アルゴリズム入門　＃3

地引　昌弘

2024.10.17

はじめに

今回は、次の二つを目標とします:

- 基本的な制御構造 (プログラムにおける処理の流れ) として、枝分かれについて理解し、条件に応じて振舞いを変えるアルゴリズム · プログラムを考えられるようになる。
- 同、繰り返しについて理解し、条件に応じて同じ振舞いを続けるアルゴリズム · プログラムを考えられるようになる。

1　前回の演習問題の解説

1.1　演習 2-1 — 整数と実数の違い

演習 2-1 は、「切捨て除算以外の」整数と実数の違いを試すというものでした。ヒントに載せたものをやってみます。足し算を直接書いてもよいのですが、ここでは以前の演習で定義した `add` 関数を呼んでいます。

```
>>> add(123451234512345, 1)
123451234512346
>>> add(123451234512345.0, 1.0)
123451234512346.0
>>>
```

Python では、どちらも基本的に同じ値を意味していますね。では、12345 をもう一つ増やしてみましょう。

```
>>> add(12345123451234512345, 1)
12345123451234512346
>>> add(12345123451234512345.0, 1.0)
1.2345123451234513e+19
>>>
```

前回の資料にも載せましたが、Python では、整数値の演算結果がある標準のビット数以内で表わせなくなった場合、適宜ビット数を増やして表わせる範囲を自動的に広げる仕様になっています。よって、整数は問題ないのですが、実数はおまかせだと適当なところで丸められた指数記法になってしまうため、正確な結果が分かりません。そこで `print` 関数を利用し、桁数を増やして表示させてみます。

```
>>> print("%.20g" % add(12345123451234512345.0, 1.0))
12345123451234512896
```

こうして見ると下 3 桁は全く違うことが分かりますね。その理由ですが、実数の場合は仮数部 (有効数字の部分) が十進法で 16 桁程度に固定されているため、それより多い桁数の部分は無視され、無意味な数字が入っているからです (前回の講義を思い出しましたか)。

以上より、コンピュータで計算する時には、整数による計算、実数による計算をきちんと計画的に使い分ける必要があると分かります。しかし、一般には、整数と実数の混ざった計算をする機会も多くあります。このような場合は、整数

を実数に、また実数を整数に変換する必要がありますね。以前にも述べましたが、Python では int 関数 (実数を整数に変換) と float 関数 (整数を実数に変換) が用意されています。前者は数値 x が整数ならそのままですが、実数なら (小数点以下を切り捨てて) 整数に変換します。後者は x が実数ならそのままですが、整数なら実数に変換します。

```
>>> int(3.14)
3
>>> float(314)
314.0
```

整数と実数は、表示されたものを見る限りでは小数点が付いているかどうかの違いしかないですが、前回の講義で述べたようにコンピュータ内部でのデータ表現は全く違うため、計算している値を整数や実数に切り替える必要がある場面では、その都度変換して使う必要があります。

1.2 演習 2-2 — 実数計算の誤差

演習 2-1 は整数と実数の違いに関するものでしたが、こちらは実数計算の誤差を観察する問題でした。当然、print 関数を使って十分な桁数を表示するようにします。このような場合は、関数を毎回書くのではなく、IDLE の対話モードを用いて直接計算してみましょう (値の括弧を忘れないように):

```
>>> print("%.20g" % (1.0/3.0))
0.33333333333333331483                  ← 有効桁数は十進で 16 桁程度
>>> print("%.20g" % ((1.0/3.0) * 3.0))
1                                       ← 3 倍すると最後の桁が丸められて元に戻る
>>>
>>> print("%.20g" % (7.0 / 10.0))
0.69999999999999995559                  ← 0.7 は二進では循環小数になる
>>> print("%.20g" % (7.0 * 0.1))
0.70000000000000006661                  ← 0.1 も同様 (誤差を含んでいる)
```

このように、実数は桁数が有限であることから、計算結果には微妙な誤差が現れます。しかし一方で、二進表現を使っているということは、2^N や $\frac{1}{2^N}$ の組み合わせで表わせる数は有限な実数 (小数) となり、コンピュータで扱える数値の範囲を超えない (これを、“溢れない” と言ったりします) ため誤差が出ません。

```
>>> print("%.40g" % (7.0 / 16.0))        ← 7.0 は IEEE 754 表現で有限な桁数となることに注意
0.4375
>>> print("%.40g" % (7.0 * 0.0625))
0.4375
>>>
>>> print("%.40g" % (7.0 / (2**16)))
0.0001068115234375
>>> print("%.40g" % (7.0 * (0.5**16)))
0.0001068115234375
>>>
>>> print("%.40g" % (7.0 / (2**32)))
1.629814505577087402343750e-09
>>> print("%.40g" % (7.0 * (0.5**32)))
1.629814505577087402343750e-09
>>>
```

ここで取り上げた誤差は、丸め誤差 (二進法で正確に表わせないことによる誤差) に分類されます。

1.3 演習 2-3 — 誤差の種類

演習 2-3 は、誤差の生じる原因を考える問題でした。まずは、演習 2-3a を実際に計算してみましょう。

```
>>> print("%.20g" % (7.0 / 10.0))
0.69999999999999995559
>>> print("%.20g" % (7.0 * 0.1))
0.70000000000000006661
>>>
```

これは、演習 2-2 の解説でも説明しましたが、二進法では 0.1 を正確に表現できないことに起因する丸め誤差となります。

次に、演習 2-3b を計算してみましょう。

```
>>> print("%.20g" % ((100000000.0 + 1.0)**2))
10000000200000002
>>> print("%.20g" % (100000000.0**2 + 2*100000000.0 + 1.0))
10000000200000000
>>>
```

左辺では $10^8 + 1$ を計算してから 2 乗するのに対し、右辺では 2 乗した後の 10^{16} に 1 を加えています。10^{16} と 1 では、絶対値が差が大きいため、両者の加減算では小さい値の下の桁が無視されてしまいます (1 には、より下の桁がないので、この例では単に無視されます)。このように、絶対値の差が大きい数値同士の計算に起因する誤差は、情報落ち誤差となります。

最後に、演習 2-3c を計算してみましょう。

```
>>> print("%.20g" % (1234567890.12345-1234567890.0))
0.12345004081726074219
>>>
```

値の近い数値同士を引き算すると、浮動小数点表現の仮数部が 1 より小さくなります。N 進法の浮動小数点表現では、仮数部は 1 以上 N 未満になるよう補正されるため、仮数部が 1 より小さい場合は、不足の桁に応じて単純に N^n 倍することで補正します。このように、値の近い数値同士を引き算に起因する誤差は、桁落ち誤差となります。

1.4 演習 2-4 — 誤差の理由

演習 2-4 は、演習 2-3 と同様、誤差の生じる原因を考える問題ですが、単純な計算ではなく、関数を介したより複雑な状況を取り上げています。問題文にもある通り、複素数 $10^8 + 1$ については、abs(*plus(10.0**8, 0.0, 1.0, 0.0)) と abs(10.0**8, 0.0) の計算結果が異なるので、何か大きな (or 致命的な) 誤差が生じているわけではありません。問題は $10^8 + i$ の方です。abs(10.0**8, 0.0) の計算については、abs 関数の定義を見てみると、b = 0 なので単純に a の値を 2 乗して平方根を求めているだけで、丸め誤差を除けば、計算に起因する誤差は生じていないように見えます。

次に、abs(*plus(10.0**8, 0.0, 0.0, 1.0)) の計算ですが、まずは plus 関数を個別に動かしてみると、plus(10.0**8, 0.0, 0.0, 1.0) は (100000000.0, 1.0) を返していることが分かります。そこで、abs(10.0**8, 1.0) について考えてみましょう。今度は $b \neq 0$ なので、先ほどの場合と事情が異なります。つまり、a の 2 乗と b の 2 乗の足し算が正確に行なわれないと、誤差の生じる恐れがあるわけです。a は 10.0 の 8 乗なので、その 2 乗は 10.0 の 16 乗になります。これに対して b は 1.0 なので、2 乗しても 1.0 のままです。演習 2-3 でも説明しましたが、絶対値の差が大きい数値同士の計算に起因する誤差は、情報落ち誤差ですね。

2 基本的な制御構造

これまでに出て来たアルゴリズムおよびプログラムは、全て「一本道」、つまり上から順番に実行して一番下まで来たら終了、というものでした。単純な計算ならそれでも問題ありませんが、手順が複雑になって来ると、実行の流れを様々

に切り換えて行くことが必要になります。このような実行の流れを切り換える仕組みのことを、一般に**制御構造** (Control Structure) と呼びます。

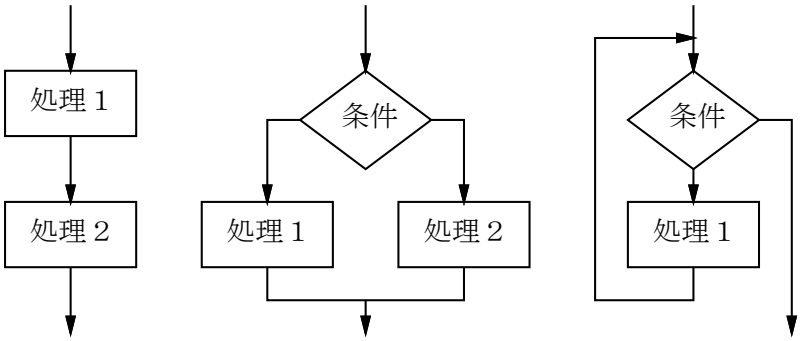

図 1: 三つの基本的な制御構造

制御構造を表現する方法の一つに**流れ図** (Flowchart) があります。流れ図では、図 1 にあるような「処理を示す箱」や「条件による枝分かれを示す箱」などを矢線で繋げることで、多様な実行の流れを表現します。流れ図は一見分かり易そうですが、作成に手間が掛かる、場所を取る、不規則でゴチャゴチャな構造を作ってしまい易い、という弱点があるため、今日のソフトウェア開発ではあまり使われません。このため本資料でも、流れ図の代わりに擬似コードを主に用いています。

アルゴリズムを記述する際は、その処理に沿って様々な実行の流れを組み立てますが、これは通常、図 1 に示す三つの制御構造を組み合せる形で作ります:

- **順次実行** or **連接** (Sequenceing) — 動作を順番に実行して行くこと (図 1 左)。
- **枝分かれ** or **分岐** (Branching) — 条件に応じて二群の動作のうちから一方を選んで実行すること (図 1 中)。
- **繰り返し** or **反復** (Repetition) — 条件が成り立つ限り一群の動作を繰り返して実行すること (図 1 右)[1]。

何故この三つが基本になるかというと、どんなにゴチャゴチャな流れ図でも、その流れ図と同等な動作をする別の流れ図を、この三つの組み合わせによって作り出すことができる、言い換えれば、この三つさえあればどのような処理の流れでも表現できると言えるからです。これを**構造化定理** (Structure Theorem) と呼びます (構造化定理を証明するための考え方を付録に載せました/興味のある方は是非読んでみて下さい)。連接については、単に動作を並べて書いたものは並べた順番に実行されるというだけなので、以下では残り二つの制御構造をコード上で表現する方法と、それらを組み合わせてアルゴリズムを組み立てて行く方法を学びます。

3 枝分かれと if 文

前述のように、枝分かれとは、条件に応じて二群の動作のうちから一方を選んで実行するものです。擬似コードでは、枝分かれを次のように書き表わすものとします (「動作 2, 3」が不要なら、「そうでなければ」を書かなくてもよいです)。また、以後の疑似コードでは、動作 (ブロック) の終わりをはっきりさせるため、“(枝分かれ終わり。)” と書きますが、Python では、枝分かれは複合文に分類されるため、インデントの有無で境界を示す点に注意して下さい:

- もし ～ ならば、
- 　　動作 1
- そうでなくて ～ ならば、
- 　　動作 2
- そうでなければ、
- 　　動作 3
- (枝分かれ終わり)

[1]実行の流れを図示すると環状になるので、**ループ** (Loop) とも呼びます。

Python では、これを **if 文** (If Statement) と呼ばれる構文により表わします (右側は「条件 2, 3」のない場合です):

```
if 条件1:                  if 条件1:
  ... 動作1 ...              ... 動作1 ...
elif 条件2:
  ... 動作2 ...
else:
  ... 動作3 ...
```

「条件」については、例えば以下のようなものがあります。具体的な使い方については、“Python によるプログラミングの初歩” にある “演算子” を参照して下さい:

- 比較演算 — 「`x > 10`」のように二つの値を比べるもの。**比較演算子** (Comparison Operator) としては、`>` (より大)、`>=` (以上)、`<` (より小)、`<=` (以下)、`==` (等しい)、`!=` (等しくない) がある。
- 条件の組み合わせ — **かつ** (and — 共に成り立つ) を表わす「条件1 `and` 条件2」、**または** (or — 少なくとも片方は成り立つ) を表わす「条件1 `or` 条件2」、**否定** (not — 〜 でない) を表わす「`not` 条件1」が使える。複数の “かつ”、“または”、“否定” を組み合わせたり、括弧で括ることもできる。

次に、このような条件およびその組み合わせを条件式と見なし、その値を求めることを考えます。一般に、条件式の最終的な結果は正しいか間違っているかの 2 通りなので、これらを以下の 2 値で表わします。

True: (最終的な) 条件式の結果は正しい (“真” と表わすこともある)。

False: (最終的な) 条件式の結果は間違っている (“偽” と表わすこともある)。

そして、条件式から True · False を求めることを**論理演算**と呼びます。また、条件として、条件式を書く代わりに、True · False を代入した変数を直接指定することもできます。これは、比較演算子だけでは表現しにくい条件を扱いたい場合などに使われます (どのような場合に使うのか、具体的な事例については、次回以降取り上げて行きます)。

では具体的な例題として、「入力 x の絶対値を計算する」ことを考えてみます。まず擬似コードを示しましょう:

- abs1: 数値 x の絶対値を返す
- 　もし $x < 0$ ならば、
- 　　$result \leftarrow -x$
- 　そうでなければ、
- 　　$result \leftarrow x$
- 　(枝分かれ終わり)
- 　$result$ を返す。

考え方としては簡単ですね。これを Python で書いてみましょう:

```
def abs1(x):
  if x < 0:
    result = -x
  else:
    result = x
  return(result)
```

実行の様子も示しておきます (0 もテストしていることに注意。作成したコードをテストする際は、“系統的かつ洩れなく” 試すことが大変重要です):

```
>>> abs1(8)        ← 正の数の絶対値は
8                  ← 元のまま
>>> abs1(-3)       ← 負の数であれば
3                  ← 正の数になる
>>> abs1(0)        ← 0 の場合は
0                  ←元のまま
>>>
```

ところで、同じ絶対値のプログラムを次のように書いたらどうでしょうか?:

- abs2: 数値 x の絶対値を返す
- 　もし $x < 0$ ならば、
- 　　$-x$ を返す。
- 　そうでなければ、
- 　　x を返す。
- 　(枝分かれ終わり)

Python では、次のようになります:

```
def abs2(x):
  if x < 0:
    return(-x)
  else:
    return(x)
```

最初の例と、どちらが皆さんの好みでしょうか? また、別のバージョンとして次のものはどうでしょうか?

- abs3: 数値 x の絶対値を返す
- 　$result \leftarrow x$
- 　もし $x < 0$ ならば、
- 　　$result \leftarrow -x$
- 　(枝分かれ終わり)
- 　$result$ を返す。

「そうでなければ」の部分で、何もすることがなければ「そうでなければ」以下を書かなくてもよいのでしたね。Python のプログラムでも示しておきます:

```
def abs3(x):
  result = x
  if x < 0:
    result = -x
  return(result)
```

これら三つのプログラムについて、皆さんはどれが好みだったでしょうか?

一般に、プログラムの書き方は「どれが絶対正解」ということはなく、場面毎に何が良いかは変わり、また人によっても基準は違います。つまり、プログラミングの学習には、自分なりの「良いと思う書き方」を発見して行くという側面があることを、心に留めておいて下さい。

演習 3-1 枝分かれを用いて、次の動作をする Python プログラムを作成せよ。

a. 二つの異なる実数 a、b を受け取り、より大きい方を返す。
b. 三つの異なる実数 a、b、c を受け取り、最大のものを返す (余裕があれば、実数が四つの場合も試してみましょう)。
c. 実数を一つ受け取り、それが正なら「positive」、負なら「negative」、零なら「zero」という文字列を返す。

演習 3-2 2 次方程式 $ax^2 + bx + c = 0$ の解 $x = (-b \pm \sqrt{D})/2a,\ D = b^2 - 4ac$ を求める関数を、以下のように作成した。

```
import math
def fa(a, b, c):
  D = b**2 - 4.0*a*c
  x1 = (-b + math.sqrt(D)) / (2.0*a)
  x2 = (-b - math.sqrt(D)) / (2.0*a)
  return(x1, x2)
```

また、得られた解がどれだけ正しいかを検算する関数を以下のように作成した。

```
def check(a, b, c, x1, x2):
  return(a*x1**2 + b*x1 + c, a*x2**2 + b*x2 + c)
```

fa 関数より 2 次方程式 $x^2 - 100x + 1 = 0$ の解を求め、check 関数により検査した結果は、下記の通りであった。

```
>>> x1, x2 = fa(1, -100, 1)
>>> x1
99.98999899979995
>>> x2
0.010001000200048793
>>> check(1, -100, 1, x1, x2)
(0.0, 1.2212453270876722e-13)
>>>
```

この結果を見る限り、解に誤差が生じていると考えられる。誤差の原因を考察し、fa 関数からこの誤差を取り除いた fb 関数を作成しなさい。

> ヒント: これまで見て来たように、丸め誤差はある意味、実数そのものが持ってしまう誤差とも考えられるので、これを減らすことは難しいです。これに対し、桁落ち誤差や情報落ち誤差は計算に伴い発生する誤差なので、計算の順番を変えるなどの工夫により、減らせる可能性があります。情報落ち誤差は、絶対値が大きい数値と小さい数値の計算により発生しますが、計算の前にこれらを因数分解できれば、絶対値の大小をある程度は緩和できます。2 次方程式の解の公式では、判別式内に掛け算が出て来るので、ここを因数分解できれば良いのですが、皆さんも御存じの通り、これ以上は因数分解できません。桁落ち誤差は、値が近い数値同士の引き算により発生します。解の公式では、同じく判別式内に引き算がありますが、これを他の計算へ変えることは、やはり出来ません。しかし、解の公式には、これ以外に $-b$ と $\sqrt{D}$ の足し算 · 引き算があります。よって、「足し算」により得られる方の解は、桁落ち誤差を緩和出来ていると言えます。但し、これだけでは片方の解しか求まらないではないかと思えるのですが、実は 2 次方程式の解は、2 解 α, β のうち一つが求まれば、解と係数の関係により、もう一方の解を「引き算」を使わずに求めることができます。さて、fa 関数内で得られた 2 解のうち、どちらが「引き算」となる解でしょうか。

4 繰り返しと while 文

これまでは、プログラム上に書かれた命令は 1 回しか実行されなかったため、プログラムが行なう計算の量はプログラムの長さ程度しかありませんでした。しかし、繰り返しがあれば、指定された範囲内の命令が何回でも反復して実行されるため、短いプログラムでも大量の計算が行われます。

まず、繰り返しの最も一般的な形である、条件を指定した繰り返しの擬似コードを示します。「〜」の部分には条件を記述しますが、ここに書けるものは if 文の条件と全く同じです。また、以後は繰り返しの疑似コードでも、動作 (ブロック) の終わりをはっきりさせるため、“(繰り返し終わり。)” と書きますが、枝分かれと同様 Python では、繰り返しは複合文に分類されるため、インデントの有無で境界を示すことに注意して下さい:

- 〜 である間、繰り返し、
- 　動作 1
- (繰り返し終わり)

この形の繰り返しは、Python では **while 文** (While Statement) として記述します:

```
while 条件 1:
  ... 動作 1 ...
```

多くのプログラミング言語では、このような繰り返しを、while というキーワードを用いて記述するので、条件を指定した繰り返しのことを **while ループ** (While Loop) とも呼びます。while ループは、形だけなら if 文より簡単ですが、慣れるまではどのように実行されるかのイメージが湧かない人も多いと思います。while ループの実行のされ方は、次のようなものだと考えて下さい:

- 「〜」を調べる (成立)。
- 動作 1 を実行。
- 「〜」を調べる (成立)。
- 動作 1 を実行。
- 「〜」を調べる (成立)。
- 動作 1 を実行。
- …
- 「〜」を調べる (不成立)。
- 繰り返しを終わる。

つまり、条件を調べ、成り立てば動作 1 を実行し、また条件を調べ、…のように繰り返して行き、条件が成り立たなくなると繰り返しを終わります。

また、条件として、条件式を書く代わりに、Ture·False を代入した変数を指定することもできます (この変数を、特定の条件·事象が成立したことを示す**フラグ** (Flag) と呼びます)。例えば、こんな感じです:

```
found = False
while not found:
  ...
  ...    # ループ内の処理に応じて、found に Ture を設定
  ...
```

このような書き方は、(前でも触れましたが) 比較演算子だけでは表現しにくい条件を扱いたい場合などに使われます。例えば、以下のような問題を扱う場合を考えてみましょう。

> 「ある数列が小さい順に並んでいるかどうかを調べ、
> 逆順になっている箇所があれば、最初に発見した逆順の例を表示して終了せよ。」

これは、逆順があるかどうかを while ループにより繰り返し調べて行くことになりますが、この問題では逆順を発見して直ちにループを終了するのではなく、発見時の処理 (逆順の例を表示) をしてから終了する必要があります。つまり、“1. 逆順を発見 → 2. 発見時の処理 → 3. ループを終了” という順番になります。このような場合は、まず 1 で発見したことを記録しておき (例えば、フラグとして変数 found を用意して、これに True を代入し)、2 までをループ内で処理した後、3 でループを終了するかどうかを 1 の記録 (フラグ found の値) により確認する、という流れになります。

以上で繰り返しの基本的な制御構造についての説明を終え、次に while を使った簡単な例として、与えられた値を繰り返し 2 で割って行き、その結果を表示するという手順を示しましょう:

```
def testdiv2(x):
  while x > 0.0:
    print(x)
    x = x / 2.0
```

このアルゴリズムは、純粋数学的には無限に繰り返すように見えます。しかし、コンピュータの計算では、次々に半分にして行くと最後は浮動小数点表現で表わせる最小限界の数になり、さらに半分にすると近似値として「0.0」になるので止まるわけです。1.0 を例に、その辺りを少し見てみると:

```
>>> testdiv2(1.0)
1.0
0.5
...
```

```
1.6e-322
8e-323
4e-323
2e-323
1e-323
5e-324
>>>
```

この結果より、このシステムで使われている浮動小数点表現では、0 でない最も小さい数はおよそ「10^{-324}」くらいであることが分かります。

演習 3-3 testdiv2 関数を参考に、このシステムで扱える最大数を示すプログラムを作成せよ。

> ヒント: Python Ver.3 では、「無限大」は `math.inf` という記号で表わせるようになっています (import math をお忘れなく)。これを条件に取り入れ、与えられた値が無限大より小さい間、2 倍ずつして行けば出来そうですね。

5 制御構造の組み合わせ

簡単なプログラムでは、制御構造として「枝分かれ」あるいは「繰り返し」のどちらか一つだけを使えば済みますが、もう少し込み入ったプログラムになると、ある制御構造 (枝分かれ or 繰り返し) の内側に、さらに別の制御構造を入れる必要が出て来ます (下記の疑似コードは一例です):

- 条件～が成り立つ間繰り返し、
- 　もし～であれば、
- 　　○○をする。
- 　(枝分かれ終わり)
- (繰り返し終わり)

このような例の一つとして、「0～与えられた数までを順に打ち出せ。但し、3 の倍数だけは fizz と打ち出すこと。」というプログラムを考えてみます[2]:

- fizz1: 3 の倍数の時だけ fizz
- 　変数 i を 0 から与えられた数の手前まで変えながら繰り返し、
- 　　もし i が 3 の倍数ならば、
- 　　「fizz」と出力。
- 　　そうでなければ、
- 　　i を出力。
- 　　(枝分かれ終わり)
- 　以上を繰り返し。

これを Python で書いたものは次のようになります (少し複雑ですね):

```
def fizz1(n):
  i = 0
  while i <= n - 1:
    if i % 3 == 0:
      print("fizz")
    else:
      print(i)
    i = i + 1
```

[2]海外で古くからある言葉遊びに、**fizzbuzz** というものがあります。これは、輪になって順に「1, 2, . . .」と数を唱えて行く遊びです。但し、数が 3 の倍数なら「fizz」、5 の倍数なら「buzz」、3 と 5 の公倍数なら「fizzbuzz」と (数の代わりに) 言わなければならず、間違えた人は輪から抜けて行きます。

では動かしてみましょう:

```
>>> fizz1(20)
fizz
1
2
fizz
(途中略)
16
17
fizz
19
```

またこの他に、特定の条件が成立したら繰り返しを途中で抜けたい場合も存在します。Python では、繰り返しを途中で抜ける手段として、**break 文**が用意されています。例えば、上で述べたフラグと break を用いることで、こんなプログラムを作ることができます:

```
found = False
while not found:
  ...
  ...               # ループ内の処理に応じて、found に Ture を設定
  ...
  if (found):
    break           # found が Ture になったら、即座にループを抜ける
  ...
```

このように、基本的な制御構造を組み合わせて行けば、どんなに複雑なプログラムでも作成できます。これはちょうど、簡単な規則と単語から、どんなに複雑な文章でも (日本語や英語で) 作れるのと同じだと考えて下さい。

演習 3-4 まずは、上の fizz1 プログラムを打ち込んでそのまま動かせ。動いたら、繰り返しと枝分かれを組み合わせて次の動作をする Python プログラムを作成せよ。

a. 0 から与えられた数までのうち、2 の倍数でも 3 の倍数でもないものだけを順に打ち出す。

b. 0 から与えられた数までを順に打ち出すが、3 の倍数の時は fizz、5 の倍数の時は buzz、3 の倍数かつ 5 の倍数の時は fizzbuzz と (いずれも数値の代わりに) 打ち出す (fizzbuzz 問題)[3]

c. 0 から与えられた数までを順に打ち出すが、3 が付く数字の時は数値の代わりに hoge と打ち出す。

ヒント: 要は、各桁の数字が 3 かどうかを調べるわけですが、これは以下を繰り返して行くことで分かります。

- 10 で割って余りを調べる。
- 1 桁小さくする。

問題は、例えば 33 が与えられても hoge の出力は 1 回だけ、3 が入っていない場合は数値を出力、といった対応が必要なことです (特に前者)。これを比較演算子による論理演算の組み合わせだけで行なうのは、少々面倒です。このような場合はフラグを利用し、例えば、各桁の数字に一つでも 3 があればフラグを True にして以後の制御に利用する、という方針が良さそうです。

[3]米国では、fizzbuzz 問題のプログラムを書けないプログラマが多いので、プログラマ募集の応募者に対するふるい分けに使っている、という噂があります。本当かなぁ。

付録: 構造化定理の補足

1. 原始流れ図 · 原始箱 · 原始矢線 · 原始処理と構造化定理の証明方針

以下では、構造化定理の証明方法について考えてみます。まずは、4 ページの流れ図 (図 1) を再掲し、この図に登場する箱 (矩形 · 菱形) や矢線の意味を定義することから始めましょう。

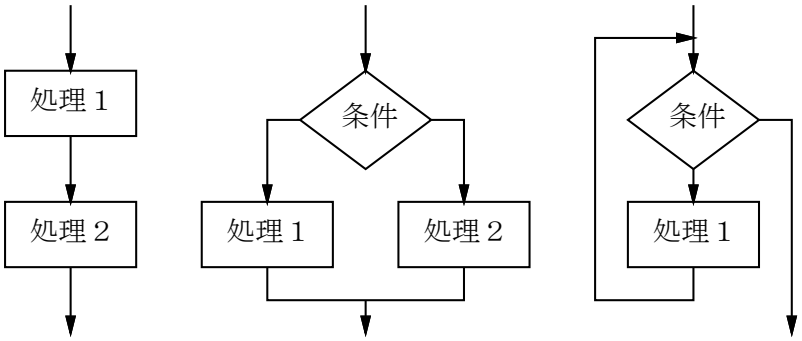

図 2: 三つの基本的な制御構造 (図 1 の再掲)

矩形: 処理の集合を表わします。例えば、計算式一つ (`x = 6+2`) あるいは計算式と関数呼び出し (`x = 6+2; y = 5+3; return(x, y)`) などが入ります[4]。但し、矩形内にある各処理は、次の条件を満たすこととします。

- 矩形内では、常に同じ処理から実行を開始します。例えば、後者の例では常に `x = 6+2` から処理が始まり、外部の状況によって `y = 5+3` から処理が始まる、ということはありません。
- 同、常に同じ処理で実行を終了します。後者の例では、常に `return(x, y)` で処理を終えます。

菱形: 矩形と同じく処理の集合ですが (満たすべき条件も同じ)、最終的な処理の結果に応じて、実行の流れ (制御) が複数の候補から選択された箱に移ります (処理の結果と選択された箱との対応関係は、常に同じです)。

入矢線: 制御が、外部からその箱へ移ることを表わします。

出矢線: 制御が、その箱から外部へ移ることを表わします (ある箱からの出矢線は、別の箱の入矢線になることに注意して下さい)。

また、分岐 (図 2 中) および反復 (図 2 右) にある矢線の交わりについては、次のように考えます。

- 分岐において処理 1 と処理 2 の出矢線が交わるということは、処理 1 および処理 2 の実行後、(図 2 には描かれていませんが) どちらも同じ処理 3 を実行することを意味する。
- 反復において処理 1 の出矢線が条件の入矢線と交わるということは、処理 1 の実行後に条件を実行することを意味する。

以上より、アルゴリズムおよびプログラムと流れ図との対応関係を定義できたので、次に構造化定理と流れ図との関係を定義します。

一般に、全てのプログラムは、流れ図により表現することができます。何故ならば、流れ図の箱は処理の集合なので、プログラム全体を一つの箱だと考えることにより、流れ図で表現したことになるからです[5]。しかし、これではあまり意味がないので、一つの箱で表わす処理の数を減らし、可能な限り細小な箱による制御構造を表現した流れ図を対象にしたいところです。例えば、次のような分岐プログラムを考えてみましょう。

```
if (a + b > 0) or (c + d < 0):
  .....
```

この分岐プログラムにある条件は、“代数演算を行なった後に比較演算を行なう” という処理を 2 回行ない、両者の結果に対してさらに論理演算を行なっています。これは、次のようにして細小化することができます。

[4] ここでの “;” は、処理の区切りを表わすことにします。

[5] もちろん、データの種類に応じて複数の終わり方があるならば、終わり用の箱を用意してそこへ全て連接することにより、入口 (プログラムの開始場所) 一つ/出口 (同、終了場所) 一つの大きな箱と考えることができます。

```
X = a + b; Y = c + d
P = X > 0; Q = Y < 0     ← 代数演算の結果に比較演算を行ない、その結果（True・False）を保存
R = P or Q               ← 比較演算の結果に論理演算を実施
if (R):                  ← 条件式の評価結果に応じて分岐
  .....
```

以下の説明では、このように可能な限り細小化した処理を**原始処理**、原始処理を表わす箱を**原始箱**、原始箱による流れ図を**原始流れ図**と呼ぶことにします。さらに、原始箱を接続する矢線の細小化については、次のように考えます。もし、同じ原始箱間で制御の移り方が二つ以上ある場合には、その数だけ矢線を引くこととします。つまり、矢線 1 本は、制御の移り方一つだけを表わすわけです。これを上と同様、**原始矢線**と呼ぶことにします。ところで、プログラムは常に一つの処理から始まります。例えば Python では、指定された一つの関数から始まり、複数の関数から同時に始まることはありません。つまり、原始流れ図では、始点となる原始箱への原始入矢線は必ず 1 本になります (図 3)。

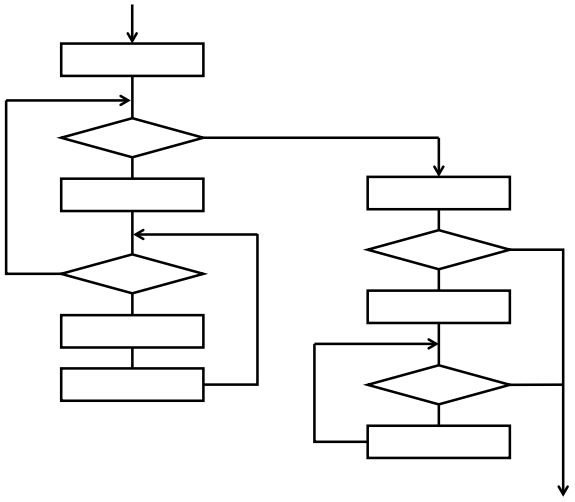

図 3: 原始流れ図の例

ここで、誤解がないように少し補足しておくと、図 2 は原始流れ図ではありません。例えば、次のような分岐プログラムでは、処理 1 と処理 2 にそれぞれ複数の計算式が含まれるため、一つの原始箱では表現できません。

```
if (a > b):
  S = i - j; T = k - l
else:
  S = i + j; T = k + l
```

図 2 の制御構造を原始流れ図により正確に表現し直すと、図 4 のようになります。構造化定理が取り上げる制御構造 (の本質) とは、(連接と) 図 4 における “矢線の二分” および “矢線の出戻り” です。

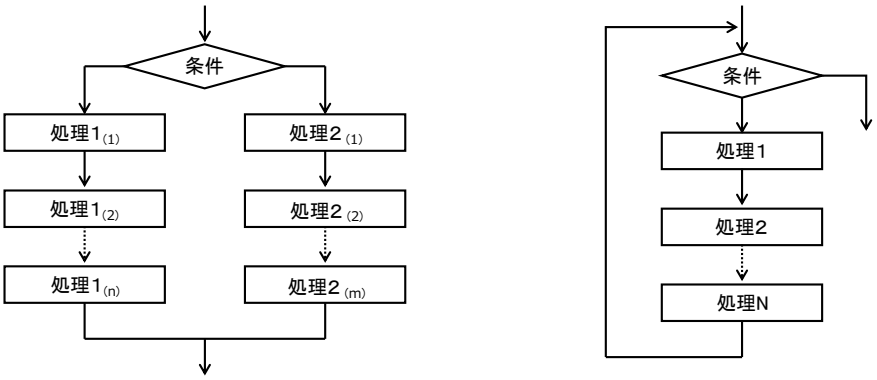

図 4: 原始流れ図を用いた分岐 · 反復の制御構造

以下の議論では、原始流れ図 · 原始箱 · 原始矢線 · 原始処理を対象としますが、特に断わりがない限り、“原始” を省略して表記します。

さて、改めて構造化定理の意味を考えてみましょう。構造化定理とは、

「どんなに複雑な制御構造でも、順次実行 (連接) · 枝分かれ (分岐) · 繰り返し (反復) の三つを組み合わせることで作り出せる。」

というものでした。これを上で定義した原始流れ図に当てはめて考えてみると、

「どんなに複雑な原始流れ図でも、図 2 (11 ページ) にある三つの制御構造 (矢線の繋がり) を組み合わせることで作り出せる。」

となります。流れ図は箱と矢線の接続で表わされますが、流れ図を構成する各箱は、一般に図 5 のような複数の入矢線/出矢線を持つ形状 (即ち、一般的なプログラムの分岐) になります (以下では、一般的という意味で矩形+菱形の箱で表現しています)。

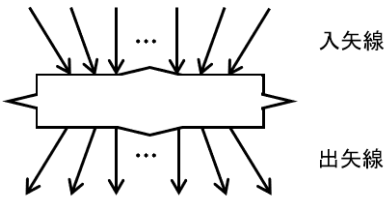

図 5: 箱の一般的な形状

また、矢線の繋がりについては、流れ図を辿って行くと、“既に経由した箱のどれかに戻る” or “戻らずに流れ図の外部へ出る” のどちらかになります (流れ図は有限なので、永遠に辿り続けることはありません)。前者はプログラムの反復を、後者は終了を意味します。つまり、一見複雑に見える原始流れ図には、「一般的な」連接 · 分岐 · 反復しか存在しないというわけです。後は、これら連接 · 分岐 · 反復が、図 2 にある三つの構造で表現できるかを調べれば済みます。よって、構造化定理を証明する方針は、下記のようになります。

- 流れ図を構成する箱は、入矢線を 1 本、出矢線を 2 本以下にできる。
- 流れ図に分岐がある場合は、図 2 中の構造 (条件への入矢線は 1 本/出矢線は 2 本、後続処理二つの出矢線は 1 本に交わる) で表現できる。
- 同、反復がある場合は、図 2 右の構造で表現できる。

以下の節では、まずは単純そうに見える 1 番目の方針から検討を始めることとし、入矢線/出矢線に着目して上記の方針に沿った証明を考えて行きます。

2. 出矢線

まずは、流れ図を構成する箱の出矢線について掘り下げるところから始めてみましょう。出矢線が 3 本以上の場合を考えてみると、これは 3 分岐以上の処理を意味しています。このような処理を仮に `switch` 文と呼び、例えば以下のような構文だったとします (Python の `if`〜`elif`〜`else` 文と同等ですが、表記簡素化のためここでは `switch` 文を使いました)。

```
switch(条件)
  case 結果 1:          ← 条件の評価結果が "結果 1" の場合に、このプログラムを実行
    .....
  case 結果 2:          ← 同、"結果 2" の場合に、このプログラムを実行
    .....
  case 結果 3:          ← 同、"結果 3" の場合に、このプログラムを実行
    .....
  .....
  .....
  default:              ← 同、どの結果にも合わなかった場合に、このプログラムを実行
    .....
```

このような 3 分岐以上の処理は、入れ子構造にすることで、以下のように二分岐の処理の組み合わせに変えることができます。

```
switch(条件)                    ← この分岐を s1 と呼ぶことする。
  case 結果 1:
    .....
  default:
    switch(条件)                ← この分岐を s2 と呼ぶことにする。
      case 結果 2:
        .....
      default:
        switch(条件)            ← この分岐を s3 と呼ぶことにする。
          case 結果 3:
            .....
          default:
            switch(条件)
              .....
              .....
```

この場合、switch 文 s1 が入る箱 (菱形) は、一つの case と一つの default しか出矢線はなく、switch 文 s2 が入る箱は、s1 の default 側から出る出矢線の下に繋がることになります。switch 文 s2 以降も同様です (図 6)。このような分解を繰り返して行くことで、流れ図を構成する箱は、“出矢線の数 ≤ 2” とすることができます。(連接の場合は出矢線が 1 本であることに注意して下さい)。

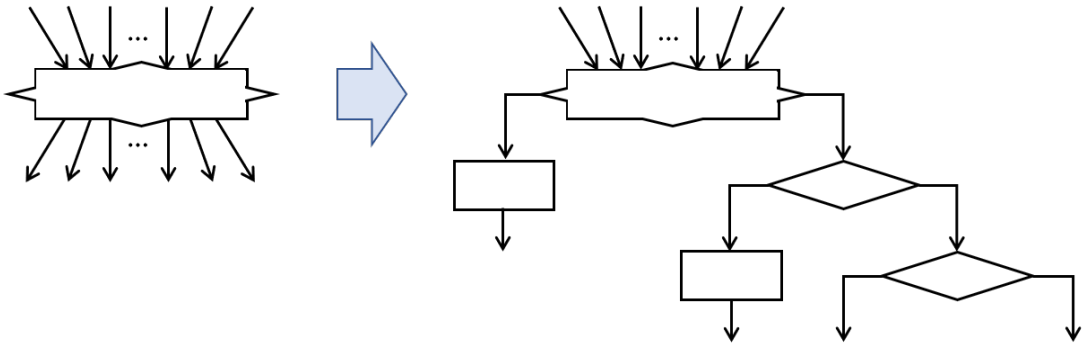

図 6: 出矢線の削減

3. 入矢線

次に、箱へ入る入矢線について考えてみましょう。以下の議論において見通しを良くするため、矢線に沿った道筋 (つまり、“箱 · 矢線 · 箱 · 矢線 ... 箱” という繋がり) を**パス**と呼ぶことにします。また、各箱に対して、流れ図の始点からその箱に入るパスを**上流パス**、その箱から出るパスを**下流パス**と呼ぶことにします。

一般に、ある箱の入矢線は、以下の二つのうちのどちらかに該当すると考えられます。

A: 入矢線は上流パス上にある。

B: 入矢線は下流パス (の延長) 上にある。

B について少し補足しておくと、これは、対象の箱から出る下流パスが、ぐるっと回って同じ箱に再度入ることを意味しています。これをプログラムの実行に対応付けると、パスが同じ箱に入るということは、同じ処理を再度実行することに該当します。つまり、反復です。これより、プログラムに反復があるかどうかを考慮して、A·B に該当する入矢線を考える必要があります。

3-1. 反復がない場合

まずは、プログラムに反復がない場合について考えます。ある箱に着目した時、反復がないことから下流パスが再度その箱へ入ることはないため、入矢線は全て上流パスからになります (つまり、入矢線は前節の A のみ)。以前に説明した通り、流れ図では始点となる箱への入矢線は必ず 1 本です。ある箱において入矢線が複数ある、言い換えれば流れ図の始点からその箱に至るまでに入矢線が増えたということは、上流パスで分岐したことを意味します (理由は脚注[6]を参照)。

さて、図 2 中 (11 ページ) にある分岐では、処理 1, 2 の出矢線を交わらせて一つにしています (その意味は、11 ページで説明しています)。これに留意して、ある箱に複数の入矢線がある場合 (つまり、一般的な分岐が存在する場合)、その上流パスにある (別の) 一般的な分岐を対象に次の手順を用いて入矢線を減らし、かつ図 2 中で示した分岐の構造に置き換えることができます (以下ではパスに着目するので、分岐の出矢線 1, 2 の下流パスをパス 1, 2 と呼ぶことにします)。

1. 対象となる箱 n の入矢線 I, J に至る上流パスを i, j とする。
2. 上流パス i 上にある (一般的な) 分岐の集合を $\mathrm{B}_i = \{\mathrm{b}_1^{(i)}, \mathrm{b}_2^{(i)}, \ldots, \mathrm{b}_m^{(i)}\}$ とする。
3. 各 $\mathrm{b}_x^{(i)} \in \mathrm{B}_i$ のうち、$\mathrm{b}_x^{(i)} \in \mathrm{B}_j$ $(i \neq j)$ を満たし (上流パス i および j の両方に存在し)、かつ箱 n までのパス 1, 2 がそれぞれ上流パス i, j に含まれる $\mathrm{b}_x^{(i)}$ の集合を $\mathrm{B} = \{\mathrm{b}_1, \mathrm{b}_2, \ldots, \mathrm{b}_m\}$ とする (集合の元を改めて b_k と置き直す)。
4. 分岐 b_k $(\in \mathrm{B})$ から箱 n までのパス 1, 2 上に B へ属する他の分岐 b_l が存在しない場合は、図 2 中で示した出矢線の交わりを適用することで、箱 n において二つの入矢線 I, J を一つに交わらせる。

この手順を、入矢線が 1 本になるまで上流パスの全組み合わせに繰り返すことで、最終的に一般的な分岐を図 2 中の分岐に置き換えることができます (矩形 · 矩形の出矢線だけでなく、矩形 · 菱形や菱形 · 菱形の出矢線も同様に扱えます)。

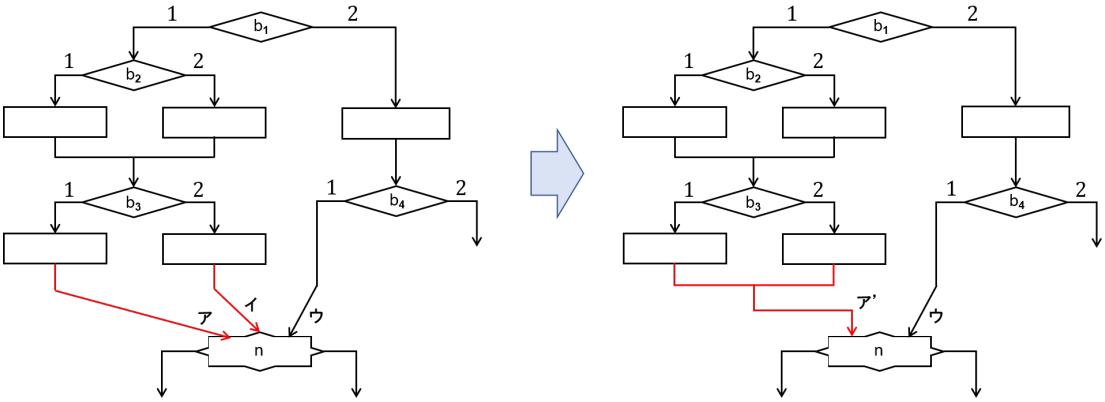

図 7: 入矢線の削減

図 7 の例では、箱 n に 3 本の入矢線「ア, イ, ウ」が存在します。まずは、入矢線「ア, イ」へ至る上流パスを対象に考えてみましょう (「イ, ウ」を対象にした場合は脚注[7]を参照)。この二つの上流パスには (一般的な) 分岐 b_1, b_2, b_3 が存在しますが、分岐 b_1 のパス 2 は入矢線ア or イへ至る上流パスに含まれないので、$\mathrm{B} = \{\mathrm{b}_2, \mathrm{b}_3\}$ となります。分岐 b_2 から箱 n までのパス 1, 2 には B に含まれる分岐 b_3 が存在します。これに対し、分岐 b_3 から箱 n までのパス 1, 2 にはどちらも B に含まれる分岐が存在しません。よって、入矢線「ア, イ」を一つに交わらせることができます (図 7 右の赤矢線「ア'」)。(以後は図に示していませんが) その後は、残りの入矢線「ア', ウ」を対象に同じ処理を行なうことで、最終的に箱 n の入矢線は 1 本にできます。

以上より、プログラムにおいて反復がない場合、それに該当する流れ図を構成する箱は、必ず入矢線の数 = 1 および出矢線の数 ≤ 2 にできることが分かりました。しかしながら、これだけでは反復がない流れ図を全て、図 2 中のような

[6]複数の入矢線は分岐により作られることを、以下に略証します。分岐がない状態で、ある箱 n に複数の入矢線 (つまり上流パス) があると仮定します。流れ図の始点を箱 0 とした場合、箱 n の上流パス 1, 2 は次のように表わせます: 上流パス 1 = {箱 0, 箱 1, ..., 箱 (n-1), 箱 n}, 上流パス 2 = {箱 0, 箱 1', ..., 箱 (n-1)', 箱 n}。ここで、上流パス 1, 2 は分岐を含まない異なるパスなので、箱 1 ～ 箱 (n-1) と箱 1' ～ 箱 (n-1)' は全て異なる箱になります。この時、箱 0 の出矢線は異なる箱 1 および箱 1' に接続しますが、これは分岐がないという前提条件に矛盾します。よって、分岐がなければ入矢線は一つになります。この対偶を取ることで、複数の入矢線は分岐により作られることになります。

[7]例としては少々分かりにくいですが、入矢線「イ, ウ」を対象にした場合、b_2, b_3 ではパス 1, 2 が入矢線ウへ至る上流パスに含まれないため、$\mathrm{B} = \{\mathrm{b}_1\}$ となります。その後は「イ, ウ」が一つに交わり (以下「イ'」と呼ぶ)、次は入矢線「ア, イ'」が対象となります。入矢線イ' へ至る上流パスは複数存在しますが (分岐 b_2, b_3 を含む上流パスと、分岐 b_4 を含む上流パス)、分岐 b_2, b_3 を含む上流パスを対象にした場合は、入矢線「ア, イ」を対象にした場合と同等な処理を行ない、また、分岐 b_4 を含む上流パスを対象にした場合は、入矢線「イ, ウ」を対象にした場合と同等な処理を行ないます。

箱で構成できたとは言い切れません。この図にある分岐の制御構造は、下記の条件を備えています。

- 処理 1, 2 の入矢線は 1 本で、分岐から直接出ている。
- 同、出矢線は 1 本である。

これに対し図 7 の流れ図は、図 8 にある黒線部分のように入矢線を 1 本にできましたが、分岐 b_1 あるいは b_4 の処理 1, 2 はこの条件を満たしていません。しかし、プログラムは (様々な流れがあったとしても) 同じ場所で終了することから (11 ページの脚注 5 を参照)、終了処理の前では入矢線が一つになっているはずであり、分岐 b_4 のパス 1, 2 はその下流パスにある箱 n' で必ず交わります (青線部分)。よって、入矢線は 1 本だが、図 2 中の条件を満たさない場合とは、図 9 左のような複数の分岐が互いに組合った構造となります。ここでは、赤枠で示した処理 1 は処理 2 からの入矢線があり (緑線)、また処理 2 は出矢線が 2 本です。

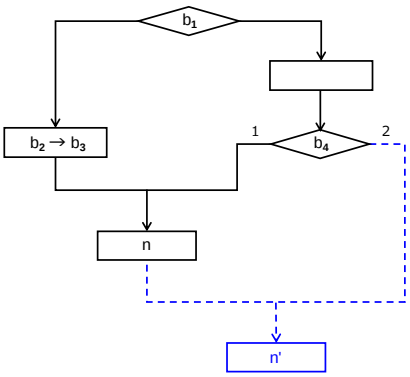

図 8: 入矢線を削減した状態

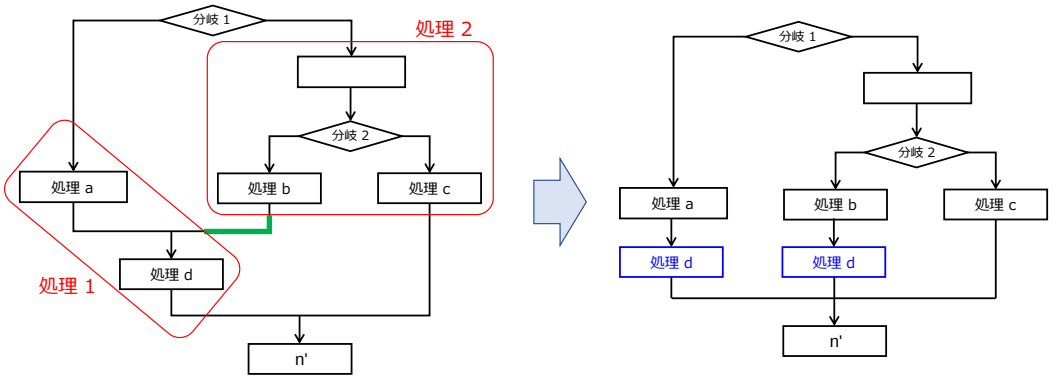

図 9: 互いに組合った分岐構造の解消

この問題を解決するために、改めて流れ図の意味について整理してみます。流れ図はプログラムに対応していることから、実は次のような解釈 (or 対応付け) をすることができます。

“流れ図に箱が追加される” = “プログラムが追加される” = “プログラムのどこかに新たなコードが記述される”

これより、流れ図では、結果として副作用がなければ、箱の追加 (= コードの追記) や矢線の変更 (= 制御の変更) を自由に行なってよいことが分かります。これに留意して図 9 左を見てみると、箱 n' に至る制御の流れは、1) 処理 a → 処理 d → 箱 n', 2) 処理 b → 処理 d → 箱 n', 3) 処理 c → 箱 n' の 3 本あり、前者の二つはどちらも (必ず) 処理 d を経由しています。つまり、処理 d の箱を副作用なくもう一つ用意できるというわけです (図 9 右)。よって、互いの処理が組合った複数の分岐も図 2 左 · 中で構成できることが分かりました[8]。

以上より、反復がない流れ図は、図 2 左 · 中の制御構造だけで構成 (表現) することができます。

3-2. 反復がある場合

次に、プログラムが反復を含む場合を考えます。反復が存在する場合の入矢線には、上流パス上の入矢線と下流パス (の延長) 上の入矢線の 2 種類が存在します (つまり、付録 3 節にある A · B の両方)。箱に複数の入矢線がある場合、反復と関わりを持たない上流パス上の入矢線は、前節の議論より 1 本にできます。ここで、反復に関係する上流パス上の入矢線 (上流パスのどこかでループを抜けて来た入矢線) や、下流パス上の入矢線 (下流パス上の入矢線は全て反復と関係していることに注意) についても前節と同様な議論を用いて本数を減らし、かつ一般的な反復を図 2 右 (11 ページ) にある構造で表現できれば良いのですが、残念ながらそう単純ではありません。図 2 右にある反復の制御構造は、下記の条件を備えています。

[8]図 9 右は、箱 n の直前で 3 本の矢線が交わっており、図 2 中の制御構造とは異なるように見えます。11 ページにある矢線の交わりの定義から考えると、これは処理 a → 処理 d, 処理 b → 処理 d, 処理 c の実行後に、同じ処理 e を行なうことを意味します。そこで、二つの処理 d と処理 e の間へ常に同じ場所で終わる (出矢線は 1 本) 処理 e' を一つ追加し、処理 e' の箱は空箱あるいは全く無関係 (or 無意味) な処理が入った箱だとすれば、処理 a → 処理 d, 処理 b → 処理 d 側から処理 e への入矢線を一つにすることができます (処理 e' への入矢線は 2 本ありますが、これは前ページの手順で一つにできます)。以下、処理 e', 処理 c の出矢線が (処理 e の前で) 交わることになるので、図 2 中と同じ構造にできます。

P: 処理 1 からの出矢線は、反復の条件を判定する分岐への入矢線と交わる。

Q: ループの外部からループの内部へ矢線が直接入らない。

R: ループの内部からループの外部へ矢線が直接出ない。

流れ図に存在する反復の制御構造には様々な種類が存在するものの、構造化定理で取り上げる反復は、反復の条件を判定する分岐から始まり、この分岐からループを抜ける構造でなければなりません。

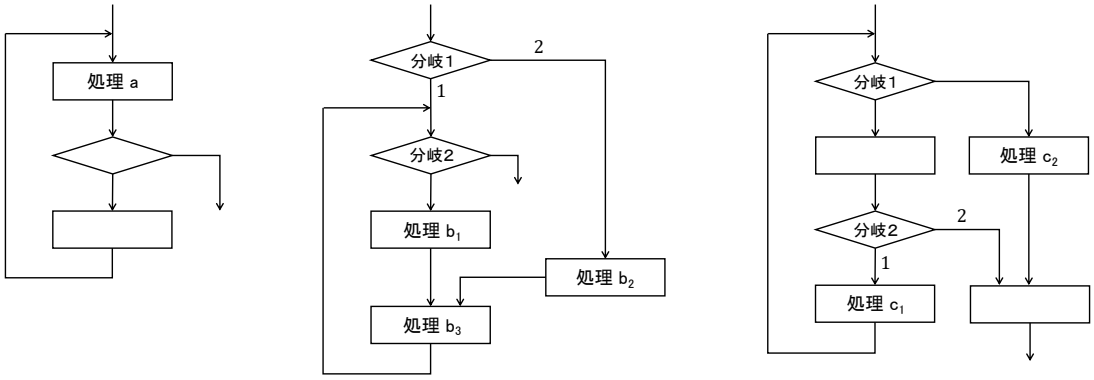

図 10: 条件 P·Q·R を満たさない反復の一般的な構造

条件 P·Q·R を満たさない、一般的な反復の構造を図 10 に示します。図 10 左では、矢線は交わった後で処理 a に入っています。つまり、処理 a はループの内部として実行されます。しかし一方で、初めてループへ入る時は処理 a を実行してから入るので、ループの外部としても実行されています。この場合、矢線の交わりを無条件に移動するわけには行きません。図 10 中では、ループの外部にある分岐 1 から出た矢線 2 (の下流パス) が、ループの内部へ直接入っています。その際、矢線 2 の下流では、処理 b_2 → 処理 b_3 → 処理 b_1 という順番で処理が実行されますが、矢線 1 の下流では、処理 b_1 → 処理 b_3 という順番で処理が実行されます。この場合、分岐 1 から出る二つの矢線を、“ループの外部” で図 2 中のように無条件で交わらせることはできません。また、図 10 右では、ループの内部で分岐をしています。ここでは、分岐 2 の矢線 1 から分岐 1 を経由してループ外に出る場合は、処理 c_1 → 処理 c_2 と実行しますが、分岐 2 から直接ループ外に出る矢線 2 ではこれらを実行しません。この場合も、分岐 2 から出る二つの矢線を、“ループの内部” で図 2 中のように無条件で交わらせることはできません。よって以下では、図 2 にある三つの制御構造を組み合わせて、条件 P·Q·R を実現する方法について考えます。

条件 P の実現

図 10 左は、処理 a をループの内部として扱いたいが (つまり、分岐の下に繋がるイメージ)、処理 a は初めてループへ入る前にも実行されるので、このままでは矢線の交わりを移動できないという問題です。この問題の本質は、処理 a がループの外と内の両方で実行されていることです。何も考慮せず、無条件で処理 a の位置や矢線の交わりを変えることはできないものの、付録 3-1 節でも説明したように、結果として副作用がなければ、流れ図には箱の追加 (= コードの追記) や矢線の変更 (= 制御の変更) を行なうことができます。これより、処理 a に該当するコード (即ち箱) をもう一つ用意してループの外と内の両方に配置すれば、処理 a をループの外と内の両方で実行することができます。またこの時、ループの外にある処理 a については初めてループへ入る時に 1 回実行すればよいので、矢線の交わりを図 11 のように処理 a と分岐の間に移すことができます。

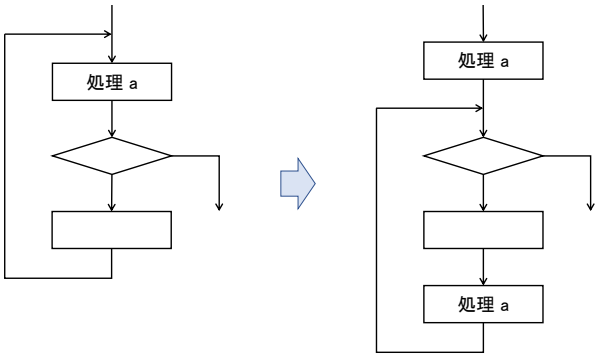

図 11: 条件 P の実現

図 11 では、ループの外にある処理 a の箱にはループ内からの矢線が接続されておらず、また、ループの中にある処理 a の箱にもループ外からの矢線は接続されていないため、これら箱の追加および矢線の変更には副作用がありません。

以上より、図 2 にある制御構造を組み合わせて条件 P を実現することができました。

条件 Q の実現

次に図 10 中は、ループ外にある分岐 1 から出る二つの矢線 1, 2 を “ループの外部” で交わらせ、反復の条件を判定する分岐 2 へ入る 1 本の矢線にしたいという問題です。矢線 2 の下流では、処理 b_2 → 処理 b_3 (→ 処理 b_1) という順番で処理が実行されますが、矢線 1 の下流では、処理 b_1 → 処理 b_3 という順番で処理が実行されるため、ループ外で 1 本の矢線にすることができません。それを妨げている問題の本質は、矢線 1, 2 で各処理を実行する順番が異なるという部分です。よって、条件 P の実現と同様、副作用のない箱を追加することで実行の順番を変える方法を考えます。但し、条件 P の場合と異なり、単純に各処理の場所を変えるだけ (or 複製を作るだけ) ではうまく行きません。例えば、矢線 1, 2 が交わって 1 本になった後、そのままでは処理 b_2 を実行してよいかどうか分からないからです。この判断は、交わる前にどちらの矢線を経由したかによって決まります。そこで、どの矢線を経由したかを記録するフラグを用意し、フラグの値に応じて実行の順番を変えることにします。

図 12 では、ループ外の分岐 1 において、どの矢線から出たかをフラグ f に記録しています。その後、二つの矢線を交わらせて 1 本にした後、反復の条件を判定する分岐 2 に入れています (赤矢線)。ループ内では、f の値に応じて (つまり、経由した矢線に応じて) 処理 b_2 を実行します。処理 b_2 の実行は 1 度だけなので、その後は f の値を矢線 1 に変えます (青矢線)。但し、分岐 2 の条件によっては、分岐 1 から初めて分岐 2 へ来た時に弾かれ、“f == 矢線 2” であってもループへ入れず、処理 b_2 を実行できない場合が考えられます。これに対応するため、分岐 2 では “f == 矢線 2” の場合、必ずループへ入るように変更します [9]。これらの修正により、ループ外にある分岐から出る二つの矢線をループの外部で交わらせ、反復の条件を判定する分岐へ入る 1 本の矢線に変えることができるため、ループを図 2 右の形状にできます。

また、図 12 には描いていませんが、ループ外にある分岐 1 以外の分岐 0 から出た矢線が、このループ内へ入る場合も考えられます。このような場合は、分岐 0 用のフラグ f’ を別途用意し、上と同様な繋ぎ替えを追加する (+該当する分岐に条件を追加する) ことで修正できます。

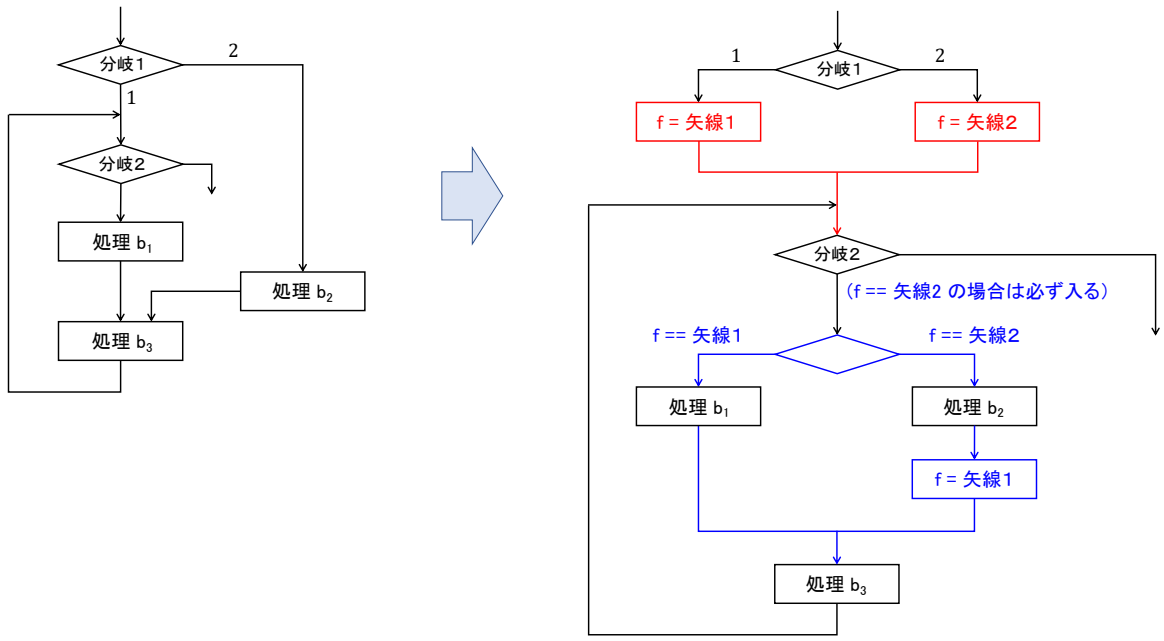

図 12: 条件 Q の実現

図 12 の修正では、フラグの設定や比較は他の処理から独立して実行可能で、既存の処理に影響を及ぼしません。また、ループの外部で “f = 矢線 2” を設定する箱には矢線 1 が接続せず、ループへ入るまでにフラグ f の値を変更する箱は他に存在しません。よって、矢線 2 を経由してループへ入った直後は必ず “f == 矢線 2” が成立し、処理 b_2 が実行されます。処理 b_2 の実行後は “f = 矢線 1” が設定されるため、以後のループ内では毎回処理 b_1 が実行され、処理 b_2 が実行されることはありません。よって、これら箱の追加および矢線の変更には副作用がありません。

以上より、図 2 にある制御構造を組み合わせて条件 Q を実現することができました。

[9] より正確に書けば、これは分岐 2 へ入る矢線の直前に (分岐 2 への入矢線と処理 3 からの出矢線との交わりの直後に)、「“f == 矢線 1” ならば分岐 2 へ入る / “f == 矢線 2” ならば青分岐へ入る」という分岐を追加することになります。追加した分岐と分岐 2 との間には処理 (矩形) が存在しないため、両者を併せて条件が複数ある一つの分岐と考えることが可能です (11 ページにある菱形の定義に抵触していません)。よって、この分岐の追加は条件 Q の実現を妨げません。

条件 R の実現

最後に図 10 右ですが、これも図 10 中と同様、ループ内の分岐 2 から出る二つの矢線 1, 2 を交わらせ、反復の条件を判定する分岐 1 から出る 1 本の矢線にしたいという問題です。矢線 1 から分岐 1 を経由してループ外に出る場合は、処理 c_1 → 処理 c_2 と実行しますが、直接ループ外に出る矢線 2 ではこれらを実行しないため、ループ内で 1 本の矢線にすることができません。それを妨げている問題の本質は、矢線 1, 2 で処理 c_1 および c_2 を実行するかどうかが異なるという部分です。つまり、二つの矢線が交わって 1 本になった後、そのままでは処理 c_2 を実行してよいかどうかが分からないということです。この問題もこれまでと同様、副作用のない箱を追加する方法で解消できます。処理 c_1 および c_2 を実行するかどうかの判断は、交わる前にどちらの矢線を経由したかによって決まるため、どの矢線を経由したかを記録するフラグを用意し、フラグの値に応じて実行の有無を変えることにします。

図 13 では、ループ内の分岐 2 において、どの矢線から出たかをフラグ f に記録しています。そして、f の値に応じて (つまり、経由した矢線に応じて) 処理 c_1 を実行します。その後、二つの矢線を交わらせて 1 本にした後、反復の条件を判定する分岐 1 に入れています (赤矢線)。分岐 1 では、f に矢線 2 を経由したことが記録されている場合、必ずループを抜けるように変更します [10]。ループを抜けた後は、矢線 2 を経由して抜けたかどうかに応じて、処理 c_2 を実行します (青矢線)。これらの修正により、ループ内にある分岐から出る二つの矢線をループの内部で交わらせ、反復の条件を判定する分岐へ入る 1 本の矢線に変えることができるため、ループを図 2 右の形状にできます。

また、図 13 には描いていませんが、ループ内にある分岐 2 以外の分岐 0 から出た矢線が、ループ外へ出る場合も考えられます。このような場合は、(条件 Q と同じく) 分岐 0 用のフラグ f' を別途用意し、上と同様な繋ぎ替えを追加する (+該当する分岐に条件を追加する) ことで修正できます。

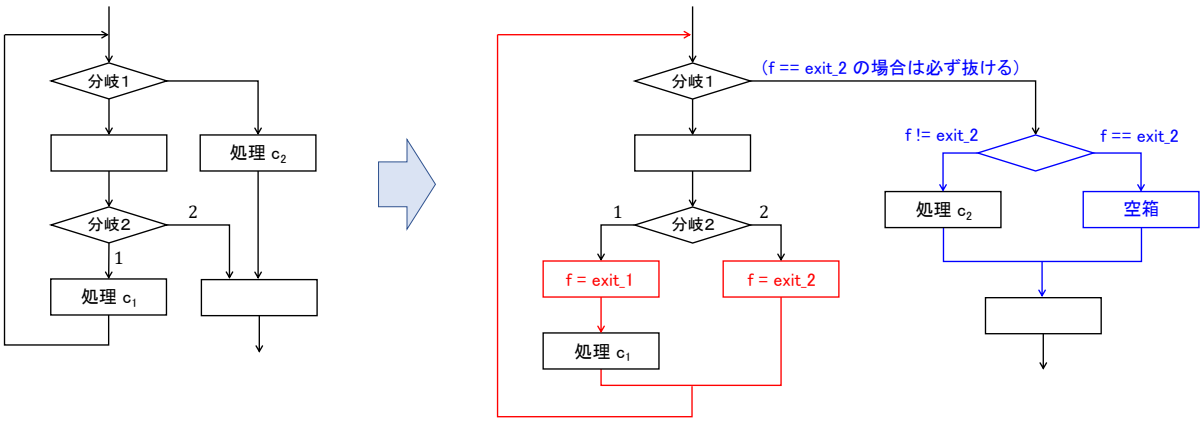

図 13: 条件 R の実現

図 13 の修正も図 12 と同様、フラグの設定や比較は他の処理から独立して実行可能なため既存の処理に影響を及ぼさないこと、フラグの設定において関係のない箱からの接続がないこと、フラグの設定や参照と対象の処理とは 1 対 1 に対応していることから、これら箱の追加および矢線の変更には副作用がありません。

ところで、処理 c_2 より下にある箱については特に制限を設けていないので、下流に分岐 1 が含まれる場合も想定されます。以下では、青分岐の下流パスにある分岐 0' の出矢線が、分岐 1 への入矢線になる場合を考えてみます (図 14 左/次ページ)。この場合は、分岐 0' から分岐 1 へ入り、分岐 1 からループを抜けた後に再び分岐 0' へ至る可能性があります。これは、分岐 0' から始まるループ 0' が存在し、分岐 1 から始まるループ 1 はループ 0' に含まれていると考えることができます (図 14 右)。図 14 右の流れ図では、以下に述べる修正が必要です。

ループ 0' では、内部の処理 d_3, d_4 からループ外に出る可能性があります (これを明示するため、図 14 では処理 d_3, d_4 の箱を一般的な形状にしています)。これは条件 R の反例に該当するため、図 13 と同様な繋ぎ替えにより修正が可能です。また、初めてループ 0' へ来た時は、分岐 1 から処理を始める必要がありますが (図 14 右では青矢線で明示)、これは条件 Q の反例に該当するため、図 12 と同様な繋ぎ替えにより修正が可能です。処理 d_2 については、分岐 0' の下流パスとして実行される場合と、分岐 0' を経由せず分岐 1 のループ内だけで実行される場合の 2 通りがあります。これまで同様、これは分岐 1 の出矢線を示すフラグで制御可能ですが、図が煩雑になるため、ここでは処理 d_2 の箱を増やすことで対応しています (図 14 右では赤箱で明示)。

[10] より正確に書けば、これは分岐 1 へ入る矢線の直前に、「"f != exit_2" ならば分岐 1 へ入る/ "f == exit_2" ならば青分岐へ入る」という分岐を追加することになります。脚注 9 で述べたように、この追加された分岐と分岐 1 を併せて条件が複数ある一つの分岐と考えることが可能であり、この分岐の追加は条件 R の実現を妨げていません。

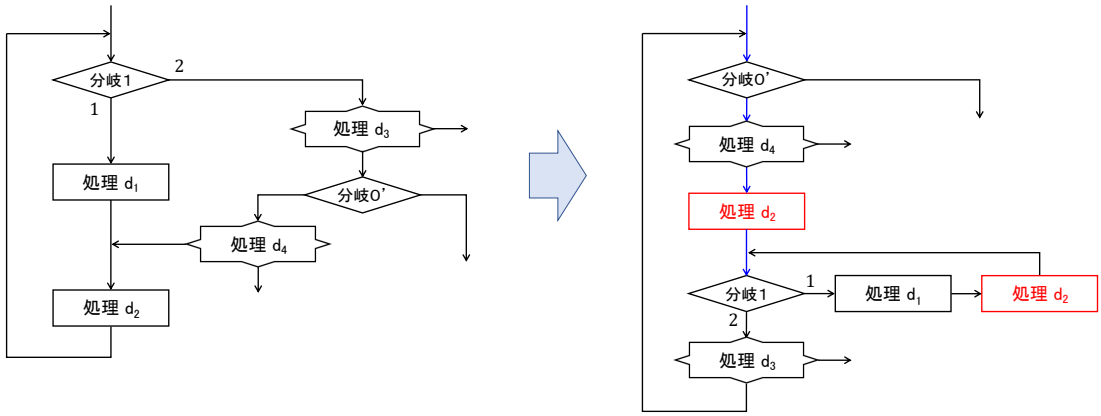

図 14: 青分岐の下流に分岐 1 が存在する場合

さらに、分岐 0' から出る二つの矢線 1', 2' の下流パスが、両方とも分岐 1 への入矢線になる (分岐 1 から始まるループ 1 へ入る) 場合も想定されます (図 15 左)。これについては、次のように修正できます。まずは矢線 1' 側について、図 14 と同様に修正します (図 15 中)。さらに、矢線 2' 側も図 14 左と同様な構造なので、これも同じように修正できます。図 15 右にある分岐 0' ～ 処理 d_3 の流れ図を確認して下さい。

但し、図 15 左では、矢線 1', 2' ともに下流パスが分岐 0' への入矢線になっています。つまり、この反例では、ループ 0' に入る/出るを判断する分岐は分岐 0' ではなく、(一般的) 処理 d_3 ～ d_5 に含まれている (隠れている) わけです。よって、ループ 0' を包含するループ 0" を用意し、分岐 0" でループ 0' に入る/出るの判断を行ないます (図 15 右)。処理 d_3 ～ d_5 では、ループ 0' の外へ分岐が生じた場合に (‡1)、それをフラグへ記録しておきます (図 13 の分岐 2 と同じ修正/ ループ外への分岐が消えるので赤矩形で明示)。分岐 0" ではフラグに応じて、‡1 の場合は矢線 2" を、それ以外の場合は 1" を選択します。さらに、ループ 0' の外にある分岐 α では、どの処理で分岐が生じたかのフラグに応じて、処理 d_3 ～ d_5 の出矢線が繋がる処理 (図 13 左で分岐 2 の出矢線 2 が繋がる箱) を実行します。

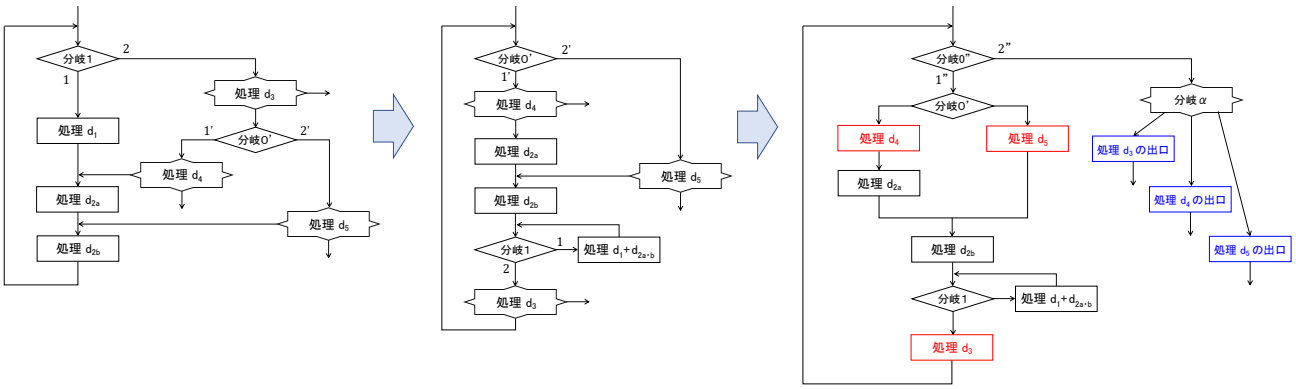

図 15: 分岐 0' のどちらの出矢線にも下流に分岐 1 が存在する場合

以上より、図 2 にある制御構造を組み合わせて条件 R を実現することができました。

3-3. 反復がある場合の入矢線の数

これまでの議論により、どのような反復の形状でも、条件 $P \cdot Q \cdot R$ を満たす図 2 右 (11 ページ) の形状に修正できることが分かりました。ここで改めて図 2 右を見てみると、反復の制御構造では矢線の数が増えていないことが分かります (つまり、制御構造全体で “入矢線 = 出矢線 = 1 本”)。よって、15 ページの脚注 6 と同じ議論により、ある箱 n において反復に関係する入矢線が複数ある場合は、上流パスおよび下流パスで分岐したことを意味します[11]。

以下では、これを用いて、反復に関係する入矢線の分岐を対象に、入矢線の数を減らす方法について考えます。条件 $Q \cdot R$ を満たす反復の制御構造では、分岐がループの外部および内部に跨ることはありません。ループの外部および内部に跨るような分岐は、フラグを導入することで、ループ外あるいはループ内に閉じ込めることができるからです。つまり、箱 n において、反復に関係する上流パス上にある複数の入矢線 (上流パスのどこかでループを抜けて来た複数の入矢

[11] 下流パス上の入矢線については、下流パス 1 = {箱 n, 箱 (n+1), ..., 箱 (n+m), 箱 n}, 下流パス 2 = {箱 n, 箱 (n+1)', ..., 箱 (n+m)', 箱 n} と考えることで (つまり、箱 n は流れ図の始点でもあり、パスの終点でもあるとして扱う)、同じ議論ができます。

線) は、ループの外部にある分岐で増えた矢線と言えます。よって、15 ページと同じ手順により、箱 n の入矢線を減らすことができます。

下流パスについては、次のように考えます。箱 n に下流パス上の入矢線があるということは、箱 n はループ内にあることを意味します。そして、ループ内の分岐はループ内に閉じているため、ループ内だけを対象に 15 ページと同様な手順を実行することで、箱 n の入矢線を減らすことができます[12](条件 R の処理 c_2 より下の箱として分岐 1 が含まれる場合などが該当しますね)。箱 n が、複数のループに階層的に (入れ子状に) 含まれている場合は、一番小さいループから順に、各ループに閉じてこの手順を実行して行けば入矢線を減らせます。

以上より、プログラムに反復がある場合、それに該当する流れ図では、反復の制御構造は必ず条件 P·Q·R を満たし、また、流れ図を構成する箱は、必ず入矢線の数 = 1 および出矢線の数 ≤ 2 となります。これを言い換えれば、反復がある流れ図は、図 2 左 · 中 · 右の制御構造だけで構成することができます。

4. まとめ

付録 2 節および 3 節での議論により、プログラムの制御を示す流れ図は、図 2 左 · 中 · 右の制御構造だけで構成することができます。従って、全てのプログラムは、連接 · 分岐 · 反復を組み合わせるだけで作成できます。

最後に

分岐については、分岐の中に分岐が存在しても (分岐が入れ子状態になっても) 図 2 中の構造が相似的に保たれるため、複雑な場合分けは不要で見通し良く議論できました。しかし、反復については、反復の中に反復が存在すると、“外側ループ/内側ループ + 矢線がループ内に直接入る/出る” 構造が生じます。これは、図 2 右の構造が相似的に保たれているとは言えないため、複雑な議論 (場合分け) が必要となりました。

[12] 15 ページの手順では、入矢線に至る上流パスを辿って分岐を調べました。ここでは、箱 n を流れ図の始点と想定し、入矢線に至る下流パスを辿って分岐を調べます (脚注 11 と同じ考え方ですね)。