苏 州 大 学 实 验 报 告

院、系	计算机科学与技术学院	年级专业	21 计科	姓名	赵鹏	学号	2127405037
课程名称	编译原理实践					成绩	
指导教师	段湘煜	同组实验者	无		实验日期	2023.11.6	

实 验 名 称 LL 语法分析

一. 实验题目

利用预测语法分析法，实现 TEST 语言的 LL 语法分析器。

二. 实验原理及流程框图

1. 构建 LL(1)分析表

1.1. 计算 FIRST 集合

假定是β是文法 G 的任意符号串，或$\beta \in (Vt \cup Vn)^*$，则 FIRST(β)={ $a \mid \beta \Rightarrow^* a \ldots, a \in V!$}。若$\beta \Rightarrow^* \varepsilon$，则规定$\varepsilon \in$FIRST($\beta$)。FIRST 集合构造方法如下：

对于文法中的符号 X$\in Vt \cup Vn$，其 FIRST(X)集合可反复应用下列规则计算，直到其 FIRST(X)集合不再增大为止。

(1) 若 X$\in Vt$，则 FIRST(X)={X}。

(2) 若 X$\in Vn$，且具有形如 X→aα的产生式(a$\in Vt$)，或具有 X→ε的产生式，则把 a 或ε加进 FIRST(X)。

(3) 设 G 中有形如 X→Y1Y2…Yk 的产生式，若 Y1$\in Vn$，则把 FIRST(Y1)中的一切非ε符号加进 FIRST(X)；对于一切 $2 \leqslant i \leqslant k$，若 Y1Y2…Yi-1 均为非终结符号，且$\varepsilon \in$FIRST(Yj)，$1 \leqslant j \leqslant i-1$，则将 FIRST(Yi)中的一切非$\varepsilon$符号加进 FIRST(X)；但若对一切 $1 \leqslant i \leqslant k$，均有$\varepsilon \in$FIRST(Yi)，则将$\varepsilon$加进 FIRST(X)。

1.2. 计算 FOLLOW 集合

假定 S 是文法的开始符号，对于 G 的任何非终结符号A，则

FOLLOW(A)={ a | S⟹* … A a …, a ∈ Vt}

若 S⟹* … A，则规定\$∈FOLLOW(A)，\$是句尾标志。

FOLLOW(A)就是在所有句型中紧接A后出现的终结符或\$。对于文法符号A ∈ Vn，FOLLOW(A)集合的计算可反复应用下列规则，直到 FOLLOW(A)集合不再增大为止：

(1) 对于 S，令\$∈FOLLOW(S)。

(2) 若 G 中形如 B→$\alpha A\beta$的产生式，且$\beta \neq \varepsilon$，则将 FIRST(β)中的一切非ε符号加进 FOLLOW(A)中。

(3) 若 G 中有形如 B→αA或 B→$\alpha A\beta$的产生式，ε ∈FIRST(β)，则 FOLLOW(B)中的全部元素均属于 FOLLOW(A)。

1.3. 构造预测分析表

(1) 对文法的每个产生式 A→a，执行(2)和(3)

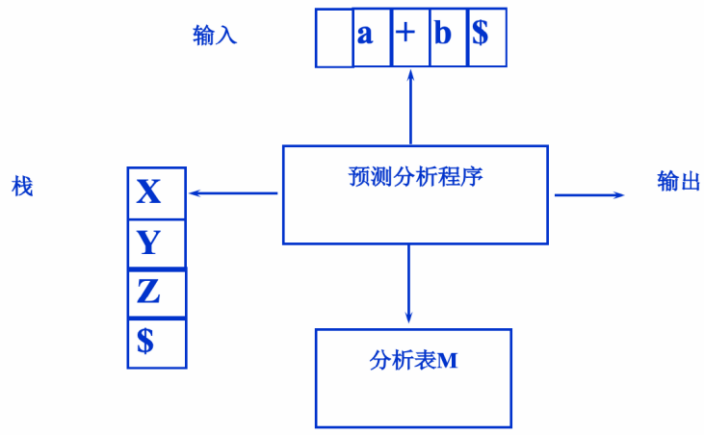
(2) 对 FIRST(a)的每个终结符 a，把 A→a 加入 M[A, a]

(3) 如果ε在 FIRST(a)中，对 FOLLOW(A)的每个终结符 b(包括\$)，把 A→a 加入 M[A, b]

(4) M 中其它没有定义的条目都是 error

1.4. 进行 LL 语法分析

LL 分析器结构如下图：



需要维护一个栈，一个输入序列，预测分析程序会根据分析表 M 做出决策，最后生成整棵语法树。

预测分析程序：

初始化：S $在栈里，其中 S 是文法的开始符号并且在栈顶；w $ 为输入序列 让 ip 指向 w $ 的第一个符号

主程序：令 X 等于栈顶符号，并且 a 等于 ip 指向的符号；

Repeat

If X 是终结符

If X == a

把 X 从栈顶弹出并推进 ip；

Else Error();

Else if M[X,a] = X → Y1Y2…Yk /*X 是非终结符*/

从栈中弹出 X；

把 Yk, Yk-1, … Y1 依次压入栈，Y1 在栈顶；

输出产生式 X→ Y1Y2…Yk /*对应子树 X 为父节点，Y1Y2…Yk 为子节点 */ Else Error();

Until X == $且 ip 指向$ /*栈空且输入序列到尾部*/

三. 实验步骤

(1) 实现简单加乘法的表达式的 LL 分析器

1.存储文法规则

进行 LL 语法分析需要依靠预测分析表，而构建预测分析表需要基于文法计算其 FIRST 集、FOLLOW 集,为了实现便于编写代码计算 FIRST 集和 FOLLOW 集，首先需要将文法存储在程序中。

实现时首先定义了语法单元类，用于存储产生式中的终结符和非终结符，每个语法单元类存储了语法单元的名称、是否是终结符、这个词法单元在源文件中的行号和列号(用于报告错误位置)、以及其在这一语法单元中的编号(用于绘制可视化的语法树)。这样就可以用 vector<GrammarItem>表示存储一个产生式。对于每个非终结符，它可能有多个产生式，因此可以用 vector<vector<GrammarItem>>存储每个非终结符的产生式集合，对于整个文法，只需要用一个 map 进行存储即可，其中 key 为非终结符的 GrammarItem,value 为其产生式的集合。因此，简单加

乘法表达式的产生式存储如下：

```
map<GrammarItem, vector<vector<GrammarItem>>> grammar
{
    {GrammarItem("E", false), {{GrammarItem("T", false), GrammarItem("E'", false)}}},
    {GrammarItem("E'", false), {{GrammarItem("+", true), GrammarItem("T", false), GrammarItem("E'", false)}, {GrammarItem(epsilon, true)}}},
    {GrammarItem("T", false), {{GrammarItem("F", false), GrammarItem("T'", false)}}},
    {GrammarItem("T'", false), {{GrammarItem("*", true), GrammarItem("F", false), GrammarItem("T'", false)}, {GrammarItem(epsilon, true)}}},
    {GrammarItem("F", false), {{GrammarItem("(", true), GrammarItem("E", false), GrammarItem(")", true)}, {GrammarItem("id", true)}}}
};
```

2.计算 FIRST 集合

编写代码计算 FIRST 集合的函数，FIRST 集合可以用 map<Grammar,Set<GrammarItem>>进行存储。首先遍历文法，将所有非终结符加入其 first 集合中，随后遍历每个产生式，依照实验原理中计算 FIRST 集合的方法，编写代码计算即可，需要注意的时，需要记录每轮遍历是否有新的元素添加到 first 集合中这一动作，若某一轮不再更新，则计算完成。

3.计算 FOLLOW 集合

编写代码计算 FIRST 集合的函数，FOLLOW 集合的存储和 FIRST 集合一致。基本思路是遍历每个产生式，将产生式中后一个语法单元的 FIRST 集合加入前一个集合的 FOLLOW 集合中，对于产生式可以为空的情况进行一些特殊处理即可。重复这一操作，直至某一轮不再有新的元素加入 FOLLOW 集合。

4.计算预测分析表

预测分析表的作用是依靠一个非终结符的每个产生式 FIRST 集合决定对于一个一个输入的终结符，选择使用哪一个产生式。可以使用 map<GrammarItem, map<GrammarItem>，vector<vector<GrammarItem>>>> 存储预测分析表，因为 TEST 文法中存在产生式 FIRST 集合冲突的问题，因此分析表中的每一项使用 vector<产生式>存储，这样也有利于代码的调试。预测分析表的基本思路是遍历每个产生式，计算其 FIRST 集合，将这个产生式加到预测分析表中对应的项中。当这个产生式可以推出空时，对于其 FOLLOW 集合中的每个非终结符，将产生式推出空加到表中的对应位置即可。

5.进行 LL(1)语法分析

LL(1)语法分析的实现相对简单，首先将$和开始符号 E 压入栈中，随后依次输入词法单元，当当前栈顶是非终结符时，使用预测分析表中的对应的产生式进行替代。若当前栈顶是终结符，则判断是否与当前输入的终结符匹配，匹配则

读取下一个词法单元，不匹配则直接报错。重复这一过程直至栈顶为$为止。为了便于将语法树可视化，我将推导的过程使用产生式集合进行了存储,在分析完成后调用绘画的函数，传入产生式集合作为参数，进行绘制。

6.绘制语法树

6.1 Dot 语言简介

语法树的绘制使用开源工具包 Graphviz 中的 Dot 语言进行绘制。Dot 语言可以使用 digraph{}绘制树，可以使用 name[lable=“content”]声明图中的节点，其中 content 为节点中的文字。可以使用 name1-name2 绘制一条将节点连接起来的边。若直接写 name1-name2 而不提前声明，则相当于 name[lable=“name1”], name2[lable=“name2”]。可以使用“ dot -Tpng draw.dot -o draw.png”即可将 dot 编译为 png 文件。

6.2 使用 dot 绘制语法树

绘制语法树的重点在于预测语法分析过程中的预处理操作，关键在于需要区分同名的语法单元，因此需要对于每个语法单元需要存储其是当前语法单元的第几个，这也就是 GrammarItem 中 id 的作用，只要在 lable 中填入词法单元名+id 即可完成区分。只需要在完成使用一个产生式替换一个非终结符时，将这个产生式同时存入结果的产生式集合中即可，值得注意的是需要正确处理 GrammarItem 的 id 信息，在使用产生式推导出新的 GrammarItem 时，需要将新产生式中每个词法单元的计数器增加，确保 id 正确。最后只需要依据预测分析的产生式集合声明 dot 语言中节点和边，再使用 system(“ dot -Tpng draw.dot -o draw.png”)即可自动生成语法树的图片。

(2) TEST 语言预测分析

在正确实现了简单加乘法表达式的预测语法分析之后，TEST 语言的预测语法分析就相当简单了，具体分为以下几个步骤。

1.存储文法

依照上述的方法将 TEST 语言的文法存储起来，如下图所示：

```
};
map<GrammarItem, vector<vector<GrammarItem>>> grammar{
    {GrammarItem("program", false), {{GrammarItem("{", true), GrammarItem("declaration_list", false), GrammarItem("statement_list", false), GrammarItem("}", true)}}},
    {GrammarItem("declaration_list", false), {{GrammarItem("declaration_listR", false)}}},
    {GrammarItem("declaration_listR", false), {{GrammarItem("declaration_stat", false), GrammarItem("declaration_listR", false)}, {GrammarItem(epsilon, true)}}},
    {GrammarItem("declaration_stat", false), {{GrammarItem("int", true), GrammarItem("ID", true), GrammarItem(";", true)}}},
    {GrammarItem("statement_list", false), {{GrammarItem("statement_listR", false)}}},
    {GrammarItem("statement_listR", false), {{GrammarItem("statement", false), GrammarItem("statement_listR", false)}, {GrammarItem(epsilon, true)}}},
    {GrammarItem("statement", false), {{GrammarItem("if_stat", false)}, {GrammarItem("while_stat", false)}, {GrammarItem("for_stat", false)}, {GrammarItem("read_s
    {GrammarItem("if_stat", false), {{GrammarItem("if", true), GrammarItem("(", true), GrammarItem("expression", false), GrammarItem(")", true), GrammarItem("stat
    {GrammarItem("else_stat", false), {{GrammarItem("else", true), GrammarItem("statement", false)}, {GrammarItem(epsilon, true)}}},
    {GrammarItem("while_stat", false), {{GrammarItem("while", true), GrammarItem("(", true), GrammarItem("expression", false), GrammarItem(")", true), GrammarItem
    {GrammarItem("for_stat", false), {{GrammarItem("for", true), GrammarItem("(", true), GrammarItem("expression", false), GrammarItem(";", true), GrammarItem("ex
    {GrammarItem("write_stat", false), {{GrammarItem("write", true), GrammarItem("expression", false), GrammarItem(";", true)}}},
    {GrammarItem("read_stat", false), {{GrammarItem("read", true), GrammarItem("ID", true), GrammarItem(";", true)}}},
    {GrammarItem("compound_stat", false), {{GrammarItem("{", true), GrammarItem("statement_list", false), GrammarItem("}", true)}}},
    {GrammarItem("expression_stat", false), {{GrammarItem("expression", false), GrammarItem(";", true)}, {GrammarItem(";", true)}}},
    {GrammarItem("expression", false), {{GrammarItem("ID", true), GrammarItem("=", true), GrammarItem("bool_expr", false)}, {GrammarItem("bool_expr", false)}}},
    {GrammarItem("bool_expr", false), {{GrammarItem("additive_expr", false), GrammarItem("bool_expr_end", false)}}},
    {GrammarItem("bool_expr_end", false), {{GrammarItem(epsilon, true)}, {GrammarItem(">", true), GrammarItem("additive_expr", false)}, {GrammarItem("<", true), G
    {GrammarItem("additive_expr", false), {{GrammarItem("term", false), GrammarItem("additive_expr_end", false)}}},
    {GrammarItem("additive_expr_end", false), {{GrammarItem("+", true), GrammarItem("term", false)}, {GrammarItem("-", true), GrammarItem("term", false)}, {Gramma
    {GrammarItem("term", false), {{GrammarItem("factor", false), GrammarItem("term_end", false)}}},
    {GrammarItem("term_end", false), {{GrammarItem("*", true), GrammarItem("factor", false)}, {GrammarItem("/", true), GrammarItem("factor", false)}, {GrammarItem
    {GrammarItem("factor", false), {{GrammarItem("(", true), GrammarItem("expression", false), GrammarItem(")", true)}, {GrammarItem("ID", true)}, {GrammarItem("N
};
```

2.修改文法

初始的 TEST 语言文法中存在很多的问题，如很多可以避免的 FIRST 集合交叉，未消除的左递归等等。

对于如下产生式需要利用提取左公因子进行修改操作：

1. $< bool\_expr >\rightarrow< additiv\_expr > | < additive\_expr > (> | < | >= | <= | == |! =) < additive\_expr >$
2. $< additive\_expr >\rightarrow< term > \{(+|-) < term >\}$
3. $< term >\rightarrow< factor > \{(* |/) < factor >\}$

以$bool_expr$举例，修改为：

$< bool\_expr >\rightarrow< additiv\_expr >< bool\_expr\_end >$

$< bool\_expr\_end >\rightarrow (> | < | >= | <= | == |! =) < additive\_expr > |\varepsilon$

对于如下产生式需要消除左递归：

$< declaration\_list > \rightarrow < declaration\_list >< declaration\_stat > |\varepsilon$

$< statement\_list >\rightarrow< statement\_list >< statement > |\varepsilon$

以$declaration_list$举例，修改为：

$< declarationlist >\rightarrow \varepsilon < R >$

$< \mathrm{R} >\rightarrow< declarationstat >< R >|\varepsilon$

这样就可以消除大部分不符合 LL(1)文法的情况。剩余的一个问题是对于$< expression >\rightarrow ID =< bool_expr > | < bool_expr >$两个产生式存在 FIRST 集合有交叉的情况，对于这种情况只需要再向前读一位，若是=则使用前一个产生式，否则使用后一个产生即可。

四. 实验结果及分析

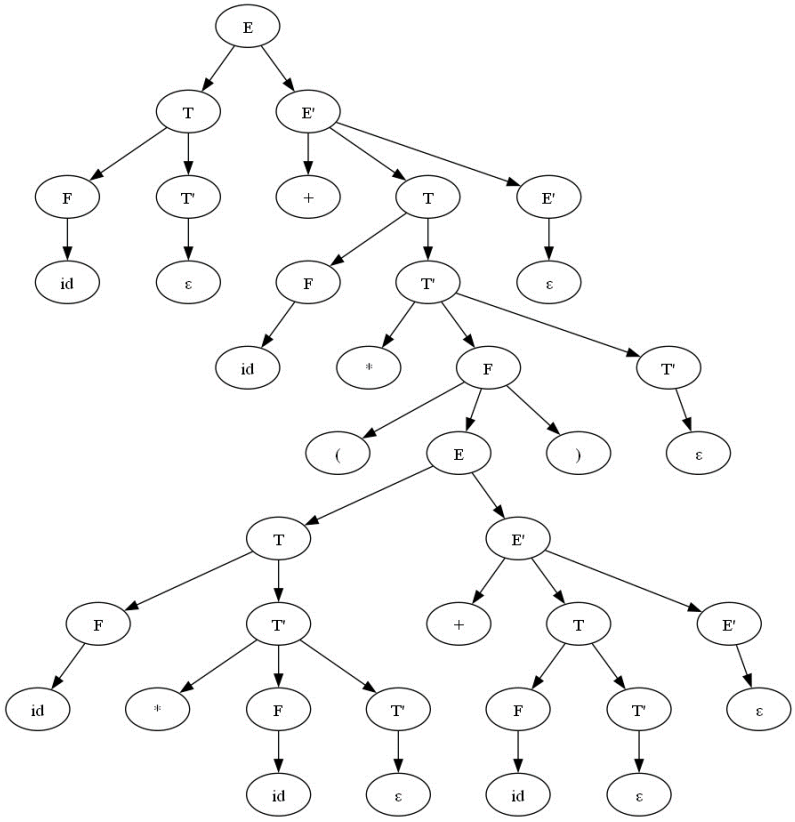
（1） 简单加乘法表达式

测试一

输入: id + id * (id * id + id) $

输出：

```
id + id * ( id * id + id ) $
E->T E'
T->F T'
F->id
T'->ε
E'->+ T E'
T->F T'
F->id
T'->* F T'
F->( E )
E->T E'
T->F T'
F->id
T'->* F T'
F->id
T'->ε
E'->+ T E'
T->F T'
F->id
T'->ε
E'->ε
T'->ε
E'->ε
```



测试二

输入: (id + id *) $

输入:

```
( id + id * ) $
E->T E'
T->F T'
F->( E )
E->T E'
T->F T'
F->id
T'->ε
E'->+ T E'
T->F T'
F->id
T'->* F T'
[ERROR] invalid synax
```

（2） TEST 语言

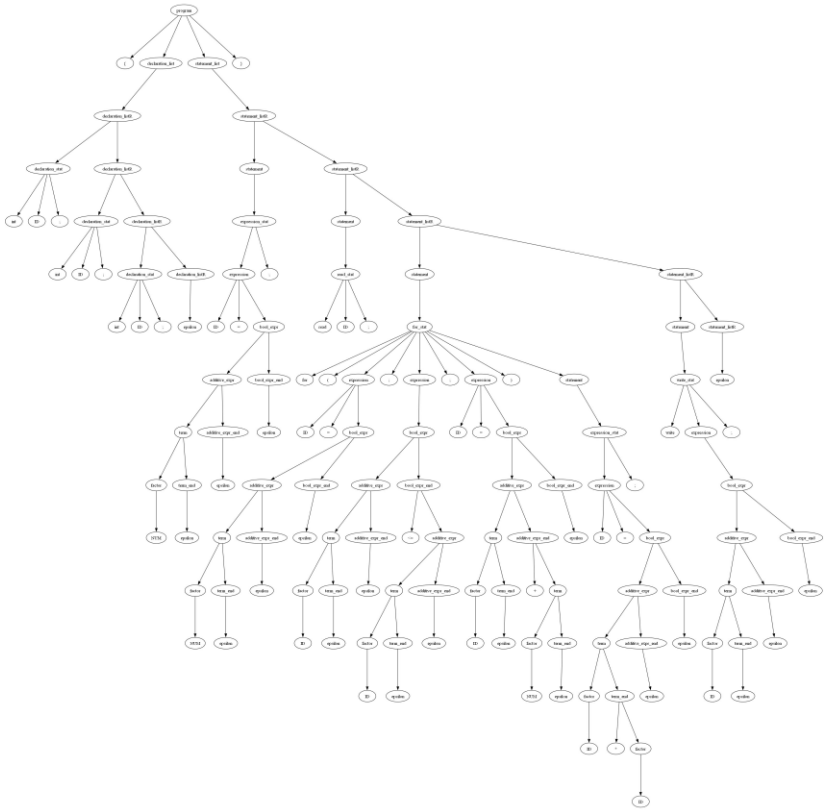
测试一

输入：

```
{
    int i;
    int n;
    int j;
    j=1;
    read n;
    for(i=1;i<=n;i=i+1)
  j=j*i;
    write j;
}
```

输出：

```
bool_expr->additive_expr bool_expr_end
additive_expr->term additive_expr_end
term->factor term_end
factor->ID
match ID
term_end->epsilon
additive_expr_end->epsilon
bool_expr_end->epsilon
match ;
statement_listR->epsilon
match }
--------------------draw:----------------------
success
```



测试二:

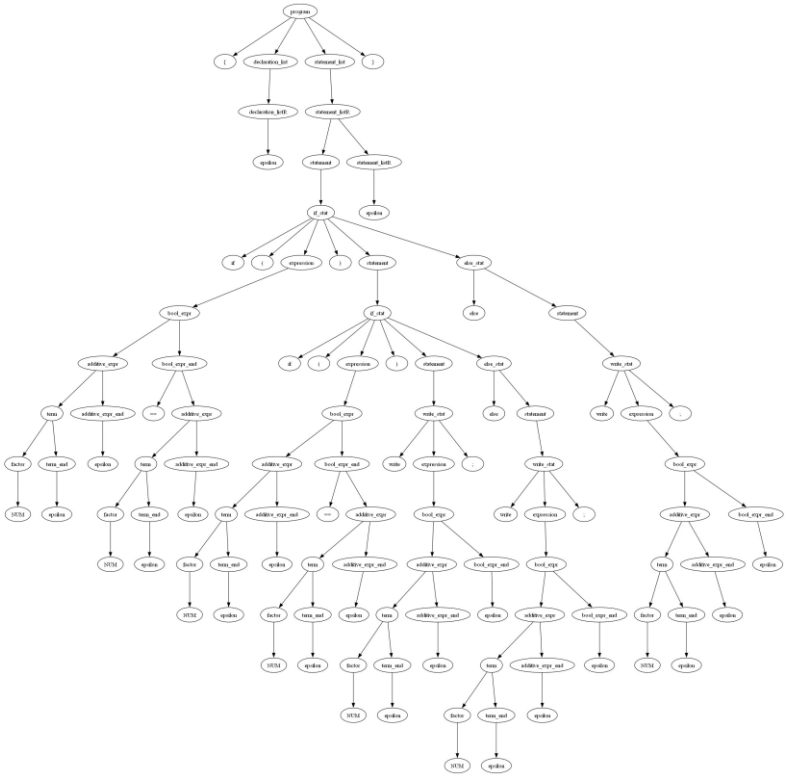
输入：

```
{
  if (1==2)
        if (2==3)
              write 1;
        else
              write 2;
  else
        write 3;
}
```

输出：

```
statement->write_stat
write_stat->write expression ;
[match write]
expression->bool_expr
bool_expr->additive_expr bool_expr_end
additive_expr->term additive_expr_end
term->factor term_end
factor->NUM
[match NUM]
term_end->epsilon
additive_expr_end->epsilon
bool_expr_end->epsilon
[match ;]
statement_listR->epsilon
[match }]
--------------------draw:-----------------------
success
```


教务处制



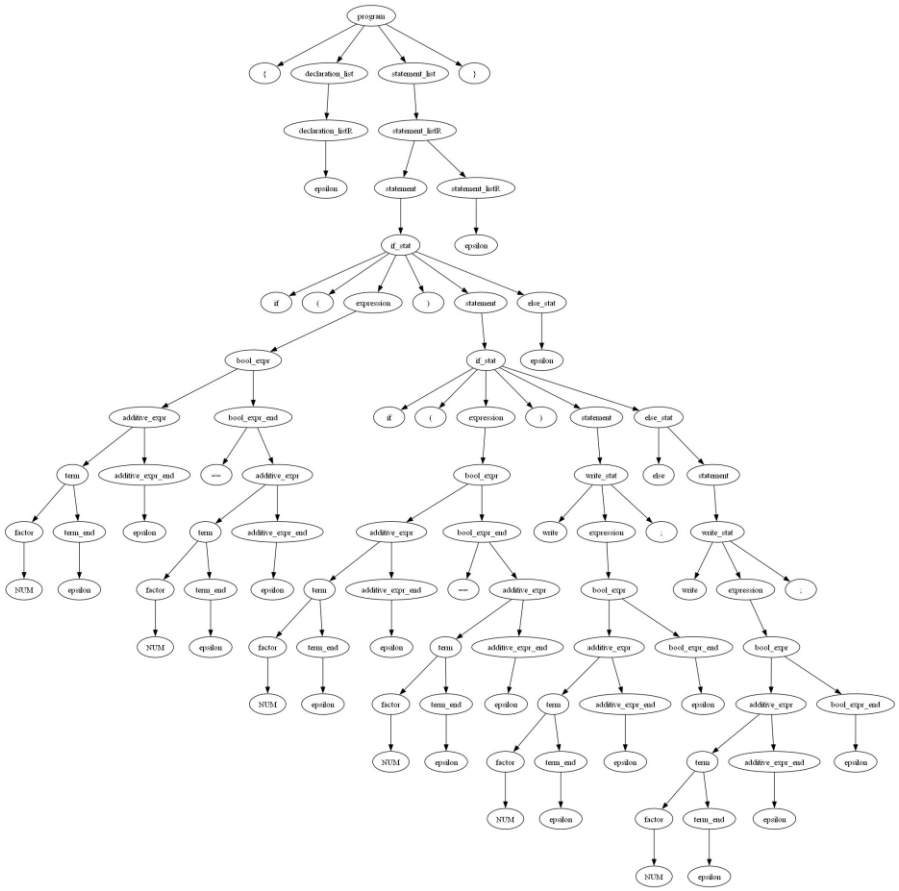
测试三:

输入：

```
{
    if (1==2)
        if (2==3)
            write 1;
        else
            write 2;
}
```

输出：

```
[match write]
expression->bool_expr
bool_expr->additive_expr bool_expr_end
additive_expr->term additive_expr_end
term->factor term_end
factor->NUM
[match NUM]
term_end->epsilon
additive_expr_end->epsilon
bool_expr_end->epsilon
[match ;]
else_stat->epsilon
statement_listR->epsilon
[match }]
------------------draw:----------------------
success
```



测试四：

输入：

```
{
    if (1==)
  write 1;


}
```

```
bool_expr->additive_expr bool_expr_end
additive_expr->term additive_expr_end
term->factor term_end
factor->NUM
[match NUM]
term_end->epsilon
additive_expr_end->epsilon
bool_expr_end->== additive_expr
[match ==]
2:12:Error in bool_expr_end
using Production:additive_expr->== additive_expr
) is in valid in additive_expr
error
```


教务处制

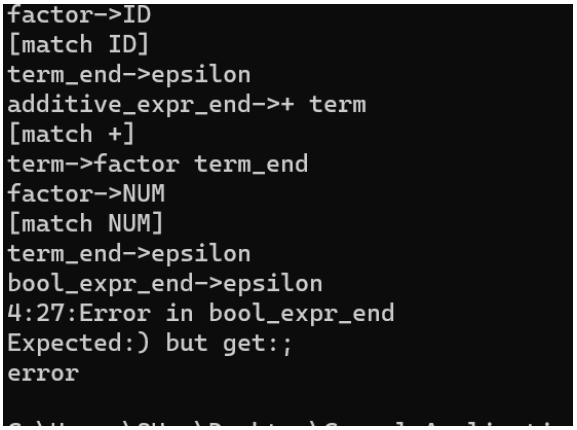
测试五：

输入：

```
{
    int a;
    a=1;
    for ( a=1;a<=100;a=a+1;)
  write a;

}
```

输出：



五. 实验总结

通过此次实验，我对 LL 语法分析有了更加深刻的理解，对进行预测语法分析的求解 FIRST、FOLLOW、计算分析表、进行预测语法分析的过程有了更加深入的了解。在实现预测语法分析的过程中，我使用了消除左递归、提取左公因子等方法修改文法，消除的文法中部分二义性的问题，对编译原理的理论知识有了更加好的掌握，在学习绘制语法树的过程中，我接触到了 dot 语言，并成功设计出能够根据输入的源代码使用 dot 语言自动绘制并输出语法树的语法分析器。代码能力有了一定的提升。

六. 代码

由于本次实验的代码设计到词法分析、语法分析等多个文件，故将代码添加到附件中展示。