# 编译器前端设计

## 程序功能描述

输入一段源程序，词法分析部分将生成单词的二元式文件，SLR(1)语法分析与语义分析部分读取二元式文件，生成四元式中间代码。

## 数据结构描述

本实验的数据结构综合专题一与专题五的数据结构，如下所示，

对于词法分析部分，程序设立了一个保留字数组用以存放C语言的保留字，数组元素类型为C++的string类型。

对于语法分析部分，本实验包含如下数据结构：first集、follow集、文法集合、产生式、非终结符号集、终结符号集、SLR分析表、识别活前缀的DFA。

首先说明first集与follow集的类型是Set。在SLR.h中定义此类型如下，

typedef map<char, set<char>> Set;

这里的map是C++的映射类型，采用key-value结构存储数据，这里的含义是以非终结符号作为key去寻找此非终结符号的first集或follow集，而集合用set类型描述。

文法集合grammar的类型是NODES，定义如下，

typedef unordered\_map<char, set<string>> NODES;

这里使用了C++的无序映射类型，通过非终结符号找到其所有产生式右部，这些产生式用set<string>来存储。set<string>是一个集合类型，里面每一个元素类型均为string，即产生式的右部。

产生式结构如下所示，

//产生式

typedef struct node {

char left;//产生式左部

string right;//产生式右部

bool operator == (const struct node item) const {

return left == item.left && right == item.right;

}

}NODE;

识别活前缀的DFA类型如下所示，

//项目集规范族DFA，通过Query找项目集编号

typedef map<Query, int, compareQuery> DFA;

SLR分析表类型如下所示，

//SLR分析表, 通过Query找分析表元素

typedef map<Query, slr, compareQuery> SLR\_table;

这里的含义是通过Query变量查询SLR分析表元素slr，从中提取出移进或归约的信息。Query类型如下所示，

//通过状态与符号连接查询SLR分析表元素

typedef struct query {

int state;//状态

char ch;//符号

}Query;

slr为SLR表元素类型，如下所示，

//SLR分析表的元素

typedef struct slr {

char kind;//'r': 归约； 's': 移进； 'a': 接受; 'e': 出错

int num;//表示用第几个产生式归约或移进的状态数,-1表示接受项目，-2表示出错

}slr;

语义分析部分的数据结构包括项目集元素、项目集、四元式和符号表，如下所示，

项目集元素类型stateElem如下所示，

//项目集的元素类型

typedef struct stateElem {

bool finish;//表示是否为归约，false表示不是，true表示是

int pos;//表示圆点的位置，取值从0-gen.size(),当为gen.size()时，表示结束；当前圆点面临的符号为：gen.right[pos]

NODE gen;//产生式

int No;//表示第几个产生式,从1开始

bool operator==(const stateElem& item)const {//等于号重载

return gen == item.gen && pos == item.pos;

}

bool operator<(const stateElem& item)const {//为了对set的key进行排序

return No < item.No || (No == item.No && pos < item.pos);

}

void operator=(const stateElem& item) {//赋值运算符重载

finish = item.finish;

pos = item.pos;

gen = item.gen;

No = item.No;

}

}stateElem;

项目集state结构如下所示,

//项目集

typedef struct state {

int number;//项目编号

//以下四个成员在closure函数中完成

set<stateElem> elem;//项目

set<char> chs;//构造完毕后，项目集面临的所有符号

bool conflict1;//第一类冲突，即归约与移进并存。表示是否有冲突，false:无冲突；true:有冲突。

bool conflict2;//第二类冲突，多个归约并存

bool canShift;//表示是否有后继项目，true表示有，false表示没有。所有项目的finish交在一起若为false，canShift=true,否则为false.

bool hasReduce;//表示是否有归约式，true表示有，false表示没有

bool operator==(const state& item)const {

return elem == item.elem;

}

}state;

四元式siYuanShi类型如下所示，

//四元式

typedef struct siYuanShi {

char op;

char arg1;

char arg2;

char res;

}siYuanShi;

用于语义分析的符号表Symbol如下所示，

//符号表

typedef map<char, deque<char>> Symbol;

## 程序结构描述

## 3.1设计方法

对于词法分析部分，设计的文法如下所示，

<标识符>→字母︱ <标识符>字母︱ <标识符>数字

<无符号整数>→数字︱ <无符号整数>数字

<单字符分界符> →+ ︱－ ︱\* ︱；︱, ︱(︱) ︱{︱}

<双字符分界符>→<大于>=︱<小于>=︱<小于>>︱<感叹号>=︱<等于>=︱<斜竖>\*

<小于>→< <等于>→= <大于>→> <斜竖> →/

<感叹号>→!

<注释1>→<斜竖><斜竖>

<注释2>→<注释前半部><注释后半部>

<注释前半部>→<斜竖>\*

<注释后半部>→\*<斜竖>

<自增运算符>→++ <自减运算符>→--

<左移运算符>→<< <右移运算符>→>>

<加等于>→+<等于> <减等于>→-<等于>

<乘等于>→\*<等于> <除等于>→/<等于>

<与运算>→& <或运算>→|

<逻辑与>→&& <逻辑或>→||

<双引号>→” <单引号>→’

<结束符号>→#

保留字包括：void、int、float、double、if、else、for、do、while

此外，该语言设计成大小写不敏感；用户定义的标识符最长不超过32个字符；字母为 a-z A-Z，数字为 0-9； “/\*……\*/”和“//”(一行内)为程序的注释部分。

对于语法分析部分，

本实验遵循基于SLR(1)的语法制导翻译过程，在此基础上进行四元式的生成。总体上分为四步走：1、根据文法生成first集、follow集；2、生成文法的项目集，构造识别活前缀的DFA；3、根据DFA构造SLR(1)分析表；4、根据SLR(1)分析表进行SLR(1)语法分析，在语法分析过程中进行语义动作，同时构造四元式。 本实验文法如下所示，

规定文法的开始符号是S

G[S]:

拓广：Q->S

S->V=E

E->E+T

E->E-T

E->T

T->T\*F

T->T/F

T->F

F->(E)

F->i

V->i

综上所述，本实验设计的编译器前端是针对赋值语句的。

## 3.2程序模块

程序包括main.cpp、lex.cpp、SLR.cpp、SLR.h、lex.h、utility.cpp。SLR.cpp负责SLR(1)语法分析、语义分析；SLR.h包含语法分析与语义分析的各种函数说明；main.cpp负责读取词法分析的中间文件，传入语法分析函数进行分析；lex.h定义各种符号的宏；lex.cpp负责词法分析，包括扫描源文件、判断单词、输出结果；utility.cpp负责将符号的宏转换为对应的符号。

## 3.3函数说明

关于词法分析部分，lex.cpp包括isAlpha、isNumber、lookup、out和scan这五个函数。isAlpha函数输入一个字符判断是否为英文字母；isNumber函数输入一个字符判断是否为数字0-9；lookup函数输入一个字符串token，在保留字数组中查找token是否存在，若存在则返回保留字的宏，否则返回0；out函数传入单词类别编号、单词和中间文件，将二元组写入中间文件；scan函数传入一个源程序文件和中间文件，从源程序文件中成功识别一个单词后将其二元组表示输出到中间文件中。

在主函数中，当文件未结束时，每识别出一个单词，就将word清空，进行下一轮循环。

在lex.cpp中设置全局变量endFile、word、row和col。endFile是文件结束的标志；word用来存放一个单词；row和col用于记录扫描过程中当前所在的行与列。

重点说明scan函数，此函数按照标识符、无符号整数、其余符号三个顺序来进行单词识别。在标识符识别中，当字符为字母或数字或下划线且单词长度不超过32时，就将字符加到wod后面，若最终长度大于32则报错，否则在保留字数组reserved\_words中查找word，若查到则将类别号置为相应的保留字宏，否则置为ID即标识符的类别，作为输入到out函数中进行输出。

判断无符号整数中，若字符为0-9之间的数字则一直加到word后面，直到出现非数字的字符，此时调用out函数输出。

剩余的一类符号识别起来相对简单，用switch-case结构依次判断属于哪种符号即可，这里识别起来较为复杂的是//、/\*\*/这2种注释。首先说明//的识别方法，若当前扫描字符为/，则再向后面扫描一个字符，若仍为/表示出现了//注释，此时继续向后扫描直到遇到回车，调用out函数输出，在回车之前的字符属于注释内容，不用关心。/\*\*/的识别过程与//类似，仍然是先扫描到/符号，此时向后扫描字符若为\*，则进入循环扫描过程，期间扫描到\*时，再向后扫描一个字符若为/，表示注释闭合，调用out函数输出。

在识别过程中会出现多向前扫描一个字符的情况，以确定当前的单词。

关于语法分析部分，SLR.cpp包含getCurrent、splitGeneration、setGrammar、genFirst、genFollow、showFirst、showFollow、closure、go、genDFA、genSLRTable、semantic、SLR等重要函数，下面一一进行分析。

getCurrent、splitGeneration、setGrammar、genFirst、genFollow、showFirst、showFollow函数已在前两次实验中进行了详细的描述，具体实现参见源代码。

为了较好的说明函数的实现及各个函数之间的关系，下面将SLR.cpp中定义的所有全局变量列举出来，

Set first, follow;//first集、follow集

NODES grammar;//文法集合

NODE generations[MAX\_NUM];//顺序存放产生式，便于后续填表，下标为产生式编号

set<char> nonTerminate, Terminate;//非终结符号集, 终结符号集

SLR\_table table;//分析表

vector<int> state\_stack;//状态栈

vector<char> ch\_stack;//符号栈

state states[MAX\_NUM];//项目集合,下标为项目集编号

Symbol symbol\_table;//符号表

vector<siYuanShi> siYuanShis;//四元式集合

int result\_count = 0;//四元式个数

int stateNum = 0;//项目集个数

int generationNum = 0;//产生式个数

DFA stateDFA;//识别活前缀的DFA

WORD current = { -1,"" };

首先进行genDFA函数的说明，首先生成第一个项目，firstElem，其中的产生式来自产生式数组的首个元素。然后将其加入项目集states的首元素,传入closure函数进行闭包操作，从而形成初态项目集。之后，顺序遍历项目集数组states，首先判断当前项目集是否有后继项目(通过项目集的canShift判断),若有则遍历项目集当前所面临的所有符号，将当前项目集states[i]与符号作为参数传入go函数进行转移操作，生成新的项目集newState，若newState与项目集之前所有项目集都不同，则将newState加入states数组的末尾，并在这两个项目集之间建立DFA的边关系，否则仅建立DFA关系。

这里涉及到go与closure函数。Go函数的参数为项目集state，当前面临的符号ch。首先将转移的初步项目集归入newState中，然后对newState进行closure操作。这里要完成第一步，就要遍历state的所有项目(在s.elem中),判断项目的圆点之后的符号是否为ch，若是则加入newState。这里圆点之后的符号是通过圆点位置pos与项目里的产生式gen求得的，表达式为gen.right[it->pos](\*it为当前项目)。最后将newState传入closure函数进行闭包操作，返回newState即可。

Closure函数传入参数为项目集s。首先定义集合targetCh，用于收集当前项目集所有圆点之后的非终结符号，然后将其传入newXiangMu函数生成新的若干项目，将这些项目加入s中。接下来通过遍历s的所有项目，来设定s中的信息包括chs、hasReduce、conflict1、conflict2、canShift。只要项目集中含有归约项目，则hasReduce就为true。对于canShift，提前设立一个judgeShift,初值为true，每次judgeShift与finish进行相与操作，若所有项目的finish均为true，则judgeShift为true，否则为false。若judgeShift为true，则说明项目集中全是归约项目，canShfit为false，否则canShift为true。Conflict1指的是归约项目与移进项目并存的冲突，其值等于canShift与hasReduce相与的结果。Conflict2指的是多个归约项目并存的冲突，若归约项目数大于1，则为true，否则为false。

newXiangMu函数负责根据非终结符号找到所有圆点在最左边的项目，只需遍历产生式数组generations，从中对每一个产生式的左部与给定符号进行比对，若相同则以这个产生式组装成一个项目，加入新项目集中，以备返回。

genSLRTable函数负责生成SLR分析表，若能生成则返回true，否则返回false。首先遍历项目集，先根据当前项目的conflict标志判断是否有冲突，若无冲突，则分为规约项目与移进项目两种情况进行填表操作。若当前项目为归约项目，则首先判断是否为接受项目，通过产生式左部是否为开始符号来判断。若是接受项目，则SLR分析表元素的第一个分量为’a’表示接受，否则为’r’表示归约，同时将项目中产生式的编号赋值给第二分量，这里action[i,a]=rj,其中的a只来自产生式左部的follow集元素。若为移进项目，则SLR分析表的元素的第一分量为’s’,第二份量通过查询stateDFA来得知转移的状态数。若当前项目集是冲突项目，则需要判断冲突是否可解决，需要将两种集合进行比较，第一种集合是移进项目中圆点之后的元素集合，第二种集合所有归约项目左部的follow集，若两两不相交，则冲突可以解决，否则不能解决。这里定义一个临时变量temp，类型为vector<set<char>>，用于存放上述集合。通过遍历当前项目集的所有项目，可以形成temp，将temp传入canSolved函数进行判断，若返回true则冲突可解决，否则不可解决。

canSolved函数使用朴素的方法来判断集合之间是否两两之间无交集，即二重循环的方法，如下所示，

//判断冲突能否解决

bool canSolved(const vector<set<char>>& t) {

int len = t.size();

for (int i = 0; i < len; ++i)

for (int j = i + 1; j < len; ++j)

if (hasCommonElem(t[i], t[j]))//两个集合有交集

return false;

return true;

}

这里的hasCommonElem函数是判断两个集合之间是否有交集，方法是遍历第一个集合的所有元素，如果在第二个集合中能找到，则说明二者有交集，否则二者无交集，如下所示，

//判断两个集合是否有交集

bool hasCommonElem(const set<char>& t1, const set<char>& t2) {

for (set<char>::const\_iterator it = t1.begin(); it != t1.end(); ++it) {//每次取出t1的一个元素在t2中找，若找到则有交集

if (t2.find(\*it) != t2.end())//找到了

return true;

}

return false;//遍历完t1的所有元素也没有在t2中找到，说明无交集

}

接下来我们进行SLR(1)语法分析，首先向状态栈中推入0，符号栈中推入’#’。每一次分析的过程是将状态栈的栈顶元素与当前输入符号组成Query，去查询SLR分析表，根据第一分量的结果来进行不同的操作。若第一分量为’s’，说明当前应移进，则将输入符号入栈，将转移状态入栈，输入串后移一位；若为’r’,说明当前应进行归约，首先根据SLR表元素中记载的产生式确定句柄长度，找到归约符号，将归约符号入栈，然后同步将符号栈与状态栈的内容出栈(长度为句柄的长度), 然后以状态栈顶的状态和归约符号查询SLR分析表，将新状态入栈。注意，在这个过程中还要同步进行语义分析semantic函数的执行，我们稍后再说。若当前第一分量为’a’，说明分析成功，只需打印出栈的状态，返回true即可。

下面我们重点讲一讲语义分析函数semantic的实现，这实在是本实验的一个亮点，因为这里使用到的关于符号表的数据结构比较巧妙，为语义动作节省了不少麻烦。

我们首先来说说符号表的数据结构typedef map<char, deque<char>> Symbol。符号表在本实验的作用是给定一个非终结符号，查找它上一次归约而来的符号，比如上一次F->B，则查找F在符号表的元素应为B。如果是这样的话，那么这个Symbol设计成map<char,char>不就行了吗，为什么要用一个deque(vector也可)呢?我们先说明语义分析生成四元式的过程，再反过来说明这个trick。

Semantic函数的参数是规约符号reduceCh和句柄字符串juBing，设句柄为A->B语义分析的过程是若当前句柄涉及运算符操作，则生成四元式(op,arg1,arg2,res),否则将B.place赋值给A.place，这里的B.place是指在符号表Symbol中的值。对于本文法来讲，分为两种情况：1、句柄长度为1；2、句柄长度大于1。若句柄长度为1，则判断右部是否为终结符号，若为终结符号则直接将其赋值给reduceCh.place，否则查询右部的place将其赋值给reduceCh.place。这里引出了数据结构变化的原因，即为什么使用deque。因为存在这样一种情况，即在某个句柄归约的时候E->…，此时E.place=A。过一段时间之后，存在这样的归约情况：E->F，则此时E.place=F.place，出现的问题是之前保存的E.place为A，但是A被之后的F.place给冲掉了，导致还没有用到A符号的时候就丢失了，此后要用到A的时候用到的却是另一个符号！基于这个问题，使用deque替换掉原先的char。含义是使用非终结符号去查找符号表，得到的是从最开始到现在这个非终结符号所保存的所有place，deque尾部是最新的place，头部是最旧的place，这样当E->F要归约时，将F.place的尾部元素加入E.place的尾部，同时要将F.place的尾部元素删除，这一点很重要！因为一旦发生E->F，意味着F中保存的信息托付给了E，F此后便不会再使用这个信息，会去装载其他place。

对于产生四元式的情况，情况与上述类似，要先将juBing的两个操作数的place保存下来，然后要分别删除其末尾元素，通过newTemp函数生成新的临时符号，将其加入reduceCh的place队列，同时新产生的四元式的res仍然是这个newTemp。语义过程如下所示，

//语义分析

void semantic(char reduceCh, string juBing) {

int len = juBing.size();

if (len == 1) {

if (Terminate.find(juBing[0]) != Terminate.end()) {//句柄是终结符号

symbol\_table[reduceCh].push\_back(juBing[0]);

}

else {//句柄为非终结符号

symbol\_table[reduceCh].push\_back(symbol\_table[juBing[0]].back());

symbol\_table[juBing[0]].pop\_back();

}

}

else {

if (juBing[0] == '(') {// F →(E)

symbol\_table[reduceCh].push\_back(symbol\_table[juBing[1]].back());

symbol\_table[juBing[1]].pop\_back();

}

else {//以下情况需要产生四元式

if (juBing[1] == '=') {//A →V=E

siYuanShi siyuanshi = { '=',symbol\_table[juBing[2]].back(),'\_',symbol\_table[juBing[0]].back() };

symbol\_table[juBing[2]].pop\_back();

symbol\_table[juBing[0]].pop\_back();

siYuanShis.push\_back(siyuanshi);

}

else {// +、-、\*、/

char arg1 = symbol\_table[juBing[0]].back();

symbol\_table[juBing[0]].pop\_back();

char arg2 = symbol\_table[juBing[2]].back();

symbol\_table[juBing[2]].pop\_back();

symbol\_table[reduceCh].push\_back(newTemp());//临时变量

siYuanShi siyuanshi = { juBing[1],arg1,arg2,symbol\_table[reduceCh].back() };

siYuanShis.push\_back(siyuanshi);

}

}

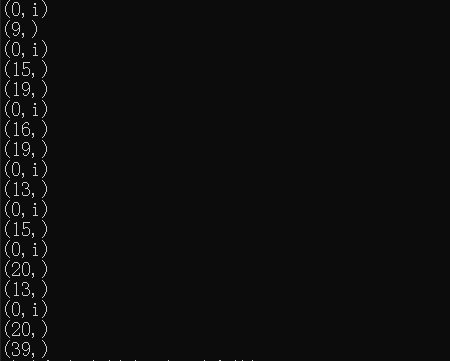
}

}

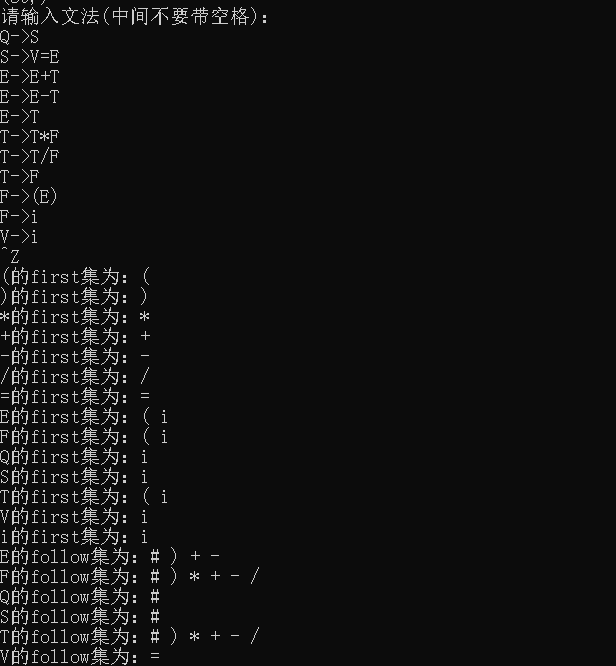
## 程序测试

输入源程序为赋值语句如下，i=i\*(i/(i+i\*i)+i)#，

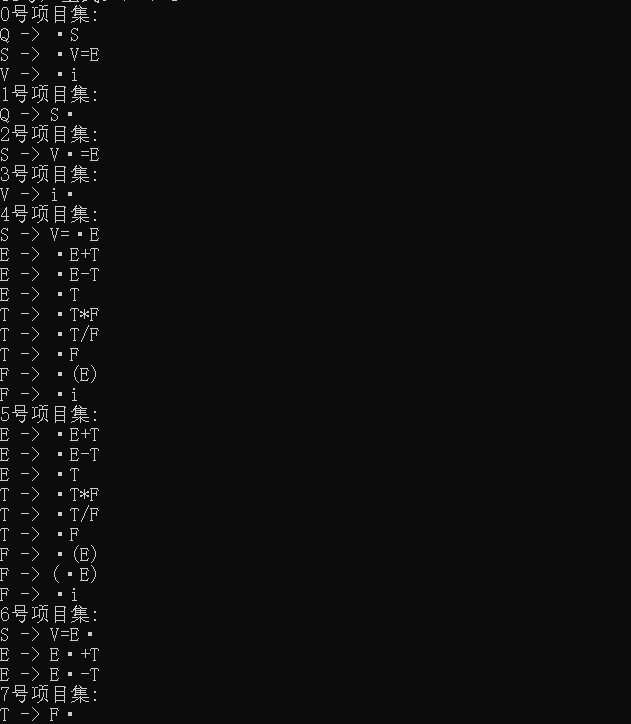
首先词法分析的二元式结果如下，

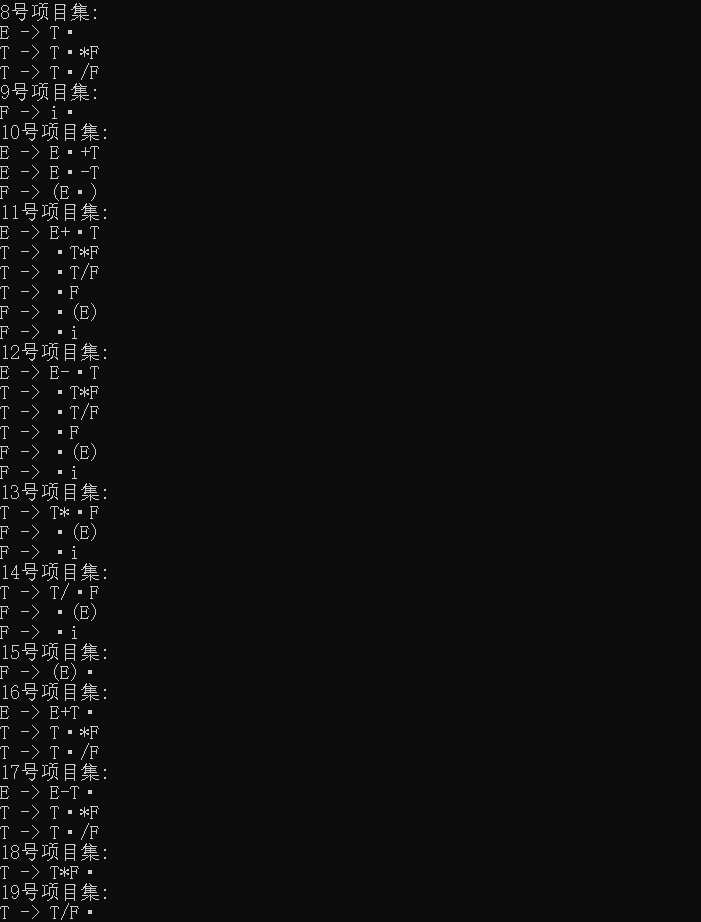


接下来要求输入文法，以便进行语法分析，如下所示，

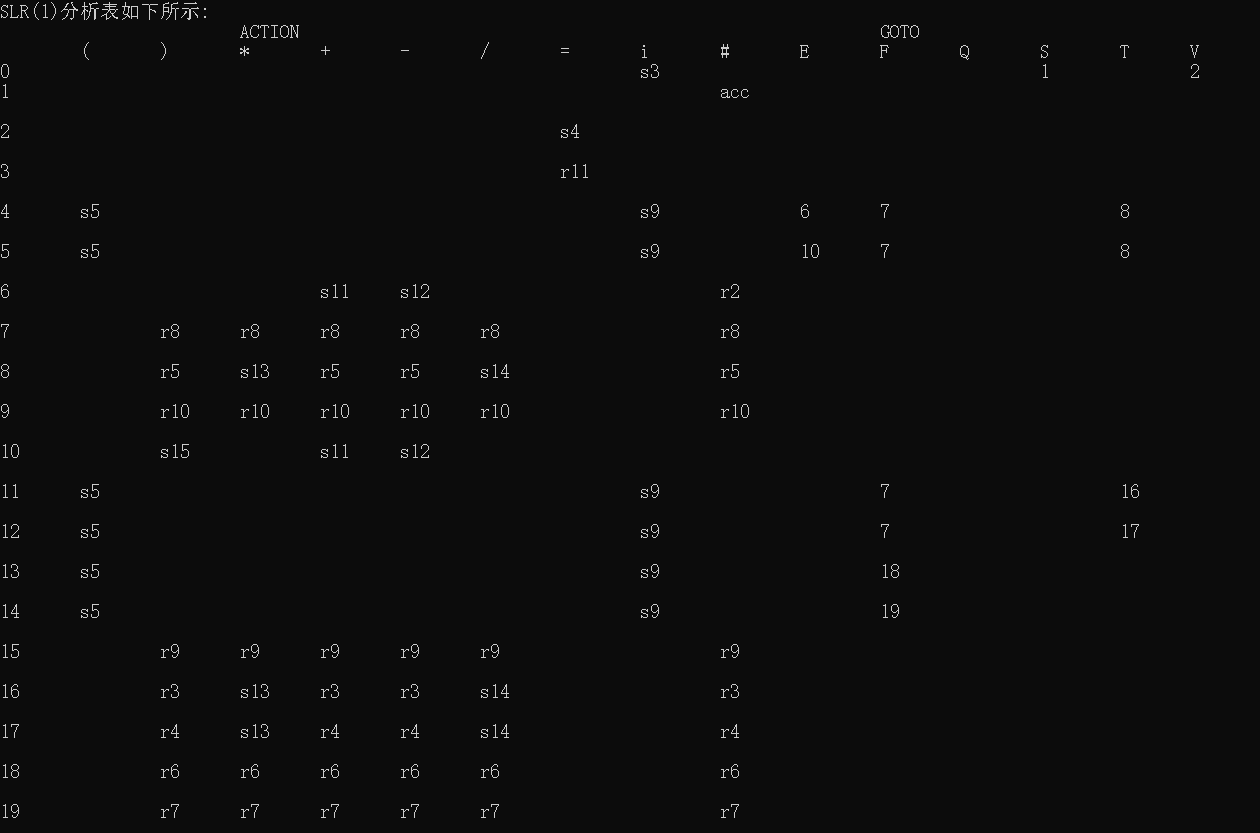


生成的项目集如下所示，





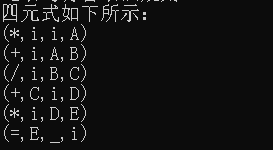
生成的SLR(1)分析表如下所示，



语法分析过程如下所示，

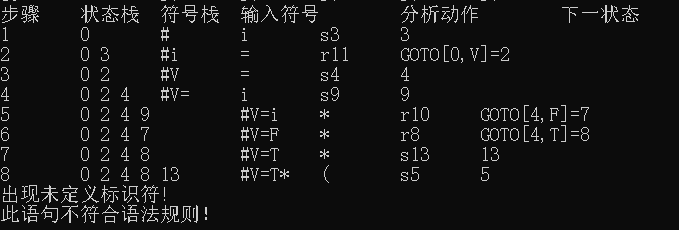


生成四元式如下所示，



第二个错误的例子如下，i=i\*(a/(i+i\*i)+i)#，

进行SLR(1)语法分析可以识别出错误，如下所示，



综上所述，本实验能够对赋值语句进行SLR(1)语法分析并生成四元式文件。但是，本实验尚有一个小问题，即语句的结束标志为’;’时，程序无法正常识别；而换为’#’时，一切正常。这一点问题目前还没有解决，但其对于整个编译器前端设计影响不大。

# 源代码附录

本实验源代码包括main.cpp、lex.cpp、SLR.cpp、SLR.h、lex.h、utility.cpp，具体见源代码目录下。