システムソフトウェア特論'19 # 6 構文解析 (1)

久野 靖 (電気通信大学)

2019.1.9

1　文脈自由文法の解析手法

構文解析はコンパイラの中で単なる 2 番目のフェーズ、というよりはだいぶ重要な位置を占めます。というのは、構文解析部はコンパイラの認識部の中枢であり、構文解析部が各構文要素を認識するのに合せて種々の動作が駆動されるようにコンパイラ (または、少なくともそのフロントエンド部) を構成することが多いからです。既に繰り返し出て来たように、今日のコンパイラでは**文脈自由文法**によって言語の構文を定義し、それに沿ってソースプログラムの構造を認識します。

ここで、任意の文脈自由文法を扱うことができればよいのですが、CYK のところで見たように、任意の文脈自由言語の認識は文 (=プログラム) の長さ n に対して $O(n^3)$ の時間計算量となります。コンパイラが受け付けるプログラムは、もちろんそのプログラムの複雑さにもよりますが、非常に長くなることも珍しくないので、このような時間計算量は到底受け入れられません。

正規文法のところで、正規言語であれば効率のよい解析器 ($O(n)$ のもの) が作れる、という説明をしましたが、実際には、文脈自由言語であってもある程度の限定があれば、同様に $O(n)$ の解析器を作ることができます。以下ではそのような限定としてどのようなものがあるかを説明しつつ、代表的な解析アルゴリズムを説明していきます。

具体例があった方がわかりやすいので、ここでは例として、次のような簡単な言語を考えましょう。なお、肩字で番号がふってあるのは、あとで生成規則を番号で参照するためです。

$$
\begin{aligned}
&\mathit{Program} ::= \mathit{StatList}^{1}\\
&\mathit{StatList} ::= \mathit{Stat}\ \mathit{StatList}^{2} \mid \texttt{nil}^{3}\\
&\mathit{Stat} ::= \mathit{Ident}\ \texttt{=}\ \mathit{Expr}\ \texttt{;}^{4} \mid \texttt{read}\ \mathit{Ident}\ \texttt{;}^{5} \mid \texttt{print}\ \mathit{Expr}\ \texttt{;}^{6}\\
&\qquad \mid \texttt{if (}\ \mathit{Cond}\ \texttt{)}\ \mathit{Stat}^{7} \mid \texttt{\{}\ \texttt{StatList}\ \texttt{\}}^{8}\\
&\mathit{Cond} ::= \mathit{Expr} < \mathit{Expr}^{9} \mid \mathit{Expr} > \mathit{Expr}^{10}\\
&\mathit{Expr} ::= \mathit{Ident}^{11} \mid \mathit{Iconst}^{12}
\end{aligned}
$$

解析部の出力としては、当面構文木を生成するものとします。そこでさらに具体例として、次のプログラムが入力されたとき、これに対応する構文木を手で組み立てみてください (ここで時間を取ってやってみてください)。

```
read x; read y; if(x > y) { z = x; x = y; y = z; } print x; print y;
```

演習　上記のプログラムを前述の文法にあてはめて構文木を描きなさい。

どうだったでしょうか。結果は図**??**のようになるはずです。ここで構文木を組み立てる過程を振り返ってみると、おおむねね上 (根) の方から描いていくか、または下 (葉) の方から描いていくかのどちらかだと思われます。

文脈自由文法の解析アルゴリズムも同様に分類でき、前者を**下向き解析** (top-down parsing)、後者を**上向き解析** (bottom-up parsing) と呼びます。今回は以下、下向き解析について説明します。

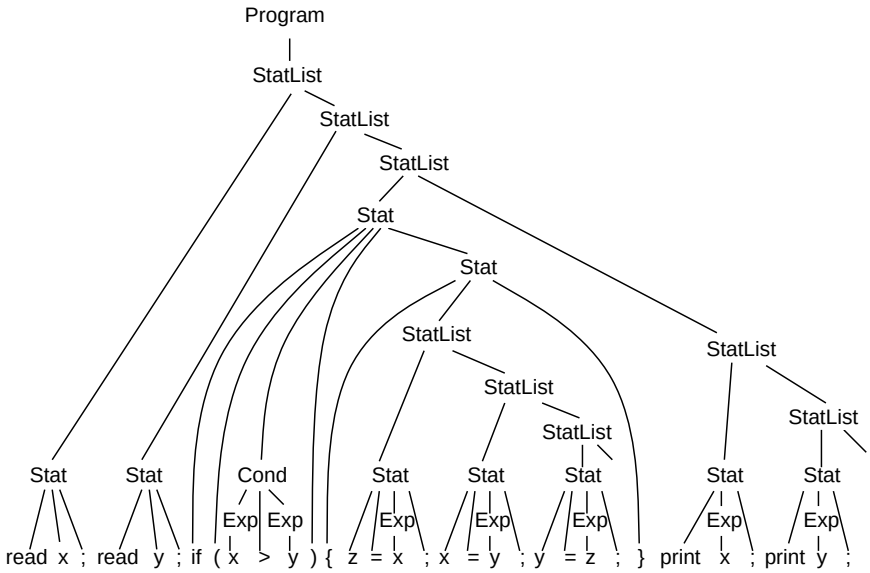

図 1: プログラムに対応する構文木

2 下 向 き 解 析

2.1 下向き解析と First/Follow

それではここで、図**??**の構文木を下向きに描く過程を少し細かく見てみます。まず構文木の根は出発記号 $Program$ に決まっています。次に $Program$ を左辺にもつ規則は 1 つしかないから、$StatList$ が根の直下の節となります。次ですが、ここで $StatList$ は 2 通りに置換できるので、そのどちらかを選択する必要があります。まず $Stat\ StatList$ に置換していいか考えましょう。その場合、入力の最初に $Stat$ がないといけませんが、具体的には最初の入力記号は `read` ですね。一方 $Stat$ は `read Ident ;` に置換できるので、こちらが正しそうだとわかります。そこで $Stat\ StatList$ を選び、この $Stat$ を `read Ident ;` に置換すると、これらが入力の最初の 3 つのトークンに対応します。次は先の $Stat\ StatList$ の $StatList$ 以下の対応に戻り、4 トークン目の `read` 以下とこれを対応させ…のように進めばよいでしょう。

これを見て「構文規則は有限だから可能なものを順にあてはめて入力との対応を検査していけばよさそうだ」と思うかも知れませんが、それでは困ります。なぜならそれだと「やってみてだめなら戻って別の枝を試す」(バックトラックする) ことになるので、入力トークンの数を n として、解析にかかる計算量のオーダが $O(n)$ より大きくなってしまいます。

そうではなく、「$StatList$ をどちらに置換するか」などの選択肢が現れたとき、先の方まで試してみることなく正しい選択を行う必要があるのです。上の過程を振り返ると、$Stat\ StatList$ を展開して行った先が `read`…になるから、こちらの枝を選んだのでした。そして、構文記号は有限個しかないので、全ての構文記号 A について「展開していくとどんな端記号から始まり得るか」の集合 (これを $First(A)$ と記す) をあらかじめ計算しておけます。併せて、各記号 A ごとに「その後に来ることができる端記号の集合」(これを $Follow(A)$ と記す) も計算しておくことにします (その用途は後述)。

2.2 First/Follow の計算

本節では $First$ と $Follow$ の計算方法を示します (プログラムにすると大変なので、アルゴリズムの形で説明します)。まず準備として、$Emp = \{X | X \in N \wedge \Rightarrow^* \epsilon\}$、つまり空列が導出可能な非端記号の集合を求めておきます。これは次の手順によりできます。

- Emp に $X \to \epsilon$ であるような生成規則を持つ非端記号 X をすべて入れる

- Emp がこれ以上変化しなくなるまで繰り返し
- 　$X \to Y_1 Y_2 \cdots Y_N$ において $^{\forall} Y_i \in Emp$ であるような生成規則を持つ X を Emp に追加
- 以上を繰り返し

ではこれを用いて、$First(X)$ は次のようにして求められます (なお、$\epsilon \in First(X)$ とは $X \in Emp$ であることを意味します)。

- 端記号なら、$Firxt(X) = \{X\}$ とする
- 全ての非端記号 X について $First(X) = \{\}$ とする
- $X \in Emp$ であれば、$First(X)$ に ϵ を追加
- すべての $First(X)$ が変化しなくなるまで繰り返し
- 　$X \to Y_1 Y_2 \cdots Y_N$ において i を $1 \cdots N$ の順に繰り返し
- 　　$First(X)$ に $First(Y_i) - \{\epsilon\}$ の各要素を追加
- 　　$Y_i \notin Emp$ なら繰り返しを抜け出す
- 　以上を繰り返し
- 以上を繰り返し

要するに、ある記号の先頭に来る端記号はその記号を左辺とする生成規則の右辺の先頭に来る端記号ですが、その先頭の記号が空列になり得るならその次、それも空列になり得るならさらにその次、…の先頭も加えるということです。

上では 1 つの記号 X について求めましたが、記号列 $X_1 X_2 \cdots X_N$ についての $First$ も次のように定めておきます。

- $First(X_1 X_2 \cdots X_N)$ を $\{\}$ とおく
- $N = 0$ または $^{\forall} X_i \in Emp$ であれば、$First(X_1 X_2 \cdots X_N)$ に ϵ を追加
- i を $1 \cdots N$ の順に繰り返し
- 　$First(X_1 X_2 \cdots X_N)$ に $First(X_i) - \{\epsilon\}$ の各要素を追加
- 　$X_i \notin Emp$ なら繰り返しを抜け出す
- 以上を繰り返し

$Follow(X)$ については X が非端記号のときだけ定義されます。その計算のアルゴリズムは次の通りです。

- すべての非端記号 X について $Follow(X) = \{\}$ とする
- X が開始記号であれば、$Follow(X)$ に\$ (入力終わりの印) を追加
- すべての $Follow(X)$ が変化しなくなるまで繰り返し
- 　$X \to Y_1 Y_2 \cdots Y_N$ において i を $1 \cdots N$ の順に繰り返し
- 　　Y_i が非端記号でなければ、次の周回に進む
- 　　$Follow(Y_i)$ に $First(Y_{i+1}, \cdots Y_N) - \{\epsilon\}$ の各要素を追加
- 　　$\epsilon \in First(Y_{i+1}, \cdots Y_N)$ なら、$Follow(Y_i)$ に $Follow(X)$ の各要素を追加
- 　以上を繰り返し
- 以上を繰り返し

では実際に、先に出て来た文法で $First/Follow$ がどうなるかを見てみましょう。

- $First(Prog) \to \{$ `if print read` $Ident$ `nil`
- $First(StatList) \to \{$ `if print read` $Ident$ `nil`

- $First(Stat) \rightarrow \{$ `if print read` $Ident$
- $First(Cond) \rightarrow Ident$
- $First(Expr) \rightarrow Ident$
- $Follow(Prog) \rightarrow \$$
- $Follow(StatList) \rightarrow \$\ \}$
- $Follow(Stat) \rightarrow$ `$ }` $\{$ `if print read` $Ident$
- $Follow(Cond) \rightarrow$ `)`
- $Follow(Expr) \rightarrow$ `< > ) ;`

2.3 LL(1) 構文解析器

何のために $First/Follow$ を求めていたかというと、下向き解析においてどの構文規則を選ぶべきかを判断するためでした。具体的には、非端記号 X を置き換えようとして $X \rightarrow Y_1 Y_2 \cdots Y_N$ という規則が複数あったときに、どれを選ぶかは $First(Y_1 Y_2 \cdots Y_N)$ を見ることで判断できます (より厳密にいえば、$\epsilon \in First(Y_1 Y_2 \cdots Y_N)$ である場合もあるので、その場合には $Follow(X)$ を使います)。

この情報は、予め文法に基づいて計算し、表の形で保持しておきます。このような、構文解析に必要な情報を集約した表のことを**構文解析表** (parsing table) と呼びます。先に出て来た文法と $First/Follow$ をもとに作成した構文解析表を図**??**に示しました。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	$	Ident	Iconst	(	)	<	>	{	}	=	;	read	print	if
Program	1	1						1				1	1	1
StatList	3	2						2	3			2	2	2
Stat		4						8				5	6	7
Cond		9,10	9,10											
Expr		11	12											

図 2: LL(1) 構文解析表

ここで説明している方法は、ソースコードを左から (先頭から) 順に (Left-to-right) 見ていき、生成される導出が最左導出 (Leftmost derivation) であり、そして常に「次の 1 記号」を見て動作を決めることから **LL(1)** 解析器と呼ばれています。その具体的な動作方法を説明しましょう。プログラム例を再掲します。

```
read x; read y; if(x > y) { z = x; x = y; y = z; } print x; print y;
```

解析器の構造とその動作を図**??**に示します。縦線が 2 本ありますが、その左側はスタックになっていて、左から要素をプッシュ/ポップします。そして右側はトークンが 1 つだけ見えていて、これが入力に現れるトークンです。

スタックに開始記号 (Program) が積まれていて、最初のトークン read が見えている状態から始まります。解析表を見ると、Program/read のところは「1」とあります。なので、生成規則 1(Program ::= StatList) が選択され、スタックから先頭要素 Program をポップして、代わりに右辺 StatList をプッシュします。これが 2 行目です。今度は解析表の StatList/read を見ると「2」ですから、生成規則「StatList ::= Stat StatList」が選ばれ、StatList がポップされてから右辺「Stat StatList」を (右から順に) プッシュします。これで 3 行目へ行きます。

こんどはスタック先頭が Stat なので、Stat/read を見ると「5」ですから、生成規則は「Stat ::= read Ident ;」であり、read がポップされて「read Ident ;」がプッシュされます。今度は先頭が端記

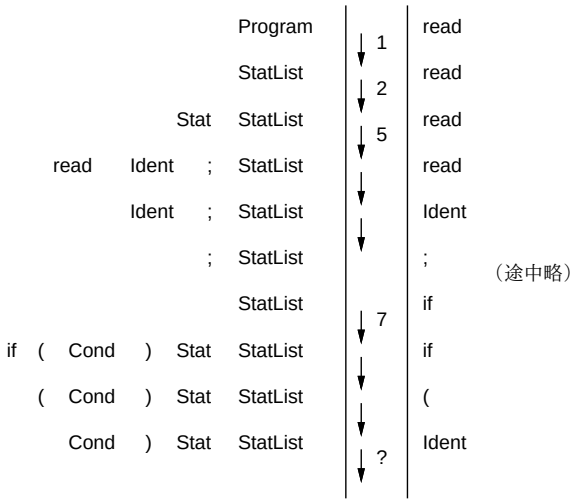

図 3: LL(1) 解析器の動作

号であり、それが先読みトークン read と一致しているので、両方とも取り除き (入力は読み進み)、次のトークンは Ident です。これもスタック先頭と一致していますから、再び両方とも取り除きます。次は「;」でこれも両方とも取り除きます。

長いので途中省略して、同様にもう 1 つの read 文も終わったものとして、次は「if」が見えます。スタック先頭は StatList なので、解析表の StatList/if を見ると「2」とあり、生成規則 2 が再度使われて「Stat StatList」となり、Stat/if で「7」が選ばれ、Stat が降ろされて「if (Cond) Stat」が積まれます。そして if、(が読み進められ、次は Cond/Ident です。

さてここですが、9 と 10 が両方書かれていますね。ということは、どちらに行けばよいか分からないわけです。それはそうで、該当する 2 つの生成規則は次のものです。

$$Cond ::= Expr \texttt{ < } Expr \mid Expr \texttt{ > } Expr$$

この 2 つの規則の右辺は先頭が同じものなので、1 トークンだけ見ていたのでは区別できません。つまり、この文法は LL(1) 解析器では解析できない (LL(1) 文法ではない)、ということになります (実際には解析表を作っている段階で分かります)。

実はこの場合は式というのは定数か変数 1 個だけなので、見えるトークンを 2 にすれば次の比較演算子が見えてどちらかは判断できます (つまりこの文法は LL(2) です)。

しかしより一般的には、式は任意の複雑さを持てるので、この方法は使えません。しかしこの問題解決方法はあります。それは、文法を次のように書き直せばよいのです。

$$Cond ::= Expr \ Cond1$$
$$Cond1 ::= \texttt{<} \ Expr \mid \texttt{>} \ Expr$$

このようにすれば、1 番目の規則をとりあえず選んで読み進み、式が済んだところで $Cond1$ を置き換えるところで、次の 1 トークンを見て規則を判別できます。

2.4 LL(1) 文法が持つ制約

文法が LL(1) でなくなる場合の 1 つは前述ように適用規則が一意に決まらない場合ですが、他に $A \Rightarrow^+ A\beta$ なる A が存在する場合 (これを文法に**左再帰性**がある、と言います) も、A を展開して行くとまた A になってしまい、無限に同じ規則の適用が続くだけで解析が進まなくなるため、LL(1) 解析器で解析できません。

文法が LL(1) でなくても、言語は同じままで文法を書き換えて LL(1) にできる場合があります。まず適用規則が一意に決まらない場合には、先の例のように共通部分を「くくり出す」ように規則を書き換えればよいです。

また、次のような左再帰性がある場合を考えてみます。

$A \to A\beta$
$A \to \gamma$

これから生成される言語は $\gamma\beta\beta\ldots\beta$ の形になります。そこで生成規則を次のように書き換えれば、言語は同じままで左再帰性のない文法になります。

$A \to \gamma A'$
$A' \to \beta A'$
$A' \to \varepsilon$

このような直接の左再帰でない場合でも、中間に現れる規則を展開して埋め込むことで直接左再帰に書き換えてから同様に処理すればよいのです。このような書き換えで LL(1) にできない文法も存在しますが、プログラミング言語の構文として現れることはまれです。

ただ、文法の書き換えで問題なのは、認識される言語は同じでも文法記述が理解しにくいものとなり、また構文木に沿った後段の処理も記述しにくくなってしまう点です。これについて対応する方法は、再帰下降解析と合わせて説明します。

3 LL(1) 解析器

ではここで、解析表を使った LL(1) 解析器を構成してみましょう。木構造を作るのは面倒なので、実際には認識器です。また、先の曖昧さの問題を避けるため、規則 10 は削除しておきます (比較演算子「>」は使わなこととします)。

まず、字句解析は JFlex を使いますが、字句解析と受け渡すトークン番号をこれまでは端記号だけにしていたのに対し、今回は非端記号まで含めて番号を振ります。このため `Symbol` という次のクラスを使います。

```
public class Symbol {
  public static final int NL = 0, EOF = 1, IDENT = 2, ICONST = 3,
    LPAR = 4, RPAR = 5, LT = 6, GT = 7, LBRA = 8, RBRA = 9,
    ASSIGN = 10, SEMI = 11, READ = 12, PRINT = 13, IF = 14,
    Program = 15, StatList = 16, Stat = 17, Cond = 18, Expr = 19;
  public static final int N = 15;
}
```

`EOF` も表にいれる都合上、正の値にしています。また、端記号を前の方にあつめて、その後ろの値を非端記号とし、境界を定数 `N` として用意しました (後で端記号かどうか判定するのに使う)。

次にこれを参照した JFLex のソースファイルを示します。必要な記号類が追加されているだけで、とくに変わったことはありません。

```
%%
%class Lexer
%int
L = [A-Za-z_]
D = [0-9]
Ident = {L}({L}|{D})*
Iconst = [-+]?{D}+
Blank = [ \t\n]+
%%
```

```
{Blank}   { /* ignore */ }
\(        { return Symbol.LPAR; }
\)        { return Symbol.RPAR; }
;         { return Symbol.SEMI; }
\{        { return Symbol.LBRA; }
\}        { return Symbol.RBRA; }
=         { return Symbol.ASSIGN; }
\>        { return Symbol.GT; }
\<        { return Symbol.LT; }
read      { return Symbol.READ; }
print     { return Symbol.PRINT; }
if        { return Symbol.IF; }
{Ident}   { return Symbol.IDENT; }
{Iconst}  { return Symbol.ICONST; }
```

今回はこの Lexer をそのまま使うのでなく、前の回でやった Toknizer と同じインタフェースになるように、下請けとして Lexer を呼ぶクラス Tokenizer を作りました。

```
class Tokenizer {
  Lexer lex;
  int tok, line = 1;
  boolean eof = false;
  public Tokenizer(String s) throws Exception {
    lex = new Lexer(new FileReader(s));
    tok = lex.yylex();
  }
  public boolean isEof() { return tok == Lexer.YYEOF; }
  public int curTok() { return tok; }
  public String curStr() { return lex.yytext(); }
  public int curLine() { return line; }
  public boolean chk(int t) { return tok == t; }
  public void fwd() {
    if(isEof()) { return; }
    try {
      tok = lex.yylex();
      while(tok == Symbol.NL) { ++line; tok = lex.yylex(); }
    } catch(IOException ex) { tok = Lexer.YYEOF; }
  }
  public boolean chkfwd(int t) {
    if(chk(t)) { fwd(); return true; } else { return false; }
  }
}
```

fwd() で複雑なことをやっていますが、おもに改行がきたときに行カウントを増やすためです。このほか、EOF のときに Symbol.EOF を使う必要がありますが、それは main() 側でやるようにしました。

では本体部分です。プログラムの他に、解析表とそれぞれの規則の右辺が必要です。分かりやすさのため、規則は左辺と右辺を並べた「配列の配列」として記述しました。番号 (添字) は先の文法と一

致させてあります。

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

public class Sam61 {
  static int[][] rules = {
    { },
    { Symbol.Program, Symbol.StatList }, //1
    { Symbol.StatList, Symbol.Stat, Symbol.StatList }, //2
    { Symbol.StatList }, //3
    { Symbol.Stat, Symbol.IDENT, Symbol.ASSIGN, Symbol.Expr, Symbol.SEMI },//4
    { Symbol.Stat, Symbol.READ, Symbol.IDENT, Symbol.SEMI }, //5
    { Symbol.Stat, Symbol.PRINT, Symbol.Expr, Symbol.SEMI }, //6
    { Symbol.Stat, Symbol.IF, Symbol.LPAR, Symbol.Cond,
                   Symbol.RPAR, Symbol.Stat }, //7
    { Symbol.Stat, Symbol.LBRA, Symbol.StatList, Symbol.RBRA }, //8
    { Symbol.Cond, Symbol.Expr, Symbol.LT, Symbol.Expr }, //9
    { Symbol.Cond, Symbol.Expr, Symbol.GT, Symbol.Expr }, //10
    { Symbol.Expr, Symbol.IDENT }, //11
    { Symbol.Expr, Symbol.ICONST }, //12
  };
```

解析表は非端記号のところだけ必要ですが、添字の位置を合わせるため始めの方に空の配列をいれています。また、トークンは 1 から始まるので最初の要素として 0 が詰めてあります。内容は前に示した LL(1) の解析表と同じですが、10 番の規則 (>に対応) は削除してあります。

```
  static int [][] ptab = {
    {},{},{},{}, {},{},{},{}, {},{},{},{}, {},{},{},
    { 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1 },
    { 0, 3, 2, 0, 0, 0, 0, 0, 2, 3, 0, 0, 2, 2, 2 },
    { 0, 0, 4, 0, 0, 0, 0, 0, 8, 0, 0, 0, 5, 6, 7 },
    { 0, 0, 9, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 0, 0,11,12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
  };
```

最後に main() を見ていただきましょう。最初にスタックに Program をプッシュした状態からはじめ、ループの中でスタックの先頭が端記号か非端記号かで分かれます。非端記号であれば、次のトークンとの一致を確認してスタックを取り降ろし、入力も進めます (不一致ならエラー終了します)。

```
  public static void main(String[] args) throws Exception {
    Tokenizer tok = new Tokenizer(args[0]);
    Stack<Integer> stk = new Stack<Integer>();
    stk.push(Symbol.Program);
    while(stk.size() > 0) {
//    System.out.printf("%s : %s\n", stk.toString(), tok.curStr());
      if(stk.peek() < Symbol.N) {
        if(!tok.chkfwd(stk.pop())) {
```

```
          System.err.printf("token mismatch: %s at %d\n",
            tok.curStr(), tok.curLine()); return;
        }
      } else {
        int t = tok.curTok(); if(tok.isEof()) { t = Symbol.EOF; }
        int r = ptab[stk.peek()][t];
        if(r == 0) {
          System.err.printf("cannot determine rule for %d: %s at %d\n",
            stk.peek(), tok.curStr(), tok.curLine()); return;
        }
        System.out.printf("rule: %d\n", r);
        int[] a = rules[r];
        stk.pop();
        for(int i = a.length-1; i > 0; --i) { stk.push(a[i]); }
      }
    }
  }
}
// 後ろに Tokenizer をいれる
```

非端記号の場合は解析表を見て動作を決めます。具体的には、解析表を引くと生成規則番号が分かるので、スタックの先頭は取り降ろして成績規則の右辺の内容を右から順に積みます。もし番号が 0 ならエラーです。

先のプログラムの「>を「<」に変更したもので実行してみました。次のように、確かに生成規則が順に出力されています。

```
% java Sam61 test.min
rule: 1
rule: 2
rule: 5
...
rule: 3
%
```

演習 6-1 例題をそのまま動かせ。簡単なプログラムを何通りか作って動かしてみて、構文木も描いた上で照合し、構文規則番号が正しいことを確認すること。

演習 6-2 条件演算子「>」も扱えるように変更せよ。本文で説明したように文法を変更したものとして、それに対応して解析表や規則を変更すればできる。

演習 6-3 次のような算術式の文法を認識するように LL(1) 解析器を変更してみよ。文法を LL(1) になるよう書き換えてから $First/Follow$ を作り、解析表を作ること。

$$
\begin{aligned}
&Prog \ ::= \ Expr \\
&Expr \ ::= \ Term \ \texttt{+} \ Expr \mid Term \ \texttt{-} \ Expr \\
&Term \ ::= \ Fact \ \texttt{*} \ Term \mid Fact \ \texttt{/} \ Term \\
&Fact \ ::= \ Ident \mid Iconst \mid \texttt{(} \ Expr \ \texttt{)}
\end{aligned}
$$

演習 6-4 前問の文法だと、演算子が右結合になってしまう (なぜか?)。これを避けるためには、次の文法を使えばよい。

$Prog$ ::= $Expr$
$Expr$ ::= $Expr$ `+` $Term$ | $Expr$ `-` $Term$
$Term$ ::= $Term$ `*` $Fact$ | $Term$ `/` $Fact$
$Fact$ ::= $Ident$ | $Iconst$ | `(` $Expr$ `)`

しかし今度は左再帰を含む文法なのでそのままでは LL(1) 解析器を作れない。左再帰を解消するためには次のように書き換えることが 1 つの方法である。

$Prog$ ::= $Expr$
$Expr$ ::= $Term$ $Expr0$
$Expr0$ ::= ϵ | `+` $Term$ $Expr0$ | `-` $Term$ $Expr0$
$Term$ ::= $Fact$ $Term0$
$Term0$ `::=` ϵ | `*` $Fact0$ | `/` $Fact0$
$Fact$ ::= $Ident$ | $Iconst$ | `(` $Expr$ `)`

この文法に対して LL(1) 解析器を構成せよ。

演習 6-5 好きな文法を決めて、その文法を認識する LL(1) 解析器を作れ。

4 課題 6A

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# 6」、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. First、Follow の計算方法と LL(1) 解析器の原理について納得しましたか。
  Q2. 左再帰やその除去など、文法を LL(1) 文法にするための変形について納得しましたか。
  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。