

システムソフトウェア特論'19 # 9b コンパイラコンパイラ (続)

久野 靖 (電気通信大学)

2019.1.9

(今回は、前回の要点を復習し、演習に時間を多く割り当てます。)

SableCC の記述ファイル

SableCC の記述ファイルは字句解析と構文解析の記述を一緒に入れるようになっています。簡単な例を見ましょう。

SableCC の記述ファイル (2)

```
Package sam94;
```

Helpers

```
digit = ['0'..'9'] ;  
lcase = ['a'..'z'] ;  
ucase = ['A'..'Z'] ;  
letter = lcase | ucase ;
```

Tokens

```
iconst = ('+'|'-'|) digit+ ;  
blank = (' '|13|10)+ ;  
if = 'if' ;  
read = 'read' ;  
print = 'print' ;  
semi = ';' ;  
assign = '=' ;  
plus = '+' ;  
minus = '-' ;  
aster = '*' ;  
slash = '/' ;  
lt = '<' ;  
gt = '>' ;  
lbra = '{' ;  
rbra = '}' ;  
lpar = '(' ;  
rpar = ')' ;  
ident = letter (letter|digit)* ;
```

Ignored Tokens

blank;

Productions

```
prog = {stlist} stlist
      ;
stlist = {stat} stlist stat
        | {empty}
        ;
stat = {assign} ident assign expr semi
      ;
expr = {gt} [left]:nexp gt [right]:nexp
      | {lt} [left]:nexp lt [right]:nexp
      | {one} nexp
      ;
nexp = {add} nexp plus term
      | {sub} nexp minus term
      | {one} term
      ;
term = {mul} term aster fact
      | {div} term slash fact
      | {one} fact
      ;
fact = {ident} ident
      | {iconst} iconst
      ;
```

覚えておくべき重要なことは次の通り。

- 文法規則ごとに {...} で囲んで名前をつける。その規則はコード上では「BNFの左辺が xxx、規則名が yyy のとき、AXxxYyy で指定」。すべての規則ごとにそのノードオブジェクトがある。
- ノードオブジェクトのメソッドとして、右辺に zzz というものがあれば、対応して getZzz() というメソッドがある。これを使ってノードをたどれる。さらに端記号の場合、getText() というメソッドもある(トークン文字列が取得できる)。このため、右辺に同じ記号が2つあるときはエラーになるので、少なくとも片方に「[名前]:」を指定して別の名前にする。
- 構文木を深さ優先でたどる Visitor オブジェクトの基底クラス DepthFirstAdaptor が自動生成される。このクラスのサブクラスを作り、必要なメソッドをオーバーライドして自分用の処理を記述。
- オーバライドするメソッドは次のいずれか。
 - public void caseAXxxYyy(AXxxYyy node) — たどるときに呼ばれる。子ノードのたどりは自分でおこなう。その場合の呼び出しは「node.getZzz().apply(this)」。
 - public void inAXxxYyy(AXxxYyy node) — 自動でたどるときに、入口で呼ばれる。
 - public void outAXxxYyy(AXxxYyy node) — 自動でたどるときに、入口で呼ばれる。ボトムアップで処理する場合はこれをおもに使う。
- 上→下、下→上に値を渡すときはそれぞれ、Inテーブル、Outテーブルにノードを鍵として Object 値を登録することでうけわたす。「setIn(ノード, 値)」「getIn(ノード)」、「setOut(ノード, 値)」「getOut(ノード)」

SableCC の記述ファイル (3)

ここで、数式の計算をするだけの Visitor を作ってみる。

```
package sam94;
import sam94.analysis.*;
import sam94.node.*;
import java.io.*;
import java.util.*;

class Executor extends DepthFirstAdapter {
    Scanner sc = new Scanner(System.in);
    PrintStream pr = System.out;
    HashMap<String,Integer> vars = new HashMap<String,Integer>();
    @Override
    public void outAAssignStat(AAssignStat node) {
        int v = (Integer)getOut(node.getExpr());
        String s = node.getIdent().getText();
        pr.printf("%s = %d\n", s, v);
        vars.put(s, v);
    }
    @Override
    public void outAGtExpr(AGtExpr node) {
        int x = (Integer)getOut(node.getLeft());
        int y = (Integer)getOut(node.getRight());
        setOut(node, new Integer((x > y) ? 1 : 0));
    }
    @Override
    public void outALtExpr(ALtExpr node) {
        int x = (Integer)getOut(node.getLeft());
```

```

        int y = (Integer)getOut(node.getRight());
        setOut(node, new Integer((x < y) ? 1 : 0));
    }

    @Override
    public void outAOneExpr(AOneExpr node) {
        setOut(node, getOut(node.getNextp()));
    }

    @Override
    public void outAAddNexp(AAddNexp node) {
        int x = (Integer)getOut(node.getNextp());
        int y = (Integer)getOut(node.getTerm());
        setOut(node, new Integer(x+y));
    }

    @Override
    public void outASubNexp(ASubNexp node) {
        int x = (Integer)getOut(node.getNextp());
        int y = (Integer)getOut(node.getTerm());
        setOut(node, new Integer(x-y));
    }

    @Override
    public void outAOneNexp(AOneNexp node) {
        setOut(node, getOut(node.getTerm()));
    }

    @Override
    public void outAMulTerm(AMulTerm node) {
        int x = (Integer)getOut(node.getTerm());
        int y = (Integer)getOut(node.getFact());
        setOut(node, new Integer(x*y));
    }

```

```

}
@Override
public void outADivTerm(ADivTerm node) {
    int x = (Integer)getOut(node.getTerm());
    int y = (Integer)getOut(node.getFact());
    setOut(node, new Integer(x/y));
}
@Override
public void outAOneTerm(AOneTerm node) {
    setOut(node, getOut(node.getFact()));
}
@Override
public void outAIdentFact(AIdentFact node) {
    Object o = vars.get(node.getIdent().getText());
    if(o == null) { o = new Integer(0); }
    setOut(node, o);
}
@Override
public void outAIconstFact(AIconstFact node) {
    setOut(node, new Integer(node.getIconst().getText()));
}
}

```

SableCCの記述ファイル (4)

main() はすべて同じ形。なっています。ここでは解析するだけなので、構文チェッカができます。

```
package sam94;
import sam94.parser.*;
import sam94.lexer.*;
import sam94.node.*;
import java.io.*;
import java.util.*;

public class Sam94 {
    public static void main(String[] args) throws Exception {
        Parser p = new Parser(new Lexer(new PushbackReader(
            new InputStreamReader(new FileInputStream(args[0])), 1024));
        Start tree = p.parse();
        tree.apply(new Executor());
    }
}
```

SableCC の記述ファイル (5)

実行のしかたを確認。コピーするのは「SableCC ファイル」
「Sam94.java」「Executor.java」の3つ。Java ソースはすべて「sam94/」の下におくこと。

```
% sablecc sam94.grammer
...
% javac sam94/Sam94.java
...
% cat test.min
x = 10;
y = x * 3;
% java sam94.Sam94 test.min
x = 10
y = 30
```

演習 9-6 上の例をそのまま動かしてみよ。動いたら、次のような拡張をおこなえ。

- a. 剰余演算が使えるようにする。
- b. 今は「式をカッコで囲む」ことができないので、できるようにする。
- c. C 言語の 3 項演算子をふやす。

演習 9-7 print 文と read 文をつくる。(答えは前回資料にあるが、できれば見ないで作れるとよい。) あらましは次の通り。

- 文法で stat の選択肢に「read ident semi」「print expr semi」をふやす。(この状態でコンパイルして文法が OK なことを確認。)

- **Executor**でこの2つの文の対応メソッドを作る。動作はこんな感じ。

```
pr.printf("> "); int x = sc.nextInt(); sc.nextLine();  
(変数の値として vars に書き込む)
```

(式の値を取得する)

```
pr.printf("%d\n", 値);
```

演習 9-8 ループ構文や枝分かれ構文をつくる。(いちばんふつうの while 文や if 文は前回資料にあるので、それと違うものが望ましい。例えば次のようなものはどうか。)

- なぜか2回実行する「twice 文」例: twice print x ;
- 条件を後から指定する「when 文」例: x = x + 1 when(x < 10);
- if-fi 文「if 条件 then 文… fi」
- 上記の if-fi 文に elsif(任意回数) や else を追加
- 変な構文 [条件 => 動作 | 条件 => 動作 | …]
- 無限ループ構文と break 文