⑤ 前ステップ④で作った計算/代入の並んだ図の中に、条件判断を必要に応じて挿入することによって、一般的な(i.e. 個別の入力データによらない) アルゴリズムを適用した例として図を再構成して下さい。

⑥ 前ステップ⑤で計算/代入や条件判断結果の並んだ図が出来ているはずである。　この図を基に、一般的なアルゴリズムを流れ図として構成して下さい。

うまく流れ図が出来ない場合は 、前ステップ⑤の作業に問題がある可能性が高いので、ステップ⑤に戻って下さい。

⑦ 質問(3) へ jump。

No $\Longrightarrow$ ⓪ 質問(1) の所で行った紙の上での計算/処理の途中に現れるデータ(e.g. 入力値, 式計算の結果) は全て、使う時点には何らかの変数の中 に記憶されている。これを明示するために、紙の上の計算途中に現れるデータ全てを箱 □ で囲んで下さい。

① 質問(1) の所で行った紙の上での計算をプログラムとして

表した場合、前ステップ⓪で描いた箱 $\square$ は全てプログラム内の変数に相当する。また、紙の上の計算では全ての時点における計算結果が1枚の紙の上に現れているので、1つの変数に相当する箱が何箇所にも現れる。

$\Longrightarrow$

前ステップ⓪で描いた箱 $\square$ を変数領域と見て、それらの脇に各々の保持するデータにふさわしい名前を付けて下さい。但し、その際、

- プログラム内で同じ変数領域に出来そうな箱には、同じ名前を付ける。
- 箱に付ける名前の種類は出来るだけ少なくし、更には個別の入力値によらずに一定・有限にする。
- 箱に付ける名前は個別の入力値に依存させない。

$\Longrightarrow$

以下では、箱 $\square$ に付けた名前をプログラム内の変数名と考え、その名前の付いた箱で囲まれたデータを変数の値と見る。

② 「Yes」の場合の②を行う。

すなわち、
質問(1)の所で行った紙の上での計算/処理の主要な時点において、それぞれ、①の変数の値がどうなっているかを明記した図/表を作り、計算/処理の時間順に並べてみて下さい。　各々の変数の値がどうなっているかを表す図/表は、計算/処理の状態を表す。

③ 「Yes」の場合の③を行う。

すなわち、
前ステップ②で注目した状態(i.e.各変数の値がどうなっているかを表す表)に関して、時間的に隣り合った状態を線で結び、それらの遷移を引き起こすために行った式計算/代入の列を線の脇に書き込むことによって、p.64やp.69に示したものと同様の状態遷移図(i.e.変数値の変遷の様子を表した図)を構成して下さい。

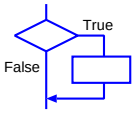
うまく式計算/代入の列が構成できない場合は、先のステップ①の作業に問題がある可能性が高いので、ステップ①に戻って下さい。

④ 「Yes」の場合の④~ ⑦を順に行う。

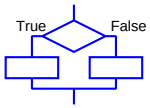
質問(3)： 前質問(2)の所で構成した流れ図を基にcプログラムを構成できますか?

Yes ⟹ cプログラムを構成して下さい。

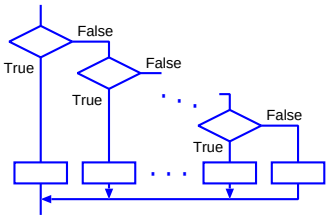
No ⟹ ① 流れ図が次の構造を組み合わせて構成されていることを確認する。



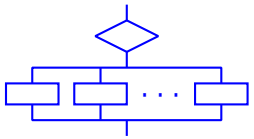
構造1



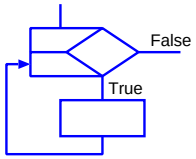
構造2



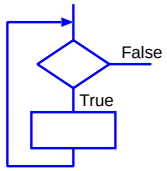
構造3



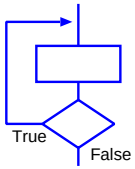
構造4



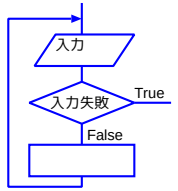
構造5



構造6



構造7



構造8

もしこれ以外の構造が含まれていたら、
流れ図をc言語向きに(上の構造の合成になるように)構成し直す。

補足：
break文があるので、構造5~ 8では繰り返しを途中で抜け出すことも可能である。 しかし、この場合、例3.5の様に、繰り返しを出る線は一旦1つに合流させる必要がある。

② 次の点に留意してCプログラムを構成する。

- 上の図の中の構造1は if文 で表す。
  (⟹ 例3.1のアルゴリズム(1))
- 上の図の中の構造2は if-else構文 で表す。
  (⟹ 例3.1のアルゴリズム(2))
- 上の図の中の構造3 は if-else-if-···-if-else構文 で表す。
  (⟹ 例3.1のアルゴリズム(3))
- 上の図の中の構造4は switch-case構文 で表す。(⇒ 例3.7)
- 上の図の中の構造5は for文 で表す。 (⟹ 例1.3)
- 上の図の中の構造6は while文 で表す。 (⟹ 例3.3)
- 上の図の中の構造7は do-while文 で表す。 (⟹ 例3.3)
- 上の図の中の構造8 は 条件部にscanfを含むwhile文 で表す。
  (⟹ 例3.6)

3–7 付録 制御構造のまとめ ---C文法のまとめ(2)

3–7–1 関係演算子, 同等演算子、論理演算子

真理値の表し方： C言語では真理値を次のように表す。

- 真 ··· 0以外（標準は1）
- 偽 ··· 0 （浮動小数点数の0.0でも、'\0'でも、ポインタ値NULLでも良い。）

演算子一覧：

種類	演算子	意味
関係演算子	<	より小さい
	>	より大きい
	<=	以下
	>=	以上
同等演算子	==	に等しい
	!=	に等しくない
論理演算子	!	論理否定（単項）
	&&	論理積
	||	論理和

関係演算子：

- 演算結果は int 型の 0 または 1 。

	e1<e2の値	e1>e2の値	e1<=e2の値	e1>=e2の値
e1＞e2 の場合	0	1	0	1
e1＝e2 の場合	0	0	1	1
e1＜e2 の場合	1	0	1	0

- 優先順位は算術演算子よりも低い。
  $\Longrightarrow$ 例えば、式 a-b<0 は (a-b)<0 と同等。

- 注意 式 -1<0<1 は文法的に誤りではなく、0(偽) という値になる。

何故なら、
関係演算子は左から右に結合するので、
これは

$$\underbrace{(-1<0)}_{1}<1 \Longrightarrow 1<1 \Longrightarrow 0(\text{偽})$$

と計算されていくから。

同等演算子：

- 演算結果はint型の 0 または 1 。

	e1==e2の値	e1!=e2の値
e1＝e2 の場合	1	0
e1≠e2 の場合	0	1

- 優先順位は算術演算子や関係演算子よりも低い。
  $\Longrightarrow$ 例えば、式 a<b==a+1<=b は （a<b） == （(a+1)<=b）と同等。
  （見にくい部分は省略可能であってもカッコを付けた方が良い。）

- 注意 if文を if （a=1） ... という風に書くと、変数 a の値が何であっても条件部は真と判定され(a=1)に続く(複合)文が実行される。

何故なら、
条件部の「a=1」は代入式であり、その値は代入結果の値である 1 となるから。

論理否定演算子：

- 演算結果は`int`型の `0` または `1` 。

	`!e`の値
`e`=`0` の場合	`1`
`e`≠`0` の場合	`0`

- 否定演算 `!` の優先順位は他の単項演算子(e.g. 符号反転の`-`,`++`)と同じ。

- 注意　条件式　`!(!e)==e`　は一般には不成立。

論理積と論理和：

- 演算結果は`int`型の `0` または `1` 。

	`e1&&e2`の値	`e1||e2`の値
`e1`=`0`, `e2`=`0` の場合	`0`	`0`
`e1`=`0`, `e2`≠`0` の場合	`0`	`1`
`e1`≠`0`, `e2`=`0` の場合	`0`	`1`
`e1`≠`0`, `e2`≠`0` の場合	`1`	`1`

演算子の優先順位：

優先順位高

演算子	結合性
関数の引数をくくる丸括弧	左から右
+(単項) -(単項) ++ -- sizeof( ) ! キャスト	右から左
* / %	左から右
+ -	左から右
< <= > >=	左から右
== !=	左から右
&&	左から右
||	左から右
= += -= *= /=	右から左

短絡評価：

- e1&&e2の評価の際、e1の値が 0 となればe2の値の評価は省略され、式全体の値は即座に 0 と結論づけられる。
- e1||e2の評価の際、e1の値が 1 となればe2の値の評価は省略され、式全体の値は即座に 1 と結論づけられる。

例3．8（短絡評価であることの利用）短絡評価であることを利用すれば、次のような書き方も出来る。

- ```
  do {
    printf("\n正整数を２つ入力して下さい：  ");
  } while ((num_input=scanf("%d %d", &x, &y))==2 && (x<=0 ||
  if (num_input != 2) {
    printf("エラーメッセージ");
    exit(EXIT_FAILURE);
  }
  ```
- ```
  if (x!=0 && y/x>10) {
    ............
  ```

}

3–7–2 複合文と空文

複合文：

{

宣言

⋮

宣言

文

⋮

文

}

ブロック：

複合文のうち、宣言が１個以上含まれるもの。

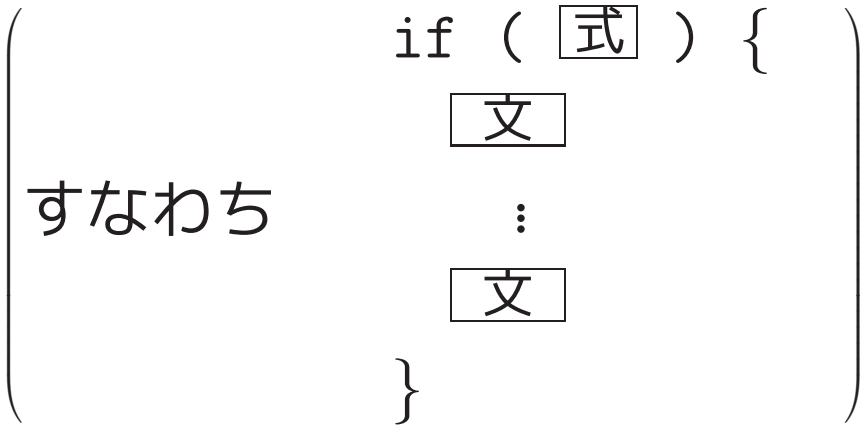
空文：

セミコロンだけの文。

3–7–3 条件分岐の制御構造

if 文：

- if (式)
  文
- if (式)
  複合文



if-else 構文：

- 構文は

```
if ( 式 )
  複合文
else
  複合文
```

- 注意　else は最も近い if と結びつく。

$\Longrightarrow$ 例えば、

```
if (a == 1)
  if (b == 2)
    printf("***\n");
else
  printf("###\n");
```

は次のものと同等。(間違った字下げはしない様に気を付ける。)

```
if (a == 1) {
  if (b == 2)
    printf("***\n");
  else
    printf("###\n");
}
```

---

switch文：

- if-else文を一般化した多分岐条件文。
- 構文は

```
switch ( 整数型の式 ) {
case 整数型の定数式 :
  ⋮
case 整数型の定数式 :
  文の列
  break;
case 整数型の定数式 :
  ⋮
case 整数型の定数式 :
  文の列
  break;
case 整数型の定数式 :
  ⋮
  ⋮
```

```
case  整数型の定数式 :
  ⋮
case  整数型の定数式 :
    文の列
  break;
default:
    文の列
  break;
}
```

- break文がないと、実行は次のcaseラベルを通り抜けてその後に続く文に移る。
- 例えば次のように使う。

```
switch (c) {
case 'a': case 'A':
  ++a_cnt;
  break;
case 'b': case 'B':
```

```
        ++b_cnt;
        break;
    case 'c': case 'C':
        ++c_cnt;
        break;
    default:
        other_cnt;
        break;
    }
```

条件演算子：

- 構文は

  式1 ? 式2 : 式3

- その意味は次の通り。

  if 式1 then 式2 else 式3

- if - else構文と違って、これを代入式の右側に持って来ることが出来る。

3–7–4 繰り返しの制御

while文：

```
while ( 式 )
  文
```

for文：

- 構文は

```
for ( 式1  ; 式2  ; 式3  )
    文
```

ここで、式1 ～ 式3 の中には、コンマ演算子を使って複数の式を並べることも可能。式2 が省略された場合、繰返しの本体は無条件に実行される。

- 利点　繰り返し制御の変数の操作を先頭にまとめることが出来る。

---

do文：

```
do {
  文
   ⋮
  文
} while ( 式 )
```

3–7–5 その他

コンマ演算子：

- 構文は

  式1 , 式2

- 注意 関数の実引数の場所で使いたい場合は、実引数全体を丸括弧で囲む。

break文：

- 構文は

```
break;
```

- それを含む、最も内側の ループ（i.e.for,while,またはdo-whileによる繰り返し）またはswitch文から抜け出す。
- 例えば次のように使う。

```
while (1) {
  scanf("%lf", &x);
  if (x < 0.0)
```

```
        break;
      printf("%f\n", sqrt(x));
    }
```

continue文：

- 構文は

```
continue;
```

- それを含む最も内側のループ(for,while,またはdo-while) の、現在の繰り返し処理を終了し、次の繰り返し処理に移る。
- 例えば次のように使う。

```
for (i=0; i<TOTAL; ++i) {
  c =getchar();
  if ('0'<=c && c<='9')
    continue;
   ..............
}
```

ここで、getchar は標準入力のストリームから1文字だけ(空白も可)読み込んで、その文字コードの値を返す関数である。[但し、ファイルの終りまたはエラーを検出した時は EOF（マクロ; 通常 -1 が割り当てられている）を返す。関数値の型は char ではなく int である。]

goto文：

一般に goto文は避けるべき。

$\Longrightarrow$ 説明省略