システムソフトウェア特論'17 # 7 構文解析 (2)

久野　靖 *

2017.9.13

1　拡張 BNF と構文図

ここまで BNF は文脈自由文法の定義に忠実に従った形で扱ってきました (「|」については、左辺が同じ複数の定義と同じことなので同等です)。しかし、プログラム言語の構文として考えると、正規表現に類似した次のようなものが書けるとより便利です。

(α)* — α の 0 回以上の繰り返し
(α)+ — α 1 回以上の繰り返し
(α | β) — α または β (右辺の途中での選択肢)
[α] — α があってもなくてもよい

繰り返しの書き方については、... で表すなど別の流儀もあります。一般に、このような追加の記法を取り入れた BNF のことを**拡張 BNF** と呼びます。今回例題として実装する小さな言語の構文を拡張 BNF で記述したものを示します。

prog ::= (*stat*)*
stat ::= *ident* `=` *expr* `;` | `read` *ident* `;` | `print` *expr* `;`
　　| `if (` *expr* `)` *stat* | `while (` *expr* `)` *stat* | `{` *prog* `}`
expr ::= *term* (`+` *expr* | `-` *expr*)*
term ::= *fact* (`*` *term* | `/` *term*)*
fact ::= *ident* | *iconst* | `(` *expr* `)`

BNF はどうしても数式っぽく見えますが、これと同等のものを図的に表す**構文図** (syntax diagram) または railroad diagram と呼ばれる記法があります。これは Niklaus Wirth が Pascal 言語のマニュアルで始めたものです。

構文図では、1 つの端記号の規則ごとに 1 つの始点と終点を持つ有向グラフを描きます。規則中に現れる端記号は円や長円、非端記号は長方形で表し、それらに間をつながりを表す矢線で結びます。そして、始点から終点まで矢線に沿って通れるとき、記号をその順で並べたものが生成できることを意味します。図 1 に上の文法を構文図にしたものを示します。

拡張 BNF でも構文図でも、「または」による部分的な分岐と合流、および規則の途中位置に戻る「ループ」が表せることが BNF と比べた場合の特徴となります。

2　再帰下降解析

再帰下降解析 (recursive descent parser) については既に取り上げましたが、その名前通り下向き解析の 1 つの手法であり、手でパーサを構成できるのでツールが使えない場合に有力な選択肢となります。改めて整理すると、その要点は次のようになります。

*電気通信大学 情報理工学研究科

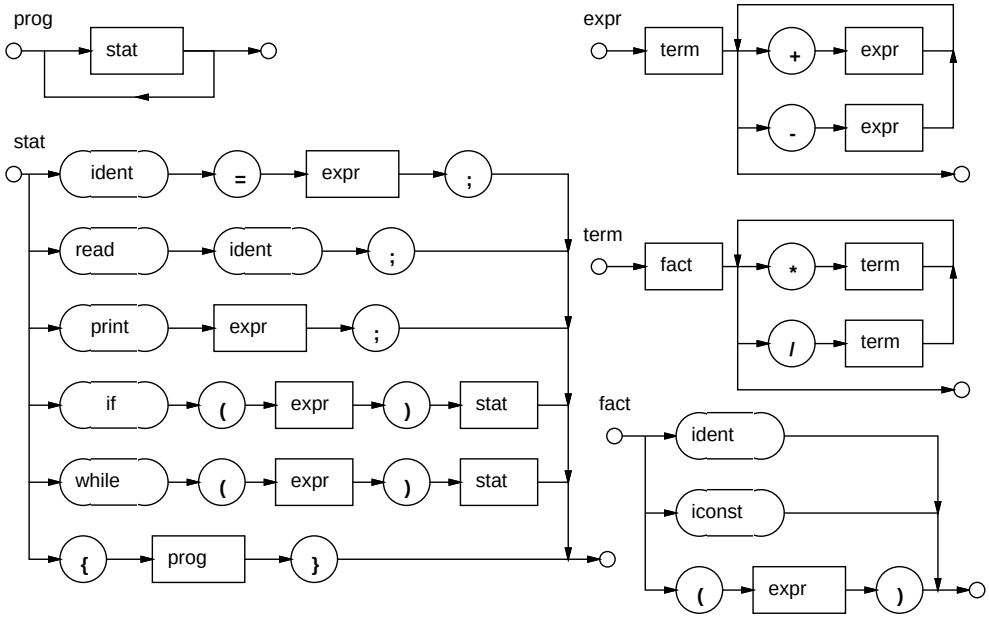

図 1: 構文図による小さな言語の文法

- 非端記号 N に対応して 1 つの手続き P を作る。
- P の中では生成規則の右辺に対応して入力を読み進める。このとき、端記号は自分でトークンを読み進めるが、非端記号については対応する箇所でその記号に対応する手続きを呼び出す。
- 適用する生成規則に複数の選択肢がある場合は、入力に基づいて適切な選択肢を選んで上記をおこなう。

再帰下降という名称は、呼び出し関係の構造がちょうど構文木を上から下に向かってたどるのと同じになること、そして構文規則の再帰性に対応して手続が直接/間接に自分を呼び出す (再帰呼び出しを行なう) ことによります。

再帰下降解析は、その本体が手続きのコードとして実行されることから、前に取り上げた LL(1) 解析器と比較して次のような利点を持ちます。

- 選択肢を選ぶときに先読み記号を多くしたり (LL(k) 文法に対応することになります)、先読み記号以外の情報に基づいたりでき、扱える文法クラスが広くなる。
- 解析の作業と合わせて構文木の生成や記号表への登録などの作業を柔軟におこなえる。
- 規則の途中での分岐/合流やその一部分の反復など、拡張 BNF や構文図で表現されるような機能を直接的に実装できる。

3 小さい言語のツリーインタプリタ

本節以下では再帰下降解析の特徴を活かして、先に文法を示した言語の解析をおこないつつ、抽象構文木を構成するような再帰下降解析器を作ってみます。

抽象構文木としては、前に扱った `eval()` メソッドを持つノード群をそのまま利用するので、それを直ちに実行してみることができます。つまり小さな言語のツリーインタプリタができるわけです。以前のものをそのまま使うので、取り扱う値は整数のみで、論理値はなく、C 言語と同様に「0 でない整数値は真」として条件を扱います。

まず前回同様、各トークンに固有の番号を割り当てるために 1 つクラスを作ります (今回は非端記号は不要なので `Token` というクラス名にしました)。このクラスはコードは一切なく、定数を定義するだけです。

```
public class Token {
  public static final int EOF = -1, NL = 0, IDENT = 1, ICONST = 2,
    STR = 3, LPAR = 4, RPAR = 5, LBRA = 6, RBRA = 7, SEMI = 7,
    ASSIGN = 8, PLUS = 9, MINUS = 10, ASTER = 11, SLASH = 12,
    WHILE = 13, IF = 14, READ = 15, PRINT = 16;
}
```

次に、JFlex ソースを示します。各種の記号や予約語をそれぞれ認識するようにしただけで、根本は変わっていません。予約語よりも識別子の規則が後なのは、JFlex では複数のあてはまりがある場合は「上に」書いたものが優先されるためです。

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

\=        { return Token.ASSIGN; }
\+        { return Token.PLUS; }
\-        { return Token.MINUS; }
\*        { return Token.ASTER; }
\/        { return Token.SLASH; }
\;        { return Token.SEMI; }
\(        { return Token.LPAR; }
\)        { return Token.RPAR; }
\{        { return Token.LBRA; }
\}        { return Token.RBRA; }
while     { return Token.WHILE; }
if        { return Token.IF; }
read      { return Token.READ; }
print     { return Token.PRINT; }
{Ident}   { return Token.IDENT; }
{Iconst}  { return Token.ICONST; }
{String}  { return Token.STR; }
{Newline} { return Token.NL; }
{Blank}   { /* ignore */ }
```

JFlex が生成した Lexer クラスを中で保持して外部とインタフェースするクラス Tokenizer については、前回と同じなので略します。

抽象構文木のクラス群も前にやったものと同一ですが、Tree というクラスの中に入れて、外部からは Tree.Node、Tree.Add などの名前で参照できるようにします (そのため、各クラスの冒頭に public 修飾子を追加します)。また、変数の値のための表もここで保持します。

```
import java.util.*;
public class Tree {
  public static Map<String,Integer> vars = new TreeMap<String,Integer>();
  public abstract static class Node {
    List<Node> child = new ArrayList<Node>();
    public void add(Node n) { child.add(n); }
    public abstract int eval();
  }
  public static class Lit extends Node {
    int val;
    public Lit(int v) { val = v; }
    public int eval() { return val; }
    public String toString() { return ""+val; }
  }
  public static class Var extends Node {
    String name;
    public Var(String n) { name = n; }
    public int eval() { return vars.get(name); }
    public String toString() { return name; }
  }
  public abstract static class BinOp extends Node {
    String op;
    public BinOp(String o, Node n1, Node n2) { op = o; add(n1); add(n2); }
    public String toString() { return "("+child.get(0)+op+child.get(1)+")"; }
  }
  public static class Add extends BinOp {
    public Add(Node n1, Node n2) { super("+", n1, n2); }
    public int eval() { return child.get(0).eval() + child.get(1).eval(); }
  }
  public static class Sub extends BinOp {
    public Sub(Node n1, Node n2) { super("-", n1, n2); }
    public int eval() { return child.get(0).eval() - child.get(1).eval(); }
  }
  public static class Mul extends BinOp {
    public Mul(Node n1, Node n2) { super("+", n1, n2); }
    public int eval() { return child.get(0).eval() * child.get(1).eval(); }
  }
  public static class Div extends BinOp {
    public Div(Node n1, Node n2) { super("/", n1, n2); }
    public int eval() { return child.get(0).eval() / child.get(1).eval(); }
  }
  public static class Mod extends BinOp {
    public Mod(Node n1, Node n2) { super("%", n1, n2); }
    public int eval() { return child.get(0).eval() % child.get(1).eval(); }
  }
  public static class Eq extends BinOp {
```

```
  public Eq(Node n1, Node n2) { super("==", n1, n2); }
  public int eval() { return child.get(0).eval()==child.get(1).eval()?1:0; }
}
public static class Ne extends BinOp {
  public Ne(Node n1, Node n2) { super("!=", n1, n2); }
  public int eval() { return child.get(0).eval()!=child.get(1).eval()?1:0; }
}
public static class Gt extends BinOp {
  public Gt(Node n1, Node n2) { super(">", n1, n2); }
  public int eval() { return child.get(0).eval()>child.get(1).eval()?1:0; }
}
public static class Ge extends BinOp {
  public Ge(Node n1, Node n2) { super(">=", n1, n2); }
  public int eval() { return child.get(0).eval()>=child.get(1).eval()?1:0; }
}
public static class Lt extends BinOp {
  public Lt(Node n1, Node n2) { super("<", n1, n2); }
  public int eval() { return child.get(0).eval()<child.get(1).eval()?1:0; }
}
public static class Le extends BinOp {
  public Le(Node n1, Node n2) { super("<=", n1, n2); }
  public int eval() { return child.get(0).eval()<=child.get(1).eval()?1:0; }
}
public static class And extends BinOp {
  public And(Node n1, Node n2) { super("&&", n1, n2); }
  public int eval() {
    int v = child.get(0).eval();
    if(v == 0) { return 0; } else { return child.get(1).eval(); }
  }
}
public static class Or extends BinOp {
  public Or(Node n1, Node n2) { super("||", n1, n2); }
  public int eval() {
    int v = child.get(0).eval();
    if(v != 0) { return v; } else { return child.get(1).eval(); }
  }
}
public abstract static class UniOp extends Node {
  String op;
  public UniOp(String o, Node n1) { op = o; add(n1); }
  public String toString() { return "("+op+child.get(0)+")"; }
}
public static class Not extends UniOp {
  public Not(Node n1) { super("!", n1); }
  public int eval() { return child.get(0).eval()==0?1:0; }
}
```

```
public static class Assign extends Node {
  Var v1; Node n1;
  public Assign(Var v, Node n) { v1 = v; n1 = n; }
  public int eval() {
    int v = n1.eval(); vars.put(v1.toString(), v); return v;
  }
  public String toString() { return v1+"="+n1; }
}
public static class Seq extends Node {
  public Seq(Node... a) { for(Node n:a) { child.add(n); } }
  public int eval() {
    int v = 0;
    for(Node n:child) { v = n.eval(); }
    return v;
  }
  public String toString() {
    String s = "{\n";
    for(Node n:child) { s += n.toString() + ";\n"; }
    return s + "}";
  }
}
public static class Read extends Node {
  Var v1;
  public Read(Var v) { v1 = v; }
  public int eval() {
    System.out.print(v1+"? ");
    Scanner sc = new Scanner(System.in);
    String str = sc.nextLine();
    int i = Integer.parseInt(str);
    vars.put(v1.toString(), i); return i;
  }
  public String toString() { return "read "+v1; }
}
public static class Print extends Node {
  public Print(Node n1) { child.add(n1); }
  public int eval() {
    int v = child.get(0).eval(); System.out.println(v); return v;
  }
  public String toString() { return "print "+child.get(0); }
}
public static class While extends Node {
  public While(Node n1, Node n2) { child.add(n1); child.add(n2); }
  public int eval() {
    int v = 0;
    while(child.get(0).eval() != 0) { v = child.get(1).eval(); }
    return v;
```

```
    }
    public String toString() {
      return "while("+child.get(0)+")"+child.get(1);
    }
  }
  public static class If1 extends Node {
    public If1(Node n1, Node n2) { child.add(n1); child.add(n2); }
    public int eval() {
      int v = 0;
      if(child.get(0).eval() != 0) { v = child.get(1).eval(); }
      return v;
    }
    public String toString() { return "if("+child.get(0)+")"+child.get(1); }
  }
}
```

では本体です。Tokenizer のインスタンスは変数 tok に格納してクラス内どこからでもアクセスできるようにします。main では Tokenizer を生成して最初の手続き prog() を呼び出し、正しく結果が得られたらそれを表示して実行します。

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

public class Sam71 {
  static Tokenizer tok;
  public static void main(String[] args) throws Exception {
    tok = new Tokenizer(args[0]);
    Tree.Node n = prog();
    if(n != null) { System.out.println(n.toString()); n.eval(); }
  }
```

ここから再帰下降解析の各手続きになります。前にやったときは認識器だったので OK かどうかを boolean で返していましたが、こんどは構文木を返したいので、OK なら対応する構文木オブジェクトを返し、OK でなければ nil を返す、という形にします。そのため、各手続きの返値は Tree.Node になります。

以下の各手続きを読むときは、図 1 の構文図または文法を見ながらにしてください。また、表 1 にこの文法の各非端記号の $First/Follow$ を挙げておきます。

表 1: 小さな言語の非端記号の $First/Follow$

端記号	$First$	Follow
$prog$	*Ident* read print if while {	} $
$stat$	*Ident* read print if while {	*Ident* read print if while { } $
$expr$	*Ident Iconst* (	*Ident* read print if while { } $) ;
$term$	*Ident Iconst* (	*Ident* read print if while { } $) ;
$fact$	*Ident Iconst* (	*Ident* read print if while { } $) ;

まず prog() ですが、最初は空の Seq を作ります。そして、中では繰り返し stat() を呼び、返さ

れた値を Seq に追加していきます。それで、いつまで繰り返し呼べばいいでしょうか? 厳密には次のようにするべきです。

- 次の記号が $First(stat)$ である間 stat() を呼び、
- 上記でなくなって、かつ次の記号が $Follow(prog)$ のとき終了。

ただ、$First(stat)$ は沢山あって面倒ですし、$First(stat)$ のチェックは stat() の中でどのみちあるので、ここでは後者の条件だけをチェックしています。stat() を呼んで失敗したときはエラーなのでその旨出力して nil を返します。

```
static Tree.Node prog() {
  Tree.Seq n1 = new Tree.Seq();
  while(!tok.chk(Token.RBRA) && !tok.chk(Token.EOF)) {
    Tree.Node n2 = stat();
    if(n2 != null) { n1.add(n2); continue; }
    System.err.printf("%d: error stat at: %s\n", tok.curLine(), tok.curStr());
    return null;
  }
  return n1;
}
```

stat() は文に対応し、文は代入文、read 文、print 文、while 文、if 文、ブロックのいずれかです。それぞれについて、その中身が正しく認識できたらそのノードを返し、どこかで失敗したら最後に来て null を返します。

```
static Tree.Node stat() {
  if(tok.chk(Token.IDENT)) {
    Tree.Var n1 = new Tree.Var(tok.curStr()); tok.fwd();
    boolean b1 = tok.chkfwd(Token.ASSIGN);
    Tree.Node n2 = expr();
    boolean b2 = tok.chkfwd(Token.SEMI);
    if(b1 && n2 != null) { return new Tree.Assign(n1, n2); }
  } else if(tok.chkfwd(Token.READ)) {
    Tree.Var n1 = null;
    if(tok.chk(Token.IDENT)) { n1 = new Tree.Var(tok.curStr()); tok.fwd(); }
    if(n1 != null && tok.chkfwd(Token.SEMI)) { return new Tree.Read(n1); }
  } else if(tok.chkfwd(Token.PRINT)) {
    Tree.Node n1 = expr();
    if(n1 != null && tok.chkfwd(Token.SEMI)) { return new Tree.Print(n1); }
  } else if(tok.chkfwd(Token.IF)) {
    boolean b1 = tok.chkfwd(Token.LPAR);
    Tree.Node n1 = expr();
    boolean b2 = tok.chkfwd(Token.RPAR);
    Tree.Node n2 = stat();
    if(b1 && n1 != null && b2 && n2 != null) { return new Tree.If1(n1,n2); }
  } else if(tok.chkfwd(Token.WHILE)) {
    boolean b1 = tok.chkfwd(Token.LPAR);
    Tree.Node n1 = expr();
```

```
      boolean b2 = tok.chkfwd(Token.RPAR);
      Tree.Node n2 = stat();
      if(b1 && n1!=null && b2 && n2!=null) { return new Tree.While(n1, n2); }
    } else if(tok.chkfwd(Token.LBRA)) {
      Tree.Node n1 = prog();
      if(tok.chkfwd(Token.RBRA)) { return n1; }
    }
    System.err.printf("%d: error stat at: %s\n", tok.curLine(), tok.curStr());
    return null;
  }
```

expr() はまず term() を呼び、次に $Follow(expr)$ がくるまで繰り返し、加算/減算のノードを作ります。木構造でははじめに出て来た部分式ほど深い位置になることに注意。

```
  static Tree.Node expr() {
    Tree.Node n = term();
    while(n != null && !tok.chk(Token.RPAR) && !tok.chk(Token.SEMI)) {
      if(tok.chkfwd(Token.PLUS)) {
        Tree.Node n1 = term(); n = (n1 == null) ? null : new Tree.Add(n, n1);
      } else if(tok.chkfwd(Token.MINUS)) {
        Tree.Node n1 = term(); n = (n1 == null) ? null : new Tree.Sub(n, n1);
      } else {
        n = null;
      }
    }
    if(n != null) { return n; }
    System.err.printf("%d: error expr at: %s\n",tok.curLine(),tok.curStr());
    return null;
  }
```

term() は expr() と同様ですが、$Follow(term)$ の方が要素が多いのでそこが違っています。

```
  static Tree.Node term() {
    Tree.Node n = fact();
    while(n != null && !tok.chk(Token.RPAR) && !tok.chk(Token.SEMI) &&
                       !tok.chk(Token.PLUS) && !tok.chk(Token.MINUS)) {
      if(tok.chkfwd(Token.ASTER)) {
        Tree.Node n1 = term(); n = (n1 == null) ? null : new Tree.Mul(n, n1);
      } else if(tok.chkfwd(Token.SLASH)) {
        Tree.Node n1 = term(); n = (n1 == null) ? null : new Tree.Div(n, n1);
      } else {
        n = null;
      }
    }
    if(n != null) { return n; }
    System.err.printf("%d: error term at: %s\n",tok.curLine(),tok.curStr());
    return null;
  }
```

最後に fact() は因子で、変数、整数定数、かっこで囲まれた式のいずれかになります。

```
  static Tree.Node fact() {
    if(tok.chk(Token.IDENT)) {
      Tree.Node n = new Tree.Var(tok.curStr()); tok.fwd(); return n;
    } else if(tok.chk(Token.ICONST)) {
      Tree.Node n = new Tree.Lit(Integer.parseInt(tok.curStr()));
      tok.fwd(); return n;
    } else if(tok.chkfwd(Token.LPAR)) {
      Tree.Node n = expr();
      if(tok.chkfwd(Token.RPAR)) { return n; }
      System.err.printf("%d: no closing ')'\n", tok.curLine());
      return null;
    }
    System.err.printf("%d: error term at: %s\n", tok.curLine(), tok.curStr());
    return null;
  }
}
// Tokenizer をここに
```

では、動かすプログラムを見てみましょう。整数を入力すると、その値から 1 まで値を減らしながら順に打ち出しますが、ただし 5 だけは打ち出しません。

```
% cat test.min
read x;
while(x) {
  if(x - 5) { print x; }
  x = x - 1;
}
%
```

実行例は次の通り。

```
% java Sam52 test.min
{
read x;
while(x){
if((x-5)){
print x;
};
x=(x-1);
};
}
x? 7
7
6
4
3
```

```
2
1
%
```

演習 7-1 例題をそのまま動かし、いくつか簡単なプログラムを実行してみよ。できたら、言語に次のような変更を行なってみよ。

a. 今の版では read は「read 文」だが、計算の途中で「read」という項が現れたらそこで入力が行なわれるという形に変更する。
b. do-while 文のような「末尾で条件を調べる」ループ文を追加する。
c. if 文に else 部がつけられるようにする。もちろん else 部があってもなくてもよいようにすること。
d. if 文を Ruby 等のように「if...elsif...elsif...end」の形のものに変更する。
e. 今の版では代入は「代入文」であるが、C 言語のように「代入演算子」にする。
f. その他、自分の好きな構文の変更をおこなう。

演習 7-2 例題では計算式が普通に「加減算より乗除算が強く、左から計算する」方式になっているが、次のような変更を行なってどんな感じか試してみよ。

a. 乗除算より加減算の方が強く結び付くように変更する。
b. べき乗演算子「**」を追加する。2 ** 3 ** 2 は「2 ** (3 ** 2)」と同じ、つまり右側を先に計算する。
c. 加減算、乗除算とも「右側から計算する」ように変更する。
d. 計算式を前置記法で記述する。「x + 1」でなく「+ x 1」のように書くことになる。
e. その他、自分の好きな計算式記法の変更をおこなう。

演習 7-3 自分の好きなミニ言語の構文を設計し、拡張 BNF や構文図で記述したのち、実装してみよ。

4 課題 7A

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# 7」、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. 構文図の読み方や、それを元にして再帰下降解析器を組み立てるやり方が分かりましたか。
Q2. 再帰下降解析器とツリーインタプリタによる言語の実装を動かしてみてどのように思いましたか。
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。