システムソフトウェア特論'17 # 4 形式言語

久野　靖 *

2017.8.30

1　形式言語入門

1.1　形式言語理論の位置付け

形式言語理論 (formal language theory) とは、言語の構文や意味を形式的に (formal に — 定義にもとづき厳密に) 取り扱うための理論全般を指します。

自然言語 (日本語・英語など人間がふだん使う言語) は多様性や曖昧さが大きいため、形式的に扱うのは難しいですが、プログラミング言語に代表されるような人工言語については、形式言語理論に基づいて扱うことで理論的基盤ができ、曖昧さのない扱いが可能になり、処理系も作りやすくなります。

なお、意味の形式的な扱い (形式的意味論) も多く研究されてはいますが、言語処理系を作るという点では、意味まで形式的に扱うことは必ずしも一般的ではありません。そこで、ここでは構文の扱いに限定して説明していきます。[1]

1.2　形式言語の諸定義

まず、どのような言語でもそれを組み立てるもととなる文字ないし記号 (の集まり) がなければ成立しません。形式言語ではこれを**アルファベット** (alphabet) と呼びます。

定義 有限個の記号から成る空でない集合 V をアルファベットと呼ぶ。V の要素を並べて得られる記号列を**語**と呼ぶ。

V の要素を 0 個以上並べて得られる語の全体を V^*、1 個以上並べて得られる語の全体を V^+ と記します。また長さ 0 の列を ε と記します。したがって、$V^* = V^+ \cup \{\varepsilon\}$ です。例えば $V = \{a, b\}$ のとき、$V^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, abb, baa, \ldots\}$ となります。そして、**言語**は次のように定義されます。

定義 V^* の任意の空でない部分集合 L を V の上の言語、L の要素を言語 L の**文** (sentence) と呼ぶ。

例えば上の例で $L = \{a, b, aab\}$ とすれば、これら 3 つの語だけが言語 L の文です。しかし「正しい文を列挙する」やり方では有限個の文を持つ L しか定義できません。一方、例えば「a がいくつか並び、続いて b がいくつか並ぶもの」という言語を考えると、これは V^* の部分集合であり、かつ無限の要素を持ちます。これを自然言語に頼らずに定義するにはどうしたらよいでしょう。それには「V^* の要素中でこのような規則にかなうもの」という形で言語を定義します。そのような「規則」を表現するのに、自然言語の代りに**生成文法**の力を借ります。

定義 T、N を互いに共通部分を持たない有限な記号の集まり、P を「$\alpha \rightarrow \beta$」(ただし $\alpha \in (T \cup N)^+$、$\beta \in (T \cup N)^*$) の形をした要素の有限集合、S を N の 1 要素としたとき、4 つ組 $G = (T, N, P, S)$ を生成文法と呼ぶ。

---

*電気通信大学 情報理工学研究科

[1] ここで説明する形式言語は Noam Chomsky の提唱した文法理論によるもので、「Chomsky 文法」などと呼ばれることもあります。

ここで T の要素はプログラミング言語などで言えば `if`、`x`、`:=`などプログラムの字面上に現れるトークンに相当し、**終端記号ないし端記号** (terminal symbol) と呼ばれます。

一方、N の要素はプログラムの字面には現れないが、その構造を説明するのに都合がいいような概念ないし構文要素 (「文」、「代入文」、「変数」など) に対応するものであり、**非終端記号ないし非端記号** (non-terminal symbol) と呼ばれます。

そして S は「プログラム」に相当するもので、**出発記号** (start symbol) と呼ばれます。最後に P は**生成規則** (production rule) と呼ばれ、**導出** (derivation) に用いられます。

定義 $u = x\alpha y$、$v = x\beta y$ かつ $\alpha \to \beta \in P$ のとき、u から v が**単導出** (one-step derivation) できると言う ($u \Rightarrow v$ と記す)。$u = u_0 \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_n = v$ ($n \geq 0$) のとき u から v が導出 (derivation) できると言う ($u \Rightarrow^* v$ と記す)。

そして、文法 G を用いて T 上の言語 $L(G)$ は次のように定義されます。

$$L(G) = \{x \in T^* | S \Rightarrow^* x\}$$

すなわち、出発記号 S から始めて次々に P の規則を適用していって生成される記号列の中で、T の要素のみから成るものの集合が $L(G)$ となります。これが P を生成規則、G を生成文法とよぶ理由です。

生成文法を用いて前出の「a がいくつか並び、続いて b がいくつか並ぶもの」という言語を定義するには次の生成規則を用いればよいのです。

$$P = \{S \to aAbB, A \to aA, A \to \varepsilon, B \to bB, B \to \varepsilon\}$$

例えば「$aabb$」という列は次のようにしてこの P により導出できます。

$$S \Rightarrow aAbB \Rightarrow aaAbB \Rightarrow aabB \Rightarrow aabbB \Rightarrow aabb$$

A や B は最後は ε に置き替わって消えることに注意。以下では当面 T の要素を英小文字、N の要素を英大文字、出発記号を S で表します。

生成文法に対して、各生成規則の形に応じて次のようなクラス分けを行います。

タイプ 0 文法: 上の定義と同じ。つまりもっとも一般的な場合。

タイプ 1 文法: 各生成規則において、左辺の記号列の長さは右辺の記号列の長さを超えないもの。これは実は各生成規則が次の形をしている、というのと等価です (証明は略します)。

$$mAn \to mtn, A \in N, t \in (N \cup T)^+, m, n \in (N \cup T)^*$$

タイプ 1 文法を**文脈依存文法** (context sensitive grammer、CSG) とも呼びますが、これは後者の書き方をした場合に非端記号 A が置き換えられるかどうかはその周囲 (つまり文脈) に何があるかによって決まることによります。

タイプ 2 文法: 各生成規則の左辺が 1 個の非端記号から成る。つまり各生成規則が

$$A \to t, A \in N, t \in (N \cup T)^*$$

の形をしているもの。タイプ 2 文法を**文脈自由文法** (context free grammer、CFG) とも言うが、これは文脈依存文法と対比して、A は周囲に何があろうと (文脈に依存せず) いつでも t に置き換えてよい、という形になっているからです。

タイプ 3 文法: 各生成規則が

$$A \to a \text{ または } A \to aB, A, B \in N, a \in T$$

の形をしているもの。タイプ 3 文法のことを**正規文法** (regular grammer) と呼びます。正規文法によって定義される言語は、それと等価な言語を表現する**正規表現** (regular expression) が存在する、という性質があることから、**正規言語** (regular language) と呼ばれます。

1.3 文法と言語の認識/解析

生成文法では G を与えて $L(G)$ の要素を何でも適当に作り出すのは容易です (適用できる P の規則を順に試していけばよい)。しかし言語処理系ではその逆、すなわち T^+ の要素 t(ソースプログラム) を与えて、それが確かに $L(G)$ に属するかどうかを判定すること、さらには S からどのように規則を適用して t が生成されるかを決めることが必要です。一般に前者 (判定) を行うプログラムを $L(G)$ の**認識器** (recognizer) ないし構文検査器、後者 (構造の決定) を行うプログラムを $L(G)$ の**解析器** (parser) と呼びます。

前掲の生成文法のクラスにおいて、後に述べたものほど認識器/解析器をつくるのが容易です。例えば「a がいくつか、続いて b がいくつか」という文法の例はタイプ 2(文脈自由文法) でしたが、これを次のように書き換えるとタイプ 3(正規文法) にできます (同一の言語を定義する生成文法は一般に複数存在し得ることに注意)。

$$P = \{S \to aA, A \to aA, A \to bB, B \to bB, B \to \varepsilon\}$$

図 1 に○と◎と矢印から成るグラフを示しましたが、これは先の正規文法の各非端記号が○や◎に、生成規則による置換りが矢印に対応したものとなっています。例えば「$S \to aA$」という規則は S の○から A の○へ a のラベルがついた矢印に対応しています。また、ε が単導出できる場合にはその非単記号は◎で表します [2]。

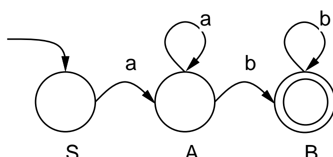

図 1: aaa...bbb... に対応する有限オートマトン

次に○と◎をプログラム中の goto で飛んでいけるラベルに対応させ、まず S のラベルから実行開始し、各ラベルの箇所で「次の記号」を読んで、それが a だったら a の矢印に従って A のラベルに飛ぶ、というプログラムを構成したとします。このプログラムに入力を与えて実行させ、もしどこかで入力に対応するラベルが見つからなければ入力は $L(G)$ の要素ではありません。◎のラベルに到着すれば、入力は $L(G)$ の要素であり、これまでにたどった矢印はそれぞれ P の規則に対応していたので、それらを覚えておけば構造もわかったことになります [3]。

このような○と◎とラベルつき矢印から成るグラフを**オートマトン** (automata、自動機械の意) と呼びます。そして、○と◎のことを**状態** (state)、最初にたどり始める状態を**初期状態**、◎の方を**最終状態**、ラベルつき矢印を**遷移** (transition) と呼び、状態の数が有限のものを**有限オートマトン**と呼びます。

ここで示したように、正規文法からは常に対応する (ということは文法が規定する言語を認識/解析する) 有限オートマトンが作り出せ、有限オートマトンは計算機プログラムに変換できるので、結果として正規言語を認識/解析するプログラムが得られることになります。

[2] $A \to a$ の形の規則があった場合には名なしの◎を用意し、A の○からそこへ a のラベルのついた矢印を描く。
[3] ある○から同じラベルをもつ矢印が 2 個以上出ていて行き先が一意でない場合もあり得ます。このような場合の扱いについてはまた別の回にやります。

一般的なプログラミング言語について言えば、その定義には正規文法では制約が強過ぎるため、文脈自由文法を使用するのが普通です。文脈自由文法に対しても、その大部分について、効率良い解析器が作り出せることが知られています。

一方、タイプ 0 文法やタイプ 1 文法の場合にはそれを解析するアルゴリズムは存在しますが、認識する列の長さの 3 乗に比例した計算時間を要します (後でその 1 つを見てみます)。このため、これらの文法を用いてプログラミング言語の構文を記述することはあまりありません。

1.4 文脈自由文法とその記法

ここまででは文法の記号を全て英字 1 文字で表してきました。しかし、プログラミング言語を記述する際には各文法記号に「意味のある名前」をつけたくなります。そこで、以下では文脈自由文法を書き記すやり方を次のように改めることにします。

- 各記号を、その意味をよく表すような名前を用いて書き表し、端記号もそのまま直接書く。[4]
- 矢印「→」の代りに「::=」を用いる。また「ε」の代りに何も書かないか、または「nil」と書く (キーボードにある文字だけで文法を書き表すため)。
- 同一の左辺をもつ規則はまとめて、右辺を「|」で区切って並べて書く。

この方式で先に出てきた「a がいくつか、b がいくつか」を書いたものを示してきます。

Start ::= SeqA SeqB
SeqA ::= ”a” SeqA | nil
SeqB ::= ”b” SeqB | nil

このような記法をその発明者の名前を取って **BNF**(Backus Normal Form) 呼ぶのでした。

次にこのようにして記した文法と実際の入力の対応関係を示す記法について考えましょう。1 つの方法は導出の系列を逐一記すことですが、非常に長々しく、記号列のどこをどの規則により置き換えたのかわかりづらいという欠点があります。

この問題を解決する方法の 1 つは、導出を**構文木**によって表すことです。構文木とは、(1) 出発記号を根に持ち、(2) 非端記号を中間の節に持ち、(3) 端記号を葉にもち、(4) 各節から葉の方向へ向かう枝はその節を左辺とする導出に対応するような木構造のグラフです。

先の文法での「aabb」の導出に対応する構文木を図 2 に示します。

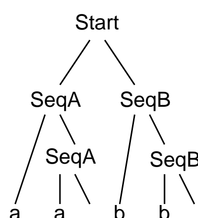

図 2: 構文木の例

ただし、構文木では導出列と比べて落ちている情報があります。例えばこの構文木は次のどちらに対応しているとも考えられます。

```
Start ⇒ SeqA SeqB ⇒ ”a” SeqA SeqB ⇒ ”a” ”a” SeqA SeqB
 ⇒ ”a” ”a” SeqB ⇒ ”a” ”a” ”b” SeqB ⇒ ”a” ”a” ”b” ”b” SeqB
 ⇒ ”a” ”a” ”b” ”b”
```

[4]両者の区別をつけるためには、非端記号は<...>で囲んで書き端記号はそのまま書く、逆に非端記号をそのまま書き端記号を"..."で囲んで書く、どちらも囲まずフォントで区別するなど、様々な流儀があります。

```
Start ⇒ SeqA SeqB ⇒ SeqA "b" SeqB ⇒ SeqA "b" "b" SeqB
 ⇒ SeqA "b" "b" ⇒ "a" SeqA "b" "b" ⇒ "a" "a" SeqA "b" "b"
 ⇒ "a" "a" "b" "b"
```

そこで次の定義を行います。

定義 導出の各ステップにおいて、記号列に含まれる非端記号のうち最も左側にあるものを常に置き換える導出を**最左導出** (leftmost derivation)、最も右側にあるものを常に置き換える導出を**最右導出** (rightmost derivation) と呼ぶ。

先の導出系列は前者が最左導出、後者が最右導出です。コンパイラで用いる解析アルゴリズムは通常、このどちらかの導出を扱います。このいずれかであると決まれば、構文木と導出系列は 1 対 1 で対応させられます。

ところで、文法によってはある 1 つの文に対し構文木が 2 つ以上存在することもあります。例えば次の文法を見てみましょう。

```
Expr ::= Expr "+" Expr | Ident
```

この文法で「`Ident + Ident + Ident`」を導出することができますが、その構文木は図 3 のように 2 つ存在します。

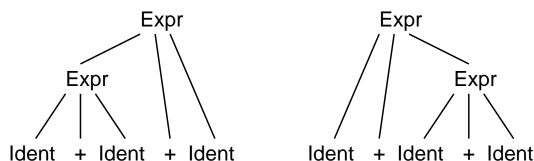

図 3: 曖昧な文法の構文木

このような文法を「**曖昧** (ambiguous) である」と言います。反対に、任意の文に対して常に構文木が 1 つだけしか存在しないような文法を「曖昧でない」と言います。通常、構文解析のアルゴリズムでは曖昧でない文法を扱います。

1.5 正規文法と正規表現

正規言語は文脈自由言語の特殊な場合なので、BNF で記述できます。しかし正規言語の場合は**正規表現** (regular expression) と呼ばれる、よりコンパクトな記法で表すことが一般的です。正規表現およびそれが表す言語とは次のようなものです。

1. 「ε」は正規表現である。これは空列を表す。
2. a が任意の終端記号のとき、「a」も正規表現である。これは a そのものを表す。
3. α、β が正規表現であれば、「$\alpha\beta$」も正規表現である。これは α が表す列の後に β が表す列を連結したものを表す。
4. α が正規表現であれば、「$\alpha*$」も正規表現である。これは α が表す列を 0 回以上反復したものを表す。

例えば、「a がいくつか、続いて b がいくつか」を正規表現で表すと「$aa*bb*$」となります。

このほか、表現の曖昧さを除くため適宜 () を使用します。さらに読みやすさのため、$\alpha\alpha*$(1 回以上の反復) を $\alpha+$、$(\alpha \mid \varepsilon)$(あってもなくてもよい) を $[\alpha]$ で表すことも多く行なわれます。

正規文法と正規表現の表現能力は等しく、一方で記述された言語は他方でも記述できることが知られています。また従って、正規表現を有限オートマトンに変換することもできます。

2　オートマトンによる正規言語の認識

理論の話ばかりでは面白くないので、先に出て来た「正規言語の認識器は容易に構築できる」を実際に確認してみましょう。先の説明では goto でそれぞれの状態にジャンプするということになっていましたが、Java では goto が使えないので、かわりに状態を番号であらわし、次のような形で扱います。

```
    int stat = 0; // 最初の状態は 0 番
    while(true) {
      switch(stat) {
case 0: 状態 0 の処理;
        break;
case 1: 状態 1 の処理;
        break;
        ...
      }
      System.out.println("error."); return;
    }
```

この場合、それぞれの状態の処理の中で `stat` に次の状態番号を入れて「`continue;`」で次の周回に進むことで、次の状態の処理に移ります。どの状態にも行けない場合は「`break;`」に来て switch 文から抜け、エラーを表示して戻ります。

では、図 1 に対応するプログラムを示します。入力を 1 文字ずつ調べるのには前に使った `CharTok` をそのまま利用しています。

```
import java.util.*;
import java.io.*;

public class Sam41 {
  static CharToknizer tok;
  public static void main(String[] args) throws Exception {
    tok = new CharToknizer(args[0]);
    int stat = 0;
    while(true) {
      switch(stat) {
case 0: if(tok.chkfwd("a")) { stat = 1; continue; }
        break;
case 1: if(tok.chkfwd("a")) { stat = 1; continue; }
        if(tok.chkfwd("b")) { stat = 2; continue; }
        break;
case 2: if(tok.chkfwd("b")) { stat = 2; continue; }
        if(tok.chkfwd("$")) { System.out.println("accept."); return; }
        break;
      }
      System.out.println("error."); return;
    }
  }
}
```

```
class CharToknizer {
  String str, tok;
  int pos = -1;
  boolean eof = false;
  public CharToknizer(String s) { str = s; fwd(); }
  public boolean isEof() { return eof; }
  public String curTok() { return tok; }
  public boolean chk(String s) { return tok.equals(s); }
  public void fwd() {
    if(eof) { return; }
    pos += 1;
    if(pos >= str.length()) { eof = true; tok = "$"; return; }
    tok = str.substring(pos, pos+1);
  }
  public boolean chkfwd(String s) {
    if(chk(s)) { fwd(); return true; } else { return false; }
  }
}
```

3つの状態がそれぞれ0、1、2となります。2は◎ (最終状態) なので、どこにも遷移がない場合は`tok.chkfwd("$")`で入力の終わりか調べ、終わりなら「成功」と表示して終わります。動かしているところを見てみましょう。

```
% java Sam41 aaabb
accept.
% java Sam41 aaaba
error.
%
```

演習 4-1 例題をそのまま動かしてみよ。動いたら、次のような正規表現の認識器を作成してみよ (正規文法にまず変換し、それからオートマトンを描き、それをそのままプログラムにすること)。

a. $a + b + c+$。
b. $a + (b * |c)d+$。
c. $a + (b + |c) * d+$。
d. $(a + (b + |c) * d+)*$。
e. その他自分で書いた正規表現。

演習 4-2 プログラミング言語に出て来る次のような部分の正規表現を書き、前問と同様にして認識器を作れ。

a. 整数定数。正負の符号もつけられること。
b. 整数定数。16 進法も扱えること。
c. 実数定数。指数部は無くてよい。
d. 実数定数。指数部もつけられること。
e. 名前。名前の定義は自分で考えてよい。
f. その他自分の好きなもの。

3　CYK 構文解析アルゴリズム

プログラミング言語の定義には文脈自由文法が使われるわけなので、コンパイラではその文法に対する構文解析が必要となります。コンパイラで構文解析に使われる手法としては、既に学んだ再帰下降解析がありましたし、そのほか複数のものをこの後取り上げます。しかし、これらはいずれも扱える文法にかなり制限があります。

ここではそのような制約のない、任意の文脈自由言語を扱える構文解析手法として、**CYK**(Cocke-Younger-Kasami、これらは考案した人の名前) と呼ばれるアルゴリズムを見てみます。このアルゴリズムは文脈自由文法のうち CNF(Chomskey Normal Form、チョムスキー標準形) と呼ばれる形のものに対する構文解析 (ないし認識) を行えます。CNF とは、すべての生成規則が次のいずれかの形であるものを言います。

$$S \to \varepsilon$$
$$X \to AB$$
$$X \to a$$

つまり、ε 規則は出発記号 S に対してしか現れず、それ以外の規則はすべて右辺の長さが 2 また 1 で、2 の場合は 2 つとも非端記号、1 の場合は端記号に限られる、ということになります。

制約だらけじゃないかと思うかも知れませんが、すべての文脈自由文法は同等の言語を表すの CNF に変形可能であることが知られています (詳細は略)。このため、CYK アルゴリズムは任意の文脈自由文法に対する認識 (解析) 器となります (そのようなものが存在するという証明となっている)。

では CYK について解説しましょう。CNF では ε 規則はあっても 1 つだけであり、別に扱えば済むので、残りの 2 種類の規則のみを考えます。このアルゴリズムは動的計画と呼ばれる手法に基づいていて、論理値型の 3 次元配列 `p`$[j]$`[`i`][`c`]` を使用します。入力列の長さを n としたとき、これらの添字の範囲は次の通りです。

- j — $0 \sim n-1$。入力列の各文字 (端記号) の位置に対応。
- i — $1 \sim n$。記号列の長さに対応。
- c — $T \cup N$(端記号と非端記号の集合) に順に番号をつけたものとして、その番号の範囲に対応。

そして `p[`j`][`i`][`c`]` は「入力列の位置 j から始まる長さ i の列 α について、$X \Rightarrow^* \alpha$ の時 `p[`j`][`i`][`c`]` が `true`」になるように印をつけていきます。印をつけ終わった時に `p[0][`n`][`S`]` を見れば、出発記号から入力列が導出できるかどうか分かることになるわけです。

さて、動的計画法のアルゴリズムですが、まず最初は $i = 1$ のものを処理します。これは $X \to a$ の形の規則を見て、入力列の j 番の位置が a_j であれば、$X \to a_j$ なるすべての X につて `p[`j`][1][`X`]` を真にすればよいのです。

以降は長さ $i = 2, 3, ..., n$ について順に処理して行きます。入力列の長さ i の部分列について、それをさらに 2 つの部分列 (それぞれの長さは k と $i - k$、$1 < k < i$) に分けます。

i について順に処理していくので、長さ k および $i - k$ については既に配列 `p` のデータは完成していることに注意。$X \to YZ$ なるすべての規則について、`p[`j`][`k`][`Y`]` も `p[`$j+k$`][`$i-k$`][`Z`]` も真であれば、入力列の位置 $j \sim j+k-1$ は Y から導出でき、位置 $j+k \sim j+i-1$ は Z から導出できるので、位置 $j \sim j+i-1$ 全体が X から導出できるとわかるため、`p[`j`][`i`][`X`]` を真にします。

これを k の範囲 $1 \sim i-1$、j の範囲 $0 \sim n-i$ に渡ってすべて行うことで、長さ i についてすべての場合をチェックできるのです。以上が CYK のあらましです。

たとえば、次の文法を考えてみましょう (CNF になっていることに注意)。

$$S \to ST$$
$$S \to a$$

$T \to US$
$U \to b$

この文法について、入力列「$ababa$」を認識させる場合の配列 p の内容を図 4 に示します。

i=1

	a	b	a	b	a
j =	0	1	2	3	4
a					
b					
S	✓		✓		✓
T					
U		✓		✓	

i=2

	a	b	a	b	a
j =	0	1	2	3	4
a					
b					
S					
T		✓		✓	
U					

i=3

	a	b	a	b	a
j =	0	1	2	3	4
a					
b					
S	✓		✓		
T					
U					

i=4

	a	b	a	b	a
j =	0	1	2	3	4
a					
b					
S					
T		✓			
U					

i=5

	a	b	a	b	a
j =	0	1	2	3	4
a					
b					
S	✓				
T					
U					

図 4: CYK による構文の認識

まず長さ $1(i = 1)$ については、$S \Rightarrow a$、$U \Rightarrow b$、なので S と入力の a、U と入力の b に対応する箇所に印がつけられます。

次に長さ $2(i = 2)$ のときは、$T \Rightarrow US$ で、U が $j = 1, i = 1$、S が $j = 2, i = 1$ で印がついているので、T の $j = 1, i = 2$ に印をつます。同様に U が $j = 3, i = 1$、S が $j = 4, i = 1$ で印がついているので、T の $j = 3, i = 2$ に印をつけます。それ以外に長さ 2 で印がつくところはありません。

長さ $3(i = 3)$ のときは、$S \Rightarrow ST$ で、S が $j = 0, i = 1$、T が $j = 1, i = 2$ で印がついているので、S の $j = 0, i = 3$ に印をつけます。同様に S が $j = 2, i = 1$、T が $j = 3, i = 2$ で印がついているので、S の $j = 2, i = 3$ に印をつます。それ以外に長さ 3 で印がつくところはありません。

以下同様にして長さ 4、5 に記入し、そして最後に、S の $j = 0, i = 5$ に印があるので、入力列 $ababa$ は文法にあてはまると分かります。[5]

これを実際に Java プログラムにしたものを示しておきます。変数 psym に言語定義で使う端記号・非端記号 (いずれも 1 文字) の並びを与え、その順に番号を割り当てるものとします。ここでは記号は a, b, S, T, U にそれぞれ $0, 1, 2, 3, 4$ の番号を割り当てました。また 2 種類の規則はそれぞれ別の 2 次元配列 p1, p2 に格納します。

```
public class Sam42 {
  public static void main(String[] args) {
    String str = args[0], psym = "abSTU"; // a:0, b:1, S:2, T:3, U:4
    int n = str.length(), nsym = psym.length(), start = psym.indexOf("S");
    int[] a = new int[n];
    for(int i = 0; i < n; ++i) { a[i] = psym.indexOf(str.charAt(i)+""); }
    int[][] p1 = { {2,0}, {4,1} };          // S -> a, U -> b
    int[][] p2 = { {2,2,3}, {3,4,2} };      // S -> S T, T -> U S
    boolean[][][] p = new boolean[n][n+1][nsym];
    for(int i = 0; i < n; ++i) {
      for(int r = 0; r < p1.length; ++r) {
        int x = p1[r][0], y = p1[r][1];
        if(a[i] == y) { p[i][1][x] = true; }
      }
    }
    for(int i = 2; i <= n; ++i) {
```

[5]なお、こうして見ると行列の端記号に対応する列に印がつくことがないので、本当は非端記号の分だけで十分なわけですが、記号に通しで番号をつけるものと考え、全部入れてあります。

```
        for(int k = 1; k < i; ++k) {
          for(int j = 0; j < n-i+1; ++j) {
            for(int r = 0; r < p2.length; ++r) {
              int x = p2[r][0], y = p2[r][1], z = p2[r][2];
              if(p[j][k][y] && p[j+k][i-k][z]) { p[j][i][x] = true;}
//            System.err.println(i+":"+j+":"+k+":"+x+":"+y+":"+z+p[j][i][x]);
            }
          }
        }
      }
      System.out.println(p[0][n][start]); // true/false(結果) 表示
      for(int i = 1; i <= n; ++i) {       // 以下は配列 p の表示
        System.out.println("---- " + i + " -----");
        System.out.println("  " + str);
        for(int s = 0; s < nsym; ++s) {
          System.out.print(psym.charAt(s) + " ");
          for(int j = 0; j < n; ++j) { System.out.print(p[j][i][s]?"t":"."); }
          System.out.println();
        }
      }
    }
  }
```

結果の真偽値出力より後の部分は、配列 p の内容を表示させてみるためのものです。「真偽値 ? X : Y」というのは真偽値が真のときは X、偽のときは Y を値とするという演算子 (if-else の演算子バージョン) です。

では動かしてみましょう。入力は「ababa」です。

```
% java Sam42 ababa
true
---- 1 -----
  ababa
a .....
b .....
S t.t.t
T .....
U .t.t.
---- 2 -----
  ababa
a .....
b .....
S .....
T .t.t.
U .....
---- 3 -----
  ababa
a .....
```

```
b .....
S t.t..
T .....
U .....
---- 4 -----
  ababa
a .....
b .....
S .....
T .t...
U .....
---- 5 -----
  ababa
a .....
b .....
S t....
T .....
U .....
%
```

長さ1のところで、aはS、tはUから導出できることが示されます。次に長さ2のところで、Tが2つの「ba」を導出できることが示されます。長さ3のところで、Sから「aba」を導出できることが示されます(2箇所あります)。長さ4のところでTから「baba」が導出でき、これに基づいて長さ5のところでSから「ababa」が導出できると分かります。

ところで、前にやった再帰下降解析器は開始記号(「プログラム」)から初めて下に向かって構文木を組み立てていく方法(下向き解析)でしたたが、この解析方法は端記号から始めて逆向きに規則を適用していく、**上向き解析**に相当します。

アルゴリズムの効率であるが、コードを見れば分かる通り、入力列の長さを n としたとき、CYKの時間計算量は $O(n^3)$ となります。これは先の再帰下降解析の $O(n)$ に比べるとかなり遅いですがが、そのかわりに。CYKは任意の文脈自由文法を(CNFに書き換えた上で)解析できるます。

ただし、本物のコンパイラでは巨大な(何万行もの)ソースプログラムを解析するため、$O(n)$ より計算量の大きいアルゴリズムでは実用になりません。プログラミング言語の文法自体もそのことを前提として、($O(n)$ で解析できるように)設計されているわけです。

演習 4-3 この例題をそのまま動かせ。動いたら文法を次のものに書き換えて正しく認識がなされることを確認せよ。[6]

a. $S \to TU, T \to TV, T \to a, U \to VU, U \to c, V \to b$。

b. $S \to LT, T \to SR, S \to 1, L \to (, R \to)$。

c. $S \to PT, T \to 1, T \to x, T \to LC, L \to (, C \to SR, R \to), P \to SQ, Q \to +$。

d. その他自分で試してみたいCNF文法。

演習 4-4 これまでに出て来た構文の好きなもの(ただし端記号が1文字)をCNFに書き換え、CYKプログラムを手直しして解析してみなさい。

[6] 文法を書き換えたらそこに出て来る記号のをすべて psym に入れる必要があることに注意。また記号の番号は psym に入れた文字列の何文字目かで決まることにも注意。

4　課題 4A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 4」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. 形式言語についてどのくらい知っていましたか。また、どの部分にとくに興味がありますか。

  Q2. オートマトンによる認識器や CYK による認識器についてどう思いましたか。

  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。