同济大学编译原理

项目报告

语法分析器设计

学院	电子与信息工程学院
学号	1751136，1653633，1752133
成员	程衍尚，贡畅，郑少宇
指导教师	卫志华

目录

1. 需求分析

1.1. 设计要求

1.1.1. 项目基本要求

要求 1：根据 LR(1)分析方法，编写一个类 C 语言的语法分析程序，可以选择以下两项之一作为分析算法的输入：

（1）直接输入根据已知文法人工构造的 ACTION 表和 GOTO 表。	
（2）输入已知文法，由程序自动生成该文法的 ACTION 表和 GOTO 表。	✔

要求 2：语法分析程序要能够调用词法分析程序，并为后续调用语义分析模块做考虑。

要求 3：对输入的一个文法和一个单词串，程序能正确判断此单词串是否为该文法的句子，并要求输出分析过程和语法树。

针对于要求 1，本小组选择较高难度的根据当场输入的文法，自动生成文法的 ACTION 表与 GOTO 表以完成最后的语法分析。

1.1.2. 小组额外要求

使用 Qt 制作图形界面，展示由类 C 语言的文法生成的 action 表和 goto 表，以及类 C 语言的词法分析结果和语法分析过程以及语法树。并实现自主定义文法并判定句子是否位于文法中。

1.2. 程序任务与主要功能

1.2.1. 词法分析功能

程序应当具有基本的词法分析功能，这是语法分析乃至整个编译器的基础模块。在分析程序语言之前，必须得将其拆分为一个一个的单词，在其基础上，还可以考虑引入如下拓展功能：

（1）增加单词（如保留字、运算符、分隔符等）的数量
（2）将整常数扩充为实常数
（3）增加出错处理功能
（4）增加预处理程序，每次调用时都将下一个完整的语句读入扫描缓冲区，去掉注解行，并能够对源程序列表打印

1.2.2. 两表构建功能

根据课程要求，本程序设计应当满足：输入已知文法，由程序自动生成该文法的 ACTION 表和 GOTO 表。程序需经受起一般 LR(1)文法的检验，得到正确的 ACTION 表与 GOTO 表。并为后续 C 语言的语法分析提供可靠的 ACTION 表与 GOTO 表。

在本小组的课程设计中，我们最终展示的是根据 C 语言文法输入，并现场计算得到的 ACTION 表与 GOTO 表。也可以给定一个新的文法，按照格式输入程序，得到新的 ACTION 表与 GOTO 表。

1.2.3. 语法分析功能

根据课程要求，语法分析程序首先要能够调用词法分析程序，并为后续调用语义分析模块做考虑。其次，对输入的一个文法和一个单词串，程序能正确判断此单词串是否为该文法的句子，并要求输出分析过程和语法树。

在本小组的课程设计中，我们实现了语法分析的完整过程，对于正确的 C 语言程序，可以得到“状态栈-符号栈-输入串”完整模拟的语法分析全过程。

1.2.4. 其他功能

代码高亮功能：显示待分析程序的行号，并对不同类型的单词采用不同颜色显示，具有一定的错误检查与拼写提示功能。在当前的程序中，我们实现了对于行号的标识功能，对于不同单词的高亮功能，我们在词法分析的时候对其作了不同颜色的区分。对于错误代码的提示，在语法分析与词法分析的时候均有考虑。

UI 界面功能：编写了一个图形界面，方便与用户作简单的交互。

1.3. 输入输出形式

1.3.1. 文法输入

首先写出类 C 语言文法的产生式（如图 x，完整版见附录 x），为了让程序能够识别我们设计的文法。将每个终结符与非终结符都分别用一个特定的 int 值予以显示，终结符从 int 值为 1 开始依次+1 递增，非终结符 int 值为 1001 开始依次+1 递增（如图,完整版见附录 7.2）。

	A	B	C	D	E
1	终结符名+A1	int值		非终结符名	int值
2	auto	1		程序	1001
3	break	2		外部声明	1002
4	case	3		函数定义	1003
5	char	4		环境声明	1004
6	const	5		类型名	1005
7	continue	6		函数名	1006
8	default	7		参数序列	1007
9	do	8		标识符	1008
10	double	9		带类型参数序列	1009
11	else	10		类型序列	1010
12	enum	11		函数声明	1011
13	extern	12		头文件声明	1012
14	float	13		[illegible]	[illegible]
15	for	14		语句块	1014
16	goto	15		语句List	1015
17	if	16		语句	1016
18	int	17		函数调用	1017
19	long	18		复合语句	1018
20	register	19		表达式语句	1019
21	return	20		选择语句	1020
22	short	21		循环语句	1021
23	signed	22		跳转语句	1022
24	sizeof	23		表达式	1023
25	static	24		[illegible]	[illegible]
26	struct	25		[illegible]	[illegible]
27	switch	26		操作语句	1026
28	typeof	27		[illegible]	[illegible]
29	union	28		三目条件赋值	1028
30	unsigned	29		二目操作	1029
31	void	30		赋值语句	1030
32	volatile	31		赋值符号	1031
33	while	32		计算语句	1032
34	=	33		计算符号	1033
35	-	34		变量定义语句	1034
36	*	35		变量定义	1035
37	/	36		待定义变量	1036

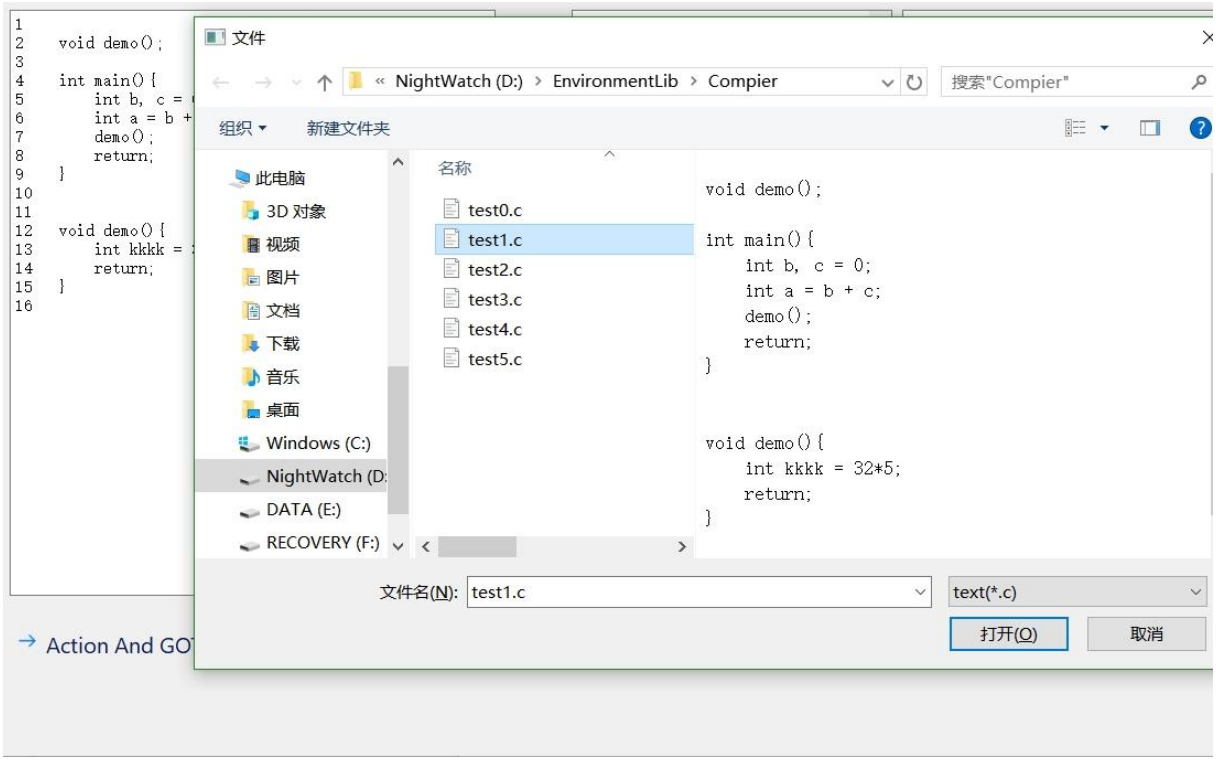
最终的产生式子，如：

<函数定义> -> <类型名> <函数名><(><带类型参数序列><)> <语句块>

被编码成{1003,{1005,1006,61,1009,62,1014}}的格式作为程序的文法输入，其编码缘由依据 2.2 节数据结构的定义。

1.3.2. 待分析语料输入

支持以（.c/）文件读入方式为程序添加分析语料。



1.3.3. 词法分析结果输出

词法分析器将源程序读入转化成单词序列以及其对应的 int 值以 pair 的形式予以输出显示。此处得到的数据也是程序的中间结果用以后续的语法分析。

1.3.4. aciton 表和 goto 表输出

action 表和 goto 表的形式如下：

```
1.    map<int, vector<pair<int, int>>>          actiontab;
2.    map<int, vector<pair<int, int>>>          gototab;
```

actiontab 为 action 表。即读入终结符时状态的移进规约情况。action.key 为分析表的第 key 个状态。action.value 表示该状态到各个其他状态的转换情况。对 pair<int,int>, 左侧的 int 转换条件，右侧的 int 表示移进规约情况；设该值为 x，如果 x 大于 0，则表示为进行移进，如果 x 小于 0，则表示以第 abs(x)文法产生式进行规约。对于不会涉及的 action 情况，不加入其中。

gototab 为 go 表，即之前状态在规约后形成非终结符，读取非终结符的跳转情况。

gototab.key 为 goto 分析表的第 key 个状态。gototab.value 表示该状态到各个其他状态的转换情况。对 pair<int,int>,左侧的 int 为非终结符的特征值，表示跳转条件，右侧的 int 表示跳转到的 action 表的状态。

1.3.5. 语法分析最终结果输出

根据小组设计的文法，当最状态为接受状态并且已规约串中只有程序时语法分析结束。

最终结果以两种类型展示：

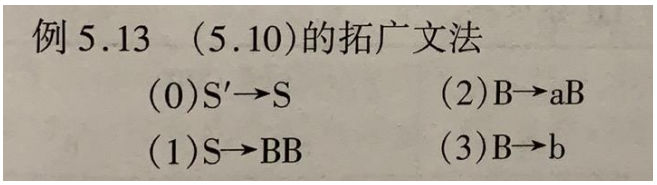
①根据输入的文法输出语法分析过程。

②根据语法分析过程构造相关语法树

1.4. 测试数据

1.4.1. “小”文法检验

对课本上（P.115,例 5.13）的小型文法产生式，生成项目集族、带活前缀的 DFA，以及相应的 ACTION 表与 GOTO 表，与书本上的最终结果进行比对。这一步在算法的控制台验证阶段完成。


例 5.13 (5.10)的拓广文法

(0)S′→S (2)B→aB

(1)S→BB (3)B→b

控制台检验结果：

```
B->b .
ACTION表
状态0          a               s3              b               s4
状态1          #               sacc
状态2          a               s6              b               s7
状态3          a               s3              b               s4
状态4          a               r3              b               r3
状态5          #               r1
状态6          a               s6              b               s7
状态7          #               r3
状态8          a               r2              b               r2
状态9          #               r2
GOTO表
状态0          S               1               B               2
状态2          B               5
状态3          B               8
状态6          B               9
```

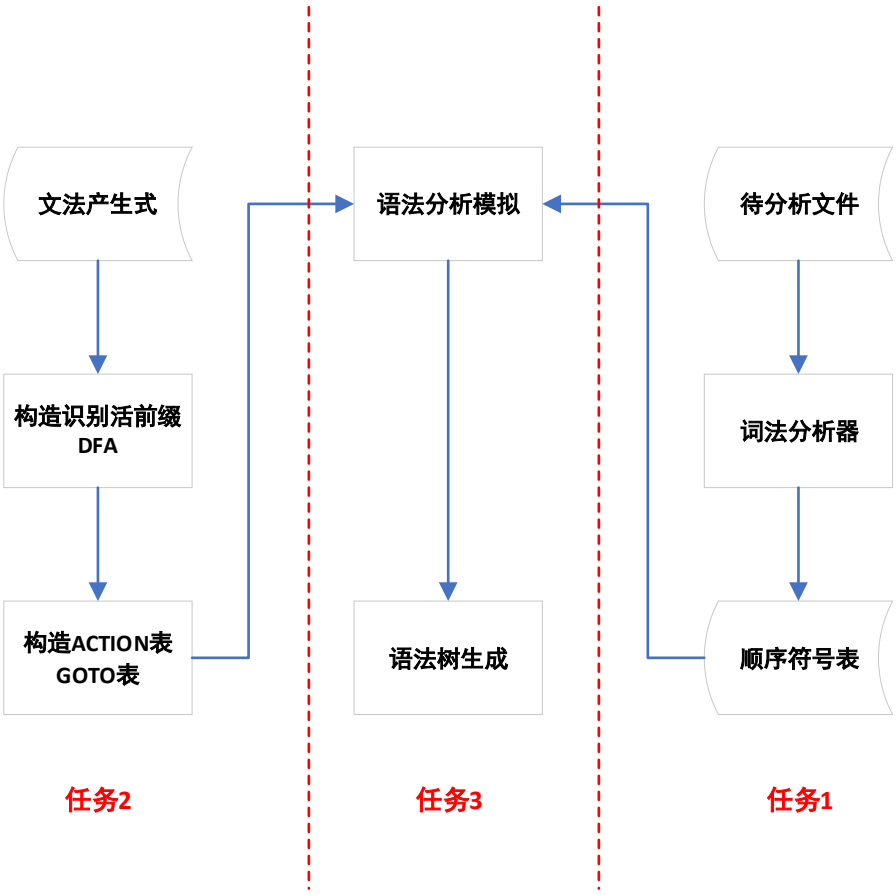
与课本标准结果比对，结果一致。成功。

1.4.2. 类 C 语言文法测试

输入附录 7.1 中的类 C 语言文法产生式表，逐条语句生成 ACTION 表与 GOTO 表。每一条语句测试完成后，整合整个表，生成 ACTION 表和 GOTO 表供后续语法分析多次调用，并读入（.c）类型的 C 语言文件，作最后的语法分析。

2. 概要设计

2.1. 子任务分解



子任务 1：待分析文件->词法分析器->顺序符号表。将待分析的源文件中的单词通过词法分析器识别并分离出来，并依据其属性按在源文件中出现的先后顺序存储。该任务在之前的课程设计中已经完成。

子任务 2：文法产生式->构造识别活前缀 DFA->构造 ACTION 表与 GOTO 表。根据文法产生式。依次获得初始文法，构造拓广文法，构造闭包，ACTION 表，GOTO 表。

子任务 3：语法分析模拟->语法树生成。依据前两个子任务的结果，编写总控程序完成完整的语法分析并生成语法树。

以上各个部分中，子任务 2 的难度最高。

2.2. 数据结构定义

2.2.1. 词法分析中的 allword 结构

```
vector<pair<string, int>> allword;
```

解释：词法分析最后的结果输出，将源程序中的每个单词分离出来并以及其意义进行编号。

2.2.2. 词法分析中的 wrong_word 结构

```
vector<pair<string, int>> wrong_word;
```

解释：词法分析的输出结果，针对的是词法分析中出错情况的收集。

2.2.3. Item 结构

```
class Item {
public:
    int                 left;
    vector<int>         right;
    int                 dotPosition;
    int                 idx;            //表示该项目的由哪一个拓展文法构造而来
    int                 expidx;         //表示该项目在所有在项目集中的位置
    bool                isItem;         //判断是否是拓展文法即是否是项目集
    int                 size;           //获取当前文法的右侧 size
    Item(int l, vector<int> r, int pos, bool sign);
    vector<int> expDot();                                   //获得所有 LR(0)项目
    Item* getExpend(int, int);                              //获得单个项目
    bool equals(const Item&);                               //重载等号运算符
};
```

解释：该数据结构将拓展文法和 LR(0)项目整合到了一起，同时表示两者，LR(0)加上项目加上一个非终结符可构成 LR(1)项目。当 isItem 为 false 时，该数据结构表示为拓展文法，可以通过 expDot 函数获得当前文法产生的所有 LR(0)项目。equals 函数通过判断各个类中参数值确定是否相等。

2.2.4. CFG 结构

```
class CFG
{
public:
    set<int>                            terSymbol;          //终结符
    set<int>                            unTerSymbol;        //非终结符
    vector < pair<int, vector<int>>>    gram;           //初始表示文法
    vector<Item*>                       initalItems;    //初始文化
    vector<Item*>                       expendItems;        //拓展文法
    unordered_map<int, set<int>>            firstSet;   //计算单个符号的 First 集
    CFG() {}
    CFG(set<int> terminal, set<int> unterminal, vector <pair<int, vector<int>>>g) {
        this->terSymbol = terminal;
        this->unTerSymbol = unterminal;
```

```
        this->gram = g;

    }
    void getInitItems();                                    //获取初始文法
    void getExpedItems();                                   //获得拓展文法
    set<int> getFirstSet(int num,set<int>);                 //计算每个符号
的 First 集
    set<int> calFirstSet(vector<int>);                      //计算符号串的 First 集
};
```

解释：

参数解释：

该结构表示上下文无关文法。t

terSymbol	表示终结符。
unTerSymbol	表示文法的非终结符。
gram	表示初始表示文法。
initalItems	表示由初始表示文法生成的初始文法。
expendItems	表示整个上下文无关文法的所有 LR(0)项目。
firstSet	为哈希表，key 为终结符和非终结符，value 为该符号的 first 集。

函数解释：

getInitItems():	根据输入的文法获得初始文法集
getExpedItems():	根据输入的文法获得所有项目。
getFirstSet()	计算输入的符号的 First 集
calFirstSet()	计算输入的符号集的 First 集

2.2.5. Closure 结构

```
class Closure
{
public:
    vector<pair<Item*, int>>family;                         //当前集族的闭包
    int       initalSize;
    void buildFamilySign(Item*, int, CFG&, bool);           //根据传递来的数据构造闭
包
    void buildFamily(CFG&);
    bool isInClosure(Item*, int);                           //判断当前 LR(1)项目是否在该闭包
内
    Closure(vector<pair<Item*, int>> F) {
        this->family = F;
        this->initalSize = F.size();
    }
};
```

解释：

该结构表示 LR(1)中的闭包。

成员变量：

initalSize: 初始项目的个数。（初始项目：作为基础构造整个函数闭包）
family: 当前闭包
family.first: LR(0)文法
framily.second 展望的符号。即 LR(0)+一个展望符号构成 LR(1)
成员函数：
void buildFamilySign(Item*, int, CFG&, bool) 求取一个 LR(1)项目的闭包，将其加入到整体闭包中。
isInClosure(Item* ,int) 判断参数表示的 LR(1)是否位于当前闭包中。
void buildFamily(CFG&); 根据初始项目构造整体闭包

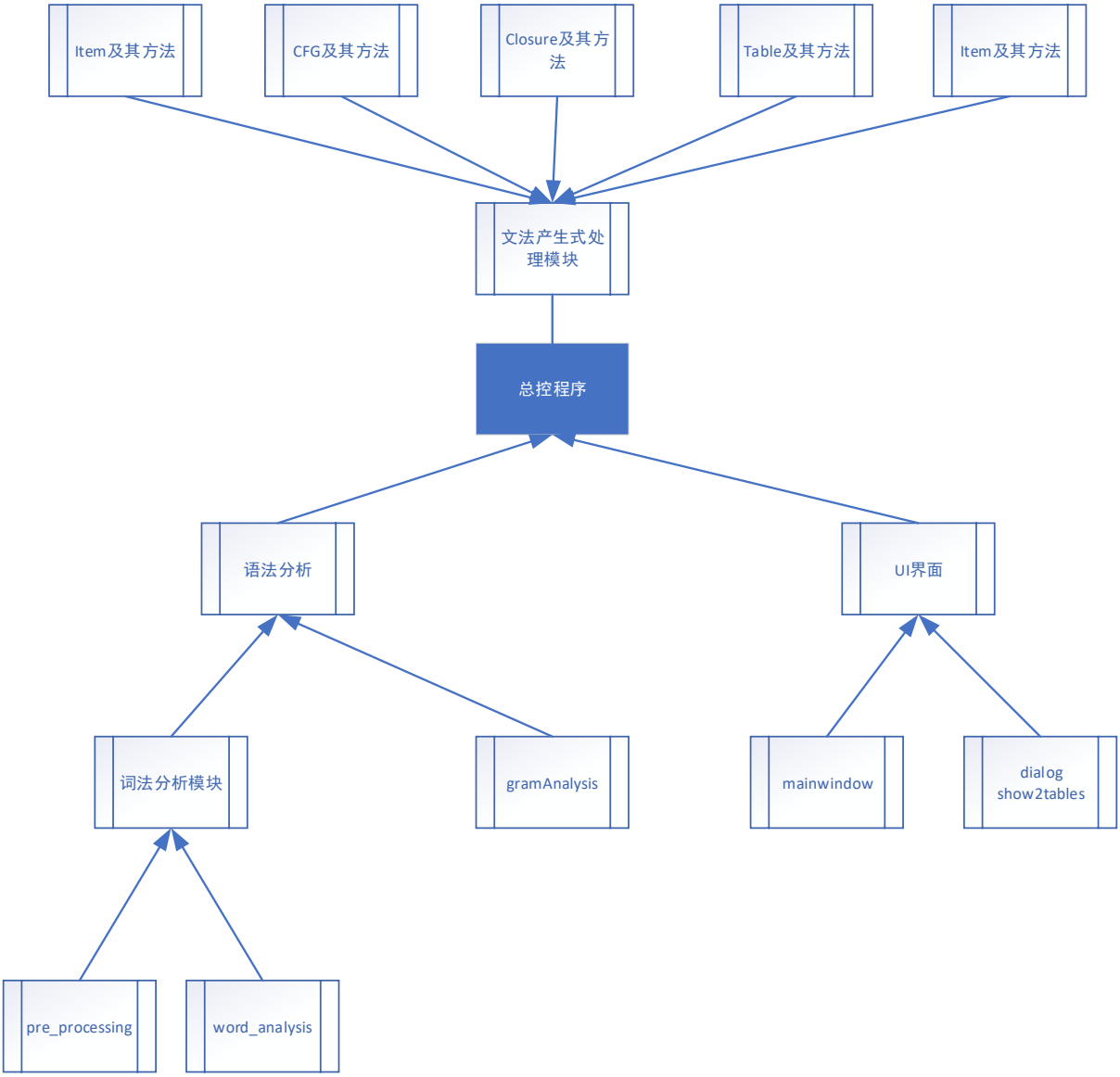
2.2.6. Table 结构

```
class table {
public:
    CFG cfggram;                                    //生成文法
    vector<Closure>  cfgClosures;                   //生成闭包
    map<int, vector<pair<int, int>>> actionetab; //ACTION 表，表示跳转，正数表示跳转到一定位置，负数表示使用文法进行规约。1W 表示全部 OK。
    map<int, vector<pair<int, int>>> gototab;  //goto 表，正数表示跳转到一定位置，0 表示啥也不干。
    void buildTable(); //生成 Action 表和 goto 表
    table(CFG c) {
        this->cfggram = c;
    }
};
```

解释：
Table：为了方便 action 和 goto 表而构造的类。
成员变量：
cfggram： 产生 action 表和 goto 表的文法。
cfgClosures： 产生文法生成的所有闭包。
map<int, vector<pair<int, int>>> actionetab;
actiontab 为 action 表。即读入终结符时状态的移进规约情况。action.key 为分析表的第 key 个状态。action.value 表示该状态到各个其他状态的转换情况。对 pair<int,int>,左侧的 int 转换条件，右侧的 int 表示移进规约情况；设该值为 x，如果 x 大于 0，则表示为进行移进，如果 x 小于 0，则表示以第 abs(x)文法产生式进行规约。对于不会涉及的 action 情况，不加入其中。
map<int, vector<pair<int, int>>> gototab;
gototab 为 go 表，即之前状态在规约后形成非终结符，读取非终结符的跳转情况。gototab.key 为 goto 分析表的第 key 个状态。gototab.value 表示该状态到各个其他状态的转换情况。对 pair<int,int>,左侧的 int 为非终结符的特征值，表示跳转条件，右侧的 int 表示跳转到的 action 表的状态。
成员函数：
void buildTable(); 根据文法和闭包生成 action 表和 goto 表。

2.3. 主程序流程与模块间调用关系

对总控程序而言，每一次语法分析

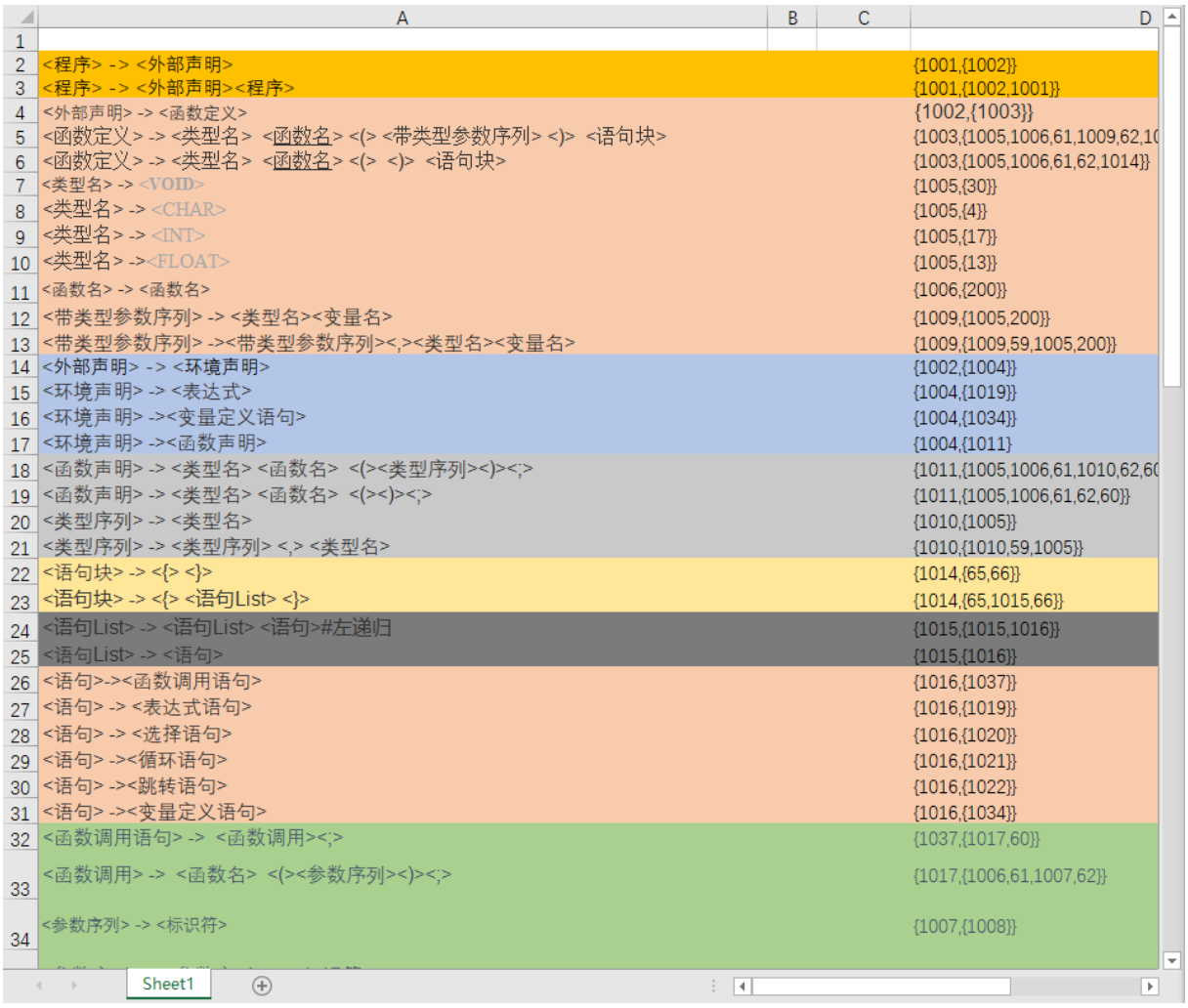


总控程序首先调用文法产生式处理模块，将输入的文法产生式转化为 Action 表与 Goto 表；然后总控程序调用语法分析模块，该模块首先调用词法分析子模块将输入的源程序拆分为一个一个的单词，之后再调用 gram_Analysis 完成语法分析的工作。整个过程通过 UI 界面与用户进行交互。

3. 详细设计

3.1. 类 C 语言文法构造

只截取部分图，其余见附件。

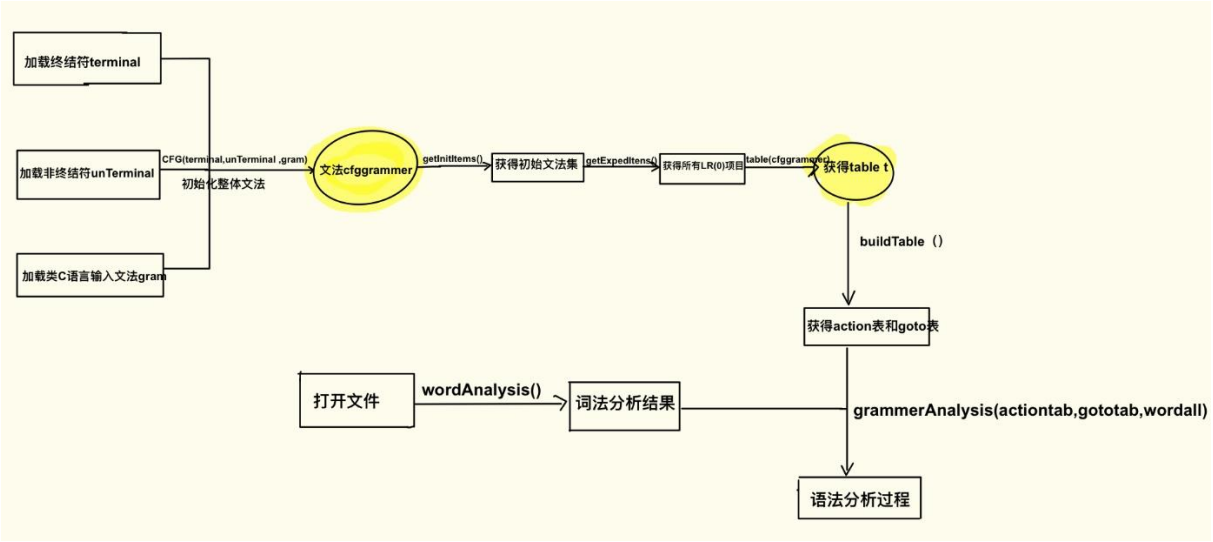


	A	B	C	D
1				
2	<程序> -> <外部声明>			{1001,{1002}}
3	<程序> -> <外部声明><程序>			{1001,{1002,1001}}
4	<外部声明> -> <函数定义>			{1002,{1003}}
5	<函数定义> -> <类型名> <函数名> <(> <带类型参数序列> <)> <语句块>			{1003,{1005,1006,61,1009,62,1
6	<函数定义> -> <类型名> <函数名> <(> <)> <语句块>			{1003,{1005,1006,61,62,1014}}
7	<类型名> -> <VOID>			{1005,{30}}
8	<类型名> -> <CHAR>			{1005,{4}}
9	<类型名> -> <INT>			{1005,{17}}
10	<类型名> -><FLOAT>			{1005,{13}}
11	<函数名> -> <函数名>			{1006,{200}}
12	<带类型参数序列> -> <类型名><变量名>			{1009,{1005,200}}
13	<带类型参数序列> -><带类型参数序列><,><类型名><变量名>			{1009,{1009,59,1005,200}}
14	<外部声明> -> <环境声明>			{1002,{1004}}
15	<环境声明> -> <表达式>			{1004,{1019}}
16	<环境声明> -><变量定义语句>			{1004,{1034}}
17	<环境声明> -><函数声明>			{1004,{1011}
18	<函数声明> -> <类型名> <函数名> <(><类型序列><)><;>			{1011,{1005,1006,61,1010,62,6
19	<函数声明> -> <类型名> <函数名> <(><)><;>			{1011,{1005,1006,61,62,60}}
20	<类型序列> -> <类型名>			{1010,{1005}}
21	<类型序列> -> <类型序列> <,> <类型名>			{1010,{1010,59,1005}}
22	<语句块> -> <{> <}>			{1014,{65,66}}
23	<语句块> -> <{> <语句List> <}>			{1014,{65,1015,66}}
24	<语句List> -> <语句List> <语句>#左递归			{1015,{1015,1016}}
25	<语句List> -> <语句>			{1015,{1016}}
26	<语句>-><函数调用语句>			{1016,{1037}}
27	<语句> -> <表达式语句>			{1016,{1019}}
28	<语句> -> <选择语句>			{1016,{1020}}
29	<语句> -><循环语句>			{1016,{1021}}
30	<语句> -><跳转语句>			{1016,{1022}}
31	<语句> -><变量定义语句>			{1016,{1034}}
32	<函数调用语句> -> <函数调用><;>			{1037,{1017,60}}
33	<函数调用> -> <函数名> <(><参数序列><)><;>			{1017,{1006,61,1007,62}}
34	<参数序列> -> <标识符>			{1007,{1008}}

Sheet1

3.2. 程序设计实现思路

3.2.1. 总体思路：



分析整个程序流程。首先根据终结符，非终结符和类 C 文法生成整体文法 CFG(grammer)。之后，通过 getInitItems()获得初始文法集，之后通过 getExpedItems()获得所有的 LR(0)项目。在做完这两个流程后，就可以通过 table(grammer)直接构造表类 table t,之后调用生成表函数 buildTable()获得 action 表和 goto 表。在获得 action 表和 goto 表之后，就可以通过 grammerAnalysis 函数进行语法分析得到结果。整体流程如图，以下对各个函数分别对其实现流程以及实现做出判断。并给出函数关系调用表。

3.2.2. getInitItems()函数分析

函数功能：根据初始文法获得拓展文法。

函数代码：

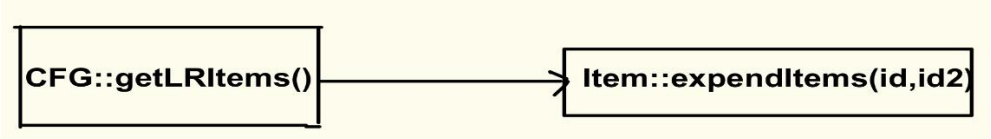
```
void CFG::getInitItems() {
    for (auto& p : gram) {
        Item* inital = new Item(p.first, p.second, 0, false);
        this->initalItems.push_back(inital);
    }
}
```

函数解释：该代码较为简单，就是将输入的文法转换为初始文化即可。然后转入到 initialItems 中保存。

3.2.3. getExpedItems()函数分析

函数功能：根据拓展文法获得所有的 LR(0)项目，以便生成 LR(1)项目。

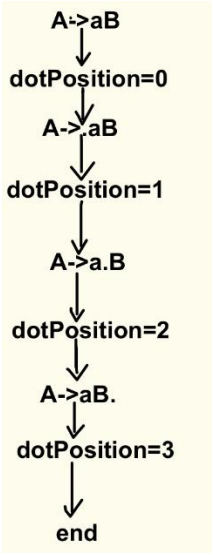
函数调用关系：



函数代码：

```
void CFG::getLRItems() {
    int idx = 0, idx2 = 0;
    for (auto& item : this->initalItems) {
        while (item->dotPosition != item->size) {
            Item* tmp = item->getExpend(idx, idx2++);//生成 LR(0)项目
            this->expendItems.push_back(tmp);//LR(0)项目加入到总集中
        }
        idx++;//下一条拓展文法
    }
}
```

其中 item::getExpend(idx,idx2)由当前拓展表示产生一个项目，该过程为循环的，当拓展文法产生项目的 dotPostion 到达尾端时表示该拓展文法已产生所有能产生的项目。以 item A->aB 这一拓展为例：函数流程如下:



dotPostion 指的是拓展文法中点的位置，该拓展文法依次生成 LR(0)项目，直到 dotPostion 走到尽头。

函数核心变量:

idx: 生成某个表示 LR(0)项目的拓展文法的序号

idx2: 该 LR(0)项目在所有项目中的序号

定义这两个变量是为了之后的项目集生成以及表的构造。

定义 idx 后，当我们遇到规约的时候，直接访问 idx 可以获得规约的文法序号。

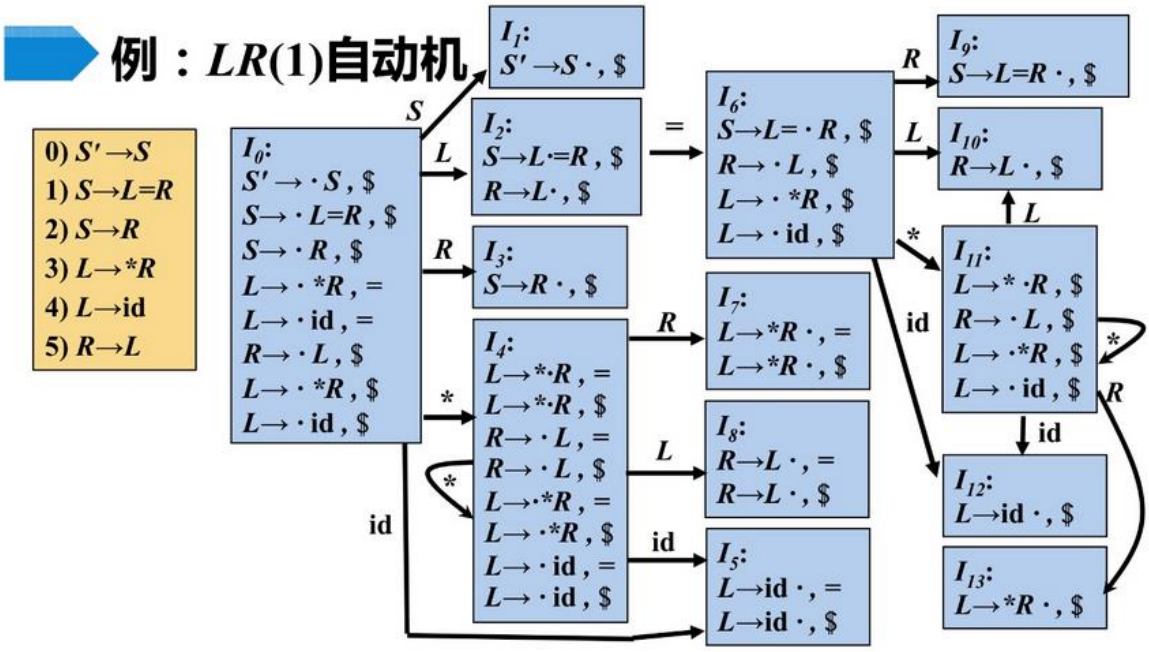
定义 idx2 后，在项目集的构造中，寻状态转换函数 GO(I,X)=COURSE(J)的中 J 的定义为:

J={任何形如 A->α X • β 的项目|A->α X • β 属于 I}

所以我们已知 A->α X • β 这一项目 p 时，直接通过 expendItems[expidx + 1] 获得 A->α X • β 。

3.2.4. buildTable() 函数完全分析

这个函数是一个非常复杂的函数，调用的函数很多，而且自身逻辑也比较复杂。我们小组在实现的时候以一个简单文法为例逐步构造的。下面给出这个简单文法以及以其来构造。

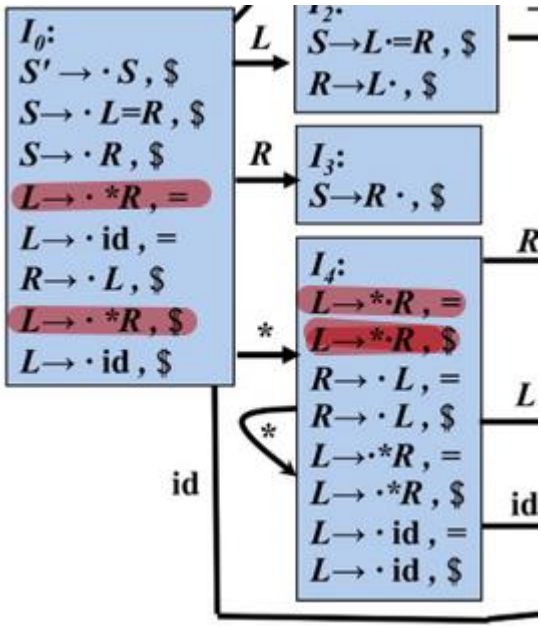


对于该过程，我们在之前的操作中已经获得了初始文法以及全体 LR(0)项目。首先，应当构造的 LR(1)项目集是 I0。其应该由[S'-> • S,#]这一 LR(1)初始项目集构造。采取的做法为，根据[S-> • S,#]构造一个项目集 Closure ifir，然后调用该项目集中的 buildFamily 方法获得进行初始项目集构造。所以研究的函数转到 buildFamily 函数中。

(1)Closure::buildFamily(CFG&)

函数功能：根据初始项目集构造闭包。

函数解释：注意是根据初始项目集来构造的闭包，而不是初始项目。因为构造闭包时初始项目集可能不止一个。举例，上图中，I0 遇到终结符*形成初始 LR(1)项目：[L->* • R,=][L->* • R,#]



如上图所示。此时在构建 I4 时我们就要根据两个初始项目来构造。所以是根据初始项目集来生成闭包。

函数代码：

```
void Closure::buildFamily(CFG& gram) {
   for (int i = 0; i<initalSize; i++) {
        if (family[i].first->dotPosition != family[i].first->right.size() - 1) {
            buildFamilySign(family[i].first, family[i].second, gram, false);
        }
   }
}
```

函数说明：正如上图所示，我们需要对初始项目集进行构造，所以要记录初始项目集的个数，即 initalSize。调用 buildFamilySign 对单个项目进行构造，由于当·的位置在最后的时候为规约，不需要对该项目继续进行构造闭包，所以要加一层判断。

所以我们来到了 buildFamilySign 函数进行对单个项目进行闭包构造。

(2)Closure::buildFamilySign (Item* init, int initNum, CFG& gram, bool judge)

函数功能：构造单个项目的闭包。

理论依据：若项目[A->α·Bβ,a]属于 CLOSURE(I),B->·γ是一个产生式 那么我们可以对于 First(βa)中每个非终结符 b,如果[B->·γ,b]原来不在 CLOSURE(I)中，则把它加进去。

函数解释：给定一个项目，对该项目闭包构造的时候，依照理论依据构造。

函数方法：回溯+记忆化搜索。

函数代码：

```
void Closure::buildFamilySign(Item* init, int initNum, CFG& gram, bool judge)
{
```

```
    if (judge&&isInClosure(init, initNum)) {
        return;
    }//判断之前是否出现过，若已经出现过，则跳过
    if (judge && (init->dotPosition == init->right.size() - 1 ||
gram.terSymbol.count(init->right[init->dotPosition + 1]))) {
        this->family.push_back({ init,initNum });
        return;
    }//判断是否是规约语句，或·之后的符号是否为终结符，若是直接跳过返回。
    if (judge) {
        this->family.push_back({ init,initNum });//否则，加入到闭包中。
    }
    int nextWait = init->right[init->dotPosition + 1];
//表示dot后面的第一个非终结符
    vector<int> follow;                    //为非终结符后面的符号集合
    int beginPos = init->dotPosition + 2;
    for (int i = beginPos; i<init->right.size(); i++) {
        follow.push_back(init->right[i]);
    }
    follow.push_back(initNum);
    set<int>first = gram.calFirstSet(follow);//寻找后序集合的First集
                                                //寻找以dot后面开头的非终结符的文法并与first中的文法组合
    for (auto p : gram.expendItems) {
        if (p->left == nextWait&&p->dotPosition == 0) {
            for (auto num : first) {
                buildFamilySign(p, num, gram, true);
            }
        }
    }
}
```

核心变量：bool judge vector<int>follow

变量解释：judge 表示当前进行构造的项目是否属于初始项目集，当为 false 时表示该项目属于，否则为不属于。

follow 为取得的非终结符之后的符号集和 LR(1)的展望符号的集合。然后，调用 gram.calFirstSet(follow)求得该符号集合的 First 集。此时需要用到求符号串的 First 集函数，下面给出解释。之后再寻找满足理论依据的 LR(1)文法，继续递归构造。直到所有可构造的完成，集合不再增加。

(3)CFG::calFirstSet(vector<int> nums)

函数功能：求取符号串的 First 集。

函数思想：首先调用 getFirstSet(int)求得第一个符号的 First 集，如果第一个符号的 First 集中有空，先保存第一个符号的 First 集，则继续求第二个符号的 First

集，直到结束或者遇到不为空的情况。

核心代码：

```
set<int> CFG::calFirstSet(vector<int> nums) {
    set<int>res;
    int befIdx = -1;
    int idx = 0;
    while (idx<nums.size() && idx != befIdx) {//当扫描所有符号或不再存在空符
号
        befIdx = idx;
        set<int>tmp = getFirstSet(nums[idx], {nums[idx]});
        for (auto& n : tmp) {
            if (n == -1) {
                idx++;
                res.insert(-1);
            }
            else {
                res.insert(n);
            }
        }
    }
    return res;
}
```

(4)set<int> CFG::getFirstSet(int num,set<int>allnum)

函数功能：求取单个符号的 First 集。

核心算法：递归+记忆化

函数解释：在文法中，求解过程中往往会遇到直接左递归以及间接左递归文法的情况。

对这两种方法求解 First 集较为麻烦。

以左递归为例：

A->Aa

A->a

处理方法为：不去对 A->Aa 进行递归求取集合。因为这样会导致栈溢出。

循环文法：

A->Bb

A->d

B->Cd

C->A

C->e

若不处理，同样会无限的递归下去，因为 A->B,B->C,C->A 直观上看不含直接左递归，但是存在间接左递归。因此需要及退出。采取的方法为：在每次递归求取 First 集

之前，先将文法产生条件加入到 allnum 中,每次递归之前先确定是否要进入的符号已经出现过。比如 A->B,B->C,C->A，A->B 时，将 A 插入到集合中，B->C 时，将 B 插入到集合中，在 C->A 时，在 allnum 中已经出现过该符号，此时不再递归。同样也要处理空产生式，方法同上。

函数代码：

```
set<int> CFG::getFirstSet(int num,set<int>allnum) {
    if (!terSymbol.count(num) && !unTerSymbol.count(num)) {
        return{};
    }
    set<int>res;
    if (firstSet.count(num)) {
        return firstSet[num];
    }//记忆化搜索
    if (terSymbol.count(num)) {
        res.insert(num);
        firstSet.insert({ num,res });//处理终结符的first集
        return res;
    }
    else {
        for (auto& item : initalItems) {
            if (item->left == num) {
                int idx = 1;
                for (int i = 0; i<idx; i++) {
                    if (item->right.size() == 1 && item->right[0] == -1) {
                        res.insert(-1);
                    }//表示当前情况下只有一个空
                    else {
                        if (!allnum.count(item->right[i])) {
                            allnum.insert(item->right[i]);
                            set<int>tmp = getFirstSet(item->right[i],allnum);
                            for (int n : tmp) {
                                if (n == -1) {
                                    idx++;//表示当前情况下含有空，则First集可以后推
                                }
                                res.insert(n);
                            }
                        }
                    }
                }
            }
        }
        this->firstSet.insert({ num,res });
```

```
    }
    return res;
}
```

(5)buildTable()

在完成由基本的初始项目集构造整个项目集后，就可以看来 buildTable()这个函数了。

函数核心思想：通过广度优先搜索和 map 实现。

函数代码逐条解析：

```
vector<pair<Item*, int>> initClosuer = { { this->cfggram.expendItems[0],-2 } };
//表示初始项目集，即[S'-> · S,#]
Closure ifir(initClosuer);
//根据初始项目集初始化闭包
ifir.buildFamily(this->cfggram);
//闭包构建
this->cfgClosures.push_back(ifir);
//将 I0 添加到闭包集合
int idx = 0;
//当前的闭包在闭包集合中的序号
queue<Closure> dfsClosure;
//构建闭包队列，由于广度优先搜索构成闭包集合
```

广度优先搜索构建 action 表和 goto 表

搜索思路：首先取出队列的首元素闭包 tmp，然后求得对闭包中的 LR(1)项目进行遍历。在搜索之前先建立映射表 map<int, vector<pair<Item*, int>>> m。int 表示转换条件，即符号。如[A-> · ab,#]转换到[A->a · b,#]转换条件为 a。

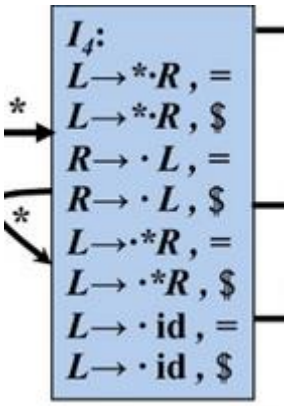
vector<pair<Item*,int>>表示转换后的初始项目集合。

LR(1)项目的组成为pair<Item*,int>。

假设有LR(1)项目p,如果p.first->dotPosition == p.first->right.size() – 1，则说明该项目为规约项目，则改变action表，actionetab[idx].push_back({ p.second,-p.first->idx });idx表示当前的项目闭包序列，p.second表示规约条件，-p.first->idx在之前已经描述过，表示产生该项目的初始文法序号。则完成。此时需要特殊判断是否是acc的情况。acc的情况置初始文法序列为100000，表示为acc。

如果p.first->dotPosition不满足上述条件，则说明需要转换，将 · 右移得到新的LR(1)，则可直接求得。将其加入到m映射表中。

在求得m表后，根据m表映射后的结果获得各个初始项目集合可以生成各个项目集。但是会有重复的项目集出现，比如以一开始的表为例子

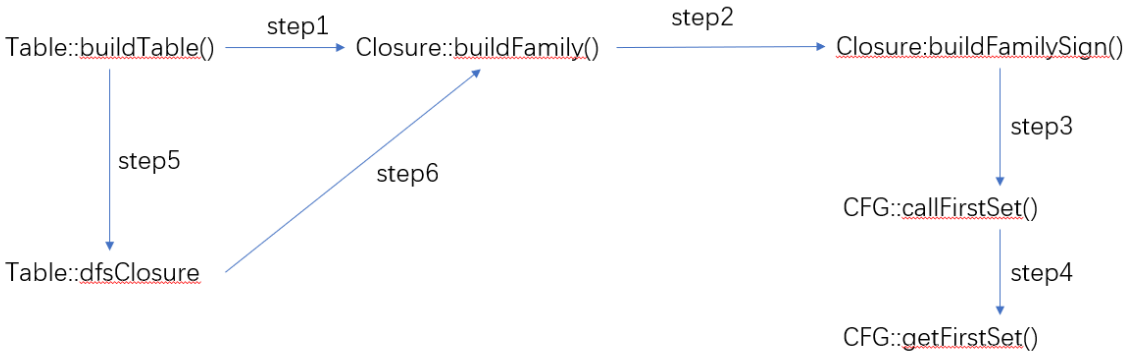


I4遇到*之后构造的闭包和原来的闭包一致，则可以不必再生成新的闭包直接指向自己即可。因此需要判断是否已经存在与本次生成闭包相同的闭包，避免重复生成相同的状态。若未找到已有状态与新闭包状态一致，则目标转移状态为新闭包状态。当为新状态闭包时，将该状态闭包加入到队列bfsClosure中和闭包集cfgClosure中。
找到状态转换条件和目标转移状态后，根据转换条件是终结符或是非终结符构造aciton表和goto表。
当队列为空时，表示不再有新的状态产生DFA构造完成。
至此ACTION表和GOTO表构造完成。

```
    while (!dfsClosure.empty()) {
        Closure tmp = dfsClosure.front();
        dfsClosure.pop();
        map<int, vector<pair<Item*, int>>> m;//表示本次生成的NEXT文法,int表示转换条件，num为接受
        for (auto& p : tmp.family) {
            if (p.first->dotPosition == p.first->right.size() - 1) {//表示不必移进接受即可
                if (p.second == -2 && p.first->left == 1 && p.first->right[0] == bg) {//表示为acc的情况
                    actionetab[idx].push_back({ p.second,10000 });//表示规约，按照第i个文法
                }
                else {
                    actionetab[idx].push_back({ p.second,-p.first->idx });
                }
            }
            else {
                int idex = p.first->dotPosition + 1;//idex表示位置
                int t = p.first->right[idex];//表示.之后的符号
                m[t].push_back({ this->cfggram.expendItems[p.first->expidx + 1],p.second });//获取后面的的CLOSURE
            }
        }
        for (auto&p : m) {//p.second=vector<pair<Item*,int>>表示转换后的情况
```

```
                bool judge = false;
                Closure newClo(p.second);//新项目集初始化
                newClo.buildFamily(this->cfggram);//构造新的项目集
                for (int i = 0; i < cfgClosures.size(); i++) {
                    if (cfgClosures[i].family.size() == newClo.family.size()) {//此时表示
已经有项目集产生了，只需要加入其中便可.
                        int j = 0;
                        for (j = 0; j<newClo.family.size(); j++) {
                            if (!cfgClosures[i].isInClosure(newClo.family[j].first,
newClo.family[j].second)) {
                                break;
                            }
                        }
                        if (j == newClo.family.size()) {
                            if (this->cfggram.terSymbol.count(p.first)) {//表示为终结符
                                actionetab[idx].push_back({ p.first,i });//表示进行移进
                            }
                            else {
                                gototab[idx].push_back({ p.first,i });
                            }
                            judge = true;
                            break;
                        }
                    }
                }
                if (!judge) {//没有找到匹配的项目集，需要构造新的项目
                    dfsClosure.push(newClo);
                    cfgClosures.push_back(newClo);
                    if (this->cfggram.terSymbol.count(p.first)) {
                        actionetab[idx].push_back({ p.first,cfgClosures.size() - 1 });//表
示此时为最后一个
                    }
                    else {
                        gototab[idx].push_back({ p.first,cfgClosures.size() - 1 });
                    }
                }
            }
            idx++;
        }
}
```

函数调用流程图：



4. 调试分析

4.1. 测试词法分析调用程序

4.1.1. 测试数据

测试集 1：

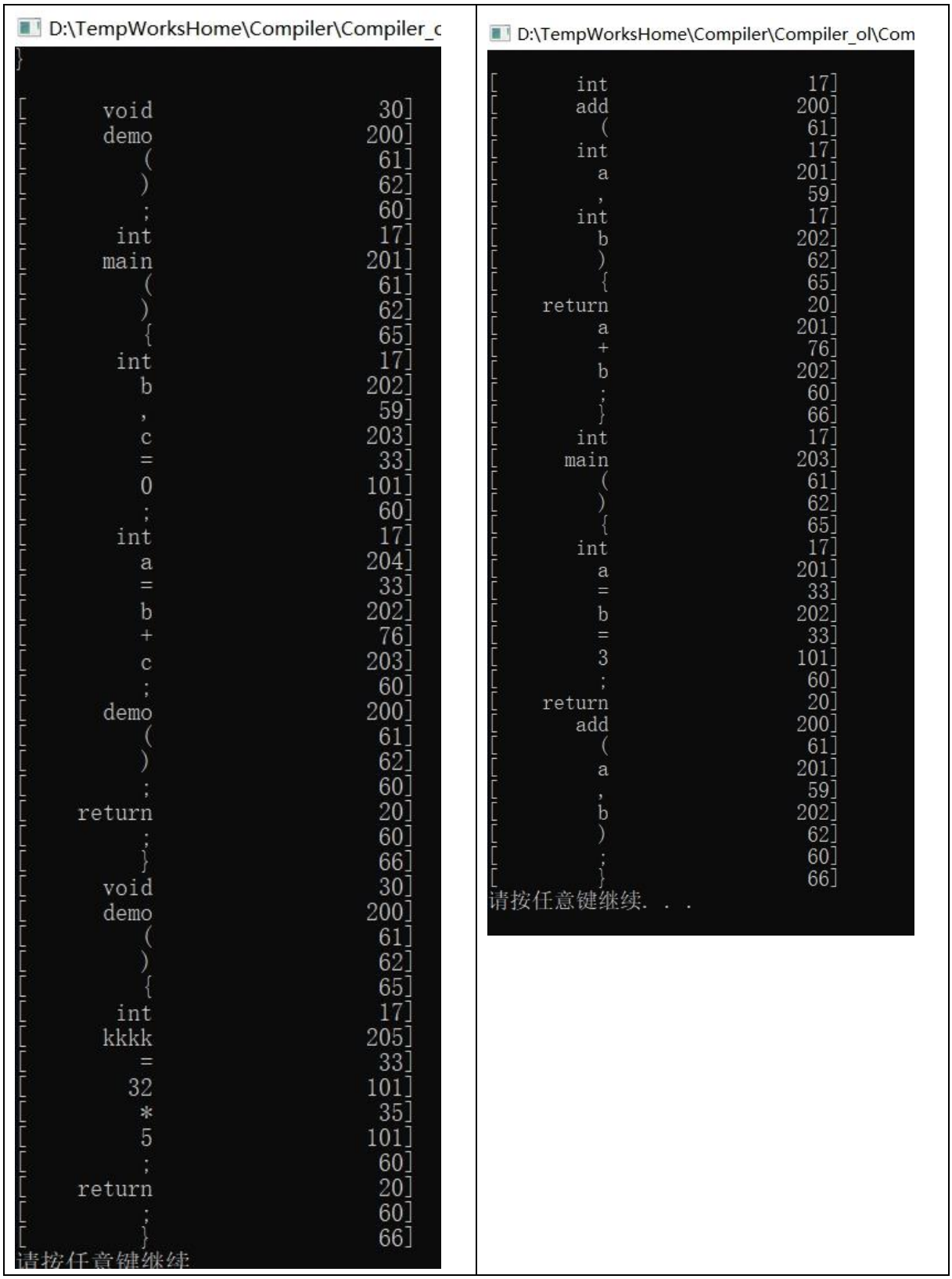
```
void demo();
int main(){
    int b, c = 0;
    int a = b + c;
    demo();
    return;
}
void demo(){
    int kkkk = 32*5;
    return;
}
```

测试集 2：

```
int add(int a, int b){
    return a+b;
}
int main(){
    int a = b =3;
    return add(a,b);
}
```

4.1.2. 测试输出结果

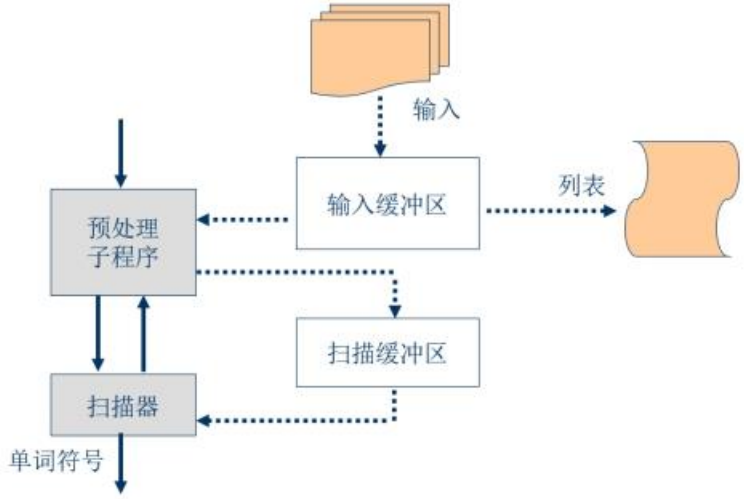
测试集 1 的结果	测试集 2 的结果



源程序中的单词划分开来并根据自身的值给予了一个 int 值的编号，有限条件下的程序正确性得以验证。

4.1.3. 时间复杂度分析

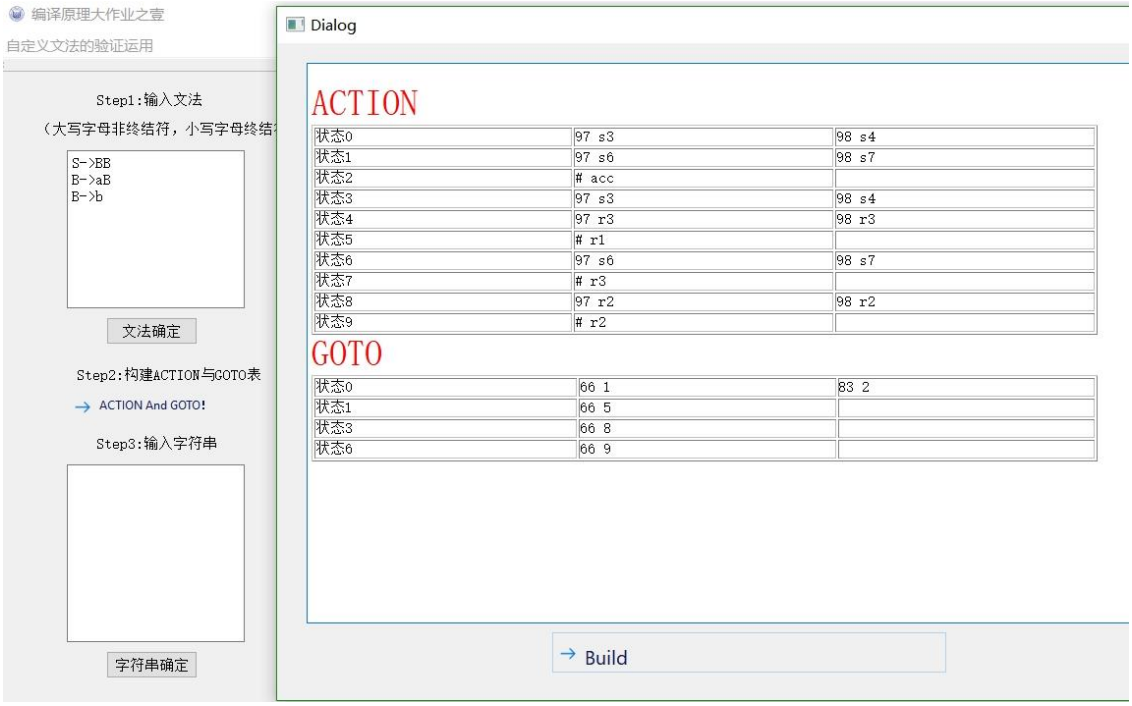
词法程序依照如下图的构造运行，对于预处理子程序与输入缓冲区操作均属于一遍完成，故而对应部分的复杂度为 O（n）。需要识别单词符号的扫描器由于需要预先识别与回退的情况，每次的开销都是常数倍的，故而总的复杂度为 O(n)。



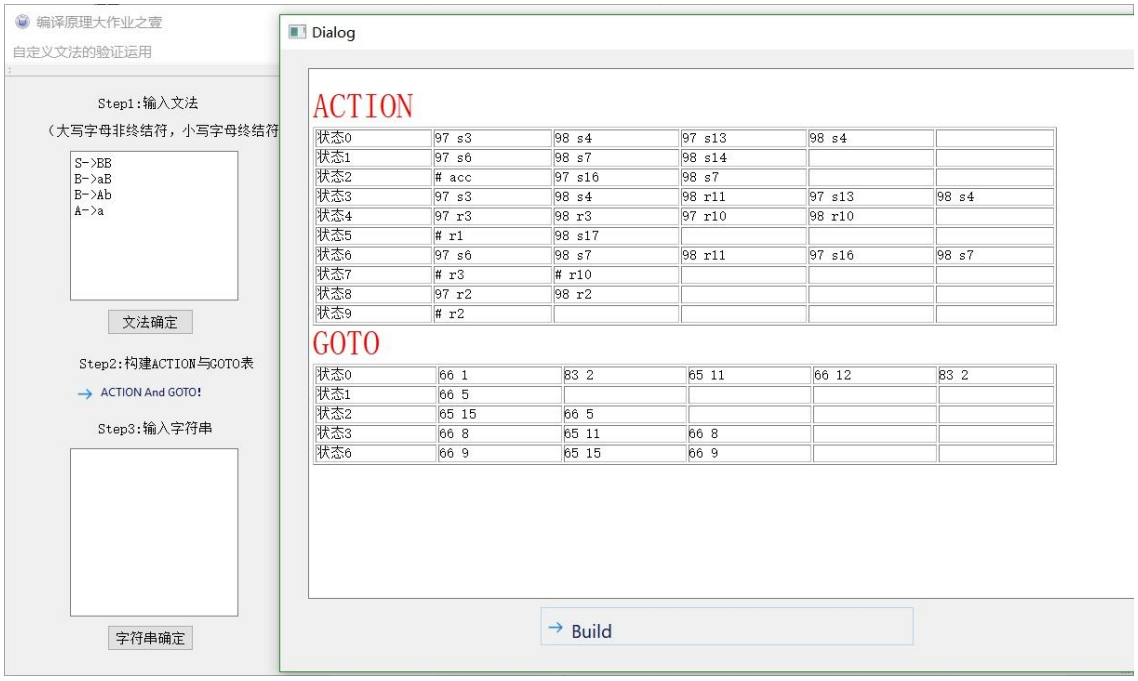
4.2. 测试分析表构造算法

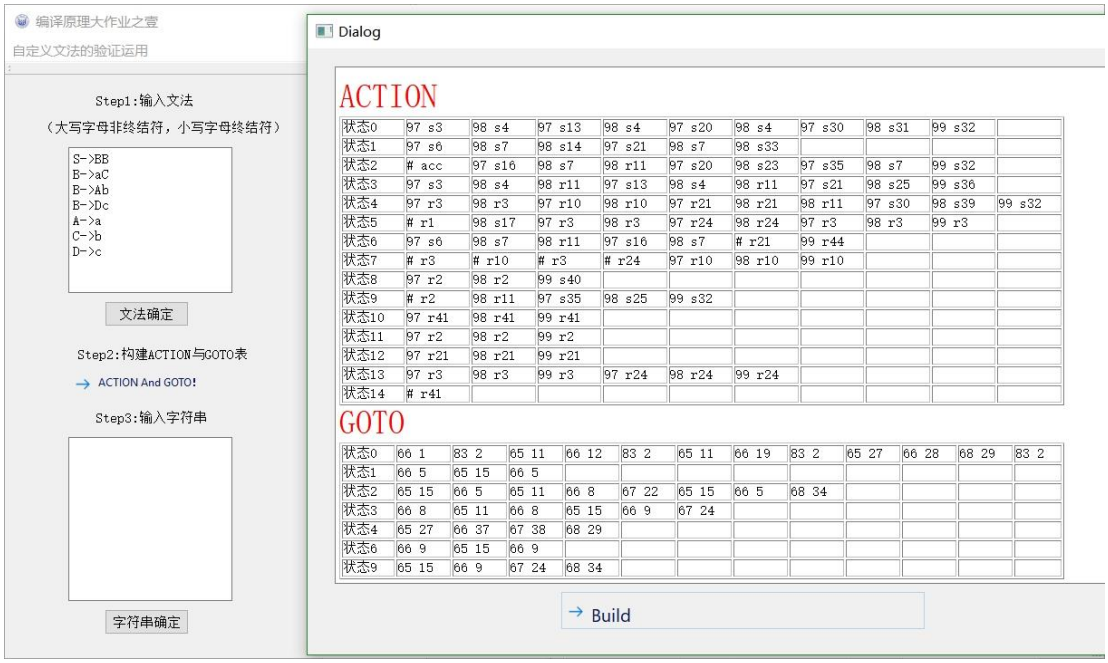
4.2.1. 自定义文法的程序测试

对书本 P.115 的小文法进行测试，输入的时候对拓广文法（S'->S）不再进行输入。文法。单击文法确定->ACTION And GOTO!->Build 之后，即可得到 ACTION 表与 GOTO 表，结果与 P.116 相一致。



不断更换文法，进行多组验证：



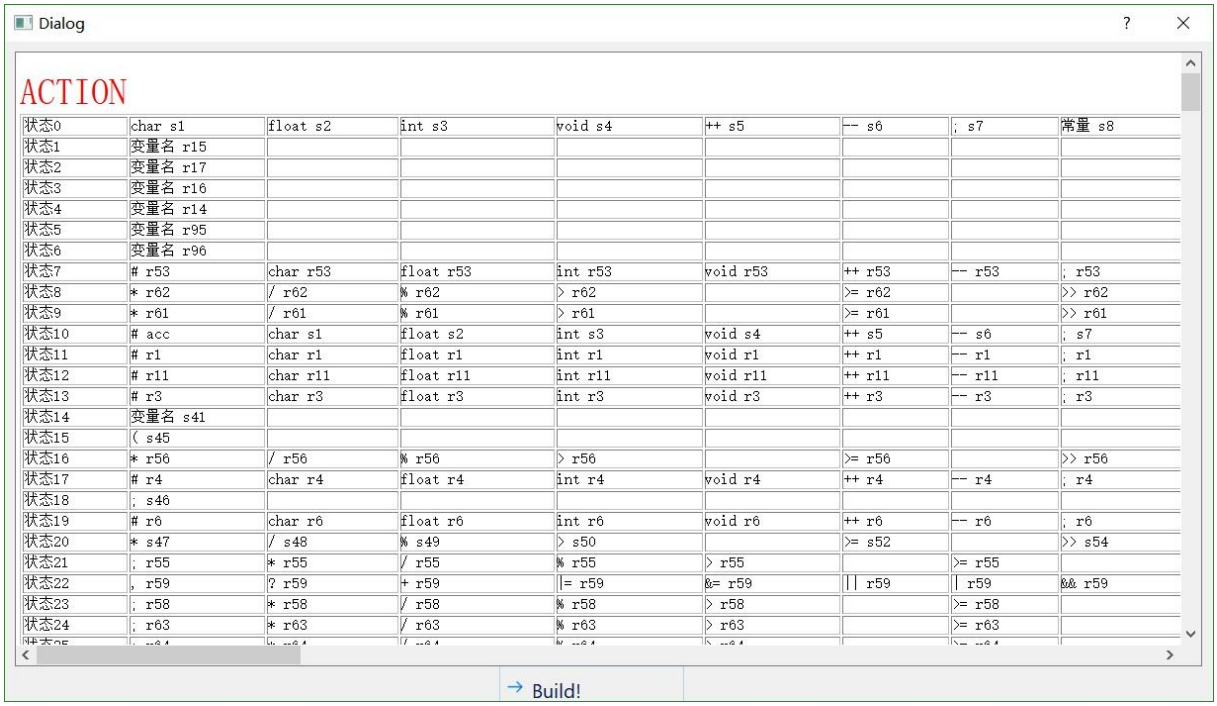


有理由相信，在文法正确的前提下，构造出的 ACTION 与 GOTO 表是有效的。

4.2.2. 类 C 语言文法的程序测试

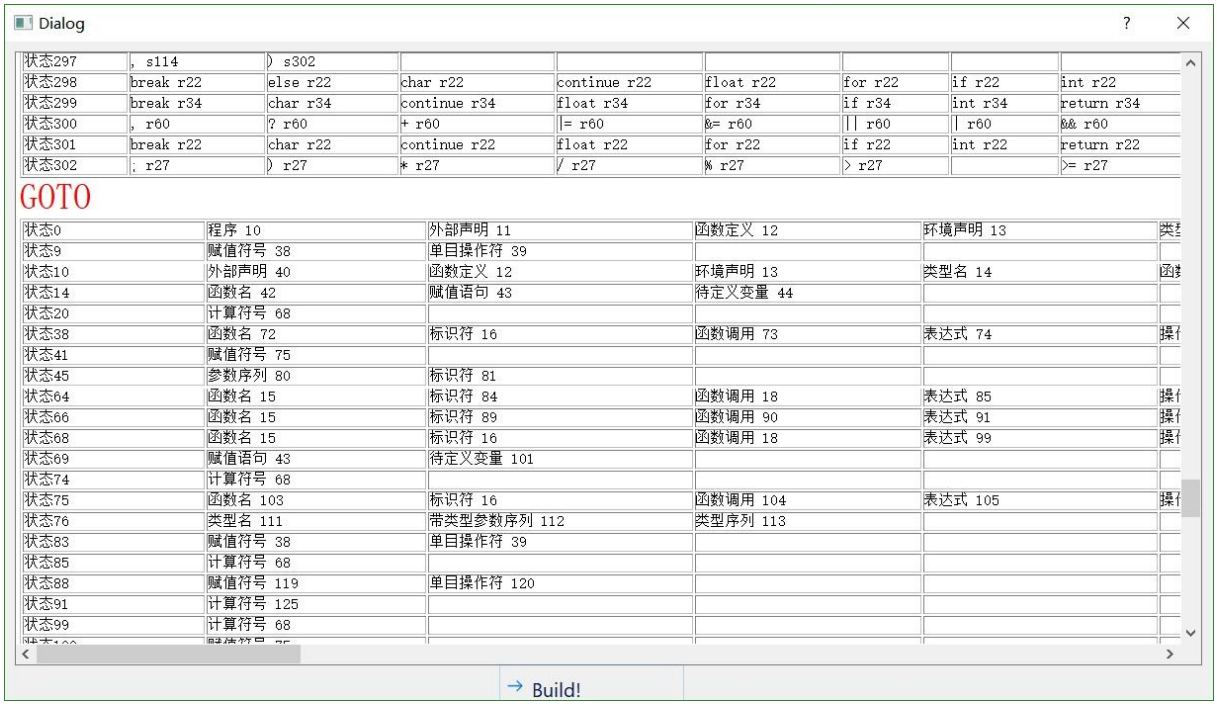
打开基于相同算法，但专为类 C 语言所定制的文法分析器，点击 ACTION And GOTO!->Build，稍等片刻后即可得到现场生成的 ACTION 表与 GOTO 表。在此程序终止之前，打开其他.c 文件并作语法分析时，两表都不再再次生成。

ACTION 表：



状态0	char s1	float s2	int s3	void s4	++ s5	-- s6	; s7	常量 s8
状态1	变量名 r15							
状态2	变量名 r17							
状态3	变量名 r16							
状态4	变量名 r14							
状态5	变量名 r95							
状态6	变量名 r96							
状态7	# r53	char r53	float r53	int r53	void r53	++ r53	-- r53	; r53
状态8	* r62	/ r62	% r62	> r62		>= r62		>> r62
状态9	* r61	/ r61	% r61	> r61		>= r61		>> r61
状态10	# acc	char s1	float s2	int s3	void s4	++ s5	-- s6	; s7
状态11	# r1	char r1	float r1	int r1	void r1	++ r1	-- r1	; r1
状态12	# r11	char r11	float r11	int r11	void r11	++ r11	-- r11	; r11
状态13	# r3	char r3	float r3	int r3	void r3	++ r3	-- r3	; r3
状态14	变量名 s41							
状态15	(s45							
状态16	* r56	/ r56	% r56	> r56		>= r56		>> r56
状态17	# r4	char r4	float r4	int r4	void r4	++ r4	-- r4	; r4
状态18	; s46							
状态19	# r6	char r6	float r6	int r6	void r6	++ r6	-- r6	; r6
状态20	* s47	/ s48	% s49	> s50		>= s52		>> s54
状态21	; r55	* r55	/ r55	% r55	> r55		>= r55	
状态22	, r59	? r59	+ r59	|= r59	&= r59	|| r59	| r59	&& r59
状态23	; r58	* r58	/ r58	% r58	> r58		>= r58	
状态24	; r63	* r63	/ r63	% r63	> r63		>= r63	

GOTO 表：



状态297	, s114	) s302						
状态298	break r22	else r22	char r22	continue r22	float r22	for r22	if r22	int r22
状态299	break r34	char r34	continue r34	float r34	for r34	if r34	int r34	return r34
状态300	, r60	? r60	+ r60	|= r60	&= r60	|| r60	| r60	&& r60
状态301	break r22	char r22	continue r22	float r22	for r22	if r22	int r22	return r22
状态302	; r27	) r27	* r27	/ r27	% r27	> r27		>= r27

状态0	程序 10	外部声明 11	函数定义 12	环境声明 13	
状态9	赋值符号 38	单目操作符 39			
状态10	外部声明 40	函数定义 12	环境声明 13	类型名 14	
状态14	函数名 42	赋值语句 43	待定义变量 44		
状态20	计算符号 68				
状态38	函数名 72	标识符 16	函数调用 73	表达式 74	
状态41	赋值符号 75				
状态45	参数序列 80	标识符 81			
状态64	函数名 15	标识符 84	函数调用 18	表达式 85	
状态66	函数名 15	标识符 89	函数调用 90	表达式 91	
状态68	函数名 15	标识符 16	函数调用 18	表达式 99	
状态69	赋值语句 43	待定义变量 101			
状态74	计算符号 68				
状态75	函数名 103	标识符 16	函数调用 104	表达式 105	
状态76	类型名 111	带类型参数序列 112	类型序列 113		
状态83	赋值符号 38	单目操作符 39			
状态85	计算符号 68				
状态88	赋值符号 119	单目操作符 120			
状态91	计算符号 125				
状态99	计算符号 68				

4.2.3. 时间复杂度分析

构造生成表：

对于类 C 语言类说，每次构造生成表时间大概在 45~60s 左右。程序还有很大的优化空间。对于我们的算法，采用了广度优先搜索和迭代查询重复元素的算法，时间复杂度较高。假设有 n 个闭包，平均每个闭包内有 m 个项目。则生成时间约为：

$$\sum_{i=1}^{i=n} i * \sum_{j=1}^{j=m} j$$

则时间复杂度为$O(m^2n^2)$

然而实际的情况应该比这个还是大许多的，因为没有考虑求解 First 集的时间耗费。

4.2.4. 问题与优化思路

当输入的文法有错误时，程序仍然能够构造出一个 ACTION 表与 GOTO 表：

可见程序尚欠缺一些对于一些不符合 LR(1)的文法的检验能力。但是此问题在这一层没有完全解决并不意味着整个程序的运行也会失败。因为在本程序中，即使构造出了 ACTION 表与 GOTO 表，对于非法文法，在语法分析阶段，试图以任何字符串去匹配的尝试都将以失败告终，并且得到明确的失败反馈，如下图可见：

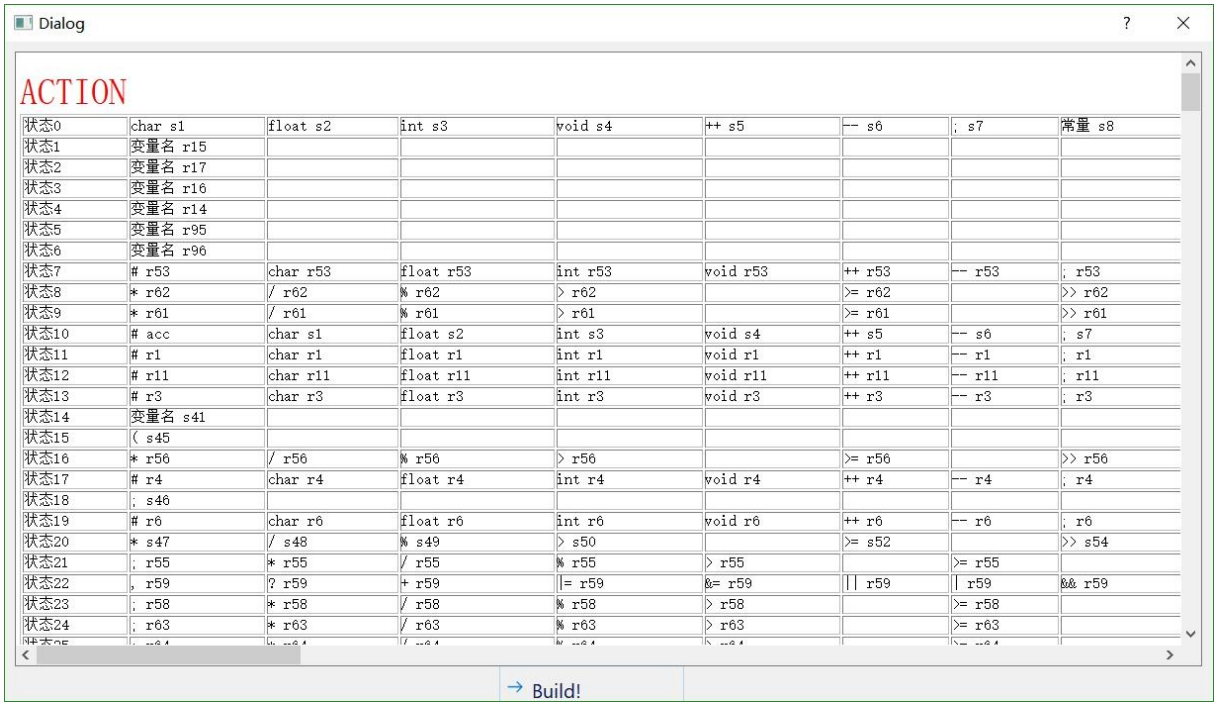
同时，对于非法文法的识别与告警，也是本程序将来的一个优化方向。

4.3. 测试总控程序

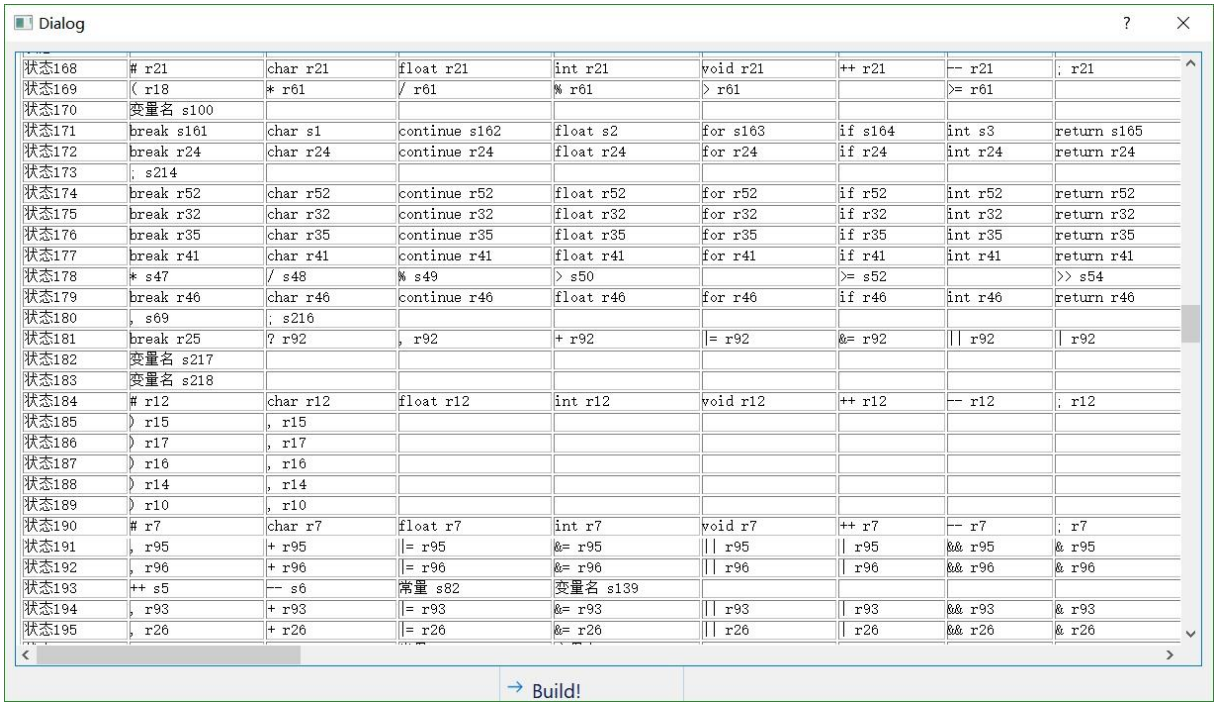
最终提交的两个可执行程序，可执行程序一“自定义文法的验证与运用”主要是为了满足设计要求三：“对输入的一个文法和一个单词串，程序能正确判断此单词串是否为该文法的句子，并要求输出分析过程和语法树”。而可执行程序一“类 C 语言语法分析器”才是重头戏。故本文档中的测试总控程序部分，将主要围绕程序二展开。

主页面：

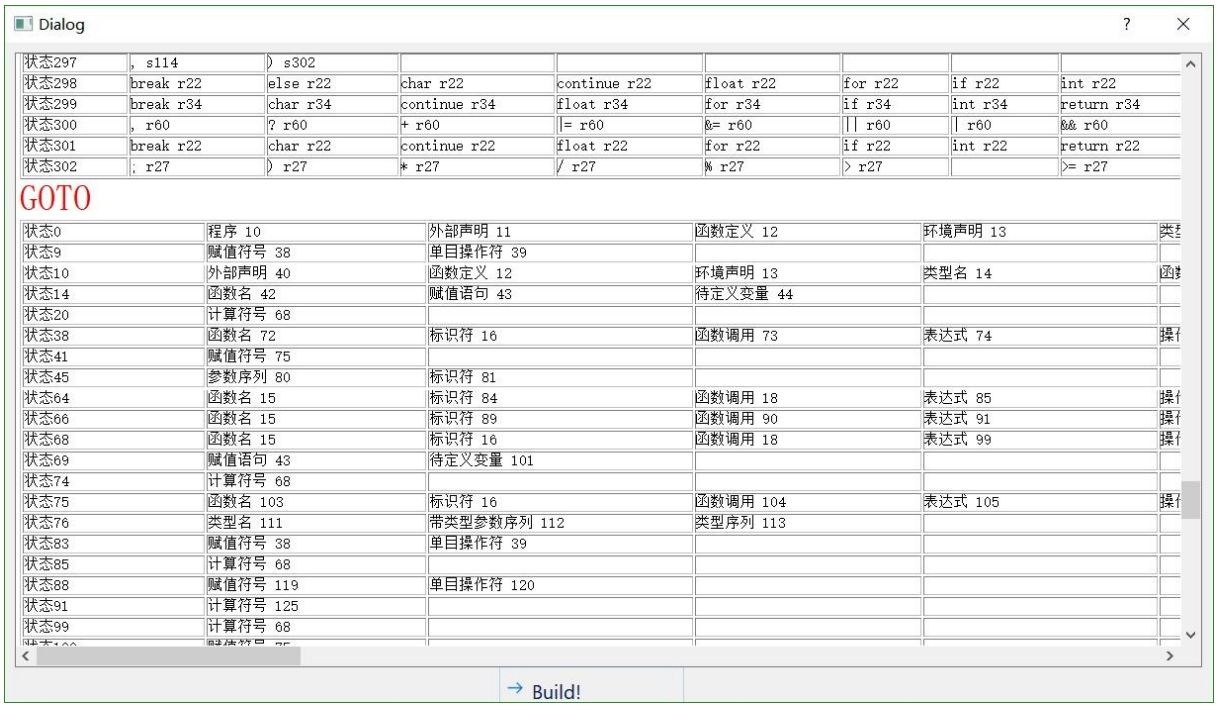
4.3.1. 两表生成



状态0	char s1	float s2	int s3	void s4	++ s5	-- s6	; s7	常量 s8
状态1	变量名 r15							
状态2	变量名 r17							
状态3	变量名 r16							
状态4	变量名 r14							
状态5	变量名 r95							
状态6	变量名 r96							
状态7	# r53	char r53	float r53	int r53	void r53	++ r53	-- r53	; r53
状态8	* r62	/ r62	% r62	> r62		>= r62		>> r62
状态9	* r61	/ r61	% r61	> r61		>= r61		>> r61
状态10	# acc	char s1	float s2	int s3	void s4	++ s5	-- s6	; s7
状态11	# r1	char r1	float r1	int r1	void r1	++ r1	-- r1	; r1
状态12	# r11	char r11	float r11	int r11	void r11	++ r11	-- r11	; r11
状态13	# r3	char r3	float r3	int r3	void r3	++ r3	-- r3	; r3
状态14	变量名 s41							
状态15	(s45							
状态16	* r56	/ r56	% r56	> r56		>= r56		>> r56
状态17	# r4	char r4	float r4	int r4	void r4	++ r4	-- r4	; r4
状态18	; s46							
状态19	# r6	char r6	float r6	int r6	void r6	++ r6	-- r6	; r6
状态20	* s47	/ s48	% s49	> s50		>= s52		>> s54
状态21	; r55	* r55	/ r55	% r55	> r55		>= r55	
状态22	, r59	? r59	+ r59	|= r59	&= r59	|| r59	| r59	&& r59
状态23	; r58	* r58	/ r58	% r58	> r58		>= r58	
状态24	; r63	* r63	/ r63	% r63	> r63		>= r63	



状态168	# r21	char r21	float r21	int r21	void r21	++ r21	-- r21	; r21
状态169	(r18	* r61	/ r61	% r61	> r61		>= r61	
状态170	变量名 s100							
状态171	break s161	char s1	continue s162	float s2	for s163	if s164	int s3	return s165
状态172	break r24	char r24	continue r24	float r24	for r24	if r24	int r24	return r24
状态173	; s214							
状态174	break r52	char r52	continue r52	float r52	for r52	if r52	int r52	return r52
状态175	break r32	char r32	continue r32	float r32	for r32	if r32	int r32	return r32
状态176	break r35	char r35	continue r35	float r35	for r35	if r35	int r35	return r35
状态177	break r41	char r41	continue r41	float r41	for r41	if r41	int r41	return r41
状态178	* s47	/ s48	% s49	> s50		>= s52		>> s54
状态179	break r46	char r46	continue r46	float r46	for r46	if r46	int r46	return r46
状态180	, s69	; s216						
状态181	break r25	? r92	, r92	+ r92	|= r92	&= r92	|| r92	| r92
状态182	变量名 s217							
状态183	变量名 s218							
状态184	# r12	char r12	float r12	int r12	void r12	++ r12	-- r12	; r12
状态185	) r15	, r15						
状态186	) r17	, r17						
状态187	) r16	, r16						
状态188	) r14	, r14						
状态189	) r10	, r10						
状态190	# r7	char r7	float r7	int r7	void r7	++ r7	-- r7	; r7
状态191	, r95	+ r95	|= r95	&= r95	|| r95	| r95	&& r95	& r95
状态192	, r96	+ r96	|= r96	&= r96	|| r96	| r96	&& r96	& r96
状态193	++ s5	-- s6	常量 s82	变量名 s139				
状态194	, r93	+ r93	|= r93	&= r93	|| r93	| r93	&& r93	& r93
状态195	, r26	+ r26	|= r26	&= r26	|| r26	| r26	&& r26	& r26



状态297	, s114	) s302						
状态298	break r22	else r22	char r22	continue r22	float r22	for r22	if r22	int r22
状态299	break r34	char r34	continue r34	float r34	for r34	if r34	int r34	return r34
状态300	, r60	? r60	+ r60	|= r60	&= r60	|| r60	| r60	&& r60
状态301	break r22	char r22	continue r22	float r22	for r22	if r22	int r22	return r22
状态302	; r27	) r27	* r27	/ r27	% r27	> r27		>= r27

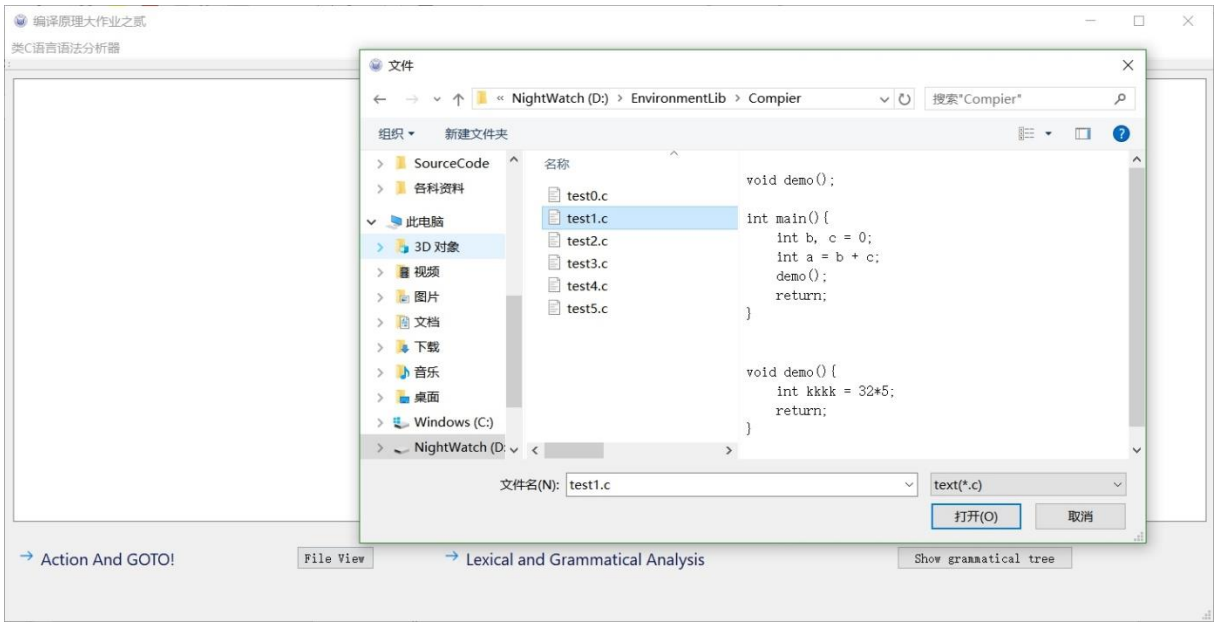
GOTO

状态0	程序 10	外部声明 11	函数定义 12	环境声明 13	类
状态9	赋值符号 38	单目操作符 39			
状态10	外部声明 40	函数定义 12	环境声明 13	类型名 14	函
状态14	函数名 42	赋值语句 43	待定义变量 44		
状态20	计算符号 68				
状态38	函数名 72	标识符 16	函数调用 73	表达式 74	操
状态41	赋值符号 75				
状态45	参数序列 80	标识符 81			
状态64	函数名 15	标识符 84	函数调用 18	表达式 85	操
状态66	函数名 15	标识符 89	函数调用 90	表达式 91	操
状态68	函数名 15	标识符 16	函数调用 18	表达式 99	操
状态69	赋值语句 43	待定义变量 101			
状态74	计算符号 68				
状态75	函数名 103	标识符 16	函数调用 104	表达式 105	操
状态76	类型名 111	带类型参数序列 112	类型序列 113		
状态83	赋值符号 38	单目操作符 39			
状态85	计算符号 68				
状态88	赋值符号 119	单目操作符 120			
状态91	计算符号 125				
状态99	计算符号 68				

该部分须先于语法分析完成，操作模式固定，完全正确，一气呵成

4.3.2. 文件读入与源代码显示

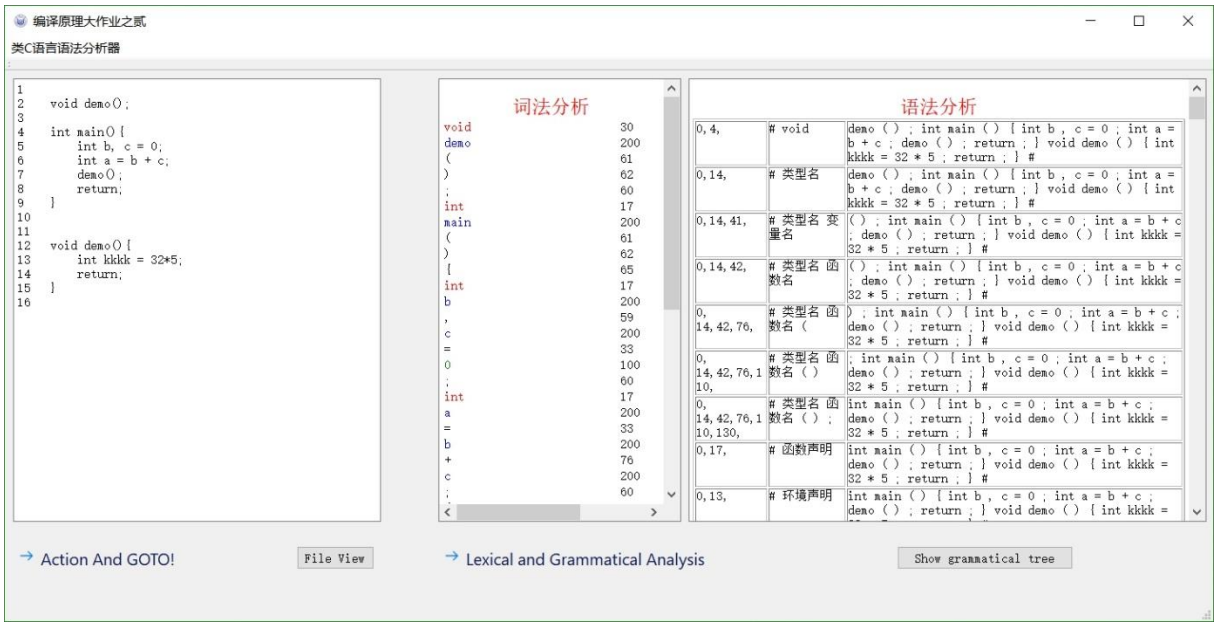
选中.c 类型文件：



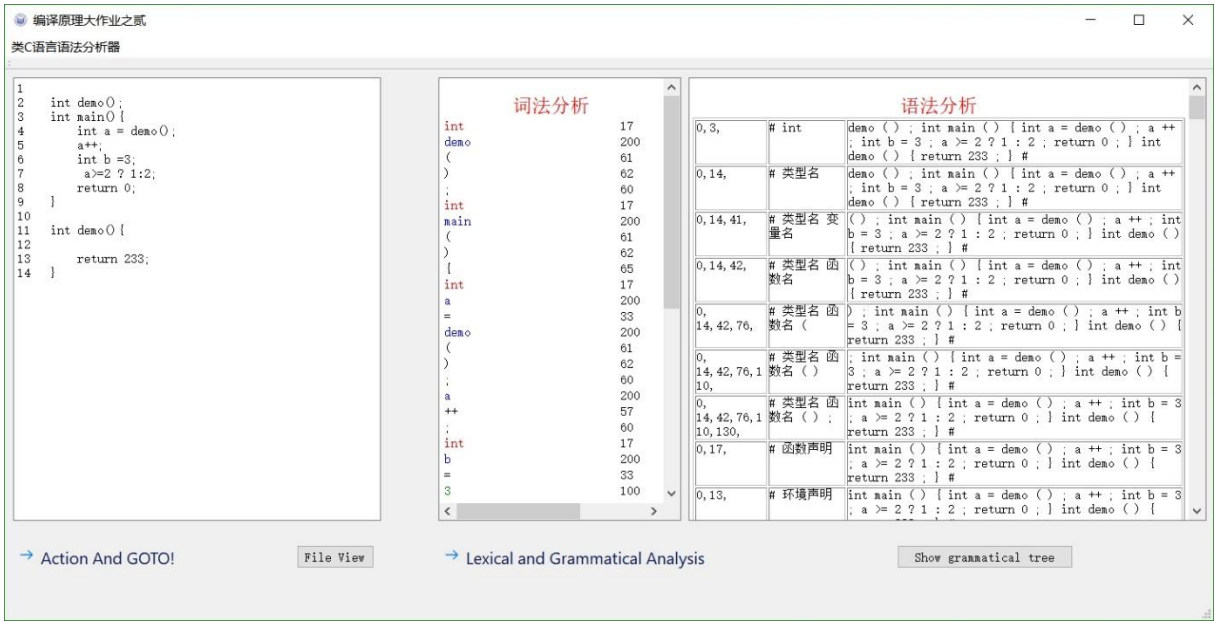
代码读入并显示行号：

若想读入.cpp/.txt 类型文件暂时还不支持，但.c 类型读入没有错误。

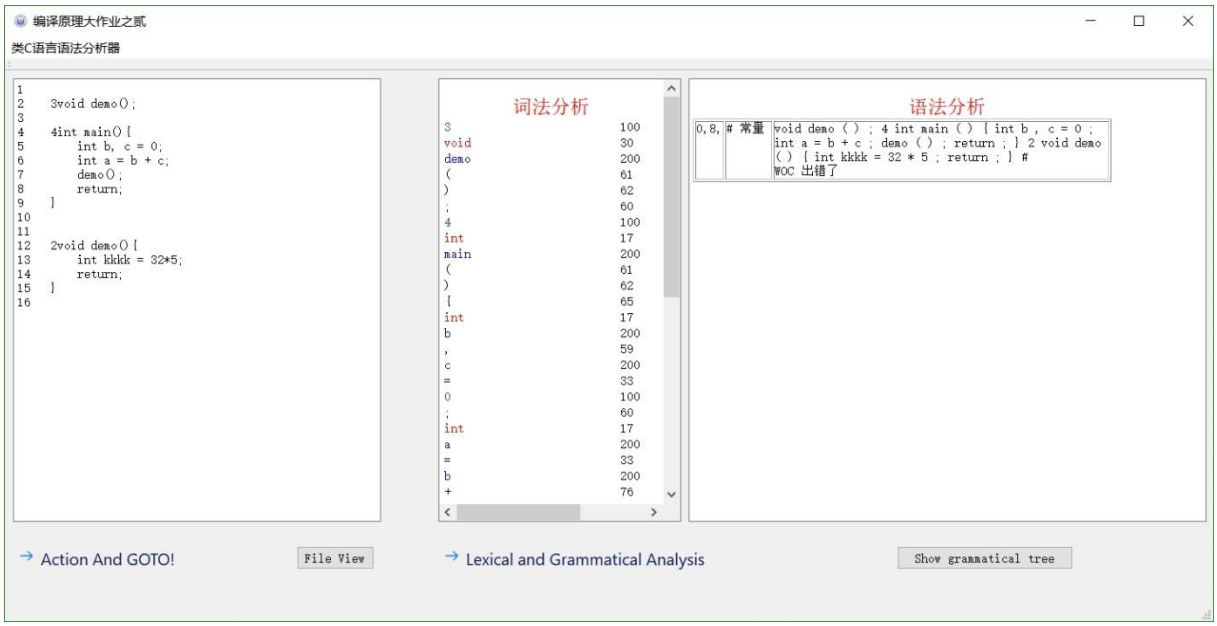
4.3.3. 语法分析及过程显示



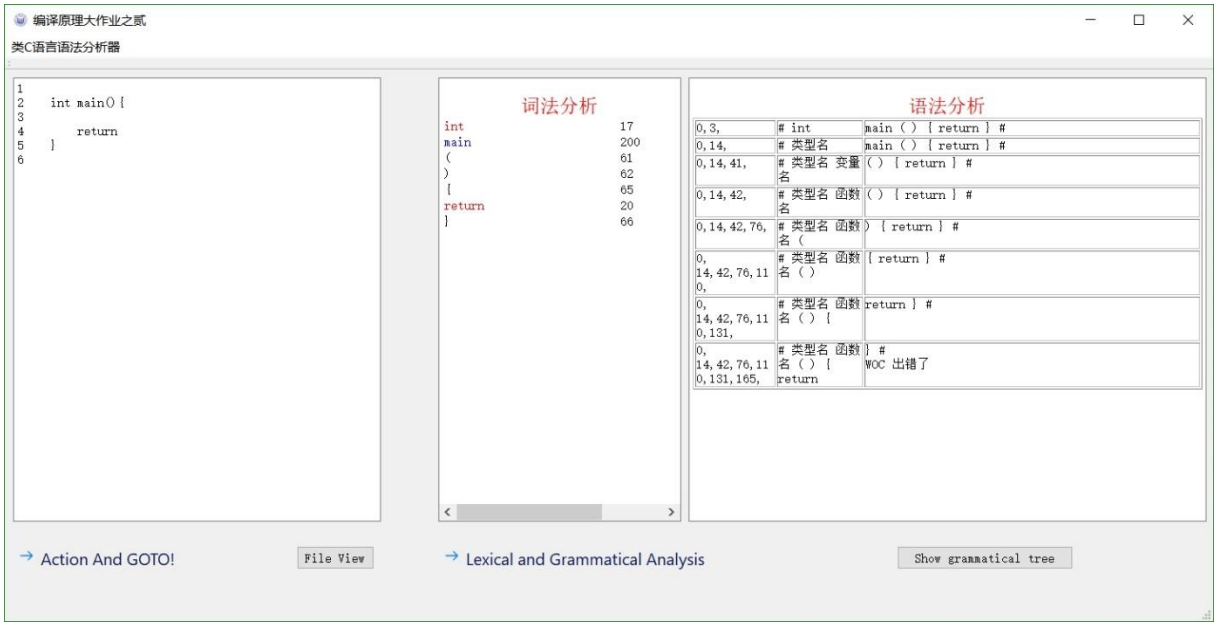
支持函数调用，函数声明，自增语句，三目运算等：



对于语法错误的检测：



上图中 3void, 4int, 2void 等语法错误被成功识别出。



上图中程序返回 return 没有加分号的语法错误被成功识别出。

4.3.4. 语法树生成

4.4. 设计局限性与改进空间

4.4.1. 一些尚不支持的 C 语言语法

①暂不支持宏定义
②暂不支持头文件引用
③不支持指针，解引用等运算。
④不支持结构体
⑤…….

4.4.2. 符号编号问题

在语法分析的过程中，为了便于程序编码运算，将终结符和非终结符进行了数值化，在调试的过程中不容易对应。目前，暂未想到更好地编码方法。

4.4.3. 程序复杂度

分析整个程序，时间耗费主要在 Tblae::buildTable()这一块。每次构造分析表平均耗时 45s-60s 左右。分析逻辑及结构和代码，主要问题在于
①在构造新的闭包的时候，需要判断新构造的闭包是否在之前出现过，因此需要对每个已存在的闭包进行比对，判断两个闭包所含的项目集是否一致。这个是一个时间耗散点。
②在根据初始集合构造闭包的过程中，采用了递归的算法，当产生新的 LR(1)项目时，要对整个闭包进行遍历判断新的 LR(1)是否之前已经存在过，这会造成极大的程序时间损耗。比如，假设当前构造中的闭包有 n 个 LR(1)，总共有 m 个 LR(1)项目，则每个新生成的 LR(1)都会去检测是否存在在闭包中，时间复杂度至少为$O(mn + m^2)$。但这仅仅是一个闭包的，对于类 C 分析表来说，有 200 多个闭包，时间复杂度还是较大的。

目前，想到可以使用哈希值来作为索引进行直接判断是否已经存在过，以空间换时间。

5. 总结与收获

5.1. 理解课程知识

本次语法分析器的设计实验，让我们对本学期前半部分的课程有了较为清晰的了解，从词法分析开始，到语法分析，乃至于基于 LR(1)的分析等等。通过本次实践与理论相结合的机会，我们加深了对课程的理解，对后半学期的编译原理课程学习有了更多的信心。

5.2. 掌握算法与数据结构

上学期学习了算法相关知识和数据结构相关知识后，一直觉得无用武之地。这次实验极大地锻炼了我们的编程能力，从一开始对着课本不知道怎么构造数据结构。通过逐步分析讨论，逐渐构造出数据结构。但是代码构造过程中也发现了数据结构的设计是非常重要的，良好的结构设计可以让代码书写难度降低。而且方便多人同时编程。这次实验加深我们我对哈希表，数组，树，队列，栈等基本数据结构的认识。而且这次实验使我们了解到了广度优先和深度优先算法的实战应用，以及记忆化搜索等时间优化方法。由于时间原因，还有许多地方可以优化，希望在之后我们能对程序补充，使其可读性，健壮性增强。

5.3. 感受团队合作魅力

本次大作业中，与队友通力协作，加班加点完成任务。既相互间取长补短增长了知识水平，又加深了革命友谊，可谓一举多得！

6. 参考文献

[1]. 《程序设计语言编译原理（第三版）》.陈火旺等；
[2]. CSDN:C 语言文法产生式. https://blog.csdn.net/zanfeng/article/details/53028345
[3]. LR(1)文法讲解. https://www.cnblogs.com/xpwi/p/11070888.html

7. 附录

7.1. 附录 1：类 C 语言文法产生式

见文件夹下’类 C 语言文法产生式.xlsx’

7.2. 附录 2：终结符与非终结符 int 值表

见文件夹下’终结符与非终结符 int 值.xlsx’