プログラミングプロジェクト＃2 – 手続き、枝分かれ

久野　靖 (電気通信大学)

2021.3.19

前回は始めてのプログラミングなので一直線のコードで計算するだけでしたが、今回はいよいよ手続きと制御構造の枝分かれをやります。今回の目標は次の通り。

- 手続きに分けた (抽象化を含む) プログラムを書けるようになる。
- 枝分かれ含むアルゴリズムやプログラムについて考えられるようになる。

まず前回の演習問題から抜粋して解説します (自分で課題をやってから読むことを勧めます)。

1　前回演習問題の解説

1.1　演習 3a — 四則演算を試す

演習 3a は和の計算でした。メソッド内の計算式を取り替えればいいだけなので簡単です。

```
def add(x, y)
  return x + y
end
```

動かしているところも見てみましょう。

```
irb> add 3.5, 6.8
=> 10.3
```

四則つまり和、差、商、積の場合も上と同じにやればいいわけですが、和、差、商、積のために 4 つメソッドを作る代わりに 1 つで済ませるという方法を考えてみましょう (半分くらいは新しい内容の紹介を兼ねています)。まず先に説明したように、メソッドの最後に値を返す代わりに、puts などで順次画面に書き出す方法があります。

```
def shisoku0(x, y)
  puts(x+y)
  puts(x-y)
  puts(x*y)
  puts(x/y)
end
```

動かしているところは次のとおり。

```
irb> shisoku0 3.3, 4.7
8.0
-1.4
15.51
0.702127659574468
=> nil
```

なるほど4つの値が順次打ち出され、最後に `shisoku0` の結果としては **nil**(何もないことを示す値) が返されています。

上の方法だと「1つの結果が返る」のでないのがちょっと、という気がするかもしれません。そこで次に、1つの文字列を返し、その中に4つの数値が埋め込まれている、というふうにしてみましょう。

Rubyでは**文字列** (string — 文字が並んだデータ) は「`'...'`」または「`"..."`」のように**シングルクォート**または**ダブルクォート**で囲んで表しますが、ダブルクォートのほうは内部に色々なものを埋め込む機能がついています。[1] 具体的には、文字列の中に「`#{...}`」という形のものがあると、中カッコ内の式を**評価** (evaluation — 値を計算すること) して、結果をそこに埋め込んでくれます。

これを利用した「四則演算」のメソッドを示します。

```
def shisoku1(x, y)
  return "#{x+y} #{x-y} #{x*y} #{x/y}"
end
```

実行しているところは次のとおり。

```
irb> shisoku1 3.3, 4.7
=> "8.0 -1.4 15.51 0.702127659574468"
```

確かに、1つの文字列中に4つの数値が埋め込まれています。もう1つ、Rubyなど多くの言語では値の並んだものを**配列** (array) という機能で扱います。Rubyでは `[...]` の中に値をカンマで区切って並べることで配列を直接書けるので、これを使って4つの数値をまとめて返すことができます。[2]

```
def shisoku2(x, y)
  return [x+y, x-y, x*y, x/y]
end
```

実行しているところは次のとおり。

```
irb> shisoku2 3.3, 4.7
=> [8.0, -1.4, 15.51, 0.702127659574468]
```

ここでは文字列の場合とあまり変わらない感じがするかもしれませんが、配列では返された値の中から「0番目」「1番目」など番号を指定して特定の要素を取り出せるので、より便利に使えます。

1.2 演習3b — 剰余演算

演習3bは剰余演算「`%`」を試すというものでした。演算子を取り換えるだけなので、プログラムは簡単ですね。

```
def jouyo(x, y)
  return x % y
end
```

実行してみましょう。

```
irb> jouyo 8, 5
=> 3
irb> jouyo 20, 5
=> 0
irb> jouyo -8, 5
```

[1] ダブルクォートは埋め込み機能等のために**特殊文字** (special character — 英数字以外の文字) を様々に解釈します。そのようなことをせずに文字列をそのまま表示させたい場合はシングルクォートを使ってください。

[2] `return` の後だと囲んでいる `[ ]` を省略できますが、場所によっては省略できないので常に書くことを薦めます。

```
=> 2
irb> jouyo -21, 5
=> 4
```

マイナスの時も試しましたか? 「ここでマイナスだとどうだろう」と気付くようになってください。

それで、マイナスの時も剰余は負にならず、つまり「5 間隔」というのがマイナスまでずっと続いている、というふうに考えるのでしょうね。では、割る数がマイナスだったらどうでしょうか?

```
irb> jouyo 8, -5
=> -2
irb> jouyo -8, -5
=> -3
```

つまり剰余の符号は割る数の符号と一致するようになっているわけです。[3]

演習 3c — 円錐の体積

演習 3c は円錐の体積でした。底面の半径 r、高さ h として、まず円錐の底面の面積は πr^2。体積はこれに高さを掛けて 3 で割ればできます。

```
def cornvol(r, h)
  return (r**2*3.1416*h) / 3.0
end
```

ちなみに「**」はべき乗の演算子です。もちろん 2 乗は「r*r」と書いても構いません。

```
irb> cornvol 3.0, 4.0
=> 37.6992
```

ところで「円周率が 3.1416 というのは不正確だ」と思う人もいそうですね。しかし、コンピュータ上の計算は「電卓での計算」と同様、有限の桁数でしか行えないのであり、自分で必要と思う適当な桁数を決めてその範囲でやるしかないので、有効数字 5 桁くらいでと思うならこれでよいわけです。[4]

1.3 演習 3d — 平方根

平方根は Math.sqrt(x) で計算できるので、とりあえずそれを出力します。ここでは puts を使って複数の値についてまとめて出力しましたが、1 個ずつ手で値を与えて試してももちろん構いません。

```
def sqrts1
  puts(Math.sqrt(2))
  puts(Math.sqrt(3))
  puts(Math.sqrt(5))
end
```

実行してみます。

```
irb> sqrts1
1.4142135623730951
1.7320508075688772
2.23606797749979
=> nil
```

これらは無理数ですから無限少数になりますが、コンピュータ上では有限の桁数で計算されます。ですから、コンピュータ上の実数計算は「一定の精度までの近似値で」計算されるものと思ってください。具体的な精度は、上に表示されているように 16〜17 桁程度ということになります。

[3]剰余演算の振舞いは、プログラミング言語によって多少の違いが見られる箇所です。

[4]3.141592653589793 くらいまでは扱える精度があるので、この定数をいちいち書くのは嫌だという人のために Math::PI と書いてもよいようになっています。同様に、自然対数の底 e は Math::E で表せます。

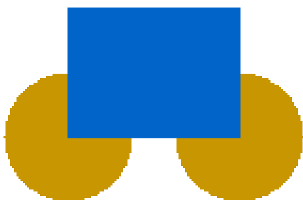
1.4 演習 4 — 画像を生成する

この課題は例題に円を追加して「車」にする、というものでした。ついでに色も変えています。

```
$img = Array.new(200) do Array.new(300) do [255,255,255] end end
def pset(x, y, r, g, b)
  if 0 <= x && x < 300 && 0 <= y && y < 200
    $img[y][x][0] = r; $img[y][x][1] = g; $img[y][x][2] = b
  end
end
def writeimage(name)
  open(name, "wb") do |f|
    f.puts("P6"); f.puts("300 200"); f.puts("255")
    $img.each do|a| a.each do |p| f.write(p.pack("ccc")) end end
  end
end
def fillrect(x0, y0, w, h, r, g, b)
  (y0-h/2).step(y0+h/2) do |y|
    (x0-w/2).step(x0+w/2) do |x| pset(x, y, r, g, b) end
  end
end
def fillcircle(x0, y0, rad, r, g, b)
  (y0-rad).step(y0+rad) do |y|
    (x0-rad).step(x0+rad) do |x|
      if (x-x0)**2 + (y-y0)**2 <= rad**2 then pset(x, y, r, g, b) end
    end
  end
end

def prac4
  fillcircle(60, 90, 30, 200, 150, 0)
  fillcircle(140, 90, 30, 200, 150, 0)
  fillrect(100, 60, 80, 60, 0, 100, 200)
  writeimage("prac4.ppm")
end
```

後から描いたものが上に来るので、`fillcircle` を 2 回呼んでタイヤを 2 個描いたあとで `fillrect` を呼んでボディを描きます。最後に `writeimage` で絵を出力します。



1.5　演習 5 — さまざまな絵

この課題は自分で色々試せばよいですが、解答例も挙げます (下請けのメソッド等は省略)。

```
def prac5a
  fillrect(100, 100, 80, 60, 0, 255, 0);
  fillcircle(100, 100, 20, 255, 0, 0);
  writeimage("pict1.ppm")
end

def prac5b
  fillrect(20, 20, 20, 20, 255, 0, 0);
  fillrect(60, 20, 20, 20, 255, 0, 0);
  fillrect(40, 40, 20, 20, 255, 0, 0);
  fillrect(20, 60, 20, 20, 255, 0, 0);
  fillrect(60, 60, 20, 20, 255, 0, 0);
  writeimage("pict1.ppm")
end

def prac5c
  fillcircle(100, 100, 60, 255, 0, 0);
  fillcircle(100, 100, 40, 0, 255, 0);
  fillcircle(100, 100, 20, 255, 0, 0);
  writeimage("pict1.ppm")
end

def prac5e
  fillcircle(40, 40, 20, 255, 0, 0);
  fillcircle(100, 40, 20, 255, 0, 0);
  fillcircle(70, 60, 30, 255, 0, 0);
  writeimage("pict1.ppm")
end
```

2　手続き/関数と抽象化

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**」と呼ぶ言語もあります。Ruby では**メソッド**が手続きに相当します。

すでに見てきたように、メソッド呼び出し時に**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られます。これを利用して、プログラムをいくつかの「かたまり」に分けることを考えます。

図 1 を見てください。これは車の設計図で、演習で作ったのとおなじです。ただし、今度は原点がボディの中心になっていて、ボディの長さは 6u、高さは 4u、タイヤの半径は 2u と言う風に設計してあります。u ってなーに? ただの変数で (unit のつもり)、実際に描くところで 10 とか 15 とか具体的な値を入れることで、さまざまな大きさの車が描けるようにしました。x や y も実際に描くときに好きな場所を指定できます。

ではプログラムを見てみます。ライブラリは同じですが、その後に車を描く `car` と、それを呼び出して車を 3 台描き絵を出力する `mipicture2` が置かれています。

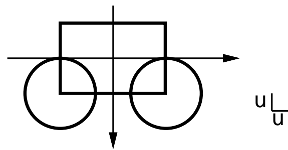


図 1: 車の設計図

```
$img = Array.new(200) do Array.new(300) do [255,255,255] end end
def pset(x, y, r, g, b)
  if 0 <= x && x < 300 && 0 <= y && y < 200
    $img[y][x][0] = r; $img[y][x][1] = g; $img[y][x][2] = b
  end
end
def writeimage(name)
  open(name, "wb") do |f|
    f.puts("P6"); f.puts("300 200"); f.puts("255")
    $img.each do|a| a.each do |p| f.write(p.pack("ccc")) end end
  end
end
def fillrect(x0, y0, w, h, r, g, b)
  (y0-h/2).step(y0+h/2) do |y|
    (x0-w/2).step(x0+w/2) do |x| pset(x, y, r, g, b) end
  end
end
def fillcircle(x0, y0, rad, r, g, b)
  (y0-rad).step(y0+rad) do |y|
    (x0-rad).step(x0+rad) do |x|
      if (x-x0)**2 + (y-y0)**2 <= rad**2 then pset(x, y, r, g, b) end
    end
  end
end

def car(u, x, y, r1, g1, b1, r2, g2, b2)
  fillcircle(x-3*u, y+2*u, 2*u, r2, g2, b2)
  fillcircle(x+3*u, y+2*u, 2*u, r2, g2, b2)
  fillrect(x, y, 6*u, 4*u, r1, g1, b1)
end
def mypicture2
  car(10, 220, 40, 20, 40, 80, 250, 200, 200)
  car(12, 180, 80, 200, 250, 40, 0, 150, 100)
  car(15, 80, 110, 150, 150, 150, 255, 0, 0)
  writeimage('pict2.ppm')
end
```

car は先程説明した u、x、y、そしてボディの色とタイヤの色をパラメタとして受け取ります。mypicture2 はこれらのパラメタを指定して car を 3 回呼び出し、3 つの車を描きます。絵を図 2 に示します。

このように、一度中身を作ってしまえば、以後ば中身を気にせずに使えるようになることを「**抽象化**」と呼びます。抽象化は複雑なプログラムを分かりやすくする重要な道具となります。

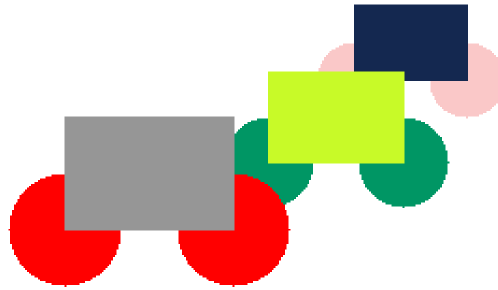


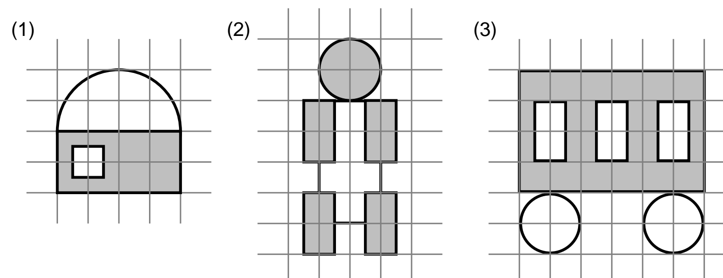
図 2: 車が 3 つある絵

なお、車が 3 つでなく 1 つでも、このようにプログラムの上で「車」などを表すメソッドが分かれている方が、抽象化のおかげで、プログラムとしての見通しはよくなります。

そして、複雑な絵であってもその絵の「部品」ごとにそれを描くメソッドを作り、また複雑な部品であればその部品の「下請け部品」のメソッドを作り、のようにしていくことで、込み入った絵が描けるわけです。

演習 1 車が 3 つの例題をまずそのまま動かしてみなさい。車の位置、大きさ、色は適宜変更してみること。納得したら、次のことをしてみなさい。

a. 下の設計図のような部品を描くメソッドを作り、それが複数回含むような絵を生成してみなさい。絵の適当な箇所の座標 (x,y) とあみ目の大きさ u を指定し、色は 2 色の RGB 値 r1,b1,g1 と r2,b2,g2 を指定するようにしなさい。



b. 好きな国の「旗」を描くメソッドを作って絵を生成してみなさい。とりあえず円と長方形で済むものに限る必要があります。

c. 複数の部品が入った好きな絵を設計し作ってみなさい。

3 基本的な制御構造

3.1 実行の流れと制御構造

ここまでに出てきたアルゴリズムおよびプログラムはすべて「1 本道」、つまり上から順番に実行して一番下まで来たらおしまい、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くると、実行の流れをさまざまに切り換えていくことが必要になります。この、実行の流れ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。

制御構造の表現方法の 1 つに**流れ図** (flowchart) があります。流れ図では、図 3 にあるような「処理を示す箱」「条件による枝分かれを示す箱」などを矢線でつなげて実行の流れを表現します。流れ図は一見分かりやすそうですが、作成に手間が掛かる、場所をとる、ごちゃごちゃの構造を作ってしまいがち、という弱点のため、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ます)。

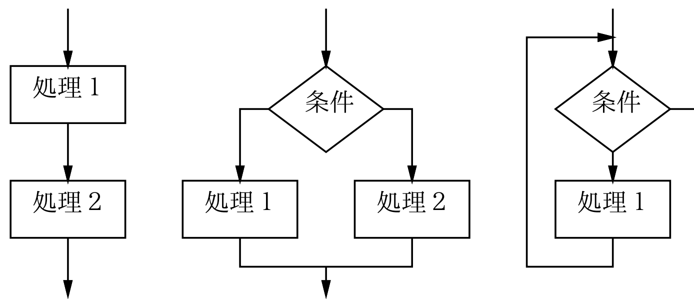


図 3: 3 つの基本的な制御構造

アルゴリズムを記述する時にはさまざまな実行の流れを組み立てますが、今日ではそれらの実行の流れは、図 3 に示す 3 つの制御構造を組み合せる形で作り出していくのが普通です。

- **順次実行**ないし**連接** — 動作を順番に実行していくこと。
- **枝分かれ**ないし**分岐** — 条件に応じて 2 群の動作のうちから一方を選んで実行すること。
- **繰り返し**ないし**反復** — 条件が成り立つ限り一群の動作を繰り返し実行すること。[5]

なぜこの 3 つが基かというと、「どんなにごちゃごちゃの流れ図でも、それと同等の動作を、この 3 つの組み合わせによって作り出せる」という定理があり、そのためにこの 3 つさえあればどのような処理の流れでも表現可能だからです。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、以下では残りの 2 つの制御構造をコード上で表現するやり方と、それらを組み合わせてアルゴリズムを組み立てていくやり方を学びます。

3.2 枝分かれと if 文

上述のように、枝分かれとは、条件に応じて 2 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すものとします (「動作 2」が不要なら「そうでなければ」も書かなくてもかまいません)。

- もし 〜 ならば、
-     動作 1。
- そうでなければ、
-     動作 2。
- 枝分かれ終わり。

Ruby ではこれを **if 文** (if statement) を使って表します (右側は「動作 2」のない場合です)。

```
if 条件 then                if 条件 then
  ... 動作 1 ...              ... 動作 1 ...
else                        end
  ... 動作 2 ...
end
```

`then` は Ruby では省略できますが、ただし「動作 1」を条件と同じ行に書く場合には省略できません。

「条件」については、当面は次の形のものがあると思っておいてください。

- 比較演算 — 「`x > 10`」等、2 値を比べるもの。**比較演算子**としては次の 6 種類がある。[6]

[5] 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。
[6] Ruby では「`!`」は「否定」を表すのに使っています。階乗の記号ではないので注意してください。

>	>=	<	<=	==	!=
より大	以上	より小	以下	等しい	等しくない

- 条件の組み合わせとして次が使える。[7] これらを「()」でくくったり複数組み合わせられる。

かつ (両方が成立)	または (最低限一方が成立)	否定 (〜でない)
条件1 && 条件2	条件1 || 条件2	! 条件1

- 条件は「はい」「いいえ」を表す結果となる。これらを直接書くこともできる。

「はい」を表す定数	「いいえ」を表す定数
true	false

では例として、「入力 x の絶対値を計算する」ことを考えます。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 Ruby にしてみましょう。

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

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ときには系統的に洩れなく試してみることが大切です)。

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

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。

[7] Ruby ではさらに演算子として and、or、not も使えますが、結合の強さが記号版と違っていて混乱しやすいので、本資料では使っていません。

- 　そうでなければ、
- 　　x を返す。
- 　枝分かれ終わり。

Ruby 版は次のようになります。

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

先のとどちらが好みでしょうか? 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。

Ruby プログラムも示します (`if` を 1 行に書いてみました。このような時は `then` が必須)。

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

3 つのプログラムについて、あなた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、場面ごとに何がよいかが違ってきますし、人によっても基準が違うところがあります。ですから、皆様がこれからプログラミングを学習するに当たっては、自分なりの「よいと思う書き方」を発見していく、という側面が大いにあります。そのことを心に留めておいてください。

演習 2 絶対値計算プログラムの好きなバージョンを打ち込んで動かせ。動いたら、枝分かれを用いて、次の動作をする Ruby プログラムを作成せよ。

a. 2 つの異なる実数 a、b を受け取り、より大きいほうを返す。

b. 3 つの異なる実数 a、b、c を受け取り、最大のものを返す。(やる気があったら 4 つでやってみてもよいでしょう。)

c. 実数を 1 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。

3.3　枝分かれのある絵

それでは次は、絵と枝分かれを組み合わせてみます。具体的には、さっきの車を描くメソッドをすこし手直しして、「場合によって少し違った形の車」が描けるようにします。

```
# ライブラリをここに

def car2(u, x, y, r1, g1, b1, r2, g2, b2)
  fillcircle(x-3*u, y+2*u, 2*u, r2, g2, b2)
  fillcircle(x+3*u, y+2*u, 2*u, r2, g2, b2)
  fillrect(x, y, 6*u, 4*u, r1, g1, b1)
  if u > 10
    fillrect(x, y-2*u, 4*u, 2*u, r2, g2, b2)
  end
end
def car3(u, x, y, r1, g1, b1, r2, g2, b2, rect)
  if rect
    fillrect(x-3*u, y+2*u, 3*u, 3*u, r2, g2, b2)
    fillrect(x+3*u, y+2*u, 3*u, 3*u, r2, g2, b2)
  else
    fillcircle(x-3*u, y+2*u, 2*u, r2, g2, b2)
    fillcircle(x+3*u, y+2*u, 2*u, r2, g2, b2)
  end
  fillrect(x, y, 6*u, 4*u, r1, g1, b1)
end
def mypicture3
  car2(10, 240, 30, 20, 40, 80, 250, 200, 200)
  car2(11, 180, 50, 200, 250, 40, 0, 150, 100)
  car2(14, 80, 50, 150, 150, 150, 255, 0, 0)
  car3(10, 240, 140, 20, 40, 80, 250, 200, 200, true)
  car3(11, 180, 150, 200, 250, 40, 0, 150, 100, false)
  car3(14, 80, 140, 150, 150, 150, 255, 0, 0, true)
  writeimage('pict3.ppm')
end
```

car2 は、u が「10 より大きい時だけ」キャビンを描くようになっています。やり方は、キャビンを描く部分を if 文にいれ、u > 10 の条件を満たす時だけ実行するようにしています。car3 は、10 番目のパラメタとして true/false (はい/いいえ) を受け取り、「はい」の時はタイヤが円でなく正方形になります。今度はタイヤを描く部分を if 文に入れていますが、ただし条件 rect が「はい」なら四角、「いいえ」なら円を描くように分岐しています。このように、枝分かれをつかうと「ちょっとだけ機能が変化する」手続きを作ることができるのです。

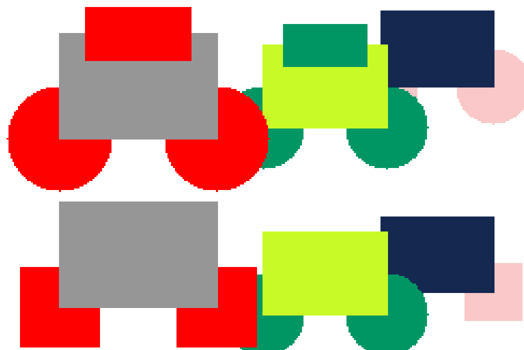


図 4: 枝分かれのある車の手続き

演習 3 枝分かれのある絵の例題を打ち込んで動かせ。動いたら、枝分かれを用いて、次の図形を描く Ruby プログラムを作成せよ。

a. u の値が 10 以上なら、タイヤが 3 つついた車に変化した旗が描かれる。そうでない場合はこれまでと同じ。
b. 10 番目のパラメタとして `true/false` (はい/いいえ) を受け取り、「はい」の時はこれまでと同じだが、「いいえ」のときはボディの色とタイヤの色がいれかわる。
c. これまでに作成した絵を描くプログラムのどれかをもとにして、枝分かれを用いて「場合によって少し違った形」が描けるようにしてみよ。

演習 4 複数の部品が集まった絵をつくりなさい。少なくとも 1 つは枝分かれを用いていること。

本日の課題 2A

「演習 1」「演習 2」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. プログラムを打ち込んで動かすのに慣れましたか?

Q2. 自分にとって次の「難しいポイント」は何だと思いますか?

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 2B

「演習 3」「演習 4」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. 手続きの「抽象化」の役割が納得できましたか。

Q2. 繰り返しは納得できましたか。

Q3. 課題に対する感想と今後の要望をお書きください。