情報科学 2007　久野クラス　＃9

久野　靖*

2006.12.14

はじめに

いよいよ今回と次回で年内はおしまいです。で、来年に入って以降はB課題 (次回までの課題) はなくなりますので、今回と次回でB課題はおしまいです。なので、ゆっくりやって頂くように、今回のB課題は1/11、次回のB課題は1/18を締切とします。さて、今回の内容は次のものです。

- プログラミング言語処理系の内幕
- 木構造の続編 — 抽象構文木
- オブジェクト指向の続編 — 動的分配と継承

その前に演習問題解説がありますが、今回もなるべく簡単に済ませようと思います。

1　演習問題解説

1.1　演習1

この演習はメソッドを個別に増やすだけなので各メソッドだけ掲載する。まず削除。これは皆様やってくれたし簡単ですね。ここでは「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た。もちろん2つの代入に分けてもよい。

```
def delete
  if atend then return end
  @cur = @prev.next = @cur.next
end
```

次に交換だが、こういうのは作業変数を使った方が間違えないで済む。まず現在行と次の行が「おしまい」だったら交換できないのでそれを除外。あとは最終的に並ぶ4つのセル (中央の2つが交換) を変数a、b、c、dに入れて、つなぎ直し、@curを変更する。

```
def exch
  if atend || @cur.next == @tail then return end
  a = @prev; b = @cur.next; c = @cur; d = @cur.next.next
  a.next = b; b.next = c; c.next = d; @cur = b
end
```

さて、1つ戻るのはどうだろう。せっかくある程度メソッドが作ってあるわけだから、「@prevを覚えておき、先頭に行ってから現在行が覚えておいた行になるまで1行ずつ進む」方法で作ってみた。

*筑波大学大学院経営システム科学専攻

```
  def backward
    if @prev == @head then return end
    a = @prev; top; while @cur != a do forward end
  end
```

全部反転はちょっと大変そうですね。要は、先頭から順にたどりながら、今まで「前→後」の対だったものを「後←前」の順になるようにポインタをつなぎ替える、ただし先頭と末尾はそれなりに対処、ということか。

```
  def invert
    top; if atend then return end
    a = @cur; b = @cur.next; a.next = @tail
    while b != @tail do c = b.next; b.next = a; a = b; b = c end
    @head.next = a; top
  end
```

先頭と末尾の対処はまず最初に先頭の次を`@tail`にし、最後にループを抜けて来た時のセルを先頭 (`@head`の次) にする、ということ。なお、バッファ内に 1 行しかないときはループ周回数が 0 で、そのときもちゃんと動作することに注意。

演習 2 がなんと 2 つありました! すみません…で、双方向リストのはやりたい人のお楽しみなので省略させてください。長くなるし。

1.2 演習 2

まず、「指定した行へ行く」機能はバッファ側で「何行目」を管理するのがいいので、エディタバッファ全体を示す。基本的に、インスタンス変数`@lineno`を追加して、現在行が変化するメソッドではこれを更新する。`invert`みたいにぐちゃぐちゃにいじる場合は最後に`top`を呼ぶので、ここで 1 にリセットされるから考えないでよい。これがあれば`backward`はもっと簡単になるのでこれも直した。

```
class Buffer
  Cell = Struct.new(:data, :next)
  def initialize
    @tail = @cur = Cell.new("EOF", nil)
    @head = @prev = Cell.new("", @cur)
    @lineno = 1
  end
  def getlineno
    return @lineno
  end
  def goto(n)
    top; (n-1).times do forward end
  end
  def atend
    return @cur == @tail
  end
  def top
    @prev = @head; @cur = @head.next; @lineno = 1
  end
  def forward
    if atend then return end
    @prev = @cur; @cur = @cur.next; @lineno = @lineno + 1
  end
```

```
    def insert(s)
      @prev.next = Cell.new(s, @cur); @prev = @prev.next; @lineno = @lineno + 1
    end
    def print
      puts("  " + @cur.data)
    end
# delete、exch は上掲の通り。backward は以下のように変更
    def backward
      goto(@lineno - 1)
    end
    def invert
      top; if atend then return end
      a = @cur; b = @cur.next; a.next = @tail
      while b != @tail do c = b.next; b.next = a; a = b; b = c end
      @head.next = a; top
    end
    def subst(str)
      if atend then return end
      a = str.split('/')
      @cur.data[Regexp.new(a[1])] = a[2]
    end
    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
  end
```

エディタドライバ側は、毎回「line[0..0]」とか書くのに疲れたので、変数に入れておくように直した。

```
  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; l = e.getlineno;
                          s.to_i.times do e.forward; e.print end; e.goto(l)
      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
      elsif c == "x" then e.exch
      elsif c == "b" then e.backward
      elsif c == "v" then e.invert
      elsif c == "a" then e.forward; e.insert(s); e.backward
      elsif c == "c" then e.delete; e.insert(s); e.backward
      elsif c == "g" then e.goto(s.to_i)
      else                e.forward; e.print
      end
    end
  end
```

「位置変更しない指定行数プリント」も、最初に行番号を覚えて、印刷し終わったらそこに戻ればいいので簡単である。なお、行番号をきちんと (間違いなく) 維持するのはきわどそうに見えるが、`@lineno` をアクセスするのもバッファ内容や現在位置を変更するのも `Buffer` の中だけなので、この中でそれぞれをきちんと維持すれば大丈夫だと分かる。つまり、オブジェクト指向の持つ**カプセル化**の機能によって、プログラムを正しく構成し易くなっているわけだ。

1.3 演習 6〜9

これも演習番号が飛んでいてすみません。表探索のは「作ってみる」「計測してみる」がメインなので 1 つだけ、前回説明しなかったハッシュ表と呼ばれる手法を説明しよう。これは、次のような原理による。

- 整数キーの範囲ぶんの配列を用意できれば、ビンソートと同様に 1 発でそのキーの値を格納/参照できるから極めて速い。
- しかし整数キーの範囲が広いと領域が大きくなり実用的でない。
- そこで整数キーを適当に「折り畳んで」用意した配列の範囲に写像し、そこに格納すればよい。この関数を $hash1(k)$ と呼ぶことにする。
- ただし、複数のキーが配列の同じ位置に写像されること (「衝突」と呼ぶ) がある。このときは、別の関数 $hash2(k)$ により値 N を計算し、N 個先、そこもふさがっていたらさらに N 個先、…のように空いている場所が見つかるまで探して行く。

この方式で実装した表のクラスを示そう。配列のサイズがヘンな数だが、10000 に近い素数を選んだもの。素数であれば、N 飛びに調べて行く時に「同じところをぐるぐる調べてしまう」ことがないので、空きがある限りその空きが見つかる。

```
  class IntStrTable3
    def initialize
      @keys = Array.new(9973, -1); @vals = Array.new(9973, nil)
    end
    def hash1(n) return n % @keys.length end
    def hash2(n) return n % 2971 + 1 end
    def put(key, val)
      h1 = hash1(key); h2 = hash2(key)
      while true do
        if @keys[h1] == key then @vals[h1] = val; return end
        if @keys[h1] < 0 then @keys[h1] = key; @vals[h1] = val; return end
        h1 = (h1 + h2) % @keys.length
      end
    end
    def get(key)
```

```
    h1 = hash1(key); h2 = hash2(key)
    while true do
      if @keys[h1] == key then return @vals[h1] end
      if @keys[h1] < 0 then return nil end
      h1 = (h1 + h2) % @keys.length
    end
  end
```

ただしこれはちょっとさぼっていて、表が満杯になったときのチェックをしていない。満杯になると、入れる場所が見つからなくて無限に探し始めてしまう。

では、計測してみよう。計測のしかたは前回資料と同じく、N 回の put、N 回の get の時間を計測。キーは 0〜100000 のランダム。上段は N 個の put、下段は NN 個の get の時間。

表 1: 表のさまざまな実装の計測

実装	1000	2000	3000	5000	10000
線形探索	0.3984375	1.578125	3.5625	9.578125	39.3671875
	0.78125	3.0546875	7.0390625	18.734375	79.515625
2 分探索木	0.03125	0.0703125	0.09375	0.1875	0.375
	0.015625	0.078125	0.078125	0.1484375	0.2890625
ハッシュ	0.0078125	0.015625	0.015625	0.0234375	0.09375
	0.0078125	0.0078125	0.015625	0.03125	0.296875
Ruby の	0.0	0.0	0.078125	0.015625	0.03125
Hash	0.0	0.0078125	0.0	0.078125	0.015625

なお、最後の Ruby の Hash は次のような簡単なクラスを作ってインタフェースを合わせて計測した。だいぶ例題が長くなってきたので、1 行で済むメソッドは 1 行に書くようにしたが、その場合、引数が無くても「()」を書く必要があるので注意。

```
class IntStrTable4
  def initialize() @hash = {} end
  def put(key, val) @hash[key] = val end
  def get(key) return @hash[key] end
end
```

さて、これを見るとハッシュ表が圧倒的に速いことはよく分かる。Ruby の Hash も名前通り、内部の実装はハッシュ表になっている。ハッシュ表の場合、表が十分空いていれば、「1 発で」項目が検索できるため、項目当たりの検索時間は $O(1)$ になる。ところで、上で挙げたハッシュ表の実装で 10,000 個の場合が妙に遅いが…これは、表のサイズが 9973 なので 10,000 回ランダムに値を投入すると表は (重複はあるにせよ) ほぼ満杯になり、そのため遅くなっているわけだ。ハッシュ表の場合は、最初に十分大きさを見積もって、表の充填率が 50%以下になるように設計するのがよいとされている。

あと残っている課題として、2 分探索木で挿入時にキーが順番になっていたらどうなるかというのがあった。これは、キーが順番だと「1 直線の」木ができてしまい、通常なら木の深さが $logN$ 程度、1 回当たりの検索に掛かる時間が $O(logN)$ のところが、木の深さが N、1 回当たりの検索に掛かる時間も $O(N)$ になってしまう。これを避ける方法としては、次のものが知られている。

- AVL 木 — 左右の木に入っている値の数が常にバランスするように制御しながら挿入していく方法
- スプレー木 — 値の挿入/削除ごとに木の構造をランダムに変形することで偏った木ができる確率が極めて低くなるという、確率的アルゴリズムを用いる方法

いずれも説明している紙面はないので、興味があれば文献等を調べてみていただきたい。

2 言語処理系の内幕

2.1 抽象構文木/式木

前回学んだ木構造は情報科学のさまざまな方面で出て来るが、その 1 つの方面として、我々が使っている Ruby のようなプログラミング言語による記述の構造を表すというものがある。たとえば「`x + 1`」という式は、+という (加算の) ノードの左右に `x`(変数ノード) と `1`(定数ノード) がぶら下がった木構造で表現できる (図 1 左)。このような、プログラミング言語の構文に対応した (ただし簡潔に整理した) 木構造を**抽象構文木** (abstract syntax tree)、その中で演算式に対応するようなものを**式木** (expression tree) と呼ぶ。

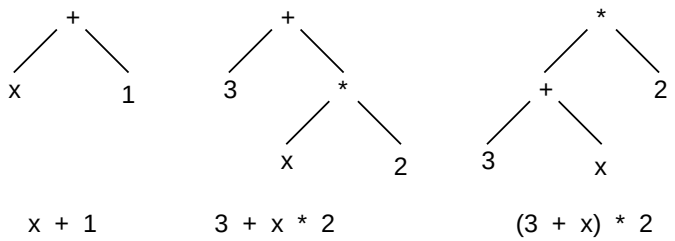

図 1: 演算式の抽象構文木 (式木)

これだけなら何ということはないが、もう少し複雑な「`3 + x * 2`」を考えると、これは `x` に 2 を書けて、それを 3 と足すのに対し、「`(3 + x) * 2`」だと 3 との足し算の方が先になるが、これらの演算順序の違いなども木構造では素直に表せる (図 1 中・右)。

なお、上の例でも分かるように、通常の数式では「乗除算は加減算より優先」とか「かっこ内は先に計算」などの規則があるが、これらは「1 列に」式を表す時に必要な解釈の規則であり、式木になった状態では演算順序が木の構造によってそのまま表されている。

2.2 抽象構文木のオブジェクト表現/動的分配

では次に、抽象構文木をオブジェクトの集まりで表現することを考えよう。前回の 2 分探索木では、各ノードをレコードで表現したが、今度は「加算」「乗算」「変数」「定数」などさまざまなノードがあるので、各ノードを 1 つのクラスで表すことにする。さっさとコードを見てみよう。変数の値はグローバルなハッシュ`$vars` に、変数名のエントリにその変数の値を格納する形で保持する (1 行目がその初期化)。

```
$vars = {}
class Add
  def initialize(l, r) @left = l; @right = r  end
  def exec() return @left.exec + @right.exec end
  def to_s() return '(' + @left.to_s + ' + ' + @right.to_s + ')'  end
end
class Mul
  def initialize(l, r) @left = l; @right = r end
  def exec() return @left.exec * @right.exec end
  def to_s() return '(' + @left.to_s + ' * ' + @right.to_s + ')' end
end
class Lit
  def initialize(v) @left = v end
  def exec() return @left end
  def to_s() return @left.to_s end
end
```

```
class Var
  def initialize(v) @left = v end
  def exec() return $vars[@left] end
  def to_s() return @left.to_s end
end
```

クラス Add と Mul は加算と乗算のノードを表現する。これらは初期化のとき左右の子供を受け取り、インスタンス変数 @left と@right に格納する。メソッド exec は計算を実行し結果を返すもので、左と右それぞれの計算を実行した後、その結果を加算/ 乗算したものを結果として返せばよい。to_s は文字列表現を返すので、左の子と右の子の文字列表現を+や*でつなげばよい…が、入れ子関係が分からなくなると困るので、全体をかっこで囲んだ。

クラス Lit と Var は定数と変数のノードを表現する。初期化時に定数はその値、変数はその名前文字列を受け取り、(上となるべく揃えるようにしたので) インスタンス変数@left に格納する。exec は定数では値を返すだけだが、変数では$vars に格納されている変数の値を取り出して返す。to_s は変数や定数の文字列を返すだけ。

ところでここで注意して欲しいのは、@left.exec のようなメソッド呼び出しは、@left に現在何が入っているかに応じて、そのオブジェクトのメソッドを呼び出しているという点である。つまり、@left が Add オブジェクトなら足し算、Var オブジェクトなら変数値の参照が行われて結果が返される。このように、「実行時に実際に入っているものに応じてメソッドが選択される」機能のことを**動的束縛**といい、オブジェクト指向言語に不可欠の機能である。これがあるおかげで、式の内容が具体的に何であっても「実行する」とだけ言えばその内容に応じて動作してくれるわけだから。

さて、以下のメソッド test はテスト用に作ったので、irb に直接打ち込んでテストしても構わない。

```
def test
  $vars['x'] = 5
  e = Add.new(Var.new('x'), Lit.new(1))
  puts(e); puts(e.exec)
  e = Add.new(Lit.new(3),
              Mul.new(Var.new('x'), Lit.new(2)))
  puts(e); puts(e.exec)
  e = Mul.new(Add.new(Var.new('x'), Lit.new(3)),
              Lit.new(2))
  puts(e); puts(e.exec)
end
```

変数 x の値を 5 とし、あとは図 1 の 3 つの式木を組み立て、それを表示した後、実行して結果を打ち出す。その様子を示す。

```
irb(main):004:0> test
(x + 1)
6
(3 + (x * 2))
13
((x + 3) * 2)
16
=> nil
```

確かに正しく実行 (計算) できている。

演習 1 上の例題を打ち込み、式の内容は変えて、正しく計算できることを確認せよ。できたら、減算と除算も追加してみよ。

演習 2 以下のようなノードのクラスを追加し、正しく実装できていることを確認せよ。かっこ内は子ノードの数。to_s での表現方法は適当に決めてよい。

a. 出力 (1)。このノードを exec すると、子ノードの内容を puts により出力し、値としてはその出力した値をそのまま返す。

b. 入力 (0)。このノードを exec すると、プロンプトを出してから「gets.to_i」で整数を読み込み、それを結果として返す。

c. 連続 (2)。このノードを exec すると、左の子ノード、右の子ノードの順に exec し、結果としては右の子ノードを exec した結果を返す。

d. ループ (2)。このノードを exec すると、左の子ノードを exec した結果の回数だけ繰り返し、右の子ノードを exec する。結果は最後に右の子ノードを exec した結果を返す。

2.3 抽象構文木によるプログラム表現/継承

ところで、前節の例題を見て、Add と Mul のコードは非常に類似していると思わなかっただろうか。クラスを沢山作り始めると、用途によってはこのようなことが起きる。そのとき、多くのクラス方式のオブジェクト指向言語では、毎回同じものをコピーする代わりに**継承** (inheritance) と呼ばれる機能を使うことで記述を簡潔にできる。継承とは次のような機能のことを言う。

- クラスを定義するとき、既にある別のクラスを土台として新しいクラスを定義できる。土台にするクラスを親クラスないし**スーパクラス**、新しく作るクラスの方を子クラスないし**サブクラス**と呼ぶ。
- 子クラスは親クラスのインスタンス変数、メソッド定義をそっくりそのまま引き継ぐ (「継承」の意味)。
- さらに、子クラスで独自のインスタンス変数やメソッド定義を追加することができる。
- また、子クラスで親クラスと同名のメソッドを定義することで、親クラスのメソッドを差し替えること (**オーバライド**) ができる。

実際に、先のコードを継承を用いて構造を整理してみる。すべてのノードの土台となるクラス Node を用意し、必要そうなメソッド一式を用意した。@left、@right が左と右の子というのは同じだが、さらに表示用の演算子を入れるインスタンス変数@op も追加。

```
$vars = {}
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
```

getleft、getright、getop は各インスタンス変数の内容を参照したいときに備えて用意してある。ではこれを土台にして Add をサブクラスとして定義する。その場合、Ruby では「< 親クラス」という指定をおこなう。

```
class Add < Node
  def initialize(l, r) super; @op = '+' end
  def exec() return @left.exec + @right.exec end
end
```

このサブクラスでは initialize と exec だけを定義していて、前者は親クラスにあるメソッドなのでオーバライド (差し替え) の定義となる。

まず initialize だが、最初にある super というのは親クラスの同名のメソッドを (同じ引数で) 呼び出すという意味なので、これにより@left、@right の初期化が正しく行える (もし引数を変更したい場合は「(...)」で独自の引数を指定してもよい)。@op については「'+'」を自分で入れている。exec についてはこれまでの (継承を使わない) 版と同じ。

以下同様にして、他のノードも定義してみる。

```
class Sub < Node
  def initialize(l, r) super; @op = '-' end
  def exec() return @left.exec - @right.exec end
end
class Mul < Node
  def initialize(l, r) super; @op = '*' end
  def exec() return @left.exec * @right.exec end
end
class Div < Node
  def initialize(l, r) super; @op = '/' end
  def exec() return @left.exec / @right.exec end
end
class Lit < Node
  def initialize(v) super; @op = '#' end
  def exec() return @left end
end
class Var < Node
  def initialize(v) super; @op = '$' end
  def exec() return $vars[@left] end
end
```

もうちょっと頑張って、「代入」「連接」「ループ」「何もしない」のノードを用意した。

```
class Assign < Node
  def initialize(l, r) super; @op = '=' end
  def exec v = @right.exec; $vars[@left.getleft] = v; return v end
end
class Seq < Node
  def initialize(l, r) super; @op = ';' end
  def exec() @left.exec; return @right.exec end
end
class Loop < Node
  def initialize(l, r) super; @op = 'L' end
  def exec() v=0; @left.exec.times do v=@right.exec end; return v end
end
class Noop < Node
  def initialize() end
  def exec() return 0 end
  def to_s() return '?' end
end
```

代入は、左辺が変数であるものとして、その`@left`に変数文字列が入っているはずだから、`$vars`のその変数名のエントリに右辺の値を格納する。連接は、左を実行して、それから右を実行し、その結果を返すだけ。ループは左を実行した結果の回数だけ右を繰り返し実行。最後に実行した値を返すため変数`v`に毎回値を保存している (0 回実行の場合は 0 が返る)。何もしないというのは後で使うが、`to_s`もオーバライドしてただの「?」を返すようにしてある。

ではちょっと試してみよう。以下は、`n`に 5 を入れ、`x`に 1 を入れ、`n`の回数だけ繰り返し、`x = x * n`と`n = n - 1`を実行し、最後に`x`の値を全体の結果とするコードの抽象構文木を組み立て、印刷して、実行する。

```
def test1
  e =
```

```
    Seq.new(
      Assign.new(Var.new('n'), Lit.new(5)),
      Seq.new(
        Assign.new(Var.new('x'), Lit.new(1)),
        Seq.new(
          Loop.new(
            Var.new('n'),
            Seq.new(
              Assign.new(Var.new('x'), Mul.new(Var.new('x'), Var.new('n'))),
              Assign.new(Var.new('n'), Sub.new(Var.new('n'), Lit.new(1))))),
          Var.new('x'))))
  puts(e)
  return e.exec
end
```

では実行してみよう。

```
irb(main):006:0> test1
(((n$)=(5#));(((x$)=(1#));(((n$)L(((x$)=((x$)*(n$)));((n$)=((n$)-(1#)))));(x$))))
=> 120
```

確かに、ちゃんと階乗が計算できている。表示はちょっと読みにくいが、ほとんど 1 箇所のメソッドで全部まとめてやっているのでこんなものか。

このように、プログラムを内部的に抽象構文木として表現し、それを実行することで、「プログラムの記述通りの動作を行うプログラム」が作れる。一般に「プログラムの記述通りの動作を行うプログラム」のことを**インタプリタ** (interpreter、解釈実行系) と呼ぶ。上で示したように、抽象構文木を直接解釈するタイプのインタプリタは作りやすくそれなりに使われている (ただし遅いという弱点がある)。たとえば Ruby の実行系はこのタイプのインタプリタを用いている。

演習 3 ノードの種別として次のようなものを増やしてみよ。

a. 大小比較の演算子。ここでは値を全部整数としているので、たとえば「`x < y`」は条件の成否に応じて 1 または 0 を結果として持つ、とかいうことにするとよい。

b. while 文。条件部分は「0 が false、それ以外はすべて true」として扱うとよいだろう。

c. if 文。とりあえず then 部だけでよいが、頑張って else 部も書けるようにしたければそうしてもよい。

d. 絶対値、2 数の最大、最小などの演算子。

e. その他、目新しい/面白い機能を持った文や演算。

2.4 字句解析器

ここまでは、Ruby の構文を使って直接、オブジェクト群を組み合わせて抽象構文木を組み立ててきた。しかしもちろん、普段我々がプログラムを作る時は、もっと「普通の」書き方で書いて、処理系がそれを解析して抽象構文木を組み立てている。せっかくだから、そのようなものの「おもちゃ」を作ってみよう。

プログラミング言語処理系の一番入口となるのは、入力を「名前」「定数」「記号」などに切り分ける字句解析器と呼ばれる部分であるが、ここでは簡単のため「プログラムは 1 行だけ、すべての名前、定数、記号は 1 文字だけ、余分な空白などは一切あってはいけない」という非常に制限された字句解析用のクラスを作る。

```
class Scanner
  def initialize(s) @str = s + '$'; @pos = 0 end
  def peek() return @str[@pos..@pos] end
  def next() if @pos < @str.length-1 then @pos = @pos + 1 end end
```

```
  def to_s() return @str[0..@pos-1] + '!' + @str[@pos..-1] end
end
```

これだけである。initalize は解析用の文字列を@str、解析位置を@pos に設定する。なお「$」は終わりの印の文字として使うもので、プログラム中には存在しないものとする。peek は「現在位置の文字を返す」メソッドで、next は「現在位置を (終わりでない場合に)1 つ進める」メソッド。最後に to_s はエラー表示用に「どこを読んでいるか」が分かりやすいように表現した文字列を返すようにした。ちょっと動かしてみよう。

```
irb(main):009:0> sc = Scanner.new('x=x+1')
=> #<Scanner:0x810b460 @str="x=x+1$", @pos=0>
irb(main):010:0> sc.peek
=> "x"
irb(main):011:0> sc.next
=> 1
irb(main):012:0> sc.peek
=> "="
irb(main):013:0> sc.next
=> 2
irb(main):014:0> sc.next
=> 3
irb(main):015:0> sc.next
=> 4
irb(main):016:0> sc.peek
=> "1"
irb(main):017:0> sc.next
=> 5
irb(main):018:0> sc.peek
=> "$"
irb(main):019:0> puts(sc)
x=x+1!$
=> nil
```

最後のは、「x=x+1 というプログラムを読み終わってファイル終端のところにいる」ということを表しているわけだ。

演習 4 Scanner のインタフェース (メソッド peek、next を使うこと) は変えないで、普通のプログラミング言語のように複数文字から成る名前や定数 (整数だけでよい) を扱え、空白や改行は無視するような字句解析クラスを作り、後に出て来る構文解析器と組み合わせて動作させてみよ。文字列を渡す代わりにファイルからプログラムを読むようにしてもよい。

2.5 BNF による構文定義

Ruby の構文もちゃんと説明していないのに恐縮ですが、構文が決まらないと解析器が作れないので「おもちゃ言語」の構文を定義しよう。

$$
\begin{aligned}
prog &::= stat \mid stat\ \text{';'}\ prog \\
stat &::= \text{'\{'}\ prog\ \text{'\}'} \mid \text{'L'}\ expr\ stat \mid expr \\
expr &::= factor \mid factor\ \text{'+'}\ expr \mid factor\ \text{'-'}\ expr \\
&\quad \mid factor\ \text{'*'}\ expr \mid factor\ \text{'/'}\ expr \mid factor\ \text{'='}\ expr \\
factor &::= identfier \mid number \mid \text{'('}\ expr\ \text{')'}
\end{aligned}
$$

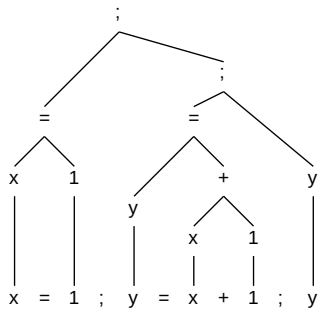

図 2: おもちゃ言語の抽象構文木

この記法は **BNF**(Backus Normal Form) と呼ばれ、::= の左辺の記号を右辺の記号列で定義する、と読めばよい (右辺中の | は「または」を表している)。上の文法では *identifier*(識別子、つまり名前のこと) と *number* が定義されていないが、これらの構造は字句解析器の方で決めるのが普通である。ここでは上記の字句解析器を使うため、それぞれ英小文字 1 文字、数字 1 文字に対応させることにする。

図 2 に「`x = 1; y = x + 1; y`」という簡単なプログラムに対する抽象構文木を示す。上掲の文法との対応関係を確認してみてほしい。

演習 5 次のプログラムに対する抽象構文木を描け。

a. `x=y=z=1`
b. `x=2*3+4*5`
c. `n=5;x=1;L(n){x=x*n;n=n-1};x`

演習 6 自分が (演習 2 などで) 導入した独自の機能の構文を決めてそれを含むように「おもちゃ言語」の BNF を拡張せよ。さらに、その構文を持つプログラム例を選び、抽象構文木を描け。

2.6 再帰下降型構文解析器

さて、字句解析を下請けに使って、プログラムの構文を解析し、抽象構文木を組み立てる機能のことを構文解析と呼ぶ。ここでは、再帰手続きの動作がプログラムの構文木をたどる動作と類似していることを利用した方式である、**再帰下降解析器** (recursive descent parser) と呼ばれる方式の解析器を作ってみる。再帰下降解析器を作る原理は次のようになる。

- 各構文記号の定義ごとに 1 つずつ手続きを用意する。各手続きの引数は字句解析器で、返値はその手続きに対応する抽象構文木となる。
- 各手続きは、その構文定義の右辺にある構文記号に対応する手続きを 1 つずつ呼び出し、返されてきた木を束ねて自分の構文木を組み立てる (呼び出すものが 1 しか無い場合は返された木をそのまま返すこともある)。
- 右辺に現れるもののうち個別の文字 (記号、英字、数字) については、次の文字がその文字であることを確認して入力を先に進める。
- 右辺が | によって複数の可能性から選択になっている場合は、どの選択肢へ進むかも入力に基づいて決定する。

特に重要なのは、最後の「入力に基づいて決定する」というところで、そのような選択がうまく行えないような文法はこの方法では解析できない。幸い、上に出て来る文法はこの方法で解析できるようになっている (というかもちろん私がそのように設計したのですが)。

図 3 に、おもちゃ言語の簡単なプログラムを再帰下降解析によって解析している様子を図示ずる。一番先頭の *prog* が *stat* を呼び出し、「;」をスキップし、*prog* を呼び出す。最初に呼び出される *stat* は次が「{」なので *prog* を呼び出し、「}」をスキップする。というふうに、抽象構文木の構造と再帰手続き群の呼び出し関係がきっちり対応していることが見て取れると思う。

では実際に、構文記号ごとに対応する手続きを見ていこう。まず *prog* から。

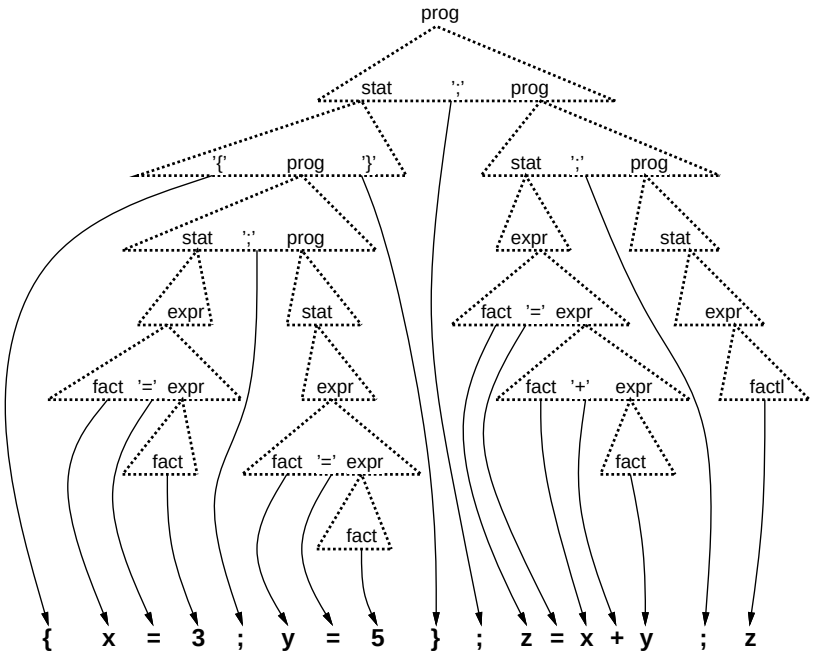

図 3: 再帰下降解析の呼び出し例

```
def prog(sc)
  s = stat(sc); c = sc.peek
  if c == '$' || c == '}' then return s
  elsif c == ';' then sc.next; return Seq.new(s, prog(sc))
  else puts('STAT:' + sc.to_s); return Noop.new
  end
end
```

prog の右辺の先頭はいずれにせよ *stat* なので、まず stat を呼ぶ。次の文字が「$」か「}」ならこれで *prog* は終わりなので (なぜそうなのかは理論的に求まるけどここでは略)、$prog ::= stat$ が適用されて *stat* の木をそのまま返す。次の文字が「;」の場合は $prog ::= stat\ ';'\ prog$ なので、「;」は読み捨てて prog を再帰呼び出しし、その結果の木と先の *stat* の木とを Seq ノードで連結したものを返す。どちらでもない場合は構文エラーなのでエラーメッセージを出力した後「何もしないノード」を返す (このクラスは実はそのために用意したのだった)。次に *stat* を見てみる。

```
def stat(sc)
  c = sc.peek
  if c == '{' then
    sc.next; p = prog(sc)
    if sc.peek != '}' then puts('NO_}:' + sc.to_s); return Noop.new end
    sc.next; return p
  elsif c == 'L' then sc.next; e = expr(sc); return Loop.new(e, stat(sc))
  else return expr(sc)
  end
end
```

これは、まず先頭が「{」である場合は $stat ::= '\{'\ prog\ '\}'$ に対応するので、まず 1 文字進め、prog を呼び、次が「}」であることを確認して (そうでない場合はエラー)、prog から返された木をそのまま返す。先頭が「L」の場合は $stat ::= 'L'\ expr\ stat$ に対応するので、1 文字進め、expr と prog を呼んでこれらを子にもつ Loop ノードを作って返す。それ以外の場合は $stat ::= expr$ に対応するので、expr を呼んでその結果をそのまま返す。次は *expr* を見てみる。

```
  def expr(sc)
    e = factor(sc); c = sc.peek
    if c == '+' then sc.next; return Add.new(e, expr(sc))
    elsif c == '-' then sc.next; return Sub.new(e, expr(sc))
    elsif c == '*' then sc.next; return Mul.new(e, expr(sc))
    elsif c == '/' then sc.next; return Div.new(e, expr(sc))
    elsif c == '=' then sc.next; return Assign.new(e, expr(sc))
    else return e
    end
  end
```

これはどれを選ぶにせよまず最初は $factor$ なので factor を呼び、次に演算子のいずれかがあれば 1 文字進めて expr を呼び、これらを 2 つの子供として持つ適切な演算ノードを作って返す。これら以外の場合は $expr ::= factor$ に対応するので、factor の結果をそのまま返す。最後は $factor$ を見てみる。

```
  def factor(sc)
    c = sc.peek; sc.next
    if c >= 'a' && c <= 'z' then return Var.new(c)
    elsif c >= '0' && c <= '9' then return Lit.new(c.to_i)
    elsif c == '(' then
      e = expr(sc)
      if sc.peek != ')' then puts('NO_):' + sc.to_s); return Noop.new end
      sc.next; return e
    else puts('FACTOR:' + sc.to_s); return Noop.new
    end
  end
```

これは、次の文字が英小文字なら変数ノード、数字なら定数ノードを作って返す。「(」なら $factor ::= '(' expr ')'$ なので expr を呼んでその結果を返すが、対応する「)」がなければエラーとする。これらのどれでもない場合はやはりエラーとする。

4 つの再帰手続きがあって分かりにくいかも知れないが、用はそれぞれの手続きが構文記号になっていて、構文規則に従って呼び出しを進めていく、ということが分かれば自分でも直せるようになると思う。実行テスト用のメソッドも作ってみた。

```
  def test2
#   e = prog(Scanner.new('x=1;y=x+1;y'))
    e = prog(Scanner.new('n=5;x=1;L5{x=x*n;n=n-1};x'))
    puts(e)
    puts(e.exec)
  end
```

やはり階乗の計算をやってみよう。

```
  irb(main):021:0> test2
  (((n$)=(5#));(((x$)=(1#));(((5#)L(((x$)=((x$)*(n$)));((n$)=((n$)-(1#)))));(x$))))
  120
  => nil
```

確かに、先に手で書いたのと同じ構文木ができていて、実行結果も (当り前だが) 同じである。

ここでは「おもちゃの言語」だったが、実際の言語処理系も基本的にはこのような構造でできている。ただし、最終的に実行するやり方が抽象構文木を解釈するインタプリタであるものの他に、マシン語を生成して実行させるもの (コン

パイラ)、マシン語よりは上位のレベルだが中間コードを生成してそれを解釈するインタプリタなどがある。また、近年のコンパイラではコードの性能を向上させるように各種の変形を行う**最適化**に多くの時間を割くようになっている。

あと、今回の処理系は構文には合っているが意味がおかしいプログラムでも実行に入ってしまう。たとえば「`(x+1)=10`」みたいなものでもエラーにならずに動作する。本来ならば構文解析の後、**意味解析**と呼ばれるフェーズがあり、意味的におかしい部分がないかどうかチェックするのが普通である。

演習 7 再帰下降構文解析器を (字句解析器とともに) 打ち込み、おもちゃ言語のプログラムが解析でき実行できることを確認せよ。動いたら、自分が (演習 2 などで) 導入した新しいの機能の構文にも対応させてみよ。

演習 8 ここで示した構文ではすべての演算子の順序が同じである。これを、乗除算が加減算に優先されるように手直ししてみよ (警告しておきますが、簡単ではないです)。

演習 9 面白い言語を設計し、その処理系 (字句解析+構文解析+インタプリタ) を作成して実際にプログラムを書き、面白さについて考察せよ。何が面白いかは各自の判断に任されるものとする。

A　本日の課題 9A

「演習 1」〜「演習 3」で動かしたプログラム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 9A」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. 抽象構文木という考え方を理解しましたか。

   Q2. 「動的分配」「継承」について納得しましたか。

   Q3. 本日の全体的な感想と今後の要望をお書きください。

B　1/11 までの課題 9B

次回までの課題は「演習 1」〜「演習 9」の (小) 課題から 2 つ以上選んで報告することです (9A で提出したものは除き、また木を描くだけの演習 5 も除く)。

各課題のために作成したプログラムは複数ある場合もすべてレポートに掲載してください。レポートは 1/11 の授業開始時刻の **10 分前** までに久野までメールで送付してください。

1. Subject: は「Report 9B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 1 つ目のプログラムのソース。
4. その説明と分析/考察。
5. 2 つ目のプログラムのソース。
6. その説明と分析/考察。
7. 下記のアンケートの回答。

   Q1. プログラミング言語処理系がどのようなものかだいたい分かりましたか。

   Q2. 再帰下降解析器が自分でも書けそうですか。

   Q3. 課題に対する感想と今後の要望をお書きください。