编译系统课程实验报告

实验 2：语法分析

姓名	卢兑玳	院系	计算机学院	学号	L170300901
任课教师			指导教师		
实验地点			实验时间		
实验课表现	出勤、表现得分		实验报告得分		实验总分	
	操作结果得分					

一、需求分析

得分

要求：采用至少一种句法分析技术（LL(1)、SLR(1)、LR(1)或 LALR(1)）对类高级语言中的基本语句进行句法分析。阐述句法分析系统所要完成的功能。

在词法分析器的基础上设计实现类高级语言的语法分析器，基本功能如下：

（1） 能识别以下几类语句：

声明语句（包括变量声明、数组声明、记录声明和过程声明）

表达式及赋值语句（包括数组元素的引用和赋值）

分支语句：if_then_else

循环语句：do_while

过程调用语句

（2）系统的输入形式：要求通过文件导入测试用例。测试用例要涵盖“实验内容”第（1）条中列出的各种类型的语句。

（3）系统的输出形式：打印输出语法分析结果，格式如下：将构造好的语法分析树按照先序遍历的方式打印每一个结点的信息，这些信息包括结点的名称以及结点对应的成分在输入文件中的行号（行号被括号所包围，并且与结点名称之间有一个空格）。所谓某个成分在输入文件中的行号是指该成分产生的所有词素中的第一个在输入文件中出现的行号。对于叶结点，如果其 token 的属性值不为空，则将其属性值也打印出来。属性值与结点名称之间以一个冒号和空格隔开。每一个结点独占一行，而每个子结点的信息相对于其父结点的信息来说，在行首都要求缩进 2 个空格。

二、文法设计

得分

要求：给出如下语言成分的文法描述。

- 声明语句（包括变量声明、数组声明、记录声明和过程声明）
- 表达式及赋值语句（包括数组元素的引用和赋值）
- 分支语句：if_then_else
- 循环语句：do_while
- 过程调用语句

我的实验根据如下代码进行设计：

```
struct k{
int b;
};
int func(){ return 10;}//过程声明
void main(){
int a;//变量声明
int b[10][10];//数组声明
a=10;//赋值语句
a=a+3;//表达式
bb[1][2]=a;//数组元素赋值
a=b[1][2];//数组元素引用
if(a>0) { a=a+3; }//分支语句
else { a=a+3 ; }
while(a>0){ a=func; }//循环语句和过程调用
}
```

完整的语法如下（LL1 文法）：

```
P'-> P
P -> D1
P -> F1
D1 -> D D1
D1 -> D
F1 -> F F1
F1 -> F
D -> V ;
D -> V0 ;
V -> X id
V -> X L
V0 ->V = E
X -> int | folat | char
V1 -> V ; V1
V1 -> V ;
D -> struct id { V1 };
D -> H
H -> #include< N >
N -> stdio.h
N -> stdlib.h
```

C -> V
C -> V , C
S1 -> S S1
S1 -> S
F -> void id (C) ;
F -> V (C) ;
F -> V (C) { S1 }
F -> void id (C) { S1 }
S -> V ;
S -> V0;
S -> id = E ; | L = E ;
E -> E + E1 | E1
E1 -> E1 * E2 | E2
E2 -> (E) | - E | id | num | L
L -> id [E] | L [E]
S -> if B S0 else S0
S -> if B S else S
S -> while B do S0
S -> while B do S
S -> if B S0
S -> if B S
S0 -> { S1 }
B -> B or B1 | B1
B1 -> B1 and B2 | B2
B2 -> not B | (B) | E R E | true | false
R -> < | <= | == | != | > | >=
S -> id (EL)
EL -> EL , E
EL -> E
S -> return E ;

三、系统设计	得分	

要求：分为系统概要设计和系统详细设计。

（1）系统概要设计：给出必要的系统宏观层面设计图，如系统框架图、数据流图、功能模块结构图等以及相应的文字说明。

（2）系统详细设计：对如下工作进行展开描述

- ✓ 核心数据结构的设计
- ✓ 主要功能函数说明
- ✓ 程序核心部分的程序流程图

1. 系统的宏观设计原理如下

1.自顶向下的分析总是采用最左推导的方式，即：总是选择每个句型中的最左非终结符进行替换

2.根据输入流中的下一个终结符（向前看一个输入字符），选择最左非终结符的一个候选产生式，即 LL(1)

3.由于 LL(1)对文法有要求，因此首先要进行文法改造，即：消除直接左递归，消除间接左递归，提取左公因子

4.对于处理好的文法，需要进行下列处理

（1）从文法中获取所有终结符和非终结符

（2）计算每个文法符号（非终结符和终结符）的 FIRST 集：终结符的 FIRST 集就是本身，非终结符的 FIRST 集要迭代到所有 FIRST 集不发生变化为止（加入“空”也算发生变化）

（3）计算每个非终结符的 FOLLOW 集（若是某个句型的最右符号，则有$终结符）

（4）根据每个非终结符的 FIRST 集和 FOLLOW 集，计算每个产生式的 SELECT 集

（5）根据 SELECT 集构造预测分析表，行是非终结符，列是终结符

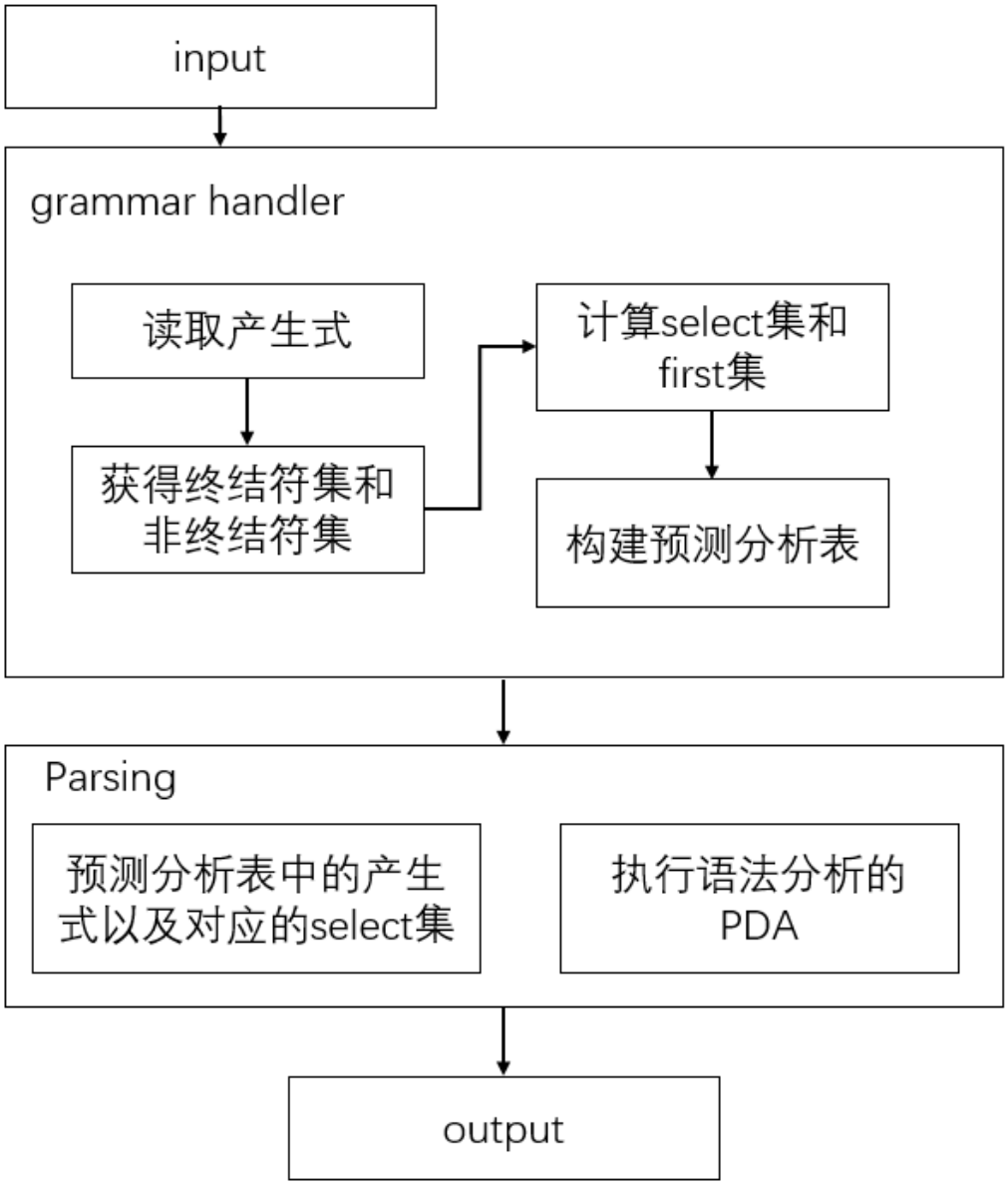
5.构造下推自动机 PDA，进行表驱动的预测分析（压栈弹栈）

6.错误处理是较为简单的恐慌模式，即：遇到错误就报错，并且忽略当前输入字符，输入指针向下移动。

grammar handler 负责从 grammar.txt 文件中读入 LL1 文法 然后自动构造 first 集 follow 集 select 集 再根据 select 集形成预测分析表 再写入到 grammar table. txt 里。

Parsing 类负责进行语法分析。

程序执行流程图如下：



（2）

我定义了如下数据结构：

在 Grammar_handler 类中：

```
ArrayList<String> terminals          = new ArrayList<String>();                                    //文法的终结符集合
        ArrayList<String> nonterminals       = new ArrayList<String>();                                    //文法的非终结符集合
        ArrayList<Production> productions  = new ArrayList<Production>();                                //文法的产生式集合
        HashMap<String, ArrayList<String>> firstSets  = new HashMap<String, ArrayList<String>>();    //FIRST 集的集合 格式:<文法符号，
FIRST 集>
        HashMap<String, ArrayList<String>> followSets  = new HashMap<String, ArrayList<String>>();   //FOLLOW 集的集合 格式:<非终结
符，FOLLOW 集>
```

在 Parsing 类中：

```
int  rowNumber;                              // 用于输出语法分析树中的结点对应的成分在输入文件中的行号
        List<String[]> stack;                    // PDA 的栈结构，栈元素格式为<元素，语法树中的层数>
        List<String[]> inputCache;               // PDA 的输入缓冲区,词法分析的结果，格式为<种别码，属性值，行号>或<种别码，行号>
        List<String> parsingResult;              // 语法分析的结果，格式为“符号（行号）”或“种别码 ： 属性值（行号）”，缩进格数为：2*节点深度
        List<String[]> errorMessages;            // 语法分析的结果，格式为“符号（行号）”或“种别码 ： 属性值（行号）”，缩进格数为：2*节点深度
        Map<String, String> predictionTable;     // 预测分析表，格式为<产生式左部-输入符号,产生式右部>
```

①核心算法描述

计算 SELECT 集的算法：

计算 FIRST 集的算法：

终结符的 FIRST 集计算：遍历所有终结符，对于终结符 X，它的 FIRST 集就是{X}

非终结符的 FIRST 集计算：迭代计算(缩进代表循环的层次)

迭代求得非终结符的 FIRST 集,即：不断重复遍历产生式并重复计算 FIRST 集,所有非终结符的 FIRST 集不发生变化则停止迭代

遍历所有产生式，对每一个产生式进行处理

将产生式右部第一个符号的 FIRST 集加入产生式左部非终结符的 FIRST 集中

若第一个符号可空，则将第二个符号做上述同样处理，以此类推

若第一个符号不可空，则直接结束该产生式的 FIRST 集的计算，开始计算下一个产生式的 FIRST 集

若整个产生式右部可空，则将 Σ 加入到产生式左部符号的 FIRST 集中

伪代码：

终结符：FIRST(X)={X};

非终结符：

for each X: FIRST(X)=null;

while(存在 FISRT 集发生变化){

for(每个产生式){

for(产生式右侧的每个符号 Y){

FISRT(X).add(FIRST(Y));

if(Y 不可空) break;

}

}

}

计算 FOLLOW 集的算法：

非终结符的 FOLLOW 集计算：迭代计算(缩进代表循环的层次)(仅有非终结符有 FOLLOW 集)

迭代求得非终结符的 FOLLOW 集,即：不断重复遍历产生式并重复计算 FOLLOW 集,所有非终结符的 FOLLOW 集不发生变化则停止迭代

遍历所有产生式，对每一个产生式进行处理

遍历产生式右部每一个符号

将当前符号的后一个的文法符号的 FIRST 集加入到当前符号的 FOLLOW 集中，并判断当前符号的后一个的文法符号是否可空

若当前文法符号不可空，则结束当前符号的 FOLLOW 集的计算

若可空，则将再后一个符号的 FIRST 集也加入到当前符号的 FOLLOW 集中，并判断是否可空（即重复上述操作）

若当前符号在该产生式的的后继字串可空，则将产生式左部符号的 FOLLOW 集加入到当前符号的 FOLLOW 集中

伪代码：

```
for each X: FOLLOW(X)=null;
FOLLOW(S).add(#);
while(存在 FOLLOW 集发生变化){
for(每个产生式 X-> α ){
for(产生式右侧的每个符号 Y){
for(Y 之后的每个符号 Z){
FOLLOW(Y).add(FIRST(Z));
if(Z 不可空) break;
}
if(符号 Y 之后的子串可空)
FOLLOW(Y).add(FOLLOW(X));
}
}
}
```

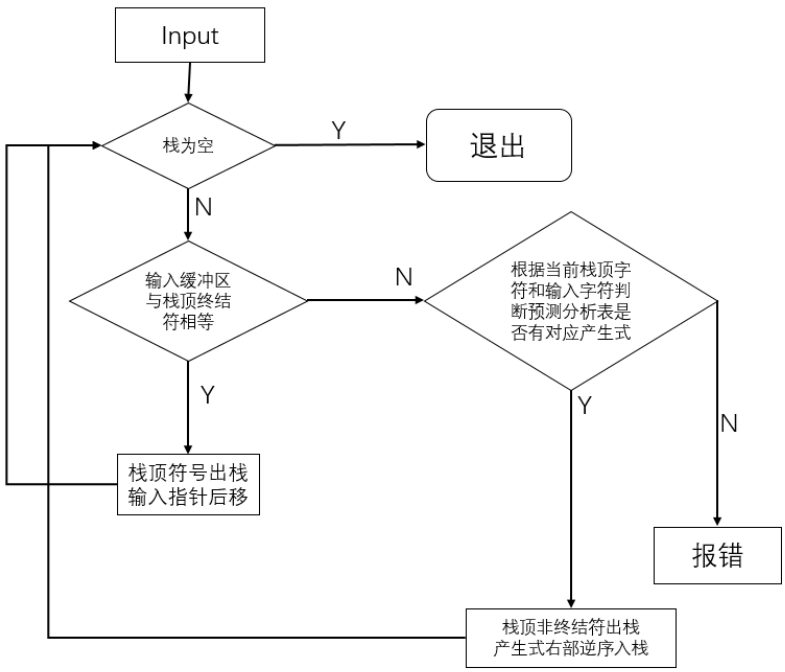
核心函数：

1. 执行语法分析时所用的 PDA

```
//执行语法分析的 PDA，在符号出栈时打印输出
public void PDA() {
        // 初始符号压入栈
        stack.add(new String[] { "Program", "0" });
        while (stack.size() > 0 && inputCache.size() > 0) {
                System.out.println("栈顶元素："+stack.get(stack.size() - 1)[0]+"  当前输入信号:"+inputCache.get(0)[0]);
                // 输入缓冲区与栈顶终结符相等时，栈顶符号出栈,输入指针指向下一个输入符号
                if (inputCache.get(0)[0].equals(stack.get(stack.size() - 1)[0])) {
                        popPrint(Integer.valueOf(stack.get(stack.size() - 1)[1]), inputCache.get(0));
                        inputCache.remove(0);
                        stack.remove(stack.size() - 1);
                        continue;
                }
                else {
                        // 根据当前栈顶字符和输入字符，在预测分析表中查找是否有对应产生式
                        String productionRights;
                        String productionLeft_Input = stack.get(stack.size() - 1)[0] + "-" + inputCache.get(0)[0];
```

```
                        // 能够在预测分析表中找到匹配的产生式
                        if ((productionRights = predictionTable.get(productionLeft_Input)) != null) {
                                // 栈顶非终结符出栈
                                int treeDepth = Integer.valueOf(stack.get(stack.size() - 1)[1]);
                                popPrint(treeDepth, new String[] { stack.get(stack.size() - 1)[0] });
                                stack.remove(stack.size() - 1);
                                //产生式右部逆序入栈(即产生式右部最末的符号最先进栈)
                                if (!productionRights.equals("$")) {
                                        String[] productionRight = productionRights.split(" ");
                                        for (int i = productionRight.length - 1; i > -1; i--) {
                                                stack.add(new String[] { productionRight[i], String.valueOf(treeDepth + 1) });
                                        }
                                }
                        }
                        // 不能在预测分析表中找到匹配的产生式的话，报错
                        else {
                                errorMessages.add(new String[]{"Error at Line"+rowNumber,
                                                stack.get(stack.size() - 1)[0] + "无法通过文法规则转换成
"  + inputCache.get(0)[0]});
                                inputCache.remove(0);
                        }
                }
        }
    }
```

我设计的 PDA 程序流程图如下：



四、系统实现及结果分析	得分	

要求：对如下内容展开描述。

（1） 系统实现过程中遇到的问题；

（2） 输出该句法分析器的分析表；

（3） 针对一测试程序输出其句法分析结果；

（4） 输出针对此测试程序对应的语法错误报告；

（5） 对实验结果进行分析。

注：其中的测试样例需先用已编写的词法分析程序进行处理。

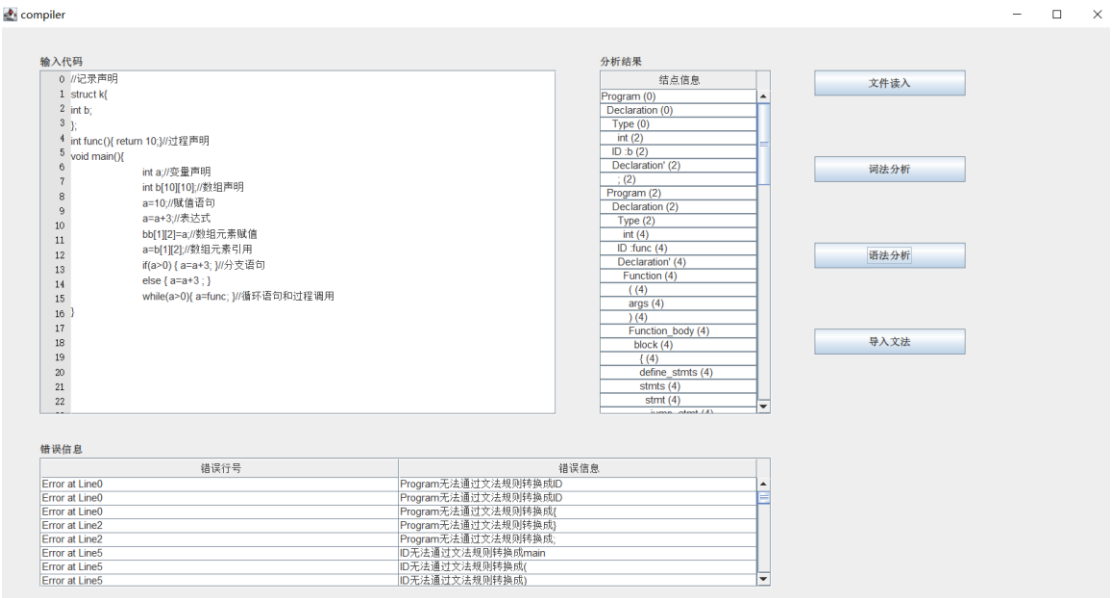
实验中遇到的主要问题在错误处理、错误恢复以及错误信息的提示，如果仅仅使用恐慌模式 进行不断读入，会导致当前的状态栈和符号栈中的信息仍然与输入的串不符合，会导致进一 步的向后的正确的代码段被掩盖。 因此，最终采用的是类似恐慌模式的错误处理，即先弹出状态栈中的栈顶状态，利用状态栈 和符号栈相差为一的性质，对符号栈栈顶的非终结符和状态栈栈顶的状态。

文法不允许外部声明，必须要有 main 函数。

在设计 LL1 文法的时候遇到了一些困难，我先完成了自动生成 fisrst 和 follow 集合，再根据预测分析表设计了文法。这样保证文法正确的话 预测分析表就正确。

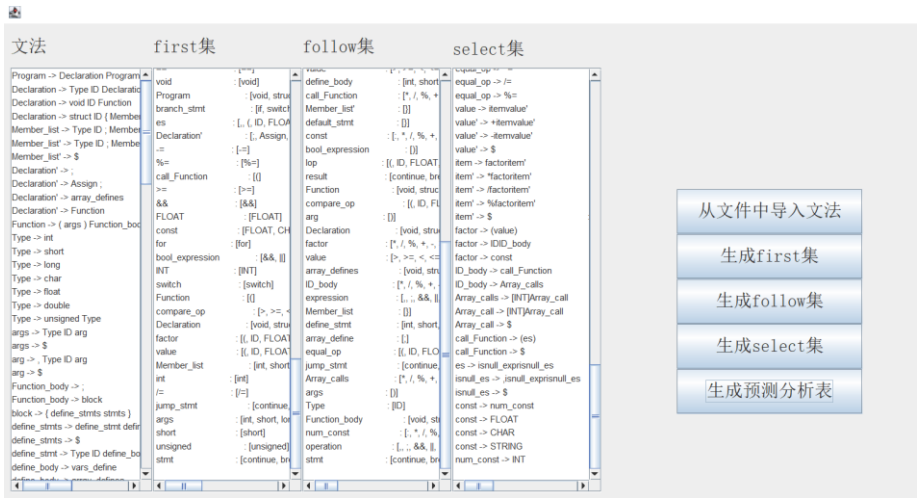
可以利用出错时的输入符号和最终恢复正常的语法分析的符号进行推测出可能的出错信息， 并作为错误信息的提示。

运行结果如下：

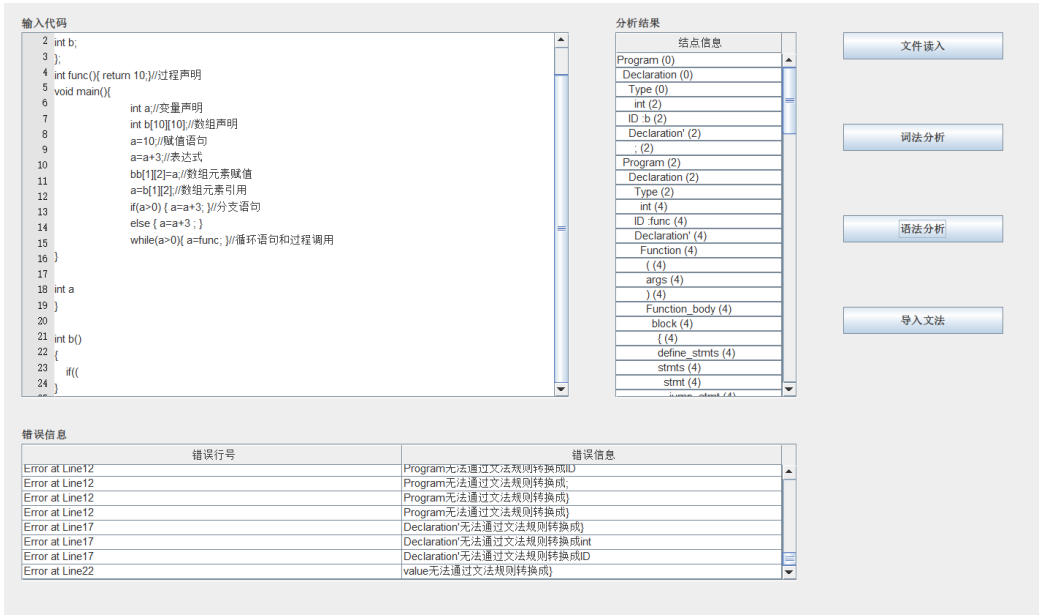


输出的语法分析表如下：

文法、first 集，follow 集和 select 集如下：



错误报告如下：



指导教师评语：   日期：