情報科学 2010　久野クラス　＃ 10

久野　靖*

2011.12.16

今回で年内最終回、来年は 3 回ありますが、出席を取るのは次回 (＃ 11) が最後、あとは出席自由です。あと少しですので、頑張ってください。今回では多くのアルゴリズムで汎用的に使われるデータ構造であるスタックとキューについて学び、またそれとも関連する話題として、状態空間の探索について取り上げます。

1　前回の演習問題解説

1.1　演習 4 — 字句解析の改良

説明の都合上、演習 4 から先に取り上げます。例題で示した字句解析器はプログラムは 1 行ですべての要素は 1 文字だけという簡易なものでしたが、「ふつうの」字句解析だとどうなるかを見てみましょう。具体的には次のように直しました (というより全部書き直したというべきですが):

- 入力はファイルから読み込み、複数行あってよい
- 空白や改行などは無視する
- 数値定数や名前は何文字あってもよい
- 制御構造等を表す特別な名前 (while、if 等) を扱う

前半の 2 つのために、Lexer オブジェクト中にファイルを保持して、次のトークンを取ろうとした時に行の終わりなら次の行を取るようにし、また空白等の時はその文字を飛ばすようにします。空白等を飛ばした結果行が終わりになることがあるのでそれをうまく扱う必要があるのに注意が必要です。

そして後半の 2 つのために、先頭が数字や英字の時は続く数字や英数字の列をまとめて探して取り出します。この時、メソッド peek が返すものは数の時は「'0'」、英字の時は「'a'」という「印」だけにして、実際の文字列は別のメソッド getstr で取得するように直しました。

名前が「if」、「while」、「read」、「print」の時は特別な印として「'I'」、「'W'」、「'R'」、「'P'」を返します。したがって、これらの名前は変数としては使えません。こうのような「予約された」名前のことを**予約語** (reserverd word) と言います。あと、「==」のような 2 文字演算子も使えるようにしてみました。ではコードを示しましょう:

```
class FileLexer
  def initialize(f)
    @file = open(f, 'r')
    @line = @str = ''; @pos = @lno = 0; @cur = '$'; self.fwd
  end
  def fwd()
    while @line != nil do
      if @pos >= @line.length
        @line = @file.gets; @pos = 0; @lno += 1; next
      end
      c = @line[@pos..@pos]; l = @line.length
      if c==' ' || c=="\n" || c=="\r" || c=="\t" then
```

*筑波大学大学院経営システム科学専攻

```
        @pos += 1; next
      elsif c>='0' && c<='9'
        p = @pos+1
        while p < l && @line[p..p]>='0' && @line[p..p]<'9' do
          p = p + 1
        end
        @cur = '0'; @str = @line[@pos..(p-1)]; @pos = p; return
      elsif c>='a' && c<='z' || c>='A' && c<='Z'
        p = @pos+1
        while p < l &&
             (@line[p..p]>='0' && @line[p..p]<='9' ||
              @line[p..p]>='a' && @line[p..p]<='z' ||
              @line[p..p]>='A' && @line[p..p]<='Z') do
          p = p + 1
        end
        @cur = 'a'; @str = @line[@pos..(p-1)]; @pos = p
        if @str == 'while' then @cur = 'W'
        elsif @str == 'if' then @cur = 'I'
        elsif @str == 'print' then @cur = 'P'
        elsif @str == 'read' then @cur = 'R'
        end
        return
      else
        @cur = c; @pos += 1
        if (c=='=' || c=='<' || c=='>' || c=='!') &&
            @pos < l && @line[@pos..@pos] == '=' then
           @cur = @cur + '='; @pos += 1
        end
        return
      end
    end
    @cur = '$'
  end
  def peek() return @cur end
  def getstr() return @str end
  def to_s() return "line #{@lno}: char #{@pos} in #{@line}" end
end
```

見て分かるとおり、ほとんどの仕事は fwd の中でやっています。@line は通常は行の文字列が入りますが、ファイルの終わりになったら nil になります。ファイルの終わりでないなら、行末だったら行を読み先頭へ行きます。next 文というのは、ループ内でループの先頭へ戻るための文です。

空白類を飛ばす場合も 1 文字飛ばしたら先頭へ戻ります。その先では、何らかの意味のある文字のところに来たので、それが数字なら数字列を取り出し、英字なら英字と数字の並びを取り出します。名前の場合は取り出した後でそれが予約語と一致したら名前の印を予約語の印にとり換えます。それ以外は基本的に 1 文字の演算子として扱いますが、ただし=!<>の 4 文字だけは次の文字が=だった場合に 2 文字の比較演算子として扱えるようにしています。

以下で動かしてみるプログラムはおなじみ素数の列挙です:

```
max = read; n = 2;
while(n <= max) {
  i = 2; sosu = 1;
  while(i < n) {
```

```
    if(n % i == 0) {sosu = 0};
    i = i + 1
  };
  if(sosu) print n;
  n = n + 1
}
```

これを新しい字句解析器で読んでみましょう:

```
irb> sc = FileLexer.new('test.prog')  ←作る
=> #<FileLexer:0x81dc114 @cur="a", @pos=3, @line="max = read;
n = 2;\n", @lno=1, @file=#<File:test.prog>, @str="max">
irb> sc.peek   ←最初は
=> "a"         ←名前
irb> sc.getstr ←その文字列は
=> "max"       ←「max」
irb> sc.fwd    ←次は
=> nil
irb> sc.peek
=> "="         ←「=」
irb> sc.fwd    ←次は
=> nil
irb> sc.peek
=> "R"         ←予約語「read」
```

確かに複数文字の名前や予約語が扱えていることが分かります。

1.2 演習 3 — 比較演算/制御構造/入出力の追加

動作の種別を増やすのは Node のサブクラスを増やすだけなので難しくありません。比較演算の例として Le(<=) と、あと目新しいものとして if、while、read、print のみ示します。read や print は数値を読み込んだり画面に出力したりします。

```
class Le < Node
  def initialize(l, r) super; @op = '<=' end
  def exec()
    if @left.exec <= @right.exec then
      return 1
    else
      return 0
    end
 end
end
# Lt(<), Gt(>), Ge(>=), Eq(==), Ne(!=) も同様
class If < Node
  def initialize(l, r) super; @op = 'I' end
  def exec() if @left.exec != 0 then @right.exec end end
end
class While < Node
  def initialize(l, r) super; @op = 'W' end
  def exec() while @left.exec != 0 do @right.exec end end
```

```
end
class Read < Node
  def initialize() super; @op = 'R' end
  def exec() print '> '; return gets.to_i end
end
class Print < Node
  def initialize(l) super; @op = 'P' end
  def exec() puts @left.exec end
end
```

1.3 演習 8 — 構文の改良

構文の改良といっても、文を増やしたりするのこれまでと同様にできます。面倒なのは、演算子の強さを分けることと、四則演算を左結合にすることですが、それには文法の書き直しが必要です。書き直した文法を示します:

$$
\begin{aligned}
&prog ::= stat \mid stat\ \texttt{;}\ prog\\
&stat ::= \{prog\} \mid \texttt{while}\ asn\ stat \mid \texttt{if}\ asn\ stat \mid \texttt{print}\ asn \mid asn\\
&asn ::= cmp \mid cmp\ \texttt{=}\ asn\\
&cmp ::= expr \mid expr\ \texttt{==}\ expr \mid expr\ \texttt{!=}\ expr\\
&\quad \mid expr\ \texttt{>}\ expr \mid expr\ \texttt{>=}\ expr \mid expr\ \texttt{<}\ expr \mid expr\ \texttt{<=}\ expr\\
&expr ::= term\ expr1\\
&expr1 ::= \epsilon \mid \texttt{+}\ term\ expr1 \mid \texttt{-}\ term\ expr1 \mid\\
&term ::= fact\ term1\\
&term1 ::= \epsilon \mid \texttt{*}\ fact\ term1 \mid \texttt{/}\ fact\ term1 \mid \texttt{\%}\ fact\ term1\\
&fact ::= \texttt{read} \mid identfier \mid number \mid \texttt{(}\ asn\ \texttt{)}
\end{aligned}
$$

read という予約語は文ではなく特別な因子になることに注意してください。たとえば「x = read + 1」のようなものも書けるわけです。ϵ というのは「空」に対応する規則で、たとえば $expr$ の先頭で $term$ を認識した次に+/-が来なければ、加減式はここで終わりなので、$expr1$ を空文字列に対応させるわけです。

ポイントの 1 つ目は、演算子の優先順位を分けるために、これまで $expr$ で全部の演算を一緒に扱っていたのを、asn(代入式) → cmp(比較式) → $expr$(加減式) → $term$(乗除式) のようにレベル分けしたことです。なぜこれでよいかというと、たとえば加減式の「+/-」で結ばれたそれぞれの部分式は乗徐式なので、その中には乗徐算が入るがそれ以外の演算は(かっこで囲んでない限り) 入らない、だから加減算より乗除算が必ず優先される、というふうになるわけです。

そしてポイントの 2 つ目は、加減式と乗除式を左結合にするために 2 つの規則に書き換えていることです。文法として素直に書くだけならば、次のようにすれば左結合になります:

$$expr ::= term \mid expr\ \texttt{+}\ term \mid expr\ \texttt{-}\ term$$

しかし、この文法では右辺の枝の中に先頭が $expr$ つまり定義しつつある記号と同じもので始まるものがあります。このような規則を**左再帰** (left recursive) な規則と呼びます。これを再帰下降解析器に直すと、expr の中で直ちに expr を呼ぶことになってしまうため、堂々めぐりで終わらなくなります。これを避けるために、先に挙げたように書き直しているわけです。

このほか、もう 1 つの注意点として、次のように先頭部分が重複している規則があります:

$$asn ::= cmp \mid cmp\ \texttt{=}\ asn$$

これも一見、どちらの枝へ行ってよいか分からなくて困るように思えますが、次のように同じ記号を「くくり出し」てまず処理し、その先でどちらの枝かを決めるようにすれば大丈夫です:

$$asn ::= cmp\ (\ \epsilon \mid \texttt{=}\ asn\ )$$

では、上の文法を再帰下降解析器に直したものを示します:

```
def prog(sc)
  s = stat(sc); c = sc.peek
  if c == '$' || c == '}' then return s
  elsif c == ';' then sc.fwd; return Seq.new(s, prog(sc))
  else puts('STAT:' + sc.to_s); return Noop.new
  end
end
def stat(sc)
  c = sc.peek
  if c == '{' then
    sc.fwd; p = prog(sc)
    if sc.peek != '}' then puts('NO_}:' + sc.to_s); return Noop.new end
    sc.fwd; return p
  elsif c == 'W' then sc.fwd; e = asn(sc); return While.new(e, stat(sc))
  elsif c == 'I' then sc.fwd; e = asn(sc); return If.new(e, stat(sc))
  elsif c == 'P' then sc.fwd; e = asn(sc); return Print.new(e)
  else return asn(sc)
  end
end
def asn(sc)
  e = cmp(sc); c = sc.peek
  if c == '=' then sc.fwd; return Assign.new(e, asn(sc)) end
  return e
end
def cmp(sc)
  e = expr(sc); c = sc.peek
  if c == '<' then sc.fwd; return Lt.new(e, cmp(sc))
  elsif c == '<=' then sc.fwd; return Le.new(e, cmp(sc))
  elsif c == '>' then sc.fwd; return Gt.new(e, cmp(sc))
  elsif c == '>=' then sc.fwd; return Ge.new(e, cmp(sc))
  elsif c == '==' then sc.fwd; return Eq.new(e, cmp(sc))
  elsif c == '!=' then sc.fwd; return Ne.new(e, cmp(sc))
  else return e
  end
end
def expr(sc)
  e = term(sc); return expr1(e, sc)
end
def expr1(e, sc)
  c = sc.peek
  if c == '+' then
    sc.fwd; e1 = term(sc); return expr1(Add.new(e, e1), sc)
  elsif c == '-' then
    sc.fwd; e1 = term(sc); return expr1(Sub.new(e, e1), sc)
  end
  return e
end
def term(sc)
  e = fact(sc); return term1(e, sc)
```

```
  end
  def term1(e, sc)
    c = sc.peek
    if c == '*' then
      sc.fwd; e1 = term(sc); return term1(Mul.new(e, e1), sc)
    elsif c == '/' then
      sc.fwd; e1 = term(sc); return term1(Div.new(e, e1), sc)
    end
    return e
  end
  def fact(sc)
    c = sc.peek; sc.fwd
    if c == 'a' then return Var.new(sc.getstr)
    elsif c == '0' then return Lit.new(sc.getstr.to_i)
    elsif c == '(' then
      e = asn(sc)
      if sc.peek != ')'
        puts('NO_):' + sc.to_s); return Noop.new
      end
      sc.fwd; return e
    elsif c == 'R' then return Read.new()
    else puts('FACTOR:' + sc.to_s); return Noop.new
    end
  end
```

最後にこれで演習 4 の解説のところで示した素数列挙を実行させてみましょう:

```
  irb> tree = prog(FileLexer.new('test.prog'))
  (表示は略)
  irb> tree.to_s
  => "((max=(R));((n=2);((n<=max)W((i=2);((sosu=1);
  (((i<=n)W((((n%i)<=0)I(sosu=0));(i=(i+1))));((sosuI(nP));(n=(n+1)))))))))"
  irb> tree.exec
  > 20   ←プログラム中の「read 式」による入力
  2
  3
  5
  7
  11
  13
  17
  19
  => nil
```

ちゃんと、素数列挙ができていることが分かります。

2 スタックとキュー

2.1 スタック

スタック (stack) とはコンピュータのアルゴリズムで多く使われる汎用の**抽象データ型**であり、「**LIFO**(last-in, first-out) の記憶領域」とも呼ばれます。つまり、スタックには多数のデータが入れられますが、取り出す時には (残っているものの中で) 一番最近に入れたものが取り出されてくるのです。これはちょうど、ものを上に積んでいって取り降ろす時の動作を同じなので、スタック (積む) と呼ばれるわけです。スタックの入れる/取る動作は伝統的に **push**/**pop** と呼ばれます (図 1 左)。

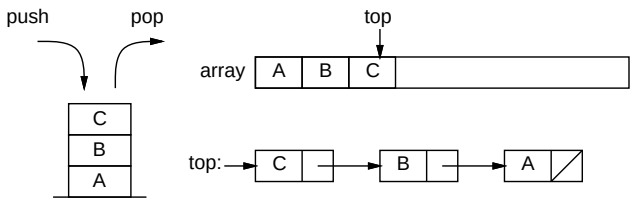

図 1: スタック

スタックの実現方法の 1 つは、配列を用意して、そこに順番に push された要素を詰めていくことです (図 1 右上)。先頭の位置は変数 `top` に覚えておき、push したら `top` を進め、pop したら 1 つ戻します。

この方法は簡潔ですが、多くのプログラミング言語では配列のサイズが作成時に指定した値から変えられないため、最大いくつの要素を格納するか予め決める必要があります。Ruby では…実は配列に `push`/`pop` というメソッドが用意されていてスタックの機能が実現済みです。しかし練習なので、ここでは固定サイズの配列でスタックを実現してみます。

スタックのもう 1 つの実現方法は、連結リストを用いて要素を覚えておく方法です (図 1 右下)。この場合、最大要素数を指定する必要はありません。

配列を使ったスタックの実現と、それを使って文字列を記憶してみるサンプルを示します:

```
class Stack1
  def initialize() @arr = Array.new(100); @ptr = -1 end
  def isempty() return @ptr < 0 end
  def push(x) @ptr = @ptr + 1; @arr[@ptr] = x end
  def pop() x = @arr[@ptr]; @ptr = @ptr - 1; return x end
end
```

動かす様子を見てみましょう:

```
irb> s = Stack1.new
...
irb> s.push(1)
=> 1
irb> s.push(2)
=> 2
irb> s.push(3)
=> 3
irb> s.isempty
=> false
irb> s.pop
=> 3
irb> s.pop
=> 2
irb> s.push(4)
```

```
=> 4
irb> s.push(5)
=> 5
irb> s.pop
=> 5
irb> s.pop
=> 4
irb> s.pop
=> 1
irb> s.isempty
=> true
```

2.2 キュー

キュー (queue) もスタックと類似した機能を提供しますが、(残っているものの中で) 一番先に入ったものが取り出される点が違います。このためキューのことを「**FIFO**(fist-in, first-out) のデータ構造」とも呼びます。そもそもキューとはおなじみ「行列」を意味する英語です (もちろん最初に並んだ人が最初にサービスしてもらえなかったら怒りますよね…)。

キューの入れる/取り出す操作は伝統的に **enq**/**deq** と呼ばれます (2 左)。実はフルスペルは enqueue と dequeue ですが、長いし発音が同じなので短く書くのが普通なわけです。

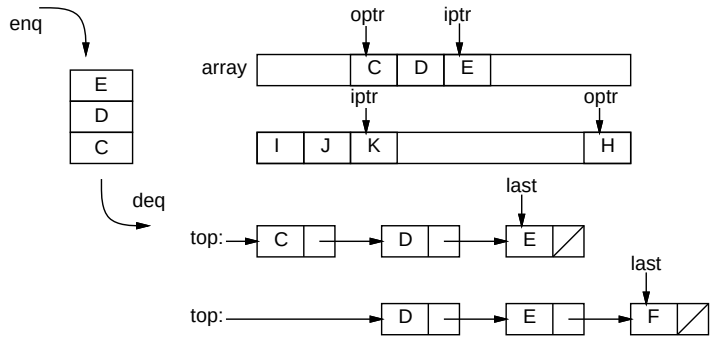

図 2: キュー

キューの実現も配列版と連結リスト版が考えられます。配列版では、`iptr` と `optr` という 2 つの変数で「入れる場所」と「取り出す場所」を覚えておきます (図 2 右上)。出し入れを繰り返すと、入っている場所が配列の端まで来るので、ぐるっと回って先頭に戻るようにします。最初と最後がくっついて輪になっているため、このようなデータ構造を**環状バッファ** (circular buffer)、**リングバッファ** (ring buffer) と呼ぶこともあります。

満杯になった/空っぽになったことを知るために、現在いくつ入っているかを別の変数で覚えておきます。別の方法として、`ipt` が `opt` に追い付きそうになったら満杯だと判断する方法もありますが、その場合は常に 1 箇所は空けておかないと空っぽと満杯の区別がつかなくなります。

連結リスト版は、リストの先端を `last` で指しておいて、そこに新しい要素を追加するようにします (図 2 右下)。リストが空っぽ (`top` が `null` の場合は `last` は意味を持たない (セルが 1 個もないのだから) ので、入れる時に特別扱いになります。

以下には配列版のキューのプログラムを示します (`test2` の使い方は先の例と同じ):

```
class Queue1
  def initialize()
    @arr = Array.new(100); @count = 0; @iptr = -1; @optr = 0
  end
  def isempty() return @count <= 0 end
  def enq(x)
```

```
      if @count + 1 >= @arr.length then return end
      @iptr = (@iptr+1)%@arr.length; @arr[@iptr] = x;
      @count = @count + 1
    end
    def deq()
      if @count <= 0 then return end
      x = @arr[@optr]; @count = @count - 1;
      @optr = (@optr+1)%@arr.length; return x
    end
  end
```

動かす様子を見てみましょう:

```
irb> q = Queue1.new
...
irb> q.enq(1)
=> 1
irb> q.enq(2)
=> 2
irb> q.isempty
=> false
irb> q.deq
=> 1
irb> q.enq(3)
=> 2
irb> q.enq(4)
=> 3
irb> q.deq
=> 2
irb> q.deq
=> 3
irb> q.deq
=> 4
irb> q.isempty
=> true
```

演習 1 スタックとキューのうち好きな方を打ち込んで動かせ。動いたら、連結リストを使ったスタックやキューの実現を作り、上の例で同じに動作することを確認せよ。

2.3 スタックとキューを使った構造のたどり

スタックやキューはどういう役に立つのでしょうか? それは、何かを入れて (覚えて) おいて、後で取り出して使うためです。当り前だと思うかもしれませんが、こういうことは結構あります。単独の変数に覚えておくのだと、1 つしか覚えておけません (2 つ目を入れると前に入っていたものは上書きされて失われます)。配列だと、「どこに」入れてあるかを覚えておかないと役に立ちません。ところが、スタックやキューでは「とにかく入れて」「とにかく取り出す」ことができるので使うのが楽であり、そして入れたものは必ずいつか出てくることが保証されます。

たとえば、前回の式木では exec や to_s で再帰を使ってたどりながら計算や文字列化を行っていましたが、これらを使わずにスタックやキューで式木をたどってみましょう。式木の構造は前章と同じとします。ただし to_s と exec は使いません。その代わり、外部から左や右の子と演算子をアクセスする必要があるので、getleft、getright、getop というメソッドを追加しました。[1]

[1] このような、インスタンス変数にアクセスすることを目的としたメソッドのことを**アクセサ**と呼びます。

```
class Node
  def initialize(l=nil, r=nil) @left = l; @right = r; @op = '?' end
  def to_s() return '(' + @left.to_s + @op.to_s + @right.to_s + ')' end
  def getleft() return @left end
  def getright() return @right end
  def getop() return @op end
end
```

スタックを使ってたどる例を見てみましょう。最初に根のノードをスタックに入れ、あとは「取り出して、処理し、子供があれば積む」わけです: [2]

```
def test1(tree)
  stk = Stack1.new; stk.push(tree)
  while !stk.isempty do
    node = stk.pop
    if node.is_a?(Var) || node.is_a?(Lit) then
      print(' ' + node.getleft.to_s)
    else
      print(' ' + node.getop)
      stk.push(node.getright); stk.push(node.getleft)
    end
  end
  puts
end
```

左と右の子をスタックに入れる時、右を先に入れるのは、取り出す時左から取り出されたほうが見やすいからです。実行のようすを見てみましょう:

```
irb> test1(Mul.new(Lit.new(2), Add.new(Var.new('x'), Lit.new(1))))
* 2 + x 1
=> nil
```

まず演算が表示され、続いてその演算の子供が左、右の順で表示されます。ではキューではどうでしょうか。キューを使った場合もたどるアルゴリズムは操作の名前が違うだけで前と同様です。

```
def test2(tree)
  que = Queue1.new; que.enq(tree)
  while !que.isempty do
    node = que.deq
    if node.is_a?(Var) || node.is_a?(Lit) then
      print(' ' + node.getleft.to_s)
    else
      print(' ' + node.getop)
      que.enq(node.getleft); que.enq(node.getright)
    end
  end
  puts
end
```

```
irb> test1(Mul.new(Lit.new(2), Add.new(Var.new('x'), Lit.new(3))))
* 2 + x 3
=> nil
```

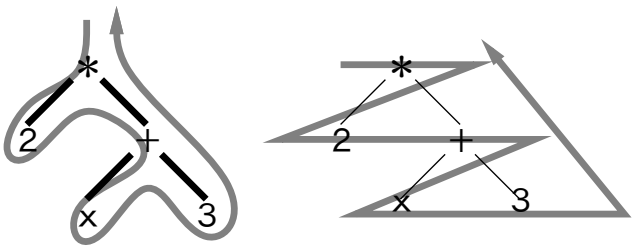

図 3: 深さ優先と幅優先

これはどういう順でしょう? スタックや再帰関数を使ったたどりでは「まず枝をどんどん深い方にたどり、行き止まりになったら一番最近の枝分かれまで戻って来て次の枝をどんどん深い方にたどり、…」という順番になります。これを**深さ優先のたどり** (depth-first traversal) と言います。前にやったような、再帰手続きによるたどりも、深さ優先のたどりになります。

これに対し、キューを使うと「まず 1 レベル目を全部順番にたどり、次に 2 レベル目を全部順番にたどり、…」という形になります。これを**幅優先のたどり** (breadth-first traversal) と呼びます (図 3)。幅優先のたどりを使って「木の中にある何か」を探すと、その「何か」のうちで木の一番浅いところにあるものが最初に見つかります。

あるノードを処理する時点として「すべての子供の処理をする前」「すべての子供の処理が終わった後」の 2 通りが考えられ、前者を**行きがけ順** (preorder)、後者を**帰りがけ順** (postorder) と呼びます。幅優先のたどりでは行きがけ順以外はまず使われませんが、深さ優先ではどちらも使われます。さらに 2 分木の場合は「左の子の処理が終わって右の子の処理をする前」という順序があり、これを**通りがけ順** (inorder) と呼びます。

演習 2 スタックまたはキューを使って式木をたどる例を打ち込んで動かせ。さまざまな木を与えた時にたどり順がどうなるか、まず予測し、次に実行して確認せよ。

演習 3 図 4 にとある鉄道路線図を示す。[3]これを表すデータ構造を設計して作れ。またそのデータをたどって「東京」から「八王子」、または「横浜」から「赤羽」の経路を出力するプログラムを作れ。[4]スタック版とキュー版で比較すること。

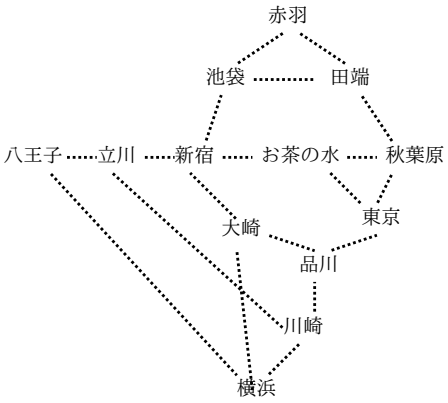

図 4: とある鉄道路線図

演習 4 エディタバッファのデータ構造として、次のようなものも考えられる。

- 2 つのスタック A、B に「現在位置より上の行」「現在位置以降の行」を入れるものとする。
- スタック B の一番上が現在行と考える。

[2] `is_a?`というメソッドは、オブジェクトがあるクラス (ないしそのサブクラス以下) のインスタンスかどうかを調べるもので、ここでは定数と変数は子供がないのでそれ以上たどらないための枝分かれに使っています。

[3] 架空のものであり、現実の場所とは関係ありません。

[4] ノード (分岐駅) ごとにレコードを作り、隣接ノードの情報を格納します。最初に出発ノードをスタック (キュー) に入れ、以後 1 つ取り出してそれがゴールでないならその隣接駅をすべてスタック (キュー) 入れます。ただし同じ場所を堂々巡りにならないために、「訪問ずみ」を表すフィールドを設け、最初にスタック (キュー) に入れる時に `true` にして、以後到達したノードでこれが `true` のものは処理ずみなのでスタック (キュー) に入れないようにします。

- 1 つ下に行くには、スタック B から 1 つ pop してスタック A に push。
- 1 つ上に行くには、スタック A から 1 つ pop してスタック B に push。
- 現在行を消すには、スタック B から 1 つ pop。
- 新しい行を挿入するには、スタック A に 1 つ push。

この方針によるエディタを作ってみよ。コマンドや機能の設計は各自に任せます。

3 状態空間の探索

3.1 状態と状態遷移

我々の日常生活において、ある同じ行動をしても結果が異なることはよくあります。たとえば大学の食堂で友人と談笑するのは構いませんが、授業中の講義室ではまずいですね。つまり談笑という同じ行動を取っても、**文脈** (context) が違うと結果が違うわけです。ここで、同等と見なせる文脈の集合を**状態** (state) と呼びます。つまり講義中という状態、映画館で映画を見ている状態、普段の状態、などの状態があり、「どの状態である」かに応じて「行動の結果」が変化すると考えるわけです。

コンピュータ上では日常生活よりもいっそう「状態」の違いを意識する必要があります。同じコンピュータの前に座っていて同じようにキーボードを打ち込んでも、動いているプログラムがワープロソフトなのかコマンドの窓なのかによって全く結果が違います。そして、プログラムの中もまた同様です。たとえば `x + 1` という式の値は、変数 `x` に入っている値が 4 であれば 5 になるし、10 であれば 11 になります。ですから、変数の値を書き換えつつ計算が進行していく手続き型計算モデルの場合、状態とは「すべての変数の値の集合」ということになるわけです。

状態について考える時は、ある状態から別の状態への移行、つまり**状態遷移** (state transition) も一緒に考える必要があります。これは、どのような場合にどの状態からどの状態へ移る、という定式化がなければ、現在いる状態はどれなのかも分からないからです。たとえば、先の演習にあった路線図の問題の場合、「現在どの駅にいるか」が状態であり、「ある駅から (電車に乗って) 別の駅へ移動する」ことが状態遷移ということになります。

3.2 有限オートマトン

有限オートマトン (finite automata) は、状態と状態遷移をモデル化する手法の 1 つです。ここでは「文字列がある規則にあてはまっているかどうか調べる」問題を題材として、有限オートマトンを紹介します。

まず、規則として「文字列は a から始まり、a と b が交互に現れ、最後は a で終わる」というものを例にしましょう。このような規則へのあてはまりを調べるのに、図 5 のようなグラフを使うことができます。

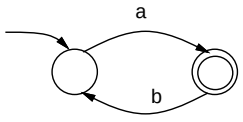

図 5: 有限オートマトンの例

グラフは○と◎の 2 種類のノードからできていて、各ノードは 1 つの状態を表します。最初は、外から矢線が来ている状態にいるものとします。これを**初期状態** (initial state) と呼びます。そしてここから、入力を 1 文字ずつ取り出し、その文字に対応するラベルを持った矢線をたどって次の状態へ遷移することを繰り返します。そして最後の文字まで終わった時、◎の状態にいれば「OK」です。

◎以外の状態にいたり、途中で文字に対応する矢線が無かったりした場合は「OK でない」ものとします。この「OK」であることをオートマトンの用語では**受理** (accept)、◎の状態のことを**受理状態** (accept state) ないし**最終状態** (final state) と言います。

整理すると、有限オートマトンとは 1 個以上の節 (=状態) をラベルのついた矢線ないし**有向辺** (directed edge) で結んだグラフで、節のうち 1 個以上が最終状態として指定されているもの、と言えます。図 5 を見て、「a から始まり、a と b が交互に現れ、最後は a」という規則にあてはまった文字列のみを受理することを確認してください。

ではこれを Ruby プログラムで表してみましょう。それぞれの状態を整数で表し、初期状態は 0 番であるものとします。次の状態への矢印を表すため、各状態は 1 つのハッシュとし、ラベルの文字に対して次の状態番号を返すようにします。たとえば、図 5 の場合、0 番の状態は (a が来たら 1 番へ行くわけですから) 次のようなハッシュで表せます:

```
{'a' => 1}
```

さらに、各状態が最終状態か否かも記録する必要がありますが、これも一緒のハッシュに入れることにして、`:final` という記号値に対応する値が `true` の場合は最終状態だということにします。すると、図 5 の場合、1 番の状態は次のようになります:

```
{'b' => 0, :final => true}
```

これらを配列としてまとめたものを、広域変数`$atm` に入れておくことにしましょう。文字列を与えてオートマトンが受理するか否かを調べるメソッド `accept` と一緒に示します。

```
$atm = [{'a' => 1}, {'b' => 0, :final => true}]
def accept(s)
  cur = 0
  s.length.times do |i|
    k = $atm[cur][s[i..i]];
    if k == nil then return false else cur = k end
  end
  return $atm[cur][:final] == true
end
```

`accept` では `cur` に現在の状態番号を保持するようにしていて、これを最初は 0 にします。そして文字列の 0 番目、1 番目、…の文字について順に、「オートマトンの現在の状態のハッシュに対してその文字を検索」した結果を変数 `k` に入れます。これが `nil` であれば「矢線がない文字」だったのでただちに不受理 (`false`) を返します。そうでなければ状態を `k` にしてさらに続けます。最後まで終わった時に、現在の状態が最終状態であれば受理 (`true`)、そうでなければ不受理 (`false`) を返します。ここで単に `return $atm[cur][:final]` とせずにわざわざ `true` と等しいかどうか比べているのは、最終状態以外の状態ではハッシュから `:final` の値を取り出すと `nil` なのでそれがそのまま返されないためです。

実際にやってみましょう:

```
irb> accept 'aba'
=> true
irb> accept 'ababa'
=> true
irb> accept 'ab'
=> false
irb> accept 'abba'
=> false
```

確かに先の規則にあてはまるものだけが受理できています。

演習 5 上の例題をそのまま打ち込んで動かせ。動いたら、図 6 の (a)〜(c) について、まずこれらのオートマトンがどのような文字を受理するかを考えて紙に書き、続いて上のコードのオートマトンのデータを変更して動かし確認せよ。

演習 6 次のような文字列のみを受理するオートマトンをまず紙に描け。続いて例題のオートマトンのデータを変更して動かし、確認せよ。

a. a が 1 個以上連続し、最後に 1 個だけ b がある。

b. a と b がさまざまに混ざって現れるが、a の連続は最大 2 個まで。

c. a と b がさまざまに混ざって現れるが、a の個数は偶数個。

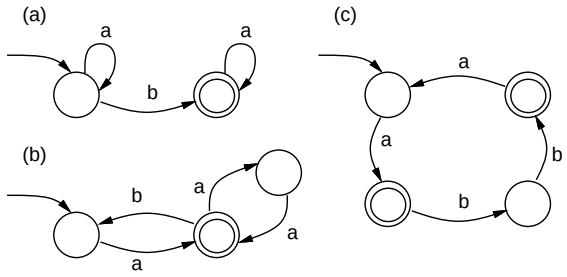

図 6: 演習 5 の有限オートマトン

3.3 ゲーム・パズルと状態空間

カードや駒などを使う多くの**ゲーム** (game) や**パズル** (puzzle) は、個々の場面を 1 つの状態と考え、プレーヤが選択する「手」を状態遷移と考えることで定式化できます。たとえばソリティア (トランプの一人遊び) では場のカードの内容 (どれが表/裏かも含む) と山のカードの内容全部を合わせたものが 1 つの状態であり、プレーヤが場のカードを移動したり次のカードをめくってどこかに置いたりするのが状態遷移となります。

この時、有限オートマトンと違うのは、有限オートマトンでは比較的少数の状態が予め分かっていてそれを予め全部用意していたのに対し、ゲームやパズルでは可能な状態の数が非常に多く、それを全部生成しておくことは非現実的だという点です。

そこで、「手」を打つごとに次の状態をその場で生成して行き、目指す状態 (自分が勝利したり、パズルが解けた状況のことです) が見つかったらそこでおしまい、という処理が必要です。これを**状態空間の探索** (state space search) と言います。

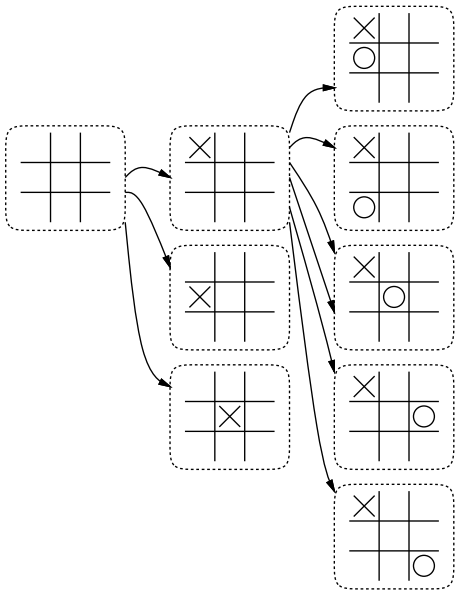

図 7: マルバツの状態空間 (一部)

簡単なゲームの例として、図 7 にマルバツの状態空間の始めの方を描いてみました (×が先手であるものとします)。マス目が 9 個あり、それぞれに (1) ○が入る、(2) ×が入る、(3) まだどちらも入っていない、の 3 つがあるので、最大で $3^9 = 19683$ 通りの状態があることになります。ただし、途中でどちらかが勝ったらおしまいですし、上下/左右/点対称のものは分けなくてもよいわけなので、実際に扱う状態はだいぶ少なくて済みます。

2 万程度の状態数であれば、今日のコンピュータではすべてを列挙することが可能です。状態を全部列挙できてしまえば、その中で「勝ち筋」をたどっていくだけでゲームに勝てることになります。または勝ち筋がなければ引き分けをめざしますが、途中で相手が失策をしたら勝ち筋に乗れるかもしれません。

一方、囲碁とか将棋とかだと、状態の数が本当にすごく多くなるので、これをコンピュータで扱うのはまだまだ大変です。

スタックを用いた深さ優先探索の場合、状態空間探索の基本的なアルゴリズムは次の形になります (スタックをキューに取り換えると幅優先探索になります):

```
s = 初期状態; s.mark; stack.push(s)
while !stack.isempty do
  s = stack.pop
  (s に隣接ずる状態を生成) do |s1|
    if s1.isgoal then ☆☆☆ゴールに到達した!☆☆☆
    elsif !s.ismarked then s1.mark; stack.push(s1)
    end
  end
end
☆☆☆全状態を生成したがゴールに到達できなかった!☆☆☆
```

マルバツのように駒を置く一方のゲームはある状態から遷移していって再度同じ状態に来ることは (駒は増える一方なので) あり得ませんが、将棋のようなゲームでは同じ状態に戻って来ることがあります。この時に、また同じ処理をしていたら終わらなくなりますから、状態ごとに「この状態は処理し終わった」という印をつけて再度同じ状態に来たら「もう処理ずみ」として飛ばす必要があります。これを上では `mark`、`ismarked` で表しています。

3.4 例題: 箱入り娘

「箱入り娘」というパズルをご存じでしょうか。これは図 8 のような枠と駒から成るパズルで、初期状態から駒をスライドさせていって、最終的には「箱入り娘」が出口から出られる状態にする、というものです。

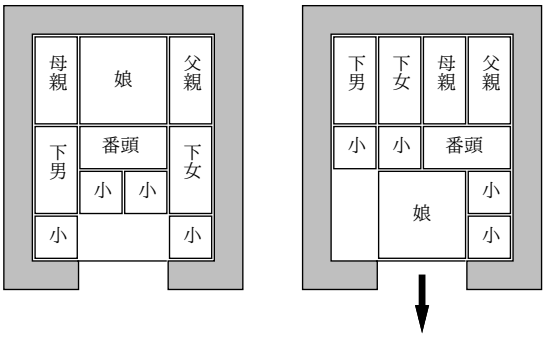

図 8: 箱入り娘の初期状態と最終状態 (の 1 つ)

では、これを解くプログラムを説明しましょう。まず、縦横のマス目の数、「目標」となる娘の位置、駒 (0〜9 で表す) の縦と横のサイズ、および駒の名前を表す 1 文字 (表示文字列生成用) をグローバル変数に入れておきます:

```
$hmax = 3; $vmax = 4; $goalx = 1; $goaly = 3
$hsize = [2, 1, 1, 1, 1, 2, 1, 1, 1, 1]
$vsize = [2, 2, 2, 2, 2, 1, 1, 1, 1, 1]
$names = 'Mvvvvwssss.'
```

次に、9 個の駒の位置 (X 座標、Y 座標をそれぞれ要素数 9 の配列に入れてあるものとします) を受け取り、盤面を表現した 2 次元配列を作るメソッド `makeboard` を示します:

```
def makeboard(x, y)
  b = Array.new($vmax+1) do Array.new($hmax+1, -1) end
  $hsize.length.times do |i|
```

```
      $hsize[i].times do |j|
        $vsize[i].times do |k| b[y[i]+k][x[i]+j] = i end
      end
    end
    return b
  end
```

まず最初に全要素が-1(何も置いてない) の 2 次元配列を作ります。そして各駒について、その駒の置かれている全範囲に駒の番号を入れます。

さて次に、盤面 b を受け取り、駒 i が現在 x,y にあるものとして、これを dx,dy だけ動かすことが可能かどうか調べるメソッド movable を見てみましょう:

```
  def movable(b, i, x, y, dx, dy)
    if x+dx < 0 || x+$hsize[i]-1+dx > $hmax ||
       y+dy < 0 || y+$vsize[i]-1+dy > $vmax then return false end
    $hsize[i].times do |j|
      $vsize[i].times do |k|
        p = b[y+dy+k][x+dx+j];
        if p != -1 && p != i then return false end
      end
    end
    return true
  end
```

まず、動かした後が盤面からはみ出るなら NO を返します。そうでないなら、動かした後の自分の置かれる範囲のマスすべてについて、そこが-1(何も置いてない) でもこれまで自分が置いてあったマスでもないなら直ちに NO を返します。最後まで NO でなければ OK を返します。

盤面の 1 つの状態を表すのにオブジェクトを使うことにしたので、State というクラスを作成しました。インスタンス変数としては、9 個の駒の XY 座標の配列、および「1 つ前の状態」を保持します。なお、配列は initailze で渡されてきたものをそのまま持っていると書き換えられてしまうので、メソッド dup でコピーを作ってそれを保持します。

getx、gety は各駒の座標値の配列をそのまま返します。isgoal は駒 0 が目標位置にあるか否かを返します。move は自分の状態から駒 i を dx,dy だけ動かした新しい状態を作って返します。この時一時的に配列中の駒の位置を動かして新しい状態を作り (self というのは「このオブジェクト」を渡すという意味)、終わったら戻します (このため、dup によるコピーが必要なのでした):

```
  class State
    def initialize(x, y, p=nil) @x=x.dup; @y=y.dup; @prev = p end
    def getx() return @x end
    def gety() return @y end
    def isgoal() return @x[0] == $goalx && @y[0] == $goaly end
    def move(i, dx, dy)
      @x[i] = @x[i] + dx; @y[i] = @y[i] + dy;
      s = State.new(@x, @y, self)
      @x[i] = @x[i] - dx; @y[i] = @y[i] - dy; return s
    end
    def to_s()
      s = ''
      makeboard(@x, @y).each do |a|
        a.each do |i| s = s + $names[i..i] end; s = s + "\n"
      end
      return s
```

```
    end
    def output(f)
      if @prev != nil then @prev.output(f) end
      f.puts(to_s); f.puts('------')
    end
  end
```

to_s は makeboard で盤面の配列を作ったあと、それを 1 つの文字列にしていますが、その時駒の番号ではなく各駒を表す文字 1 文字にしているのに注意してください。これは、縦長の駒や小僧の駒はどれでも同じことなので、これらを同じ名前にすることで状態数を少なくできるためです。output は最終状態に到達できた時使うメソッドで、ファイル f を渡されて最初に近い状態から順にファイルに状態を書き出します。@prev は逆向きのリストなので、先頭から表示するため再帰を使っています。

では最後に main を示します。今回は速度が問題なので自前のスタックではなく Ruby の配列をスタックに使うことにしました。x と y は各駒の初期配置で、これらを指定して最初の状態を作り、スタックに積みます。また、ハッシュ visited は状態の文字列表現を入れることでその状態を既に処理ずみであることを表すのに使うので、まずは最初の状態を「訪問ずみ」としました。その後が処理本体で、まずスタックが空ならもう調べる状態がないので失敗です。そうでなければスタックから 1 つ取り出し (進捗を見るためここで画面に表示してみるのも参考になります)、取り出したのが目的状態ならループを抜け出します。そうでない場合は、処理する状態の各駒の座標を取り出し、全ての駒について上下左右に移動できるか調べ、できるのならその新しい状態を生成し、それが既に処理ずみ (な状態と同等) でないなら、スタックに積んで処理ずみとして登録します:

```
  def main
    x = [1,0,3,0,3,1,0,1,2,3]; y = [0,0,0,2,2,2,4,3,3,4]
    stack = [State.new(x,y)]; visited = {stack[0].to_s => true}
    while true do
      if stack.length == 0 then puts("impossible."); return false end
      s = stack.pop       # or s = stack.shift
      puts(s.to_s); puts('----')
      if s.isgoal then break end
      x = s.getx; y = s.gety; b = makeboard(x, y)
      $hsize.length.times do |i|
        [[1,0],[-1,0],[0,1],[0,-1]].each do |a|
          if movable(b, i, x[i], y[i], a[0], a[1])
            s1 = s.move(i, a[0], a[1]); k1 = s1.to_s
            if visited[k1] == nil then stack.push(s1); visited[k1] = true end
          end
        end
      end
    end
    File.open('out.data', 'w') do |f| s.output(f) end
    return true
  end
```

成功してループを抜けた時にはファイル out.data に初期状態から目的状態までの各状態を出力しておしまいです。長かったですね、おつかれ様でした。では動かしてみましょう:

```
irb> main
vMMv
vMMv
vwwv
vssv
s..s
----
vMMv
```

```
vMMv
vwwv
vssv
s.s.
----
vMMv
vMMv
vwwv
vssv
ss..
----
(途中略)
vvvv
vvvv
ssww
MM.s
MM.s
----
vvvv
vvvv
ssww
.MMs
.MMs
----
=> true
```

解けたようですね。そして、実行時の表示には「行き詰まった」枝も全部含まれますが、ファイルには見た目はこれと同様でも、正しく解ける経路だけが記録されています。

ちなみに、スタックを使うと探索が深さ優先なので、解けるまでの時間は短いですが、得られた解が最短とは限りません。幅優先探索にすると状態数が多くなりがちですが、得られた解は最短手数のものとなります。そうするには`stack.pop`の代わりに配列の先頭から取り除くメソッド`stack.shift`を使えばよいでしょう (もはやスタックでないのに`stack`という名前なのはまずいですけど)。

演習 7 上のプログラムを打ち込み、動作を確認せよ。成功したら、駒の大きさや配置を変更して正しく動作するか確認せよ。できれば、幅優先と深さ優先の比較もやってみてほしい。

演習 8 自分の好きなゲームないしパズルを扱う状態空間探索プログラムを作れ。

演習 9 その他、ここまでにこの授業で学んだ内容を活かした面白いプログラムを作れ。何が面白いかの定義は各自に任されます。

A　本日の課題 10A

「演習 1」「演習 2」「演習 5」「演習 6」で動かしたプログラムどれか 1 つを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 10A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. プログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

   Q1. スタック、キューについて理解しましたか。

   Q2. 状態、状態遷移、オートマトンとか納得しましたか。

   Q3. 本日の全体的な感想と今後の要望をお書きください。