情報科学 2006　久野クラス　＃7

久野　靖*

2006.12.8

はじめに

前回の有限オートマトンはわりと評判よかったようで、幸いでした。でも残りの材料が分量多すぎだったと思うので、今回はもうちょっと減らすようにします。今回もオートマトンの続きぽい話題と動的データ構造の続きぽい話題ですが。

- 生成文法、文脈自由文法
- スタックとキュー — 汎用的な抽象データ型

今回の演習問題解説はオートマトンのが多いですが、いちいち Java のプログラムにしても読みづらいだけなのでほとんどはグラフの図までにしておきます。多くなるので表と検索の解説は最後に。

1　非決定性オートマトン再び

頂いたご意見の中に、「非決定性オートマトンが分からない」というのが複数ありましたので、ちょっとだけ補足説明をします。ε 遷移と一緒に説明したのがよくなかったですね。非決定オートマトン (NFA) というのは「行き先が一通りに決まらない」オートマトンという。図 1 の上のようなものがその例である。なぜなら、初期状態で「a」が来たとき、右へ行くのか斜め上へ行くのか決まらないですから。

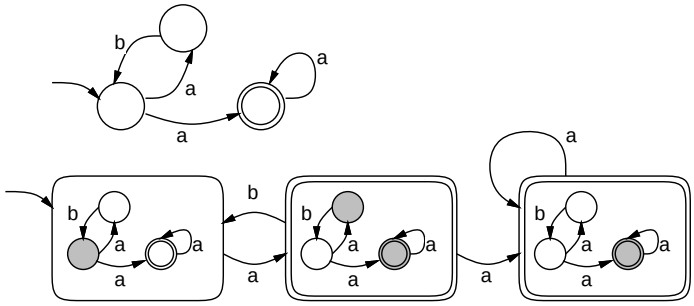

図 1: 非決定性オートマトンの例

で、これを決定性オートマトン (DFA) に直すためには、元の非決定性オートマトンの「状態の集合」を考えます。もっとひらたくいうと、「どことどこにいる可能性があり得るか」を考えるわけ。すると、初期状態だけにいて (図下左)、「a」が来ると 2 つの行き先両方に行く可能性があるので、その「両方にいる」状態に移る (図下中)。ここで「b」が来ると、初期状態に戻るしか可能性はない。また「a」が来ると、元の NFA では最終状態に行くしか可能性がない (図下右)。これらをまとめると、図 1 下のような決定性オートマトンができるわけだ。なお、この DFA の状態のうち「元の NFA の最終状態が含まれている」ものはすべて最終状態となる。元の NFA とこの DFA できっかり同じ列が受理されることを確かめて見て欲しい。

次に、ε 遷移を説明しよう。ε 遷移とは、「空文字列に対する遷移」つまり何も文字が来なくても通れる矢線を意味する。何も文字がなくても通れるのだから、そこはいつでも通れ、通っても通らなくてもよい。したがって、ε 遷移があ

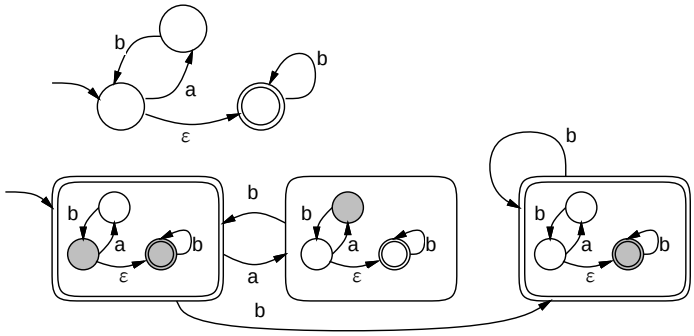

図 2: ε 遷移を持つ非決定性オートマトンの例

るオートマトンは必ず非決定性になる (通る場合と通らない場合の 2 通りがあり得るから)。たとえば図 2 上では、初期状態からいきなり ε 遷移が出ているので、空文字列でも「OK」になる。

これを決定性オートマトンにするとき、初期状態は下の 2 つの状態両方を含む集合 (状態) ということになる。ここで「a」が来たときは、上の状態だけを含む集合 (状態)、「b」が来たときは右の状態だけを含む集合 (状態) に遷移する。上だけを含む状態からは「b」が来たときに初期状態に戻る。右だけ含む状態からは「b」が来たとき自分自身に戻る。従って図 2 下のようなオートマトンができる。こちらも、NFA と DFA を比べてみてきっかり同じ列が受理されることを確かめてみて頂きたい。で、これを一般にどうやって作るかについて、前回資料で説明していたわけだ。

2 演習問題解説

2.1 演習 1

これはオートマトンを用意するところだけ示せば十分ですね。

(a) 「a が 0 個以上、続いて b、続いて a が 0 個以上」

```
Automata atm = new Automata(2, 1, new int[]{2});
atm.trans(1, "a", 1); atm.trans(1, "b", 2); atm.trans(2, "a", 2);
```

(b) 「まず a があり、その後 ba または aa が 0 個以上」

```
Automata atm = new Automata(3, 1, new int[]{2});
atm.trans(1, "a", 2); atm.trans(2, "b", 1);
atm.trans(2, "a", 3); atm.trans(3, "a", 2);
```

(c) 「まず a があり、その後 bb、aa が交互に続く (0 個でもいい)」

```
Automata atm = new Automata(4, 1, new int[]{2,4});
atm.trans(1, "a", 2); atm.trans(2, "b", 3);
atm.trans(3, "b", 4); atm.trans(4, "a", 1);
```

2.2 演習 2

これはオートマトンだけ示しますね (図 3)。

2.3 演習 3

これも結果の決定性オートマトンだけ示しますね (図 4)。ちなみに演習 4 は大変なので省略させてください。すみません。

*筑波大学大学院経営システム科学専攻

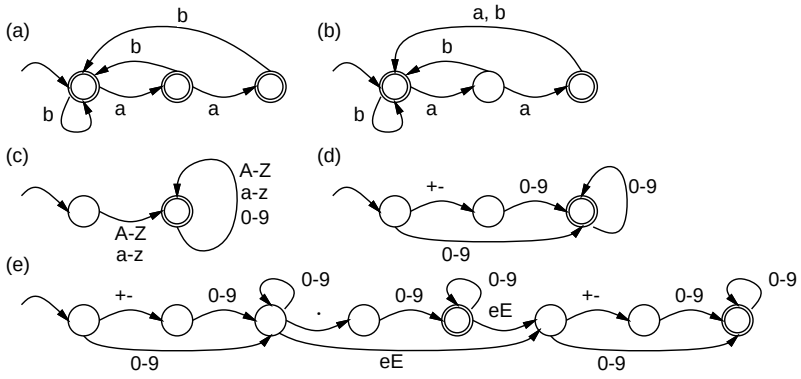

図 3: 演習 2 の回答例

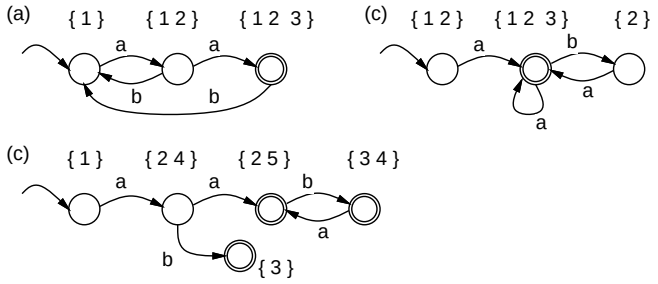

図 4: 演習 3 の回答例

2.4 演習 5

課題にあったような操作をするためには、ポインタだけ見ているのではつらいところがある。そこで、「現在何行目にいるか」「全部で何行あるか」も常に覚えておき、それに基づき「何行目に行く」関数 go() を作ってみた。これを利用することで多くの操作が素直に実現できる。あと、print() の機能が変わるためこれまでの print()(現在行を表示) を print1() という名前で残した。まず main() 部分はこれまで通り枝分かれするだけで、コマンドが増えたぶん枝分かれも増えている。

```
import java.io.*;

public class r6ex5 {
  static Cell head, tail, prev, cur;
  static int curnum, allnum;
  public static void main(String[] args) throws Exception {
    head = new Cell(""); tail = new Cell("EOF"); head.next = tail; top();
    curnum = 1; allnum = 1;
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    while(true) {
      System.out.print(">");
      String line = in.readLine();
      if(line.equals("") || line.charAt(0) == 'n') { next(); print1(); }
      else if(line.charAt(0) == 'b') { back(); print1(); }
      else if(line.charAt(0) == 't') { top(); print1(); }
      else if(line.charAt(0) == 'p') { print(line.substring(1)); }
      else if(line.charAt(0) == 'i') { insert(line.substring(1)); }
      else if(line.charAt(0) == 'd') { delete(); print1(); }
      else if(line.charAt(0) == 'x') { exch(); print1(); }
      else if(line.charAt(0) == 's') { subst(line.substring(1)); print1(); }
      else if(line.charAt(0) == 'r') { read(line.substring(1)); }
      else if(line.charAt(0) == 'w') { write(line.substring(1)); }
      else if(line.charAt(0) == 'q') { break; }
      else                           { System.out.println("?"); }
    }
  }
```

go() は top() してから現在行が指定番号になるまで next() するだけ。back() は現在行-1 に go() するだけ。

```
  static void go(int n) { top(); while(curnum < n) { next(); } }
  static void back() { if(curnum > 1) { go(curnum-1); } }
  static void print1() { System.out.println("  " + cur.str); }
```

print() は番号指定がないときは print1() を呼ぶだけ。あるときはその番号を整数にし、先頭から最後までたどりながら、現在行が指定範囲内なら表示。最後に元の行に戻る。

```
  static void print(String s) {
    if(s.equals("")) { print1(); return; }
    int n = new Integer(s).intValue(), c = curnum;
    for(top(); curnum < allnum; next()) {
      if(c-n <= curnum && curnum <= c+n) { print1(); }
    }
    go(c);
  }
```

top() 等はこれまで通りだが加えて行番号と行数を適切に維持する。

```
  static void top() { prev = head; cur = head.next; curnum = 1; }
  static void next() { if(cur!=tail){prev=cur;cur=cur.next;++curnum;} }
  static void insert(String s) {
    prev.next = new Cell(s); prev = prev.next; prev.next = cur;
    ++curnum; ++allnum;
  }
  static void delete() {
    if(cur != tail) { prev.next=cur.next;cur=prev.next;--allnum; }
  }
```

exch() はひたすらポインタ操作による入れ換え。

```
  static void exch() {
    if(cur != tail && cur.next != tail) {
      Cell p=cur.next; prev.next=p; cur.next=p.next; p.next=cur; cur=p;
    }
  }
```

subst() は区切り「/」の位置をまず調べて検索文字列と置き換え文字列を取り出し、現在行中に検索文字列があればそれを置き換える。

```
  static void subst(String s) {
    int[] a = new int[s.length()]; int count = 0;
    for(int i = 0; i < s.length(); ++i) {
      if(s.charAt(i) == '/') { a[count] = i; ++count; }
    }
    if( count != 3 || a[0] != 0 || a[2] != s.length()-1) { return; }
    String s1 = s.substring(a[0]+1, a[1]), s2 = s.substring(a[1]+1, a[2]);
    for(int i = 0; s1.length()+i <= cur.str.length(); ++i) {
      if(cur.str.substring(i, i+s1.length()).equals(s1)) {
        cur.str = cur.str.substring(0,i)+s2+cur.str.substring(i+s1.length());
        return;
      }
    }
  }
```

ファイルの読み書きは前回のヒントにあった通り。

```
  static void read(String s) throws Exception {
    BufferedReader in = new BufferedReader(new FileReader(s));
    String line;
    while((line = in.readLine()) != null) { insert(line); }
    in.close();
  }
  static void write(String s) throws Exception {
    PrintWriter out = new PrintWriter(new FileWriter(s));
    int c = curnum; top();
    while(curnum < allnum) { out.println(cur.str); next(); }
    go(c); out.close();
  }
  static class Cell {
    public String str; Cell next;
    public Cell(String s) { str = s; }
  }
}
```

3　生成文法と構文木

3.1　生成文法の枠組み

皆様は「文法」というと何を思い浮かべますか。英文法? それとも Java の構文規則だろうか。以下ではチョムスキー (言語学者だが社会思想家としても有名 — Wikipedia などで調べてみてください) が考案した**生成文法**という枠組みについて解説する。前回やった正則 (正規) 表現、Java の文法を記述するのに使った BNF などもこの枠組みの 1 要素に相当する。

以下では文法を扱う上で、個々の言語の文字 (漢字だとか英字だとか) と独立に検討するために、一般の記号 a、b、… の集合 T を考える。T の要素のことを**端記号**と呼ぶ。

また、文法の構造を説明する上で「副詞節」だとか「名詞句」のような中間概念を表すものも必要である。これを A、B、…の集合 N とする。N の要素は**非端記号**と呼ぶ。N の要素のうち特別なもの S (**出発記号**) があり、これが「文」を表す。

そして、両方を合わせたもの、つまり $T \cup N$ の要素を**アルファベット**と呼ぶ。アルファベットの列を α、β 等で表す。

最後に、生成規則の集合 P がある。P の要素は一般に $\alpha \to \beta$ の形をしている。これは、α に一致するアルファベット列があったら、それを β に書き換え「てもよい」ことを意味する。生成文法 G はこの 4 つの組 (T, N, S, P) で定まる。

一般に、$s = \gamma\ \alpha\ \delta$、$t = \gamma\ \beta\ \delta$、$\alpha \to \beta \in P$ のとき、s の中の α を β に置き換えることで t ができる。これを $s \Rightarrow t$ と書く。置き換えを 1 回でなく 0 回以上任意回行って s から t ができる場合、$s \Rightarrow^* t$ と書く。

生成文法 $G = (T, N, S, P)$ が定める**言語** $L(G)$ とは、$S \Rightarrow^*\ \alpha$ なる α で、T の要素のみから成るもののの集合を言う。$L(G)$ の要素のことをその言語の**文**とも呼ぶ。

簡単な例を示そう。$T = \{a, b\}$、$N = \{S, A, B\}$、$P = \{S \to ASB, S \to AB, A \to a, B \to b\}$ とする。この文法はどういう言語を定義しているだろうか? いくつか例を挙げてみよう。

$S \Rightarrow AB \Rightarrow aB \Rightarrow ab$
$S \Rightarrow ASB \Rightarrow AABB \Rightarrow^* aabb$
$S \Rightarrow ASB \Rightarrow AASBB \Rightarrow AAAABB \Rightarrow^* aaabbb$

つまりこれは、「a が N 個あって、その後に b が同じく N 個ある」という言語なのだった (前回やったように、有限オートマトンではこれは指定できない)。

演習 1 以下の文法はどのような言語を定義しているか考えよ (簡単のため生成規則だけ記す)。

a. $S \to aA, A \to bB, A \to b, B \to aA$
b. $S \to aSa, S \to bSb, S \to cSc, S \to \varepsilon$
c. $S \to aSBC, S \to aBC, CB \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc$

3.2　生成文法のクラス

生成文法の定義は上で述べた通りだが、生成規則のカタチに制約を加えることでいくつかのクラス (タイプ) に分類される。

- Type 0 — 上の定義通り。とくに制約なし。**チューリングマシン** (計算モデルの 1 つで、一般のコンピュータと同等の計算能力を持つ) で受理が計算できるクラスと呼ぶこともある。
- Type 1 — すべての文法規則が $\alpha\ X\ \beta \to \alpha\ \gamma\ \beta$ の形をしているもの。つまり、基本的に非端記号 X を γ に置き換えるが、ただしそれは前後に α と β がくっついているときだけ、という形。Type 1 文法のことを**文脈依存文法**とも呼ぶ。
- Type 2 — すべての文法規則が $X \to \gamma$ の形をしているもの、いいかえれば規則の左辺が非端記号 1 個だけのもの。これは上の Type 1 の α と β のような制約がなく、X があればそれをいつでも γ に置き換えてよい (文脈を考慮しなくてよい) ことから**文脈自由文法**と呼ぶ。BNF による定義は左辺の記号が 1 つなので文脈自由文法。
- Type 3 — すべての文法規則が $A \to a$ または $A \to aB$ の形をしているもの。つまり右辺が端記号 1 個、または端記号 1 個+非端記号 1 個のもの。これを**正則文法**ないし**正規文法**と呼ぶ。

条件を見れば分かるように、これらは Type 0 ⊃ Type 1 ⊃ Type 2 ⊃ Type 3 という包含関係になっている。ちなみに、演習 1 の文法は (a) が Type 3、(b) が Type 2、(c) が Type 1(そのままだとそうと分からないが変形により Type 1 の形にできるらしい)。

ところで、Type 3 文法が正則 (正規) 文法と呼ばれるゆえんは、正則 (正規) 文法が定める言語は有限オートマトンや正則 (正規) 表現で定めることができ、またその逆も言えるから。つまりこれら 3 つの表現能力は等しい。

前回、正則 (正規) 表現を同じ言語を受理する非決定性オートマトンに変換でき、非決定性オートマトンを決定性オートマトンに変換できることを示した。その続きとして、決定性オートマトンからそれと同じ言語を定める正則 (正規) 文法が作れることを示そう。それは簡単で、次のようにする。

1. オートマトンの各状態に非端記号を割り当てる。初期状態には S を割り当てる。
2. オートマトンで状態 A から B に記号 a を持つ矢線があれば、対応して $A \to aB$ という文法規則を追加する。
3. オートマトンで状態 A から◎の状態 (受理状態) に記号 a を持つ矢線があれば、対応して $A \to a$ という文法規則を追加する。

正則 (正規) 文法で作られるアルファベット列は常に一番最後にだけ非端記号があるから、それがオートマトンの状態に対応し、状態を移るごとに対応する端記号が非端記号の直前に追加されるようになっているので、最後に $A \to a$ を適用してすべてが端記号になったときにできている列はオートマトンで矢線をたどった時の記号の列と同じになっているわけだ。実は上記演習 1 の (a) は図 5 を上の方法で正規文法に変換したものである。

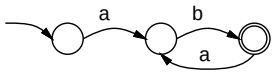

図 5: 正規文法に対応する決定性オートマトン

3.3 構文木

プログラミング言語の構文定義に使われるのは文脈自由文法 (Type 2) なので、以下ではこれに絞って扱っていく。文脈自由文法では、左辺が 1 個の非端記号、右辺が記号列なので、これを図 6 のように木の形に書き表すことができる。そして、これを「くっつけて行く」ことで、S から始めて端記号の列 (つまりその言語の文) までの導出列を 1 つの木で表すことができる。これを**構文木** (syntax tree) と呼ぶ。

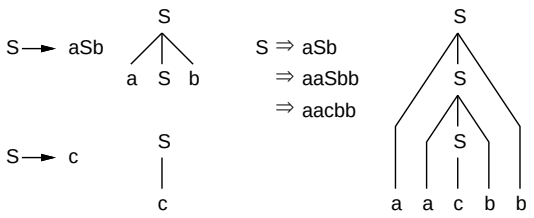

図 6: 文脈自由文法と構文木

構文木は通常、まず文 (端記号の列) を与えて、S からその文を生成するような構文木を組み立てるという形で使う。なぜかというと、そうすることで「この文全体の形はどういう部分から成っているか」が分かるようになるから。この過程のことを**構文解析** (syntax analysis) と呼ぶ。

演習 2 文法として $S \to a + S, S \to a, S \to (S)$ を考える (ただし、「+」「(」「)」も端記号として扱う)。次の文に対する構文木を描け。

a. 「a+a+a+a」
b. 「(((a)))」
c. 「(a+a)+(a+(a+a))」

3.4 式の文法と構文木

では、我々が普段お世話になっている Java 言語の**式** (expression) の構文を考えてみよう。たとえば次のものはどうだろうか。「i」は定数、「v」は変数名を表す端記号とする。

$$E \to i,\ E \to v,\ E \to E\ \texttt{+}\ E,\ E \to E\ \texttt{-}\ E,\ E \to E\ \texttt{*}\ E,\ E \to E\ \texttt{/}\ E$$

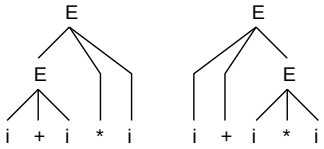

図 7: あいまいな文法の構文木

しかしこれではよろしくない。なぜかいうと、この文法ではたとえば「`i + i * i`」という文に対して図 7 のように複数の構文木が作れてしまう。このように、構文木が一意に決まらない場合のある文法のことを**曖昧**である、という。これまでに出て来た文法はすべて曖昧さは無かったことに注意。

ところで、図 7 の 2 つの構文木のうち、どちらがより「望ましい」か分かりますか? それは「右側」です。なぜかというと、「*」は「+」より先に計算するべきだから、それが「より近くくっついている」ようでないと処理するとき都合がよくないから。つまり、我々が使う計算式の場合、次の性質がある。

1. `*`、`/`は`+`、`-`より強く結び付く。
2. 同じ強さの演算どうしの場合は、左側が先に結び付く。

じゃあ、計算の順番を変えたい時は? そりゃ「`( )`」を使う。これがうまく実現できるような文法が必要なわけである。

実はそれは次の文法でできる。

$$E \to E\ \texttt{+}\ T,\ E \to E\ \texttt{-}\ T,\ E \to T,\ T \to T\ \texttt{*}\ F,\ T \to T\ \texttt{/}\ F\ T \to F,\ F \to i,\ F \to v,\ F \to (E)$$

ここで E は式 (expression)、T は式の中の項 (term)、F は項の中の因子 (factor) に相当している。この文法だと、曖昧さ無く上記 (1)(2) の条件にあてはまる構文木だけができることを確認していただく。

演習 3 文法として上記のものを前提とし、次の式に対する構文木を描け。ただし数値定数は i、変数名は v に相当するものとする。

a. 「`3 * x + y`」
c. 「`x + y + 4 * z`」
c. 「`(x + 1 ) / (y - 2)`」

3.5 抽象構文木、式木

このようにしてあいまいさのない式の文法はできたが、その構文木はいかにも長ったらしい (図 8 左に演習 3(a) の構文木を示した)。

構文解析の中途ではこうなるのはやむを得ないが、それが終わってプログラムを内部で扱っているときはもっと簡単化した木構造を使う。具体的には、「E → T」などの導出は省いてしまい、あと演算子などは枝分かれのところに書く。これを構文木の必要な部分だけ取り出したという意味で**抽象構文木** (abstract syntax tree) という。また、式の抽象構文木のことを**式木** (expression tree) ともいう。

さて、待ちくたびれたと思うがようやく今回の Java プログラムに進む。その前に 1 つだけ。これまで Java でオブジェクトはそのオブジェクトの型の変数だけに入れて来た。しかし、`Object` という特別なクラスの変数を作ると、そこには任意のオブジェクト値を入れることができる。

```
Object x = new Integer(999);
Object y = "ABC";  // String もオブジェクトだから
```

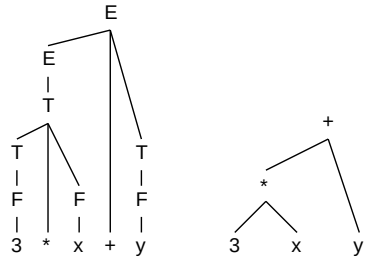

図 8: 構文木と抽象構文木 (式木)

しかし、Object 型が持っているメソッドはごく少数なので、たいていの場合は元の型に戻して来て扱う必要がある。そのときは**キャスト**が必要になる (これは、数値の場合の「広い型→そのまま、狭い型→キャスト」というのと類似している)。

```
Integer i = (Integer)x;
String s = (String)y;
```

このとき、**もともとオブジェクトへのキャストでないと失敗する**ので注意。たとえば変数 x には Integer オブジェクトが入っているので、それを String にキャストしようとしたらエラーになる。

では本題で、今回は式木のノード (中間節) を Tree クラスのオブジェクト/レコードで表す。Tree クラスはそれが何の演算 (等) であるかをあらわす op という char 型のフィールド、そして左と右の子をあらわす left と right というフィールドを持つ。で、left と right はそこにさまざまなオブジェクトが入れられるように上記の Object 型になっている。図 9 に図 8 の式木を Tree レコードで表したものを示す (斜線が引いてあるのは null が入っていることをあらわす)。[1]

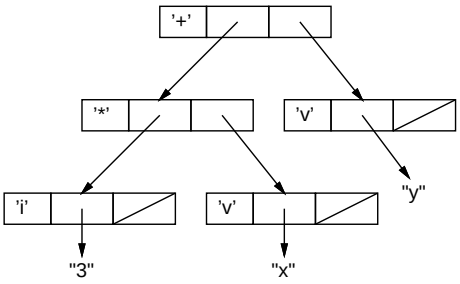

図 9: 動的データ構造による式木の表現

これを用いて、図 8(図 9) の式木を内部に作り、それを文字列として打ち出すプログラムを示す。Tree の初期化のためのコンストラクタは子を 1 つだけ持たせる場合用と 2 つ持たせる場合用の 2 つを用意した。

```
public class R7Sample1 {
  public static void main(String[] args) {
    Tree t =
      new Tree('+',
        new Tree('*', new Tree('i', "3"), new Tree('v', "x")),
        new Tree('v', "y"));
    System.out.println(t.toString());
  }
  static class Tree {
    public 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 String toString() {
      if(op == 'i' || op == 'v') { return (String)left; }
      String l = ""; if(left != null) l = ((Tree)left).toString() + " ";
      String r = ""; if(right != null) r = " " + ((Tree)right).toString();
      return "(" + l + op + r + ")";
    }
  }
}
```

[1] 「'+'」のように「'」で囲んだものは char 型 (1 文字) の値、「"ABC"」のように「"」で囲んだものは String オブジェクトであることに注意。

打ち出すメソッド `toString()` は再帰手続きになっている。整数と変数の場合は `left` にその文字列を入れてあるので、それを取り出して `String` にキャストして返す。そうでない場合は、左と右の子について `toString()` を呼んでそれを `l` と `r` に入れるが、呼ぶために `left` と `right` を一旦 `Tree` にキャストし戻して、それから呼んでいる (実はこの場合は戻さなくても呼べるがその説明をしだすと大変なのでご勘弁を)。実行すると次のようになる。

```
% java R7Sample1
((3 * x) + y)
```

演習 4 上のプログラムをそのまま打ち込んで動かせ。動いたら図 10 の式木に対する構造を作って打ち出すように直してみよ。

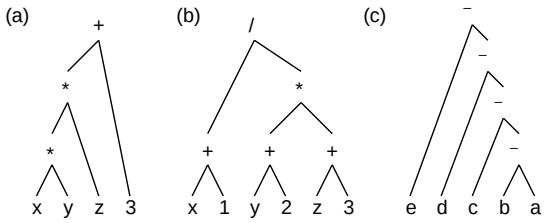

図 10: いくつかの式木

3.6 前置記法、中置記法、後置記法

式木のようなものを文字列として表記するときの記法として、次のものがある。

- **中置記法** (infix notation) — 演算記号を被演算子どうしの間に置く。
- **前置記法** (prefix notation) — 演算記号を被演算子の前に置く。ポーランド記法と呼ぶこともある (ポーランドの人が考案したらしい)。
- **後置記法** (postfix notation) — 演算記号を被演算子の後に置く。逆ポーランド記法と呼ぶこともある (ポーランドの人が逆立ちして考案したらしい — 嘘です)。

普段われわれが使っている「`x + y`」のようなものは中置記法に相当する。前置記法だと「`+ x y`」、後置記法だと「`x y +`」になる。もう少し込み入った例を示そう。

```
((x + 3) * (y - 2))  --- 中置記法
(* (+ x 3) (- y 2))  --- 前置記法
((x 3 +) (y 2 -) *)  --- 後置記法
```

ところが、前置記法や後置記法はかっこを取ってしまっても (演算子がいくつ被演算子を取るかが分かっていれば…上の例ではすべて 2 個ずつ取るわけで…) 曖昧さなく演算ができる。

```
* + x 3 - y 2  --- 前置記法
x 3 + y 2 - *  --- 後置記法
```

演算順序が違ったらどうなるか見てみよう。

```
x + 3 * y - 2  --- 中置記法、*を先に計算する
- + x * 3 y 2  --- 前置記法
x 3 y * + 2 -  --- 後置記法
```

なお、日本人には後置記法が読みやすいとされている。それは 1 番目の場合「x と 3 を足し、y から 2 を引き、それらを掛ける」2 番目の場合「x に、3 に y を掛けたものを足し、2 を引く」のように、日本語では動詞を最後に置くから。

なお、前置記法、後置記法、中置記法に対応する出力 (処理) の順番のことを**行きがけ順** (preorder)、**帰りがけ順** (postorder)、**通りがけ順** (inorder) と呼ぶこともある。通りがけ順は 2 分木にのみ意味がある (「左の子→処理→右の子」の順だから。枝が 1 個とか 3 個とかだとどこに「処理」を入れるか困るでしょう?)。

演習 5 先のプログラムを改造して、出力を前置記法にせよ。また後置記法にしてみよ。いずれも、かっこ有りと無しと両方試して読みやすさを考えてみよ。

演習 6 式だけではなく他の文も含めた次のような簡単な言語を考えてみる (式は前記の文法の式であるものとする)。

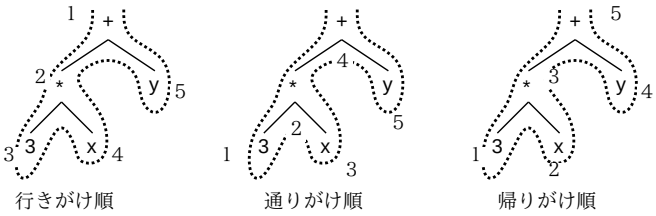

図 11: たどりの順序

```
プログラム ::= 文 プログラム | ε
文 ::= 代入文 | 入力文 | 出力文 | ループ文 | { プログラム }
代入文 ::= 変数 = 式 ;
入力文 ::= read 変数 ;
出力文 ::= print 式 ;
ループ文 ::= loop ( 式 ) 文        ←「式」で指定された回数「文」を実行
```

図 12 にこの言語による簡単なプログラム (「入力に 1 足す」「階乗の計算」) の抽象構文木を示す。これに対応する木構造を作って表示させてみよ (「;」は文の連接を表す。演算の種類によって表示のしかたを変化させてプログラムをそれらしく打ち出してみてもよい)。

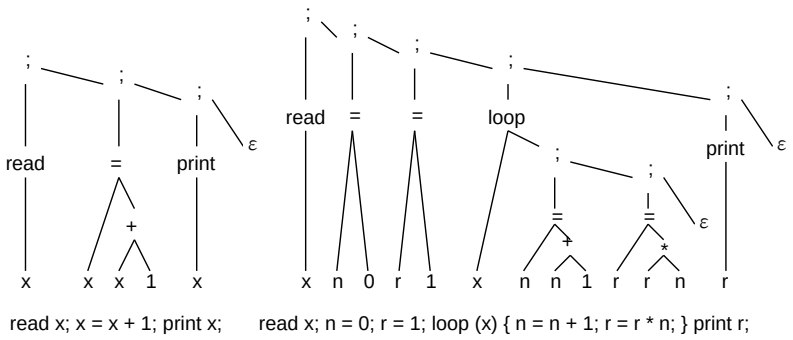

図 12: 「簡単な言語」の抽象構文木

4 スタックとキュー

4.1 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」とも呼ばれる。つまり、スタックには次々にデータを入れて置けるが、取り出す時には (まだ取り出されてしまってないもののうち) 一番最近に入れたものが取り出されてくる。これはちょうど、ものを上に積んでいって取り出す時の動作を同じなのでスタック (積む) と呼ばれる。スタックの入れる/取る動作は伝統的に push/pop と呼ばれる (図 13 左)。

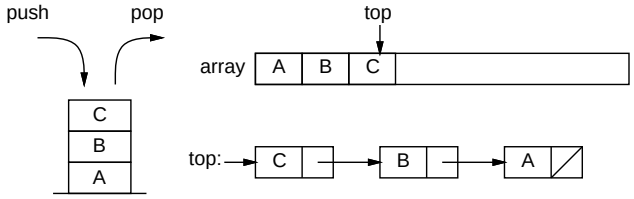

図 13: スタック

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ある。先頭の位置は変数 ptr に覚えておく (図 13 右上)。この方法はシンプルだが、最大いくつまで要素を格納するか予め決めておく必要がある。

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ある (図 13 右下)。この方法では最大要素数を指定する必要は特にない。

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す。

```
import java.io.*;

public class R7Sample2 {
  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 {
    Object[] a; int ptr = -1;
    public Stack(int s) { a = new Object[s]; }
    public boolean isEmpty() { return ptr < 0; }
    public void push(Object o) { ++ptr; a[ptr] = o; }
    public Object pop() { Object o = a[ptr]; --ptr; return o; }
  }
}
```

動かす様子を見てみましょう。

```
% java R7Sample2
> A  ←入れる
> B  ←入れる
> C  ←入れる
>    ←リターンで
C    ←取り出し表示
>
B    ← B まで取り出し
> D  ← 3
> E  ←つ
> F  ←積む
>    ←全部取り出し
F
>
E
>
D
>
A     ←最初の「A」が最後に出る
> q   ←「q」で終わり
%
```

演習 7 連結リストを使ったスタックの実現を作って上の例題で同じに動作することを確認せよ。

4.2 キュー

キュー (queue) もスタックと類似の汎用抽象データ型だが、違うのは取り出すときに、もっとも始めに近く入ったもの (でまだ取り出されていないもの) が取り出されることである。これを「**FIFO**(fist-in, first-out) のデータ構造」とも呼ぶ。キューとはおなじみ「行列」を意味する英語である (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…)。キューの入れる操作/取り出す操作は伝統的に enq/deq と呼ばれる (14 左)。[2]

[2]実はフルスペルは enqueue と dequeue だが長いし発音が同じなので短く書くのが普通。

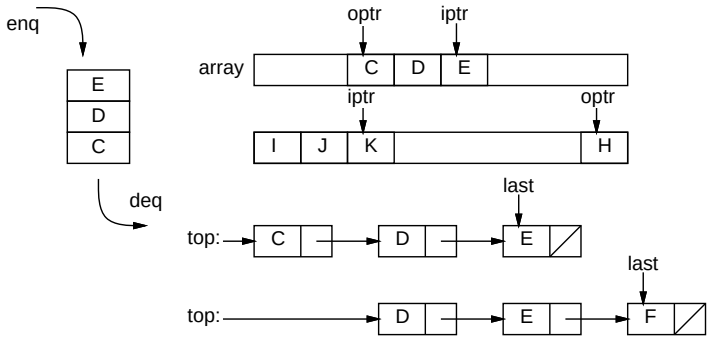

図 14: キュー

キューの実現も配列版と連結リスト版が考えられる。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」を覚えておく。そうすると、ずっと入れて出してしていくと入っている場所が配列の端まで来るので、ぐるっと回って次は先頭に戻るようにする必要がある (図 14 右上)。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くのが普通 (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るが、その場合は 1 箇所は空けて置かないと空っぽの状態と区別がつかなくなる)。

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する (図 14 右下)。ただし、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る。

以下には配列版のキューのプログラムを示しておく (main() の使い方は先の例と同じ)。

```
import java.io.*;

public class R7Sample4 {
  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 {
    Object[] a; int count = 0, iptr = -1, optr = 0;
    public Queue(int s) { a = new Object[s]; }
    public boolean isEmpty() { return count <= 0; }
    public void enq(Object o) { iptr=(iptr+1)%a.length; a[iptr] = o; ++count; }
    public Object deq() {
      Object o = a[optr]; --count; optr = (optr+1)%a.length; return o;
    }
  }
}
```

演習 8 連結リストを使ったキューの実現を作って上の例題で同じに動作することを確認せよ。

4.3 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だろう。それは、何かを入れて (覚えて) おいて、後で取り出して使うから。当り前だと思うかもしれないが、こういうことは結構ある。そして、単独の変数に覚えておくのだと、1 つしか覚えておけない (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る)。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たない。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」ことができるので簡単であり、そして入れたものは必ずいつか出て来ることが保証される。

たとえば、先の再帰手続きでやった式木のたどりをスタックやキューでやってみよう。まずスタック版から。

```
public class R7Sample6 {
  public static void main(String[] args) {
    Stack s = new Stack(100);
    s.push(
      new Tree('+',
        new Tree('*', new Tree('i', "3"), new Tree('v', "x")),
        new Tree('v', "y")));
    while(!s.isEmpty()) {
      Tree t = (Tree)s.pop();  // とりあえず 1 個取り出して来て、
      if(t.op == 'i' || t.op == 'v') {
        System.out.println((String)t.left); // 末端なら出力
      } else {
        System.out.println(t.op); // 中間なら種別を出力し、
        Tree t1 = (Tree)t.right; if(t1 != null) { s.push(t1); }
        Tree t2 = (Tree)t.left;  if(t2 != null) { s.push(t2); }
     }  // 左と右の子をスタックに入れる
    }
  }
  static class Stack ... 同じにつき略
  static class Tree  ... 同じにつき略
}
```

なお、左と右の子をスタックに入れるとき、右を先に入れるのは、取り出す時左から取り出された方が見やすいから。実行すると次のようになる。

```
+
*
3
x
y
```

つまり、木の行きがけ順のたどりがスタックで行える。では、キューだとどうだろうか。

```
public class R7Sample7 {
  public static void main(String[] args) {
    Queue q = new Queue(100);
    q.enq(
      new Tree('+',
        new Tree('*', new Tree('i', "3"), new Tree('v', "x")),
        new Tree('v', "y")));
    while(!q.isEmpty()) {
      Tree t = (Tree)q.deq();
      if(t.op == 'i' || t.op == 'v') {
        System.out.println((String)t.left);
      } else {
        System.out.println(t.op);
        Tree t1 = (Tree)t.left;  if(t1 != null) { q.enq(t1); }
        Tree t2 = (Tree)t.right; if(t2 != null) { q.enq(t2); }
     }
    }
  }
  static class Queue ... 同じなので略
  static class Tree  ... 同じなので略
}
```

スタックがキューに変わることと、今度は入れた順番に出て来るので左から入れるようにした点を除けば、まったく同じ。このコードを実行すると次のようになる。

```
+
*
y
3
x
```

これはどういう順だろう? 実は、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、…」という順番になる。これを**深さ優先のたどり** (depth-first traversal) という。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、…」という形になる。これを**幅優先のたどり** (breadth-first traversal) と呼ぶ (図 15)。

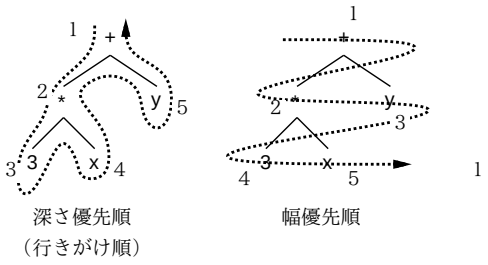

図 15: 深さ優先と幅優先

演習 9 図 16 はとある地域の鉄道路線図のようなものである (架空のものであり現実の場所とは関係ありません)。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。スタック版とキュー版で比較すること。(ヒント: 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する。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) 入れる。ただし同じ場所を堂々巡りにならないために、`visited` という `boolean` 型のフィールドを設け、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 `visited` が `true` のものは処理ずみなのでスタック (キュー) に入れないようにする。)

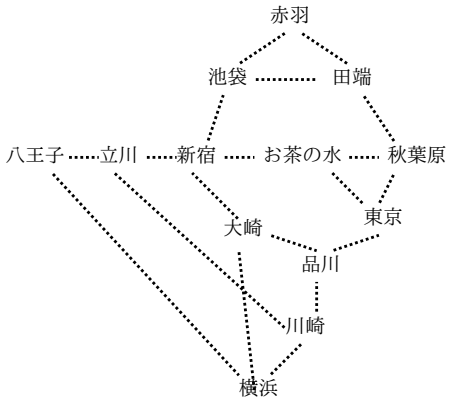

図 16: 鉄道路線図

演習 10 エディタ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。
- 1 つ下に行くには、スタック B から 1 つ pop してそれをスタック A に push。
- 1 つ上に行くには、スタック A から 1 つ pop してそれをスタック 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る。

演習 11 木構造またはスタックまたはキューを使った面白いプログラムを作れ。何が面白いかの基準は各自に任せる。

5 演習問題解説: 表と検索

表と検索の話題は前回内容が多かったところだけど、課題でやってくれている人もいるので解説を載せておきます。他の人も一応目を通しておいてください。まず演習 6 だが、前回資料のままの線形探索はあまり速くないので、速くするにはどうするか、というところで 2 つほど方法を説明してあった。以下表のクラスだけ示す。1 つ目は 2 分探索。

```
static class IntStrTable {
  Pair[] a;
  int size, count;
  boolean sorted;
  public IntStrTable(int s) { a = new Pair[s]; size = s; count = 0; }
  private int find(int k) {
    int b = 0, t = count;
    while(b < t) {
      int h = (b + t) / 2; if(h == b) { break; }
      if(a[h].key == k) { b = h; break; }
      if(a[h].key < k) { b = h; } else { t = h; }
    }
    return b;
  }
  public void put(int k, String s) {
    if(count == 0) { a[count] = new Pair(k, s); ++count; return; }
    int i = find(k);
    if(a[i].key == k) { a[i].str = s; return; }
    for(int j = count; j > i+1; --j) { a[j] = a[j-1]; }
    a[i+1] = new Pair(k, s); ++count; return;
  }
  public String get(int k) {
    if(count == 0) { return null; }
    int i = find(k);
    if(a[i].key != k) { return null; } else { return a[i].str; }
  }
}
```

2 分探索の場合、find() では常に区間 $[b, t)$ にキーがある (またはキーが登録されているすべてのキーより大きい場合には t がキーの入るべき位置) という条件を常に保ったまま、ループで区間を半分ずつにしていく。それには、b と t の中点 h を取り、その位置とキーの大小関係を見て b か t を h で置き換えていく。これ以上半分にできなくなった時か、ぴったり見つかった時にループを終わる。ただし、表が整列されていないといけないので、put() するときキーが見つからなければ (find() は挿入すべき位置を探してくれるので) 挿入位置以降を 1 つ後ろへずらしてそこに挿入する。

もう 1 つはハッシュ表と呼ばれる方法で、キーを適当に折り畳んで配列上の入るべき位置を計算する。ただし衝突した場合は空いている場所を探して代わりにそこに入れる。

```
static class IntStrTable {
  Pair[] a;
  int size, count;
  boolean sorted;
  public IntStrTable(int s) {
    s = s + s; while(!isPrime(s)) { ++s; }
    a = new Pair[s]; size = s; count = 0;
  }
  private boolean isPrime(int n) {
    for(int i = 2; i*i <= n; ++i) { if(n%i == 0) { return false; } }
    return true;
  }
  private int h1(int k) { return k % a.length; }
  private int h2(int k) { return k % (a.length-2) + 1; }
  private int find(int k) {
    int pos = h1(k), d = h2(k);
    while(a[pos] != null && a[pos].key != k) { pos = (pos+d)%a.length; }
    return pos;
  }
  public void put(int k, String s) {
    int i = find(k);
    if(a[i] == null) {
      if(count*2 >= size) { return; } else { a[i] = new Pair(k,s); ++count; }
    } else { a[i].str = s; }
  }
  public String get(int k) {
    int i = find(k);
    if(a[i] == null) { return null; } else { return a[i].str; }
  }
}
```

なぜ素数を調べているのかというと、衝突が起きにくく、また別の場所を探している時に「堂々めぐり」にならないためには、配列サイズが素数であることが望ましいため。また、空きがある程度ないと衝突だらけになるので、配列は倍程度の余裕を持って作っている。`find()` では 2 つの関数 `h1()`、`h2()` で最初の折り畳みと代替場所を探す時の飛び数を計算し、キーが一致する場所か空き場所が見つかるまで探して行く。`put()` では探した場所が空き場所ならそこに入れるが、表を一杯にしてしまわないため、容量の半分を超えたら入れないようにしている。では時間を計測してみよう (表 1)。

表 1: 表の探索プログラムの所要時間

データ数 (10^3)	5	10	20	30	50	100	200	400
線形探索	1,063	4,046	16,071	-	-	-	-	-
2 分探索	135	369	1,188	2,622	8,416	-	-	-
2 分探索木	73	113	226	369	588	1,426	2,923	4,514
ハッシュ表	71	109	213	346	551	1,306	2,821	4,506

計算量は線形探索が $O(n^2)$、2 分探索は探索については $O(nlogn)$ なのだが、表を挿入時に整列しているので単純挿入法を実行しているのと同じで、こちらで $O(n^2)$ になってしまう (挿入が終わってから一気に quicksort などで整列したらよさそうだが、挿入時にも重複するキーの挿入があるため検索が必要なので駄目である)。2 分探索木は $O(nlogn)$。ハッシュ表は衝突の確率が低いうちは 1 回の検索が n に依存しないので、$O(n)$ というすごい性能を持つ。しかし計測してみると、2 分探索木でもほとんど遜色ない性能なのですね、この程度だと。

A　本日の課題 7A

「演習 4」または「演習 5」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 7A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 「演習 4」または「演習 5」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. 構文木、抽象構文木について学びましたが、納得しましたか。

   Q2. スタックとキューについて学びましたが、どんなものか分かりましたか。

   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　次回までの課題 7B

次回までの課題は「演習 5」〜「演習 11」の (小) 課題から 2 つ以上選んで報告することです。実力相応に選んでください。あまり無理はしないで結構ですので。

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 7B」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. 構文木のようなデータが扱えるようになりましたか。

   Q2. スタックまたはキューが扱えるようになりましたか。

   Q3. 課題に対する感想と今後の要望をお書きください。