基礎プログラミング+演習 ＃10 – 動的データ構造+情報隠蔽

久野 靖 (電気通信大学)

2017.12.11

今回は新たな題材として動的データ構造を扱いますが、これをオブジェクト指向による情報隠蔽の具体例としても捉えます。

- 動的データ構造の概念と考え方、単連結リストの操作
- 情報隠蔽 (カプセル化) の考え方

1 前回の演習問題の解説

1.1 演習 1 — クラス定義の練習

この演習はメソッドとインスタンス変数が追加できればよいということで、コードだけ掲載しておきましょう。

```
class Dog
  def initialize(name)
    @name = name; @speed = 0.0; @count = 3
  end
  def talk
    puts('my name is ' + @name)
  end
  def addspeed(d)
    @speed = @speed + d
    puts('speed = ' + @speed.to_s)
  end
  def setcount(c)
    @count = c
  end
  def bark
    @count.times do puts('Wan! ') end
  end
end
```

1.2 演習 2 — 簡単なクラスを書いてみる

この演習はクラスの書き方の練習みたいなものなので、見ていただけば十分でしょう。Memory2 はちょっと頭を使う必要がありますかね。

```
class Memory
  def initialize()
    @mem = nil
  end
```

```
  def put(x)
    @mem = x
  end
  def get()
    return @mem
  end
end

class Memory2
  def initialize()
    @mem2 = @mem1 = nil
  end
  def put(x)
    @mem2 = @mem1; @mem1 = x
  end
  def get()
    x = @mem1; @mem1 = @mem2; @mem2 = nil; return x
  end
end

class Concat
  def initialize
    @str = ""
  end
  def add(s)
    @str = @str + s
  end
  def get()
    return @str
  end
  def reset()
    @str = ""
  end
end
```

1.3 演習 3 — 有理数クラス

この演習も Ratio クラスのメソッドを「同様に」増やせばよいだけなので、難しくはありません。追加するメソッドだけ掲載します。

```
  def -(r)
    return Ratio.new(@a*r.getDivisor-r.getDividend*@b,
                     @b*r.getDivisor)
  end
  def *(r)
    return Ratio.new(@a*r.getDividend, @b*r.getDivisor)
  end
  def /(r)
```

```
    return Ratio.new(@a*r.getDivisor, @b*r.getDividend)
  end
```

要は、引き算は足し算と同様、乗算は分母どうし掛け、除算はひっくり返して掛けるということですね。

1.4 演習 4 — 複素数クラス

複素数も 2 つの値 (実部、虚部) の組なので、有理数によく似ています。ただし個々の値として整数でなく実数を使います。演算はちょっと面倒 (とくに除算) ですが、作る時に約分とか分母が 0 とか考えなくてよい部分は簡単になります。

```
class Comp
  def initialize(r = 1.0, i = 0.0)
    @re = r; @im = i
  end
  def get_re
    return @re
   end
  def get_im
     return @im
  end

  def to_s
    if @im < 0 then
      return "#{@re}#{@im}i"
    else
      return "#{@re}+#{@im}i"
    end
  end
  def +(r)
    return Comp.new(@re + r.get_re, @im + r.get_im)
  end
  def -(r)
    return Comp.new(@re - r.get_re, @im - r.get_im)
  end
  def *(r)
    return Comp.new(@re*r.get_re - @im*r.get_im,
                    @im*r.get_re + @re*r.get_im)
  end
   def /(r)
    norm = (r.get_re**2 + r.get_im**2).to_f
    return Comp.new((@re*r.get_re + @im*r.get_im) / norm,
                    (@im*r.get_re - @re*r.get_im) / norm)
  end
end
```

to_s でヘンなことをやっているのは、「$a + bi$」の形で表示させようとした時、虚数部が負の場合には「$a - bi$」にしたいためです。

2　動的データ構造/再帰的データ構造

2.1　動的データ構造とその特徴 exam

データ構造 (data structure) とは「プログラムが扱うデータのかたち」を言います。ここでは、プログラムの実行につれて構造を自在に変化させられる、**動的**データ構造 (dynamic data structure) について学びます。これに対し、ここまでに扱ってきたプログラムのように、それぞれの変数に決まった形のデータが入っていて、全体の形が変わらないものを**静的**データ構造 (static data structure) と呼びます。一般に、静的は「プログラム記述時に決まる」、動的は「プログラム実行時に決まる」という意味で使われます。

動的データ構造は、プログラム言語が持つ「データのありかを指す」機能を用いて作られます。Ruby では、複合型 (配列、レコード等) や一般のオブジェクトの値は実際は、それらのデータやオブジェクトのありかを指す**参照**になっているので、これを利用します。既に忘れている人がいるかもしれませんが、**レコード**とは複数のフィールドが集まったデータであり、Ruby では `Struct.new` においてフィールド名を表す記号を必要なだけ指定して定義するのでしたね。

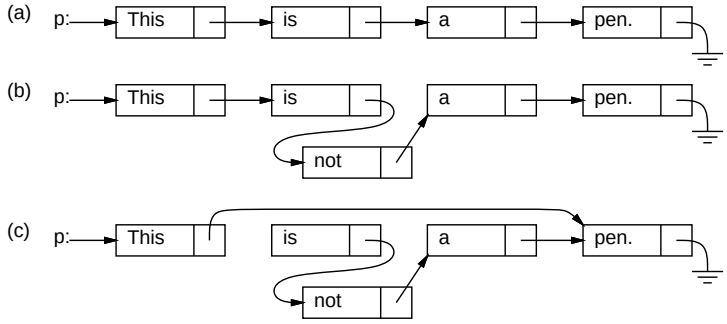

図 1: 単連結リストの動的データ構造

たとえば、次のレコードを見てみましょう。

```
Cell = Struct.new(:data, :next)
```

これは 2 つのフィールド `data` と `next` を持つ `Cell` という名前のレコードを定義していますが、ここで各セルのフィールド `next` に「次」のセルへの参照を入れることで、「数珠つなぎ」の動的データ構造を作ることができます (図 1(a))。[1] このような「数珠つなぎ」の構造のことを**単連結リスト** (single linked list) ないし単リストと呼びます。

この `Cell` の使い方は、各 `Cell` の `next` がまた `Cell` になっていて、自分の中に自分が入っているように思えます。これは再帰関数と同様で、このようにデータ型 (構造) の中に自分自身と同じデータ型 (構造) への参照を含むものを**再帰的データ構造** (recursive data structure) と呼びます。実際には自分自身が入っているわけではなく図 1 のように「同種のデータへの参照」が入っているだけですから、何ら問題はありません。

一番最後のところ (アース記号で表している) は「何も入っていない」という印である **nil** が入っています。このあたりも、「簡単な場合は自分自身を呼ばずにすぐ値が決まる」再帰関数とちょっと似ていますね。

ところで、動的データ構造だと何がよいのでしょうか? たとえば、図 1(a) で途中に単語「not」を入れたくなったとします。文字列の配列であれば、途中に挿入するためには後ろの要素を 1 個ずつずらして空いた場所に入れる必要があります。しかし、単連結リストでは、矢線 (参照) を (b) のようにつけ替えるだけで挿入ができてしまうのです。逆に、数単語削除したいような場合も、(c) のように

[1] 本当はフィールド `data` も文字列オブジェクトを参照しているので、文字列を箱の外に描いて矢線で指させるべきなのですが、ごちゃごちゃして見づらくなるのでここでは箱の中に直接描いていています。

参照のつけ換えで行えます。このように、動的データ構造は柔軟な構造の変更が行えるという特徴を持っています。

参照のつけ替えは、具体的にはどうすればいいのでしょう? 参照というのは要するに「場所を示す値」なので、その値をコピーすることは「矢印の根本を別の場所にコピーする (矢印自体も 2 本になる) と考えればいいのです。たとえば、図 2 では、`p.next.next` というのは B の箱の `next` フィールド、`p.next` は A の箱の `next` フィールドなので、「`p.next = p.next.next`」で B の箱を迂回して A の箱の `next` フィールドに C の箱を指させることになります。参照を入れてある単独変数 (この例では `p`) などでも同様です。

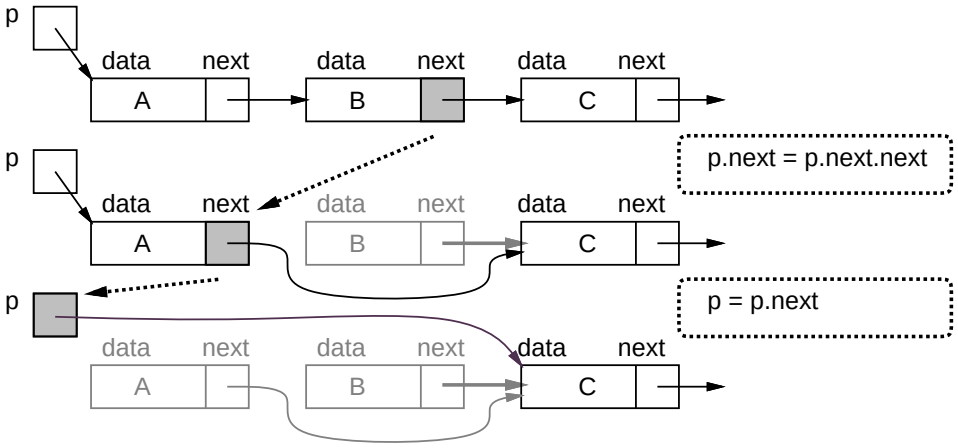

図 2: 参照のつけ替え

ときに、図 1 や 2 で使わなくなった箱はどうなるのでしょう? Ruby では、使われなくなったオブジェクトの領域は**ごみ集め** (garbage collection) と呼ばれる機構によって自動的に回収され、再利用されます。「使っている」かどうかは、どれかの変数からたどれるかどうかで決まります。たとえば図 1 の場合は、先頭のセルを変数 `p` が指していて、ここからたどれるセルは「使っている」と見なされるのです。

2.2 再帰的メソッドと単連結リスト exam

再帰的データ構造は名前から分かるように再帰的メソッドと相性がよいです。たとえば、単連結リストに何個セルがつながっているかを調べるとします。それには次のメソッドでできます。

```
def listlen(p)
  if p == nil
    return 0
  else
    return 1 + listlen(p.next)
  end
end
```

どういうことでしょうか。渡された `p` が `nil` なら、長さは 0 ですね。そしてそうでなければ、自分の次のセルからの長さを (再帰呼び出しによって) 調べ、それに 1 を足せばよい、でしょう? 実際にやってみましょう。最初に `Cell` を定義し、そしてつながった 3 つのセルを作ります。「C が入った next が nil のセルを」「B が入ったセルの next にして」「さらにそれを A が入ったセルの next にする」と考えてください。

```
irb> Cell = Struct.new(:data, :next)  ← Cell を定義
```

```
=> Cell
irb> p = Cell.new('A', Cell.new('B', Cell.new('C', nil))) ←セル 3 個のリスト
=> ...
irb> listlen p
=> 3                ←確かに長さ 3
```

たどりながら出力する方が動作が分かりやすいかも知れません。各データを順に出力してみましょう。

```
def printlist(p)
  if p != nil then puts(p.data); printlist(p.next) end
end
```

今度は nil のときは打ち出すものがないので何もしないため、より短くなっています。そうでない場合は、まず打ち出し、そして再帰呼び出しで残りを打ち出してもらえばいいわけです。実行例を示します。

```
irb> printlist p
A
B
C
=> nil
irb> printlistrev p
C
B
A
=> nil
```

しかし printlistrevって? それは次のものです。まず残りを打ち出してもらい、最後に自分の担当を打ち出す、というふうに逆にしただけでこうなるわけです。

```
def printlistrev(p)
  if p != nil then printlistrev(p.next); puts(p.data) end
end
```

演習 1 上の例 (メソッド 3 つ) をそのまま打ち込んで動かせ。動いたら、次のメソッドを作ってみよ。

a. data に数値が入っている単連結リストに対して、その数値の合計を求める listsum。
b. 各セルの data(文字列) を連結した 1 つの文字列を返す listcat。
c. 上と同様だがただし逆順に連結する listcatrev。
d. printlist と同様だが、1 行目は 1 回、2 行目は 2 回、3 行目は 4 回、…と倍倍で打ち出す回数が増える printmany。打ち出す順番は任意の順番で (ごちゃまぜで) よい。
e. listsum と同様だが、ただし奇数番目のセルの値だけ合計する listoddsum。
   ヒント: これまでのように自分自身を再帰呼び出しする代わりに、奇数番目用 (加算をする) は偶数番目用 (加算しない) を呼び、偶数番目用は奇数番目用を呼ぶ、というふうに交互にやります。これを**相互再帰**と呼びます。
f. 単リストの並び順を逆向きにした単リストを返す listrev。これには元のリストを変更してしまう版と変更しない版とが考えられるが、どちらでもよいことにする。

3 情報隠蔽

3.1 例題: 単連結リストを使ったエディタ

ではここで、単連結リストを使った例題として、簡単な**テキストエディタ** (text editor) を作ってみましょう。「簡単」なので、編集に使うコマンドは次のものしかありません。

- 「i 文字列」 — 文字列を新しい行として現在位置の直前に挿入する。
- 「d」 — 現在位置の行を削除する。
- 「t」 — 先頭行を表示し、そこを現在位置とする。
- 「p」 — 現在位置の内容を表示する。
- 「n」または改行 — 現在位置を次の行へ移しその行を表示する。
- 「q」 — 終了する。

実際にこれを使っている様子を示します (すごく面倒そうですが、実際にこういうプログラムを使ってファイルの編集をしていた時代は実在しました)。

```
>iThis is a pen.       ←挿入
>iThis is not a book. ←挿入
>iHow are you?         ←挿入
>t                     ←先頭へ
  This is a pen.
>                      ←次の行
  This is not a book.
>                      ←次の行
  How are you?
>                      ←次の行
  EOF                  ←おしまい
>t                     ←再度先頭へ
  This is a pen.
>iI am a boy.          ←挿入
>                      ←次の行
  This is not a book.
>iWho are you?         ←挿入

>t                     ←再度先頭へ行き全部見る
  I am a boy.
>
  This is a pen.
>
  Who are you?
>
  This is not a book.
>
  How are you?
>
 EOF
>q                     ←おしまい
```

これをこれから実現してみましょう。

3.2 エディタバッファ

以下では、単連結リストのデータ構造を先頭や現在位置などの各変数も含めてクラスとしてパッケージします。

```
class Buffer
  Cell = Struct.new(:data, :next)
  def initialize
    @tail = @cur = Cell.new("EOF", nil)
    @head = @prev = Cell.new("", @cur)
  end
  def atend
    return @cur == @tail
  end
  def top
    @prev = @head; @cur = @head.next
  end
  def forward
    if atend then return end
    @prev = @cur; @cur = @cur.next
  end
  def insert(s)
    @prev.next = Cell.new(s, @cur); @prev = @prev.next
  end
  def print
    puts("  " + @cur.data)
  end
end
```

レコード定義もクラスに入れることにしました。また、「1 つの値を複数箇所に代入する」のに=を連続して書いてみました。もちろん、2 つの代入に分けても一向に構いません。

このクラスでは、単連結リストのセルを上記の Cell レコードであらわし、これを指すための変数として次の 4 つを使っています。

- @head — 一番先頭に「ダミーの」セルを置き、そのセルを常にこの変数で指しておく (ダミーがあると、先頭行を削除するのを特別扱いしないで済ませられるため、プログラムの作成が楽になります)。
- @cur — 「現在行」のセルを指しておく。
- @prev — 「現在行の 1 つ前」のセルを指しておく (挿入や削除の時にこの変数があるとコードを書くのが楽です)。
- @tail — 一番最後にも「ダミーの」セルを置き、そのセルをこの変数で指しておく (表示することがあるので内容は「EOF」(end of file) としてあります)。

initialize では 2 つのダミーセルと上記 4 変数を用意します。head の次が tail であるように Cell.new にパラメタを渡していることにも注意。

では次に、メソッドを見てみましょう。図 3 に、適当なバッファの状態で操作を行った例を示します (最後の delete は課題の参考用です)。atend は現在行が末尾にあるか (@tail と等しいか) を調べます。top は@prev と@cur を先頭に設定します。forward は@prev と@cur を 1 つ先に進めますが、現在行が@tail の時は「果て」なので何もしません。print は現在行の文字列を表示します。insert

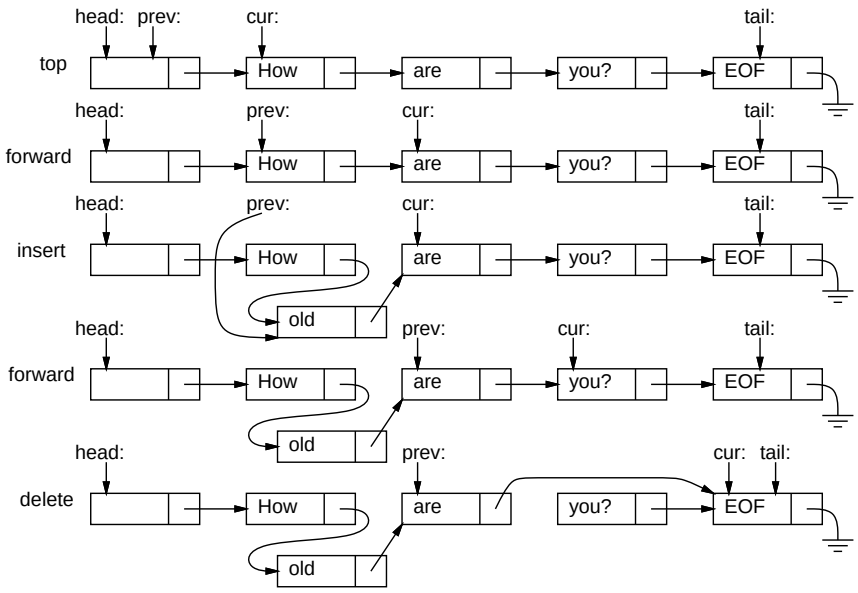

図 3: エディタバッファに対する操作

は新しいセルが@prev となり、元の@prev のセルの次が新しいセル、新しい@prev の次が@cur のセルとなります。これを動かした様子を見てみましょう。[2]

```
irb> e = Buffer.new
=> ...
irb> e.insert('abc')
=> ...
irb> e.insert('def')
=> ...
irb> e.insert('ghi')
=> ...
irb> e.top
=> nil
irb> e.print
   abc
=> ...
irb> e.forward
=> ...
irb> e.print
   def
=> nil
```

確かに文字列が順序どおり挿入でき、それをたどることができています。

このクラスは「行の挿入や削除が自在にできる機能を持ったオブジェクト」を作り出しています。内部では込み入ったデータ構造を管理していますが、その様子はクラスの外側からは見えません。このように内部構造を外から見せないようにして外部に機能を提供することを**情報隠蔽** (information hiding) または**カプセル化** (encapsulation) と呼びます。その利点は、内部のデータにアクセスするのはそのクラスのコードだけなので、データ構造の整合性が保て、またプログラムの正しさの確信が持ちやすいことです。

[2] irb の結果表示はうるさいので省略しています。

また、カプセル化を通用いて、操作だけが外から呼び出せ、それを用いて整合性のある汎用的な機能を提供するものを、**抽象データ型** (abstract data type, ADT) と呼びます。クラス方式のオブジェクト指向言語では、抽象データ型はクラスによって定義するのが自然です。前章に出てきた有理数クラスや複素数クラスも抽象データ型の例だといえます。

演習 2 図 4 は「How」という行と「are」という行の間に「old」という行を挿入する様子 (A → B)、および、「old」「are」「you?」という 3 行のうちから「are」を削除する様子 (B → C) を示しています。資料 (ないし同じ図を描き写したもの) の上に赤ペンで次のものを記入しなさい。

a. (A) の図の上に、(A) から (B) につながりが変化するための矢線のつけ替えを、(1)、(2) のようにつけ替えを行う順番つきで記入しなさい。ただし、矢印のつけ替えを行う時には、その出発点がどれかの変数そのものであるか、またはどれかの変数から矢線でたどれる箱であることが必要である。

b. (B) の図の上に、(B) から (C) につながりが変化するための矢線のつけ替えを、(1)、(2) のようにつけ替えを行う順番つきで記入しなさい。ただし、矢印のつけ替えを行う時には、その出発点がどれかの変数そのものであるか、またはどれかの変数から矢線でたどれる箱であることが必要である。

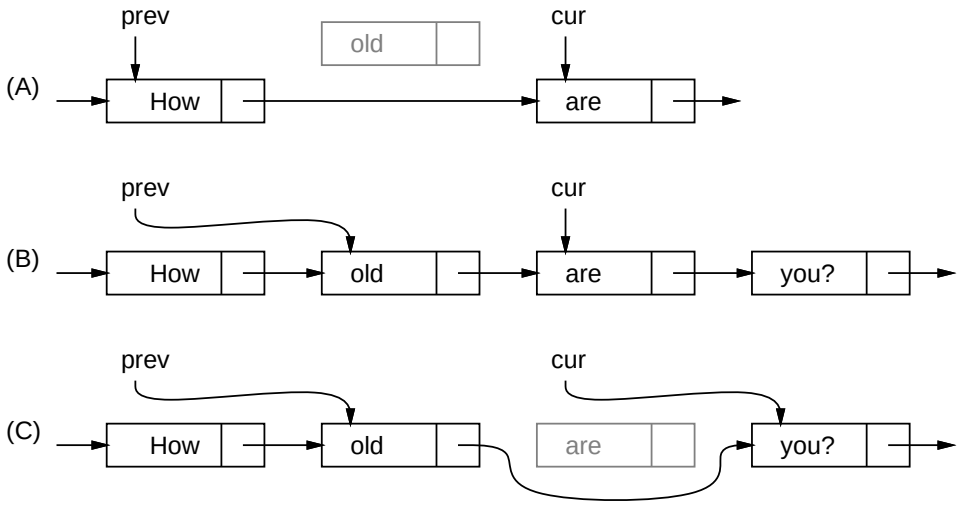

図 4: 挿入と削除のようす

演習 3 クラス `Buffer` を打ち込み、動作を確認せよ。動いたら、以下の操作 (メソッド) を追加してみよ。

a. 現在行を削除する (EOF 行は削除しないように注意…)

b. 現在行と次の行の順序を交換する (EOF は交換しないように…)

c. 1 つ前の行に戻る (実は大変かも)

d. すべての行の順番を逆順にする (かなり過激)

演習 4 単連結リストでは各セルが「次」の要素への参照だけを保持していたが、各セルが「次」と「前」2 つの参照を持つようなリストもある。これを**双連結リスト** (double linked list) と呼ぶ。編集バッファの双連結リスト版を作り、その得失を検討せよ。[3]

[3]双連結リストでは単連結リストでの「頭」と「最後」を 1 つで兼ねることもできます (無理に兼ねなくてもよい)。

3.3 エディタドライバ

バッファのメソッドを呼ぶだけでも編集はできますが、面倒です。先にお見せしたように「コマンド (+パラメタ)」ですらすら編集ができるように、エディタとして動作するコードも作ってみました。内容はとても簡単で、バッファを生成し、その後無限ループでプロンプトを出し、1 行読んでは先頭の 1 文字でどのコマンドを実行するか枝分かれします (コメントにしてあるのはあなたが作るか、後で機能を追加するためのものです)。

```
def edit
  e = Buffer.new
  while true do
    printf(">")
    line = gets; c = line[0..0]; s = line[1..-2]
    if c == "q"    then return
    elsif c == "t" then e.top; e.print
    elsif c == "p" then e.print
    elsif c == "i" then e.insert(s)
#   elsif c == "r" then e.read(s)
#   elsif c == "w" then e.save(s)
#   elsif c == "s" then e.subst(s); e.print
#   elsif c == "d" then e.delete
    else                e.forward; e.print
    end
  end
end
```

文字列の一部を取り出すには「[位置..位置]」という添字指定を使います。1 文字だけの場合でも文字列として取り出したい場合は位置を 2 つ指定するので、先頭の文字は `line[0..0]` で取り出しているわけです。

`i`(insert) コマンド等では、2 文字目から最後の文字の手前まで (最後の文字は改行文字) も必要なので、これも取り出しています。Ruby では文字列や配列中の位置として負の整数を指定すると末尾からの位置指定になります。

どのコマンドでもない場合 (や改行だけの場合) はいちばんよく使う「1 行進んで表示」にしました。

演習 5 エディタドライバを打ち込んで先のクラスと組み合わせて動作を確認せよ。動いたら以下のような改良を試みよ (クラス側を併せて改良しても、このメソッドだけを改良しても、どちらでも構いません。文字列を数値にする必要が生じたら、メソッド `to_i` を使ってください)。

a. 演習 3 で追加した機能が使えるようにコマンドを増やす。
b. 現在行の「次に」新しい行を追加するコマンド「a」を作る (追加した行が新たな現在行になるようにしてください)。
c. 現在行の内容をそっくり打ち直すコマンド「r」を作る。
d. 「g 行数」で指定した行へ行くコマンド「g」を作る。
e. コマンド「p」を「p 行数」でその行数ぶん打ち出すように改良 (その際、できれば現在位置は変更しないほうが望ましいです)。
f. その他、自分が使うのに便利だと思うコマンドを作る。

3.4 文字列置換とファイル入出力

せっかくエディタができたのに、行内の置き換えとかファイルの読み書きができないと実用になりませんから、これらを一応解説しておきます。

まず、行内の置き換は「s/α/β/」により現在行中の部分文字列αをβに置き換えるというコマンドにしました。エディタドライバからはバッファのメソッド subst を呼ぶだけとしたので、こちらの中身を示します。

```
def subst(str)
  if atend then return end
  a = str.split('/')
  @cur.data[Regexp.new(a[1])] = a[2]
end
```

文字列のメソッド split は、渡されたパラメタ「/」のところで文字列を分割した配列を返します。その 1 番目を Regexp(パターン) オブジェクトに変換して文字列に添字アクセスすると、そのパターンの箇所があれば、代入によりそこを別の文字列に置き換えられます。

ファイルの読み書きみは、＃5 で学んだ open でファイルを開き、読む場合は付属ブロック内でそのファイルの各行を insert、書く場合は逆にバッファの各行をファイルに puts で書き出します。

```
def read(file)
  open(file, "r") do |f|
    f.each do |s| insert(s) end
  end
end
def save(file)
  top
  open(file, "w") do |f|
    while not atend do f.puts(@cur.data); forward end
  end
end
```

演習 6 自作のエディタでどれか 1 課題ぶんの編集を行い、体験を述べよ。エディタの機能は何があれば必要十分か、使いやすさは何で決まるかについて考察すること。

演習 7 動的データ構造を活用した、何か面白いプログラムを作れ。面白さの定義は各自に任されます。

本日の課題 10A

「演習 1」「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. 動的データ構造とはどのようなものか理解しましたか。

Q2. 連結リストの操作ができるようになりましたか。

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 10B

「演習 1」「演習 3」〜「演習 7」の (小) 課題から選択して 1 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. 何らかの動的データ構造が扱えるようになりましたか。

Q2. 複雑な構造をクラスの中にパッケージ化する利点について納得しましたか。

Q3. 課題に対する感想と今後の要望をお書きください。