南 京 理 工 大 学

编译原理课程设计

姓　　名: 蒋旭钊　　**学　号:** 918106840727

学院(系): 计算机科学与工程学院

专　　业: 计算机科学与技术

课　　程: 软件课程设计（Ⅱ）

2021年　5月

实验 1：词法分析

一、需求分析

要求：创建一个词法分析程序，该程序支持分析常规语法。

1. 设计实现类高级语言的词法分析器，基本功能为识别以下几类单词：

   (1) 关键字

   ①类型关键字：整型、浮点型、布尔型、记录型；

   ②分支结构中的 if 和 else；

   ③循环结构中的 do 和 while。

   (2) 标识符

   由大小写字母、数字以及下划线组成，但必须以字母或者下划线开头。

   (3) 常量

   无符号整数和浮点数等。

   (4) 限定符

   ①用于赋值语句的界符，如“=”；

   ②用于句子结尾的界符，如“；”。

   (5) 运算符

   ①算术运算符；

   ②关系运算符；

   ③逻辑运算。

2. 具体要求

   (1) 要求基于 DFA 技术设计词法分析器。

   (2) 词法分析程序可以准确识别:科学计数法形式的常量（如 0.314E+1），复数常量（如 10+12i），可检查整数常量的合法性，标识符的合法性（首字符不能为数字等），尽量符合真实常用高级语言要求的规则。

二、文法设计

文法产生式详见 lexicalgram.txt。开始符号为 S，相关内容如下：

(1) 整数

```
S->0A
S->1A
S->2A
S->3A
S->4A
S->5A
S->6A
S->7A
S->8A
S->9A
S->+S
S->-S
A->0A
A->1A
A->2A
A->3A
A->4A
A->5A
A->6A
A->7A
A->8A
A->9A
A->ε
```

(2) 小数

```
25 A->.B
26 B->0B
27 B->1B
28 B->2B
29 B->3B
30 B->4B
31 B->5B
32 B->6B
33 B->7B
34 B->8B
35 B->9B
36 B->0
37 B->1
38 B->2
39 B->3
40 B->4
41 B->5
42 B->6
43 B->7
44 B->8
45 B->9
46 B->ε
```

(3) 运算符

```
48 S->+G
49 S->-G
50 S->*G
51 S->/G
52 S->!G
53 S->&G
54 S->%G
55 S->~G
56 S->|G
57 S->^G
58 S->=G
59 G->ε
```

(4) 科学计数法

```
61 B->0C
62 B->1C
63 B->2C
64 B->3C
65 B->4C
66 B->5C
67 B->6C
68 B->7C
69 B->8C
70 B->9C
71 A->eD
72 C->eD
73 D->+E
74 D->-E
75 D->εE
76 E->0E
77 E->1E
78 E->2E
79 E->3E
80 E->4E
81 E->5E
82 E->6E
83 E->7E
84 E->8E
85 E->9E
86 E->0
87 E->1
88 E->2
89 E->3
90 E->4
91 E->5
92 E->6
93 E->7
94 E->8
95 E->9
```

(5) 复数

```
 97 A->+L
 98 A->-L
 99 L->1P
100 L->2P
101 L->3P
102 L->4P
103 L->5P
104 L->6P
105 L->7P
106 L->8P
107 L->9P
108 P->0P
109 P->1P
110 P->2P
111 P->3P
112 P->4P
113 P->5P
114 P->6P
115 P->7P
116 P->8P
117 P->9P
118 P->i
```

(6) 界符

```
120 S->(Z
121 S->)Z
122 S->{Z
123 S->}Z
124 S->;Z
125 S-><Z
126 S->>Z
127 S->:Z
128 S->#Z
129 Z->ε
```

（7）标识符

```
131 S->aH
132 S->bH
133 S->cH
134 S->dH
135 S->eH
136 S->fH
137 S->gH
138 S->hH
139 S->iH
140 S->jH
141 S->kH
142 S->lH
143 S->mH
144 S->nH
145 S->oH
146 S->pH
147 S->qH
148 S->rH
149 S->sH
150 S->tH
151 S->uH
152 S->vH
153 S->wH
154 S->xH
155 S->yH
156 S->zH
157 H->aH
158 H->bH
159 H->cH
160 H->dH
161 H->eH
162 H->fH
163 H->gH
164 H->hH
165 H->iH
166 H->jH
167 H->kH
168 H->lH
169 H->mH
170 H->nH
171 H->oH
172 H->pH
173 H->qH
174 H->rH
175 H->sH
176 H->tH
177 H->uH
178 H->vH
179 H->wH
180 H->xH
181 H->yH
182 H->zH
183 H->_H
184 H->0H
185 H->1H
186 H->2H
187 H->3H
188 H->4H
189 H->5H
190 H->6H
191 H->7H
192 H->8H
193 H->9H
194 H->ε
```

（8）关键字内置

```
//关键词
keywords.add("auto");  keywords.add("double"); keywords.add("int");
keywords.add("break"); keywords.add("else"); keywords.add("switch");
keywords.add("case"); keywords.add("return"); keywords.add("float");
keywords.add("continue"); keywords.add("for"); keywords.add("void");
keywords.add("if"); keywords.add("while"); keywords.add("String");
keywords.add("complex"); keywords.add("default"); keywords.add("main");
```

三、系统设计

3.1 设计方案

1) 程序使用右线性的正规文法作为输入，调用 LexicalGram 类中的 read 方法从文本文件中逐行读取产生式并将产生式用 LexicalProc 类保存，最终所有类都存储在 ArrayList<LexicalProc> *LexProcF*。
2) 按照正规文法到 NFA 的转换规则构造 NFA 的边，用 Edge 类存储 NFA 边，所有边存储在 NFAutil 类定义的 ArrayList<Edge> *NFAEdge* 容器中。
3) 再调用 NFAtoDFA 类中定义的闭包函数 Closure 和 Move 函数，将 NFA 确定化为 DFA，所有 DFA 状态用 DfaState 类存储，所有状态存储在 NFAtoDFA 类中的 ArrayList<DfaState> DfaStates 容器中,然后用 DfaEdge 类存储 DFA 边，所有边存储在 NFAtoDFA 类定义的 ArrayList<DfaEdge> *DFAEdges* 容器中。
4) 最后只需要调用 Lexical 类中的 lex()函数，根据建立好的 DFA 状态对输入的 test.txt 进行词法分析，将类别区分开来，同时标记出符号和常量，将最终结果用 GUI 展示出来。

3.2 核心数据结构

(1) NFAutil 中：

```
//关键字
public static TreeSet<String> keywords = new TreeSet<String>();
//界符
public static TreeSet<String> limiters = new TreeSet<~>();
//运算符
public static TreeSet<String> calculates = new TreeSet<~>();
// 非终结符集
public static TreeSet<String> VN = new TreeSet<~>();
// 终结符集
public static TreeSet<String> VT = new TreeSet<~>();
//  终态
public static TreeSet<String> FinalChars = new TreeSet<~>();
//NFA中的各条边
public static ArrayList<Edge> NFAEdge= new ArrayList<>();
```

(2) NFAtoDFA 中：

```
//初始化DFA状态集数目
public static int Statenum = 0;
//每个DFA状态集
public static ArrayList<DfaState> DfaStates = new ArrayList<>();
//DFA中的各条边转换关系
public static ArrayList<DfaEdge> DFAEdges = new ArrayList<>();
```

(3) LexicalProc:

```
public class LexicalProc {
    public String left;  // 产生式左部
    public String list; //产生式右部
```

(4) LexicalGram:

```
// 三型文法产生式集
public static ArrayList<LexicalProc> LexProcF = new ArrayList<~>();
```

(5) Lexical:

```
// 记录符号表位置
public static int symbol_pos = 0;
// 符号表及其位置的HashMap
public static Map<String, Integer> symbol = new HashMap<~>();

// 记录常量位置
public static int constant_pos = 0;
// 常量表及其位置的HashMap
public static Map<String, Integer> constant = new HashMap<~>();  // 常量表HashMap

// 保存token列表
public static StringBuffer result = new StringBuffer();
```

(6) Edge:

```
public class Edge { //NFAedges
    public char u, v;
    public char key;
```

(7) DfaState:

```
public class DfaState { //DfaState
    public int index;
    public TreeSet<String> charset;
    public int isfinal;
```

(8) DfaEdge:

```
public class DfaEdge { //Dfa转换关系
    public int u, v;
    public char key;
```

3.3 主要函数及其功能

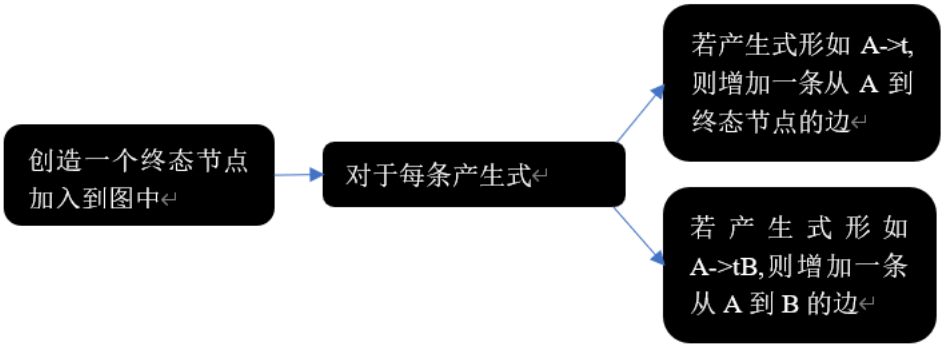
(1) Void Read()

```java
/**
 * 从文件中读取文法并且存储到CFG类的静态容器中，编号就是容器自带的index
 * @param filename
 * @throws FileNotFoundException
 */
private static void read(String filename) throws FileNotFoundException
{
    File file = new File(filename);
    Scanner scanner = new Scanner(file);
    while(scanner.hasNext())
    {
        String line = scanner.nextLine().trim();
        if(!line.equals(""))
        {
            String[] div = line.split( regex: "->");
            String right = div[1];
            LexicalProc derivation = new LexicalProc( s: div[0].trim()+"->"+ri
            LexProcF.add(derivation);//存储到静态的容器中
        }
    }
    scanner.close();
}
```

(2) Void EdgeForm()

```java
//形成NFA中的各条边
private static void EdgeForm()
{
    Iterator<LexicalProc> iterProc = LexicalGram.LexProcF.iterator();
    while(iterProc.hasNext())
    {
            LexicalProc Proc=iterProc.next();
            char[] procleft =Proc.left.toCharArray();
            char[] proclist = Proc.list.toCharArray();
            if(proclist.length>1)//产生两个状态间的边
            {
                NFAEdge.add(new Edge(procleft[0],proclist[1],proclist[0]));
            }
            else//产生到终结状态的边
            {
                NFAEdge.add(new Edge(procleft[0], v: 'X',proclist[0]));
            }
    }
}
```

原理如下：



(3) TreeSet<String> Closure(TreeSet<String> set)

```java
/**
 * 计算符号集合的闭包
 *
 * @param set 状态集
 * @return
 */
private static TreeSet<String> Closure(TreeSet<String> set) {
    TreeSet<String> closure = new TreeSet<>();
    closure.addAll(set);
    Iterator<String> setiter = set.iterator();
    while (setiter.hasNext()) {
        String setelement = setiter.next();
        for (Edge edge : NFAutil.NFAEdge) {
            if (String.valueOf(edge.u).equals(setelement) && String.valueOf(ed
                closure.add(String.valueOf(edge.v));
            }
        }
    }
    return closure;
}
```

首先将初始符号集合加入到闭包里，然后遍历该初始符号集，同时对于其中的每个符号 setelement，遍历所有的 NFAEdge,若 NFAEdge 中有出发状态 u 和 setelement 相同并且该边的 key 为 emp 的，就将结束状态 v 加入到 closure 中，最终返回 closure.

(4) TreeSet<String> Move(TreeSet<String> set, String a)

```java
/**
 * 计算某DFA状态move VT后的符号集合
 *
 * @param set 状态集
 * @param a   VT
 * @return
 */
private static TreeSet<String> Move(TreeSet<String> set, String a) {
    TreeSet<String> Movea = new TreeSet<>();
    for (Edge edge : NFAutil.NFAEdge) {
        for (String s : set) {
            if (String.valueOf(edge.u).equals(s) && String.valueOf(edge.key)
                Movea.add(String.valueOf(edge.v));
            }
        }
    }
    return Movea;
}
```

首先遍历所有 NFAEdge，然后对于每条边，遍历 set 中的每个元素，如果存在 NFAEdge 中有出发状态 u 和 s 相同，该边的 key 和 a 相同，就将结束状态加入 Movea 中，最终返回。

(5) void formDfa()

```java
/**
 * 形成所有DFA状态集，并确定DFA各边转换关系
 */
public static void formDfa() {
    TreeSet<String> chars = new TreeSet<>();
    chars.add("S");
    TreeSet<String> closchars = Closure(chars);
    //初始化开始DFA状态
    DfaState startdfa = new DfaState(Statenum++, closchars, isfinal: 0);
    DfaStates.add(startdfa);

    //新状态队列
    Queue<DfaState> newqueue = new LinkedList<>();
    newqueue.offer(startdfa);
    // System.out.println(NFAutil.VT);
    //新状态逐个走出队列
    while (!newqueue.isEmpty()) {
        DfaState leftdfa = newqueue.poll();
        //System.out.println("出队"+d.index+d.charset);
        for (String vt : NFAutil.VT) {
            TreeSet<String> movevtchars = Move(leftdfa.charset, vt);
            if (movevtchars.isEmpty()) continue; //move后无产生式
            TreeSet<String> closmovevt = Closure(movevtchars);

            //如果在已有的DFaState状态序列中，查到index
            if (DfaState.Dfacontains(closmovevt)) {
                int toindex = DfaState.indexDfacontains(closmovevt);
                char[] tokeychar = vt.toCharArray();
                DfaEdge newedge = new DfaEdge(leftdfa.index, toindex, tokeychar[0]);
                DFAEdges.add(newedge);
            }

            //如果不在已有的DfaState状态序列中，添加newDfaState
            else {
                //  System.out.println("能够产生新集的VT："+vt);
                //判断是否为终结状态
                int finalcharflag = 0;
                for (String finalchar : NFAutil.FinalChars) {
                    if (closmovevt.contains(finalchar)) {
                        finalcharflag = 1;
                        break;
                    }
                }
                int leftindex = leftdfa.index;
                int toindex = Statenum;
                char[] tokeychar = vt.toCharArray();
                DfaEdge newedge = new DfaEdge(leftindex, toindex, tokeychar[0]);
                DFAEdges.add(newedge);

                DfaState addque = new DfaState(Statenum++, closmovevt, finalcharflag);
                //  System.out.println("入队"+addque.index+addque.charset);
                newqueue.offer(addque);
                DfaStates.add(addque);
            }
        }
    }
```

创建初始的 DFA 状态，在初始状态中输入符号 S，建立闭包，并加入 DfaStates 中，同时创建新状态队列 newqueue，对所有新状态进行闭包运算。取出队首的状态，遍历所有 VT，对于该状态的符号集进行 Move,Closoure 运算，如果在已有的所有 DfaStates 状态中，已经有此符号集对应的状态，则只记录 DfaEdge。如果在已有的所有 DfaStates 状态中，不存在此符号集对应的状

态，则生成新状态，产生新的 DfaEdge 边，同时将新状态加入到队列 newqueue 以便下次遍历。

(6) formDfa（）中用到的几个函数

```java
/**
 * 判断已有的DFA状态集中，是否包含和状态a相同符号集的状态，有则返回相同状态的序号
 *
 * @param a DFA状态a
 * @return 相同状态的序号
 */
public static int indexDfacontains(TreeSet<String> a) {
    for (int i = 0; i < NFAtoDFA.DfaStates.size(); i++) {
        DfaState d = NFAtoDFA.DfaStates.get(i);
        if (ContainEqual(d.charset, a)) {
            return d.index;
        }
    }
    return -1;
}
```

```java
/**
 * 判断两DFA状态的符号集是否相同
 *
 * @param a 状态集a
 * @param b 状态集b
 * @return
 */
public static boolean ContainEqual(TreeSet<String> a, TreeSet<String> b) {
    boolean flag1 = true;
    boolean flag2 = true;
    boolean flag;
    for (String i : a) {
        if (!b.contains(i)) flag1 = false;
    }
    for (String j : b) {
        if (!a.contains(j)) flag2 = false;
    }
    flag = flag1 & flag2;
    return flag;
}
```

(7) 根据 DfaState 结束状态判断识别类型

定义相应的终态符号：

```
//终态字符
FinalChars.add("A"); //整数
FinalChars.add("B"); //小数
FinalChars.add("G"); //运算符
FinalChars.add("H"); //标识符
FinalChars.add("P"); //复数
FinalChars.add("X"); //终结状态
FinalChars.add("E"); //科学计数法
FinalChars.add("Z"); //界符
```

写出根据产生式序号找到产生式的函数：

```
/**
 * 返回序号对应的dfastate
 * @param dfastateindex
 * @return
 */
public static DfaState indextoState(int dfastateindex)
{
    DfaState dfastate = null;
    for(DfaState d:NFAtoDFA.DfaStates)
    {
        if (d.index==dfastateindex) dfastate=(DfaState)d.clo
    }
    return dfastate;
}
```

类别判断函数：

```
/**
 * 根据终结的Dfa状态判定是否为标识符
 * @param Dfaindex
 * @return
 */
public static boolean isName(int Dfaindex)
{
    DfaState s= DfaState.indextoState(Dfaindex);
    if (s.charset.contains("H")) return true;
    else return false;
}
```

```java
/**
 * 根据终结的Dfa状态判定是否为复数
 * @param Dfaindex
 * @return
 */
public static boolean isFushu(int Dfaindex)
{
    DfaState s= DfaState.indextoState(Dfaindex);
    if (s.charset.contains("P")) return true;
    else return false;
}
```

```java
/**
 * 根据终结的Dfa状态判定是否为整数
 * @param Dfaindex
 * @return
 */
public static boolean isZhenShu(int Dfaindex)
{
    DfaState s= DfaState.indextoState(Dfaindex);
    if (s.charset.contains("A")) return true;
    else return false;
}
```

```java
/**
 * 根据终结的Dfa状态判定是否为小数
 * @param Dfaindex
 * @return
 */
public static boolean isXiaoShu(int Dfaindex)
{
    DfaState s= DfaState.indextoState(Dfaindex);
    if (s.charset.contains("B")) return true;
    else return false;
}
```

```java
/**
 * 根据终结的Dfa状态判定是否为科学计数法
 * @param Dfaindex
 * @return
 */
public static boolean isSci(int Dfaindex)
{
    DfaState s= DfaState.indextoState(Dfaindex);
    if (s.charset.contains("E")) return true;
    else return false;
}
```

(8) Lex()识别核心函数：

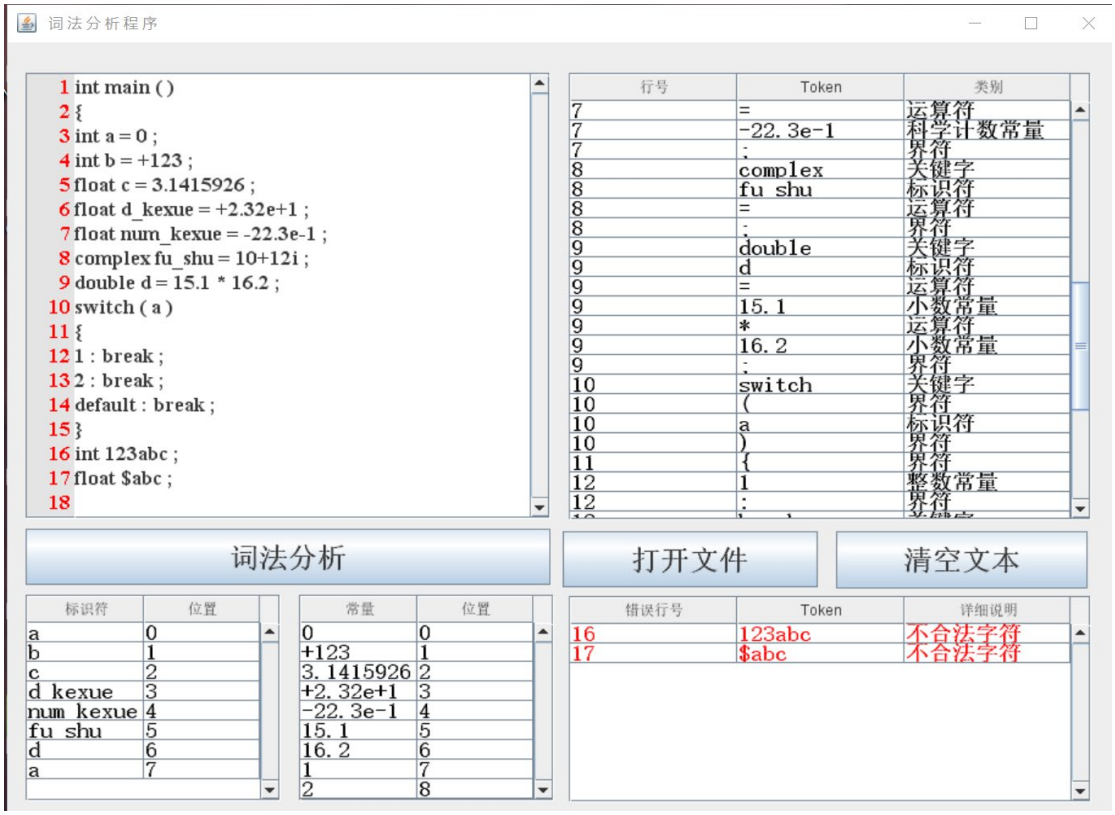
```
/**
 * 核心函数
 * 根据已经构成的DFA状态转换表
 * 按行分析数据，识别相应信息
 */
public void lex() {
    String[] texts = text.split( regex: "\n");
    //String[] texts={"int","a"};
    symbol.clear();
    symbol_pos = 0;
    constant.clear();
    constant_pos = 0;
    //System.out.println(texts.length);
    for (int m = 0; m < texts.length; m++) {
        int line = m + 1;
```

```
if (DfaState.isFinalState(edgeend) && j == splitcharstr.length) //如果是终结状态
{
    boolean havedone = false;
    if (NFAutil.isKeywords(token) && !havedone) {
        havedone = true;
        DefaultTableModel tableModel = (DefaultTableModel) jtable1.getModel();
        tableModel.addRow(new Object[]{line, token, "关键字"});
        jtable1.invalidate();
        result.append(" " + line + ",关键字," + token + "\n");
```

```
    }
} else if (havemistake == 0) {
    DefaultTableModel tableModel2 = (DefaultTableModel) jtable2.getModel();
    tableModel2.addRow(new Object[]{line, String.valueOf(splitcharstr), "不合法字
    jtable2.invalidate();
    //System.out.println(spacesplit[i]);
}
```

将 test.txt 中的每一行读入，将每一行的字符进行识别，产生相应的状态，若最终识别不出来，则进行错误提示。

四、结果分析



结果成功的识别了相应的文法符号，并且清楚地进行了展示。

```
识别的行为int main ( )
输入i变为i edgeend为: 5
输入n变为in edgeend为: 5
输入t变为int edgeend为: 5
输入m变为m edgeend为: 5
输入a变为ma edgeend为: 5
输入i变为mai edgeend为: 5
输入n变为main edgeend为: 5
输入(变为( edgeend为: 2
输入)变为) edgeend为: 2
识别的行为{
输入{变为{ edgeend为: 2
识别的行为int a = 0 ;
输入i变为i edgeend为: 5
输入n变为in edgeend为: 5
输入t变为int edgeend为: 5
输入a变为a edgeend为: 5
输入=变为= edgeend为: 1
```

同时在控制台中输出了相应的 DFA 状态转换关系以便查错。

实验 2：语法分析

一、需求分析

要求：创建一个使用 LR(1) 方法的语法分析程序。

1. 具体要求

能够根据用户输入的 2 型文法，生成 ACTION 表和 GOTO 表，能够演示其中间过程。

二、文法设计

拓广文法产生式详见 grammar.txt。开始符号为 S’，相关内容如下：

(1) 产生多行变量命名和变量赋值：

```
1 S'-> S
2 S -> D
3 S -> A
4 A -> A A
5 D -> D D
6 D -> D A
```

(2) 变量命名和类型表示

```
 7 D -> T id ;
 8 T -> int
 9 T -> double
```

(3) 变量赋值以及代数运算和嵌套

```
10 A -> id = E ;
11 E -> E + E1
12 E -> E1
13 E1 -> E1 * E2
14 E1 -> E2
15 E2 -> ( E )
16 E2 -> id
17 E2 -> num
```

(4) 循环语句以及嵌套

```
18 A -> A1
19 A -> A2
20 A1 -> while B do A0
21 A2 -> if B then A0
22 A0  -> { A3 }
23 A1 -> { A3 }
24 A2 -> { A3 }
25 A3 -> A3 ; A
26 A3 -> A
```

(5) Bool 判断和 bool 运算语句

```
27 B -> E JG E
28 B -> true
29 B -> false
30 JG -> <
31 JG -> <=
32 JG -> ==
33 JG -> !=
34 JG -> >
35 JG -> >=
```

三、系统设计

3.1 设计方案

语法识别主要分为两部分：

一是要根据输入的二型文法构造语法动作表。

二是要根据构造的语法动作表对输入的测试集进行相应的状态转换动作。

1) 首先读入二型文法产生式，用 Production 类存储，并将所有产生式存储在 ProcsandFirst 类的 ArrayList<Production> GramProcF 容器中。
2) 同时在 ProcsandFirst 类中添加使用的所有终结符和非终结符，同时调用 GenerateFirst （ ） 产 生 所 有 符 号 的 First 集 ， 存 在

HashMap<String,TreeSet<String> > firstMap 中。

3) 在 TableForm 类中构造语法动作表，首先用 createTableHeader（）函数建立 ACTION 表和 GOTO 表表头，然后调用 InitiateDfa()构造所有项目集族及其转换关系，将关系记录在 ArrayList<gramDfaEdge> gramDfaEdges 中。根据上面的基础，调用 createAnalyzeTable()调充语法分析表。

4) 在 SyntaxParser 类中进行语法分析，用 ReadToken()不断读入 Token 符号，构造相应的状态栈 Stack<Integer>stateStack,符号栈 Stack<String>tokenStack。在核心函数 analyze()中，根据动作表产生相应的动作。

3.2 核心数据结构

(1) ProductionState 类

```
public class ProductionState implements java.lang.Cloneable {
    public Production d;  // 产生式
    public String firstsearch;  //  向前搜索符
    public int dotindex;  // .位置,产生式符号左边

    /**
     * DFA状态集中每一个状态
     * 此时表示类似于“A->BC.D, a”的形式
     *
     * @param d      产生式
     * @param firstsearch     向前搜索符
     * @param dotindex .位置
     */
```

(2) DfaState 类

```
public class DfaState
{
    // 项目集编号,即DFA状态号
    public int id;
    // 该DFA状态中的产生式状态集列表,每个元素表示一个产生式状态
    public ArrayList<ProductionState> set = new ArrayList<~>();
```

(3) gramDfaEdge

```
public class gramDfaEdge {
    public int Dfafrom; //DfaEdge出发状态
    public int Dfato; //DfaEdge到达状态
    public String DfaDirection; //DfaEdge上的Key
```

(4) 非终结符，终结符和 First 集合

```
public static TreeSet<String> VN = new TreeSet<~>();  // 非终结符集
public static TreeSet<String> VT = new TreeSet<~>();  // 终结符集
public static ArrayList<Production> GramProcF = new ArrayList<~>();  // 产生式集
//  每个符号的first集
public static HashMap<String,TreeSet<String> > firstMap = new HashMap<~>();
```

(5) 语法分析表和状态栈、符号栈

```
private TableForm table;  //构造的语法分析表
private Stack<Integer> stateStack;  //用于存储相应的DFA状态号
private Stack<String> tokenStack;  //用于存储符号栈
```

3.3 主要函数及其功能

(1) GenerateFirst()

```
/**
 * 计算所有符号的first集合
 */
private static void GenerateFirst()
{
    //将所有的终结符的first都设为本身
    Iterator<String> iterVT = VT.iterator();
    while(iterVT.hasNext())
    {
        String vt = iterVT.next();
        firstMap.put(vt,new TreeSet<String>());
        firstMap.get(vt).add(vt);
    }
    //计算所有非终结符的first集合
    Iterator<String> iterVN = VN.iterator();
    while(iterVN.hasNext())
    {
        String vn = iterVN.next();
        firstMap.put(vn, new TreeSet<String>());//因为后续操作没有交叉涉及firstMap,
        firstMap.get(vn).addAll(IterFirstFind(vn));
    }
}
```

(2) TreeSet<String> IterFirstFind(String vn)

```java
/**
 * 用于查找VN的First集
 * @param vn
 * @return 返回first集
 */
private static TreeSet<String> IterFirstFind(String vn)
{
    TreeSet<String> firstset = new TreeSet<~>();
    int lastsize = 0;
    int rearsize = 0;
    while(true)
    {
        //直到没有新的VT或emp加入到该vn的First集合中停止
        lastsize = firstset.size();
        for(Production d:GramProcF)
        {
            if(d.left.equals(vn))//first集看产生式左部
            {
                if(VT.contains(d.right.get(0)))  // 终结符，直接加入
                {
                    firstset.add(d.right.get(0));
                }
                else if(d.right.get(0).equals(emp))  // 空符号，直接加入
                {
                    firstset.add(emp);
                }
                else if(VN.contains(d.right.get(0)))  // 非终结符，递归
                {
                    if(!vn.equals(d.right.get(0)))  // 去除类似于E->E*E这样的左递归
                    {
                        TreeSet<String> set2 = IterFirstFind(d.right.get(0));
                        firstset.addAll(set2);
                        // 如果产生式右侧的VN能推出emp，接下来继续findfirst
                        if(set2.contains(emp))
                        {
                            for(int j=1; j<d.right.size(); j++)
                            {
                                TreeSet<String> set3 = IterFirstFind(d.right.get(j)
                                firstset.addAll(set3);
                                // 再次递归直到不包含emp
                                if(!set3.contains(emp))
                                {
                                    break;
                                }
                            }
                        }
                    }
                }
            }
        }
        rearsize = firstset.size();
        if(lastsize == rearsize)
            break;
    }
    return firstset;
}
```

计算非终结符 VN 的 First 集合，While(True)循环，终止条件是 First 集合中不会有新的元素加入。

- 对于某一个 VN，遍历所有产生式，找到产生式左边和 VN 相等的，查看产生式右部的第一个符号。
- 如果是 VT 和 emp 直接加入。
- 如果是 VN，则需要进一步递归，但是考虑到左递归的死循环但是对求 First 无影响，所以在不是左递归的情况下，求出右侧 VN 的 First 集合加入结果中，如果包含 emp，则需要依产生式右部一直往下求，直到某一个符号的 FirstSet 不包含 emp 退出。

(3) InitiateDfa()

```
/**
 * 利用这个递归方法建立一个用于语法分析的DFA，初始化开头部分
 */
private void InitiateDfa()
{   //初始化建立LR(1)项目集族dfa
    this.dfa = new AllDfaStates();
```

- 从 state0 开始建立，将 S'->.S,#加入 state0 状态中，接着遍历该 state0 的所有产生式。
- 对于所有的非规约状态，即 dotindex 比 right.size 小，我们才可能会有新的产生式状态加入。
- 求取该产生式的向前搜索符 searchfirst，类似于“A->B.C, #”的状态,向前搜索符不做改动，直接用原来的，否则类似求 first，分情况如下：
  a) 首先设置一个标志位 flag=true，对于.后面的每一个符号遍历调用 firstofV 方法得到 ProcsandFirst 里面已经求好的 first 集，将这些 first 集加入到 searchfirst 集合中，在此过程中，如果该 first 集合不包含 emp 则置 flag=false 跳出循环，最终 first 集合包含前面所求。
  b) 如果遍历完了之后每个都包含 emp，则最后 first 集合中还要加入原来的向前搜索符集合 firstsearch。
- 接着就是衍生产生式状态的加入，只有在.后面是 VN 时，才需要加入新的产生式状态,用 leftvgetDerivation(dotA)找到.之后这个 VN 为左部的所有产

生式，遍历所有产生式，遍历所有之前求得的 searchfirst,产生新产生式状态 procstate1,若原产生式推出 emp，则直接把.加在最后，若原产生式不能推出 emp，则将.加到最前面。

- 用 lrdStatecontains 判断新产生是状态 procstate1 是否已经存在 state0 中，若有则不添加，若无则添加。
- 最后将 state0 加入到 dfa 状态表中,同时获取该状态中,所有.后面的符号，以此为 direction，同时对于每一种符号，用 getLRDmatchs 得到这一类产生式状态，最后调用 DerivateState 衍生其他状态。

(4) DerivateState(intFromState,String,direction,ArrayList<ProductionState> DirectionSame)

```
/**
 * 通过输入一个从上一个状态传下来的LR产生式的list获取下一个状态，
 * 如果该状态已经存在，则不作任何操作，跳出递归，如果该状态不存在，则加入该状态，继续进行递
 * @param FromState 上一个状态的编号
 * @param direction .后面的符号
 * @param DirectionSame 由某一个direction引起的所有产生式的集合
 */
private void DerivateState(int FromState,String direction,ArrayList<ProductionState
{
```

- 首先将所有产生式状态的.后移，然后将所有产生式加入到新状态 newderiState 状态中。
- 对与 newderiState 状态进行产生式状态的衍生和加入，类似于上面 state0,如果最后产生的 newderiState 和以前已经有的 dfa 状态一样，则存储 DfaEdge 转换关系后，该函数返回。
- 如果产生的状态不一样，则添加新 dfastate 的序号和内容,增加和新状态有关的新边转换关系。
- 最后就是对于新加入的状态，继续调用调用 DerivateState 衍生其他 DFA 状态。

(5) void createAnalyzeTable()

```
/**
 * 利用这个方法填充语法分析表的相关内容
 */
public void createAnalyzeTable()
{
```

```
 //完善语法分析表的goto部分
 int gotoCount = gramDfaEdges.size();
 for(int i = 0;i < gotoCount;i++)
 {
     int start = gramDfaEdges.get(i).Dfafrom;
     int end = gramDfaEdges.get(i).Dfato;
     String direction = gramDfaEdges.get(i).DfaDirection;
     int pathIndex = gotoIndex(direction);
     this.gotoTable[start][pathIndex] = end;
 }
```

```
//完善语法分析表的action部分
int stateCount = dfa.states.size();
for(int i = 0;i < stateCount;i++)
{
    DfaState state = dfa.get(i);//获取每个dfa的状态
    for(ProductionState procstate:state.set)
    {//DFA状态的产生式逐个进行分析
```

- 在 GOTO 表的构造中，遍历所有的 DFA 状态转换边 gramDfaEdges，将其中的出发状态，到达状态以及输入符号 key 存入 GOTO 表中。
- 在 ACTION 表的构造中，遍历所有 DFA 状态 DfaStates，在此基础上，遍历每个 DFA 状态的每条边。
  a) 如果是规约产生式，即 procstate.dotindex == procstate.d.right.size()，并且产生式左端不是 S'，则找到该产生式的编号，同时将 ACTION 表中所有以该 DFA 状态 i 为行，该产生式向前搜索符集 actionIndex(procstate.firstsearch)为列的 ACTION 表置为“规约状态 r”；如果产生式左端是 S'，设为“接受状态 acc”。
  b) 如果不是规约产生式，则获取.后面的符号 next，如果是非终结符的话，则在 ACTION 表中添加以 i 为行，actionIndex(next)为列的“移进状态 s”。

(6) GramAnalyze()语法分析

```
 * 主体部分 语法分析
 */
public void GramAnalyze()
{
    while(true)
    {
        result.append("符号串: ");
        System.out.print("符号串: ");
        String inputBuffer=printInputandOut();
        result.append("\n");
        System.out.println();

        Token token = readToken();
        String value = getValue(token);
```

```
//碰到Si的处理，移入新的状态
if(action.startsWith("s"))
{
    int newState = Integer.parseInt(action.substring(1));
    String newInput =inputBuffer.trim().split( regex: " ")[0];
    stateStack.push(newState);
    tokenStack.push(newInput);
```

```
//碰到Ri的处理，根据产生式长度弹栈，并根据弹栈后的Goto表格决定入栈的状态
else if(action.startsWith("r"))
{
    Production derivation = ProcsandFirst.GramProcF.get(Integer.parseInt(
    result2.add(derivation.toString());
    int r = derivation.right.size();
    dotindex--;//输入串不变，下次还是
    //根据产生式长度，符号栈和状态栈依次弹栈
    if(!derivation.right.get(0).equals("ε"))
    {
        for(int i = 0;i < r;i++)
        {
            stateStack.pop();
            tokenStack.pop();
        }
    }
```

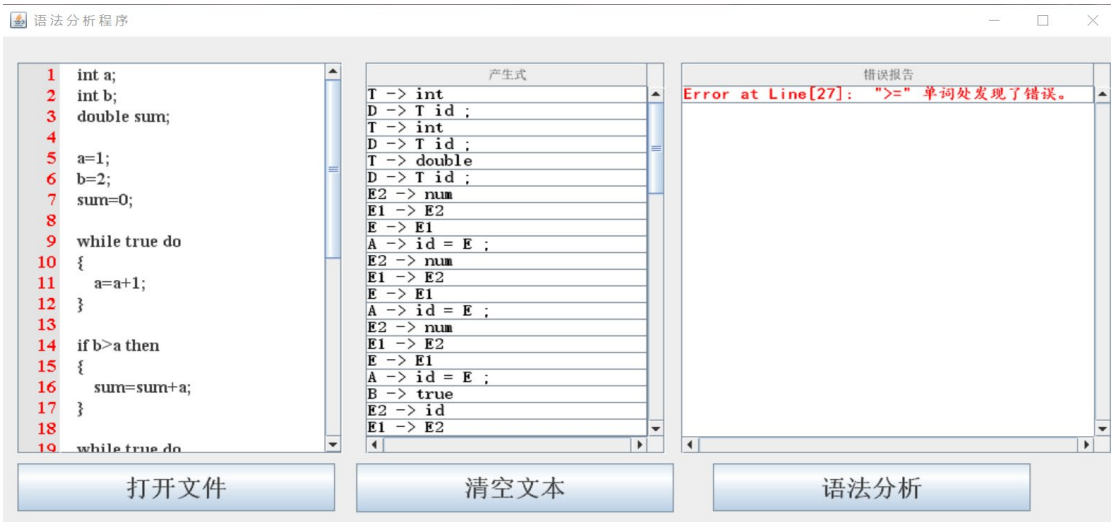
```
else if(action.equals(TableForm.acc))
{
    System.out.println("acc: ");
    result.append("acc: "+"\n");
```

- 在 while(true)循环中进行语法分析，终止条件是接受状态或者 error()。
- 首先将状态 0 压入 statestack,将符号#压入 tokenstack,读取第一个符号，通过 getActionElement(state, value)找到 ACTION 中的内容。

- 如果碰到 Si，则将 i 新状态压入状态栈中，将移入符号串中的第一个 token 压入符号栈。
- 如果碰到 Ri,则获取到第 i 个产生式，将该产生式右部的长度获得，将 tokenstack 中的 token 弹出同时将 statestack 中的 state 弹出，最后根据 getGotoElement(stateStack.lastElement(), derivation.left)得到规约后状态，压入状态栈，此时 analyzeindex—保证下次移入符号串分析不动。
- 如果碰到 acc，则规约完成 break;
- 否则碰到 error，则通过 error()给出错误报告。

四、结果分析

输入 testerror.txt，带错误报告的分析如下：



输入 testperfect.txt，语法分析如下：

实验 3：语义分析

一、需求分析

要求：创建符合属性文法的语义分析程序，可以进行简单的表达式计算，最终输出四元式序列。

二、文法设计

二型文法详见 grammar.txt，相应的语义动作见语义动作.txt：

```
P'-> P
P -> D S
S -> S M S
D -> D D
D -> T id ;

T -> int
T -> double


S -> id = E ;
E -> E + E1
E -> E1
E1 -> E1 * E2
E1 -> E2
E2 -> id
E2 -> num

M -> ε
```

语义动作的含义：

(1) 全局动作

```
0:  P' -> P
1:  P -> D S  {print("语义分析完成")}
2:  S -> S M S  {backpatch(S1.nextlist,M.huiAddress); S.nextlist=S2.nextlist;}
```

(2) 声明语句

```
3:  D -> D D
4:  D -> T id ; {addChartoTable(top(tblptr), id.name, T.type, top(offset)); top(offset) :=
5:  T -> int  {T.type=int; T.width=4;}{type=T.type; width=T.width;}
6:  T -> double  {T.type=double; T.width=8;}{type=T.type; width=T.width;}
```

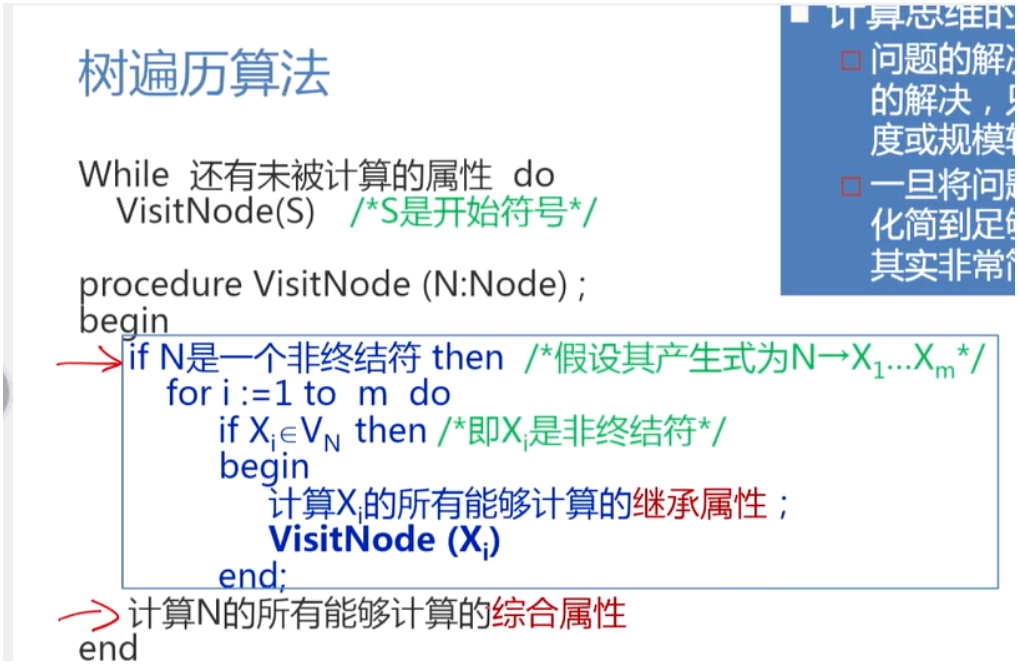
(3) 赋值及算数表达式语句

```
7:  S -> id = E ;  {p=isInTable(id.lexeme); if p==nil then error; gencode(p'='E.addr);} {
8:  E -> E + E1  {E.addr=newtemp(); gencode(E.addr'='E.addr'+'E1.addr);}
9:  E -> E1  {E.addr=E1.addr}
10:  E1 -> E1 * E2  {E1.addr=newtemp(); gencode(E1.addr'='E1.addr'*'E2.addr);}
11:  E1 -> E2  {E1.addr=E2.addr}
12:  E2 -> id  {E2.addr=isInTable(id.lexeme); if E2.addr==null then error;}
13:  E2 -> num  {E2.addr=isInTable(num.lexeme); if E2.addr==null then error}
```

三、系统设计

3.1 设计方案

基于属性文法的语义分析程序可以看做是 LR(1)语法分析程序的一种拓展，我们只需要在语法分析的时候，在语法树上建立相应的属性文法树节点即可。同时需要事先定义好语义动作，相关的语义动作在 Semantic 类中分别予以实现。接着只需要深度遍历语法树，计算出全部的文法属性，执行相应的语义动作，就能够实现语义分析。



3.2 核心数据结构

(1) Treenode 类

```
public class TreeNode
{
    private int id;  // 节点编号
    private String prodChar;  // 符号类型
    private String value;  // 符号值
    private int line;  // 所在行数
```

(2) Tree 类

```
public class Tree
{
    private TreeNode fatherNode;  // 父节点
    private ArrayList<TreeNode> children;  // 孩子列表
```

(3) CharsID 类，保存符号表中的每个符昊

```
public class CharsID
{
    private String name;  // 符号名
    private String type;  // 符号类型
    private int offset;  // 偏移量
```

(4) FourAddr 类

```
public class FourAddr
{
    private String op;  // 操作符
    private String param1;  // 参数一
    private String param2;  // 参数二
    private String toaddr;  // 地址
```

(5) Properties 类

```
public class Properties
{
    private String name;  // 变量或者函数的name
    private String type;  // 节点类型
    private String offset;  // 数组类型的属性
    private int width;  // 类型大小属性


    private String addr;  // 表达式类型的属性

    private int huiAddress;  // 回填用到的属性,指令位置
    private List<Integer> truelist = new ArrayList<~>();  // 列表
    private List<Integer> falselist = new ArrayList<~>();  // 列表
    private List<Integer> nextlist = new ArrayList<~>();  // 指令列表
```

用以保存树上每个节点的属性，根据语义动作的执行填入相应的属性。

(6) Semantic 类，语义分析核心函数

```
public class Smantic
{
    private static ArrayList<Tree> tree =  new ArrayList<~>();  // 语法树
    static List<Properties> treeNodeProperty;  // 语法树节点属性

    static Stack<CharsID> IDcharstable = new Stack<>(); // 符号表

    static List<String> three_addr = new ArrayList<~>();  // 三地址指令序列
    static List<FourAddr> four_addr = new ArrayList<~>();  // 四元式指令序列

    static String type;  // 类型
    static int width;  // 变量大小
    static int offset;  // 偏移量
    static int temp_cnt = 0;  // 新建变量计数
    static int nextInsAddress = 1;  // 指令位置

    static Stack<Integer> offTableNext = new Stack<~>();  //  符号表偏移大小栈

    static int nodeSize;  // 语法树上的节点数
    static int treeSize;  // 语法树大小
    static int initialPC = nextInsAddress;  // 记录第一条指令的位置
```

(7) 语法树建立时

```
public static ArrayList<Tree> tree = new ArrayList<~>();  // 所有语法树包括子树
private Stack<TreeNode> treeNodeID = new Stack<TreeNode>();;  // 用于存储相应的树节点
private int nodecount=0;  // 记录树中节点号
```

存储所有语法子树以及所有的结点的结点栈 Stack<TreeNode>treeNodeID。

3.3 主要函数及其功能

(1) 在语法分析时同时建立语法树

- 碰到 s 状态，直接将该 VT 作为树节点压入

```
//碰到Si的处理，移入新的状态
if(action.startsWith("s"))
{
    int newState = Integer.parseInt(action.substring(1));
    String newInput =inputBuffer.trim().split( regex: " ")[0];
    stateStack.push(newState);
    tokenStack.push(newInput);
    System.out.println("移入:"+ "+newInput);
    result.append("移入:"+ "+newInput+"\n");
    System.out.println("ACTION表:"+ "+action);
    result.append("ACTION表:"+ "+action+"\n");
    System.out.println("GOTO表:"+ "+"--");
    result.append("GOTO表:"+ "+"--"+"\n");
    System.out.println("状态栈:"+stateStack.toString());
    result.append("状态栈:"+stateStack.toString()+"\n");
    System.out.println("符号栈:"+tokenStack.toString());
    result.append("符号栈:"+tokenStack.toString()+"\n");

    //语义分析中直接将该VT符号作为树结点
    treeNodeID.push(new TreeNode(nodecount++,value,token.value,existline));
```

- 碰到 r 状态，则先获取该规约状态产生式右部的长度，将 Stack<TreeNode>treeNodeID 中的 VT 弹出作为树的子节点，将规约后的 VN 作为树的父亲结点，存储到树列表 ArrayList<Tree>tree 中。

```
//存储某棵树的所有子结点
LinkedList<TreeNode> sonnode = new LinkedList<~>();
//根据产生式长度，符号栈和状态栈依次弹栈
if(!derivation.list.get(0).equals("ε"))
{
    for(int i = 0;i < r;i++)
    {
        stateStack.pop();
        tokenStack.pop();
        sonnode.addFirst(treeNodeID.pop());//将子节点加入
    }
}
```

```
//语义分析中将规约状态VN也作为树结点，同时在树列表中添加树（父树结点和子树节点列表）
treeNodeID.push(new TreeNode(nodecount++,derivation.left, value: "--",existline));
ArrayList<TreeNode> sonList = new ArrayList<~>(sonnode);
tree.add(new Tree(treeNodeID.peek(),sonList));
```

(2) String treeToProduction(Tree tree)

```java
/**
 * 将树转换成相应的产生式，从而判断语义动作
 * @param tree
 * @return
 */
public static String treeToProduction(Tree tree)
{
    String result = tree.getFather().getProdChar()+" -> ";
    for(TreeNode c:tree.getChildren())
    {
        result += c.getProdChar();
        result += " ";
    }
    return result.trim();
}
```

将树结构转换成相应的 String，可以和 Semantic 里的语义执行动作匹配执行。

(3) Boolean endPoint(TreeNode t)

```java
/**
 * 判断该结点的值是否已经算出
 * @param t 树节点
 * @return
 */
public static Boolean endPoint(TreeNode t)
{
    if (t.getValue().equals("--"))
    {
        return false;
    }
    return true;
}
```

用以判断某个树节点是否有值了，以此作为深度搜索结束的标志。

(4) void dfs(Tree tree)

```java
/**
 * 深搜遍历语法树
 * @param tree 语法树根节点
 */
public static void dfs(Tree tree)
{
    int flag = 0;
    for(int i=0; i<tree.getChildren().size(); i++)
    {
        TreeNode tn = tree.getChildren().get(i);
        if (!util.endPoint(tn))  // 树结点值是否已算出
        {
            flag = 1;
            // 找到邻接表的下一节点
            Tree f = findTreeNode(tn.getId());
            dfs(f);  // 递归遍历孩子节点
            findSemantic(f);  // 查找相应的语义动作函数
        //  System.out.println( util.treeToProduction(f));
        }
    }
    if (flag == 0)
    {
        return;
    }

}
```

- 深度遍历语法树，如果该树的所有孩子结点的值都已经算出，则 return。
- 否则以孩子结点为树节点进行递归 dfs，同时用 findSemantic 查找相应的语义动作进行执行。

(5) 实现语义动作要用到的函数

- void addChartoTable(String name, String type, int offset)

```java
/**
 * 向符号表中增加元素
 * @param name 元素名字
 * @param type 元素类型
 * @param offset 偏移量
 */
private static void addChartoTable(String name, String type, int offset)
{
    CharsID s = new CharsID(name,type,offset);
    IDcharstable.push(s);
}
```

- int isInTable(String s)

```java
/**
 * 查找符号表，查看变量是否存在
 * @param s 名字
 * @return 该名字在符号表中的位置
 */
private static int isInTable(String s)
{
    int a;
        for (int j=0; j<IDcharstable.size(); j++)
        {
            if(IDcharstable.get(j).getName().equals(s))
            {
                a = j;
                return a;
            }
        }
    a = -1;
    return a;
}
```

✧ void backpatch(List<Integer> list, int huiAddress)

```java
/**
 * 回填地址
 * @param list 需要回填的指令序列
 * @param huiAddress 回填的地址
 */
private static void backpatch(List<Integer> list, int huiAddress)
{
    for(int i=0; i<list.size(); i++)
    {
        int x = list.get(i) - initialPC;
        three_addr.set(x, three_addr.get(x)+huiAddress);
        four_addr.get(x).setToaddr(String.valueOf(huiAddress));
    }
}
```

(6) 几类语义动作

✧ 语句回填

```
// S -> S1 M S2  {backpatch(S1.nextlist,M.huiAddress); S.nextlist=S2.nextlist;}
public static void semantic_2(Tree tree)
{
    int S = tree.getFather().getId();  // S
    int S1 = tree.getChildren().get(0).getId();  // S1
    int M = tree.getChildren().get(1).getId();  // M
    int S2 = tree.getChildren().get(2).getId();  // S2

    //将M的回填地址填入S1的nextlist中
    backpatch(treeNodeProperty.get(S1).getNextList(), treeNodeProperty.get(M).getQu

    //将S2的nextlist赋给S
    Properties newproperty = new Properties();
    newproperty.setNext(treeNodeProperty.get(S2).getNextList());
    treeNodeProperty.set(S,newproperty);
}
```

✧ 符号表添加

```
// D -> T id ; {addChartoTable(top(tblptr),id.name,T.type,top(offset));
//              top(offset) = top(offset)+T.width}
public static void semantic_4(Tree tree)
{
    int T = tree.getChildren().get(0).getId();  // T
    String id = tree.getChildren().get(1).getValue();  // id

    int i = isInTable(id);
    if (i == -1)
    {
        //在符号表中添加，同时更新符号表offset
        addChartoTable(id, treeNodeProperty.get(T).getType(), offTableNext.peek());
        int s = offTableNext.pop();
        offTableNext.push( item: s + treeNodeProperty.get(T).getWidth());
        offset = offset + treeNodeProperty.get(T).getWidth();
    }
}
```

✧ 声明类型

```
// T -> int  {T.type=int; T.width=4;}{t=T.type; width=T.width;}
public static void semantic_5(Tree tree)
{
    int T = tree.getFather().getId();  // T
    type = "int";
    width = 4;

    Properties newproperty = new Properties();
    newproperty.setType("int");
    newproperty.setWidth(4);
    treeNodeProperty.set(T,newproperty);
}
```

✧ 算术运算

```
//  E -> E1 * E2  {E.addr=newtemp(); gencode(E.addr'='E1.addr'*'E2.addr);}
public static void semantic_10(Tree tree)
{
    int E = tree.getFather().getId();  // E
    int E1 = tree.getChildren().get(0).getId();  // E1
    int E2 = tree.getChildren().get(2).getId();  // E2
    //计算新的newtemp的值
    String newtemp = newtemp();

    Properties newproperty = new Properties();
    newproperty.setValue(newtemp);
    treeNodeProperty.set(E,newproperty);

    String code = newtemp + " = " + treeNodeProperty.get(E1).getAddr() +
            "*" + treeNodeProperty.get(E2).getAddr();
    three_addr.add(code);
    four_addr.add(new FourAddr( op: "*",treeNodeProperty.get(E1).getAddr(),
            treeNodeProperty.get(E2).getAddr(),newtemp));
}
```

四、结果分析

将 test.txt 输入查看结果