プログラミングプロジェクト＃4 – 手続きと抽象化+再帰

久野 靖 (電気通信大学)

2017.8.16

1　はじめに

今回の主な内容は次の通りです。

- 手続き (メソッド) による抽象化
- 再帰的な呼び出し

手続きが使えるとプログラムを見通しよく書けるようになります。また、「再帰」の考え方を使うと、複雑な問題がこなせるようになります。ぜひマスターしてください。

2　前回演習問題の解説

2.1　演習 1 — fizzbuzz

演習 1 は繰り返しと枝分かれの基本的な組み合わせなので、さっさと Ruby コードを示します。 まず 2 の倍数と 3 の倍数以外を打ち出すものから。

```
def fizz2
  100.times do |i|
    if i % 2 != 0 && i % 3 != 0
      puts(i)
    end
  end
end
```

条件が読みにくいかもしれませんが、「2 の倍数でなく、3 の倍数でもないもの」を打ち出すと考えれば、これでよいと分かります。

別案として、条件を変換するかわりに「素直に」枝分かれしてしまい、打ち出さないのは「何もしない」という案もあります。

```
def fizz2
  100.times do |i|
    if i % 2 == 0
      # do nothing
    elsif i % 3 == 0
      # do nothing
    else
      puts(i)
    end
  end
end
```

「何も書いてない」のは不安なので、「何もしない」というコメントを書いてあります。この方が分かりやすいでしょうか。

次に演習 1b ですが、これは if-else の連鎖で「3 の倍数」「5 の倍数」「3 と 5 の公倍数 (15 の倍数)」「それ以外」の 4 つに場合分けするのが一番素直です。

```
def fizzbuzz1
  100.times do |i|
    if i % 15 == 0
      puts('fizzbuzz')
    elsif i % 3 == 0
      puts('fizz')
    elsif i % 5 == 0
      puts('buzz')
    else
      puts(i)
    end
  end
end
```

なぜ 15 の倍数を最初に調べているのか分かりますか。それは、else-if の連鎖では上から順に条件を調べていくので、先に「3 の倍数」や「5 の倍数」を調べてしまうと、15 の倍数は 3 や 5 の倍数でもあるので条件が成り立ってその枝が選ばれてしまい、15 の倍数だけの枝には決して来なくなってしまうからです。

ところで、せっかく「3 の倍数なら fizz」「5 の倍数なら buzz」「両方の倍数なら fizzbuzz」となっているのだから、両者をうまく組み合わせたいと思う人もいるかもしれません。そのようなコードも作ってみましょう。[1]

```
def fizzbuzz2
  100.times do |i|
    num = i
    if i % 3 == 0
      print('fizz'); num = ''
    end
    if i % 5 == 0
      print('buzz'); num = ''
    end
    print(num); puts
  end
end
```

考え方としては、変数 num に打ち出す数を入れておきますが、3 の倍数なら fizz、5 の倍数なら buzz を打ち出すとともに num には空っぽの文字列を入れ直すので、最後の print で何も出力しなくなります。3 や 5 の倍数でないなら num には i が入っているので、そのまま出力されます。パラメタなしの puts は最後に行換えをするためです。

ところで、このコードはよくできている、と思いますか? 個人的には、このコードは先のコードより分かりにくいので、好きではありません。2 つの if の切れ目のところに合流がありますし、最後の数の出力を抑制するために変数 num を使ったりして、流れを追うのが難しくなっています。そんなこ

[1] print は puts と同様に文字列や数値を打ち出すけれども、行換えはしないメソッドです。なぜこれを使うかというと、fizz と buzz をくっつけて 1 行に打ち出すためです。

とをするより、4 方向に枝分かれしてそれぞれの場合を明快に処理する先のコードのほうがずっと読みやすくてスマートだと思うのですが、どうでしょうか?

最後は「世界のナベアツ」ですが、「3 がつく数」はどうしましょうか。それは、対象とする数が 1 桁または 2 桁なので、「3 がつく」というのは 1 桁目が 3 か、2 桁目が 3 という意味になります。1 桁目が 3 というのは、10 で割った余りが 3 ということですし、2 桁目が 3 というのは、10 で切捨て除算した結果が 3 ということですね。ここまで分かればあとは書くだけです。

```
def fizz3
  100.times do |i|
    if i % 3 == 0 || i % 10 == 3 || i / 10 == 3
      puts('aho')
    else
      puts(i)
    end
  end
end
```

2.2 演習 2 — 最大公約数

課題の擬似コードを Ruby に直したものは次のとおり。

```
def gcd1(x, y)
  while x != y
    if x > y
      x = x - y
    else
      y = y - x
    end
  end
  return x
end
```

なぜこれで最大公約数が求まるのでしょうか? 次のように考えてください (x と y は正の整数であるものとします)。

- $x = y$ であれば、$gcd(x, y)$ は x そのもの。当然ですね。
- $x > y$ であれば、$gcd(x, y)$ は $gcd(x - y, y)$ に等しい。[2]
- したがって、$x - y$ を改めて x と置いて $gcd(x, y)$ を求めればよい。
- $x < y$ の場合も同様。
- この手順の反復ごとに、x または y のどちらかがより小さくなるが、0 以下にはならない (大きい方から小さい方を引くから)。
- ということは、この反復は有限回で止まる。
- ということは、そのとき $x = y$ が成り立ち、x が一番最初の x と y の最大公約数に等しい。

繰り返しを使うときは「必ず止まって、止まった時には求める状況が成り立っている」ように設計する、という感じがお分かりになりましたか?

[2]証明: 最大公約数を G と置くと、x も y も G の整数倍なのだから、$x - y$ もまた G の整数倍です。ということは、G は $x - y$ と y の公約数です (最大かどうかはまだ分かりません)。ところで、もし最大公約数で「なかった」とすると、最大公約数 $H(> G)$ が別にあるわけで、H は y の約数かつ $x - y$ の約数になります。ということは、H は $x - y + y = x$ の約数でもあります。これは G が x と y の最大公約数であるということに矛盾します。したがって G は $x - y$ と y の最大公約数でもあることになります。

2.3 演習 3 — 素数判定

素数の判定ですが、擬似コードは次のとおり。

- isprime1: N が素数か否かを返す
- 　sosu ← 「真」。
- 　i を 2 から $N-1$ まで変化させながら繰り返し、
- 　　もし N が i で割り切れるならば、sosu ←「偽」
- 　繰り返し終わり。
- 　sosu を返す。

変数 sosu は先に「はい」を表す true を入れておきます。そして素数でないと思ったら「いいえ」を表す false 入れ、最後に結果がどちらか見ます。このような使い方の変数のことを**旗** (flag) と呼びます。最初「旗」が立っていて、後で見たら「旗」が降りていたとすれば、誰が降ろしたかは分からないとしても、少なくとも誰かが旗を降ろしたことは確実に分かるわけです。では Ruby コードを見てみましょう。

```
def isprime1(n)
  sosu = true
  2.step(n-1) do |i|
    if n % i == 0 then sosu = false end
  end
  return sosu
end
```

2.4 演習 4 — 素数列挙とその改良

もっとも素朴な素数列挙プログラムは、上の素数判定を利用すれば簡単にできます。Ruby のコードを直接示しましょう。

```
def primes1(n)
  2.step(n-1) do |i|
    if isprime1(i) then puts(i) end
  end
end
```

これを手もとのマシンで動かしてみましたところ、10 秒間でおよそ 17,000 まで調べられました。これはあまり速くはないです。ところで、先の素数判定 isprime は「割り切れる」と分かってもそこでやめないで n の手前までずっと割っていきますから、早い段階で割り切れた数に対しては非常に無駄が大きいと思われます。そこで、改良版を作ってみました。

```
def isprime2(n)
  2.step(n-1) do |i|
    if n % i == 0 then return false end
  end
  return true
end
```

こちらは割り切れると分かったら直ちに return で「いいえ」を返しますから、無駄な割り算はしなくてすみます。上の primes1 をこちらを使うように直したところ、10 秒間で 50,000 くらいまで調べられました。つまり速度が 3 倍くらいになったわけです。

さらに考えると、割り算は $N-1$ までやる必要はなく、$\sqrt{N}$ まで調べて割り切れなければそれ以上やっても割り切れないと分かります ($\sqrt{N}$ よりも大きい因数があるなら、小さい因数もあるはずですから)。そこで素数判定を次のように改良します。

```
def isprime3(n)
  2.step(Math.sqrt(n)) do |i|
    if n % i == 0 then return false end
  end
  return true
end
```

これで試してみると、10 秒間で 800,000 くらいまで調べられました。

次に、2 は素数であり、2 の倍数は素数でないことが分かり切っているので調べる必要がない、ということを利用しましょう。このため、3 以上の奇数だけで割ってみる「改編版」の素数判定を作って使います。

```
def isprime4(n)
  3.step(Math.sqrt(n), 2) do |i|
    if n % i == 0 then return false end
  end
  return true
end
def primes4(n)
  puts(2)
  3.step(n-1, 2) do |i|
    if isprime4(i) then puts(i) end
  end
end
```

2 は「別建てで」出力しています (これでも仕様としては合ってるわけです)。こんどは 10 秒間で 1,000,000 くらいまで調べられました。

もうちょっとだけ頑張って、では 2 と 3 より大きい素数は 6 の倍数± 1 だけ (それ以外は 2 と 3 の倍数になってしまう)、ということを利用してさらに調べる数を減らしてみましょう。

```
def isprime5(n)
  6.step(Math.sqrt(n), 6) do |i|
    if n % (i-1) == 0 then return false end
    if n % (i+1) == 0 then return false end
  end
  return true
end
def primes5(n)
  puts(2)
  puts(3)
  6.step(n-1, 6) do |i|
    if isprime5(i-1) then puts(i-1) end
    if isprime5(i+1) then puts(i+1) end
  end
end
```

こんどは 10 秒間で 1,400,000 くらいまで調べられました。コンピュータが高速だといっても、大量に計算する場合にはやはり工夫する価値はあるわけです。

演習 5 — 配列の生成

これは簡単にコードだけ示します。

```
Arran.new(10) do |i| 10-i end       #(a)
Arran.new(10) do |i| i%2 end        #(b)
Arran.new(10) do |i| (i-4).abs end  #(c)
Arran.new(10) do |i| if i>=5 then 0 else 1 end end  #(d)
```

(a)、(b) は簡単なクイズという感じですね。(c) は「`if i<5 then 4-i else i-4 end`」でもよいのですが、ここでは数値が持つ絶対値のメソッド「x.abs」を使ってみました。または、以前作った絶対値のメソッドを呼び出しても構いません。(d) は if で枝分かれするのが一番素直ですね。

演習 6 — 配列の演習

演習 7 も擬似コード略して Ruby のコードだけ記します。まず最大。

```
def arraymax(a)
  max = a[0]
  a.each do |x|
    if x > max then max = x end
  end
  return max
end
```

このような形で配列を使う場合は、ふつうは「とりあえず max に最初の値を入れておき、より大きい値が出てきたら入れ換える」方法になります。each は配列の各要素を順に取り出して来るメソッドでした。

次は最大の値が何番目に出てくるかなので、普通の計数ループにします。また、「何番目か」も変数に記録し、最大を更新した時にに同時に更新します。

```
def arraymaxno(a)
  max = a[0]
  pos = 0
  a.each_index do |i|
    if a[i] > max then max = a[i]; pos = i end
  end
  return pos
end
```

配列の各添字を列挙するには `a.length.times` を使えばよいのですが、ここに示したように配列のメソッド `a.each_index` を使うこともできます。

最大を 1 箇所だけ記録するのは変数でもできましたが、最大が複数あった時にその位置を全部打ち出すのには、(1) まず最大を求め、(2) その最大と等しいものがあったら位置を打ち出す、という形で 2 回ループを使う必要があります。

```
def arraymaxno2(a)
  max = a[0]
  a.each_index do |i|
    if a[i] > max then max = a[i] end
  end
  a.each_index do |i|
```

```
    if a[i] == max then puts(i) end
  end
end
```

平均より大きい値を打ち出すのもこれと同様です。

```
def arrayavglarger(a)
  sum = 0.0
  a.each do |x| sum = sum + x end
  avg = sum / a.length
  a.each do |x|
    if x > avg then puts(x) end
  end
end
```

ところで、sum の初期値を 0.0 にしていることに注意。こうすれば、sum の内容は常に実数なので、最後の割り算も実数の割り算になります。ただの 0 だと、配列の中身もすべて整数のときは切捨て除算になってしまって、平均の計算が正しくできません。

演習 7 — 配列を使った素数列挙

まず最初は、これまでに見つかった素数を配列に覚えておく方法です。[3]

```
def isprime8(a, n)
  limit = Math.sqrt(n).to_i
  a.each do |i|
    if i >= limit then a.push(n); return true end
    if n % i == 0 then return false end
  end
  a.push(n); return true
end
def primes8(n)
  a = []
  2.step(n-1) do |i|
    if isprime8(a, i) then puts(i) end
  end
end
```

素数判定メソッドは素数の入った配列を受け取り、そこから順に素数を調べて候補の数に対して割り切れるかどうか調べていきます。ただし、候補の数の平方根まで来たらそれ以上やっても割り切れないことが分かるので「素数である」という答えを返します。なお、素数だった時は後に備えて配列にその素数を追加しておきます。この方法だと、「2 や 3 の倍数を除外」などのワザを使っていないのにもかかわらず、手元のマシンで 10 秒間で 1600,000 くらいまでの素数が調べられましした。

ただし、このあたりをやっていると分かりますが (もっと早く気づいた人もいることでしょう)、実は数値を表示するという処理にもかなり手間が掛かっています。計測するという観点からは、表示を省略して内部のチェックだけの時間を計った方がいいでしょう。でも、速さの違いを実感していただくには、画面に出力が出た方が分かりやすいので、課題としては画面に出力するようにしてあります。

もう 1 つ (エラトステネスのふるい) はこれまでと大幅に違う方法です。

[3] 配列のメソッド push は配列の末尾に新たな値を追加します。

```
def primes9(n)
  a = Array.new(n, true);
  2.step(n-1) do |i|
    if a[i] then
      puts(i)
      i.step(n-1, i) do |k| a[k] = false end
    end
  end
end
```

まず最初に、添字が 0〜$N-1$ の配列 a を作り、各要素の値は true としておきます。次に、2 から始めて各候補の数値 i について、a[i] が true ならそれは素数なので打ち出すとともに、その素数の倍数 k について a[k] を false にします。そうすると、調べて行ってまだ true の要素が素数として次々に拾われていくわけです。

この方法は非常に高速で、10 秒間で 3,000,000 以上の素数がチェックできました。最初の素朴版が千のレベル、こちらが百万のレベルだから、比べると 1000 倍!!!も速くなったわけです。日常的なものの世界では「1000 倍の差」というのはなかなかありません。我々の歩く速度がおよそ 4km/h ですが、4,000km/h というのはジェット機でもだめでロケットの速度ですね。これに対し、プログラムの動作速度については簡単に「ものすごい差」が生まれてしまう世界なわけです。

3　手続き/関数と抽象化

3.1　手続き/関数が持つ意味

手続きないし**サブルーチン**とは、ひとまとまりの動作に名前をつけ、他の箇所からの**呼び出し**により実行できるようなもののことです。

多くのプログラミング言語では、手続きから値を返すことができ、「ある決まった手順で値を計算するもの」とも捉えられます。このため、手続きのことを**関数** function と呼ぶ言語もあります。そして既に学んできたように、Ruby では**メソッド**が手続きに相当します。

また、既にたくさん使ってきたように、手続き呼び出し時に**パラメタ**を渡すことで、その渡した値に応じた動作や処理を行わせることができます。

たとえば前節では「整数 n が素数かどうかを調べる」というメソッドを作り、「素数を列挙する」メソッドからはそれを呼び出していました。そのコードを少し手直して再掲します。

```
def isprime(n)
  2.step(Math.sqrt(n)) do |i|
    if n % i == 0 then return false end
  end
  return true
end
def primes(n)
  2.step(n-1) do |i|
    if isprime(i) then puts(i) end
  end
end
```

2 つのメソッドに分けると、何がよいのでしょうか? それは、2 番目のメソッド中で「もし i が素数なら」とひとことで書けるようになることです。

別の例として「n 未満の数で、2 つ違いの素数になっているものを打ち出す」という作業を考えてみます。これも上のメソッドがあれば、次のように書けます。

```
def adjacentprimes(n)
  2.step(n-3) do |i|
   if isprime(i) && isprime(i+2) then puts "#{i} #{i+2}" end
  end
end
```

つまり「もし i が素数で、かつ、i+2 が素数 なら」とひとことで言えます。中では複雑な計算が必要な手順であったとしても、まとめて名前をつけることによって、必要なら何箇所からでも呼び出せ、コードも分かりやすくなるわけです。これをもっと一般的に言うと、手続きによって**抽象化**が行える、ということになります。

抽象化とは、不要な細部を省いて問題の検討に必要なことがらだけを残すことです。たとえば、「n が素数かどうか」を調べる方法が一度分かれば、あとはそれを参照すればよいのであって、その中でどのように処理しているかは「不要な細部」として見ないで済むことが利点なのです。

3.2 手続き/関数と副作用

関数という言葉は数学でも使われますが、数学で言う関数は「入力空間ないし定義域から出力空間ないし値域への写像」であって、同じ入力 (パラメタ) を与えた場合は同じ結果を返します。

たとえば $f(x) = x^2$ であれば、$f(2)$ の値は 4 であり、計算するたびに違うということはあり得ません。ですから、関数の値を 1 回計算して取っておき、2 回目は取っておいた値を利用するのでも、2 回とも計算するのとで、結果は一緒です。

これに対し、プログラムにおける関数は「単なる計算手順」ですから、その計算のやり方によっては、毎回違う値を返すこともありますし、どこかに観測できる変化を及ぼすこともあります。これを一般に**副作用** (side effect) と呼びます。

たとえば一番簡単な例として、puts は呼び出すたびに画面に文字が出力されますから、1 回呼び出すのと 2 回呼び出すのでは結果が違います。つまり、**入出力** (input/output — キーボードや画面やファイルなどとの間でのデータのやりとり) は副作用の形で扱われます。また、関数や手続きの中で外部の (関数や手続きの外で定義された) 変数を書き換える場合も副作用になります。

これまで使って来た変数は**局所変数** (local variable) と呼ばれ、そのメソッドが実行されている間だけ存在していて、実行が終わると消滅します (メソッドのパラメタも局所変数の一種と考えられます)。これに対し、プログラムの実行中ずっと存在し続け、さまざまなメソッド中から参照できる変数を**広域変数** (global variable) と呼びます。Ruby では先頭に$のついた名前の変数が広域変数です。広域変数は通常、複数のメソッド呼び出しをまたがって値を共有するのに使います。たとえば、次に示すようなやり方でいくつもの値を合計することを考えます。

```
irb> sum 1.5  ←次々に指定した値の
=> 1.5        ←合計が返される
irb> sum 2
=> 3.5
irb> sum 0.8
=> 4.3
irb> reset    ←ご破算もできる
=> 0
irb> sum 2
=> 2
irb> sum 0.7
=> 2.7
```

これを実現するためには、メソッド sum と reset を作りますが、これらの間で (および複数の sum 呼び出し間で) 値を保持するのに広域変数を使うわけです。

```
$x = 0
def sum(v)
  $x = $x + v; return $x
end
def reset
  $x = 0
end
```

この場合、sum や reset は広域変数$x を変更するという形の副作用を持っています。手続きが副作用を持つのは、広域変数に対する書き換えだけとは限りません。たとえば、配列をパラメタとして受け取って、その配列の内容を書き換えた場合、その変更を配列を渡した側にも影響します。このようなものも副作用になります。

3.3 例題: RPN 電卓

上述の sum と reset では合計という簡単な計算しかできませんでしたが、もう少し込み入った計算もできる仕組みとして、**逆ポーランド記法** (RPN、Reverse Polish Notation) 電卓というのを作ってみます。私たちが普段書いている数式の書き方は**中置記法** (infix notation) と呼び、演算子が被演算子の間に書かれますね。

$$8 + 5 \times 3 \rightarrow 23$$
$$(8 + 5) \times 3 \rightarrow 39$$

この方法は私たちが慣れてはいますが、「演算子の強さ (乗除算を優先)」とか「かっこの中を優先」などの規則があり、実は複雑です。これに対し、(1) 演算子は被演算子の後に書き、(2) 被演算子は演算子のできるだけ近くにある「残っている値」とする、という規則で書くのが RPN です。上の 2 つの例を RPN で書くと次のようになります。[4]

$$8\ 5\ 3\ \times\ +\ \rightarrow\ 8\ 15\ +\ \rightarrow\ 23$$
$$8\ 5\ +\ 3\ \times\ \rightarrow\ 13\ 3\ \times\ \rightarrow\ 39$$

上の例からも分かるように、RPN を使って式を記述する場合は、かっこが不要です。

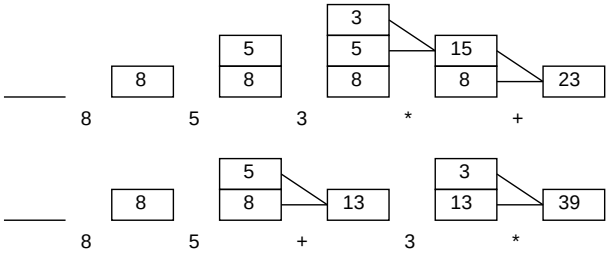

図 1: RPN 電卓による計算

RPN を使った計算は、図 1 のように値の並びを内部で保持し、数値が出て来た時には値を並びの末尾 (上が末尾です) につけ加え、演算子が出て来た時には最後の 2 つを取って演算し、代わりに結果を最後につけ加えるようにします。そうすると、式の最後まで来た時に並びに残っている値が結果となります。実は Mac の「電卓」は「Command-R」で RPN モードにできるので、少し計算してみると様子が分かります。

さて、先の合計と同様、この RPN 電卓を Ruby で実現してみます。数値の入力は「e」というメソッドで実行し、演算は「add」「mul」をとりあえず用意しました。動かした様子を見ましょう。

[4]演算子を「後に」置くことから、**後置記法** (postfix notation) とも呼びます。

```
irb> e 8
=> [8]
irb> e 5
=> [8, 5]
irb> e 3
=> [8, 5, 3]
irb> mul
=> [8, 15]
irb> add
=> [23]
irb> clear
=> []
irb> e 8
=> [8]
irb> e 5
=> [8, 5]
irb> add
=> [13]
irb> e 3
=> [13, 3]
irb> mul
=> [39]
```

ではプログラムですが、並びとしてはもちろん配列を使用します。配列には末尾に値を追加するメソッド「a.push(値)」と末尾から結果を取り除いて返すメソッド「a.pop」が用意されているので好都合です。

```
$vals = []
def e(x)
  $vals.push(x); return $vals
end
def add
  x = $vals.pop; $vals.push($vals.pop + x); return $vals
end
def mul
  x = $vals.pop; $vals.push($vals.pop * x); return $vals
end
def clear
 $vals = []; return $vals
end
```

演習 1 「合計を求める」例題をそのまま打ち込んで動かしなさい。動いたら、さらに次の機能を実現するメソッドを追加しなさい。

a. 加える代わりに指定した値を引く機能 dec(x)。
b. うっかり間違って reset した時にそれを取り消せる機能 undo(undo の undo はできなくてもよいが、できるようにしてもよい)。
c. これまでに加えた (そして引いた) 値の一覧を表示した上で合計を表示する機能 list(reset はできた方がよい。reset の undo もできるとなおよい)。

演習 2 「RPN 電卓」の例題をそのまま打ち込んで動かしなさい。動いたら、さらに次の機能を実現するメソッドを追加しなさい。

a. 加算と乗算に加えて減算 (sub)、除算 (div)、剰余 (mod) を追加。
b. 現在の演算結果の符号を反転する操作 `inv`。たとえば「1 2 add inv → -3」となる。
c. 最後の結果とその 1 つ前の結果を交換する操作 `exch`。たとえば「1 3 exch sub → 2」となる。
d. ご破産の機能 `clear` と、開始またはご破産から現在までの操作をすべて横に並べて (つまり RPN で) 表示する機能 `show`。[5]
e. その他、RPN 電卓にあったらよいと思う任意の機能。

4 再帰呼び出し

4.1 再帰手続き・再帰関数の考え方

関数や手続きの興味深い用法として、ある関数の中から直接または間接に自分自身を呼び出す、というものがあります。これを**再帰** (recursion) と呼びます。たとえば、前章でやった内容から、正の整数 x、y について、その最大公約数は次のように定義できます。

$$gcd(x,y) = \begin{cases} x & (x = y) \\ gcd(x-y,\ y) & (x > y) \\ gcd(x,\ y-x) & (x < y) \end{cases}$$

これにそのまま従って Ruby のメソッドを書くことができます。

```
def gcd(x, y)
  if x == y
    return x
  elsif x > y
    return gcd(x-y, y)
  else
    return gcd(x, y-x)
  end
end
```

プログラムそのものは大変分かりやすいですが、なぜ「堂々めぐり」にならずに計算が終わるのでしょうか。それは、図 2 を見れば分かります。

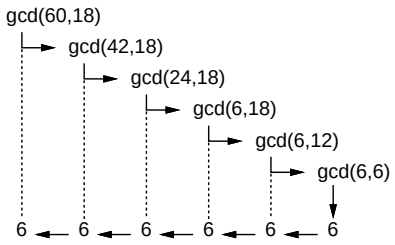

図 2: 再帰関数による最大公約数の計算

再帰関数 (再帰手続き) を作る時は、必ず次の原則に従います。

- 問題の「簡単な場合」は、すぐに答えを返す (上の例では $x = y$ の場合)。

[5] show を実現するためには、すべての演算にその操作内容を記録するコードを追加する必要がある。

- それ以外は問題を「少し簡単な問題に変形した上で」自分自身を呼び出す (上の例では、少し小さい数の最大公約数問題に変形)。

これがうまくできていれば、堂々めぐりにならずに正しく実行できるわけです。

演習 3 上の例題をそのまま打ち込んで動かせ。うまく動いたら、次のような再帰的定義に従った計算を再帰関数として書いて動かせ。また、典型的な実行の様子を表す、図 2 のような図を描いてみよ。

a. 階乗の計算。

$$fact(n) = \begin{cases} 1 & (n = 0) \\ n \times fact(n-1) & (otherwise) \end{cases}$$

b. フィボナッチ数。

$$fib(n) = \begin{cases} 1 & (n = 0 \ or \ n = 1) \\ fib(n-1) + fib(n-2) & (otherwise) \end{cases}$$

c. 組み合わせの数の計算。

$$comb(n, r) = \begin{cases} 1 & (r = 0 \ or \ r = n) \\ comb(n-1,\ r) + comb(n-1,\ r-1) & (otherwise) \end{cases}$$

d. 正の整数 n の 2 進表現。[6]

$$binary(n) = \begin{cases} \text{“0”} & (n = 0) \\ \text{“1”} & (n = 1) \\ binary(n \div 2) + \text{“0”} & (n \text{ が } 2 \text{ 以上の偶数}) \\ binary(n \div 2) + \text{“1”} & (n \text{ が } 2 \text{ 以上の奇数}) \end{cases}$$

4.2 再帰呼び出しの興味深い特性

再帰呼び出しの興味深い特性として、「現在実行しているコードと、再帰的に呼び出した自分とは、動作は同一だが (同じプログラムだから当然!)、人格としては別人」だということがあります。たとえば ${}_nC_r$ の計算の様子を図 3 に示します。${}_5C_3$ を計算するとして、その「私」は「手下」として ${}_4C_2$ を計算する人と ${}_4C_3$ を計算する人に作業を依頼します。これらの「人」はデータ (n とか r) は「私」とは違っているので別人ですが、動作は「私」と同じわけです。

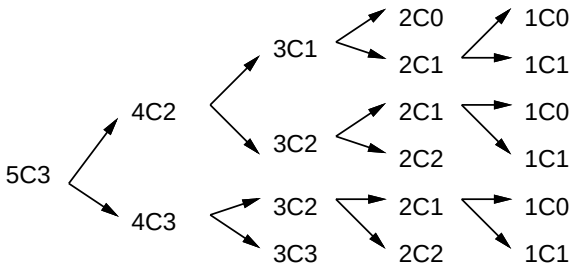

図 3: 再帰関数による組合せの数の計算

このような別人格を利用すると、興味深い処理が可能になります。たとえば、1〜3 を打ち出す場合、次のように 1 重ループを使えばできますね。

[6] この場合、関数の返す値は文字列であることと、+ は文字列の連結演算、÷は整数の除算 (切捨て除算) を表していることに注意してください。Ruby では整数どうしの「/」は自動的に切捨て除算になるのでしたね。

```
1.step(3) do |i| puts(i) end
```

では、「1〜3 が 2 つ並んだ全ての組合せ」だと…ループを 2 重にします (to_s は数値を文字列に変換するメソッドで、文字列どうしの＋は「連結」になります)。

```
1.step(3) do |i| 1.step(3) do |j| puts(i.to_s + j.to_s) end end
```

では「3 つ」たど 3 重…一般に n を指定して「1〜3 が n 並んだ全ての組合せ」が作れるでしょうか? プログラムでループを n 個書く方法では、プログラムを生成しない限り無理そうですよね? ところが、次のようにすればできるのです。

```
def nest3(n, s)  # 呼び方: nest3(3, "") ←空文字列を渡す
  if n <= 0 then
    puts(s)
  else
    1.step(3) do |i| nest3(n-1, s + i.to_s) end
  end
end
```

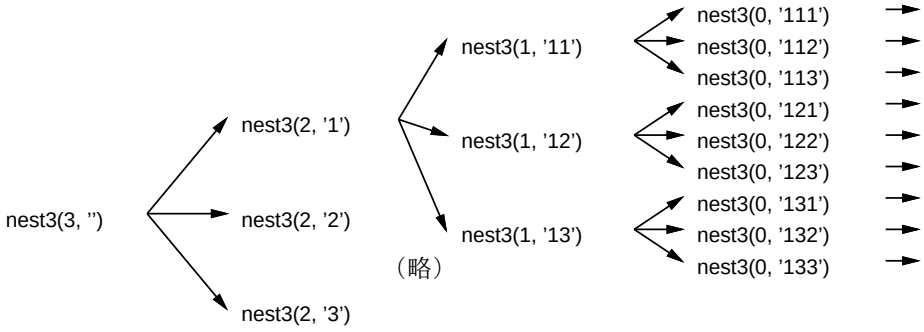

図 4: 1〜3 が N 個並んだすべての場合を出力

つまり、それぞれの「私」は自分の担当として 1〜3 を順番に生成し、「親の私」から渡された文字列にそれをくっつけ、「子の私」を呼び出します。入れ子になる (内側の) ループはその「子の私」の中で実行されるわけです。そして 0 の場合は…「文字列を打ち出す」のが仕事になります。この呼び出しの様子を図 4 に示します。

演習 4 この例題では結果に同じ数字が何回も出て来ていたが、1 つの数字は 1 回しか使わないようにすると**順列** (permutaiton) を生成したことになるので、やってみよう。できれば、配列を渡すとその要素のすべての順列を次々に生成するのがよい。

ヒント: 渡された配列から 1 つ列に追加したら、その要素は配列に無いことにすればよい (たとえばそこに nil を入れるなどして)。子供の処理が終わったら元に戻すことを忘れないように。

演習 5 与えられた配列の全ての並び替えを生成できるということは、その中から昇順に並んだものを選ぶことで、元の配列を昇順に整列するアルゴリズムができることになる。実際にそのようなプログラムを作ってみよ。また、この方法の弱点を検討し、できれば改良する方法についても検討せよ。

演習 6 自分の名前のローマ字表記を与えると、そのアナグラム (さまざまな順で文字を入れ替えたもの) を表示するプログラムを作りなさい。ただしローマ字として成立しないものは表示しないように工夫すること。

演習 7 再帰を利用して自分の興味のあるプログラムを作りなさい。

本日の課題 4A

「演習 1」または「演習 3」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. 手続き/関数/広域変数について学びましたが、納得しましたか。

Q2. 再帰的な呼び出しについてはどうですか。

Q3. 本日の全体的な感想と今後の要望をお書きください。

次回までの課題 4B

「演習 1」〜「演習 7」の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 手続きが使いこなせるようになりましたか。

Q2. 再帰的な呼び出しについてはどうですか。

Q3. 課題に対する感想と今後の要望をお書きください。