2016

Lab2 实验报告

基于 LL(1)的语法分析器

杜天蛟 141250031

实验目的

自行定义文法，运用 LL(1)方法对输入语句进行语法分析并输出结果，加深对语法分析过程的理解。

内容描述

此程序使用 Java 语言，读取一个文本文件，用上次实验写的词法分析器得到一个 token 序列，再对该序列进行语法分析。该程序采用 LL(1)方法自顶向下分析，最后输出产生式序列。因为时间关系，本程序没有继续生成语法分析树。定义了一般赋值语句、条件语句、while 循环语句的文法。

思路方法

1. 自定义文法
2. 对文法进行预处理
3. 构造预测分析表
4. 基于分析表编写代码
5. 根据输入队列当前队首的字符和状态栈的栈顶进行分析，如果遇到终结符则匹配，遇到非终结符则继续分析，循环直至处理完输入队列。

假设

程序中的所有变量都为 id，所有 int， double 类型都为 num。

相关分析过程描述

预处理后的文法如下：

```
0.S->id=E;
1.S->if(C){S}else{S}
2.S->while(C){S}
3.E->TE’
4.E’->+TE’
5.E’->ε
6.T->FT’
7.T’->*FT’
```

8.T’->ε
9.F->(E)
10.F->num
11.F->id
12.C->DC’
13.C’->||DC’
14.C’->ε
15.D->(C)
16.D->id==num

预测分析表

	id	=	;	if	(	)	{	}	else	while	+	*	num	||	==	$R
S	0			1						2						
E	3				3								3			
E’			5			5					4					5
T	6				6								6			
T’			8			8					8	7				8
F	11				9								10			
C	12				12											
C’						14								13		14
D	16				15											

重要数据结构

```
/**
 * token 序列
 */
private List<String> tokens;
/**
 * 状态栈
 */
private Stack<String> stack;
```

```
/**
 * parse table
 */
public static final int[][] parseTable = {
        //   id    =  ;  if (  )  {  }  e  w  +  *  n  || == $
        {0, -1, -1, 1, -1, -1, -1, -1, -1, 2, -1, -1, -1, -1, -
1, -1},//S
        {3, -1, -1, -1, 3, -1, -1, -1, -1, -1, -1, -1, 3, -1, -
```

```
1, -1},//E
        {-1, -1, 5, -1, -1, 5, -1, -1, -1, -1, 4, -1, -1, -1, -
1, 5},//E'
        {6, -1, -1, -1, 6, -1, -1, -1, -1, -1, -1, -1, 6, -1, -
1, -1},//T
        {-1, -1, 8, -1, -1, 8, -1, -1, -1, -1, 8, 7, -1, -1, -
1, 8},//T'
        {11, -1, -1, -1, 9, -1, -1, -1, -1, -1, -1, -1, 10, -1,
-1, -1},//F
        {12, -1, -1, -1, 12, -1, -1, -1, -1, -1, -1, -1, -1, -
1, -1, -1},//C
        {-1, -1, -1, -1, -1, 14, -1, -1, -1, -1, -1, -1, -1,
13, -1, 14},//C'
        {16, -1, -1, -1, 15, -1, -1, -1, -1, -1, -1, -1, -1, -
1, -1, -1}//D
};
```

核心算法

首先将$和开始符 S 压栈，读取状态栈的栈顶和输入队列的队首进行分析，如果栈顶是非终结符，则查表找到对应的产生式，记录该产生式（方便输出），弹出当前栈顶，将新产生的元素压栈。如果是栈顶是终结符，则进行匹配，如果匹配成功则弹出栈顶，将输入的字符序列的指针后移。重复上述步骤，直到匹配到最后的终止符号为止。

在上述过程中，查表是一项比较繁杂的过程。在程序中我采用了硬编码的方法，将所有的产生式用一个 String 数组表示，用一个 int 型的二维数组表示 parse table，数组中存储的是各个产生式在数组中的下标，若没有对应产生式则标记为-1。查表的时候只要找到对应的行和列就可以得到完整的产生式。

另外，将产生式中的各个 token 压栈也是比较麻烦的事情，所以我将所有的产生式的 token 序列按顺序存储在一个二维数组里。压栈的时候只要从数组里拿数据循环压入即可，只是要注意左边的要最后压栈。

运行截图

File/test.txt 作为测试文件：

```
while(a==0){
   if(b==1||b==2){
      a=1*(2+3);
   }
   else{
      b=1+2;
```

```
   }
}
```

file/output.txt 为输出文件：

```
S->while(C){S}
C->DC'
D->id==num
C'->ε
S->if(C){S}else{S}
C->DC'
D->id==num
C'->||DC'
D->id==num
C'->ε
S->id=E;
E->TE'
T->FT'
F->num
T'->*FT'
F->(E)
E->TE'
T->FT'
F->num
T'->ε
E'->+TE'
T->FT'
F->num
T'->ε
E'->ε
T'->ε
E'->ε
S->id=E;
E->TE'
T->FT'
F->num
T'->ε
E'->+TE'
T->FT'
F->num
T'->ε
E'->ε
```

问题与解决

跑测试文件的时候老是中途卡壳，提示找不到对应的产生式。后来慢慢调试才发现当时为了方便压栈声明的 splitCfgs 中的一行出现了错误，把 T 记成了 T’。

```
public static final String[][] splitCfgs = {
        {";", "E", "=", "id"},
        {"}", "S", "{", "else", "}", "S", "{", ")", "C", "(",
"if"},
        {"}", "S", "{", ")", "C", "(", "while"},
        {"E'", "T"},
        {"E'", "T", "+"},
        {"-1"},
        {"T'", "F"},
        {"T'", "F", "*"},
        {"-1"},
        {")", "E", "("},
        {"num"},
        {"id"},
        {"C'", "D"},
        {"C'", "D", "||"},
        {"-1"},
        {")", "C", "("},
        {"num", "==", "id"}
};
```

感受与总结

通过自己动手查找资料、编写简单的语法分析程序，有助于对 LL(1)有更深入的了解。对了，还发现了上次词法分析程序中的 bug，哭晕在厕所。