一、内容

一）目标

设计并实现 Lexical Analyzer 中的 C 语言子集的语法分析程序，实现以下功能：

a) 可以识别出用 C 语言子集中数据类型和语句集编写的源程序，并以语法分析树的形式输出分析结果。
（基本数据类型包含整数、布尔类型；包含赋值语句、变量申明语句、算数加/减法表达式、布尔表达式、循环语句、分支语句）
b) 检查源程序中存在的语法错误，并报告错误所在的位置。
c) 能识别函数调用语句。
d) 对源程序中出现的错误进行适当的恢复，使得语法分析可以继续进行，检查并报告源程序中存在的所有语法错误及错误位置

二）C 语言子集

```
数据类型：int，无符号整数，取值范围 0-9999
        int a;
        int a,b;
        int a = 1;
算术运算符：+,-
        a = b + 1;
        a = b + c;
赋值运算符：=
        a = 1;
关系运算符：== ,>,<,<>,>=,<=
         a = (b==c);
         a = (b>c);
         a = (b<c);
逻辑运算符：&&，||，!
         a = (b&&c);
         a = (b||c);
         a = (!b);
条件语句：if
        if(a==b)
        {
        };
循环语句：while
       while(a==b)
       {

       };
输入,输出：get,put
    get(a);
    put(a);
语句结束符：  ;

条件语句 if else
if(a==b)
{

};
else
{

};
```

二、过程或算法

1、语言说明：

这里的文法只是为了描述设计的语法分析器包含 C 语言哪些子集，为了方便描述，这些文法均以最简洁且最具可读性的形式给出，但它们可能存在二义性、左递归、左公因子等问题，后面将对这些问题予以消除。

数据类型：

int：有符号整数，十进制或十六进制，32 位。

_Bool: 取值为 0 或 1，只有当赋值为 0 时取值为 0，赋值为其它整数时取值为 1。

文法：

$E \rightarrow int\ F;\ |\ \_Bool\ F;$

$F \rightarrow G, F\ |\ G$

$G \rightarrow id\ |\ id = expr$

其中 expr 为任意运算表达式，如无特殊说明，后面出现的 expr 含义与之相同。

运算符：

包括

算术运算符: +,-,*,/

关系运算符: ==,>,<,!=,>=,<=

逻辑运算符: &&,||,!

赋值运算符: =

文法：

$E \rightarrow id = F; |(id) = F;$

$F \rightarrow F + F|F - F|F * F|F/F|F == F|F > F|F < F|F! = F|F \geq F|F \leq F|F\&\&F|F\backslash|\backslash|F|!F|(F)|id$

条件语句：

文法：

$stmt \rightarrow if\ (\ expr\ )\{\ stmt\};\ |\ if\ (expr)\ \{\ stmt\}\ else\ \{stmt\}\ |\ other$

其中 other 表示任意其他语句。

循环语句：

文法：

$stmt \rightarrow while\ (\ expr\ )\ \{\ stmt1\};$

$stmt1 \rightarrow stmt\ |\ other$

函数调用语句:

文法：

$stmt \rightarrow id\ (\ expr\_list);$

$expr\_list \rightarrow expr\_list, expr\ |\ expr$

结构体：
文法：
$stmt \rightarrow struct\ id\ \{\ stmt1\ \};$
$stmt1 \rightarrow stmt \mid other$

语句结束符：
文法：$E \rightarrow ;$

2、选用语法分析方法及相应设计：
选用 LL(1)方法，计算 FIRST 集和 FOLLOW 集。采用非递归的预测分析方式。采用恐慌模式的恢复从错误中进行恢复。
首先消除前面所列文法的二义性、左递归和左公因子问题。
文法改写如下：
数据类型：
$E \rightarrow int\ F; \mid \_Bool\ F;$
$F \rightarrow G\ G'$
$G' \rightarrow ,F \mid \varepsilon$
$G \rightarrow id\ H$
$H \rightarrow \varepsilon \mid = expr$
运算符：
$E \rightarrow id = F; \mid (id) = F;$
$F \rightarrow G\ F'$
$F' \rightarrow\ \&\&\ G\ F' \mid \backslash\mid\backslash\mid G\ F' \mid \varepsilon$
$G \rightarrow\ H\ G'$
$G' \rightarrow == HG' \mid\ != HG' \mid \varepsilon$
$H \rightarrow I\ H'$
$H' \rightarrow >\ IH' \mid < IH' \mid\ >= IH' \mid <= I\ H' \mid \varepsilon$
$I \rightarrow J\ I'$
$I' \rightarrow\ +J\ I' \mid -J\ I' \mid \varepsilon$
$J \rightarrow K\ J'$
$J' \rightarrow * K\ J' \mid\ /\ K\ J' \mid \varepsilon$
$K \rightarrow\ !K \mid F \mid id \mid decimal \mid hex$
条件语句：
文法：
$stmt \rightarrow if\ (\ expr\ )\{\ stmt'\}; E$
$E \rightarrow \varepsilon \mid else\ \{stmt'\};$
$stmt' \rightarrow stmt \mid other$
循环语句：
文法：
$stmt \rightarrow while\ (\ expr\ )\ \{\ stmt1\ \};$
$stmt1 \rightarrow stmt \mid other$
函数调用语句:
文法：

$stmt \rightarrow id\ (\ expr\_list)$
$expr\_list \rightarrow expr\ E$
$E \rightarrow ,\ expr\ E \mid \varepsilon$
结构体：
文法：
$stmt \rightarrow struct\ id\ \{\ stmt1\ \};$
$stmt1 \rightarrow stmt \mid other$
语句结束符：
文法：$E \rightarrow ;$

将上述文法整合成一个 C 语言的文法，并给其中一部分非终结符更名：

$S \rightarrow T\ S \mid O\ S \mid C\ S \mid W\ S \mid id\ CC\ S \mid ST\ S \mid ;\ S \mid \varepsilon$
$T \rightarrow int\ decs\ ; \mid \_Bool\ decs\ ;$
$decs \rightarrow dec\ dec'$
$dec' \rightarrow ,decs \mid \varepsilon$
$dec \rightarrow id\ assign$
$assign \rightarrow \varepsilon \mid = expr$
$O \rightarrow (id) = expr;$
$expr \rightarrow G\ expr'$
$expr' \rightarrow\ \&\&\ G\ expr' \mid \backslash|\backslash|\ G\ expr' \mid \varepsilon$
$G \rightarrow\ L\ G'$
$G' \rightarrow \backslash|\ L\ G' \mid \varepsilon$
$L \rightarrow M\ L'$
$L' \rightarrow \&\ M\ L' \mid \varepsilon$
$M \rightarrow H\ M'$
$M' \rightarrow == HM' \mid\ != HM' \mid \varepsilon$
$H \rightarrow I\ H'$
$H' \rightarrow >\ IH' \mid < IH' \mid\ >= IH' \mid <= I\ H' \mid \varepsilon$
$I \rightarrow J\ I'$
$I' \rightarrow\ +J\ I' \mid -J\ I' \mid \varepsilon$
$J \rightarrow K\ J'$
$J' \rightarrow *\ K\ J' \mid\ /\ K\ J' \mid \varepsilon$
$K \rightarrow\ !\ K \mid (expr) \mid id \mid decimal \mid hex$
$C \rightarrow if\ (\ expr\ )\{\ S\ \};EL$
$EL \rightarrow \varepsilon \mid else\ \{\ S\ \};$
$W \rightarrow while\ (\ expr\ )\ \{\ S\ \};$
$CC \rightarrow (\ expr\_list) \mid 4\ =expr\ ;$
$expr\_list \rightarrow expr\ E$
$E \rightarrow ,\ expr\ E \mid \varepsilon$
$ST \rightarrow\ struct\ id\ \{\ ST'\ \};$
$ST' \rightarrow T\ ST' \mid ST\ ST' \mid ;ST' \mid \varepsilon$
每两个文法符号间都用空格隔开，为了提高可读性，有些非终结符用多个大写字母的组合来表示，如 FC 表示生成函数调用语句的非终结符。
根据该文法计算出的预测分析表过大，这里不再展示。

3、数据结构说明

输入符号数据结构：

```
struct inelm{
    string val,type;
    int id;
    pos p;
};
```

type 是符号所属的词法单元，val 是对应词素，id 是该词法单元在词法编码表中的编号，p 是该词素在待编译的代码中的位置，具体定义见 Lexical Analyzer。输入符号流用 vector<inelm>存储。

采用 map 存储文法，键是非终结符(产生式的头)，值是该非终结符可以一步推导出的式子(产生式的体)，用二维字符串数组 vector<vector<string> >来存储产生式的体。外层的 vector 存储同一个非终结符可以产生的不同体，内层的 vector 存储一个体的文法符号序列，每个符号以一个字符串的形式存储。

采用 set<string>存储 FIRST 集和 FOLLOW 集。采用 map 存储任意文法符号串到其 FIRST 集的映射关系和非终结符到其 FOLLOW 集的映射关系。其具体结构分别为 map<vector<string>, set<string> >和 map<string, set<string> >。

采用 map 存储为非终结符生成 FOLLOW 集，根据第三条规则所依赖的所有其它非终结符，具体结构为 map<string, set<string> >。

采用 map<pair<string,string>,vector<string> >结构存储预测分析表。它是一个非终结符-终结符对到用字符串数组表示的产生式体的映射。

4、语法树及输出说明

可在 webgraphviz 网站上可视化绘制语法树。语法分析器采用 LL(1)方法，在语法分析过程中将绘制语法树的代码输出到 Tree.txt 文件中。可直接复制该 txt 文件中的绘图代码，到网站 http://www.webgraphviz.com/上进行在线绘制。语法树中，空节点表示空串。待编译的代码位于文件“inputs.c”中。

5、错误处理

本语法分析器处理的错误包括三类：第一类是输入字符流中当前看到的终结符与当前栈顶终结符不一致；第二类是根据当前栈顶非终结符和输入字符流中当前看到的终结符，查找预测分析表，找不到对应的推导（产生式）；第三类是其它未知错误。每次检测到错误时，都根据当前错误类型输出相应的提示，并输出发生错误的位置（用待编译代码中的行和列表示）。采用恐慌模式的恢复策略，语法分析器一旦发现错误，就不断丢弃输入中的符号，一次丢弃一个符号，直到找到同步词法单元集合中的某个元素为止。

6、程序说明：

程序允许用户输入其设计的文法。getGrammars 函数用于接收用户输入的文法，并将其存储到 map 结构中。要求输入的文法满足以下条件：

1）开始符号为“S”。
2）非终结符的名字不为“end”。
3）一行输入一条产生式，相邻产生式之间以换行符隔开。
4）同一条产生式的相邻文法符号之间以一个空格隔开，文法符号与‘|’之间也以一个空格隔开。
5）每一行的开头是当前产生式的头，产生式的头与体之间以一个空格隔开，不需要输入箭头“→”。
6）作为文法符号的一部分而出现的‘|’字符前需加以转义字符‘\’。
7）转义字符‘\’只能用于转义，不能用于组成文法符号。
8）所有以同一个非终结符为头的体都应在同一行中输入，这些体之间用‘|’分隔。
9）用“\e”表示ε。
10）　　文法输入以一行单独的“end”结束。即输入完所有产生式后，另起一行输入“end”以结束输入。

设计 getF 函数用于辅助计算给定文法符号 X 的 FIRST 集。该函数首先从 FIRST 集的映射表 FIRST 中查找该 FIRST 集是否已被计算过，若找到，则直接返回，否则按计算 FIRST 集的步骤计算 X 的 FIRST 集，并将它存到映射表 FIRST 中，然后返回。计算过程中，可能需要递归调用该函数本身来计算非终结符产生式体中其它非终结符的 FIRST 集，文法设计保证该递归调用过程可以结束。

中其它非终结符的 FIRST 集，文法设计保证该递归调用过程可以结束。

```
set<string> getF(const string& X){
    /*
    Return FIRST(X) for grammar symbol X.
    */
    vector<string> F;
    set<string> s;
    F.push_back(X);
    if(isnull(X)){
        s.insert(X);
        return FIRST[F]=s;
    }
    if(lct.count(X)||isnull(X)){
        s.insert(X);
        return FIRST[F]=s;
    }
    if(FIRST.count(F))return FIRST[F];
    for(auto prd = grammars[X].begin();prd!=grammars[X].end();++prd)
{
        getFS(s,prd);
    }
    return FIRST[F]=s;
}
```

考虑到 LL(1)方法中只用到每个产生式体的 FIRST 集，因此可以预先计算给定文法的所有产生式体的 FIRST 集，并把它们存储起来。编写 getFIRST 函数完成这一功能。

```
void getFIRST(){
    /* get FIRST for every productions in the given grammars */
    for(auto prod=grammars.begin();prod!=grammars.end();++prod){
        for(auto body=prod->second.begin();body!=prod->second.end();
++body){
            getFS(FIRST[*body],body);
        }
    }
}
```

需要求每个非终结符的 FOLLOW 集，基本思想是不断应用求 FOLLOW 集的三条规则。但具体实现时，如果要求 FOLLOW 集的非终结符在产生式体的尾部，此时如果递归调用求 FOLLOW 集的函数来获得产生式头的 FOLLOW 集，则可能出现死循环，例如 A->a B B->a A。为了避免这种情况，首先遍历一遍整个文法，按照文法给出产生式的顺序，尝试求位于每个产生式头部的非终结符的 FOLLOW 集，将每个非终结符的 FOLLOW 集中所有不依赖于其它非终结符的 FOLLOW 集的部分确定下来，这些部分包括直接跟在非终结符后的终结符和非终结符的 FIRST 集等。同时获得为位于当前产生式头部的非终结符生成 FOLLOW 集所依赖的所有其它非终结符，如 A->a B 这条产生式中，要为非终结符 B 生成 FOLLOW 集，则依赖于非终结符 A，这时就把非终结符 A 添加到为 B 生成 FOLLOW 集所依赖的其它非终结符集中。之后不断遍历所有 FOLLOW 集依赖于其它非终结符的非终结符。每次遍历中，将每个非终结符的 FOLLOW 集替换为目前得到的该非终结符的 FOLLOW 集和所有生成该非终结符的 FOLLOW 集所依赖的其它非终结符的 FOLLOW 集的并集，直到一次遍历中没有任何一个非终结符的 FOLLOW 集发生变化。

```
// get FOLLOW sets for all the nonterminals in the given grammars
void getFOLLOW(){
    FOLLOW["S"].insert("$");
    printf("\n\nBegin getFOLLOW!\n\n");
    for(auto prd=grammars.begin();prd!=grammars.end();++prd){
        for(auto body = prd->second.begin();body!=prd->second.end();
++body){
            for(int si=0;si<body->size();++si){
                if(!lct.count((*body)[si])){
                    for(int j=si+1;j<body->size();++j){
                        string tj=(*body)[j];
                        set<string> ft=getF(tj);
                        string null(1,127);
```

```
                    ft.erase(null);
                    set_union(FOLLOW[(*body)[si]].begin(),
                        FOLLOW[(*body)[si]].end(),ft.begin(),
                        ft.end(),inserter(FOLLOW[(*body)[si]],
                        FOLLOW[(*body)[si]].begin()));
                    if(!getF(tj).count(null)){
                        break;
                    }
                    if(j==body->size()-1){
                        FD[(*body)[si]].insert(prd->first);
                    }
                }
                if(si==body->size()-1){
                    FD[(*body)[si]].insert(prd->first);
                }
            }
        }
    }
    while(true){
        int bk=0;
        for(auto sym=FD.begin();sym!=FD.end();++sym){
            int os=FOLLOW[sym->first].size();
            for(auto sd=sym->second.begin();sd!=sym->second.end();++
sd){
                set_union(FOLLOW[sym->first].begin(),FOLLOW[sym->fir
st].end(),
                    FOLLOW[*sd].begin(),FOLLOW[*sd].end(),
                    inserter(FOLLOW[sym->first],FOLLOW[sym->first].b
egin()));
            }
            if(FOLLOW[sym->first].size()>os){
                ++bk;
            }
        }
        if(bk==0)break;
    }
}
```

构建预测分析表可直接按照“龙书”上的算法。预测分析表中，空串无法显示，因此后面展示预测分析表时，对表中出现空串的位置进行了单独的逻辑判断。

```
void getPPT(){
    /* get predictive parsing table*/
    for(auto prd=grammars.begin();prd!=grammars.end();++prd){
```

```
        for(auto body=prd->second.begin();body!=prd->second.end();++
body){
            set<string> fA=FIRST[*body];
            string null(1,127);
            for(auto nsym=fA.begin();nsym!=fA.end();++nsym){
                if(*nsym!=null){
                    pair<string,string> pk;
                    pk.first=prd->first;
                    pk.second=*nsym;
                    ppt[pk]=*body;
                }else{
                    set<string> fF=FOLLOW[prd->first];
                    for(auto sym=fF.begin();sym!=fF.end();++sym){
                        if(*sym!=null){
                            pair<string,string>pk;
                            pk.first=prd->first;
                            pk.second=*sym;
                            ppt[pk]=*body;
                        }
                    }
                }
            }
        }
    }
}
```

根据 LL1 方法，进行语法分析，可直接按照“龙书”上的表驱动的预测语法分析算法。有一点需要注意，当在预测分析表中找到可以使用的产生式，要将产生式中的文法符号压栈时，需先检查要压栈的符号是否为空串，只有非空的符号才能被压栈。其中，采用恐慌模式的恢复从检测到的错误中进行恢复。

```
void LL1(){
    /*
    LL1 method
    Construct the parse tree
    */
    printf("Begin to construct parse tree!\n");
    ofstream tree("Tree.txt");
    int n=0;
    St.push(pair<string,int>("$",-1));
    St.push(pair<string,int>("S",0));
    tree << "digraph G{" << endl;
    pair<string,int> X("S",0);
    genNode(tree,X.first,n);
    auto a=inputs.begin();
    while(X.first!="$"){
        if(X.first==a->type){
```

```
            St.pop();
            ++a;
        }else if(lct.count(X.first)){
            printf("SYNTAX ERROR at Ln %d, Col %d! Expected a \"",a-
>p.line,a->p.column);
            cout << X.first << "\", but got a \"" << a->type << "\"!
" << endl;
            // Panic-Mode Recovery
            ++a;
            if(a->type=="$"){
                printf("Ended with error!\n");
                break;
            }
            continue;
        }else if(!ppt.count(pair<string,string>(X.first,a->type))){
            printf("SYNTAX ERROR at Ln %d, Col %d! Unexpected symbol
 \"",a->p.line,a->p.column);
            cout << a->type << "\"!" << endl;
            // Panic-Mode Recovery
            ++a;
            if(a->type=="$"){
                printf("Ended with error!\n");
                break;
            }
            continue;
        }else if(ppt.count(pair<string,string>(X.first,a->type))){
            vector<string> body=ppt[pair<string,string>(X.first,a->t
ype)];
            vector<pair<string,int>> notp;
            for(auto sym=body.begin();sym!=body.end();++sym){
                genNode(tree,*sym,++n);
                genEdge(tree,X.second,n);
                notp.push_back(pair<string,int>(*sym,n));
            }
            St.pop();
            /* push symbols onto the stack in a reverse order*/
            for(auto sel=notp.rbegin();sel!=notp.rend();++sel){
                if(sel->first!="\177")
                    St.push(*sel);
            }
        }else{
            printf("Unknown SYNTAX ERROR at Ln %d, Col %d!\n");
        }
        X=St.top();
    }
    tree << "}" << endl;
    printf("Finished constructing parse tree!\n");
}
```

三、测试结果

1、 测试样例:

1) 简易计算器

```
//输入数据 num1,num2,op，根据 op 确定操作进行运算，最后输出运算结果 ans
int num1,num2,op,ans;
get(num1,num2,op);
if(op==0)
{
   ans = num1 + num2;
};
if(op==1)
{
   ans = num1 - num2;
};
if(op==2)
{
   ans = num1 & num2;
};
if(op==3)
{
   ans = num1 | num2;
};
put(ans);
```

该样例生成的语法树过于庞大，附在附件“简易计算器.png”中。

2) 跑马灯

```
//循环输入 op，改变输出结果 out，输入 0 则结束程序
int num0,num1,out,op;
num1 = 3333;
num2 = 6666;
num3 = 9999;
op = 1;
while(op>0)
{
   if(op==1)
   {
      out = num1;
    };
   if(op==2)
   {
      out = num2;
   };
   if(op==2)
   {
      out = num3;
   };
   put(out);
   get(op);
};
```

该样例生成的语法树过于庞大，附在附件“跑马灯.png”中。

2、 自定义测试样例(包含自定义的词法错误类型):

C 语言文法输入样例：

S T S | O S | C S | W S | id CC S | ST S | ; S | \e

T int decs ; | _Bool decs ;

decs dec dec'

dec' , decs | \e

dec id assign

```
assign \e | = expr
O ( id ) = expr ;
expr G expr'
expr' && G expr' | \|\| G expr' | \e
G L G'
G'  \| L G' | \e
L M L'
L' & M L' | \e
M H M'
M' == H M' | != H M' | \e
H I H'
H' > I H' | < I H' | >= I H' | <= I H' | \e
I J I'
I' + J I' | - J I' | \e
J K J'
J' * K J' | / K J' | \e
K ! K | ( expr ) | id | decimal | hex
C if ( expr ) { S } ; EL
EL \e | else { S } ;
W while ( expr )  { S } ;
CC ( expr_list ) | = expr ;
expr_list expr E
E , expr E | \e
ST struct id { ST' } ;
ST' T ST' | ST ST' | ; ST' | \e
```

使用该样例对输入文法函数 getGrammars 进行单元测试，结果如下：

435 × 629

```
D:\Programs\Language\VSCode\C++\syntax_analyzer.exe
decs dec dec'
dec' , decs | \e
dec id assign
assign \e | = expr
O ( id ) = expr ;
expr G expr'
expr' && G expr' | \|\| G expr' | \e
G L G'
G' \| L G' | \e
L M L'
L' & M L' | \e
M H M'
M' == H M' | != H M' | \e
H I H'
H' > I H' | < I H' | >= I H' | <= I H' | \e
I J I'
I' + J I' | - J I' | \e
J K J'
J' * K J' | / K J' | \e
K ! K | ( expr ) | id | decimal | hex
C if ( expr ) { S } ; EL
EL \e | else { S } ;
W while ( expr ) { S } ;
CC ( expr_list ) | = expr ;
expr_list expr E
E , expr E | \e
ST struct id { ST' } ;
ST' T ST' | ST ST' | ; ST' | \e
end
C -> if ( expr ) { S } ; EL
CC -> ( expr_list ) | = expr ;
E -> , expr E |
EL ->  | else { S } ;
G -> L G'
G' -> | L G' |
H -> I H'
H' -> > I H' | < I H' | >= I H' | <= I H' |
I -> J I'
I' -> + J I' | - J I' |
J -> K J'
J' -> * K J' | / K J' |
K -> ! K | ( expr ) | id | decimal | hex
L -> M L'
L' -> & M L' |
M -> H M'
M' -> == H M' | != H M' |
O -> ( id ) = expr ;
S -> T S | O S | C S | W S | id CC S | ST S | ; S |
ST -> struct id { ST' } ;
ST' -> T ST' | ST ST' | ; ST' |
T -> int decs ; | _Bool decs ;
W -> while ( expr ) { S } ;
assign ->  | = expr
dec -> id assign
dec' -> , decs |
decs -> dec dec'
expr -> G expr'
expr' -> && G expr' | || G expr' |
expr_list -> expr E
Press any key to continue . . .
```

使用上述文法样例对 getFIRST 函数进行测试，结果如下：

```
FIRST( != H M' )={ != }
FIRST( & )={ & }
FIRST( & M L' )={ & }
FIRST( && )={ && }
FIRST( && G expr' )={ && }
FIRST( ( )={ ( }
FIRST( ( expr ) )={ ( }
FIRST( ( expr_list ) )={ ( }
FIRST( ( id ) = expr ; )={ ( }
FIRST( * )={ * }
FIRST( * K J' )={ * }
FIRST( + )={ + }
FIRST( + J I' )={ + }
FIRST( , )={ , }
FIRST( , decs )={ , }
FIRST( , expr E )={ , }
FIRST( - )={ - }
FIRST( - J I' )={ - }
FIRST( / )={ / }
FIRST( / K J' )={ / }
FIRST( ; )={ ; }
FIRST( ; S )={ ; }
FIRST( ; ST' )={ ; }
FIRST( < )={ < }
FIRST( < I H' )={ < }
FIRST( <= )={ <= }
FIRST( <= I H' )={ <= }
FIRST( = )={ = }
FIRST( = expr )={ = }
FIRST( = expr ; )={ = }
FIRST( == )={ == }
FIRST( == H M' )={ == }
FIRST( > )={ > }
FIRST( > I H' )={ > }
FIRST( >= )={ >= }
FIRST( >= I H' )={ >= }
FIRST( C )={ if }
FIRST( C S )={ if }
FIRST( G )={ ! ( decimal hex id }
FIRST( G expr' )={ ! ( decimal hex id }
FIRST( H )={ ! ( decimal hex id }
FIRST( H M' )={ ! ( decimal hex id }
FIRST( I )={ ! ( decimal hex id }
FIRST( I H' )={ ! ( decimal hex id }
FIRST( J )={ ! ( decimal hex id }
FIRST( J I' )={ ! ( decimal hex id }
FIRST( K )={ ! ( decimal hex id }
FIRST( K J' )={ ! ( decimal hex id }
FIRST( L )={ ! ( decimal hex id }
FIRST( L G' )={ ! ( decimal hex id }
FIRST( M )={ ! ( decimal hex id }
FIRST( M L' )={ ! ( decimal hex id }
FIRST( O )={ ( }
FIRST( O S )={ ( }
FIRST( ST )={ struct }
FIRST( ST S )={ struct }
FIRST( ST ST' )={ struct }
FIRST( T )={ _Bool int }
FIRST( T S )={ _Bool int }
FIRST( T ST' )={ _Bool int }
FIRST( W )={ while }
FIRST( W S )={ while }
FIRST( _Bool )={ _Bool }
FIRST( _Bool decs ; )={ _Bool }
FIRST( dec )={ id }
FIRST( dec dec' )={ id }
FIRST( decimal )={ decimal }
FIRST( else )={ else }
FIRST( else { S } ; )={ else }
FIRST( expr )={ ! ( decimal hex id }
FIRST( expr E )={ ! ( decimal hex id }
FIRST( hex )={ hex }
FIRST( id )={ id }
FIRST( id CC S )={ id }
FIRST( id assign )={ id }
FIRST( if )={ if }
FIRST( if ( expr ) { S } ; BL )={ if }
FIRST( int )={ int }
FIRST( int decs ; )={ int }
FIRST( struct )={ struct }
FIRST( struct id { ST' } ; )={ struct }
FIRST( while )={ while }
FIRST( while ( expr ) { S } ; )={ while }
FIRST( | )={ | }
FIRST( | L G' )={ | }
FIRST( || )={ || }
FIRST( || G expr' )={ || }
FIRST(  )={  }
```

使用上述文法样例对 getFOLLOW 函数进行测试，结果如下：

```
Begin getFOLLOW!

FOLLOW(C) = { $ ( ; _Bool id if int struct while } }
FOLLOW(CC) = { $ ( ; _Bool id if int struct while } }
FOLLOW(E) = { ) }
FOLLOW(EL) = { $ ( ; _Bool id if int struct while } }
FOLLOW(G) = { && ) , ; || }
FOLLOW(G') = { && ) , ; || }
FOLLOW(H) = { != & && ) , ; == | || }
FOLLOW(H') = { != & && ) , ; == | || }
FOLLOW(I) = { != & && ) , ; < <= == > >= | || }
FOLLOW(I') = { != & && ) , ; < <= == > >= | || }
FOLLOW(J) = { != & && ) + , - ; < <= == > >= | || }
FOLLOW(J') = { != & && ) + , - ; < <= == > >= | || }
FOLLOW(K) = { != & && ) * + , - / ; < <= == > >= | || }
FOLLOW(L) = { && ) , ; | || }
FOLLOW(L') = { && ) , ; | || }
FOLLOW(M) = { & && ) , ; | || }
FOLLOW(M') = { & && ) , ; | || }
FOLLOW(O) = { $ ( ; _Bool id if int struct while } }
FOLLOW(S) = { $ } }
FOLLOW(ST) = { $ ( ; _Bool id if int struct while } }
FOLLOW(ST') = { } }
FOLLOW(T) = { $ ( ; _Bool id if int struct while } }
FOLLOW(W) = { $ ( ; _Bool id if int struct while } }
FOLLOW(assign) = { , ; }
FOLLOW(dec) = { , ; }
FOLLOW(dec') = { ; }
FOLLOW(decs) = { ; }
FOLLOW(expr) = { ) , ; }
FOLLOW(expr') = { ) , ; }
FOLLOW(expr_list) = { ) }
FOLLOW( ) = { != $ & && ( ) + , - ; < <= == > >= _Bool id if int struct while | || } }
```

因为设计的 C 语言文法过于庞大，为了便于展示，采用以下简单文法展示构建预测分析表的测试效果。

S T S'

S' + T S' | \e

T F T'

T' * F T' | \e

F (S) | id

结果如下：

```
-------------------------------------------------------------------------------------------------------
          $       (       )       *       +       F       S       S'      T       T'      id
-------------------------------------------------------------------------------------------------------
F               ( S )                                                                   id
S               T S'                                                                    T S'
S'                                      + T S'
T               F T'                                                                    F T'
T'                              * F T'
-------------------------------------------------------------------------------------------------------
1 1 1 1 1
```

因为空串无法显示，因此在应该产生空串的位置，单独添加了判断语句(详见第五节问题 5)，共 5 处空串，因此最下面一行共输出 5 个 1。

自定义测试样例如下：

```
struct l{
    int (a),b=c+(d|e*!(c!=f>g&h||i));
    _Bool 1j{j;
}i;if(x-y){
    while(p==q){


    };
};else{
    if(a){


    };
    gets(s,a,b);
};
```

报错提示如下：

514 × 696 rams\Language\VSCode\C++\syntax_analyzer.exe

```
LEXICAL ERROR at Ln 4, Col 12! "1j" can't be identified!
Lexical analysis finished!
Please input grammars(ended with "end"):
S T S | O S | C S | W S | id CC S | ST S | ; S | \e
T int decs ; | _Bool decs ;
decs dec dec'
dec' , decs | \e
dec id assign
assign \e | = expr
O ( id ) = expr ;
expr G expr'
expr' && G expr' | \|\| G expr' | \e
G L G'
G' \| L G' | \e
L M L'
L' & M L' | \e
M H M'
M' == H M' | != H M' | \e
H I H'
H' > I H' | < I H' | >= I H' | <= I H' | \e
I J I'
I' + J I' | - J I' | \e
J K J'
J' * K J' | / K J' | \e
K ! K | ( expr ) | id | decimal | hex
C if ( expr ) { S } ; EL
EL \e | else { S } ;
W while ( expr ) { S } ;
CC ( expr_list ) | = expr ;
expr_list expr E
E , expr E | \e
ST struct id { ST' } ;
ST' T ST' | ST ST' | ; ST' | \e
end


Begin getFOLLOW!

Begin to construct parse tree!
SYNTAX ERROR at Ln 3, Col 10! Unexpected symbol "("!
SYNTAX ERROR at Ln 3, Col 12! Unexpected symbol ")"!
SYNTAX ERROR at Ln 4, Col 14! Unexpected symbol "{"!
SYNTAX ERROR at Ln 5, Col 2! Expected a ";", but got a "id"!
Finished constructing parse tree!
Press any key to continue . . .
```

生成的语法分析树见附件“自定义样例.png”。

上述测试结果说明语法分析器可以识别出用 C 语言子集中数据类型和语句集编写的源程序，并以语法分析树的形式输出分析结果。并且可以检查源程序中存在的语法错误，并报告错误所在的位置。以及能够识别函数调用语句，并对源程序中出现的错误进行适当的恢复，使得语法分析可以继续进行。

四、总结

1、过程中所遇到的问题及解决办法

问题 1：打印不出文法存储结构中的文法，从文法存储结构中输出为空。

```
S T S | O S | C S | W S | FC S | ST S | ; S | \e
end
S -> | | | | | |
Press any key to continue . . .
```

解决办法：检查 getGrammars 函数，发现用来暂存产生式体的 vector body 被错误地声明在了遍历输入产生式符号的循环体内。

```
while(iss >> t){
    vector<string> body;
```

将其声明到该循环体外面。

```
vector<string> body;
while(iss >> t){
```

问题解决。

```
E:\programs\Language\VSCode\C++\syntax_analyzer.exe
S T S | O S | C S | W S | FC S | ST S | ; S | \e
end
S -> T S | O S | C S | W S | FC S | ST S | ; S
Press any key to continue . . .
```

问题 2：问题 1 中虽然解决了文法存储失败的问题，但又出现了空字符“\e”存储失败的问题。上图中，输出的文法应以“|”结尾。

解决方法：检查 getGrammars 函数，发现产生式存储逻辑出现问题，将产生式体添加到产生式中的唯一时机是检测到“|”时，这就意味着结尾的产生式体无法被添加到产生式中。

```
if(t=="|"){
    prods.push_back(body);
    body.clear();
}
```

在遍历产生式中符号的循环结束后将结尾的产生式体添加到产生式中即可。

```
E:\programs\Language\VSCode\C++\syntax_analyzer.exe
S T S | O S | C S | W S | FC S | ST S | ; S | \e
end
S -> T S | O S | C S | W S | FC S | ST S | ; S |
Press any key to continue . . .
```

因为空字符对应的字符串只有一个 ASCII 码值为 127 的字符，所以显示不出来，输出的产生式以“|”结尾。

问题 3：获取 FOLLOW 集的具体代码实现问题，例如 A-aB，如果在获取 B 的 FOLLOW 集时先递归调用获取 FOLLOW 集的函数以获取 A 的 FOLLOW 集，则可能出现循环调用，无法退出递归。

解决方法：采用第四节中程序说明部分所描述的方法，先遍历一遍文法获得所有由 FIRST 集产生的 FOLLOW 集部分，以及获取每个非终结符的 FOLLOW 集所依赖的其它非终结符集。然后不断遍历每个 FOLLOW 集依赖于其它非终结符的非终结符，遍历过程中根据依赖关系更新 FOLLOW 集，直到所有 FOLLOW 集都不再发生变化。

问题 4：FOLLOW 集获取算法得到的 FOLLOW 集不正确。仍采用本报告前文中给出的文法进行测试，非终结符 T 的 FOLLOW 集没有包含开始符号 S 的 FOLLOW 集。

```
FOLLOW(C) = { ( ; _Bool id if int struct while }
FOLLOW(FC) = { ( ; _Bool id if int struct while }
FOLLOW(G) = { && || }
FOLLOW(H) = { != == }
FOLLOW(I) = { < <= > >= }
FOLLOW(J) = { + - }
FOLLOW(K) = { * / }
FOLLOW(O) = { ( ; _Bool id if int struct while }
FOLLOW(S) = { $ } }
FOLLOW(ST) = { ( ; _Bool id if int struct while }
FOLLOW(ST') = { } }
FOLLOW(T) = { ( ; _Bool id if int struct while }
FOLLOW(W) = { ( ; _Bool id if int struct while }
FOLLOW(dec) = { , }
FOLLOW(decs) = { ; }
FOLLOW(expr) = { ) , ; }
FOLLOW(expr_list) = { ) }
Press any key to continue . . .
```

解决方法：检查 getFOLLOW 函数发现，第一遍遍历文法时，没能将生成非终结符 FOLLOW 集的依赖关系存储起来。存储该关系的 map 大小为 0。进一步检查发现，实现 getFOLLOW 时，遗漏了非终结符位于产生式体尾端时，应把该非终结符加入到该产生式头非终结符的依赖集中的逻辑。

```
if(si==body->size()-1){
    FD[(*body)[si]].insert(prd->first);
```

```
}
```

补充该逻辑后，问题得到部分解决，存储依赖关系的 map 大小从 0 变成了 14，但 T 的 FOLLOW 集中依然不含 S 的 FOLLOW 集。

进一步检查发现，求 FIRST 集时出现问题，getF 函数未能把产生式

$E \rightarrow$，$expr\ E \mid \varepsilon$ 中的空字符添加到 E 的 FIRST 集中。检查 getF 函数发现，该函数中没有处理参数为空字符串这种特殊情况，为此单独添加处理空串的逻辑如下。这里把空串看作终结符来处理。

```
    if(lct.count(X)||isnull(X)){
        s.insert(X);
        return FIRST[F]=s;
    }
```

修改后问题仍然存在，检查 FD["T"]，发现“T”依赖的非终结符集为空。单步调试发现，存储 FIRST 集映射关系的 map 中，在没有存入空串的映射关系时，该 map 的 count 函数却可以找到空串这个键。鉴于空串比较特殊，对空串特殊处理，getF 中，如果当前符号是空串，则直接在 FIRST 映射中添加空串到空串本身的映射。修改如下：

```
    if(isnull(X)){
        s.insert(X);
        return FIRST[F]=s;
    }
```

修改后问题解决。解决后的结果见前面对 getFOLLOW 函数的测试。

问题 5：使用第四部分测试预测分析表的简单文法样例对 getPPT 函数进行测试时，发现预测分析表结果不对。如下：

```
-----------------------------------------------------------------------------------------
          (      )      *      +      F      S      S'     T      T'     id
-----------------------------------------------------------------------------------------
F         ( S )
S         T S'
S'                              + T S'
T         F T'
T'                     * F T'
-----------------------------------------------------------------------------------------
0 0 0 0 0
Press any key to continue . . .
```

因为空串看不到，所以单独添加语句对空串进行判断：

```
 printf("%d %d %d %d %d\n",
```

```
    ppt[pair<string,string>("S'",")")]
==vector<string>(1,"\177"),
    ppt[pair<string,string>("S'","$")]
==vector<string>(1,"\177"),
    ppt[pair<string,string>("T'","$")]
==vector<string>(1,"\177"),
    ppt[pair<string,string>("T'",")")]
==vector<string>(1,"\177"),
    ppt[pair<string,string>("T'","+")]
==vector<string>(1,"\177"));
```

从图中可以看到，空串没能被成功填入预测分析表中。

解决方法：检查发现，FIRST[“id”]出现错误。

```
FIRST( ( )={ ( }
FIRST( ( S ) )={ ( }
FIRST( * )={ * }
FIRST( * F T’  )={ * }
FIRST( + )={ + }
FIRST( + T S’  )={ + }
FIRST( F )={ ( }
FIRST( F T’  )={ ( }
FIRST( T )={ ( }
FIRST( T S’  )={ ( }
FIRST( id )={ }
FIRST(   )={   }
```

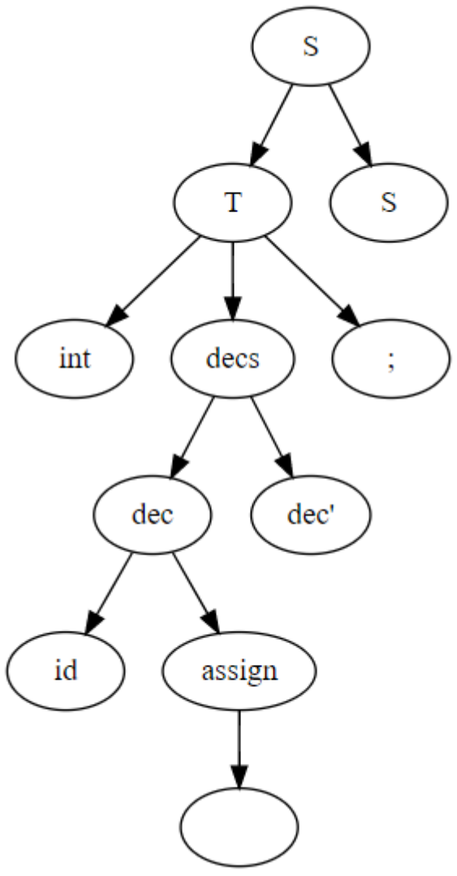
检查 getF，发现错误和之前无法添加\e 一样，FIRST 集中莫名其妙地多出了要添加的键。导致 count 条件成立，直接退出了。直接将 count 判断条件后移，使得对于终结符总是向 FIRST 中加入对应的键值。修改后问题得到部分解决：

```
-------------------------------------------------------------------------------------------
          $      (      )      *      +      F      S      S’     T      T’     id
-------------------------------------------------------------------------------------------
F             ( S )                                                          id
S             T S’                                                           T S’
S’                                  + T S’
T             F T’                                                           F T’
T’                          * F T’
-------------------------------------------------------------------------------------------
0 0 0 0 0
```

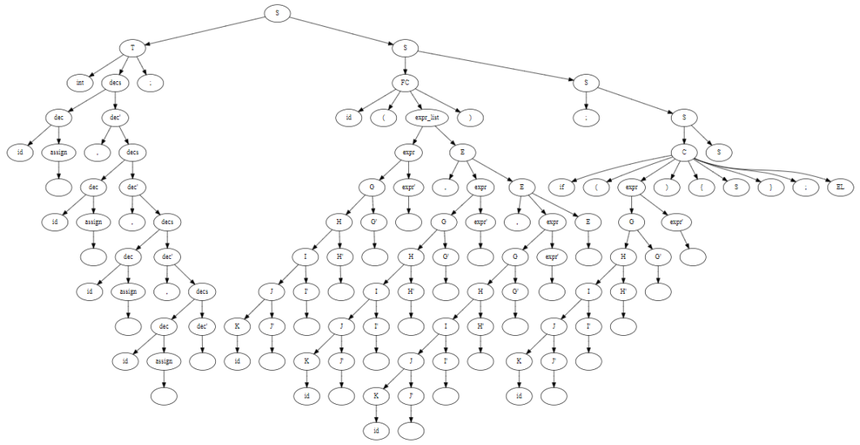
但空串仍未被添加到预测表中。经检查，发现输入的字符“’”是中文编码，即在 word 中直接复制粘贴到命令行中，“’”变成了中文编码，使得带该字符的终结符与 printf 中输入的英文编码版的对不上。手动从命令行中输入英文编码的文法，问题解决。解决后的效果见第四部分中的测试。

问题 6：LL1 生成的树不对，出现预测分析表中找不到可以继续执行下去的产生式的问题。

解决方法：经检查发现，遍历输入符号时，错误地使用了输入结构中的值 val，要与文法中的终结符相匹配，应输入 type，如输入流中的标识符 a，其值 val 为 a，type 为 id。修改后问题部分解决。



生成了一部分语法分析树，又卡在了分析表中找不到能用的产生式这一错误上。继续检查发现出错的位置，当前栈顶非终结符为空串，即之前错误地把产生式体中的空串压到了栈中。压栈时添加检查空串的逻辑，过滤掉空串。修改后陷入死循环，不断往分析树上添加表示空串的空节点。检查发现栈顶非终结符找到推导的产生式后，忘记弹出栈顶非终结符。修改后问题解决。该问题解决后生成的语法分析树如下：

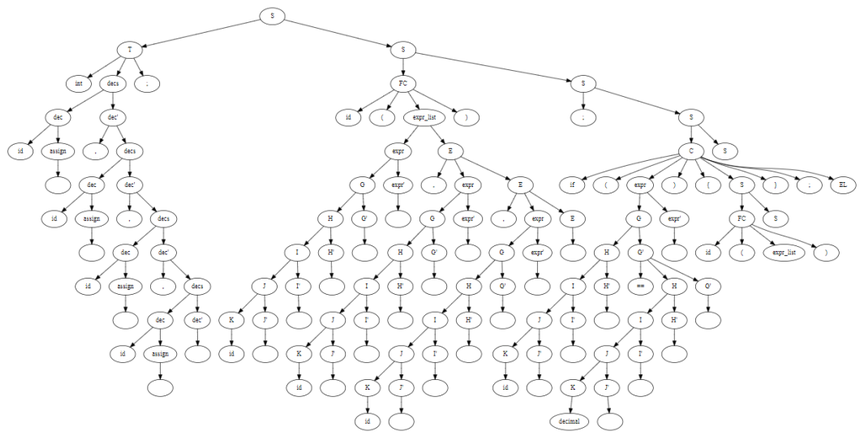


问题 7：问题 6 解决后，绘制出的语法分析树仍然不完全正确，出现了栈中终结符与输入流中终结符不一致的问题。

解决方法：可以看到，问题 6 的分析树中，if(op==0)中的条件的分析有误，没有“==”这一节点。进一步检查发现文法设计有误，FOLLOW(G')和 FIRST(G')有公共符号“==”，不满足 LL(1)文法。

将文法中$K \rightarrow !\ K \mid (expr) \mid id \mid decimal \mid hex$ 修改为

$K \rightarrow !\ K \mid (expr) \mid id \mid decimal \mid hex$，情况得到改善，但仍有此类问题。修改后生成的分析树：



这时 if(op==0)已经得到正确分析，但是后面 ans = num1 + num2;这一句分析错误，错误地选用了函数调用的推导。检查发现，文法设计存在纰漏，开始符直接推出赋值语句和直接推出函数调用语句的产生式存在左公因子。

修改文法，移除非终结符 FC，添加非终结符 CC，将

$S \rightarrow T\ S \mid O\ S \mid C\ S \mid W\ S \mid FC\ S \mid ST\ S \mid ;\ S \mid \varepsilon$

修改为

$S \rightarrow T\ S \mid O\ S \mid C\ S \mid W\ S \mid id\ CC\ S \mid ST\ S \mid ;\ S \mid \varepsilon$

将$O \rightarrow id = expr; \mid (id) = expr;$

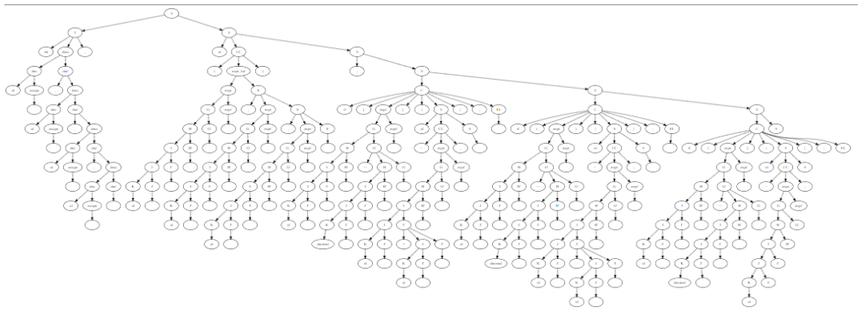
修改为

$O \rightarrow (id) = expr;$

将$FC \rightarrow id\ (\ expr_list)$

修改为$CC \rightarrow (\ expr_list) \mid = expr\ ;$

修改后问题得到改善，分析树变为：



这次的问题是文法中没有设计按位与“&”和按位或“|”这两种运算。补充这两种运算后，问题得到完全解决。