情報科学 2006　久野クラス　＃ 9

久野　靖*

2006.12.22

はじめに

今回は年内最終回ということで、3 回続いた言語処理系の最終回として、残っていた「構文解析」をやります。これで言語処理系が作れるようになりますので、それを冬休み課題とします (腕前に応じて一部でもいいようにしたので、自信のない人でも大丈夫です)。

1　前々回演習問題解説 (残り)

1.1　演習 7:スタック

これはスタックを連結リストで実装してみよという課題で、連結リストに慣れて来ていれば楽勝と思う。図 1 を再掲する。Java のコードは次の通り:

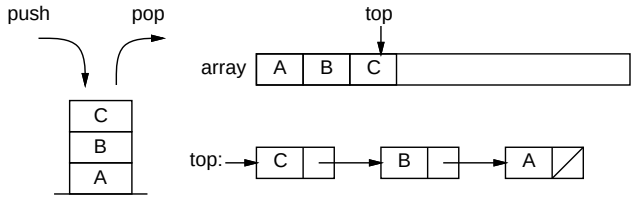

図 1: スタック

```
import java.io.*;

public class r7ex7 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    Stack s = new Stack(1000);
    while(true) {
      System.out.print("> "); String line = in.readLine();
      if(line.equals("q")) { break; }
      else if(!line.equals("")) { s.push(line); }
      else if(!s.isEmpty()) { System.out.println((String)s.pop()); }
    }
  }
  static class Stack {
    Cell top = null;
    public Stack(int s) { }
    public boolean isEmpty() { return top == null; }
    public void push(Object o) { top = new Cell(o, top); }
    public Object pop() { Object o = top.info; top = top.next; return o; }
  }
  static class Cell {
    public Object info;
    public Cell next;
    public Cell(Object i, Cell n) { info = i; next = n; }
  }
}
```

*筑波大学大学院経営システム科学専攻

1.2 演習 8:キュー

キューの方がちょっと面倒かも知れないが、それは「最後に追加していく」ため変数をもう 1 つ持つことと、「空っぽ」の時を特別扱いする必要があることから来る。図 2 も再掲。Java のコードは次の通り:

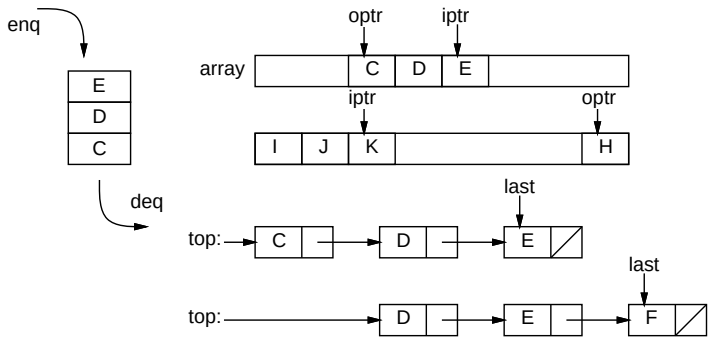

図 2: キュー

```
import java.io.*;

public class r7ex8 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    Queue q = new Queue(1000);
    while(true) {
      System.out.print("> "); String line = in.readLine();
      if(line.equals("q")) { break; }
      else if(!line.equals("")) { q.enq(line); }
      else if(!q.isEmpty()) { System.out.println((String)q.deq()); }
    }
  }
  static class Queue {
    Cell top = null, last = null;
    public Queue(int s) { }
    public boolean isEmpty() { return top == null; }
    public void enq(Object o) {
      if(top == null) { top = last = new Cell(o, null); }
      else { last.next = new Cell(o, null); last = last.next; }
    }
    public Object deq() { Object o = top.info; top = top.next; return o; }
  }
  static class Cell {
    public Object info;
    public Cell next;
    public Cell(Object i, Cell n) { info = i; next = n; }
  }
}
```

ところで、前問もそうだけど題意としては「クラスの使い方は変更せず、中身だけ取り替える」ことを求めていたのに注意。そのため、コンストラクタで大きさ指定の数値を受け取って単に無視している。

1.3 演習 9:構造のたどり

Node というレコード (クラス) を作り、そこに最大 5 個まで「隣」のノードを記録できるようにしておく。また、出発点からの距離 (線の数) も記録できるようにし、最初は-1 にしておく。

そして、駅の個数ぶんの Node を作る。また、名前で指定できないと不便なので名前を指定してそのノードを持って来る関数 nd() を用意し、あと 2 つ名前を指定してそれらの間を「結ぶ」関数 cn() も用意した。これらを使って路線図の接続関係をデータ構造に記録する。記録し終わったら traverse() という関数で出発点から目的地までたどる。路線図を図 3 に再掲しておく。

```
public class r7ex9 {
  static String[] names;
```

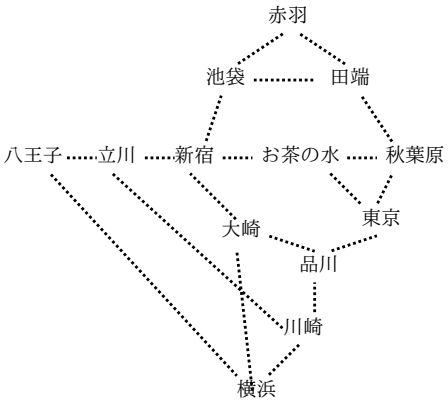

図 3: 路線図

```
static Node[] nodes;
public static void main(String[] args) throws Exception {
  names = new String[]{ "赤羽", "池袋", "田端", "八王子", "立川", "新宿",
    "お茶の水", "秋葉原", "東京", "大崎", "品川", "川崎", "横浜"};
  nodes = new Node[names.length];
  for(int i = 0; i < names.length; ++i) { nodes[i] = new Node(names[i]); }
  cn("赤羽", "池袋"); cn("赤羽", "田端"); cn("池袋", "田端");
  cn("八王子", "立川"); cn("立川", "新宿"); cn("池袋", "新宿");
  cn("新宿", "お茶の水"); cn("お茶の水", "秋葉原"); cn("田端", "秋葉原");
  cn("お茶の水", "東京"); cn("秋葉原", "東京"); cn("東京", "品川");
  cn("新宿", "大崎"); cn("大崎", "品川"); cn("品川", "川崎");
  cn("立川", "川崎"); cn("大崎", "横浜"); cn("川崎", "横浜");
  cn("八王子", "横浜"); traverse1(nd("横浜"), nd("赤羽"));
}
static class Node {
  public String name;
  public Node[] a;
  public int count = 0, dist = -1;
  public Node(String n) { name = n; a = new Node[5]; }
  public void connect(Node n) { a[count] = n; ++count; }
}
static Node nd(String s) {
  for(int i = 0; i < nodes.length; ++i) {
    if(s.equals(nodes[i].name)) { return nodes[i]; }
  }
  return null;
}
static void cn(String s1, String s2) {
  Node n1 = nd(s1), n2 = nd(s2); n1.connect(n2); n2.connect(n1);
}
```

さて、traverse() を見てみよう。まず出発点の距離は 0 とし、ここをスタックに積む。そして、ループの中でスタックから 1 つ降ろして、それを打ち出し、その隣接駅でまだ訪れていないもの (距離が-1 のまま) があれば、距離を現在駅までの距離+1 に設定し、スタックに積む。

```
static void traverse(Node start, Node goal) {
  Stack s = new Stack(100); start.dist = 0; s.push(start);
  System.out.println("START: " + start.name);
  while(!s.isEmpty()) {
    Node n = (Node)(s.pop());
    for(int i = 0; i < n.count; ++i) {
      Node n1 = n.a[i];
      if(n1.dist < 0) {
        n1.dist = n.dist + 1; System.out.println(n1.dist + ":" + n1.name);
        if(n.a[i] == goal) { return; } else { s.push(n1); }
      }
    }
  }
}
```

クラス Stack や、後で出て来るクラス Queue(および必要ならその下請けのクラス) については、既に出て来たものでよいので省略。これを動かした様子を見てみよう。

```
START: 横浜
1:大崎
1:川崎
1:八王子
2:立川
3:新宿
4:池袋
4:お茶の水
5:秋葉原
5:東京
6:品川
6:田端
7:赤羽
```

このように、スタックを使うと深さ優先探索、つまり「行き詰まるまでどんどん先へ行き、行き詰まったら最後の枝分かれまで戻って別の方にまたどんどん行く」探索になるため、到達はできるがえらく無駄な経路を通っている (品川でいちど行き詰まったことが分かる)。ではスタックをキューに取り替えてみる。

```
static void traverse1(Node start, Node goal) {
  Queue q = new Queue(100); start.dist = 0; q.enq(start);
  System.out.println("START: " + start.name);
  while(!q.isEmpty()) {
    Node n = (Node)(q.deq());
    for(int i = 0; i < n.count; ++i) {
      Node n1 = n.a[i];
      if(n1.dist < 0) {
        n1.dist = n.dist + 1; System.out.println(n1.dist + ":" + n1.name);
        if(n1 == goal) { return; } else { q.enq(n1); }
      }
    }
  }
}
```

これの実行結果は次の通り。

```
START: 横浜
1:大崎
1:川崎
1:八王子
2:新宿
2:品川
2:立川
3:池袋
3:お茶の水
3:東京
4:赤羽
```

キューを使うと幅優先探索になるので、横浜から距離 1 のものをまず全部調べ、次に距離 2 のものを全部調べ、…となるため、必ず最短の距離が求まる。演習 10 以降は略させてください。

2 文脈自由文法の構文解析

2.1 生成文法の復習＋α

構文解析は文法へのあてはめなので、前々回の生成文法の復習から入らせていただく。

- 端記号 a、b、… の集合 T
- 非端記号 A、B、…の集合 N
- 出発記号 S ($S \in N$)
- α、β を $T \cup N$ の要素の列、X を非端記号として、$\alpha \rightarrow \beta$ の形の生成規則の集合 P

を合わせた $G = (T, N, S, P)$ を生成文法、そのうち生成規則の左辺を 1 個の非端記号に制約したものを文脈自由文法と言うのだった。さらに、列 s に生成規則を 1 回適用して列 t ができるとき $s \Rightarrow t$、0 回以上任意回適用して列 t ができるとき $s \Rightarrow^* t$ と書くのだった。そして、$S \Rightarrow^* \alpha$ なる α で、端記号のみから成る列のことを G が定める言語と呼び、$L(G)$ とも表記するのだった。ここで、今回新たに少しだけ定義を追加する。記号 X について、集合 $First(X)$ と $Follow(X)$ を次のように定める。

> $Follow(X)$ とは、$S \Rightarrow^* \alpha\ X\ \beta$ なる β (非端記号の列) について、その先頭の端記号の集合を意味する。また、β が空列になる場合には特別な記号\$もその要素に加える。
>
> $First(X)$ とは、$X \Rightarrow^* \alpha$ なる α (非端記号の列) について、その先頭の端記号の集合を意味する。また、$X \Rightarrow^* \varepsilon$ (ε は空列) であるなら、$Follow(X)$ の要素すべてを加える。

要するに、$Follow(X)$ とは「その文法において α の次に来る可能性がある端記号の集合」である (\$は終わりを表し、\$が $Follow(X)$ に含まれる場合は X が言語の文の最後に来ることがあることを表す)。そして、$First(X)$ とは「その文法において X の先頭に来る可能性がある端記号の集合」である (ただし X が空列になり得る場合は、その後にある記号が「透けて」見える)。

たとえば簡単な例として $S \to ASB, S \to \varepsilon, A \to a, B \to b$ を考えると、各非端記号の First と Follow は次のようになる:

$First(A) = \{a\}$, $Follow(A) = \{a, b\}$.
$First(B) = \{b\}$, $Follow(B) = \{b, \$\}$.
$First(S) = \{a, b, \$\}$, $Follow(S) = \{b, \$\}$.

$First(S)$ がなんでこうなの? と思うかも知れないが、S は ε になり得るので、その場合は $Follow(S)$ の内容も加えるわけだ。

演習 1 以下の文法において、各非端記号の $First$、$Follow$ を書き出せ。なお「(」なども普通の 1 文字として扱う。

a. $S \to aSa, S \to bSb, S \to c$
b. $S \to aAbS, S \to c, A \to aA, A \to \varepsilon$
c. $S \to aT, S \to (S)T, T \to +ST, T \to \varepsilon$

なお、ここまで $First$ は 1 つの非端記号について考えてきたが、後の話の都合上、次のように拡張する。まず、端記号 a について、$First(a) = \{a\}$。次に記号列 $\alpha = A_1 A_2 \cdots A_n$ について、$First(\alpha)$ は次のものとする。

- $First(A_1) \Rightarrow^* \varepsilon$ でなければ、$First(A_1)$ と同じ。
- $First(A_1) \Rightarrow^* \varepsilon$ であれば、$First(A_1) \cup First(A_2 \cdots A_n)$。
- 以下同様にして、一番最後まで ε になるようであれば、$First(A_1) \cup \cdots First(A_n) \cup Follow(A_n)$。

この場合でもやはり、記号列 α についてその一番最初に来る端記号の集合が $First(\alpha)$ であり、$\alpha \Rightarrow^* \varepsilon$ であれば「透けて見える」ためにその最後の要素の $Follow$ も加える、ということ。

2.2 下向き構文解析

前にも述べたように、構文解析とは入力列の文法への当てはめであり、これはひらたく言えば「構文木を作る」処理だと考えればよい。それで、皆様は前に演習で構文木を描いたとき、「上から」木を作りましたか、それとも「下から」作りましたか?

一般に構文解析において、上から下に向かって木を作って行く方法を**下向き構文解析** (top-down parsing)、下から上向きに木を作って行く方法を**上向き構文解析** (bottom-up parsing) という。人間がやる場合は「分かるところから」やるのでいいのだが、プログラムでやる場合はその手順が明確になっていないとプログラムにできない。ここでは、比較的分かりやすい下向き解析を取り上げる。

下向き構文解析の概観を図 4 に示す。(1) が最初の状況で、一番上に S、一番下に解析する入力列がある (↑が次の入力の位置を示す)。

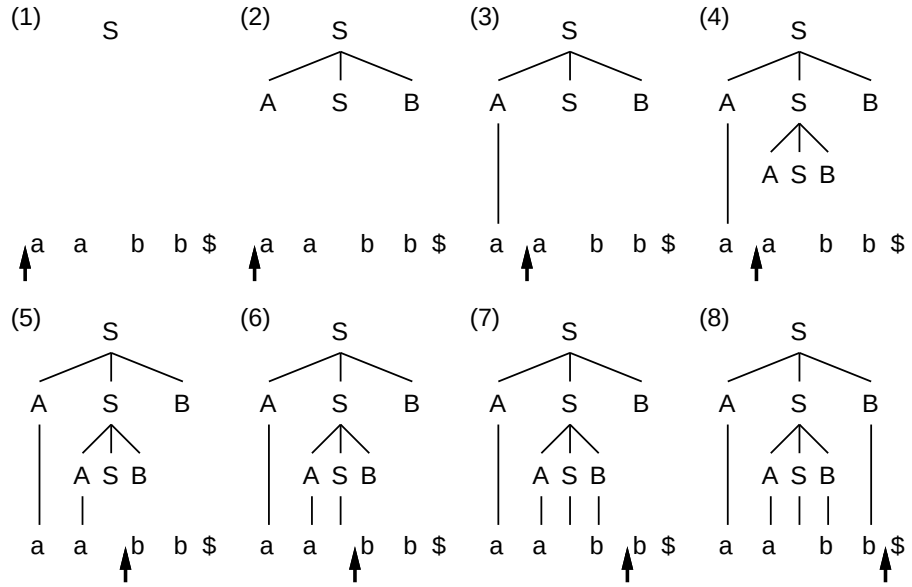

図 4: 下向き構文解析

(1) まず S から始める。S を左辺に持つ規則は $S \to ASB$ と $S \to \varepsilon$ の 2 つであり、前者を適用する場合は $First(X)$ として a が出て来る。また、後者を適用すると (ε になるから)$First(X)$ として b と \$ が出て来る。ということは、次の入力 ($\uparrow$ の位置) を見て、それが a なら 1 番目、b か \$ なら 2 番目の規則を選択すればよい。ここでは次は a なので 1 番目を選び、S の下に ASB を展開して (2) のようになる。

(2) 以下展開してできたものを左から順に処理るので、次は A に対応する部分木を作る。A を左辺に持つ規則は $A \to a$ だけで、対応する $First$ は a。次の入力も a なので、これらの間を結び、入力を 1 つ進めて (3) の状態になる。

(3) 次は 2 段目の S に対応する部分木だが、これも上と同じにして ASB に展開し、(4) の状態とする。

(4) 次は展開してできた A の部分木だが、これも上と同じにして a と対応させ、入力を進めて (5) の状態になる。

(5) 次は 3 段目の S に対応する部分木だが、今度は次の入力が b であるので、$S \to \varepsilon$ が対応する。ε は空列だから、入力は進めずに S の処理は終わり、(6) の状態になる。

(6) 次は 3 段目の B に対応する部分木だが、B を左辺とする規則は $B \to b$ だけで、対応する $First$ は b。そこでこれを B と対応づけて入力を進め、(7) の状態になる。

(7) 次は残っていた 2 段目の B に対応する部分木だが、上と同様にして次の b と対応させて入力を進め、(8) の状態になる。これで、すべての部分木が完成し、入力は \$ まで来たので、これで OK ということになる。

2.3 再帰下降解析

前記のような動作をすべてデータ構造を操作するプログラムで実現して行くこともできるが、ここでは比較的分かりやすい方法として、**再帰下降解析** (recursive descent parsing) を説明する。その前に、前記のように文字列の「入力位置」の文字を調べたりその位置を進めたりする処理が必要なので、それを `CScan` というクラスで用意した。

- `CScan(文字列)` — 指定した文字列を順に調べるための `CScan` オブジェクトを初期化するコンストラクタ。
- `boolean match(文字列)` — 指定した文字列のどれかが入力位置にあれば `true`、それ以外なら `false` を返す。
- `boolean fwd(文字)` — 入力位置が指定した文字なら入力位置を 1 つ進めて `true` を返す。そうでなければ `false` を返す。
- `String toString()` — 現在位置が分かるような形で保持している文字列を表示。

さて再帰下降解析だが、非端記号ごとにそれと同名のメソッドを作る。たとえば S に対応して `S()` を作る。パラメタは `CScan` オブジェクトを 1 つ取り (仮に名前は `s` とする)、返値は `boolean` で構文の解析に成功したか否かを返す。

本体の中では、次の入力に応じて、その非端記号 X を左辺とする規則それぞれについてその右辺を α として、次の入力が $First(\alpha)$ に含まれるなら、その規則を選ぶ (α が ε の場合は次の入力が $Follow(X)$ に含まれるなら、その規則を選ぶ)。どれも選べないなら失敗なので `false` を返す。

選べた場合は、α が ε なら空列に対応していて入力を進める必要はないのですぐ成功にして `true` を返す。それ以外であれば、α の各要素について左から順に以下を行う。

- 非端記号の場合 — その非端記号に対応するメソッドを呼ぶ。
- 端記号の場合 — `s.fwd('` その文字`')` を呼ぶ。

どこかで `false` が返ってきたらそこで失敗と分かるので、自分自身もすぐ `false` を返して終わる。それには、上記の呼び出しを`&&`でつなげた形で書いておけばよい。

全体としては、たとえば規則が $X \to bAc$ であれば (右辺の $First$ は「b」なので) 次のように対応づけられる。

```
if(s.match("b")) { return s.fwd('b') && A(s) && s.fwd('c'); }
```

return 文の中は、まず次の文字が「(」であるかチェックして NO なら直ちに `false` を返し、YES なら次は A(s) を呼ぶがその結果が NO なら直ちに `false` を返し、YES なら「)」のチェックに進む、というふうに動作するわけである。

実際にプログラムを見てみよう。なお、`main()` は出発記号のメソッドを呼び出すだけである (ただし出発記号が成功した場合でも次が「$」(終わり) であることはチェックする)。

```
public class R9Sample1 {
  public static void main(String[] args) {
    CScan s = new CScan(args[0]);
    System.out.println((S(s) && s.match("$")) + ": " + s);
  }
  static boolean S(CScan s) {
    if(s.match("a")) { return s.fwd('a') && S(s) && s.fwd('b'); }
    if(s.match("b$")) { return true; }
    return false;
  }
  static class CScan {
    String str; int ptr;
    public CScan(String s) { str = s + "$"; ptr = 0; }
    public boolean match(String s) { return s.indexOf(str.charAt(ptr)) >= 0; }
    public boolean fwd(char c) {
      if(str.charAt(ptr) == c) { ++ptr; return true; }
      else                     { return false; }
    }
    public String toString() {
      return str.substring(0,ptr)+"!"+str.substring(ptr);
    }
  }
}
```

これを動かしている様子を示す。

```
% java R9Sample1 aabb
true: aabb!$
% java R9Sample1 aaabbb
true: aaabbb!$
% java R9Sample1 aabbb
false: aabb!b$
% java R9Sample1 aaabb
false: aaabb!$
% java R9Sample1 abab
false: ab!ab$
```

確かに文法に合った文字列だけ `true` となり、また合わない場合は正しくないことが分かる最小限の位置までしか入力を読まないことが分かる。

演習 2 上のプログラムを打ち込んで動かせ。動いたら文法を次のものに変更してみよ。なお「(」なども普通の 1 文字として扱うが、このような記号を含んだ文字列をコマンド行で指定するときは「`'...'`」で囲む必要があるので注意。

a. $S \to aSa, S \to bSb, S \to c$

b. $S \to aAbS, S \to c, A \to aA, A \to \varepsilon$

c. $S \to aT, S \to (S)T, T \to +ST, T \to \varepsilon$

2.4 LL(1) 文法

ここまでの説明では触れなかったが、上で説明した方法で「どのような文脈自由文法でも解析できる」わけではないことはなんとなく分かると思う。具体的には、次の条件が必要である。

- すべての X について、X を左辺とする生成規則のうちからどれを選ぶかが曖昧でなく決められる。具体的には、それらの規則の右辺部分の $First$ が互いに共通部分を持たない。
- $X \Rightarrow^+ \alpha$ (1 回以上の生成規則による置き換えによってできる α) について、α の先頭に X が現れる (これを**左再帰**と呼ぶ) ことがない。

1 番目の規則は、どの規則を選ぶかを決められなければどうしょうもないので当然のことである。2 番目の規則は、X から下向きに木を作っていって、また左端に X が現れたら、さっきと同じ状態だから「堂々めぐり」に陥ることになるからである。たとえば $E \to E+T$ という規則があると、$E \Rightarrow E+T \Rightarrow E+E+T \Rightarrow E+E+E+T \Rightarrow E+E+E+E+T\cdots$ のようになって止まらなくなるわけだ。

これらのどちらも、文法を書き換えることによって避けられる「場合がある」。たとえば $X \to A+B, X \to A-B$ という規則があると、その先頭部分は A で共通だから、次の入力を見てどちらの規則を選択したらいいかは知ることができない。これは新しい端記号 X' を追加して $X \to AX', X' \to +B, X' \to -B$ のようにすれば、今度は X' の 2 つの規則からどちらを選ぶかは先頭が $+$、$-$ のどちらかを見ればよいのだから問題ない。

同様に、$E \to E+T, E \to T$ という左再帰を含む規則の組があったとして、これから生成されるものは要するに「$T+T+\cdots+T$」なわけだから、新しい非端記号 E' を導入して $E \to T+E', E' \to +E', E' \to \varepsilon$ のように書き換えることができる。ただし、他の規則とのからみもあって「どんな文法でも常に」等価な LL(1) 文法に直せるわけではない。

演習 3 次の文法を等価な LL(1) 文法に変換し、それを認識する再帰下降解析プログラムを (演習 2 と同様に) 構成せよ。

a. $E \to (E+E), E \to (E-E), E \to (E), E \to a$

b. $E \to E+a, E \to E-a$

c. $E \to E+T, E \to E-T, T \to T*F, T \to T/F, F \to a, F \to (E)$

3 プログラミング言語処理系

3.1 字句解析器

ここまでの例題では、入力文字列を「1 文字ずつ」処理してきたが、前回述べたように実際の言語処理系では「名前」「数値」などの単位は「かたまり」で認識する。そのとき、構文解析器には「名前」「数値」などの印だけを返すとしても、「どんな名前/数値の文字列だったか」という情報は抽象構文木を構築するとき (当然) 必要である。また、プログラムは見やすさのため字下げや改行が使ってあるので、これらの改行や空白は無視する動作も必要である。

これらの処理は有限オートマトンをもとに行うこともできるが、ここでは普通に文字列を処理するプログラムとして実現例を示す。次に示すクラス `TokScan` は使い方は (ほぼ) 先の `CScan` と同じだが、ファイルからプログラミング言語のソースファイルを読み込んで動作するようになっている。

インスタンス変数としてはファイルを読み込むための `BufferedReader` オブジェクト `in`、現在読み込んでいる行の文字列を格納する `line`, 名前や数値の場合にその文字列を入れておく変数 `str`、次の記号の種別を表す文字 `cur`、`line` 中のどの位置を次に見るかを示す整数 `ptr`、`line` の長さを保持しておく整数 `length`、`line` がファイルの何行目なのかを保持しておく整数 `lineno` がある。

```
static class TokScan {
  BufferedReader in;
  String line, str; char cur = ' ';
  int ptr = 0, length = 0, lineno = 0;
  public TokScan(String fname) throws Exception {
    in = new BufferedReader(new FileReader(fname)); advance();
  }
  public char peek() { return cur; }
  public boolean match(String s) { return s.indexOf(cur) >= 0; }
  public String getStr() { return str; }
  public boolean fwd(char ch) {
    if(ch != cur) { return false; } else { advance(); return true; }
  }
```

```
  public void advance() {
    if(cur == '$') { return; }
    while(ptr<length && Character.isSpaceChar(line.charAt(ptr))) { ++ptr; }
    while(ptr >= length) {
      try {
        if((line = in.readLine()) == null) { in.close(); cur='$'; return; }
      } catch(Exception ex) { cur = '$'; }
      ptr = 0; ++lineno; length = line.length();
      while(ptr<length && Character.isSpaceChar(line.charAt(ptr))) { ++ptr; }
    }
    if(Character.isLetter(line.charAt(ptr))) {
      int i = ptr+1;
      while(i < length && Character.isLetterOrDigit(line.charAt(i))) { ++i; }
      str = line.substring(ptr, i); ptr = i; cur = 'v';
      if(str.equals("loop")) { cur = 'l'; }
      if(str.equals("read")) { cur = 'r'; }
      if(str.equals("print")) { cur = 'p'; }
    } else if(Character.isDigit(line.charAt(ptr))) {
      int i = ptr+1;
      while(i < length && Character.isDigit(line.charAt(i))) { ++i; }
      str = line.substring(ptr, i); ptr = i; cur = 'i';
    } else {
      cur = line.charAt(ptr); ++ptr;
    }
  }
  public String toString() {
    if(line == null) { return "#" + lineno; }
    else { return lineno+": "+line.substring(0,ptr)+"!"+line.substring(ptr); }
  }
}
```

大多数のメソッドはごく簡単なものであり、ほとんどの処理は「次のかたまりに進む」メソッド `advance()` の中で行っている。その中は非常にごちゃごちゃしているようだが、粗筋としては次の通り。

- `$`はおしまいの印なので、`cur` が`$`なら何もしない (先に進む動作は行わない)。
- 見ている文字が空白文字である間 `ptr` を進める。
- `ptr` が `line` のおしまいまで来たら次の行を読む。このときエラーがあれば `cur` に`$`をセットしておしまいということにする。
- ちゃんと読めたら `ptr` 等を設定し、再度空白を読み飛ばす。while 文になっているので、全部空白の行ならさらに次の行を読む。
- 繰り返しを抜けたら、次の文字は空白ではない。
- 次の文字が英字なら、それは名前ということになるので、英数字が続く間 `ptr` を進め、英数字の列を `str` に保存する。かたまりの種別はとりあえず v(変数) にするが、もしも loop、read、print であれば l、r、p にとりかえる。
- 次の文字が数字なら、それは数値ということになるので、数字が続く間 `ptr` を進め、数字列を `str` に保存し、種別を i にする。
- それ以外の文字なら何らかの記号なのでその文字を `cur` に設定する。

ここで面白いのは、名前が来た時にそれが loop、read、print なら種別を決め打ちで変更してしまうことである。このような形で特別扱いするような名前のことを**予約語** (reserved word) と呼ぶ。予約語は変数名などに使うことはまったくできないのに注意。課題で「独自の言語」を作る時に必要ならここを増やして予約語を追加すること。

3.2 簡単な言語の認識器

字句解析ができたので、これを再帰下降解析と組み合わせて簡単なプログラミング言語の構文を認識するプログラムを作ってみよう。なお、**認識器** (recognizer) とは構文が合っているかどうかをチェックするだけのプログラムであり、**解析器** (parser) は構文木や抽象構文木を組み立てるまでを行うプログラムのこと。なので、これまでに出て来たサンプルもすべて実は認識器だった。認識する言語の LL(1) 文法を示しておく。

```
Program ::= StList
StList ::= Stat StList | ε
Stat ::= AssignStat | ReadStat | PrintStat | LoopStat | { StList }
AssignStat ::= 変数 = Exp ;
ReadStat ::= read 変数 ;
PrintStat ::= print Exp ;
LoopStat ::= loop ( Exp ) 文
Exp ::= Term Exp1
Exp1 ::= + Term Exp1 | - Term Exp1
Term ::= Fact Term1
Term1 ::= * Fact Term1 | / Fact Term1
Fact ::= 変数 | 数値 | ( Exp )
```

Exp のあたりの文法は先の演習問題の例解になっていますので。「変数」とか「数値」とかは字句解析器から返される種別なので文法では定義されていないことに注意。再帰下降解析器は既に学んだとおりでただ文法の量が多いというだけ。

```
import java.io.*;

public class R9Sample2 {
  public static void main(String[] args) throws Exception {
    TokScan s = new TokScan(args[0]);
    System.out.println((StList(s) && s.match("$")) + ": " + s);
  }
  static boolean StList(TokScan s) {
    if(s.match("vrpl{")) { return Stat(s) && StList(s); }
    if(s.match("}$")) { return true; }
    return false;
  }
  static boolean Stat(TokScan s) {
    if(s.match("v")) { return AssignStat(s); }
    if(s.match("r")) { return ReadStat(s); }
    if(s.match("p")) { return PrintStat(s); }
    if(s.match("l")) { return LoopStat(s); }
    if(s.match("{")) { return s.fwd('{') && StList(s) && s.fwd('}'); }
    return false;
  }
  static boolean AssignStat(TokScan s) {
    if(s.match("v")) { return s.fwd('v')&&s.fwd('=')&&Exp(s)&&s.fwd(';'); }
    return false;
  }
  static boolean ReadStat(TokScan s) {
    if(s.match("r")) { return s.fwd('r') && s.fwd('v') && s.fwd(';'); }
    return false;
  }
  static boolean PrintStat(TokScan s) {
    if(s.match("p")) { return s.fwd('p') && Exp(s) && s.fwd(';'); }
    return false;
  }
  static boolean LoopStat(TokScan s) {
    if(s.match("l")) {
      return s.fwd('l') && s.fwd('(') && Exp(s) && s.fwd(')') && Stat(s);
    }
    return false;
  }
  static boolean Exp(TokScan s) {
    if(s.match("vi(")) { return Term(s) && Exp1(s); }
    return false;
  }
  static boolean Exp1(TokScan s) {
    if(s.match("+")) { return s.fwd('+') && Term(s) && Exp1(s); }
    if(s.match("-")) { return s.fwd('-') && Term(s) && Exp1(s); }
    if(s.match(");")) { return true; }
    return false;
  }
  static boolean Term(TokScan s) {
```

```
    if(s.match("vi(")) { return Fact(s) && Term1(s); }
    return false;
  }
  static boolean Term1(TokScan s) {
    if(s.match("*")) { return s.fwd('*') && Fact(s) && Term1(s); }
    if(s.match("/")) { return s.fwd('/') && Fact(s) && Term1(s); }
    if(s.match("+-);")) { return true; }
    return false;
  }
  static boolean Fact(TokScan s) {
    if(s.match("v")) { return s.fwd('v'); }
    if(s.match("i")) { return s.fwd('i'); }
    if(s.match("(")) { return s.fwd('(') && Exp(s) && s.fwd(')'); }
    return false;
  }
  static class TokScan //... 上記の通り
}
```

演習 4 この例題はさすがに長いので、Web からファイルを取れるようにしました。取り寄せてコンパイルし、「簡単な言語 (文法は上記)」のソースファイルを自分で作って「java R9Sample2 ファイル名」のようにして動かしてみなさい。正しい構文、間違った構文の両方とも試すこと。

3.3 意味スタックと抽象構文木の組み立て

前節までで構文のチェックはできたが、解釈実行やコード生成に進むためには抽象構文木を作る必要がある。そのためには通常、構文解析と並行してスタックを使用する。このスタックを**意味スタック** (semantic stack) と呼ぶ。

意味スタックは図 5 のように、コードを読み進むごとにその時点で作ることのできる部分木を順に詰めて行く。そして、新しいノードは「変数や整数に対応するものを積む」か、「スタックから 1 個または 2 個降ろしてそれを子供とする新しいノードを積む」形で作られて行き、最終的にはプログラム全体に対応する抽象構文木ができあがる。

では実際に、構文を解析し、抽象構文木を作り、直ちに解釈実行する処理系を見てみよう。`main()` ではスタックを新たに用意し、再帰下降解析の手続きにはすべて `TokScan` と `Stack` を渡すようにする。解析に失敗したらエラーメッセージを出すが、成功したらスタックから `Tree` オブジェクトを取り出して来て実行する。

```
import java.io.*;

public class R9Sample3 {
  static BufferedReader in;
  static int[] var = new int[26];
  public static void main(String[] args) throws Exception {
    in = new BufferedReader(new InputStreamReader(System.in));
    TokScan s = new TokScan(args[0]);
    Stack t = new Stack(100);
    boolean result = StList(s,t) && s.match("$");
    if(!result) {  System.out.println("syntax: " + s); }
    else { Tree t1 = (Tree)t.pop(); t1.exec(); }

  }
```

スタックの操作は上で述べたように、「1 つか 2 つ降ろして新しいノードを作って積む」ので、それぞれ用の手続きを用意した。`boolean` 値を返すようにして常に `true` を返しているのは、解析手続きの「`&&`」の連鎖の中にはさみ込めるようにするため。

```
  static boolean st1(Stack t, char op) {
    Object t1 = t.pop(); t.push(new Tree(op,t1));
    return true;
  }
  static boolean st2(Stack t, char op) {
    Object t2 = t.pop(), t1 = t.pop(); t.push(new Tree(op,t1,t2));
    return true;
  }
```

解析手続き群は適宜スタックの操作を挿入しているだけ。AssignStat、ReadStat、Fact では変数や整数が現れた時に新たなノードを積む処理が必要だが、ちょうど `TokScan` から v や i が返されている時に `getStr()` を呼ばないと正しい変数名や数値の文字列が得られないことに注意する必要がある。`st1()` や `st2()` を呼ぶのはもっと単純に挿入すればいいだけ。あと、StList で ε になるときは「最後」を表す `null` を積む必要がある。

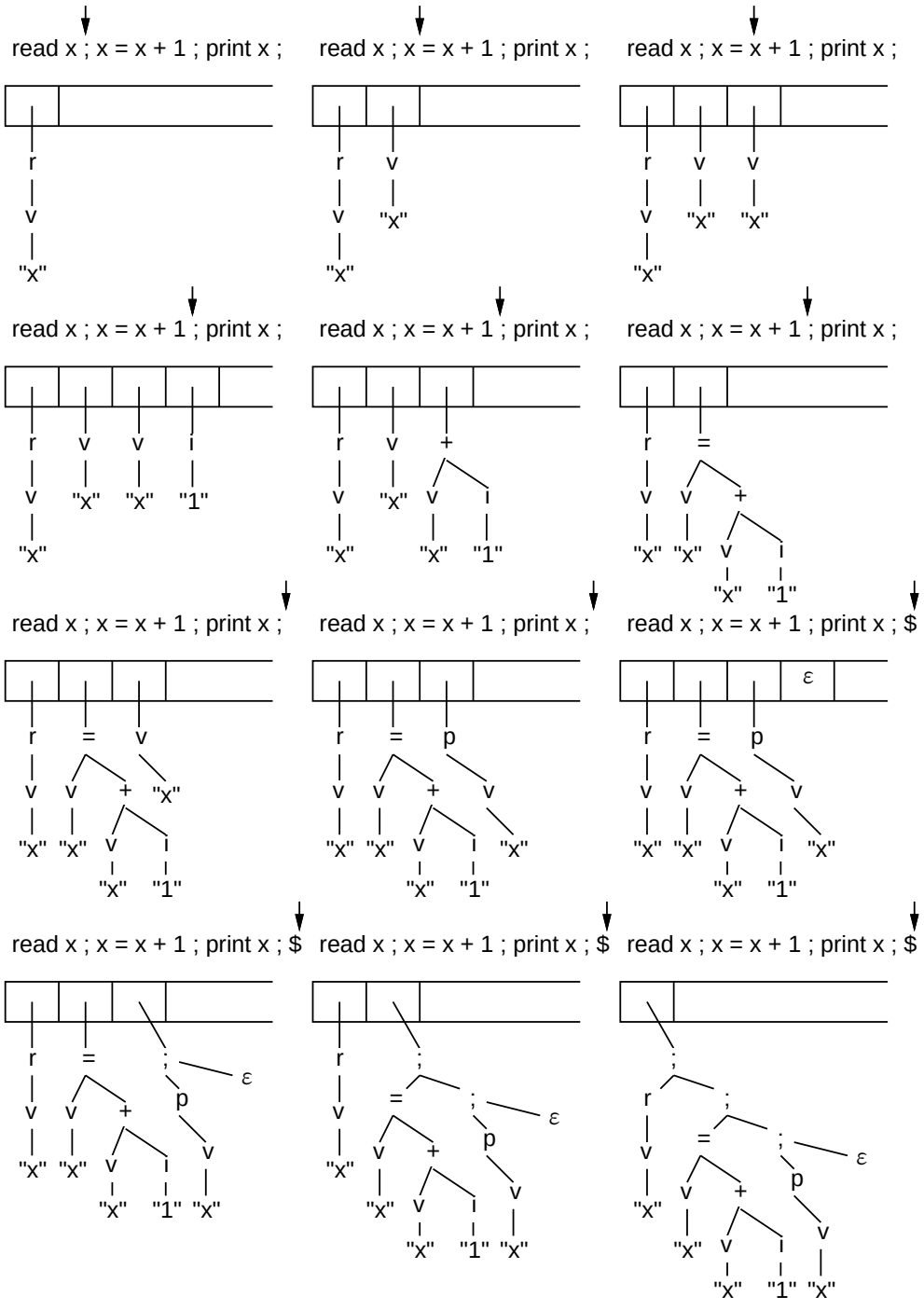

図 5: 意味スタックの動作

```
static boolean StList(TokScan s, Stack t) {
  if(s.match("vrpl{")) { return Stat(s,t) && StList(s,t) && st2(t,';'); }
  if(s.match("}$")) { t.push(null); return true; }
  return false;
}
static boolean Stat(TokScan s, Stack t) {
  if(s.match("v")) { return AssignStat(s,t); }
  if(s.match("r")) { return ReadStat(s,t); }
  if(s.match("p")) { return PrintStat(s,t); }
  if(s.match("l")) { return LoopStat(s,t); }
  if(s.match("{")) { return s.fwd('{') && StList(s,t) && s.fwd('}'); }
  return false;
}
static boolean AssignStat(TokScan s, Stack t) {
  if(s.match("v")) {
    t.push(new Tree('v', s.getStr()));
    return s.fwd('v') && s.fwd('=') && Exp(s,t) && s.fwd(';') && st2(t,'=');
  }
  return false;
}
static boolean ReadStat(TokScan s, Stack t) {
  if(s.match("r")) {
    s.fwd('r'); t.push(new Tree('v', s.getStr()));
    return s.fwd('v') && s.fwd(';') && st1(t,'r');
  }
  return false;
}
static boolean PrintStat(TokScan s, Stack t) {
  if(s.match("p")) {
    return s.fwd('p') && Exp(s,t) && s.fwd(';') && st1(t,'p');
  }
  return false;
}
static boolean LoopStat(TokScan s, Stack t) {
  if(s.match("l")) {
    return s.fwd('l') && s.fwd('(') && Exp(s,t) &&
           s.fwd(')') && Stat(s,t) && st2(t,'l');
  }
  return false;
}
static boolean Exp(TokScan s, Stack t) {
  if(s.match("vi(")) { return Term(s,t) && Exp1(s,t); }
  return false;
}
static boolean Exp1(TokScan s, Stack t) {
  if(s.match("+")) { return s.fwd('+')&&Term(s,t)&&st2(t,'+')&&Exp1(s,t); }
  if(s.match("-")) { return s.fwd('-')&&Term(s,t)&&st2(t,'-')&&Exp1(s,t); }
  if(s.match(");")) { return true; }
  return false;
}
static boolean Term(TokScan s, Stack t) {
  if(s.match("vi(")) { return Fact(s,t) && Term1(s,t); }
  return false;
}
static boolean Term1(TokScan s, Stack t) {
  if(s.match("*")) { return s.fwd('*')&&Fact(s,t)&&st2(t,'*')&&Term1(s,t); }
  if(s.match("/")) { return s.fwd('/')&&Fact(s,t)&&st2(t,'/')&&Term1(s,t); }
  if(s.match("+-;")) { return true; }
  return false;
}
static boolean Fact(TokScan s, Stack t) {
  if(s.match("v")) { t.push(new Tree('v',s.getStr())); return s.fwd('v'); }
  if(s.match("i")) {
    t.push(new Tree('i',new Integer(s.getStr()))); return s.fwd('i');
  }
  if(s.match("(")) { return s.fwd('(') && Exp(s,t) && s.fwd(')'); }
  return false;
}
```

Stack、Tree、TokScan は既に出て来たもの。だたし Tree の exec() については-、*、/と loop 文も実行できるように追加した (これも演習の例解を兼ねている)。

```
  static class Stack {
    Object[] a; int ptr;
    public Stack(int s) { a = new Object[s]; ptr = -1; }
    void push(Object o) { ++ptr; a[ptr] = o; }
    Object pop() { Object o = a[ptr]; --ptr; return o; }
    Object top() { return a[ptr]; }
    int size() { return ptr; }
  }
  static class Tree {
    char op; Object left, right;
    public Tree(char o, Object l, Object r) { op = o; left =l; right = r; }
    public Tree(char o, Object l) { op = o; left =l; }
    public int exec() throws Exception {
      if(op == ';') {
        ((Tree)left).exec(); if(right != null) { ((Tree)right).exec(); }
      } else if(op == 'i') {
        Integer i = (Integer)left; return i.intValue();
      } else if(op == 'v') {
        String s = (String)left; return var[s.charAt(0)-'a'];
      } else if(op == 'r') {
        String s = (String)(((Tree)left).left); System.out.print("input> ");
        var[s.charAt(0)-'a'] = new Integer(in.readLine()).intValue();
      } else if(op == 'p') {
        System.out.println(((Tree)left).exec());
      } else if(op == 'l') {
        int count = ((Tree)left).exec();
        for(int i = 0; i < count; ++i) { ((Tree)right).exec(); }
      } else if(op == '=') {
        String s = (String)(((Tree)left).left);
        return var[s.charAt(0)-'a'] = ((Tree)right).exec();
      } else if(op == '+') {
        return ((Tree)left).exec() + ((Tree)right).exec();
      } else if(op == '-') {
        return ((Tree)left).exec() - ((Tree)right).exec();
      } else if(op == '*') {
        return ((Tree)left).exec() * ((Tree)right).exec();
      } else if(op == '/') {
        return ((Tree)left).exec() / ((Tree)right).exec();
      }
      return 0;
    }
    public String toString() {
      if(op == 'i' || op == 'v') { return ""+left; }
      String l = ""; if(left != null) l = ((Tree)left).toString() + " ";
      String r = ""; if(right != null) r = " " + ((Tree)right).toString();
      return "(" + l + op + r + ")";
    }
  }
  static class TokScan //... 前と同じなので略
}
```

これで簡単な言語が実行できる言語処理系が完成したわけだ。実行させてみよう。

```
  % cat test.prog
  read x;
  n = 0;
  r = 1;
  loop(x) {
    n = n + 1;
    r = r * n;
  }
  print r;
  % java R9Sample3 test.prog
  input> 10
  3628800
  %
```

演習 5 この例題も Web からファイルを取れるようにしました。取り寄せてコンパイルし、「簡単な言語 (文法は上記)」のプログラムを実行させてみなさい。

4 おまけ:プログラミング言語のさまざまな構文

課題のヒントになるかも知れないので、ここで「おまけ」としてさまざまな言語のさまざまな構文について思い付くままに紹介してみよう。そもそも、ここで説明しているような、文脈自由文法 (BNF) による構文規則で構文を定義した言語の最初のものは Algol 60 という言語である (1960 年生まれなのでもう 50 年くらい前ですね!)。

この言語では代入に「:=」を使い、「=」は等しいという意味に使っている。この方がずっといいと思うのだが… また、各種記号の代わりに予約語を多く使う。Java の流儀は C 言語から来ているのだが、C 言語は短く簡潔に書ける代わりに構文は読みづらいという側面がある。

たとえば while 文で 2 つの流儀を比べると次のような感じになる。

```
while n > 10 do begin r := r * n; n := n - 1 end;  (Algol)
while(n > 10) { r = r * n; n = n - 1; }            (Java)
```

つまり Algol では while の条件は () で囲む代わりに do までの間に書き、また複数の文をまとめるブロックには begin-end を使っている。さらに、「;」を文どうしの区切りとして使っているのも特徴 (end の直前には「;」が要らない)。C 言語系でもパラメタ等は「`func(a, b, c)`」のように「`,`」を区切りとして挿入していることを考えれば、「;」も文の終わりとして使うより文の区切りとして使う方が統一的に思える。なお、上の例では end の後に「;」があるが、これは次に別の文が置かれることを想定していて、次が end の場合は不要ということになる。

また、for 文の構文もまったく違う。

```
for i := 1 step 1 until 10 do begin ... end;       (Algol)
for(i = 1; i <= 10; ++i) { ... }                   (Java)
```

for 文はもともと、変数を一定ずつ増やしながら/減らしながらループするものなので、while 文のように一般的な条件が指定できてしまう C 系列の流儀はあまり普通ではない。ところで、ここまでループとしては while と for だけを扱って来たが、実は 3 番目のループとして「末尾判定型ループ」というものが多くの言語に (そして Java にも) 存在する。Pascal と Java で条件の向きが反対 (ループを終わる条件なのか、続ける条件なのかが違う) なのに注意。

```
repeat x := x + 1; r := r * x until x >= 10;    (Pascal)
do { x = x + 1; r = r * x } while(x < 10);      (Java)
```

なんだかよく分からない感じだけど、フローチャートで示すと図 6 のようになっているわけだ (言語によって予約語が違うので名前がつけにくいが、要はループの末尾で条件を判定するので、1 回は本体を実行してしまうわけだ。久野個人としては、別にこれを知らなくても困らないと思っているので (自分ではまず絶対使わない)、説明を省略しました。

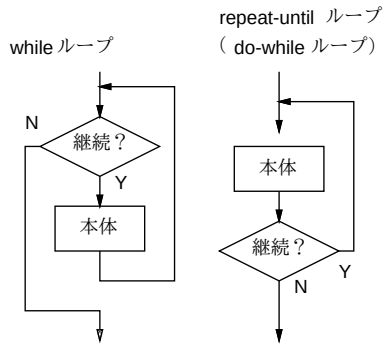

図 6: 末尾判定型ループ

ここまでループ文の説明だったが、if 文についても Algol 系は予約語を使うところが違っている。

```
if x = 0 then begin
  ...
end else if x > 0 then begin
  ...
end else begin
  ...
end;
```

こういうのを見るとC系統の { ... } の方が書きやすいかも知れない。end を行頭に書きたくない人はこういう感じで書いたりもしていた。

```
if x = 0 then
  begin
   ...
  end
else if x > 0 then
  begin
   ...
  end
else
  begin
    ...
  end;
```

さて、if 文や while 文の後に文が 1 つだけであれば {...} や begin...end は不要なのだが、1 つだけと 2 つ以上とで切替えるのは嫌だとも思える。(そのため久野の例ではすべて必ず {...} で囲んであった)、また、この方式には次のような弱点がある。

```
if(x == 0)
  if(y == 0)
    z = 1;
else
  y = 1;
```

この「y=1;」はいつ実行されるだろうか? 正解は「x が 0 で、かつ y が 0 のとき」だが、こういうのは分かりにくいですね? 常に囲むようにすれば、次のようにはっきりどちらかが分かるようになる。

```
if(x == 0) {          if(x == 0) {
  if(y == 0) {          if(y == 0) {
    z = 1;                z = 1;
  }                     } else {
} else {                  y = 1;
  y = 1;                }
}                     }
```

Perl などはこの方針で、常に {...} をつけること強制している。予約語を使う言語の場合も、常に end をつけて次のように書くものがある。

```
while x < 10 do r := r * x; x := x + 1 end;
for i := 1 to 10 do r := r * i end;
if x < 0 then x := -x; y := 1 else y := -1 end;
```

このような構文の言語では必ず end がつくのがわずらわしいという意見もある。なお、全部 end ではよくないと、終わりのキーワードを個別に変えているものもある。

```
while x < 10 do r := r * x; x := x + 1 od;
for i := 1 to 10 do r := r * i od;
if x < 0 then x := -x; y := 1 else y := -1 fi;
```

このような構造の言語の場合、else-if の連鎖は次のようにそれ専用の特別な構文になる (elif は else if の略。また elsif と書く言語もある)。左側は CLU(Algol 系)、右側は Perl(C 系) の場合。

```
if x = 0 then      if(x = 0) {
  ...                ...
elif x < 0 then    } elsif(x < 0) {
  ...                ...
else               } else {
  ...                ...
end                }
```

Perl の場合、これだったら Java でもいいのではという気になるけど。

これらについて、やっていることはどれも同じなので構文などどうでもいいという意見もあるが、構文は見た目に影響し、見た目によって考えやすさが結構違うこともあるので、簡単な問題ではない。皆様もせっかく構文を自分で決めて言語処理系を作る方法を学んだわけなので、よかったら自分独自の工夫などしてみて頂けると私としては嬉しいと思う。以上、雑談でした。

A　本日の課題 9A

「演習 2」または「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. 「演習 2」または「演習 3」で動かしたプログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

   Q1. 「再帰下降解析」による構文解析の仕組みが分かりましたか。

   Q2. 言語処理系における、プログラム言語ソースから実行までの流れが分かりましたか。

   Q3. 本日の全体的な感想と今後の要望をお書きください。

B　冬休み課題 9B

冬休みの課題のテーマは「プログラミング言語処理系において、自分独自の工夫を行う」ことです。工夫の例としては、次のようなものが考えられます (あくまで例示であり、これに限るわけではありません):

- 独自の構文ないし機能を追加する。たとえば例題プログラムにあった loop 文のような変わった (普通の言語にない) ものでもよいですし、おまけの節にあったような構文の工夫でもよいです。
- Java 言語 (やその他の知っている言語) にあるけれど例題では実現されていなかったものを実現する (配列とか手続きなどもチャレンジしがいがあると思います)。または例題の手抜きな制約 (変数名は先頭 1 文字しか見ない等) を改良するとかでもよいです。
- 処理系の性能や適用範囲におけるさまざまな工夫を行う。解釈実行系であれば処理を速くするための工夫をする、変換系であれば生成されるコードの改良や i386 以外のマシンのアセンブリコード生成など。

処理系全部を扱うのが大変であれば「ソースから構文認識部分まで (構文チェックがちゃんとできる段階まで)」「抽象構文木から実行まで (抽象構文木は前回のようにソースプログラム中で自分で組み立てる)」など一部だけを扱うのでも構いません。その範囲でも上記の工夫の多くは試して見られます。

実力以上のことを要求するつもりはありません。やる気なら非常に高度なことまでチャレンジできる課題ですが、それに幻惑されて高度すぎる (期限内に作れない) 課題にチャレンジしないように注意しておきます。プログラミングは「できることから徐々に、動く状態を維持したまま、徐々に改良していく」のがコツですから。

むしろ今回は、紙のレポートを提出していただくので、プログラムはほどほどでいいので、「レポートをきちんと書く」という点に重点を置いてください。レポートは A4 版の紙を縦づかいとし、必ず綴じること。さらに、次の内容構成とすること。

1. 表紙。表紙には表題「Report9B」、学籍番号、氏名、提出日付のみを書くこと。
2. 課題の再掲。この場合は「言語処理系の工夫をおこなう」ですね。
3. 方針の説明。自分がその課題をどのようにしてこなそうと計画したかの、方針を書く。
4. 回答。この場合は Java プログラムのプリントアウトと、その中の要点を説明したもの。また、言語処理系なので「ソースプログラム例」や「実行例」も入れてください。
5. 考察。この課題をやってどんなことが分かったか、まだ疑問な点は何かを書く。
6. 感想。「おまけ」として、ここまでの授業の内容や進め方についてどうだったか、良かったこと、悪かったことなど自由に記述してください。ついでに、授業におおむね出席した/しなかったの別と、そのことは良かった/悪かったとかも。
7. 付録 (あれば)。たとえば、あなたの「新言語」で大量のプログラムを書いてみた場合は、本文で多すぎるようなら付録に書いたものをつけるなどが考えられます。

レポートの〆切は 2 月 2 日正午まで、提出場所は演習室前のレポートボックスとなります。では頑張ってください。(なお、1/26 は講義休講ですが、TA さんに出勤してもらってプログラムの直し相談をやっていただきます。おおよそその時までにはプログラムが書けていないと、紙のレポートを書く時間が足りないと思います。)