编译原理实验报告

实验题目：		学号：
日期：2024.06.14	班级：	姓名：
Email：		

实验目的：

在本实验中，需要依靠实验一中完成的词法分析器对输入进行预处理得到 Token 流，利用 LR（1）方法进行语法分析并进行语法制导翻译生成符号表和四元式，并将其输出。

实验环境介绍：

硬件环境：

Legion R7000P2021 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz

软件环境：

Clion

解决问题的主要思路：

本实验中将会给定 C++语言风格的源程序，我们需要利用实验一的词法分析器作预处理。如果这是一段正确的程序（即不存在语法错误），就需要输出该程序的符号表以及四元式表

本实验解决过程分以下几个步骤：

1. 针对给出的 C++风格的源程序，用实验一对其进行词法分析，得到源程序的 Token 流。

2. 根据实验中所以给出的终结符、非终结符以及文法，求出每个非终结符的 first 集合。

3. 在得到每个非终结符的 first 集合后，先进行拓广文法，然后求出 LR（1）分析的项目集规范族，同时构建 LR(1)表。

4. 得到 LR（1）表后，接下来定义一个状态栈、一个符号栈，根据 Token 流进行移进归约，同时处理归约过程中的相应动作。

5. 如果归约过程中存在错误的话，输出 Syntax Error 直接结束，如果不存在错误的话，最后输出该程序的符号表以及四元式表即可。

实验步骤：

在本实验中，使用文法中出现的终结符集合如下：

int	double	scanf	printf	If	then	while
do	,	;	+	-	*	/
=	==	!=	<	<=	>	>=
(	)	{	}	!	&&	||
id	UINT	UFLOAT	END	SPACE		

注意：由于题目中并未给出空字符和结束字符，因此我们自己加入空字符和结束字符，即空字符对应的终结符为 SPACE，结束字符对应的终结符为 END。

文法中出现的所有非终结符集合如下：

PROG	SUBPROG	M	N	VARIABLES	STATEMENT	VARIABLE
T	ASSIGN	SANCF	PRINTF	L	B	EXPR
ORITEM	ANDITEM	RELITEM	NOITEM	ITEM	FACTOR	BORTEARM
BANDREAM	MUL_DIV	REL	SACNF_BEGIN	PRINTF_BEGIN	ID	PLUS_MINUS
START						

注意：为进行拓广文法，我们额外加入一个非终结符，即 START。

语法产生式和制导翻译规则如下：

总程序部分：

PROG -> SUBPROG{}

SUBPROG -> M VARIABLES STATEMENT

{backpatch(STATEMENT.nextlist,nxq);gen(End,-,-,-);}

M -> ^{OFFSET=0}

N -> ^{N.quad=nxq}

变量声明部分：

VARIABLES -> VARIABLES VARIABLE; {}

VARIABLES -> VARIABLE; {}

T -> int {T.type=int;T.width=4;}

T -> double {T.type=double;T.width=8;}

ID -> id {ID.name=id}

VARIABLE ->T ID

{enter(ID.name,type=T.type,offset=OFFSET);OFFSET+=T.width;VARIABLE.type=T.type;

VARIABLE.width=T.width}

VARIABLE -> VARIABLE_1,ID

{enter(ID.name,type=VARIABLE_1.type,offset=OFFSET);OFFSET+=VARIABLE_1.width;

VARIABLE.type=VARIABLE_1.type;VARIABLE.width=VARIABLE_1.width;}

语句部分(赋值、读写、if-then,do-while,复合语句)

STATEMENT -> ASSIGN{STATEMENT.nextlist=mklist()}

STATEMENT -> SCANF{STATEMENT.nextlist=mklist()}

STATEMENT -> PRINTF{STATEMENT.nextlist=mklist()}

STATEMENT -> ^{STATEMENT.nextlist=mklist()}

STATEMENT -> { L ; }{STATEMENT.nextlist=L.nextlist}

STATEMENT -> while N_1 B do N_2 STATEMENT_1

{backpatch(STATEMENT_1.nextlist,N_1.quad);backpatch(B.truelist,N_2.quad);

STATEMENT.nextlist=B.falselist;gen(j,-,-,N_1.quad)}

STATEMENT -> if B then N STATEMENT_1

{backpatch(B.truelist,N.quad),STATEMENT.nextlist=merge(B.falselist,

STATEMENT_1.nextlist)}

ASSIGN -> ID = EXPR
{p=lookup(ID.name);gen(=,EXPR.place,-,p) }

L -> L_1 ; N STATEMENT

{backpatch(L1.nextlist,N.quad),L.nextlist=STATEMENT.nextlist}

L -> STATEMENT {L.nextlist=STATEMENT.nextlist}

数值表达式部分

EXPR ->EXPR_1 || ORITEM{EXPR.place=newtemp(int);EXPR.type=int;

gen(||,EXPR_1.place,ORITEM.place,EXPR.place)}

EXPR -> ORITEM {EXPR.place=ORITEM.place;EXPR.type=ORITEM.type}

ORITEM -> ORITEM_1 && ANDITEM

{ORITEM.place=newtemp(int);ORITEM.type=int;

gen(&&,ORITEM_1.place,ANDITEM.place,ORITEM.place)}

ORITEM -> ANDITEM

{ORITEM.place=ANDITEM.place;ORITEM.type=ANDITEM.type}

ANDITEM -> NOITEM
{ANDITEM.place=NOITEM.place;ANDITEM.type=NOITEM.type;}

ANDITEM -> !NOITEM

{ANDITEM=newtemp(int);ANDITEM.type=int;gen(!,NOITEM.place,-,ANDITEM.place)}

NOITEM -> NOITEM_1 REL RELITEM {NOITEM.place=newtemp(int);NOITEM.type=int;

gen(REL.op,NOITEM_1.place,RELITEM.place,NOITEM.place)}

NOITEM -> RELITEM

{NOITEM.place=RELITEM.place;NOITEM.type=RELITEM.type}

RELITEM -> RELITEM_1 PLUS_MINUS ITEM

{RELITEM.place=newtemp(RELITEM_1.type);RELITEM.type=RELITEM_1.type;

gen(PLUS_MINUS.op,RELITEM_1.place,ITEM.place,RELITEM.place)}

RELITEM -> ITEM {RELITEM.place=ITEM.place;RELITEM.type=ITEM.type}

ITEM -> FACTOR {ITEM.place=FACTOR.place;ITEM.type=FACTOR.type}

ITEM -> ITEM MUL_DIV FACTOR

{ITEM.place=newtemp(FACTOR.type);ITEM.type=FACTOR.type;

gen(MUL_DIV.op,ITEM_1.place,FACTOR.place,ITEM.place)}

FACTOR -> ID {FACTOR.place=lookup(ID.name);FACTOR.type=lookup_type(ID.name)}

FACTOR -> UINT

{FACTOR.place=newtemp(int);FACTOR.type=int;gen(=,UINT,-,FACTOR.place)}

FACTOR -> UFLOAT

{FACTOR.place=newtemp(double);FACTOR.type=double;gen(=,UFLOAT,-,FACTOR.place)}

FACTOR -> (EXPR) {FACTOR.place=EXPR.place;FACTOR.type=EXPR.type}

FACTOR -> PLUS_MINUS FACTOR_1

{FACTOR.place=newtemp(FACTOR_1.type);FACTOR.type=FACTOR_1.type;

gen(PLUS_MINUS.op,0,FACTOR_1.place,FACTOR.place)}

作为条件控制的表达式

```
B -> B_1 || N BORTERM {backpatch(B_1.falselist,N.quad);
B.truelist=merge(B_1.truelist,BORTERM.truelist);B.falselist=BORTERM.falselist}
B -> BORTERM
{B.truelist=BORTERM.truelist;B.falselist=BORTERM.falselist}
BORTERM -> BORTERM_1 && N BANDTERM{backpatch(BORTERM_1.truelist,N.quad);
BORTERM.falselist=merge(BORTERM_1.falselist,BANDTERM.falselist;
BORTERM.truelist=BANDTERM.truelist)}
BORTERM -> BANDTERM
{BORTERM.truelist=BANDTERM.truelist;BORTERM.falselist=BANDTERM.falselist}
BANDTERM -> ( B )
{BANDTERM.truelist=B.truelist;BANDTERM.falselist=B.falselist}
BANDTERM -> ! BANDTERM_1
{BANDTERM.truelist=BANDTERM_1.falselist;BANDTERM.falselist=BANDTERM_1.truelist}
BANDTERM -> BFACTOR_1 REL BFACTOR_2
{BANDTERM.truelist=mklist(nxq);BANDTERM.falselist=mklist(nxq+1);
gen(j+REL.op,BFACTOR_1.place,BFACTOR_2.place,0);gen(j,-,-,0);}
BANDTERM -> BFACTOR
{BANDTERM.truelist=mklist(nxq);BANDTERM.falselist=mklist(nxq+1);
gen(jnz,BFACTOR.place,-,0);gen(j,-,-,0)}
BFACTOR -> UINT
{BFACTOR.place=newtemp(int);BFACTOR.type=int;gen(=,UINT,-,FACTOR.place)}
BFACTOR -> UFLOAT
{BFACTOR.place=newtemp(double);BFACTOR.type=double;gen(=,UFLOAT,-,BFACTOR.place)}
BFACTOR -> ID
{BFACTOR.place=lookup(ID.name);BFACTOR.type=lookup_type(ID.name)}
```

运算符

```
PLUS_MINUS -> +  {PLUS_MINUS.op='+'}
PLUS_MINUS -> -  { PLUS_MINUS.op='-'}
```

MUL_DIV -> * {MUL_DIV.op='*'}

MUL_DIV -> / { MUL_DIV.op='/'}

REL -> == {REL.op='=='}

REL -> != { REL.op='!='}}

REL -> < { REL.op='<'}}

REL -> <= { REL.op='<='}

REL -> > { REL.op='>'}}

REL -> >= { REL.op='>='}

读写语句

SCANF -> SCANF_BEGIN) {}

SCANF_BEGIN -> SCANF_BEGIN , ID {p=lookup(ID.name);gen(R,-,-,p)}

SCANF_BEGIN -> scanf (ID {p=lookup(ID.name);gen(R,-,-,p)}

PRINTF -> PRINTF_BEGIN) {}

PRINTF_BEGIN -> printf (ID {p=lookup(ID.name);gen(W,-,-,p)}

PRINTF_BEGIN -> PRINTF_BEGIN , ID {p=lookup(ID.name);gen(W,-,-,p)}

1. 本实验中涉及到符号表，而符号表中的符号有四个属性 name，type，value，offset，为此我们定义一个结构体来表示，如下所示：

```
// 符号表中的符号
struct symbol {
    string name;
    int type;
    int offset;
    int index; // 处在符号表中的位置
};
```

2.为方便调试，除了定义结构体外，我们在额外定义一个符号表类，类中有 vector<symbol> symbols 来记录语法分析过程中定义的符号，另有 enter、lookup、looktype 三个函数分别对应相关操作，即 enter（name，type，offset）对应向符号表中添加名称为 name，类型为 type，偏移量为 offset 的变量，lookup（name）可以理解为返回给定变量在符号表的位置，而 looktype(name)则是根据输入的变量名，查找并返回符号表中对应变量的类型。

```
class symbolTable {
public:
    void enter(string name, int type); // 往符号表中加入符号
    string lookup(string name);
    int looktype(string name);
    vector<symbol> symbols;
    int nextIndex = 0;
    int Offset = 0; // 记录当前符号表的偏移量
};
```

3.本实验中涉及到输出所有产生的四元式，故我们还需定义一个四元式结构体，而一个四元式中包含四个部分，分别是 op（运算符）、arg1（参数 1）、arg2（参数 2）、result（结果），如下所示：

```
// 定义四元式类
struct Quadruple {
    string op;
    string arg1;
    string arg2;
    string result;
    Quadruple(string op, string arg1, string arg2, string result) : op(op),
arg1(arg1), arg2(arg2), result(result) {}
};
```

4.同理地，我们定义一个四元式类，其中 vector<Quadruple> quads 记录 LR1 分析过程中所有产生的四元式，另有 gen 和 backpatch 两个函数，其中 gen()函数代表的操作是产生四元式，而 backpatch()函数代表的操作则是回填链表，如下所示：

```
class QuadrupleGen {
public:
    void gen(string op, string arg1, string arg2, string result);
    void backpatch(vector<int>&vec, int  quad);
    vector<Quadruple> quads;
};
```

5.在处理完符号表和四元式表后，我们需对文法进行处理，同样地，我们定义一个文法表达式类，其中该类有四个成员变量，分别是 left（文法产生式左部）、vector<string> right（文法产生式右部）、dot（当前文法产生式点所在位置）、lookhead（向前搜索符），以及我们重载了该类的==、<运算符以便后续操作，如下所示：

```
class expression { // 文法表达式
public:
    string left;
    vector<string> right;
    int dot; // 加入点
    vector<string> lookhead;
    expression(string Left, vector<string> Right) : left(Left), right(Right) {};

    bool operator==(const expression &c) const {
        if (left == c.left && right == c.right && dot == c.dot && lookhead ==
c.lookhead)
            return true;
        else
            return false;
    }

    // 重载<操作符以便可以放到set中
    bool operator<(const expression &expre) const {
        if (left != expre.left)
            return left < expre.left;
        if (right != expre.right)
            return right < expre.right;
        if (dot != expre.dot)
            return dot < expre.dot;
        return lookhead < expre.lookhead;
    }
};
```

6.在 LR（1）分析中会产生很多项目集规范族，为此也需要单独一个类来表示，其中该类中有成员变量 set<expression>item 代表的是该项目集中所有产生式和 next 记录跳转,如下所示：

```
class closure {
public:
    set<expression> item; // 项目集规范族的所有产生式
    map<string, int> next; // 记录下跳转
    bool operator==(const closure &other) const {
        return item == other.item;
    }
};
```

7.为构建 GOTO 表，我们需要在构建项目集规范族的同时还需记得它们之间的转移关系，为此定义一个 actionItem 类用来记录移进规约，如下所示：

```
struct actionItem {
public:
    ActionStatus status = ACTION_ERROR;
    int nextState = -1;
    string l;
    vector<string> r;
};
```

8.在所有前置工作完成后，我们定义 Parser 类，该类是我们解决本实验的关键。该类拥有成员变量 first（用来计算所有非终结符的 first 集合）、terminated（非终结符集合）、nonterminated（终结符集合）、collection（所有项目集规范族）、P（所有文法）和构建 LR（1）分析表的 ACTION 和 GOTO 集合。同时定义相关成员函数进行计算，具体如下：

```
class Parser {
public:
    map<string, set<string>> first;
    vector<string> terminated; // 终结符
    vector<string> nonterminated; // 非终结符
    vector<closure> collection; // 所有项目集规范族
    map<string, vector<vector<string>>> P; // 文法
    map<int, vector<actionItem>> ACTION;
    map<int, vector<int>> GOTO;

    void First(); // 计算First集合
    set<string> Getfirst(vector<string> right); // 获取right集合中的first集合
    set<string> computeFirst(const vector<string> &symbols, size_t position, const
vector<string> &lookahead);

    closure GenClosure(const closure &I); // 生成Closure闭包
    void GenLR1(); // 生成LR(1)分析表
    void GenLR1TABLE(); // 建表
    set<string> Getsymbols(const closure &clo); // 获取·后的符号
    closure gotoSet(const closure &I, const string &X); // 对闭包进行移进

    Parser(); // 构造函数
};
```

在构造函数中，对 terminated、nonterminated 和 P 进行初始化，具体如下：

```
Parser::Parser() {
    terminated = { // 终结符
            "INTSYM", "SPACE", "DOUBLESYM", "SCANFSYM", "PRINTFSYM", "IFSYM",
"THENSYM", "WHILESYM", "DOSYM",
            "COMMA", "SEMICOLON", "PLUS", "MINUS", "TIMES", "DIVISION", "EQU",
"EQU1", "NOTEQU", "GREAT", "GE", "LESS",
            "LE", "LBRACE",  // (
            "RBRACE",  // )
            "LBRACE1", // {
            "RBRACE1", // }
            "NOT", "AND", "OR", "UFLOAT", "UINT", "IDENT", "END"
    };

    nonterminated = { // 非终结符
            "START", "PROG", "SUBPROG", "M", "N", "VARIABLES", "STATEMENT",
"VARIABLE", "T",
            "ASSIGN", "SCANF", "PRINTF", "L", "B",
            "EXPR", "ORITEM", "ANDITEM", "RELITEM", "NOITEM", "ITEM", "FACTOR",
"BORTERM", "BANDTERM", "BFACTOR",
            "PLUS_MINUS", "MUL_DIV",
            "REL", "SCANF_BEGIN", "PRINTF_BEGIN", "ID"
    };
```

```
    //总程序部分
    P["START"] = {{"PROG"}},
    P["PROG"] = {{"SUBPROG"}},
    P["SUBPROG"] = {{"M", "VARIABLES", "STATEMENT"}},
    P["M"] = {{"SPACE"}},
    P["N"] = {{"SPACE"}},
    //变量声明部分
    P["VARIABLES"] = {{"VARIABLES", "VARIABLE", "SEMICOLON"},{"VARIABLE",
 "SEMICOLON"}},
    P["T"] = {{"INTSYM"}, {"DOUBLESYM"}},
    P["ID"] = {{"IDENT"}},
    P["VARIABLE"] = {{"T",        "ID"},{"VARIABLE", "COMMA", "ID"}},
     //语句部分
    P["STATEMENT"] = {{"ASSIGN"},{"SCANF"},{"PRINTF"},{"SPACE"},{"LBRACE1",  "L",
"SEMICOLON", "RBRACE1"},
   {"WHILESYM", "N", "B", "DOSYM", "N", "STATEMENT"}, {"IFSYM",    "B", "THENSYM",
  "N",     "STATEMENT"}},
    P["ASSIGN"] = {{"ID", "EQU", "EXPR"}},
    P["L"] = {{"L", "SEMICOLON", "N", "STATEMENT"}, {"STATEMENT"}},
    //数值表达式部分
    P["EXPR"] = {{"EXPR", "OR", "ORITEM"}, {"ORITEM"}},
    P["ORITEM"] = {{"ORITEM", "AND", "ANDITEM"}, {"ANDITEM"}},
    P["ANDITEM"] = {{"NOITEM"},  {"NOT", "NOITEM"}},
    P["NOITEM"] = {{"NOITEM", "REL", "RELITEM"}, {"RELITEM"}},
    P["RELITEM"] = {{"RELITEM", "PLUS_MINUS", "ITEM"}, {"ITEM"}},
    P["ITEM"] = {{"FACTOR"}, {"ITEM", "MUL_DIV", "FACTOR"}},
    P["FACTOR"] = {{"ID"}, {"UINT"}, {"UFLOAT"}, {"LBRACE",     "EXPR", "RBRACE"},
{"PLUS_MINUS", "FACTOR"}},
```

```
    //条件控制的表达式
    P["B"] = {{"B", "OR", "N", "BORTERM"}, {"BORTERM"}},
    P["BORTERM"] = {{"BORTERM", "AND", "N", "BANDTERM"}, {"BANDTERM"}},
    P["BANDTERM"] = {{"LBRACE",  "B",   "RBRACE"},  {"NOT",     "BANDTERM"},
{"BFACTOR", "REL", "BFACTOR"},
                     {"BFACTOR"}},
    P["BFACTOR"] = {{"UINT"},{"UFLOAT"}, {"ID"}},
    //运算符
    P["PLUS_MINUS"] = {{"PLUS"}, {"MINUS"}},
    P["MUL_DIV"] = {{"TIMES"}, {"DIVISION"}},
    P["REL"] = {{"EQU1"}, {"NOTEQU"}, {"LESS"}, {"LE"}, {"GREAT"}, {"GE"}},
    //读写语句
    P["SCANF"] = {{"SCANF_BEGIN", "RBRACE"}},
    P["SCANF_BEGIN"] = {{"SCANF_BEGIN", "COMMA",  "ID"}, {"SCANFSYM",    "LBRACE",
"ID"}},
    P["PRINTF"] = {{"PRINTF_BEGIN", "RBRACE"}},
    P["PRINTF_BEGIN"] = {{"PRINTFSYM",    "LBRACE", "ID"}, {"PRINTF_BEGIN",
"COMMA",  "ID"}};
}
```

为构建 LR（1）分析表，我们首先构建 first 集合，对于终结符来说，其 first 集合则是它本身，对于非终结符来说则需要计算，为此我们定义两个相关函数来进行计算，如下所示：

```
void Parser::First() {
    for (auto &it: terminated) // 终结符的first就是它自己
        first[it].insert(it);
    for (auto &it: nonterminated) // 非终结符的first要算
        first[it] = set<string>();
    while (1) {
        bool flag = false; // 用来判断每次是否有变化，没有就退出
        for (auto &it: P) {
            auto l = it.first; // 文法左部
            for(auto &item : it.second) {
                auto r = item; // 文法右部
                set<string> result = Getfirst(r);
                if (first[l].size() == 0 && result.size() != 0) {
                    first[l] = result;
                    flag = true; // first集合有变化
                } else {
                    for (auto &str: result) {
                        auto res = first[l].insert(str);
                        if (res.second)
                            flag = true;
                    }
                }
            }
        }
        if (!flag) break;
    }
}
```

```
set<string> Parser::Getfirst(vector<string> right) { // 计算文法产生式右部的first集合
    set<string> ans;
    int size = right.size();
    if ((right.size() == 1) && (right[0] == "SPACE")) {
        ans.insert("SPACE");
        return ans;
    }
    ans = first[right[0]];
    ans.erase("SPACE");
    bool isEmpty = true;
    for (int i = 1; i < size; i++) {
        if (first[right[i - 1]].find("SPACE") != first[right[i - 1]].end()) {
            ans.insert(first[right[i]].begin(), first[right[i]].end());
            ans.erase("SPACE");
        } else {
            isEmpty = false;
            break;
        }
    }
    if (isEmpty && first[right[size - 1]].find("SPACE") != first[right[size -
1]].end())
        ans.insert("SPACE");
    return ans;
}
```

在计算完 first 集合后，就可以开始构建 LR(1)项目集规范族，在计算之前，先进行拓广文法，之后计算得到所有项目集规范族，如下所示：

```
void Parser::GenLR1() {
    // 加入拓广文法S->PROG
    expression expre{"START", {{"PROG"}}};
    expre.dot = 0, expre.lookhead.push_back("END");
    // 构建初始项目
    closure I = GenClosure({{expre}});
    collection.push_back(I);

    bool flag = true;
    while (flag) {
        flag = false;
        vector<pair<closure, string>> newSets;
        // 对于每个项目集，用他们所有的当前待移进符号进行移进产生新的项目集并计算闭包加入规范
族中
        int cnt = collection.size();
        for (auto &one_set: collection) {
            for (const auto &symbol: Getsymbols(one_set)) {
                closure gotoResult = gotoSet(one_set, symbol);
                // 将goto得到的新的项目集唯一地加入闭包中
                // 且可知每个已经加入的项目集都是已经确定且唯一的
```

```
                auto tep = find(collection.begin(), collection.end(), gotoResult);
                if (!gotoResult.item.empty() && tep == collection.end()) {
                    newSets.push_back({gotoResult, symbol});
                    one_set.next[symbol] = cnt++;
                    flag = true;
                }
                if (!gotoResult.item.empty() && tep != collection.end()) {
                    int t = tep - collection.begin();
                    one_set.next[symbol] = t;
                }
            }
        }
        for (const auto &newSet: newSets)
            collection.push_back(newSet.first);
    }
}
```

在构建完所有项目集规范族以后，就可以进行最后的 LR（1）表的构建，为此我们需要使用到前面定义的 ACTION 和 GOTO 这两个集合，如下所示：

```
void Parser::GenLR1TABLE() {
    for (int i = 0; i < collection.size(); i++) {
        vector<actionItem> newaction(terminated.size());
        vector<int> newgoto(nonterminated.size(), -1);
        auto set = collection[i].item;
        map<string, int> next = collection[i].next;
        for (auto q = next.begin(); q != next.end(); q++) {
            vector<string>::iterator pos;
            if ((pos = find(nonterminated.begin(), nonterminated.end(), q->first))
!= nonterminated.end())
                newgoto[pos - nonterminated.begin()] = q->second;
            else if ((pos = find(terminated.begin(), terminated.end(), q->first))
!= terminated.end()) {
                newaction[pos - terminated.begin()].status = ACTION_STATE;
                newaction[pos - terminated.begin()].nextState = q->second;
            }
        }
        for (auto &it: set) {
            if (it.dot == it.right.size()) {
                if (it.left == "START")
                    (newaction.end() - 1)->status = ACTION_ACC;
                else {
                    expression tmpcan = it;
                    for (int j = 0; j < tmpcan.lookhead.size(); j++) {
                        auto pos = find(terminated.begin(), terminated.end(),
tmpcan.lookhead[j]);
                        newaction[pos - terminated.begin()].status =
ACTION_REDUCTION;
                        newaction[pos - terminated.begin()].l = tmpcan.left;
                        newaction[pos - terminated.begin()].r = tmpcan.right;
                    }
                }
            }
        }
```

```
            ACTION[i] = newaction;
            GOTO[i] = newgoto;
        }
    }
```

9.在前面我们已经完成了 LR（1）分析表的构建，接下来就可以对词法分析后得到的 Token 流进行分析，同样地，我们定义一个 Syntax 类进行语法分析。该类中除了符号表、四元式表外，还有一个属性栈，用于在语法分析时完成相应动作，如下所示：

```
class Syntax { // 语法分析
public:
    Syntax();
    void analysis(vector<string> input, vector<string> input1);
    void GenQuad(string Left, vector<string> Right, string argv); // 产生四元式
    string newtemp(string type);
    void showSymbolTable();
    void showQuadrupleTable();
    Parser parser;
    symbolTable SymbolTable; // 符号表
    QuadrupleGen QuadrupleTable; // 四元式表
    stack<attribute> attrista; // 属性栈
    int nxq = 0;
};
```

其中 analysis()和 GenQuad()是最关键的两个函数，一个是用来移进归约，另一个是归约过程中完成相关的动作，先看 analysis()函数，该函数输入是两个数组，其中 input 数组是我们在实验一得到的 Token 流，input1 数组则是输入原程序，如下所示：

```
void Syntax::analysis(vector<string> input, vector<string> input1) {
    vector<int> state; // 状态栈
    vector<string> Symbol; // 符号栈

    // 初始化
    state.push_back(0);
    Symbol.push_back("END");
    input.push_back("END"),input1.push_back("");
    input.push_back("END"),input1.push_back("");
    string str = "SPACE";
    string st, st1;
    do {
        int pos = find(parser.terminated.begin(), parser.terminated.end(), str) -
parser.terminated.begin();
```

```
        if(state.empty() || state.back() < 0 || state.back() >=
parser.ACTION.size() || pos >= parser.terminated.size()) handler();
        actionItem item = parser.ACTION[state.back()][pos];
        if (item.status == ACTION_ERROR) {
            item.nextState = parser.ACTION[state.back()]
[find(parser.terminated.begin(), parser.terminated.end(), "SPACE") -
                    parser.terminated.begin()].nextState;
            if (item.nextState != -1) {
                Symbol.push_back("SPACE");
                state.push_back(item.nextState);
                if(state.empty() || state.back() < 0 || state.back() >=
parser.ACTION.size() || pos >= parser.terminated.size()) handler();
                item = parser.ACTION[state.back()][pos];
            }
        }
```

```
        switch (item.status) {
            case ACTION_ACC:
                return;
            case ACTION_ERROR:
                handler();
            case ACTION_STATE:
                state.push_back(item.nextState);
                Symbol.push_back(str);
                str = *input.begin();
                st1 = st;
                st = *input1.begin();
                input.erase(input.begin());
                input1.erase(input1.begin());
                break;
            case ACTION_REDUCTION: {
                for (int i = item.r.size() - 1; i >= 0; i--) {
                    if (!Symbol.empty() && !state.empty() && Symbol.back() ==
item.r[i]) {
                        Symbol.pop_back();
                        state.pop_back();
                    } else {
                        handler();
                    }
                }
                Symbol.push_back((item.l));
                GenQuad(item.l,item.r,st1);
                int S = find(parser.nonterminated.begin(),
parser.nonterminated.end(), Symbol.back()) - parser.nonterminated.begin();
                if(state.empty() || state.back() < 0 || state.back() >=
parser.ACTION.size() || S >= parser.nonterminated.size()) handler();
                state.push_back(parser.GOTO[state.back()][S]);
                break;
            }
        }
```

```
    } while(!input.empty() && !state.empty());
}
```

首先定义两个数组，一个状态栈，一个符号栈，状态栈压入初始状态 0，符号栈压入结束符 END，之后在对 input 和 input1 两个数组做一些处理，即可开始语法分析。在分析过程中首先查 ACTION 表，因为 Token 流中的字符串都是终结符，如果 ACTION 表中存在，即 item.status != ACTION_ERROR，则进行相关操作（移进归约等），如果等于 ACTION_ERROR，则就先考虑插入 SPACE（空字符），然后看能不能进一步分析，如果还不能的话，就说明该程序无法分析，直接结束即可。

如果可以分析，那么就进入 SWITCH 语句中，要是处于移进状态，直接移进即可，同时还需对 input 和 input1 进行处理。要是处于归约状态，那么根据文法产生式右部进行规约，同时伴随着相关动作，在 GenQuad()函数实现。针对归约过程中的属性，我们定义一个属性栈来实现，，具体如下（若要包含所有动作十分复杂，为此选出几个具有代表性的进行说明）：

如 VARIABLE -> T ID 和 VARIABLE -> VARIABLE_1, ID

```
    else if(Left == "VARIABLE"){
        attribute att;
        if(attrista.empty()) handler();
        auto t = attrista.top(); // ID
        attrista.pop();
        if(attrista.empty()) handler();
        auto t1 = attrista.top(); // T | variable
        attrista.pop();
        int Type;
        if(t1.type == "UINT") Type = 0;
        else Type = 1;
        SymbolTable.enter(t.name,Type); // 加入符号表
        SymbolTable.Offset += t1.width;
        att.type = t1.type;
        att.width = t1.width;
        attrista.push(att);
    }
```

在实验二中，每个非终结符都对应一个属性，当进行归约时，相对应的属性同样要被弹出。

VARIABLE ->T ID

{enter(ID.name,type=T.type,offset=OFFSET);OFFSET+=T.width;VARIABLE.type=T.type;VARIABLE.width=T.width}

VARIABLE -> VARIABLE_1 , ID

{enter(ID.name,type=VARIABLE_1.type,offset=OFFSET);OFFSET+=VARIABLE_1.width;VARIABLE.type=VARIABLE_1.type;VARIABLE.width=VARIABLE_1.width;}

如 VARIABLE -> T ID，在归约时先弹出两个属性，并且在符号表中加入 ID，即上图中对应的 SymbolTable.enter(t.name, Type)，同时偏移量加上符号宽度，处理完之后再将新生成的符号入栈即可。

再如，BFACTOR -> UINT,BFACTIR -> UFLOAT,BFACTOR -> ID

```
    else if(Left == "BFACTOR"){
        attribute att;
        if(Right[0] == "UINT"){
            att.place = newtemp("UINT");
            att.type = "UINT";
            QuadrupleTable.gen("=", argv, "-", att.place);
            nxq++;
        }
        else if(Right[0] == "UFLOAT"){
            att.place = newtemp("UFLOAT");
            att.type = "UFLOAT";
            QuadrupleTable.gen("=", argv, "-", att.place);
            nxq++;
        }
        else{
            if(attrista.empty()) handler();
            auto t = attrista.top();
            attrista.pop();
            att.place = SymbolTable.lookup(t.name);
            auto T = SymbolTable.looktype(t.name);
            if (T == 0) att.type = "UINT";
            else att.type = "UFLOAT";
        }
        attrista.push(att);
    }
```

先判断具体是哪种归约情况，然后产生相应的临时变量，如果是 UINT 或 UFLOAT 归约，还需产生四元式。

10.在完成 LR（1）表构建以及分析后，还需完成 symbolTable 类和 QuadrupleGen 类的相关函数，这样才能生成对的四元式，如下所示：

```
void symbolTable::enter(string name, int type) {
    if (!se.count(name)) {
        se.insert(name);
        symbols.push_back({name, type, Offset, nextIndex++});
    } else handler();
}

string symbolTable::lookup(string name) {
    if (!se.count(name)) {
        handler();
    } else {
        for(auto &it : symbols)
            if(it.name == name)
                return "TB" + to_string(it.index);
    }
}
```

```
int symbolTable::looktype(string name) {
    if (!se.count(name)) {
        handler();
    } else{
        for(auto &it : symbols)
            if(it.name == name)
                return it.type;
    }
}
void QuadrupleGen::gen(std::string op, std::string arg1, std::string arg2,
std::string result) {
    quads.push_back({op, arg1, arg2, result});
}

void QuadrupleGen::backpatch(vector<int>&vec, int  quad){
    for(auto &it : vec)
        quads[it].result = to_string(quad);
}
```

11. 在完成整个语法分析过程且没有出现语法错误时，最后我们只需输出符号表和四元式表中的内容即可，如下所示：

```
void Syntax::showSymbolTable() {
    cout << SymbolTable.symbols.size() << endl;
    for (auto &it: SymbolTable.symbols) // 输出符号表
        cout << it.name << " " << it.type << " " << "null" << " " << it.offset <<
endl;
}

void Syntax::showQuadrupleTable() {
    int size = QuadrupleTable.quads.size();
    cout << nowtempcnt << endl;
    cout << size << endl;
    for (int i = 0; i < size; i++) {
        auto it = QuadrupleTable.quads[i];
        cout << i << ": " << "(" << it.op << "," << it.arg1 << "," << it.arg2 <<
"," << it.result << ")" << endl;
    }
}
```

至此，我们完成了语法分析的整个过程，在最后只需输出符号表以及四元式即可，而这些将用于实验三中。

实验结果展示及分析：

1. 对于如下输入程序：

```
int a,b;
{
a=0;
b=1;
c=2;
}
```

由于该程序中使用变量 c 未经定义，所以该程序存在语法错误，为此需输出 Syntax Error

2. 对于如下输入程序:

```
int a,b,c;
double x,y;
int d,e;
double z;
```

该程序不存在语法错误，只是该程序中只有定义语句，故只产生一条四元式，程序输出如下所示：

```
8
a 0 null 0
b 0 null 4
c 0 null 8
x 1 null 12
y 1 null 20
d 0 null 28
e 0 null 32
z 1 null 36
0
1
0: (End,-,-,-)
```