

同济大学计算机科学与技术系

编译原理课程综合实验一



院 系 电子与信息工程学院

专 业 计算机科学与技术

组 员 1852877 赵昊堃

组 员 1853047 孔庆晨

组 员 1951727 史睿琪

日 期 2021 年 11 月 6 日

目录

1 需求分析.....	4
1.1 输入输出约定.....	4
1.1.1 源程序输入.....	4
1.1.2 文法规则输入/人工分析表输入.....	4
1.1.3 词法分析输出.....	6
1.1.4 语法分析输出.....	7
1.1.5 语义分析与中间代码生成.....	10
1.2 程序功能.....	12
1.3 测试数据.....	13
1.3.1 测试代码 1（标准的含过程调用的 cpp 文件）	13
1.3.2 测试代码 2（含过程调用、注释的较为复杂的 cpp 文件） ...	14
2 概要设计.....	15
2.1 任务分解.....	15
2.1.1 词法分析.....	15
2.1.2 语法分析.....	15
2.1.3 语义分析.....	16
2.2 数据类型定义.....	27
2.2.1 词法分析相关.....	27
2.2.2 语法分析相关.....	28
2.2.3 语义分析相关.....	29
2.3 主程序流程.....	33
3 详细设计.....	36
3.1 主要函数分析及设计.....	36

3.2 函数调用关系.....	40
4 调试分析.....	42
4.1 测试数据及测试结果.....	42
4.1.1 正常运行结果展示.....	42
4.1.2 错误与警告示例.....	47
4.2 调试过程存在的问题及解决方法.....	52
5 总结与收获.....	53
6 小组分工.....	54
7 参考文献.....	54

1 需求分析

1.1 输入输出约定

1.1.1 源程序输入

本项目实现一个基于 LR(1) 分析方法的类 C 语言词法语法分析器，源程序通过文件读取以字符串形式输入。文件内容应为类 C 语言编写的程序代码，可包含单行及多行注释。为了成功进行词法与语法分析，程序中的符号、关键字以及语法规则应与分析器中定义相符，否则分析器将给出相应报错信息。

1.1.2 语法规则输入/人工分析表输入

包含过程调用的语法规则如下：

1. $\langle \text{Program} \rangle \rightarrow \langle \text{DeclareString} \rangle$
2. $\langle \text{DeclareString} \rangle \rightarrow \langle \text{DeclareString-option} \rangle \langle \text{DeclareString} \rangle \mid [\text{z}]$
3. $\langle \text{DeclareString-option} \rangle \rightarrow$
 $\langle \text{ParameterDeclare} \rangle [\text{ID}] [\text{SEMI}] \mid \langle \text{FunctionDeclare} \rangle \langle \text{FunDec} \rangle \langle \text{Block} \rangle$
4. $\langle \text{ParameterDeclare} \rangle \rightarrow [\text{INT}]$
5. $\langle \text{FunctionDeclare} \rangle \rightarrow [\text{VOID}] \mid [\text{INT}]$
6. $\langle \text{FunDec} \rangle \rightarrow [\text{ID}] \langle \text{CreateFunTable_m} \rangle [\text{LPAREN}] \langle \text{VarList} \rangle [\text{RPAREN}]$
7. $\langle \text{CreateFunTable_m} \rangle \rightarrow [\text{z}]$
8. $\langle \text{VarList} \rangle \rightarrow \langle \text{ParamDec} \rangle [\text{COMMA}] \langle \text{VarList} \rangle \mid \langle \text{ParamDec} \rangle \mid [\text{z}]$
9. $\langle \text{ParamDec} \rangle \rightarrow \langle \text{ParameterDeclare} \rangle [\text{ID}]$
10. $\langle \text{Block} \rangle \rightarrow [\text{LBBRACKET}] \langle \text{DefList} \rangle \langle \text{StmtList} \rangle [\text{RBBRACKET}]$
11. $\langle \text{DefList} \rangle \rightarrow \langle \text{Def} \rangle \langle \text{DefList} \rangle \mid [\text{z}]$
12. $\langle \text{Def} \rangle \rightarrow \langle \text{ParameterDeclare} \rangle [\text{ID}] [\text{SEMI}]$

13. $\langle \text{StmtList} \rangle \rightarrow \langle \text{Stmt} \rangle \langle \text{StmtList} \rangle \mid [\text{z}]$
14. $\langle \text{Stmt} \rangle \rightarrow \langle \text{AssignStmt} \rangle [\text{SEMI}] \mid \langle \text{ReturnStmt} \rangle [\text{SEMI}] \mid \langle \text{IfStmt} \rangle \mid \langle \text{WhileStmt} \rangle \mid \langle \text{CallStmt} \rangle [\text{SEMI}]$
15. $\langle \text{AssignStmt} \rangle \rightarrow [\text{ID}] [\text{ASSIGN}] \langle \text{Exp} \rangle$
16. $\langle \text{Exp} \rangle \rightarrow \langle \text{AddSubExp} \rangle \mid \langle \text{Exp} \rangle \langle \text{Relop} \rangle \langle \text{AddSubExp} \rangle$
17. $\langle \text{AddSubExp} \rangle \rightarrow \langle \text{Item} \rangle \mid \langle \text{Item} \rangle [\text{PLUS}] \langle \text{Item} \rangle \mid \langle \text{Item} \rangle [\text{SUB}] \langle \text{Item} \rangle$
19. $\langle \text{Item} \rangle \rightarrow \langle \text{Factor} \rangle \mid \langle \text{Factor} \rangle [\text{MUL}] \langle \text{Factor} \rangle \mid \langle \text{Factor} \rangle [\text{DIV}] \langle \text{Factor} \rangle$
20. $\langle \text{Factor} \rangle \rightarrow [\text{NUM}] \mid [\text{LPAREN}] \langle \text{Exp} \rangle [\text{RPAREN}] \mid [\text{ID}] \mid \langle \text{CallStmt} \rangle$
21. $\langle \text{CallStmt} \rangle \rightarrow [\text{ID}] [\text{LPAREN}] \langle \text{CallFunCheck} \rangle \langle \text{Args} \rangle [\text{RPAREN}]$
22. $\langle \text{CallFunCheck} \rangle \rightarrow [\text{z}]$
23. $\langle \text{Args} \rangle \rightarrow \langle \text{Exp} \rangle [\text{COMMA}] \langle \text{Args} \rangle \mid \langle \text{Exp} \rangle \mid [\text{z}]$
24. $\langle \text{ReturnStmt} \rangle \rightarrow [\text{RETURN}] \langle \text{Exp} \rangle \mid [\text{RETURN}]$
25. $\langle \text{Relop} \rangle \rightarrow [\text{GT}] \mid [\text{LT}] \mid [\text{GE}] \mid [\text{LE}] \mid [\text{EQ}] \mid [\text{NQ}]$
26. $\langle \text{IfStmt} \rangle \rightarrow [\text{IF}] \langle \text{IfStmt_m1} \rangle [\text{LPAREN}] \langle \text{Exp} \rangle [\text{RPAREN}] \langle \text{IfStmt_m2} \rangle \langle \text{Block} \rangle \langle \text{IfNext} \rangle$
27. $\langle \text{IfStmt_m1} \rangle \rightarrow [\text{z}]$
28. $\langle \text{IfStmt_m2} \rangle \rightarrow [\text{z}]$
29. $\langle \text{IfNext} \rangle \rightarrow [\text{z}] \mid \langle \text{IfStmt_next} \rangle [\text{ELSE}] \langle \text{Block} \rangle$
30. $\langle \text{IfStmt_next} \rangle \rightarrow [\text{z}]$
31. $\langle \text{WhileStmt} \rangle \rightarrow [\text{WHILE}] \langle \text{WhileStmt_m1} \rangle [\text{LPAREN}] \langle \text{Exp} \rangle [\text{RPAREN}] \langle \text{WhileStmt_m2} \rangle \langle \text{Block} \rangle$

32. <WhileStmt_m1> -> [z]

33. <WhileStmt_m2> -> [z]

为了编程方便，将非终结符替换为其对应的英文名，非终结符使用“<>”进行标识，终结符使用“[]”进行标识，空串用“[z]”进行标识。

为了方便语义分析中创建符号表、记录四元式位置、回填等操作，引入相应非终结符，使其产生空串，并在语义分析时为其赋予相应的语义动作。

1.1.3 词法分析输出

词法分析结果以两种形式输出。其一是面向用户的分析表形式，生成一个中间文本文件，在文件中列表输出词法分析结果，包含词元所在行、词元内容和词元类型。下图只截取了部分。

TOKEN STREAM		
Line	Content	Type
1	int	[INT]
1	a	[ID]
1	;	[SEMI]
2	int	[INT]
2	b	[ID]
2	;	[SEMI]
3	int	[INT]
3	program	[ID]
3	([LPAREN]
3	int	[INT]

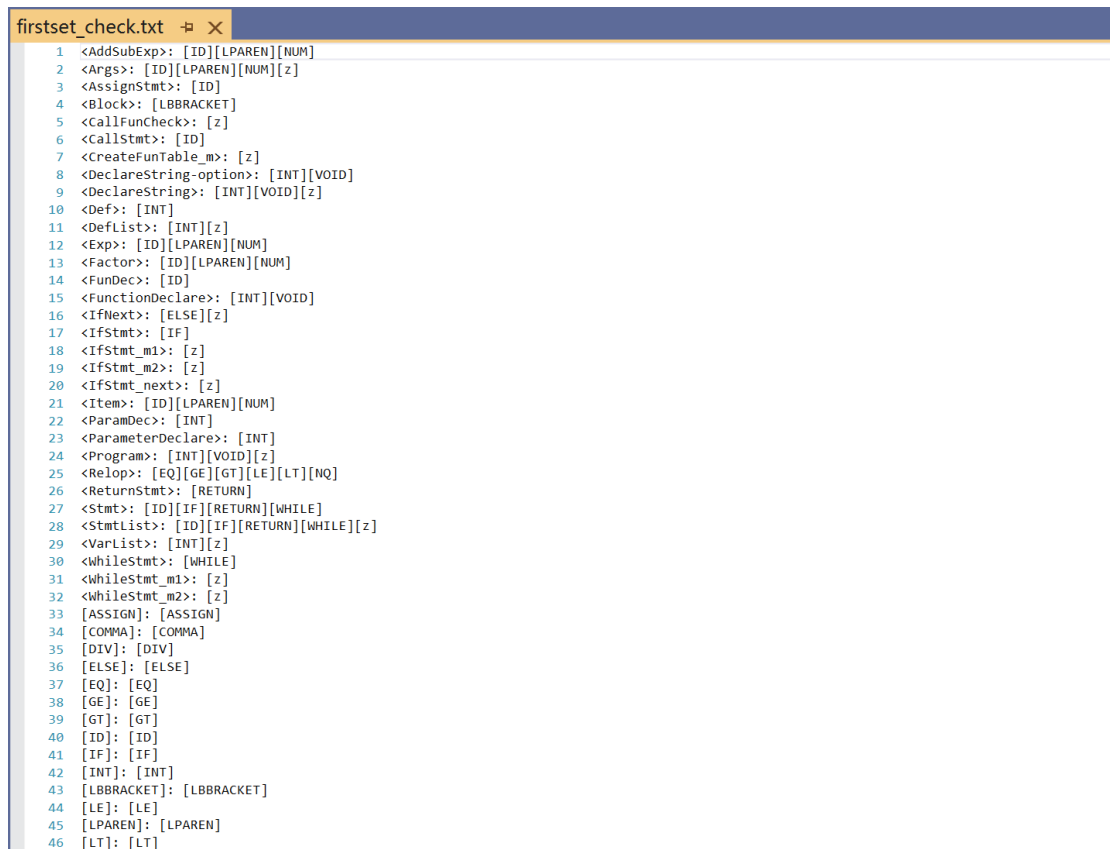
其二是面向语法分析器的 Result 队列形式，队列中按照词元出现顺序依次存储词元类型（终结符）与词元内容（对应终结符名），作为下一步语法分析的材料来源。

1.1.4 语法分析输出

语法分析根据给定的语法规则，分析用户指定路径的代码文本，能够识别、输出文本终结符号、非终结符号，求算所有非终结符号的 FIRST 集与 FOLLOW 集。结合词法分析内容，对给定的代码文段进行规约，生成 LR1 分析表（GOTO 表与 ACTION 表）给出“规约成功”、“非法字符”、“ACTION 为空，规约错误！”等相应反馈与错误提示，具有一定交互性与鲁棒性。并借助 graphviz 实现了语法分析树的可视化生成，完善了借助 QT 实现的用户友好界面。

● FIRST 集输出

通过对产生式输出符号<<的重载，实现了较为方便的集合遍历输出，这对 LR1 分析法额外细节的了解和 DEBUG 都具有很大意义。



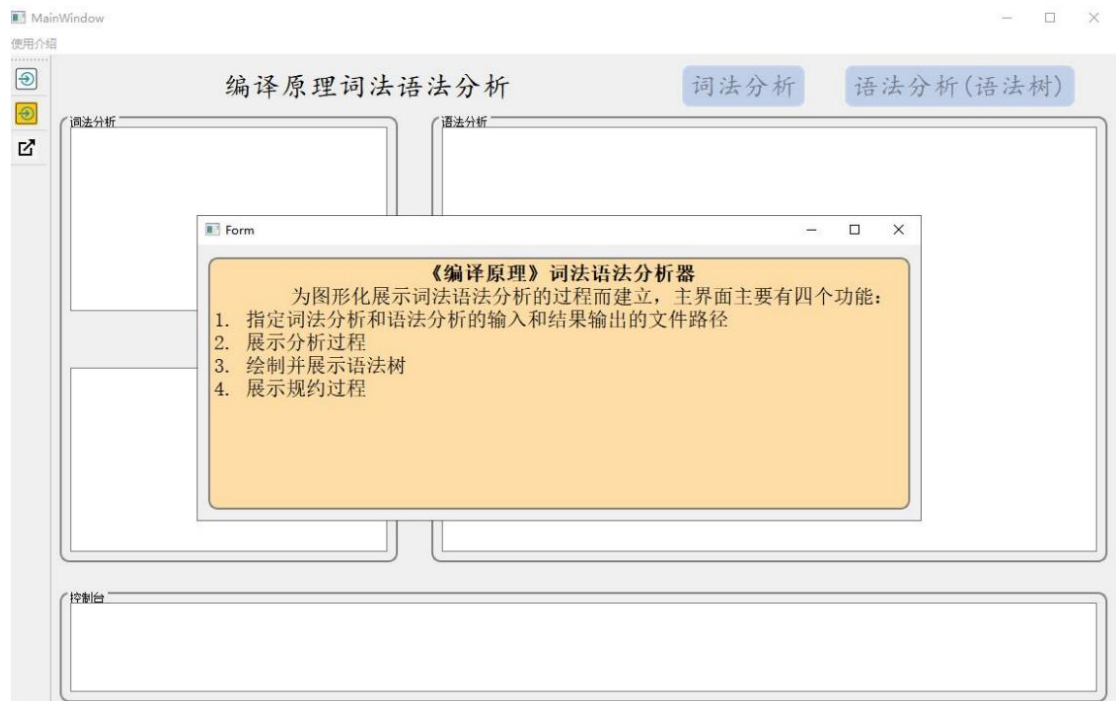
```
firstset_check.txt
1 <AddSubExp>: [ID][LPAREN][NUM]
2 <Args>: [ID][LPAREN][NUM][z]
3 <AssignStmt>: [ID]
4 <Block>: [LBBRACKET]
5 <CallFunCheck>: [z]
6 <CallStmt>: [ID]
7 <CreateFunTable_m>: [z]
8 <DeclareString-option>: [INT][VOID]
9 <DeclareString>: [INT][VOID][z]
10 <Def>: [INT]
11 <DefList>: [INT][z]
12 <Exp>: [ID][LPAREN][NUM]
13 <Factor>: [ID][LPAREN][NUM]
14 <FunDec>: [ID]
15 <FunctionDeclare>: [INT][VOID]
16 <IfNext>: [ELSE][z]
17 <IfStmt>: [IF]
18 <IfStmt_m1>: [z]
19 <IfStmt_m2>: [z]
20 <IfStmt_next>: [z]
21 <Item>: [ID][LPAREN][NUM]
22 <ParamDec>: [INT]
23 <ParameterDeclare>: [INT]
24 <Program>: [INT][VOID][z]
25 <Relop>: [EQ][GE][GT][LE][LT][NQ]
26 <ReturnStmt>: [RETURN]
27 <Stmt>: [ID][IF][RETURN][WHILE]
28 <StmtList>: [ID][IF][RETURN][WHILE][z]
29 <VarList>: [INT][z]
30 <WhileStmt>: [WHILE]
31 <WhileStmt_m1>: [z]
32 <WhileStmt_m2>: [z]
33 [ASSIGN]: [ASSIGN]
34 [COMMA]: [COMMA]
35 [DIV]: [DIV]
36 [ELSE]: [ELSE]
37 [EQ]: [EQ]
38 [GE]: [GE]
39 [GT]: [GT]
40 [ID]: [ID]
41 [IF]: [IF]
42 [INT]: [INT]
43 [LBBRACKET]: [LBBRACKET]
44 [LE]: [LE]
45 [LPAREN]: [LPAREN]
46 [LT]: [LT]
```

● LR1 分析表的生成

ACTION 与 GOTO 联表作为 CSV 数据源输入 EXCEL 表格，生成图像如下，表格数据量较大，这里只截取了部分。

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI
	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]	[=]
1																																			
2																																			
3																																			
4																																			
5																																			
6																																			
7																																			
8																																			
9																																			
10																																			
11																																			
12																																			
13																																			
14																																			
15																																			
16																																			
17																																			
18																																			
19																																			
20																																			
21																																			
22																																			
23																																			
24																																			
25																																			
26																																			
27																																			
28																																			
29																																			
30																																			
31																																			
32																																			
33																																			
34																																			
35																																			

- QT GUI 用户友好界面

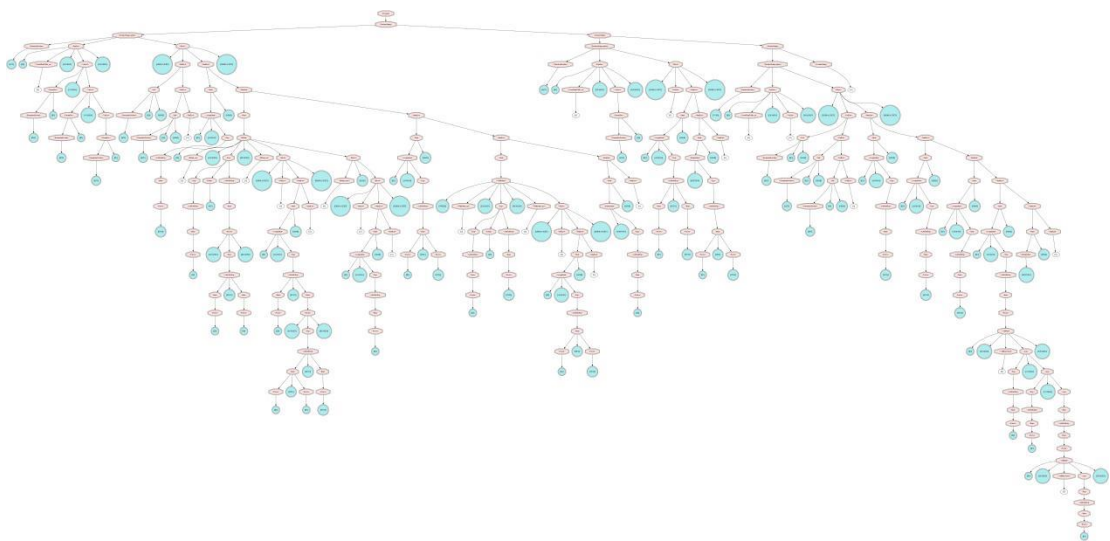


● 语法分析结果的生成

```
process.txt  + - x
1  [INT]
2  -----[INT] => <FunctionDeclare>
3  [ID]
4  -----[z] => <CreateFunTable_m>
5  [LPAREN]
6  [INT]
7  -----[INT] => <ParameterDeclare>
8  [ID]
9  -----[ID]<ParameterDeclare> => <ParamDec>
10 [COMMA]
11 [INT]
12 -----[INT] => <ParameterDeclare>
13 [ID]
14 -----[ID]<ParameterDeclare> => <ParamDec>
15 [COMMA]
16 [INT]
17 -----[INT] => <ParameterDeclare>
18 [ID]
19 -----[ID]<ParameterDeclare> => <ParamDec>
20 -----<ParamDec> => <VarList>
21 -----<VarList>[COMMA]<ParamDec> => <VarList>
22 -----<VarList>[COMMA]<ParamDec> => <VarList>
23 [RPAREN]
24 -----[RPAREN]<VarList>[LPAREN]<CreateFunTable_m>[ID] => <FunDec>
25 [LBBRACKET]
26 [INT]
27 -----[INT] => <ParameterDeclare>
28 [ID]
29 [SEMI]
30 -----[SEMI][ID]<ParameterDeclare> => <Def>
31 [INT]
32 -----[INT] => <ParameterDeclare>
33 [ID]
34 [SEMI]
35 -----[SEMI][ID]<ParameterDeclare> => <Def>
36 -----[z] => <DefList>
37 -----<DefList><Def> => <DefList>
38 -----<DefList><Def> => <DefList>
39 [ID]
40 [ASSIGN]
41 [NUM]
```

● 语法分析树的生成

利用 Graphviz 实现了树状图的绘制，使得语句归结的过程更加清楚明了。



1.1.5 语义分析与中间代码生成

语义分析与中间代码生成模块完成两个任务：静态语义检查和四元式中间代码翻译，与语法分析中生成语法分析树的工作同时进行，在语法分析结束后在前端输出结果。

其中，静态语义检查主要包括类型检查、一致性检查、变量作用域检查等，若发现错误，会在前端控制台输出相应报错信息。

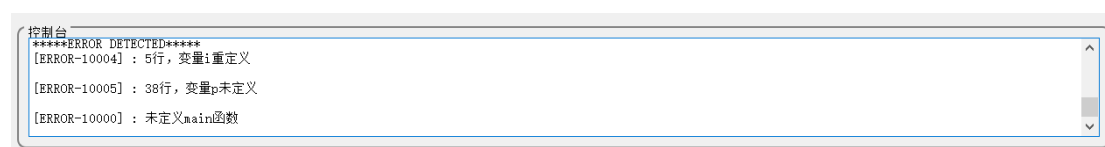
生成的中间代码为四元式形式，根据语法分析传递来的语法符号流以及当前产生式，执行相应语义动作，产生相应的四元式，分析完毕后输出在前端窗口。

● 静态语义检查报错示例

输入程序：

```
int program(int a,int b, int c)
{
    int i;
    int j;
    int i;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    //kongqingchenhaoshuai
    i=j*2;
    while(i<=100)
    {
        i=i*2;
    }
    return i;
}
void main0()
{
    int a;
    int b;
    int c;
    a=3;
    b=4;
    c=2;
    p = 6;
    a=program(a,b,demo(c));
    return ;
}
```

控制台输出错误信息：



● 中间代码生成示例

输入程序:

```
int program(int a,int b, int c)
{
    int i;
    int j;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    //kongqingchenhaoshuai
    i=j*2;
    while(i<=100)
    {
        i=i*2;
    }
    return i;
}
```

```
int demo(int a)
{
    a=a+2;
    return a*2;
}

/*wo
weishenme
zhemecai*/
void main()
{
    int a;
    int b;
    int c;
    a=3;
    b=4;
    c=2;
    a=program(a,b,demo(c));
    return ;
}
```

前端输出:



中间代码文档:

0 (j,-,39)	26 (j,-,27)
1 (program,-,-)	27 (*,i,2,T7)
2 (defpar,-,-,a)	28 (=,T7,-,i)
3 (defpar,-,-,b)	29 (j,-,21)
4 (defpar,-,-,c)	30 (=,i,-,program_return_value)
5 (=,0,-,i)	31 (return,program_return_value,-,program)
6 (+,b,c,T0)	32 (demo,-,-)
7 (j>,a,T0,10)	33 (defpar,-,-,a)
8 (=,0,-,T1)	34 (+,a,2,T8)
9 (j,-,11)	35 (=,T8,-,a)
10 (=,1,-,T1)	36 (*,a,2,T9)
11 (j=,T1,0,18)	37 (=,T9,-,demo_return_value)
12 (j=-,-,13)	38 (return,demo_return_value,-,demo)
13 (*,b,c,T2)	39 (main,-,-)
14 (+,T2,1,T3)	40 (=,3,-,a)
15 (+,a,T3,T4)	41 (=,4,-,b)
16 (=,T4,-,j)	42 (=,2,-,c)
17 (j,-,19)	43 (param,c,-,-)
18 (=,a,-,j)	44 (call,demo,-,T10)
19 (*,j,2,T5)	45 (param,T10,-,-)
20 (=,T5,-,i)	46 (param,b,-,-)
21 (j<=,i,100,24)	47 (param,a,-,-)
22 (=,0,-,T6)	48 (call,program,-,T11)
23 (j,-,25)	49 (=,T11,-,a)
24 (=,1,-,T6)	50 (return,-,-,main)
25 (j=,T6,0,30)	

1.2 程序功能

本项目程序完成了类 C 语言编译过程中的词法、语法、语义分析工作，并产生语法分析树与四元式形式表示的中间代码，为后续代码优化与目标代码生成做了准备。程序通过 GUI 界面，读入用户指定的类 C 语言文件，通过预先给定的语法分析规则与词法分析规则进行分析，得到词法分析流和基于文法产生式自动生成的 ACTION 和 GOTO 表，并且在此基础上进行自下而上的 LR(1) 语法分析，判断程序语句是否符合给定的文法规则，若出错，给出相应报错；若语法分析正常运行，在生成语法分析树的同时向语义分析模块传递语法符号流及当前规约所用的产生式，语义分析模块进行静态语义检查，若出错，报相应错误；执行规约对应的语义动作，产生并保存四元式。语法分析完毕时，语义动作和中间代码生成也执行完毕，前端同时输出语法分析过程流、语法分析树和中间代码序列。

1.3 测试数据

我组实现的 LR1 语法分析程序能够包容含有过程调用的类 C 代码，特别的，对含有注释的代码进行了测试，均取得了较好的结果。

1.3.1 测试代码 1（标准的含过程调用的 cpp 文件）

```
1  int a;
2  int b;
3  int program(int a, int b, int c)
4  {
5      int i;
6      int j;
7      i = 0;
8      if (a > (b + c))
9      {
10         j = a + (b * c + 1);
11     }
12     else
13     {
14         j = a;
15     }
16     while (i <= 100)
17     {
18         i = j * 2;
19     }
20     return i;
21 }
22 int demo(int a)
23 {
24     a = a + 2;
25     return a * 2;
26 }
27 void main(void)
28 {
29     int a;
30     int b;
31     int c;
32     a = 3;
33     b = 4;
34     c = 2;
35     a = program(a, b, demo(c));
36     return;
37 }
```

1.3.2 测试代码 2（含过程调用、注释的较为复杂的 cpp 文件）

```
1  int program(int a,int b, int c)
2  {
3      int i;
4      int j;
5      i=0;
6      if(a>(b+c))
7      {
8          j=a+(b*c+1);
9      }
10     else
11     {
12         j=a;
13     }
14     //kongqingchenhaoshuai
15     i=j*2;
16     while(i<=100)
17     {
18         i=i*2;
19     }
20     return i;
21 }
22
23 int demo(int a)
24 {
25     a=a+2;
26     return a*2;
27 }
28
29 /*wo
30 weishenme
31 zhemecai
32 */ void main()
33 {
34     int a;
35     int b;
36     int c;
37     a=3;
38     b=4;
39     c=2;
40     a=program(a,b,demo(c));
41     return ;
42 }
```

2 概要设计

2.1 任务分解

类 C 语言语义分析器能够读取一份类 C 语言编写的程序源码，在正确识别程序中各个词元的基础上，判断程序语句是否符合给定文法，若符合，能够呈现语法分析过程、语法分析树，并据此产生等效的四元式中间代码；若不符合，能够输出相应的报错信息。将总体任务分解为词法分析、语法分析和语义分析（含中间代码生成）三个部分，主要阐述语义分析部分的设计。

2.1.1 词法分析

词法分析是计算机科学中将字符序列转换为单词序列的过程，是编译过程的第一个阶段，也是编译的基础。这个阶段的任务是从左到右一个字符、一个字符地读入源程序，通过对构成源程序的字符流进行扫描，根据构词规则识别出现的所有单词。词法分析器通常不关心单词之间的关系，但由于词法分析阶段需要过滤注释，这边在词法分析器中加入了对多行注释起止符号（/*、*/）的匹配检测。

- 词法分析器识别的单词类型：

- ①关键字：由程序语言定义的具有固定意义的标识符。称这些标识符为保留字或基本字。

- ②标识符：是用来表示用户自定义的名字，如变量、数组名、过程名等等。

- ③常数：是程序运行过程中不能改变的值，类型包括整型、布尔型、字符串常量类型等。

- ④运算符：承担执行程序代码运算功能。如+、-、*、/等等。

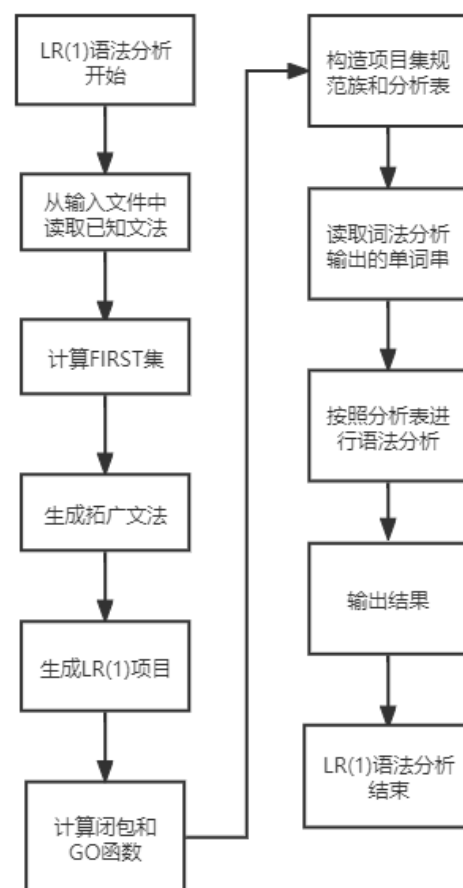
- ⑤界符：如逗号、分号、括号、/*，*/等等。

2.1.2 语法分析

语法分析是编译过程的一个逻辑阶段，其任务是在词法分析的基础上将单词序列组合成各类语法短语，判断源程序在结构上是否正确。本项目中，语法

分析器的实现基于自下而上分析法中的 LR(1) 分析法，在逻辑上由一个输入符号串，一个下推分析栈，以及一个总控程序和分析表组成。LR 语法分析器在总控程序的控制下自左至右扫视输入串的各个符号，并根据当前分析栈中所存放之文法符号的状况及正注视的输入符号，按分析表的指示完成相应的分析动作。在分析的每一时刻，分析栈中记录了迄今为止所移进或归约出的全部文法符号，即记录了从分析开始到目前为止的整个历程。

• LR(1) 语法分析的分解步骤



2.1.3 语义分析

语义分析是编译过程的一个逻辑阶段，本项目设计中，语义分析模块的功能包括静态语义检查和中间代码生成。其中，静态语义检查的任务是对结构上正确

的源程序进行上下文有关性质的类型审查；中间代码生成的任务是根据规约式及语法符号流执行相应语义动作，并产生四元式。

①静态语义检查支持的错误检测：

*main 函数未定义

```
//如果没有定义main函数，则报错
if (main_index == -1) {
    errorInfo.push_back("[ERROR-10000] : 未定义main函数");
}
```

*变量重定义检测

```
//用于判断该变量是否在当前层已经定义
bool existed = false;
SemanticSymtable current_table = allTable[cur_tableStack.back()];
if (current_table.findSym(ident.toke.content) != -1) {
    errorInfo.push_back("[ERROR-10001] : " + to_string(ident.toke.line) + "行，变量" + ident.toke.content + "重定义");
}
```

*函数重定义检测

```
//首先在全局的table判断函数名是否重定义
if (allTable[0].findSym(ident.toke.content) != -1) {
    errorInfo.push_back("[ERROR-10002] : " + to_string(ident.toke.line) + "行，函数" + ident.toke.content + "重定义");
}
```

*函数参数重定义检测

```
//如果已经进行过定义
if (function_table.findSym(ident.toke.content) != -1) {
    errorInfo.push_back("[ERROR-10003] : " + to_string(ident.toke.line) + "行，函数参数" + ident.toke.content + "重定义");
}
```

*变量未定义错误检测

```
//检查id是否存在，不存在则报错
bool existed = false;
int table_index = -1, index = -1;
//从当前层开始向上遍历
for (int scope_layer = cur_tableStack.size() - 1; scope_layer >= 0; scope_layer--) {
    auto current_table = allTable[cur_tableStack[scope_layer]];
    if ((index = current_table.findSym(ident.toke.content)) != -1) {
        existed = true;
        table_index = cur_tableStack[scope_layer];
        break;
    }
}
if (existed == false) {
    errorInfo.push_back("[ERROR-10005] : " + to_string(ident.toke.line) + "行，变量" + ident.toke.content + "未定义");
}
```

*非法参数检测

```

//检查参数个数
int para_num = allTable[check.table_index].Symtable[check.index].paraNum;
if (para_num > stoi(args.token.content)) {
    errorInfo.push_back("ERROR-10007 : " + to_string(identifier.token.line) + "行, 函数" + identifier.token.content + "调用时参数过少");
}
else if (para_num < stoi(args.token.content)) {
    errorInfo.push_back("ERROR-10008 : " + to_string(identifier.token.line) + "行, 函数" + identifier.token.content + "调用时参数过多");
}

```

*函数未定义错误检测

```

SemanticSymbol fun_id = symbolList[symbolList.size() - 2];

int fun_id_pos = allTable[0].findSym(fun_id.token.content);

if (-1 == fun_id_pos) {
    errorInfo.push_back("ERROR-10009 : " + to_string(fun_id.token.line) + "行, 函数" + fun_id.token.content + "调用未定义");
}
if (allTable[0].Symtable[fun_id_pos].type != FUNC) {
    errorInfo.push_back("ERROR-10010 : " + to_string(fun_id.token.line) + "行, 函数" + fun_id.token.content + "调用未定义");
}

```

*遗漏返回值错误检测

```

//检查函数的返回值是否为void
if (allTable[0].Symtable[allTable[0].findSym(function_table.name)].specialType != "void") {
    errorInfo.push_back("ERROR-10011 : " + to_string(symbolList.back().token.line) + "行, 函数" + function_table.name + "必须有返回值");
}

```

③语义动作和中间代码生成

根据当前产生式的不同，语义动作与相应中间代码规定如下：

<Program> → <DeclareString>

规约含义：最后一个归结式，（变量与函数的）定义构成类 C 语言程序

语义动作：1. 产生跳转至 main 函数的四元式（j, -, -, main），插入至四元式表开头

2. 语法符号栈中弹出被规约符号

3. <Program>压入语法符号栈

（后面将 2、3 步统称“更新语法符号栈”）

<DeclareString-option> → <ParameterDeclare> [ID] [SEMI]

规约含义：识别出一个全局变量定义

语义动作：1. 变量信息插入当前符号表

2. 更新语法符号栈

<DeclareString-option> -><FunctionDeclare> <FunDec> <Block>

规约含义：识别出一个完整的函数定义

语义动作：1. 退出作用域，从符号表栈中弹出当前符号表

2. 更新语法符号栈

<ParameterDeclare> ->[INT]

规约含义：所定义参数的类型是整型 INT

语义动作：更新语法符号栈

<FunctionDeclare> ->[VOID] | [INT]

规约含义：识别出函数定义的返回值类型，是空 VOID 或整型 INT

语义动作：更新语法符号栈

<FunDec> ->[ID] <CreateFunTable_m>[LPAREN] <VarList>[RPAREN]

规约含义：识别出函数除返回值外的声明部分，包括函数名和参数列表

语义动作：更新语法符号栈

<CreateFunTable_m> ->[z]

规约含义：即将声明一个函数，准备创建函数符号表

语义动作：1. 获取函数名和返回类型，创建新的函数表，保存存储所有符号表的结构 allTable 中

2. 在全局符号表(allTable[0])中创建当前函数的符号项，参数个数和入口地址未知，待回填

3. 新函数表索引压入当前符号表索引栈 cur_tableStack

4. 函数表中插入代表返回值的变量

5. 若为 main 函数，保存当前四元式地址 q_index
6. 生成四元式([funcID], -, -, -)，定义函数头。q_index++
7. 更新语法符号栈

<ParamDec> -> <ParameterDeclare>[ID]

规约含义：识别出函数的一个形式参数

语义动作：1. 在当前函数符号表中插入该形参

2. 全局符号表中记录的当前函数的形参个数++

3. 生成四元式(defpar, -, -, [param])，定义当前形参。q_index++

4. 更新语法符号栈

<Block> ->[LBBRACKET] <DefList> <StmtList>[RBBRACKET]

规约含义：识别出一个语句块

语义动作：更新语法符号栈

<Def> -> <ParameterDeclare>[ID] [SEMI]

规约含义：识别出局部变量定义

语义动作：1. 在当前符号表中插入新定义符号

2. 更新语法符号栈

<AssignStmt> ->[ID] [ASSIGN] <Exp>

规约含义：识别出赋值语句

语义动作：1. 生成四元式(=, [exp].name, -, [id].name)

2. 更新语法符号栈

<Exp> -> <AddSubExp>

规约含义：识别出表达式

语义动作：更新语法符号栈

<Exp1> → <Exp2> <Relop> <AddSubExp>

规约含义：识别出带关系运算符的表达式

语义动作：1. 产生中间变量 newTemp，表示 Exp1

2. 产生四元式(j[relop], exp2.name, addSubExp.name, q_index+3),

q_index++

3. 产生四元式(=, 0, -, newTemp), q_index++

4. 产生四元式(j, -, -, q_index+2), q_index++

5. 产生四元式(=, 1, -, newTemp), q_index++

6. 更新语法符号栈

**<AddSubExp> → <Item> | <Item1>[PLUS] <Item2> | <Item1>[SUB]
<Item2>**

规约含义：识别出（外层）加减表达式

语义动作：1. 产生中间变量 newTemp，表示 addSubExp

2. 产生四元式(op, item1, item2, newTemp), q_index++

3. 更新语法符号栈

**<Item> → <Factor> | <Factor>[MUL] <Factor> | <Factor>[DIV]
<Factor>**

规约含义：识别出（内层）乘除表达式

语义动作：1. 产生中间变量 newTemp，表示 addSubExp

2. 产生四元式(op, item1, item2, newTemp), q_index++

3. 更新语法符号栈

<Factor> ->[NUM] | [LPAREN] <Exp>[RPAREN] | [ID] | <CallStmt>

规约含义：识别出原子表达式

语义动作：更新语法符号栈

<CallStmt> ->[ID] [LPAREN] <CallFunCheck> <Args>[RPAREN]

规约含义：识别出函数调用

语义动作：1. 产生中间变量 newTemp，表示函数返回值

2. 产生四元式(call, [func], -, newTemp), q_index++

3. 更新语法符号栈

<CallFunCheck> ->[z]

规约含义：准备进行函数调用，执行预检查工作

语义动作：检查函数是否有定义

<Args> -> <Exp> [COMMA] <Args> | <Exp> | [z]

规约含义：识别出函数调用的实参（列表）

语义动作：1. 产生四元式(param, [exp].name, -, -)，为即将被调用的函数传递参数，q_index++

2. 记录已传参数数目

3. 更新语法符号栈

<ReturnStmt> ->[RETURN] <Exp>

规约含义：识别出返回语句

语义动作：1. 生成四元式(=, exp, -, [func_return])，给返回值赋值，
func_return = allTable[cur_tableStack.back()][0], q_index++

2. 生成四元式 (return, [func_return], -, [func_name]), 函数返回, q_index++

3. 更新语法符号栈

<ReturnStmt> -> [RETURN]

规约含义：识别出返回语句

语义动作：1. 生成四元式 (return, [func_return], -, [func_name]), 函数返回, q_index++

2. 更新语法符号栈

<Relop> -> [GT] | [LT] | [GE] | [LE] | [EQ] | [NQ]

规约含义：识别出关系运算符

语义动作：更新语法符号栈

<IfStmt> -> [IF] <IfStmt_m1> [LPAREN] <Exp> [RPAREN] <IfStmt_m2>

<Block> <IfNext>

规约含义：识别出 IF 语句

语义动作：1. 若有 else 语句，首先用当前 q_index 回填跳出四元式，回填列表 pop

2. 用 IfStmt_m2 记录的地址（if 为真时首条语句）回填真出口四元式，回填列表 pop

3. 用 IfStmt_next 记录的地址（if 为假时首条语句）回填假出口四元式，回填列表 pop

4. 回填层次 backpatching—

5. 更新语法符号栈

<IfStmt_m1> -> [z]

规约含义：进入 if 语句前，记录回填层次

语义动作：1. 回填层次 `backpatching_level++`

2. 记录当前四元式地址，压入语法符号栈

<IfStmt_m2> ->[z]

规约含义：读完 if 语句条件，分别产生假出口和真出口的待回填四元式

语义动作：1. 产生假出口四元式 (`j`, `[if_condition]`, `0`, `-`)，地址入回填列表，`q_index++`

2. 产生真出口四元式 (`j`, `-`, `-`, `-`)，地址入回填列表，`q_index++`

3. 记录当前四元式地址，压入语法符号栈

<IfNext> -> <IfStmt_next>[ELSE] <Block>

规约含义：识别出 ELSE 分支

语义动作：更新语法符号栈

<IfStmt_next>->[z]

规约含义：即将进入 else 分支，生成待回填的 else 分支前的跳出四元式

语义动作：1. 产生跳出四元式 (`j`, `-`, `-`, `-`)，地址入回填列表，`q_index++`

2. 记录当前四元式地址，压入语法符号栈

<WhileStmt>->[WHILE]<WhileStmt_m1>[LPAREN]<Exp>[RPAREN]

<WhileStmt_m2> <Block>

规约含义：识别出 WHILE 语句

语义动作：1. 产生四元式 (`j`, `-`, `-`, `WhileStmt_m1.addr`)，无条件跳转至 while 条件判断语句处，`q_index++`

2. 用 `WhileStmt_m2` 存入的地址回填真出口四元式，回填列表 pop

3. 用当前四元式地址回填假出口四元式，回填列表 pop
4. 回填层次 backpatching_level—
5. 更新语法符号栈

<WhileStmt_m1> ->[z]

规约含义：进入 while 语句前，记录回填层次

- 语义动作：
1. 回填层次 backpatching_level++
 2. 记录当前四元式地址，压入语法符号栈

<WhileStmt_m2> ->[z]

规约含义：读完 while 语句条件，分别产生假出口和真出口的待回填四元式

- 语义动作：
1. 产生假出口四元式(j , [while_condition], 0, -)，地址入回填列表，q_index++
 2. 产生真出口四元式(j , -, -, -)，地址入回填列表，q_index++
 3. 记录当前四元式地址，压入语法符号栈

Others

语义动作：更新语法符号栈

③符号表管理

采用了动态创建和删除的符号表管理策略。在语义分析模块中内置一个符号表列表，用以存储所有创建的符号表并可以按照索引访问，其中[0]是全局符号表；同时维护一个当前符号表索引栈，栈顶是当前符号表在符号表列表中的索引。每新定义一个函数，就在总符号表列表中新建一张符号表，同时将其索引压入索引栈，在函数定义结束前都可以直接利用栈顶索引对当前符号表进行

查找和修改，函数定义结束时，弹出符号表索引，就自动回到上一级符号表（通常是全局符号表）。条理清晰，实现也较为简单。

④代码优化设计

代码优化从四个方面进行。

1. 常量传播

常量传播是一个渗透在中间代码生成和优化中间代码之间的过程，因为语法分析程序的设计特点，分析因子的时候如果是常数会返回常数的值，所以在生成中间代码时，如果有两个操作数都是常数，会把运算指令转化为赋值语句。

具体实现方法是，在执行语义动作时，引入常量标识，在规约表达式时单独处理并直接计算出由常量构成的子表达式的值，从而在生成的代码中直接进行常量结果赋值。

2. 消除块内公共子表达式

如果代码中存在重复的子表达式，每遇到一次都要引入一个新的临时变量，并且重新计算一次，这会造成代码的冗余。这部分优化所做的事情是把后出现的、具有相同子表达式的临时变量删除，所有对它的引用都用先出现的临时变量替换。为此，将代码按块划分，函数域、IF 域、ELSE 域、WHILE 域都构成一个独立的块，只有在同一各区块内出现的子表达式可以相互替换，并且需要检查在两个子表达式的四元式之间，涉及到的变量是否出现值的变化，只有值无变化的情况下，才可以替换。

3. 消除无用代码

考虑到有些临时变量可能并未用于跳转判断或对变量的赋值，那么对它们的运算和赋值是无意义的操作，这部分优化删去了这些无用代码。

4. 窥孔优化

对中间代码的窥孔优化主要针对一个翻译过程中出现的问题，当把一个表达式的值赋给一个变量的时候，只要表达式不是一个单独的常数或变量，就出现类似下面的代码：

```
ADD #1 a, b
```

```
ASSIGN c #1
```

这样的代码只要有比较复杂的赋值语句就一定会出现，所以对其进行窥孔优化效果是十分明显的，考虑将其收缩成一条：

```
ADD c a, b
```

2.2 数据类型定义

2.2.1 词法分析相关

单词类 Token 与词法分析器类 LexAnalyzer

```
typedef struct {
    int line;
    string content;
    LexComponent type;
} Token;

class LexAnalyzer {
public:
    LexAnalyzer() { init(); }
    int resultNum; // 分析结果vector的元素数量
    void init() { resultNum = 0; curLine = 1; LexResult.clear(); while (!Result.empty()) { Result.pop(); } } // 重置
    bool lex_analyze(string inFile);
    void output_result(string outFile);
    void getResult(); // LexResult->Result
    int curLine; // 当前扫描的行数
    queue<Symbol>Result; // 输出给语法分析
    vector<Token>LexResult; // 存储词法分析结果
};
```

2.2.2 语法分析相关

①**Symbol 类**: 定义文法符号, 其中 content 与词法分析结果 type 字段相关联, 指明符号为括号、分号等, symbol_type 将符号划分为终结符 (VT), 非终结符 (VN), 以及拓广文法开始符号 S' 等。

```
/*文法符号类*/
#define VN 0
#define VT 1
#define EPSILON 2
#define END 3
#define symEps Symbol(EPSILON, "[z]")
#define symEnd Symbol(END, "[#]")
#define symS0 Symbol(VN, "<S0>")

class Symbol {
    /*定义文法符号*/
public:
    int symbol_type; //符号类型
    string content; //存放内容
    friend bool operator==(const Symbol& s1, const Symbol& s2);
    friend bool operator!=(const Symbol& s1, const Symbol& s2);
    friend bool operator< (const Symbol& s1, const Symbol& s2);
    friend ostream& operator<<(ostream& out, const Symbol& s);
    Symbol();
    Symbol(const int& st, const string& c);
    Symbol(const Symbol& s);
};
```

②**Production 类**: 定义产生式类, 左部为 Symbol 类型, 右部定义了一个 Symbol 向量, 用于保存所产生的文法符号。

```
/*产生式类*/
class Production {
public:
    Symbol Left; //左部
    vector<Symbol> Right; //右部
    friend ostream& operator<<(ostream& out, const Production& p);
    friend bool operator==(const Production& p1, const Production& p2);
    friend bool operator<(const Production& p1, const Production& p2);
};
```

③**Event 类**: 用于生成 LR1 文法所需的项目类型, 含有一个 Production、• 符号所在的位置以及前瞻展望符号。

```
/*项目类*/
//形如A->.Ba
class Event {
public:
    Production prod; //对应的production
    int dotPos; //对应点的位置
    Symbol ahead; //展望符号
    Event(const Production& p, const int& pos, const Symbol& a);
    friend ostream& operator<<(ostream& out, const Event& e);
    friend bool operator==(const Event& ev1, const Event& ev2);
    friend bool operator<(const Event& ev1, const Event& ev2);
};
```

④EventClosure 类：项目闭包类，即一组 Event 的向量类。

⑤LR1 类：语法分析器类，包含程序执行与 DEBUG 所需的全部信息。

• 类成员：

```
/*LR1类*/
class LR1 {
public:
    /*词法*/
    LexAnalyzer lex;

    /*符号与产生式*/
    vector<Production> generator;//产生式集合
    set<Symbol> vSet;//文法符号集合
    set<Symbol> vnSet;//非终结符集合
    set<Symbol> vtSet;//终结符集合
    map<Symbol, set<Symbol>> first_set;//FIRST集

    /*项目与项目集族*/
    vector<Event> items;//拓广文法项目集合,S' -> S.#,
    vector<EventClosure> EventCluster;//项目闭包集构成的项目集族

    /*DFA转移信息*/
    map<pair<int, Symbol>, int> GotoInfo;//DFA转移信息
    map<pair<int, Symbol>, op> GotoTable;//GOTO表: 只包含转移op, 不讨论归约
    map<pair<int, Symbol>, op> ActionTable;//Action表: ACTION[i,a]=op

    /***/
};
```

• 成员函数：

```
/***/
/*符号与产生式*/
int read_generators(const string dat_path); //读取产生式
void get_firstset_of_vn(); //获取所有符号first集
set<Symbol> get_firstset_of_string(const vector<Symbol>& str); //获取给定符号串first集

/*项目闭包*/
void init_items();//根据generator生成拓广文法的items集合, 生成项目闭包集合的vector

/*ACTION & GOTO*/
void getTable();//根据DFA转移信息, 生成ACTION表与GOTO表
void printTable(const string file_path);//打印ACTION表与GOTO表输出到EXCEL文件

/*语法分析*/
void grammarTree(const string tree_path, const string process_path, queue<Symbol>& code);//语法分析, 并生成语法树

/*综合使用*/
void Analysis(const string generator_path, const string TableOut_path, const string TreeOut_path, const string ProcessOut_path);//封装上述步骤, 方便调用
};
```

2.2.3 语义分析相关

①语义符号类 SemanticSymbol

语义符号栈中存储的类型，在语义分析中通过压栈和弹出传递必要的信息

```

struct SemanticSymbol
{
    /*语义分析符号*/
    TokSym toke: //int line ,string content, string type
                //{line为token中的Line, content为string, 这个token的具体内容,
                //最后的type是identifier或int 或者和content内容相同}
    int table_index://符号所在table的index
    int index://符号在table内部的index
};

```

②四元式类 Quadruple

直接对应生成的四元式中间代码。

```

struct Quadruple
{
    /*四元式*/
    int index://索引号
    string op://运算符
    string arg1;
    string arg2;
    string res;
};

```

③标识符类 Identifier

符号表中存储的类型，包含了变量和函数标识符所必要的信息

```

struct Identifier
{
    /*标识符信息*/
#define FUNC 0 //函数
#define VAR 1 //变量
#define TEMP 2 //临时变量
#define CONNST 3 //常量
#define RET 4 //返回值

    int type: //上述5种类型
    string specialType: //函数—返回类型 常量—类型 变量—类型
    string name: //标识符名称,如果是常量则为取值
    int paraNum: //函数—参数个数
    int entry: //函数—入口地址
    int tableIndex: //函数符号表索引
};

```

④符号表类 SemanticSymtable

符号表，主要成员是一个标识符 vector，存储表中定义的标识符

```
class SemanticSymtable
{
    /*符号表*/
#define GLOBAL 0
#define FUNC 1
#define BLOCK 2

public:
    int type; //上述3种类型
    string name; //符号表名称
    vector<Identifier> Symtable; //符号表
    SemanticSymtable(const int t_type, const string& t_name);
    int findSym(const string& id_name); //查找符号,返回index
    int insertSym(const Identifier& id); //插入符号,返回index
};
```

⑤语义分析器类 SemanticAnalyzer

在语法分析的过程中接收语法符号流和当前产生式，调用分析程序选择合适的语义动作函数。

```

class SemanticAnalyzer
{
    /*语义分析类*/
public:
    vector<Quadruple> quadruple;//四元式表
    int main_index;//main函数对应的四元式标号
    int backpatching_level;//回跳层次
    vector<int> backpatching_list;//回跳列表
    int nextQ_index;//下一个四元式标号
    int cnt;//临时变量计数
    vector<SemanticSymbol> symbolList;//语义分析过程的符号流
    vector<SemanticSyntable> allTable;//程序所有符号表
    vector<int> cur_tableStack;//当前作用域对应的符号表索引栈
    vector<string>errorInfo;

public:
    //构造
    SemanticAnalyzer();
    //插入符号
    void insertSymbol(const SemanticSymbol& sym);//将所有的符号信息放入symbolList
    //四元式表输出
    void printQuadruple(const string outpath);
    //语义分析主体
    void semanticANL(const Production& pro);//传入参数为产生式的两侧

private:
    void SemanProd_Program(const Production& pro);
    void SemanProd_DecOption(const Production& pro);
    void SemanProd_ParaDec(const Production& pro);
    void SemanProd_FunctionDeclare(const Production& pro);
    void SemanProd_FunDec(const Production& pro);
    void SemanProd_CreateFunTable_m(const Production& pro);
    void SemanProd_ParamDec(const Production& pro);
    void SemanProd_Block(const Production& pro);
    void SemanProd_Def(const Production& pro);
    void SemanProd_AssignStmt(const Production& pro);
    void SemanProd_Exp(const Production& pro);
    void SemanProd_AddSubExp(const Production& pro);
    void SemanProd_Item(const Production& pro);
    void SemanProd_Factor(const Production& pro);
    void SemanProd_CallStmt(const Production& pro);
    void SemanProd_CallFunCheck(const Production& pro);
    void SemanProd_Args(const Production& pro);
    void SemanProd_ReturnStmt(const Production& pro);
    void SemanProd_Relop(const Production& pro);
    void SemanProd_IfStmt(const Production& pro);
    void SemanProd_IfStmt_m1(const Production& pro);
    void SemanProd_IfStmt_m2(const Production& pro);
    void SemanProd_IfNext(const Production& pro);
    void SemanProd_IfStmt_next(const Production& pro);
    void SemanProd_WhileStmt(const Production& pro);
    void SemanProd_WhileStmt_m1(const Production& pro);
    void SemanProd_WhileStmt_m2(const Production& pro);
};

```

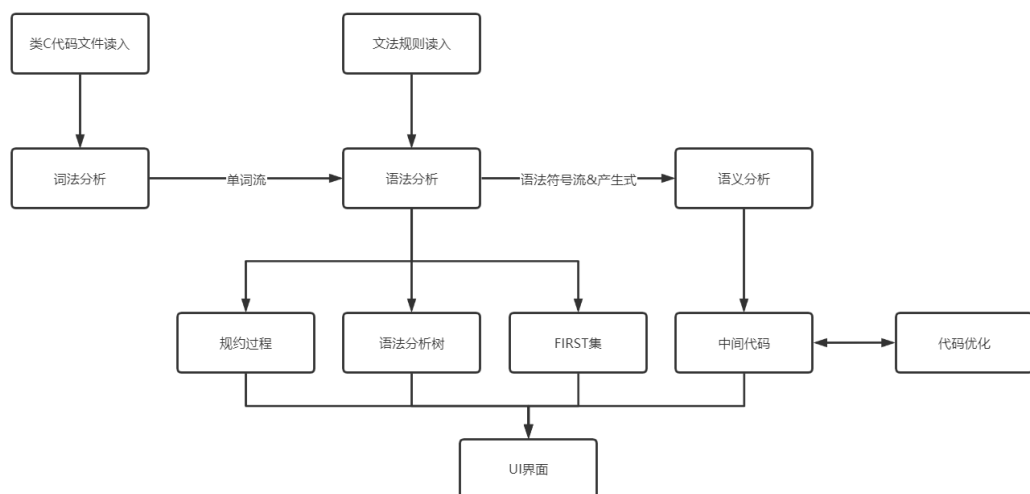
⑥四元式类型及含义

形式	含义
(j, -, -, [addr])	无条件跳转指令，[addr]指示跳转目标四元式的地址
(j[op], [a], [b], [addr])	条件跳转指令，若[a][op][b]为真(op为关系运算符)，则跳转至[addr]

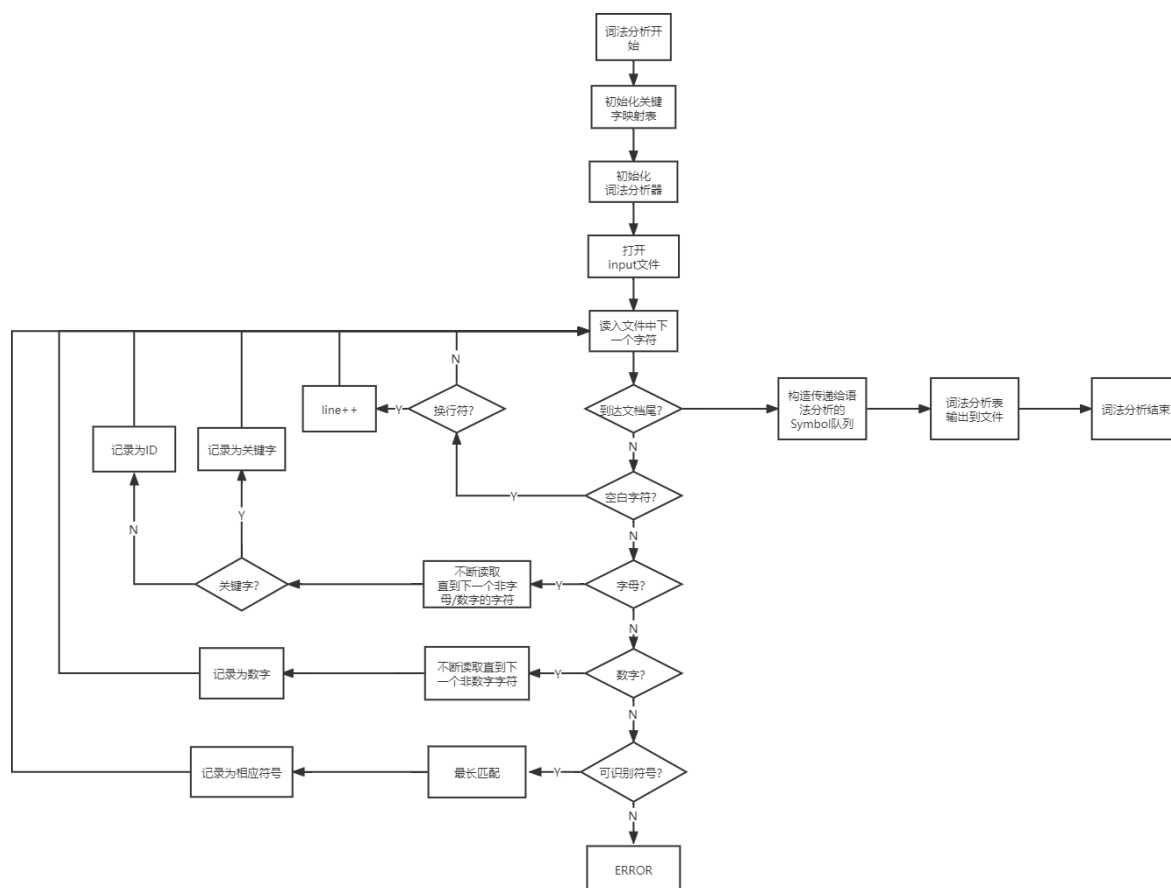
([func], -, -, -)	函数声明指令
(defpar, -, -, [par])	函数形参声明指令
(=, [a], -, [b])	赋值指令, [b]:=[a]
([op], [a], [b], [c])	赋值指令, [c]:=[a][op][b], [op]为运算符
(param, [par], -, -)	调用函数前, 向目标函数传递实际参数[par], 须按顺序
(call, [func], -, [ret])	函数调用指令, 调用函数[func], 规定返回值为[ret]
(call, [func], -, -)	函数调用指令, 调用函数[func]
(return, [ret], -, [func])	函数返回指令, [func]函数返回, 返回值为[ret]
(return, -, -, [func])	函数返回指令, [func]函数返回

2.3 主程序流程

①总体流程

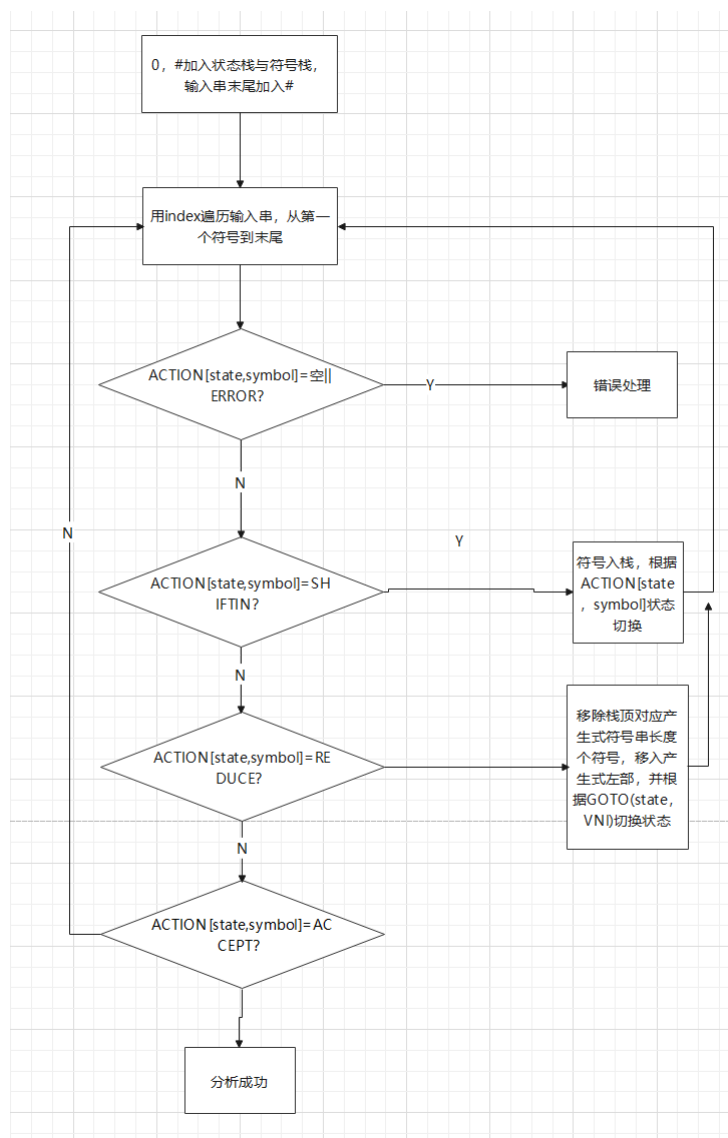


②词法分析器内部流程



③语法分析器内部流程

1) 主控程序



2) 总体流程简述

读取给定文法文本中的内容，产生终结符集、非终结符集以及产生式集合；根据非终结符集合计算对应非终结符的 FIRST 集；根据产生式集合生成拓广文法以及项目集合；计算项目集闭包与转移函数，得到 DFA 状态转移表与项目集族；根据状态转移表与项目集族生成 ACTION 与 GOTO 分析表；通过词法分析模块传递而来的输入串序列，结合 LR1 分析表执行主控程序所示分析方法，实现语法分析。

④语义分析器内部流程



3 详细设计

3.1 主要函数分析及设计

①词法分析器主要函数

<code>void init();</code>	初始化词法分析器，各变量及数组清零
---------------------------	-------------------

<code>bool lex_analyze(string inFile);</code>	以输入文件路径为参数，对文件内代码进行词法分析，结果储存在类内数组结构中，返回值代表词法分析是否成功
<code>void output_result(string outFile);</code>	以输出文件路径为参数，将词法分析结果按简易表格形式输出至文件
<code>void getResult();</code>	由词法分析结果产生一个可以直接被语法分析器使用的输入队列

②语法分析器主要函数

<code>void read_generators(generator_path)</code>	根据给定文法信息文本，生成终结符集、非终结符集与产生式集。
<code>void get_firstset_of_vn();</code>	对非终结符集合中的所有符号求算 FIRST 集合，并保存。
<code>void init_items();</code>	根据产生式集合构建项目集族与 DFA 转移表。
<code>void getTable();</code>	根据 DFA 转移表与项目集族生成 ACTION 表与 GOTO 表。
<code>void printTable(TableOut_path);</code>	打印 LR1 分析表。
<code>grammartree(TreeOut_path, ProcessOut_path, lex.Result);</code>	进行归约，如果接受则打印语法生成树与归约过程。

--	--

③语义分析器主要函数

<code>SemanticAnalyzer();</code>	语义分析器构造函数
<code>void opt_remove_public_exp_useless_code();</code>	消除公共子表达式和无用代码
<code>void opt_kk();</code>	窥孔优化
<code>void optimize();</code>	优化中间代码，结果存在 opt 里
<code>void insertSymbol(const SemanticSymbol& sym);</code>	向语法符号栈中插入语法符号
<code>void printQuadruple(const string outpath);</code>	打印四元式表
<code>void semanticANL(const Production& pro);</code>	语义分析主体函数
<code>void SemanProd_Program(const Production& pro);</code>	规约到<Program>
<code>void SemanProd_DecOption(const Production& pro);</code>	规约到 < DeclareString-option >
<code>void SemanProd_ParaDec(const Production& pro);</code>	规约到 < ParameterDeclare >
<code>void SemanProd_FunctionDeclare(const Production& pro);</code>	规约到< FunctionDeclare>
<code>void SemanProd_FunDec(const Production& pro);</code>	规约到< FunDec >

void SemanProd_CreateFunTable_m(const Production& pro);	规约到< CreateFunTable_m >
void SemanProd_ParamDec(const Production& pro);	规约到< ParamDec >
void SemanProd_Block(const Production& pro);	规约到< Block >
void SemanProd_Def(const Production& pro);	规约到< Def >
void SemanProd_AssignStmt(const Production& pro);	规约到< AssignStmt >
void SemanProd_Exp(const Production& pro);	规约到< Exp >
void SemanProd_AddSubExp(const Production& pro);	规约到< AddSubExp >
void SemanProd_Item(const Production& pro);	规约到< Item >
void SemanProd_Factor(const Production& pro);	规约到< Factor >
void SemanProd_CallStmt(const Production& pro);	规约到< CallStmt >
void SemanProd_CallFunCheck(const Production& pro);	规约到< CallFunCheck >
void SemanProd_Args(const Production& pro);	规约到< Args >

void SemanProd_ReturnStmt (const Production& pro);	规约到< ReturnStmt >
void SemanProd_Relop (const Production& pro);	规约到< Relop >
void SemanProd_IfStmt (const Production& pro);	规约到< IfStmt >
void SemanProd_IfStmt_m1 (const Production& pro);	规约到< IfStmt_m1>
void SemanProd_IfStmt_m2 (const Production& pro);	规约到< IfStmt_m2>
void SemanProd_IfNext (const Production& pro);	规约到< IfNext >
void SemanProd_IfStmt_next (const Production& pro);	规约到< IfStmt_next >
void SemanProd_WhileStmt (const Production& pro);	规约到< WhileStmt >
void SemanProd_WhileStmt_m1 (const Production& pro);	规约到< WhileStmt_m1>
void SemanProd_WhileStmt_m2 (const Production& pro);	规约到< WhileStmt_m2>

3.2 函数调用关系

①词法分析器函数调用关系

调用者函数	被调用函数
lex_analyze(string inFile)	init(); getResult()

②语法分析器函数调用关系

调用者函数	被调用函数
void LR1::get_firstset_of_vn()	init_firstset(); bool mergeSetNoEpsilon(set<Symbol> & dst, const set<Symbol> & src); bool mergeSet(set<Symbol> & dst, const set<Symbol> & src);
void LR1::init_items()	EventClosure& GetEventClo(LR1& lr1, EventClosure& CloJ); EventClosure GetEventTO(LR1& lr1, EventClosure& CloI, const Symbol sym); void get_extension(LR1& lr1);

③语义分析器函数调用关系

调用者函数	被调用函数
void semanticANL (const Production& pro);	所有规约式处理函数
void optimize();//优化中间代码，结果存在 opt 里	void opt_remove_public_exp_useless_code(); void opt_kk();

4 调试分析

4.1 测试数据及测试结果

4.1.1 正常运行结果展示

①测试所用代码

```

int a;
int b;
int program(int a,int b,int c)
{
    int i;
    int j;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    while(i<=100)
    {
        i=j*2;
    }
    return i;
}

int demo(int a)
{
    a=a+2;
    return a*2;
}

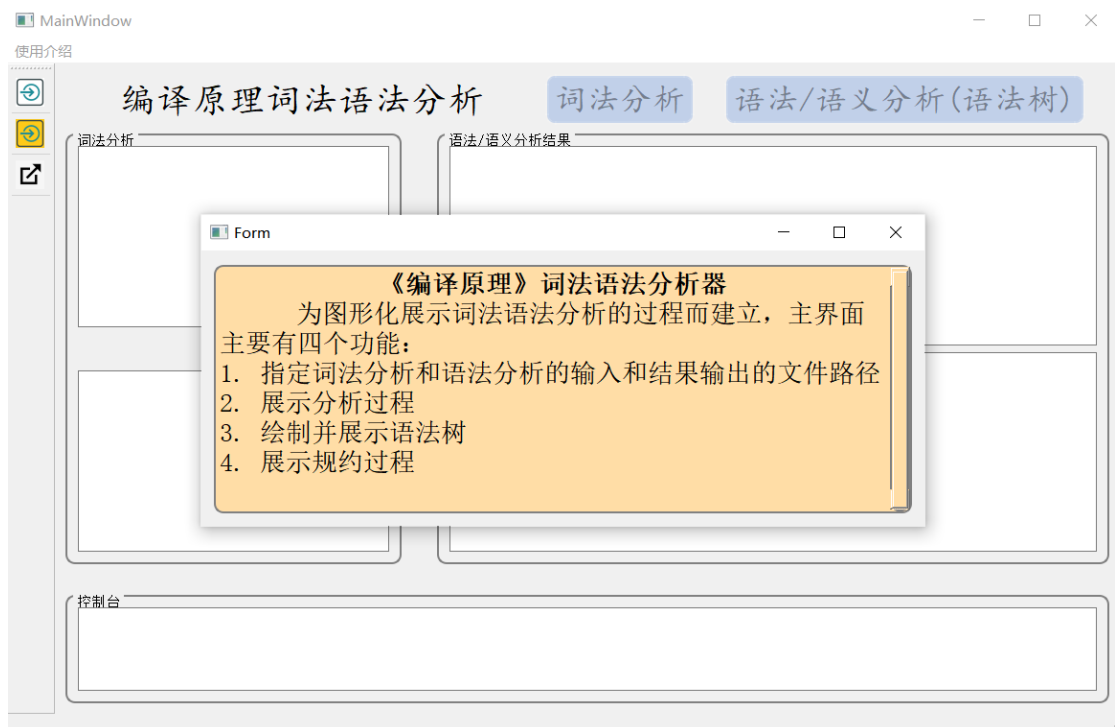
void main(void)
{
    int a;
    int b;
    int c;
    a=3;
    b=4;
    c=2;
    a=program(a,b,demo(c))
    return;
}

```

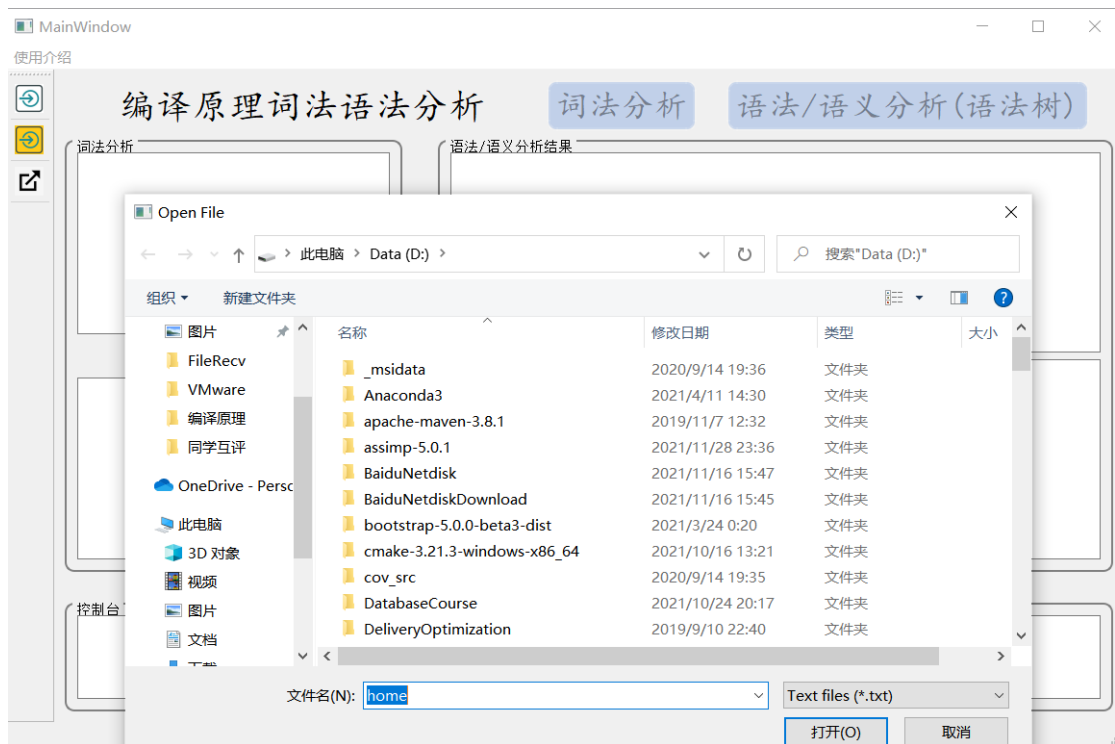
②初始页面



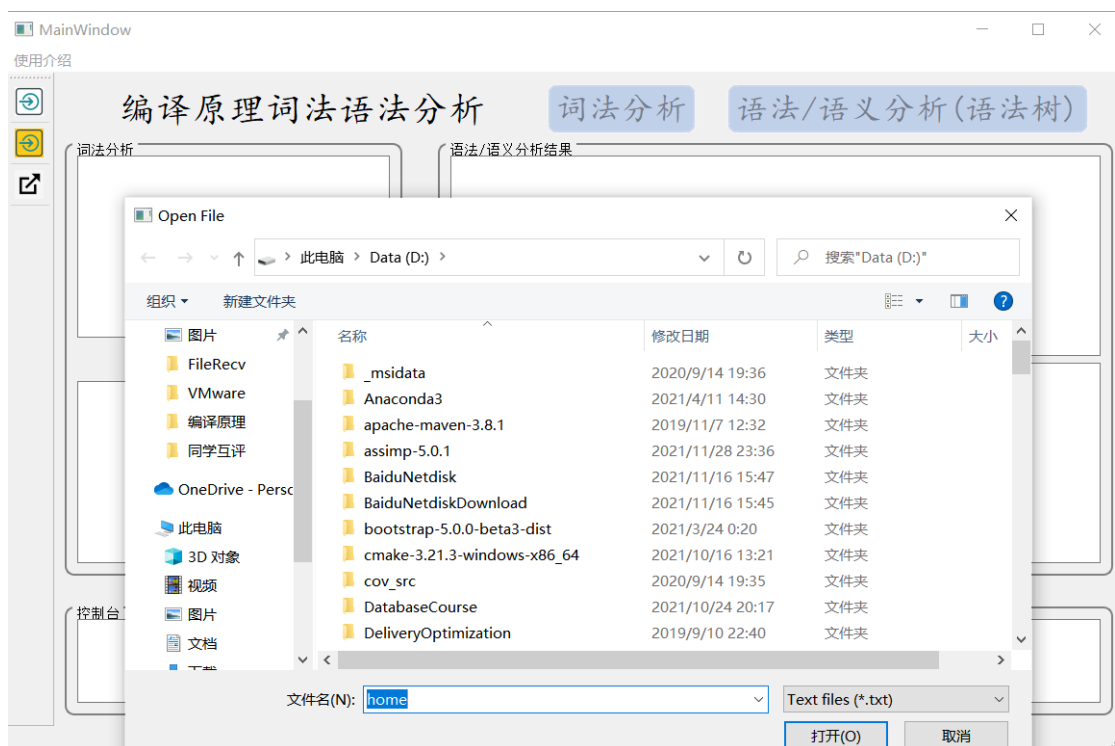
③欢迎页面



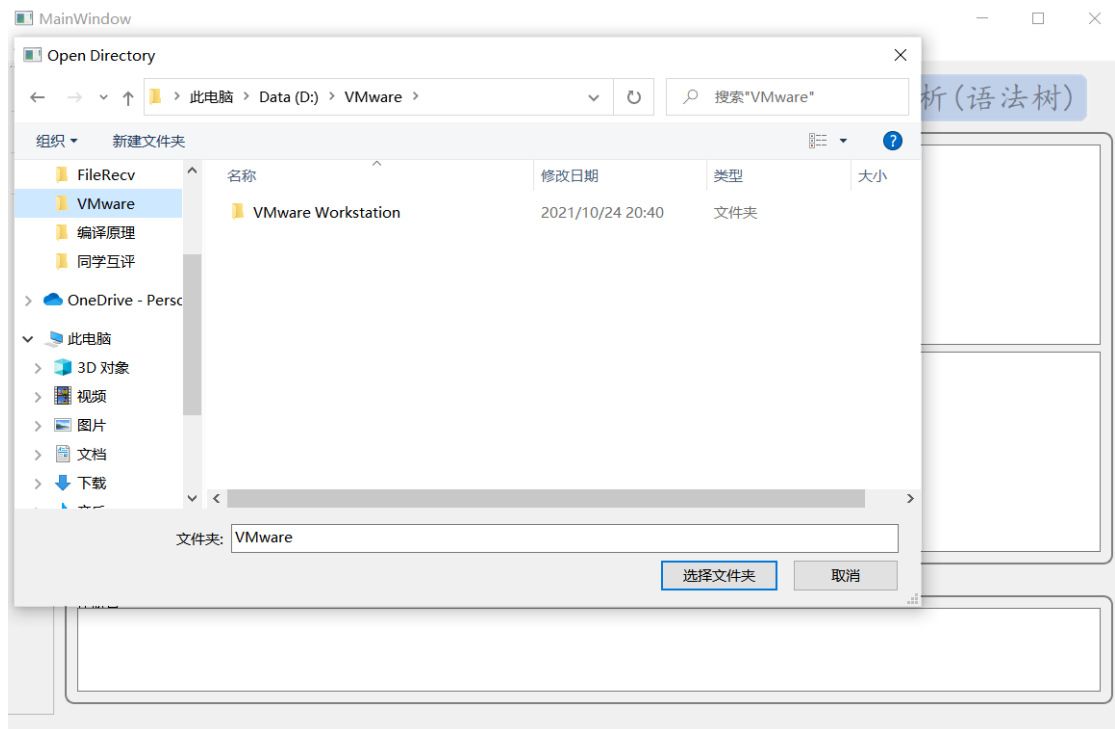
④导入代码文本



⑤导入文法规则文本



⑥指定输出语法分析树的文件目录

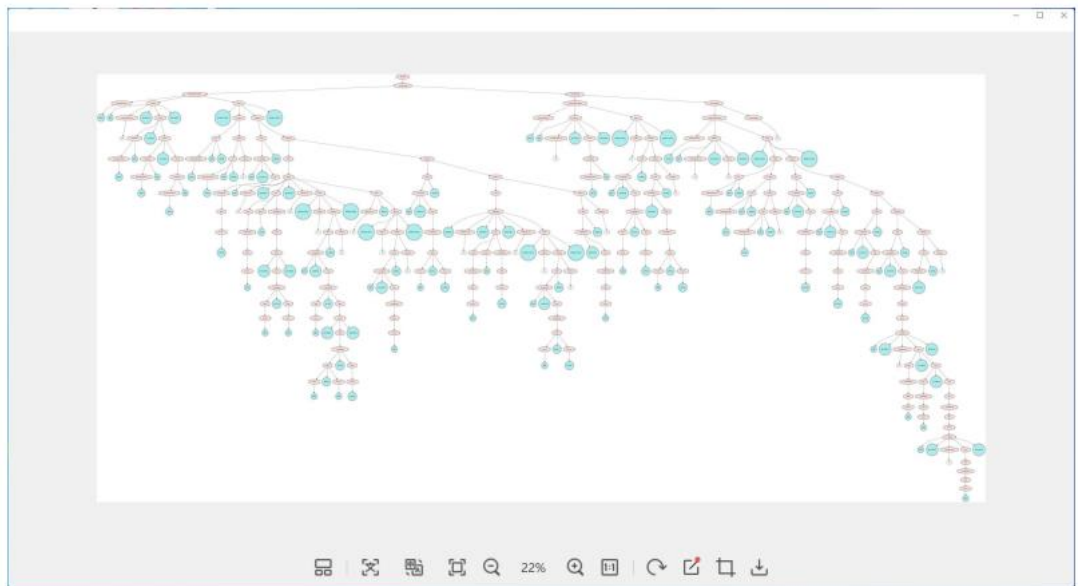


⑦点击词法分析按钮，进行词法分析













⑧单击语法/语义分析按钮，执行程序，输出各步骤使用的规约文法与对应文本所属符号类型的详细信息。如若规约成功，控制台将会显示“语法分析结束”

提示，生成的中间代码显示在输出框内，并弹出语法分析树绘制图像，中间代码文档和图像均保存在给定路径。



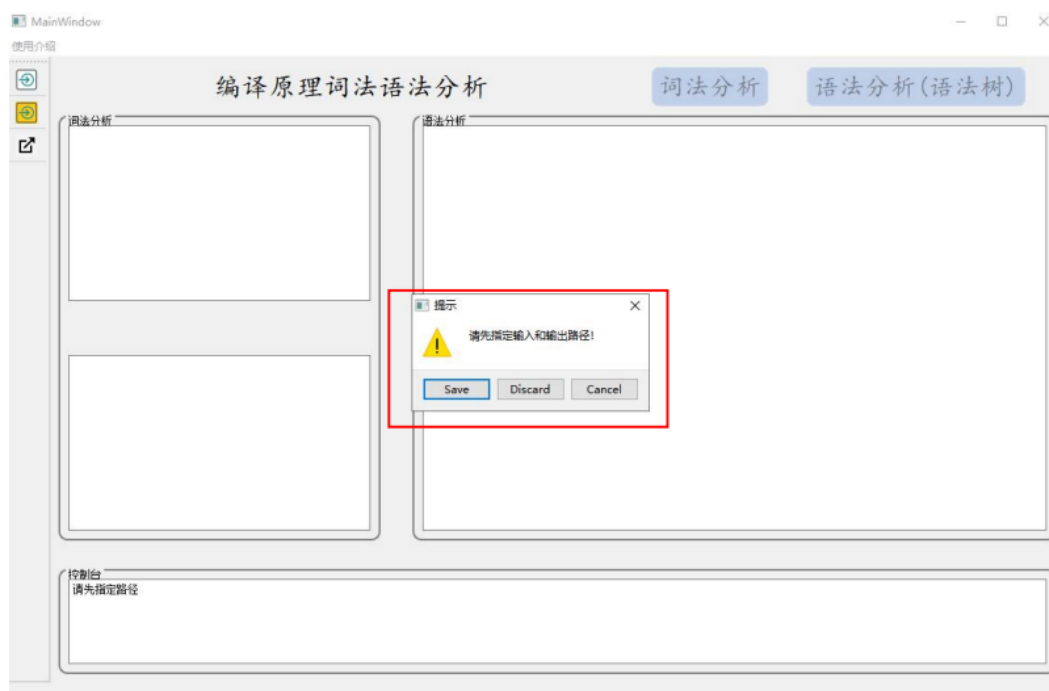
⑨查看给定路径的各个输出文件

 ActionGotoTable.txt	2021/12/14 23:09	文本文档	36 KB
 ErrInput.txt	2021/12/14 15:53	文本文档	1 KB
 grammar.txt	2021/11/8 12:06	文本文档	2 KB
 Input.txt	2021/12/14 15:43	文本文档	1 KB
 lexoutput.txt	2021/12/14 23:38	文本文档	3 KB
 OptInput.txt	2021/12/14 23:09	文本文档	1 KB
 process.txt	2021/12/14 23:09	文本文档	3 KB
 semanresult.txt	2021/12/14 23:09	文本文档	1 KB
 tree.dot	2021/12/14 23:09	Microsoft Word ...	10 KB
 tree.jpg	2021/11/9 9:53	JPG 文件	2,209 KB

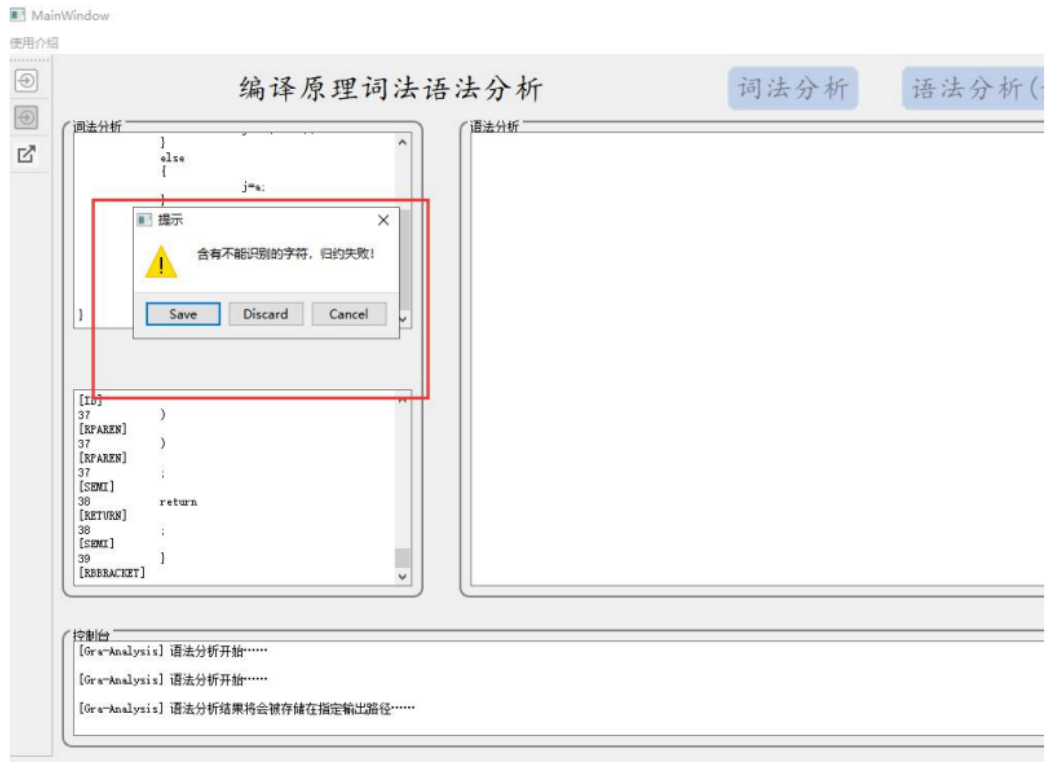
4.1.2 错误与警告示例

本程序能对用户非法操作进行提示与警告，具有一定鲁棒性。在词法、语法、语义分析的过程中，也分别可以对非法字符输入、指定文法错误导致的归约失败以及多种语义错误进行反馈。

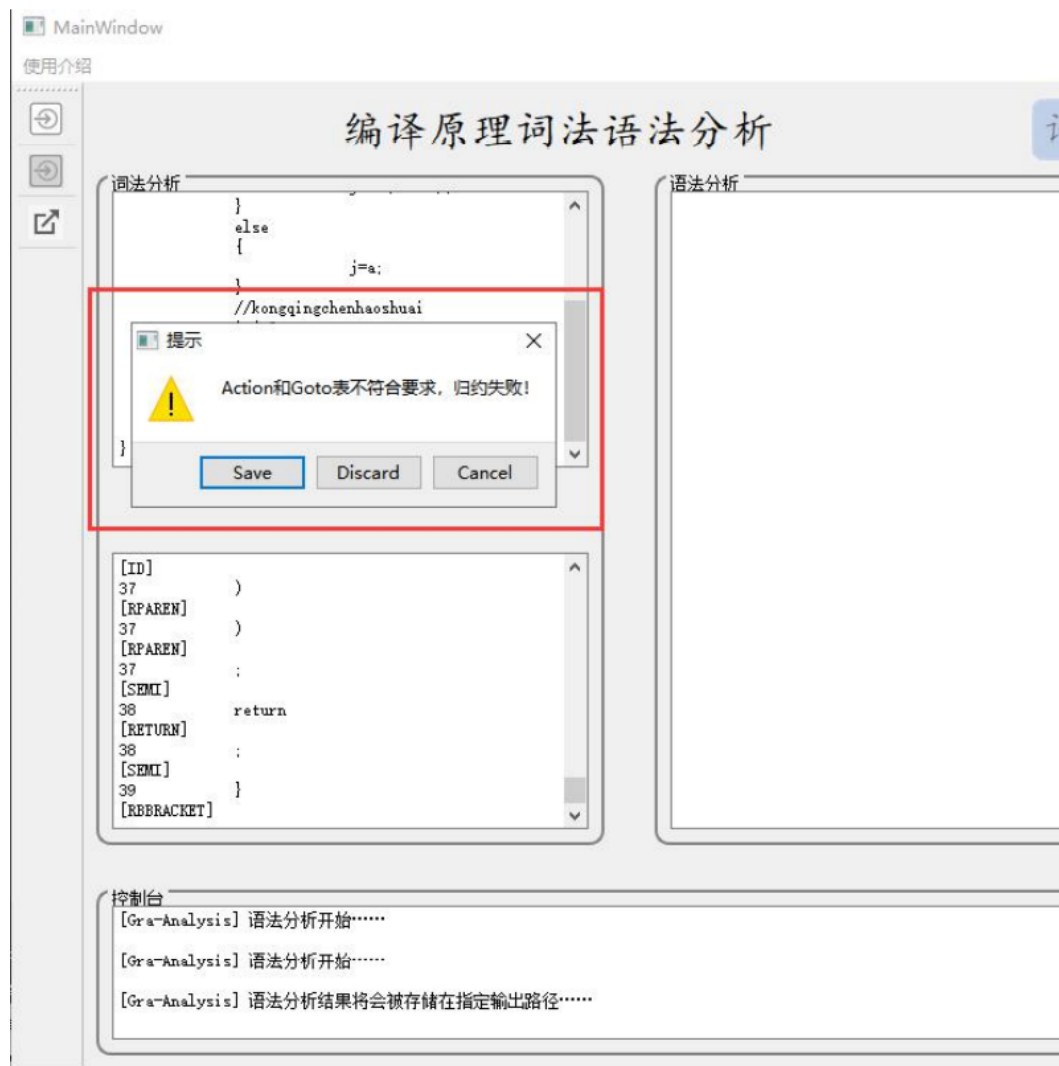
①用户非法操作



②非法字符输入



③指定文法错误



④语义错误

测试代码:

```

int program(int a,int b, int c)
{
    int i;
    int j;
    int i;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    //kongqingchenhaoshuai
    i=j*2;
    while(i<=100)
    {
        i=i*2;
    }
    return i;
}
int demo(int a)
{
    a=a+2;
    return a*2;
}

void re(int a)
{
    a = 0;
}

```

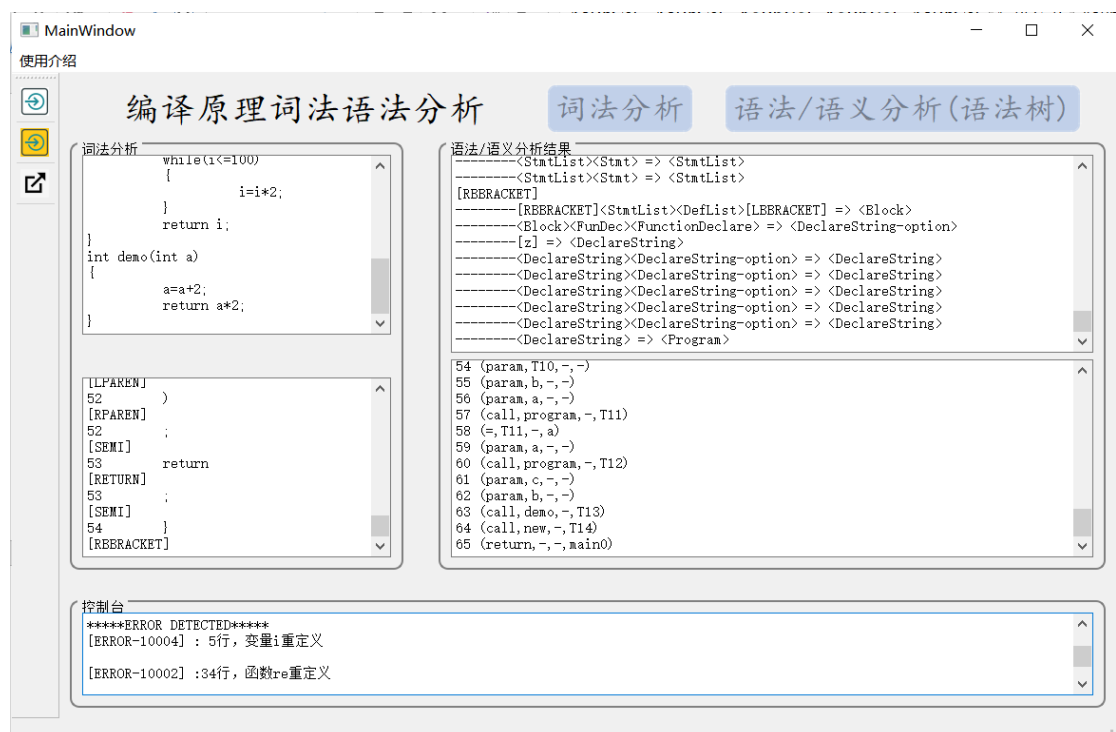
```

int re(int a, int b)
{
    return b;
}

/*wo
weishenme
zhemecai*/
void main0()
{
    int a;
    int b;
    int c;
    p=6;
    a=3;
    b=4;
    c=2;
    a=program(a,b,demo(c));
    program(a);
    demo(b,c);
    new();
    return ;
}

```

前端输出：



可见控制台输出了报错信息，完整报错信息如下：

```
*****ERROR DETECTED*****  
[ERROR-10004] : 5行, 变量i重定义  
[ERROR-10002] :34行, 函数re重定义  
[ERROR-10005] : 45行, 变量p未定义  
[ERROR-10007] : 50行, 函数program调用时参数过少  
[ERROR-10008] : 51行, 函数demo调用时参数过多  
[ERROR-10009] : 52行, 函数new调用未定义  
[ERROR-10010] : 52行, 函数new调用未定义  
[ERROR-10007] : 52行, 函数new调用时参数过少  
[ERROR-10000] : 未定义main函数
```

4.1.3 优化示例

简单代码:

```
void main()  
{  
    int a;  
    int b;  
    int c;  
    int d;  
    a = (3+2)-4;  
    b = 8;  
    c = a+b;  
    d = a+b;  
    //(a+b)+c;  
    return ;  
}
```

优化结果:

```

*****Original Code*****
0 (j,-,-,1)
1 (main,-,-,-)
2 (=,1,-,a)
3 (=,8,-,b)
4 (+,a,b,T0)
5 (=,T0,-,c)
6 (+,a,b,T1)
7 (=,T1,-,d)
8 (return,-,-,main)

*****Optimized Code*****
0 (j,-,-,a)
1 (main,-,-,-)
2 (proc,-,-,-)
3 (=,8,-,b)
4 (+,a,b,c)
5 (=,T0,-,d)
6 (return,-,-,main)

```

4.2 调试过程存在的问题及解决方法

实际上，教材中针对各种语句的语义分析动作已经讲述十分详尽，语义分析代码编写时比较顺利，无重大 bug。我们对四元式组进行了输出重载，从而使得中间代码能够输出到屏幕，方便调试，并对每一次中间代码生成的过程打印新增的四元式组，与源代码对照，即可检查语义分析的正确性。

```

26     friend ostream& operator<<(ostream& out, const Quadruple& q);
27 };
28
29 ostream& operator<<(ostream& out, const Quadruple& q) {
30     cout << q.index << " ("
31         << q.op << ',' << q.arg1 << ',' << q.arg2 << ',' << q.res
32         << ")" << endl;
33     return out;
34 }

```

```

SemanProd_CreateFunTable_m:
1 (program, -, -, -)

SemanProd_ParamDec:
1 (program, -, -, -)
2 (defpar, -, -, a)

SemanProd_ParamDec:
1 (program, -, -, -)
2 (defpar, -, -, a)
3 (defpar, -, -, b)

SemanProd_ParamDec:
1 (program, -, -, -)
2 (defpar, -, -, a)
3 (defpar, -, -, b)
4 (defpar, -, -, c)

```

```

int program(int a,int b, int c)
{
    int i;
    int j;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    i=j*2;
}

```

```

SemanProd_AddSubExp:
1 (program, -, -, -)
2 (defpar, -, -, a)
3 (defpar, -, -, b)
4 (defpar, -, -, c)
5 (=, 0, -, i)
6 (+, b, c, T0)
7 (j=, a, T0, 10)
8 (=, 0, -, T1)
9 (j=, -, T1)
10 (=, 1, -, T1)
11 (j=, T1, 0, )
12 (j=, -, )
13 (*, b, c, T2)
14 (+, T2, 1, T3)

SemanProd_AddSubExp:
1 (program, -, -, -)
2 (defpar, -, -, a)
3 (defpar, -, -, b)
4 (defpar, -, -, c)
5 (=, 0, -, 1)
6 (+, b, c, T0)
7 (j=, a, T0, 10)
8 (=, 0, -, T1)
9 (j=, -, T1)
10 (=, 1, -, T1)
11 (j=, T1, 0, )
12 (j=, -, )
13 (*, b, c, T2)
14 (+, T2, 1, T3)
15 (+, a, T3, T4)

```

```

int program(int a,int b, int c)
{
    int i;
    int j;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    i=j*2;
    while(i<=100)
    {
        i=i*2;
    }
    return i;
}

int demo(int a)
{
    a=a+2;
    return a*2;
}

```

5 总结与收获

在本次实验中，我们小组各成员分工合作，从底层开始实现了一个功能完善的类 C 语言语义分析器，回顾过程，实是收获良多。

在着手实践之前，我们充分思考了项目的整体架构、数据结构的选择、函数功能及其调用关系、结果展示和交互功能等，这锻炼了我们的工程思维与设计能力；而在编码过程之中，我们也 将尽可能多的情况纳入考量，在实践中不断发现、追踪、解决问题，培养了耐心、细致、严谨的工作习惯。

在实际编程中，我们熟练掌握了相关工具的使用，比如 STL 容器中 map 与 set 数据结构，对 iterator 迭代器有了深入的了解；此外，还使用到了 QT 与 Graphviz 针对工程进行了可视化的优化。

对于语义动作的分析，我们从书本出发，分门别类地给出了各个产生式对应的语义规则，并正确地实现了语义分析过程。这启发我们：动手实践不能脱离理论基础，同时要学会借鉴权威的思路与设计。

整个项目从一而终，我们小组内部分工明确，氛围融洽，保持了及时、高效的沟通与协作。制定计划以后，每一位组员都积极承担自己的工作任务；在项目进程的节点时期，我们也总是通过讨论交流进展，明确下一步的方向。可以说，这次大作业也是我们锻炼团队合作、分工协调能力的一次机会，让我们能在合作中相互学习、共同进步。

6 小组分工

姓名	分工
赵昊堃	语义分析与报告书写
孔庆晨	语义分析与前端设计
史睿琪	代码优化与报告书写

7 参考文献

- [1] 陈火旺，刘春林，谭庆平，et al. 程序设计语言编译原理：第3版[M]. 国防工业出版社，2008.
- [2] 陆文周. Qt 5 开发及实例[M]. 电子工业出版社，2015.
- [3] 李艾. SLR 语法分析器及其语法错误修正[J]. 计算机研究与发展，1982, 019(011):20-28+41.
- [4] 付争方，张海娟. LR 语法分析器构造方法初探[J]. 中国科技信息，2005,

000(15A):113-113.

[5] 李磊. C 编译器中间代码生成及其后端的设计与实现[D]. 2016.