实验四. 语义分析及中间代码生成

设计思想

根据对属性文法及语义分析、中间代码生成的学习，可以将实验二、三的两种语法分析器进行一定的改造，以达到进行语法分析的同时进行语义分析并生成中间代码。根据PL0文法的特点以及尝试进行一次语法分析完成语义分析并产生对应的中间代码，本实验对实验三自下而上语法分析进行改造，添加一定的属性文法，实现对表达式的分析，对于算术表达式给出分析后的值，对于一般的表达式给出最后生成的四元式中间代码。

本实验对PL0文法的表达式文法进行设计自下而上语法分析，表达式巴斯克范式如下：

文法的初始化

$$
\begin{aligned}
&< \text{表达式} >::= [+|-] < \text{项} > \{< \text{加法运算符} >< \text{项} >\}\\
&< \text{项} >::=< \text{因子} > \{< \text{乘法运算符} >< \text{因子} >\}\\
&< \text{因子} >::=< \text{标识符} > | < \text{无符号整数} > |'('< \text{表达式} >')'\\
&< \text{加法运算符} >::= +|-\\
&< \text{乘法运算符} >::= *|/
\end{aligned}
$$

对以上文字描述的文法可以给出对应的英文表示，以及对应的非终结符的 $First$集合 和 $Follow$集合：

$$
\begin{aligned}
&< Expression >::= [+|-] < Term > \{< Addop >< Term >\}\\
&< Term >::=< Factor > \{< Mulop >< Factor >\}\\
&< Factor >::=< Ident > | < UnsignInt > |(< Expression >)\\
&< Addop >::= +|-\\
&< Mulop >::= *|/
\end{aligned}
$$

该文法的非终结符结合以及终结符集合如下：

$$
\begin{aligned}
V_T &= \{+, -, *, /, (,), Ident, UnsignInt\}\\
V_N &= \{Expression, Term, Addop, Factor, Mulop\}
\end{aligned}
$$

将其简化一下文法：

$$
\begin{aligned}
&E \to T|+T|-T\\
&E \to E+T|E-T\\
&T \to F\\
&T \to T*F|T/F\\
&F \to i|u|(E)
\end{aligned}
$$

对应的终结符和非终结符集合如下：

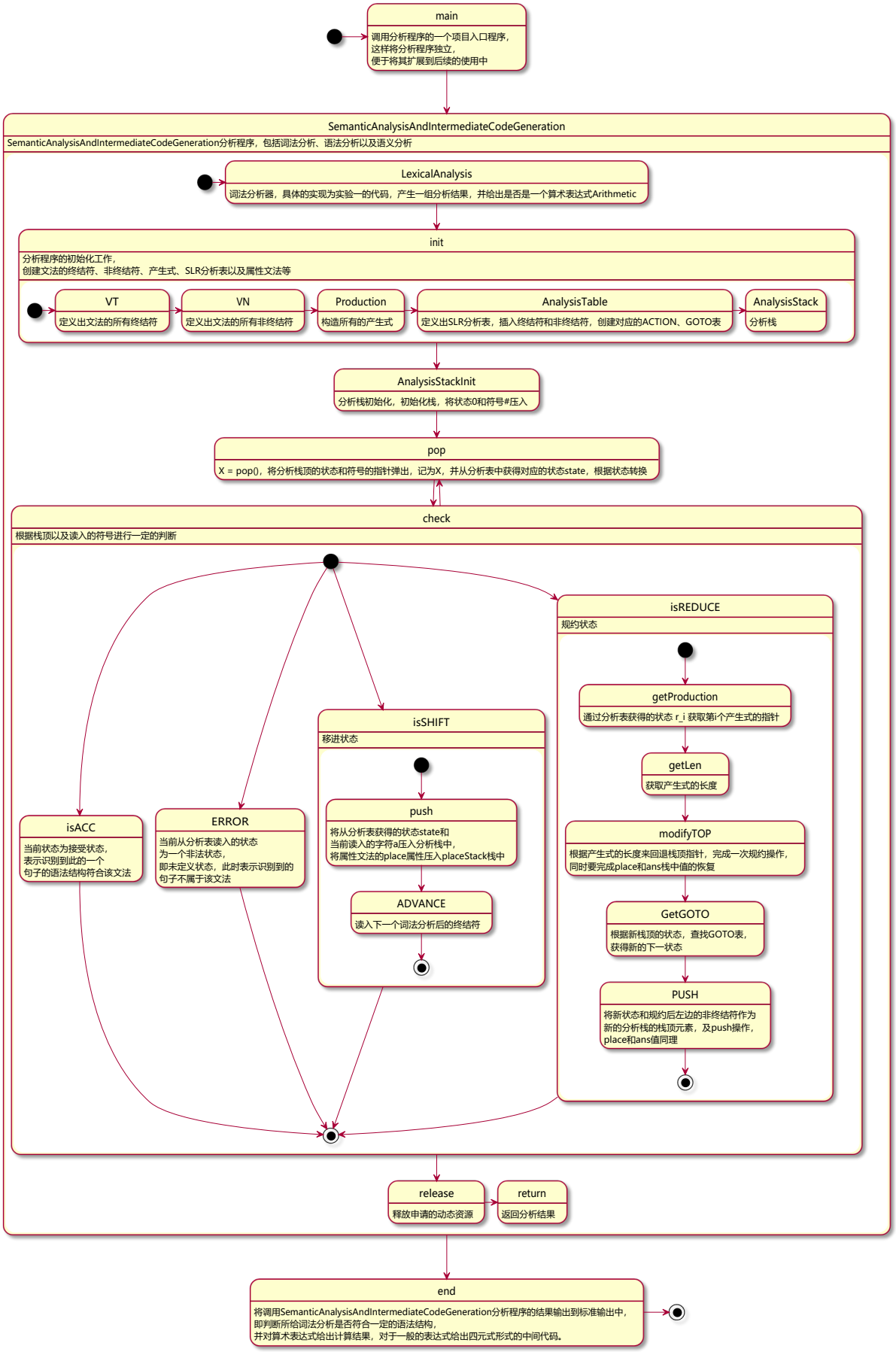
$$
\begin{aligned}
V_T &= \{+, -, *, /, (,), i, u\}\\
V_N &= \{S', E, T, F\}
\end{aligned}
$$

拓广文法

文法的拓广，将文法 $G(S)$ 拓广为 $G'(S')$，并规定每一个产生式的编号，以便后续的方便使用，同时给出每个产生式的属性文法：

$$
\begin{array}{rlll}
0: & S' \to & E & \{if\ Arithmetic = true\ then\ S'.ans = E.ans; S'.place = newtemp; emit(S'\ ' :=' E.place\}\\
1: & E \to & T & \{if\ Arithmetic = true\ then\ E.ans = T.ans; E.place = newtemp; emit(E.place' :=' T.place\}\\
2: & E \to & +T & \{if\ Arithmetic = true\ then\ E.ans = T.ans; E.place = newtemp; emit(E.place' :=' T.place\}\\
3: & E \to & -T & \{if\ Arithmetic = true\ then\ E.ans = -T.ans; E.place = newtemp; emit(E.place' :=' \ 'uminus' T.place\}\\
4: & E \to & E_1 + T & \{if\ Arithmetic = true\ then\ E.ans = E_1.ans + T.ans; E.place = newtemp; emit(E.place' :=' E_1.place' +' T.place\}\\
5: & E \to & E_1 - T & \{if\ Arithmetic = true\ then\ E.ans = E_1.ans - T.ans; E.place = newtemp; emit(E.place' :=' E_1.place' -' T.place\}\\
6: & T \to & F & \{if\ Arithmetic = true\ then\ T.ans = F.ans; T.place = newtemp; emit(T.place' :=' F.place\}\\
7: & T \to & T_1 * F & \{if\ Arithmetic = true\ then\ T.ans = T_1.ans * F.ans; T.place = newtemp; emit(T.place' :=' T_1.place' *' F.place\}\\
8: & T \to & T_1/F & \{if\ Arithmetic = true\ then\ T.ans = T_1.ans/F.ans; T.place = newtemp; emit(T.place' :=' T_1.place'/'F.place\}\\
9: & F \to & i & \{F.place = newtemp; emit(F.place' :=' i.place\}\\
10: & F \to & u & \{if\ Arithmetic = true\ then\ F.ans = u; F.place = newtemp; emit(F.place' :=' u.place\}\\
11: & F \to & (E) & \{if\ Arithmetic = true\ then\ F.ans = E.ans; F.place = newtemp; emit(F.place' :=' E.place\}
\end{array}
$$

算法流程

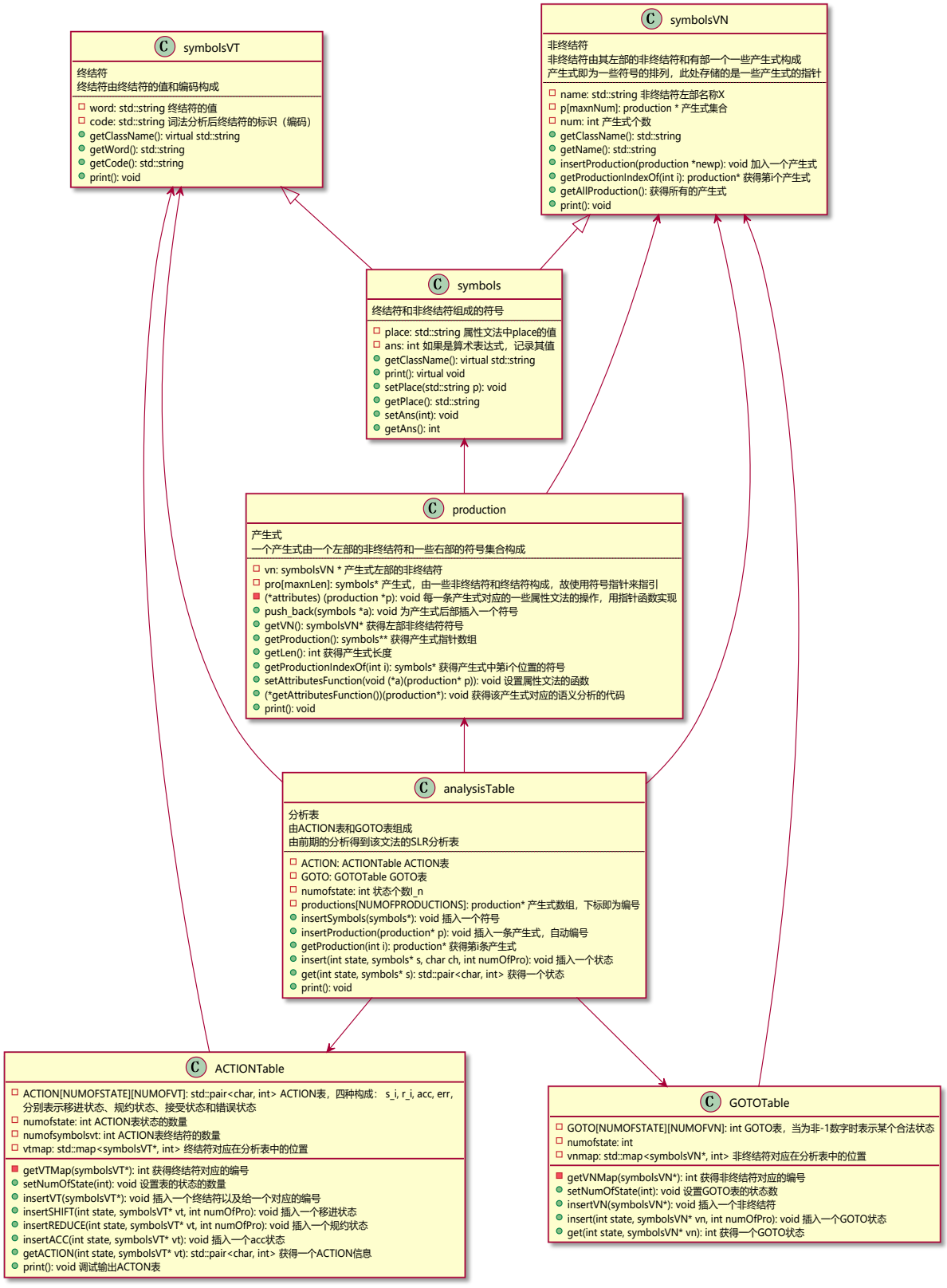


源程序

整个分析程序是由上一实验的SLR分析程序以及实验一的词法分析程序修改而来，根据语义分析的所需，简单的修改了非终结符类的组成，对非终结符类增加一对应的属性文法，例如place、ans等等属性；对于其他的一些操作，如中间代码emit的生成函数以及计算表达式的值等操作，由产生式类的对应一系列函数*attributes构成，为了实现每一个产生式对象有一个相同的调用属性文法函数的入口，这里在产生式类中添加一个函数指针 `void (*attributes)(production *p)` ，这样就可以在对应的某个产生式对象的实现中，根据所需构建一个属性文法的函数，然后设置函数指针指向即可，这样做的好处可以使得产生式类简单易于实现，更重要的是在后面的总控程序中，可以不在添加大量的产生式判断的代码就能实现语义分析和语法分析两个操作的结合，充分的实现模块化编程。

最后将原来的分析总控程序中的具体的执行流程根据中添加语义分析的过程，也就是在每一次语法分析的规约操作中，添加对改规约产生式的属性文法函数的调用即可，因为在具体分析中，分析栈中一定会出现多个同样的非终结符，而后压入栈中的非终结符会将前面的属性文法的某些值覆盖，导致语义分析结果出错，故添同分析栈一同添加placeStack和ansStack，保存分析过程中的值，在规约前恢复这一段产生式非终结符的属性文法的值，其他内容保持不变即可：

整个分析程序所设计到的类以及相互的关系如下：



项目地址

源程序

symbols.h

```
// symbols.h
#ifndef symbols_h
#define symbols_h
#include<iostream>
/*
终结符和非终结符的一个共同的基类
这样可以通过基类来引用终结符和非终结符

*/
class symbols
{
private:
    /* data */
    // 属性文法的内容

    std::string place;  // 属性文法中place的值
    int ans;                    // 如果是算术表达式，记录其值
public:
    symbols(){};
    virtual ~symbols(){};
    virtual std::string getClassName(){
        return "symbols";
    }
    void print(){};
    void setPlace(std::string p){       设置place的值
        place = p;
    }
    void setAns(int a){
        ans = a;
    }
    std::string getPlace(){
        return place;
    }
    int getAns(){
        return ans;
    }
};

#endif /*symbols_h*/
```

symbolsVN.h

```
// symbolsVN.h
#ifndef symbolsVN_h
#define symbolsVN_h
#include<iostream>
#include"symbols.h"
#include"production.h"
const int maxnNum = 5;     // 一个终结符所有的产生式的数量
class production;
/*
非终结符
非终结符由其左部的非终结符和有部一个一些产生式构成
产生式即为一些符号的排列，此处存储的是一些产生式的指针
*/

class symbolsVN: public symbols{
private:
    std::string name;                                           // 非终结符左部名称X
    production *p[maxnNum];                                     // 产生式集合
    int num;
public:
    symbolsVN();
    symbolsVN(std::string);
    ~symbolsVN() {}
    std::string getClassName();
    std::string getName();
    void insertProduction(production *newp);    // 加入一个产生式
    production* getProductionIndexOf(int i);    // 获得第i个产生式
    production** getAllProduction();                    // 获得所有的产生式
    void print();
};

#endif /*symbolsVN_h*/
```

symbolsVN.cpp

```cpp
// symbolsVN.cpp

#include<iostream>
#include"symbolsVN.h"

symbolsVN::symbolsVN(){
    num = 0;
}
symbolsVN::symbolsVN(std::string n):name(n){
    symbolsVN();
    std::cerr << "V_N: " << name << " created..." << std::endl;
}

std::string symbolsVN::getClassName(){
    return "VN";
}
std::string symbolsVN::getName(){
    return name;
}
void symbolsVN::insertProduction(production *newp){
    p[num++] = newp;
    return;
}
production* symbolsVN::getProductionIndexOf(int i){
    if(i >= num){
        std::cerr << "index overflow..." << std::endl;
        return NULL;
    }
    return p[i];
}
production** symbolsVN::getAllProduction(){
    return p;
}
void symbolsVN::print(){
    std::cerr << "VN: " << name << std::endl;
    std::cerr << "ALL production: " << std::endl;
    for(int i = 0; i < num; ++i){
        std::cerr << name << " \\to ";
        p[i]->print();
    }
    std::cerr << std::endl;
}
```

symbolsVT.h

```cpp
// symbolsVT.h
#ifndef symbolsVT_h
#define symbolsVT_h
#include<iostream>
#include"symbols.h"
/*
一个终结符
终结符由终结符的值和编码构成

*/
class symbolsVT : public symbols{
private:
    std::string word;   // 终结符的值
    std::string code;   // 词法分析后终结符的标识（编码）
    std::string var;    // 终结符实际的值
public:
    symbolsVT(){}
    symbolsVT(std::string W, std::string C):word(W), code(C) {
        std::cerr<< "V_T: " << word << " " << code << " created..." << std::endl;
    }
    ~symbolsVT() {}
    std::string getClassName();
    std::string getWord();
    std::string getCode();
    void setVar(std::string v);
    std::string getVar();
    void print();
};

#endif /*symbolsVT_h*/
```

symbolsVT.cpp

```cpp
// symbolsVT.cpp
#include<iostream>
#include"symbolsVT.h"

std::string symbolsVT::getClassName(){
    return "VT";
}
std::string symbolsVT::getWord(){
    return word;
}
std::string symbolsVT::getCode(){
    return code;
}
void symbolsVT::setVar(std::string v){
    var = v;
}
std::string symbolsVT::getVar(){
    return var;
}
void symbolsVT::print(){
    std::cerr << "VT: " << word << " " << code << std::endl;
}
```

production.h

```cpp
// production.h
#ifndef production_h
#define production_h
#include<iostream>
#include"symbols.h"
#include"symbolsVT.h"
#include"symbolsVN.h"
/*
产生式
一个产生式由一个左部的非终结符和一些右部的符号集合构成
*/

const int maxnLen = 10;        // 一个产生式的右部符号的数量
class symbolsVN;
class production
{
private:
    symbolsVN *vn;                                  // 产生式左部的非终结符
    symbols *pro[maxnLen];                      // 产生式，由一些非终结符和终结符构成，故使用符号指针来指引
    int len;

    // 属性文法的一些内容
    void (*attributes) (production *p); // 每一条产生式对应的一些属性文法的操作，用指针函数实现
public:
    production();
    production(symbolsVN *v);
    ~production(){}
    void push_back(symbols *a);         // 为产生式后部插入一个符号
    symbolsVN* getVN();                         // 获得左部非终结符符号
    symbols** getProduction();          // 获得产生式指针数组
    int getLen();                                   // 获得产生式长度
    symbols* getProductionIndexOf(int i); // 获得产生式中第i个位置的符号
    void setAttributesFunction(void (*a)(production* p)){ // 设置属性文法的函数
        attributes = a;
    }
    void (*getAttributesFunction())(production*){ // 获得该产生式对应的语义分析的代码
        return attributes;
    }
    void print();
};

#endif /*production_h*/
```

production.cpp

```cpp
// production.cpp

#include<iostream>
#include"production.h"
#include"symbolsVT.h"
#include"symbolsVN.h"

production::production(){
    len = 0;
}
production::production(symbolsVN *v){
    vn = v;
    production();
    std::cerr << "A production of " << vn->getName() << " has created..." << std::endl;
}
void production::push_back(symbols *a){
    pro[len++] = a;
}
symbolsVN* production::getVN(){
    return vn;
}
symbols** production::getProduction(){
    return pro;
}
symbols* production::getProductionIndexOf(int i){
    if(i >= len){
        std::cerr << "index Overflow..." << std::endl;
        return NULL;
    }
    return pro[i];
}
int production::getLen(){
    return len;
}
void production::print(){
    std::cerr << vn->getName() << "->";
    for(int i = 0; i < len; ++i){
        if(pro[i]->getClassName() == "VT"){
            std::cerr << ((symbolsVT*)pro[i])->getWord();
        }
        else{
            std::cerr << ((symbolsVN*)pro[i])->getName();
        }
    }
    std::cerr << std::endl;
}
```

analysisTable.h

```
// analysisTable.h
#ifndef analysisTable_h
#define analysisTable_h
#include<map>
#include"symbols.h"
#include"symbolsVN.h"
#include"symbolsVT.h"
#include"production.h"
const int NUMOFVT = 20;
const int NUMOFVN = 20;
const int NUMOFSTATE = 30;
const int NUMOFPRODUCTIONS = 30;
/*
分析表子程序
由前期的分析得到该文法的分析表，由ACTION表和GOTO表构成
*/
class ACTIONTable{
private:
    std::pair<char, int> ACTION[NUMOFSTATE][NUMOFVT];
    int numofstate;                                          // ACTION表状态的数量
    int numofsymbolsvt;                                      // ACTION表终结符的数量
    std::map<symbolsVT*, int> vtmap;    // 终结符对应在分析表中的位置
    int getVTMap(symbolsVT*);                   // 获得终结符对应的编号
public:
    ACTIONTable();
    ACTIONTable(int);
    ~ACTIONTable(){}
    void setNumOfState(int);                    // GOTO状态数量
    void insertVT(symbolsVT*);                  // 插入一个终结符以及给一个对应的编号
    void insertSHIFT(int state, symbolsVT* vt, int numOfPro);       // 插入一个移进状态
    void insertREDUCE(int state, symbolsVT* vt, int numOfPro);      // 插入一个规约状态
    void insertACC(int state, symbolsVT* vt);                       // 插入一个acc状态
    std::pair<char, int> getACTION(int state, symbolsVT* vt);       // 获得一个ACTION信息
    void print();
};
class GOTOTable{
private:
    int GOTO[NUMOFSTATE][NUMOFVN];
    int numofstate;                                          // GOTO状态数量
    int numofsymbolsvn;
    std::map<symbolsVN*, int> vnmap;    // 非终结符对应在分析表中的位置
    int getVNMap(symbolsVN*);                   // 获得非终结符对应的编号
public:
    GOTOTable();
    GOTOTable(int);
    ~GOTOTable(){}
    void setNumOfState(int);                    // 设置GOTO表的状态数
    void insertVN(symbolsVN*);          // 插入一个非终结符
    void insert(int state, symbolsVN* vn, int numOfPro);    // 插入一个GOTO状态
    int get(int state, symbolsVN* vn);                      // 获得一个GOTO状态
    void print();
};
class analysisTable
{
private:
    ACTIONTable ACTION;                 // ACTION表
    GOTOTable GOTO;                     // GOTO表
    int numofstate;                     // 状态个数I_n
    int numofpro;                       // 产生式数量
    production* productions[NUMOFPRODUCTIONS];  // 产生式数组，下标即为编号
public:
    analysisTable(int ns);
    // analysisTable(int, int, int);
    ~analysisTable() {}
    void insertSymbols(symbols*);               // 插入一个符号
    void insertProduction(production* p);       // 插入一条产生式，自动编号
    production* getProduction(int i);   // 获得第i条产生式
    void insert(int state, symbols* s, char ch, int numOfPro);  // 插入一个状态
    std::pair<char, int> get(int state, symbols* s);    // 获得一个状态
    void print();
};

#endif /*analysisTable_h*/
```

analysisTable.cpp

```cpp
// analysisTable.cpp
#include<iostream>
#include<algorithm>
#include<string.h>
#include"analysisTable.h"

ACTIONTable::ACTIONTable(){
    numofsymbolsvt = 0;
    vtmap.clear();
    std::pair<char, int> init = std::make_pair('e', -1);
    // fill(begin(ACTION), end(ACTION), std::make_pair('e', -1));
    for(int i = 0; i < NUMOFSTATE; ++i)
        for(int j = 0; j < NUMOFVT; ++j)
            ACTION[i][j] = init;
    std::cerr << "ACTIONTable has created..." << std::endl;
}
void ACTIONTable::setNumOfState(int ns){
    numofstate = ns;
}
int ACTIONTable::getVTMap(symbolsVT* vt){
    if(vtmap.find(vt) != vtmap.end())return vtmap[vt];
    return -1;
}
void ACTIONTable::insertVT(symbolsVT* vt){
    vtmap[vt] = numofsymbolsvt++;
}
void ACTIONTable::insertSHIFT(int state, symbolsVT* vt, int numOfPro){
    int nvt = getVTMap(vt);
    if(state < numofstate && ~nvt){
        ACTION[state][nvt] = std::make_pair('s', numOfPro);
    }
}
void ACTIONTable::insertREDUCE(int state, symbolsVT* vt, int numOfPro){
    int nvt = getVTMap(vt);
    if(state < numofstate && ~nvt){
        ACTION[state][nvt] = std::make_pair('r', numOfPro);
    }
}
void ACTIONTable::insertACC(int state, symbolsVT* vt){
    int nvt = getVTMap(vt);
    if(state < numofstate && ~nvt){
        ACTION[state][nvt] = std::make_pair('a', 0x3f3f3f3f);
    }
}
std::pair<char, int> ACTIONTable::getACTION(int state, symbolsVT* vt){
    int nvt = getVTMap(vt);
    if(state < numofstate && ~nvt){
        return ACTION[state][nvt];
    }
    return std::make_pair('e', -1);
}
void ACTIONTable::print(){
    std::cerr << "ACTION:" << std::endl;
    std::cerr << numofsymbolsvt << std::endl;
    std::cerr << "\t";
    // for(auto i: vtmap)std::cerr << i.first->getWord() << "\t";
    for(std::map<symbolsVT*, int>::iterator i = vtmap.begin(); i != vtmap.end(); ++i)std::cerr << i->first->getWord() << "\t";
    std::cerr << std::endl;
    for(int i = 0; i < numofstate; ++i){
        std::cerr << i << ": \t";
        for(int j = 0; j < numofsymbolsvt; ++j){
            if(~ACTION[i][j].second)
                std::cerr << ACTION[i][j].first << ACTION[i][j].second << " \t";
            else
                std::cerr << "   \t";
        }
        std::cerr << std::endl;
    }
    std::cerr << std::endl;
}


GOTOTable::GOTOTable(){
    numofsymbolsvn = 0;
    vnmap.clear();
    memset(GOTO, -1, sizeof GOTO);
    std::cerr << "GOTOTable has created..." << std::endl;
}
void GOTOTable::setNumOfState(int ns){
    numofstate = ns;
}
void GOTOTable::insertVN(symbolsVN* vn){
    vnmap[vn] = numofsymbolsvn++;
}
int GOTOTable::getVNMap(symbolsVN* vn){
    if(vnmap.find(vn) != vnmap.end())return vnmap[vn];
    return -1;
}
void GOTOTable::insert(int state, symbolsVN* vn, int numOfPro){
    int nvn = getVNMap(vn);
    if(state < numofstate && ~nvn){
```

```
        GOTO[state][nvn] = numOfPro;
    }
}
int GOTOTable::get(int state, symbolsVN* vn){
    int nvn = getVNMap(vn);
    if(state < numofstate && ~nvn){
        return GOTO[state][nvn];
    }
    return -1;
}
void GOTOTable::print(){
    std::cerr << "GOTO:" << std::endl;
    std::cerr << numofsymbolsvn << std::endl;
    std::cerr << "\t";
    // for(auto i: vnmap)std::cerr << i.first->getName() << "\t";
    for(std::map<symbolsVN*, int>::iterator i = vnmap.begin(); i != vnmap.end(); ++i)std::cerr << i->first->getName() << "\t";
    std::cerr << std::endl;
    for(int i = 0; i < numofstate; ++i){
        std::cerr << i << ": \t";
        for(int j = 0; j < numofsymbolsvn; ++j){
            if(~GOTO[i][j])
                std::cerr << GOTO[i][j] << "\t";
            else
                std::cerr << "   \t";
        }
        std::cerr << std::endl;
    }
    std::cerr << std::endl;
}


analysisTable::analysisTable(int ns):numofstate(ns){
    ACTION.setNumOfState(numofstate);
    GOTO.setNumOfState(numofstate);
    numofpro = 0;
    std::cerr << "An AnalysisTable has created..." << std::endl;
}
void analysisTable::insertSymbols(symbols* s){
    if(s->getClassName() == "VT"){
        ACTION.insertVT((symbolsVT*)(s));
    }
    else if(s->getClassName() == "VN"){
        GOTO.insertVN((symbolsVN*)(s));
    }
}
void analysisTable::insertProduction(production* p){
    productions[numofpro++] = p;
}
production* analysisTable::getProduction(int i){
    if(i < numofpro)return productions[i];
    // return nullptr;
    return NULL;
}
void analysisTable::insert(int state, symbols* s, char ch, int numOfPro){
    if(s->getClassName() == "VT"){
        if(ch == 'a'){
            ACTION.insertACC(state, (symbolsVT*)(s));
        }
        else if(ch == 's'){
            ACTION.insertSHIFT(state, (symbolsVT*)(s), numOfPro);
        }
        else if(ch == 'r'){
            ACTION.insertREDUCE(state, (symbolsVT*)(s), numOfPro);
        }
    }
    else if(s->getClassName() == "VN"){
        GOTO.insert(state, (symbolsVN*)(s), numOfPro);
    }
}
std::pair<char, int> analysisTable::get(int state, symbols* s){
    if(s->getClassName() == "VT"){
        return ACTION.getACTION(state, (symbolsVT*)(s));
    }
    else if(s->getClassName() == "VN"){
        return std::make_pair('g', GOTO.get(state, (symbolsVN*)(s)));
    }
    return std::make_pair('e', -1);
}
void analysisTable::print(){
    std::cerr << "analysisTable: " << std::endl;
    ACTION.print();
    GOTO.print();
    std::cerr << std::endl;
}
```

LexicalAnalysi.cpp

```cpp
// LexicalAnalysi.cpp
#include<iostream>
#include<cstdio>
#include<map>
#include<vector>
#include<string.h>
const int MAXNWORDLEN = 1 << 7;
const int MAXMSTATENUM = 1e2 + 5;
//词汇表
const std::string words[] = {"begin", "call", "const", "do", "end", "if", "odd", "procedure", "read", "then", "var", "while", "write",
                        "+", "-", "*", "/", "=", "<>", "<", "<=", ">", ">=", ":=",
                        "(", ")", ",", ";", "."};
//对应的助记符
const std::string codes[] = {"beginsym", "callsym", "constsym", "dosym", "endsym", "ifsym", "oddsym", "proceduresym", "readsym", "thensym", "varsym", "whilesym
                        "plus", "minus", "times", "slash", "eql", "neq", "lss", "leq", "gtr", "geq", "becomes",
                        "lparen", "rparen", "comma", "semicolon", "period"};

char CH;                                                                    //当前读入的字符
char strToken[MAXNWORDLEN];                                                 //当前读入的单词串
int strTokenLen;                                                            //单词串的长度

std::map<std::string, int> symbolTable;                                     //符号表，此处因为没有输出该项，所以没有使用
std::map<std::string, int> constTable;                                      //常数表，此处因为没有输出该项，所以没有使用
std::map<std::string, std::pair<int, std::string> > reserveWords;           //单词和对应助记符的一个映射表，当发现单词时，将strToken中保存的单词在词汇表查询，存在即返回

int stateTrans[MAXMSTATENUM][MAXNWORDLEN];                                  //状态转换矩阵，stateTrans[i, j]表示当前在状态i，读入字符为j时下一个状态的
int isAcceptingState[MAXMSTATENUM];                                         //是否是终态数组，isAcceptingState[i]表示状态i是否是终态
int curState;                                                               //当前的状态编号

bool Arithmetic;                                // 是否为算术表达式
std::vector<std::pair<std::string, std::string> > ans;                      //最后分析的结果

bool IsLetter(){                                                            //判断一个字符是否为字母
    if((CH >= 'a' && CH <'z') || (CH >= 'A') && (CH <= 'Z'))return true;
    return false;
}
bool IsDigital(){                                                           //判断数字
    if(CH >= '0' && CH <= '9')return true;
    return false;
}
bool IsBlank(){                                                             //判断是否为空白字符
    if(CH == ' ' || CH == '\n' || CH == '\r' || CH == '\t')return true;
    return false;
}
bool FileEnd;                                                               //是否读到文件末
void GetChar(){                                                             //读入一个字符到ch，当读到文件末是scanf返回-1，此时FileEnd的值就为假False
    FileEnd = ~scanf("%c", &CH);
    // CH = getchar();
}
void GetBC(){                                                               //跳过空白符
    GetChar();
    while(FileEnd && IsBlank())GetChar();
}
void Concat(){                                                              //将ch加入到strToken中
    if(strTokenLen < MAXNWORDLEN)strToken[strTokenLen++] = CH;
}
std::pair<int, std::string> Reserve(){                                      //根据词汇表的映射返回当前识别到的单词的助记符
    std::string s = std::string(strToken);
    if(reserveWords.count(s))return reserveWords[s];
    else if(curState == 2)return std::make_pair(0, "ident");
    else return std::make_pair(0, "number");
    return std::make_pair(0, "");
}
void pushAns(){                                                             //增加结果二元组
    std::string res = Reserve().second;
    if(res == "ident")Arithmetic = false;
    ans.push_back(make_pair(std::string(strToken), res));
    // ans.push_back(make_pair(std::string(strToken), Reserve().second));
}
void InsertId(){                                                            //将识别到的单词插入符号表（此程序未使用）
    symbolTable[std::string(strToken)] = symbolTable.size() + 1;
}
void InsertConst(){                                                         //将识别到的常数插入常数表（此程序未使用）
    constTable[std::string(strToken)] = constTable.size() + 1;
}

void LexicalAnalysiInit(){                                                  //初始化函数，除了各变量的置空初始化外，根据不同文法的DFA初始化状态
    Arithmetic = true;
    FileEnd = true;
    CH = ' ';
    symbolTable.clear();
    constTable.clear();
    reserveWords.clear();
    memset(strToken, '\0', sizeof strToken);
    strTokenLen = 0;

    //构造单词表中预定义单词于编码的映射关系
    int len = sizeof(words) / sizeof(words[0]);
    for(int i = 0; i < len; ++i)reserveWords[words[i]] = make_pair(i, codes[i]);

    //构造状态转化矩阵
```

```
    //-1 表示未定义状态，即出错
    //inf表示终态，表示识别到一个单词（使用isaccepting来表示
    memset(stateTrans, -1, sizeof stateTrans);
    //对于状态0，读入空白仍为该状态，字母进入状态1，数字进入状态2等等
    stateTrans[0][' '] = stateTrans[0]['\n'] = stateTrans[0]['\r'] = stateTrans[0]['t'] = 0;
    for(int i = 'a'; i <= 'z'; ++i)stateTrans[0][i] = 1;
    for(int i = 'A'; i <= 'Z'; ++i)stateTrans[0][i] = 1;
    for(int i = '0'; i <= '9'; ++i)stateTrans[0][i] = 3;
    stateTrans[0]['+'] = 5;
    stateTrans[0]['-'] = 6;
    stateTrans[0]['*'] = 7;
    stateTrans[0]['/'] = 8;
    stateTrans[0]['='] = 9;
    stateTrans[0]['<'] = 10;
    stateTrans[0]['>'] = 14;
    stateTrans[0][':'] = 17;
    stateTrans[0]['('] = 19;
    stateTrans[0][')'] = 20;
    stateTrans[0][','] = 21;
    stateTrans[0][';'] = 22;
    stateTrans[0]['.'] = 23;

    //对其他状态定义：
    //1:
    for(int i = 0; i < MAXNWORDLEN; ++i)stateTrans[1][i] = 2;
    for(int i = 'a'; i <= 'z'; ++i)stateTrans[1][i] = 1;
    for(int i = 'A'; i <= 'Z'; ++i)stateTrans[1][i] = 1;
    for(int i = '0'; i <= '9'; ++i)stateTrans[1][i] = 1;

    //3:
    for(int i = 0; i < MAXNWORDLEN; ++i)stateTrans[3][i] = 4;
    for(int i = '0'; i <= '9'; ++i)stateTrans[3][i] = 3;

    //10:
    std::fill(stateTrans[10], stateTrans[10] + MAXNWORDLEN, 13);
    stateTrans[10]['>'] = 11;
    stateTrans[10]['='] = 12;

    //14:
    std::fill(stateTrans[14], stateTrans[14] + MAXNWORDLEN, 13);
    stateTrans[14]['='] = 15;

    //17:
    stateTrans[17]['='] = 18;

    //确定终态：
        //0:表示非终态
        //1:表示根据当前读入的字符拼接到strToken后即为一个单词（显然这样下一次单词分析需要再读入新字符）
        //2:表示根据当前读入字符可以判断出strToken中为一个单词（显然此时读入的字符要归入到下一次单词分析）
    std::fill(isAcceptingState, isAcceptingState + MAXMSTATENUM, 1);
    isAcceptingState[0] = isAcceptingState[1] = isAcceptingState[3] = isAcceptingState[10] = isAcceptingState[14] = isAcceptingState[17] = 0;
    isAcceptingState[2] = isAcceptingState[4] = isAcceptingState[13] = isAcceptingState[16] = 2;

    ans.clear();
}


void work(){                                                                    //词法分析一般控制过程
    curState = 0;
    GetBC();
    while(~stateTrans[curState][CH]){                   //当当前的状态合法时进行分析
        if(!FileEnd)CH = '\0';                                  //如果是读到文件末，对最后遗留在strToken进行分析后退出子程序
        curState = stateTrans[curState][CH];    //根据当前状态和读入字符进行状态转移
        if(isAcceptingState[curState] == 0){    //非终态，将CH中字符拼接到strToken中，继续读入字符分析过程
            Concat();
            GetChar();
        }
        else if(isAcceptingState[curState] == 1){//识别到一个单词，并且当前读入字符也是单词的一部分
            Concat();                                                           //将读入字符CH拼接
            std::cerr << "1.find a words: " << strTokenLen << ": " << strToken << std::endl;
            pushAns();                                                          //调用保存结果函数，查表等获得二元组
            memset(strToken, '\0', sizeof strToken);//清空strToken等，为下一次分析做准备
            strTokenLen = 0;
            GetBC();                                                            //读到下一个非空字符
            curState = 0;
        }
        else if(isAcceptingState[curState] == 2){//识别到一个单词，并且当前读入字符不是单词的一部分时
            std::cerr << "2.find a words: " << strTokenLen << ": " << strToken << std::endl;
            pushAns();
            memset(strToken, '\0', sizeof strToken);
            strTokenLen = 0;
            // Concat();                                                        //当前字符要进入下一次分析，所以不拼接到strToken中，也不进行读入新字符的操作（除空白字符外）
            curState = 0;
            if(IsBlank())GetBC();                                   //如果当前读入的字符是空白符，也就是用空白符分隔所得到的单词时，显然为了下一次分析要不断地读到非空字符
        }
        else{                                                                   //未定义的状态，此时读入的字符是文法所为定义的字符，提示报错，退出程序
            std::cerr << "error!" << std::endl;
            break;
        }
        if(CH == '\0')break;                                    //分析到文件末结束分析
    }
```

```
}

std::string getAns(){
    LexicalAnalysiInit();
    work();
    std::string ret;
    for(int i = 0; i < ans.size(); ++i)ret += "(" + ans[i].second + "," + ans[i].first + ")\n";
    return ret;
}


// int main(){

//     freopen("test.txt", "r", stdin);
//     freopen("ans.txt", "w", stdout);

//     init();
//     work();
//     // for(auto i: ans)cout << "(" << i.second << "," << i.first << ")" << endl;
//      //输出二元组结果
//     for(int i = 0; i < ans.size(); ++i)std::cout << "(" << ans[i].second << "," << ans[i].first << ")" << std::endl;

//     return 0;
// }
```

SemanticAnalysisAndIntermediateCodeGeneration.cpp

```cpp
// SemanticAnalysisAndIntermediateCodeGeneration.cpp
#include<iostream>
#include<cstdio>
#include<string.h>
#include<sstream>
#include"symbols.h"
#include"symbolsVN.cpp"
#include"symbolsVT.cpp"
#include"production.cpp"
#include"analysisTable.cpp"
#include"LexicalAnalysi.cpp"
const int maxnAnalysisStack = 1e2 + 5;

// 定义出文法的所有终结符
symbolsVT* PLUS = new symbolsVT("+", "plus");
symbolsVT* MINUS = new symbolsVT("-", "minus");
symbolsVT* times = new symbolsVT("*", "times");
symbolsVT* slash = new symbolsVT("/", "slash");
symbolsVT* lparen = new symbolsVT("(", "lapren");
symbolsVT* rparen = new symbolsVT(")", "rparen");
symbolsVT* ident = new symbolsVT("i", "ident");
symbolsVT* unsignint = new symbolsVT("u", "unsignint");
symbolsVT* END = new symbolsVT("#", "end");
symbolsVT* epslion = new symbolsVT("e", "epslion");
// 定义出文法的所有非终结符
symbolsVN* Sdot = new symbolsVN("S'");
symbolsVN* E = new symbolsVN("E");
symbolsVN* T = new symbolsVN("T");
symbolsVN* F = new symbolsVN("F");

// 构造所有的产生式
production* Sdotproduction[1];
production* Eporduction[5];
production* Tproduction[3];
production* Fproduction[3];

// 定义出预测分析表
analysisTable AnalysisTable(21);

// 分析栈
std::pair<int, symbols*> analysisStack[maxnAnalysisStack];
std::string placeStack[maxnAnalysisStack];
int ansStack[maxnAnalysisStack];
int top;

/****************** some basic function ******************/
std::string to_string(int a){
    std::string s;
    while(a){
        s.push_back((char)(a % 10 + '0'));
        a /= 10;
    }
    s.reserve();
    return s;
}
int _stoi(std::string s){
    int ans = 0;
    for(int i = 0; i < s.size(); ++i){
        ans *= 10;
        ans += s[i] - '0';
    }
    return ans;
}

/****************** some basic function ******************/

// 属性文法的内容
int numofnewtemp;
std::string newtemp(){
    return "t" + std::to_string(numofnewtemp++);
}
std::vector<std::string> IntermediateCode;
void SdotToE(production *p){
    symbolsVN *vn = p->getVN();
    if(Arithmetic){
        vn->setAns(p->getProductionIndexOf(0)->getAns());
    }
    else{
        vn->setPlace(((symbolsVN*)(p->getProductionIndexOf(0)))->getPlace());
    }
}
void EToT(production *p){
    symbolsVN *vn = p->getVN();
    if(Arithmetic){
        vn->setAns(p->getProductionIndexOf(0)->getAns());
    }
    else{
        vn->setPlace(((symbolsVN*)(p->getProductionIndexOf(0)))->getPlace());
    }
}
void EToPlusT(production *p){
```

```
    symbolsVN *vn = p->getVN();
    if(Arithmetic){
        vn->setAns(p->getProductionIndexOf(1)->getAns());
    }
    else{
        vn->setPlace(((symbolsVN*)(p->getProductionIndexOf(1)))->getPlace());
    }
}
void EToMinusT(production *p){
    symbolsVN *vn = p->getVN();
    if(Arithmetic){
        vn->setAns(-(p->getProductionIndexOf(1)->getAns()));
    }
    else{
        std::string oldPlace = ((symbolsVN*)(p->getProductionIndexOf(1)))->getPlace();
        vn->setPlace(newtemp());
        IntermediateCode.push_back("(uminus," + oldPlace + ",," + vn->getPlace() + ")");
    }
}
void EToEPlusT(production *p){
    symbolsVN *vn = p->getVN();
    if(Arithmetic){
        vn->setAns(p->getProductionIndexOf(0)->getAns() + p->getProductionIndexOf(2)->getAns());
    }
    else{
        std::string oldPlace = ((symbolsVN*)(p->getProductionIndexOf(0)))->getPlace();
        vn->setPlace(newtemp());
        IntermediateCode.push_back("(+," + oldPlace + "," + ((symbolsVN*)(p->getProductionIndexOf(2)))->getPlace() + "," + vn->getPlace() + ")");
    }
}
void EToEMinusT(production *p){
    symbolsVN *vn = p->getVN();
    if(Arithmetic){
        vn->setAns(p->getProductionIndexOf(0)->getAns() - p->getProductionIndexOf(2)->getAns());
    }
    else{
        std::string oldPlace = ((symbolsVN*)(p->getProductionIndexOf(0)))->getPlace();
        vn->setPlace(newtemp());
        IntermediateCode.push_back("(-," + oldPlace + "," + ((symbolsVN*)(p->getProductionIndexOf(2)))->getPlace() + "," + vn->getPlace() + ")");
    }
}
void TToF(production *p){
    symbolsVN *vn = p->getVN();
    if(Arithmetic){
        vn->setAns(p->getProductionIndexOf(0)->getAns());
    }
    else{
        vn->setPlace(((symbolsVN*)(p->getProductionIndexOf(0)))->getPlace());
    }
}
void TToTTimesF(production *p){
    symbolsVN *vn = p->getVN();
    if(Arithmetic){
        vn->setAns(p->getProductionIndexOf(0)->getAns() * p->getProductionIndexOf(2)->getAns());
    }
    else{
        std::string oldPlace = ((symbolsVN*)(p->getProductionIndexOf(0)))->getPlace();
        vn->setPlace(newtemp());
        IntermediateCode.push_back("(*," + oldPlace + "," + ((symbolsVN*)(p->getProductionIndexOf(2)))->getPlace() + "," + vn->getPlace() + ")");
    }
}
void TToTSlashF(production *p){
    symbolsVN *vn = p->getVN();
    if(Arithmetic){
        vn->setAns(p->getProductionIndexOf(0)->getAns() / p->getProductionIndexOf(2)->getAns());
    }
    else{
        std::string oldPlace = ((symbolsVN*)(p->getProductionIndexOf(0)))->getPlace();
        vn->setPlace(newtemp());
        IntermediateCode.push_back("(/," + oldPlace + "," + ((symbolsVN*)(p->getProductionIndexOf(2)))->getPlace() + "," + vn->getPlace() + ")");
    }
}
void FToi(production *p){
    symbolsVN *vn = p->getVN();
    if(Arithmetic){
        vn->setAns(p->getProductionIndexOf(0)->getAns());
    }
    else{
        vn->setPlace(((symbolsVT*)(p->getProductionIndexOf(0)))->getVar());
    }
}
void FTou(production *p){
    symbolsVN *vn = p->getVN();
    if(Arithmetic){
        vn->setAns(_stoi(placeStack[top])); //???????
    }
    else{
        vn->setPlace(((symbolsVT*)(p->getProductionIndexOf(0)))->getVar());
    }
}
void FToSpanE(production *p){
```

```
    symbolsVN *vn = p->getVN();
    if(Arithmetic){
        vn->setAns(p->getProductionIndexOf(1)->getAns());
    }
    else{
        vn->setPlace(((symbolsVN*)(p->getProductionIndexOf(1)))->getPlace());
    }
}
void SemanticAnalysisAndIntermediateCodeGenerationInit(){
    numofnewtemp = 1;
    // 初始化所有变量
        // 根据文法的不同，得到的分析表的结构也不同，此时初始化部分也不同

    // 定义出预测分析表
        // 为预测分析表插入终结符、非终结符
    AnalysisTable.insertSymbols(PLUS);
    AnalysisTable.insertSymbols(MINUS);
    AnalysisTable.insertSymbols(times);
    AnalysisTable.insertSymbols(slash);
    AnalysisTable.insertSymbols(lparen);
    AnalysisTable.insertSymbols(rparen);
    AnalysisTable.insertSymbols(ident);
    AnalysisTable.insertSymbols(unsignint);
    AnalysisTable.insertSymbols(END);

    AnalysisTable.insertSymbols(Sdot);
    AnalysisTable.insertSymbols(E);
    AnalysisTable.insertSymbols(T);
    AnalysisTable.insertSymbols(F);

    // 根据文法定义E的三条产生式，同理处理其他的产生式
    for(int i = 0; i < 1; ++i)Sdotproduction[i] = new production(Sdot);
    Sdotproduction[0]->push_back(E);
    Sdotproduction[0]->setAttributesFunction(SdotToE);
    Sdotproduction[0]->print();


    for(int i = 0; i < 5; ++i)Eporduction[i] = new production(E);

    Eporduction[0]->push_back(T);
    Eporduction[1]->push_back(PLUS); Eporduction[1]->push_back(T);
    Eporduction[2]->push_back(MINUS); Eporduction[2]->push_back(T);
    Eporduction[3]->push_back(E); Eporduction[3]->push_back(PLUS); Eporduction[3]->push_back(T);
    Eporduction[4]->push_back(E); Eporduction[4]->push_back(MINUS); Eporduction[4]->push_back(T);
    for(int i = 0; i < 5; ++i)E->insertProduction(Eporduction[i]);
    Eporduction[0]->setAttributesFunction(EToT);
    Eporduction[1]->setAttributesFunction(EToPlusT);
    Eporduction[2]->setAttributesFunction(EToMinusT);
    Eporduction[3]->setAttributesFunction(EToEPlusT);
    Eporduction[4]->setAttributesFunction(EToEMinusT);
    for(int i = 0; i < 5; ++i)Eporduction[i]->print();

    for(int i = 0; i < 3; ++i)Tproduction[i] = new production(T);
    Tproduction[0]->push_back(F);
    Tproduction[1]->push_back(T); Tproduction[1]->push_back(times); Tproduction[1]->push_back(F);
    Tproduction[2]->push_back(T); Tproduction[2]->push_back(slash); Tproduction[2]->push_back(F);
    for(int i = 0; i < 3; ++i)T->insertProduction(Tproduction[i]);
    Tproduction[0]->setAttributesFunction(TToF);
    Tproduction[1]->setAttributesFunction(TToTTimesF);
    Tproduction[2]->setAttributesFunction(TToTSlashF);
    for(int i = 0; i < 3; ++i)Tproduction[i]->print();


    for(int i = 0; i < 3; ++i)Fproduction[i] = new production(F);
    Fproduction[0]->push_back(ident);
    Fproduction[1]->push_back(unsignint);
    Fproduction[2]->push_back(lparen); Fproduction[2]->push_back(E); Fproduction[2]->push_back(rparen);
    for(int i = 0; i < 3; ++i)F->insertProduction(Fproduction[i]);
    Fproduction[0]->setAttributesFunction(FToi);
    Fproduction[1]->setAttributesFunction(FTou);
    Fproduction[2]->setAttributesFunction(FToSpanE);
    for(int i = 0; i < 3; ++i)Fproduction[i]->print();

    for(int i = 0; i < 1; ++i)AnalysisTable.insertProduction(Sdotproduction[i]);
    for(int i = 0; i < 5; ++i)AnalysisTable.insertProduction(Eporduction[i]);
    for(int i = 0; i < 3; ++i)AnalysisTable.insertProduction(Tproduction[i]);
    for(int i = 0; i < 3; ++i)AnalysisTable.insertProduction(Fproduction[i]);

    // 给出LR分析表
    AnalysisTable.insert(0, PLUS, 's', 5); AnalysisTable.insert(0, MINUS, 's', 4); AnalysisTable.insert(0, lparen, 's', 8); AnalysisTable.insert(0, ident, 's'
    AnalysisTable.insert(1, PLUS, 's', 9); AnalysisTable.insert(1, MINUS, 's', 10); AnalysisTable.insert(1, END, 'a', -1);
    AnalysisTable.insert(2, PLUS, 'r', 1); AnalysisTable.insert(2, MINUS, 'r', 1); AnalysisTable.insert(2, times, 's', 11); AnalysisTable.insert(2, slash, 's'
    AnalysisTable.insert(3, PLUS, 'r', 6); AnalysisTable.insert(3, MINUS, 'r', 6); AnalysisTable.insert(3, times, 'r', 6); AnalysisTable.insert(3, slash, 'r',
    AnalysisTable.insert(4, T, ' ', 13);
    AnalysisTable.insert(5, T, ' ', 14);
    AnalysisTable.insert(6, PLUS, 'r', 9); AnalysisTable.insert(6, MINUS, 'r', 9); AnalysisTable.insert(6, times, 'r', 9); AnalysisTable.insert(6, slash, 'r',
    AnalysisTable.insert(7, PLUS, 'r', 10); AnalysisTable.insert(7, MINUS, 'r', 10); AnalysisTable.insert(7, times, 'r', 10); AnalysisTable.insert(7, slash, 'r
    AnalysisTable.insert(8, PLUS, 's', 5); AnalysisTable.insert(8, MINUS, 's', 4); AnalysisTable.insert(8, lparen, 's', 8); AnalysisTable.insert(8, ident, 's'
    AnalysisTable.insert(9, lparen, 's', 8); AnalysisTable.insert(9, ident, 's', 6); AnalysisTable.insert(9, unsignint, 's', 7); AnalysisTable.insert(9, T, '
    AnalysisTable.insert(10, lparen, 's', 8); AnalysisTable.insert(10, ident, 's', 6); AnalysisTable.insert(10, unsignint, 's', 7); AnalysisTable.insert(10, T
    AnalysisTable.insert(11, lparen, 's', 8); AnalysisTable.insert(11, ident, 's', 6); AnalysisTable.insert(11, unsignint, 's', 7); AnalysisTable.insert(11, F
```

```
    AnalysisTable.insert(12, lparen, 's', 8); AnalysisTable.insert(12, ident, 's', 6); AnalysisTable.insert(12, unsignint, 's', 7); AnalysisTable.insert(12, F
    AnalysisTable.insert(13, PLUS, 'r', 3); AnalysisTable.insert(13, MINUS, 'r', 3); AnalysisTable.insert(13, rparen, 'r', 3); AnalysisTable.insert(13, END, 'r
    AnalysisTable.insert(14, PLUS, 'r', 2); AnalysisTable.insert(14, MINUS, 'r', 2); AnalysisTable.insert(14, rparen, 'r', 2); AnalysisTable.insert(14, END, 'r
    AnalysisTable.insert(15, PLUS, 's', 9); AnalysisTable.insert(15, MINUS, 's', 10); AnalysisTable.insert(15, rparen, 's', 20);
    AnalysisTable.insert(16, PLUS, 'r', 4); AnalysisTable.insert(16, MINUS, 'r', 4); AnalysisTable.insert(16, times, 's', 11); AnalysisTable.insert(16, slash,
    AnalysisTable.insert(17, PLUS, 'r', 5); AnalysisTable.insert(17, MINUS, 'r', 5); AnalysisTable.insert(17, times, 's', 11); AnalysisTable.insert(17, slash,
    AnalysisTable.insert(18, PLUS, 'r', 7); AnalysisTable.insert(18, MINUS, 'r', 7); AnalysisTable.insert(18, times, 'r', 7); AnalysisTable.insert(18, slash,
    AnalysisTable.insert(19, PLUS, 'r', 8); AnalysisTable.insert(19, MINUS, 'r', 8); AnalysisTable.insert(19, times, 'r', 8); AnalysisTable.insert(19, slash,
    AnalysisTable.insert(20, PLUS, 'r', 11); AnalysisTable.insert(20, MINUS, 'r', 11); AnalysisTable.insert(20, times, 'r', 11); AnalysisTable.insert(20, slash
    AnalysisTable.print();

        // 初始化分析栈
    top = -1;
}
void release(){
        // 释放所有的动态申请的资源
    delete PLUS;
        delete MINUS;
        delete times;
        delete slash;
        delete lparen;
        delete rparen;
        delete ident;
        delete unsignint;
        delete END;
        delete epslion;
    delete E;
        delete T;
        delete F;
    for(int i = 0; i < 1; ++i)delete Sdotproduction[i];
    for(int i = 0; i < 5; ++i)delete Eporduction[i];
    for(int i = 0; i < 3; ++i)delete Tproduction[i];
    for(int i = 0; i < 3; ++i)delete Fproduction[i];
}
std::string word, code;
// char word[10], code[10];
char ch;
symbolsVT* a;
std::string LexicalAnalysis;    // 调用词法分析结果
std::stringstream ss(LexicalAnalysis = getAns());               // 将词法分析的结果作为输入流
void ADVANCE(){

        // 读入一个词法分析的结果项，同时给出对应的终结符a
    // if(scanf("(%s,%s)", code, word) != -1){
    // std::cin >> ch;
    ss >> ch;

    if(!ss.eof()){
    // if(scanf("%c", &ch) != -1){
        std::getline(ss, code, ',');
        std::getline(ss, word);
        word.resize(word.size() - 1);
        // std::cin >> ch;
        std::cerr << word << " " << code << std::endl;
        if(code == "plus")a = PLUS;
        else if(code == "minus") a = MINUS;
        else if(code == "times") a = times;
        else if(code == "slash") a = slash;
        else if(code == "lparen") a = lparen;
        else if(code == "rparen") a = rparen;
        else if(code == "ident") a = ident;
        else if(code == "number") a = unsignint;
        a->setVar(word);
    }
    else{
        a = END;
        a->setVar("#");
    // if(std::cin.eof() == EOF){
        std::cerr << "ADVANCE In End....." << std::endl;
    }
    std::cerr << word << "_____________" << code << std::endl;
}
bool SemanticAnalysisAndIntermediateCodeGeneration(){
        // 预测分析程序的总控程序
    SemanticAnalysisAndIntermediateCodeGenerationInit();
    std::cerr << "Arithmetic: " << Arithmetic << std::endl;
    std::cerr << "LexicalAnalysis: \n" << LexicalAnalysis << std::endl;
    bool grammer = true;                // 表示句子是否符合一定的文法
    bool flag = true;                   // 总控程序的运行标志
    analysisStack[++top] = std::make_pair(0, (symbols*)END);  // 初始化栈，将状态0和符号#压入
    std::pair<int, symbols*> X;                                 // 定义一个公共变量：状态和符号的指针
    production *p;                              // 定义一个产生式的指针
    std::pair<char, int> state; // 从分析表中获得的状态信息
    ADVANCE();                                  // 读入一个词法分析的结果项
    while(flag){

                //*********************************************************//
                // 调试信息：状态栈和符号栈的中内容
        std::cerr << std::endl << std::endl;
        std::cerr << "================" << std::endl;
        a->print();
```

```
        std::cerr << "stack: " << std::endl;
        std::cerr << "state: \t" ;
        for(int i = 0; i <= top; ++i){
            std::cerr << analysisStack[i].first << " ";
        }
        std::cerr << std::endl;
        std::cerr << "symbols: \t" ;
        for(int i = 0; i <= top; ++i){
            if(analysisStack[i].second->getClassName() == "VT")std::cerr << ((symbolsVT*)(analysisStack[i].second))->getWord() << " ";
            else std::cerr << ((symbolsVN*)analysisStack[i].second)->getName() << " ";
        }
        std::cerr << std::endl;
        std::cerr << "place: \t" ;
        for(int i = 0; i <= top; ++i){
            std::cerr << placeStack[i] << " ";
        }
        std::cerr << std::endl;
        std::cerr << "ans: \t";
        for(int i = 0; i <= top; ++i){
            std::cerr << ansStack[i] << " ";
        }
        std::cerr << std::endl << "================" << std::endl;
        std::cerr << std::endl;
                //*********************************************************//


        X = analysisStack[top]; // 得到分析栈的栈顶元素，pop操作
        state = AnalysisTable.get(X.first, a);  // 根据栈顶的状态以及分析表中的变化情况来获得下一转换的状态s_i, r_i, acc, i等等
        std::cerr << state.first << "  " << state.second << std::endl;
        if(state.first == 's'){         // 如果是移进状态
            analysisStack[++top] = std::make_pair(state.second, a);
            placeStack[top] = a->getVar();
            ADVANCE();
            std::cerr << "One SHIFT..." << std::endl << std::endl;;
        }
        else if(state.first == 'r' || state.first == 'a'){      // 如果是规约状态
            if(state.first == 'a')state.second = 0;
            p = AnalysisTable.getProduction(state.second);      // 获得第i个产生式
            p->print();
            int len = p->getLen();
                        // 恢复产生式对应到的非终结符的属性文法的值
            for(int i = 0; i < len; ++i)p->getProductionIndexOf(len - i - 1)->setPlace(placeStack[top - i]);
            for(int i = 0; i < len; ++i)p->getProductionIndexOf(len - i - 1)->setAns(ansStack[top - i]);
            p->getAttributesFunction()(p);              // 调用该产生式对应的语义分析函数，实现中间代码生成或者表达式值的计算
            top -= len;                                 // 将栈顶的符号按照产生式来规约
            X = analysisStack[top];             // 获得此时的栈顶元素，据此来获得GOTO表的下一状态
            analysisStack[++top] = std::make_pair(AnalysisTable.get(X.first, p->getVN()).second, p->getVN());
            placeStack[top] = p->getVN()->getPlace();
            ansStack[top] = p->getVN()->getAns();
            std::cerr << "One REDUCE..." << std::endl << std::endl;
            if(state.first == 'a'){
                std::cerr << "ACC!!!" << std::endl << std::endl;
                flag = false;
            }
        }
        // else if(state.first == 'a'){ // 如果是acc状态
        //     std::cerr << "ACC!!!" << std::endl << std::endl;
        //     flag = false;
        // }
        else{                                           // 到达分析表的其他状态，错误
            grammer = false;
            flag = false;
        }
    }

    release();          // 释放资源
    if(Arithmetic)std::cout << Sdot->getAns() << std::endl;
    else for(int i = 0; i < IntermediateCode.size(); ++i)std::cout << IntermediateCode[i] << std::endl;
    return grammer;     // 返回结果，true表示句子符合一定的语法
}
```

调试数据

```
// input
2+3*5
// output
17
```

```
// input
a*(b+c)
// output
(+,b,c,t1)
(*,a,t1,t2)
```

实验体会

本实验是对属性文法和语义分析及中间代码的生成的学习后的一个实现，语义分析的主要解决的问题是将代码进行词法分析、语法分析等后要得到对应的中间代码，实现编译器将一种语言像另一种语言转化的基础。在分析过程中要分析每一个产生式对应的属性文法，根据确定的属性文法的内容，进行中间代码的生成。为了实现语法分析和语义分析的一次性扫描生成，本实验选取实验三完成的自下而上的SLR分析器作为基础，添加相关的语义分析内容，并将实验一的词法分析器加入到整个项目当中，实现由基础的句子进行词法分析、语法分析、语义分析以及中间代码的产生过程。在实验的开始时，虽然很快的得出了每一个产生式的属性文法，但在开始代码实现时遇到了困难：如何为不同的产生式对象赋予不同的属性文法的执行函数，此时首先想到的一个简单的解决方法就是在每一次产生式的规约过程中，判断是哪一个产生式，然后对其进行执行相应的操作即可，这样逻辑上显然是没有问题的，但是，一个显而易见的问题就是，这样的实现会使总控程序的规约操作中产生一系列的产生式的判断以及相应的语义分析代码，暂且不考虑程序的效率问题，这样的代码首先没有良好的维护性，该实验只是尝试分析PL0的表达式相关的文法，倘若要实现所有的文法的分析，这一块的判断的代码量就会很大很大。最后想到为每一个产生式提供一个调用语义分析的入口，也就是一个简单的函数指针，然后对于不同的产生式根据属性文法实现对应的语义分析代码函数，在实例化产生式时，将函数指针指向即可，这样在总控程序的规约过程中，只要简单的调用函数指针指向的函数即可，当然为了语义分析能够使用的是对应的产生式，只需给出产生式的指针作为其参数即可。除此之外，在第一次调试代码时，我发现对于一些句子的执行是正确的，而有一些是错误的，调试分析后，发现是一些长句子中，分析栈中会出现多个同样的非终结符，而我将属性文法的一些成员变量设置在符号上（例如place属性），这样后来压入栈中的非终结符的属性文法的值就会覆盖前面栈中的同样非终结符的值，导致最后的分析结果出错，所以增加了一个 `placeStack` 栈，与分析栈同步保存对应位置上的非终结符的place值，当要规约时，首先将该规约产生式的所有符号的属性值从栈中恢复后再进行语义分析即可，同样对于算术表达式，设置一个 `ansStack` 保存子表达式的值，不断的进行规约同时进行语义分析即可。最后在提交到评测机上时，因为评测机的编译器不支持多文件的链接，所以将所有项目文件合并一个进行提交，此前几次实验总是出现某个类成员函数因为使用到了某个前向声明的类的成员，导致编译错误，因为这个函数只是调试函数，所有前几次实验没有管它解决的方法，简单的删除了实现就提交了，这次实验查找了很多的解决方法，最后根据自己的理解将这个函数的实现放在了所需的类的后面就解决了这个编译问题，c++中，类的前向声明只能使用它自己，不能使用该类的成员，仔细想一下就能明白：此时类的成员声明还在后面，g++编译器自上而下分析时显然不知道这个前向声明的类的具体成员有什么的。这就是这次是实验的体会，无论是编译原理还是用了很久的c++都使我收获很多。

HTML