



江蘇大學

编译技术课程设计报告

程序的小型编译器

学 院 计算机学院

专业班级 计算机 1302

学生学号 3130602050

学生姓名 曾 彪

指导教师 年 轶

2016 年 6 月

目 录

一、课程目标.....	2
二、课程设计的题目及要求.....	2
三、问题分析及相关原理介绍.....	2
3.1 问题分析.....	2
3.2 相关原理介绍.....	3
四、设计思路及关键问题的解决方法.....	4
4.1 设计思路.....	4
4.2 关键问题的解决方法.....	4
五、结果及测试分析.....	20
5.1 软件运行环境.....	20
5.2 测试方法和测试数据.....	20
5.3 系统的运行结果及功能说明.....	25
六、总结及心得体会.....	36
七、参考文献.....	38

一、课程目标

结合所学知识，并查阅参考相关资料，通过设计、编制、调试一个具体的程序的小型编译器，加深对编译器内部工作原理的理解，掌握词法分析、语法分析、语义分析、中间代码生成等相关原理和程序实现方法，从而透彻地领悟编译原理的精髓。

二、课程设计的题目及要求

1. 题目

程序的小型编译器

2. 要求

(1) 词法分析 产生语言的单词序列

(2) 语法分析

1) 识别由加+ 乘* 括号（）操作数所组成的算术表达式

2) 识别布尔表达式

3) 识别条件语句

4) 识别循环语句

(3) 中间代码生成 产生包含上述语句的程序的中间代码

(4) 错误处理 错误定位及出错信息

三、问题分析及相关原理介绍

3.1 问题分析

本次实验的任务是编制一个程序的小型编译器，输入一段程序代码，程序由条件语句、循环语句、布尔表达式和算术表达式构成，编译器能对程序的词法、语法、语义等进行分析，并生成相关的中间代码，如果程序代码有错误也能对错误进行定位并输出是何种错误信息。

程序小型编译系统的功能是输入或读入一段用设计的语法规则编写的源程序代码，完成词法分析，输出语言的单词序列、输出符号表、常数表。完成语法分析，识别出程序中的条件语句、循环语句、布尔表达式和算术表达式。完成语义分析和中间代码生成，输出程序的四元式。能识别出程序中出现的错误，输出

错误的位置和错误的类别。

3.2 相关原理介绍

词法分析，程序从左到右逐个地对待识别的源程序代码进行扫描，产生一个个单词符号，把字符串形式的源程序改造成单词符号串形式的中间程序。即从文本中读取出一段源代码，然后用词法分析器识别出源代码中的每个标识符、常数、保留字、运算符和界符，并将它们登记到相应的表项中。

DFA（确定有限状态自动机）是根据正则表达式来识别字符串的，词法分析的作用就是描述字符串的规则。语法分析与词法分析相比则较大的不同，这是因为语法分析中的文法含有非终结符，且非终结符在文法中的主要作用是用来表示嵌套和递归，以便形成比字符串更为复杂的句子。如果用栈来记录 DFA 输入句子（仅由终结符组成）和生成非终结符的过程，则 DFA 的输入过程与移进过程对应，而生成非终结符的过程则与规约过程对应。符号栈里实时记录着 DFA 的状态转换路径。符号栈里的每一个符号都对应着状态转换图中的一条有向边；这些有向边首尾相连，起始于 DFA 的初态 S_0 ，暂停于 DFA 的当前扫描状态 S_i ，用一个状态序列来替代符号栈中的符号串；因为状态序列与符号栈中的符号串作用相同，都记录着 DFA 在识别句子过程中的状态转换路径。这样，符号栈就变成了状态栈，而 DFA 的状态转换可以用栈顶的状态与当前扫描的输入符号来决定。由此，对符号的移进就变成了对状态的移进，对符号串的规约就变成了对状态序列的替换。

LR 分析法的基本思想是，在规范规约的过程中，一方面记住已移进和规约出的整个符号串，另一方面根据所用的产生式推测未来可能遇到的输入符号。当一串貌似句柄的符号串呈现于分析栈的顶端时，LR 分析器能根据所记载的“历史”和“展望”以及“现实”的输入符号等三方面的材料，来确定栈顶的符号是否构成相对某一产生式的句柄。

假定有一个自底向上的 LR 分析器，使它能在用某一产生式进行规约的同时调用相应的语义子程序进行有关的翻译工作。每个产生式的语义子程序执行之后，某些结果（语义信息）必须作为此产生式的左部符号的语义值暂时保存下来，以便以后语义子程序引用这些信息。此外，原 LR 分析器的分析栈也加以扩充，以便能够存放与文法符号对应的语义值。这样，分析栈可以存放三类信息：分析状态，文法符号及文法符号对应的语义值。在用某一个规则进行规约之后，调用

相应的语义子程序完成与所用产生式相应的语义动作，并将每次工作后的语义值保存在扩充后的语义栈中。

四、设计思路及关键问题的解决方法

4.1 设计思路

首先设计需要的各类文法，其中包括程序语句的文法、算术表达式和布尔表达式的文法，然后根据文法画 DFA 状态转换图，再根据 DFA 写 SLR 分析表，然后就是编写程序代码。根据 SLR 分析过程设计程序的大体框架，再根据文法设计分析语句的程序的移进、规约出错等结构。

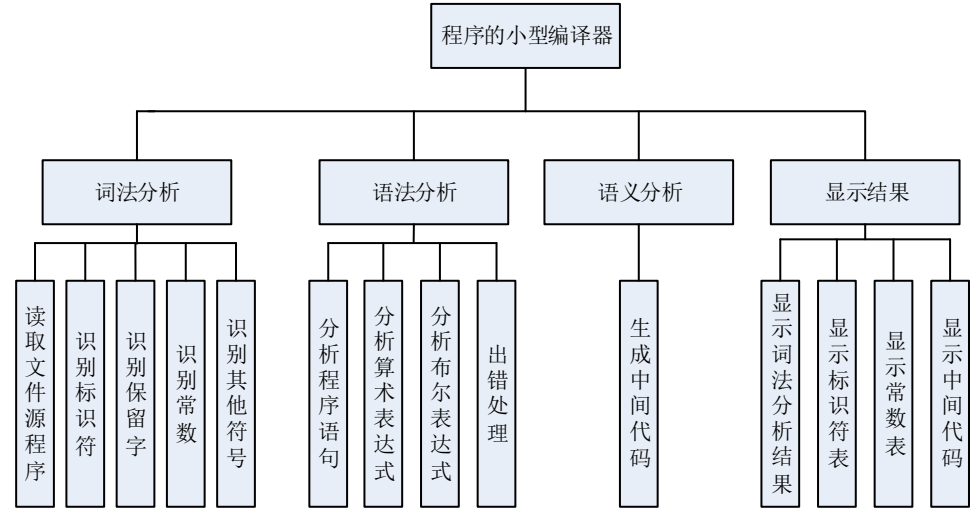


图 1 系统结构图

4.2 关键问题的解决方法

第一步词法分析，程序中的单词符号如表 1 所示：

表 1 单词符号表：

单词符号	种别编码	助记符	内码值
if	0	s_if	——
else	1	s_else	——
switch	2	s_switch	——
case	3	s_case	——
break	4	s_break	——
continue	5	s_continue	——
int	6	s_int	——

for	7	s_for	——
while	8	s_while	——
do	9	s_do	——
return	10	s_return	——
void	11	s_void	——
char	12	s_char	——
include	13	s_include	——
标识符	20	iden	iden 在符号表中的位置
常数	21	cons	cons 的值
下划线	22	under line	——
!	23	not	——
&&	24	and	——
	25	or	——
错误标志	30	err	err 在非法符号表中的位置
>	50	bigger	——
<	51	smaller	——
+	52	jia	——
-	53	jian	——
*	54	chen	——
/	55	chu	——
=	56	deng	——
>=	57	dadeng	——
<=	58	xiaodeng	——
!=	59	budeng	——
==	60	equal	——
,	70	douhao	——
;	71	fenhao	——
{	72	da l	——
}	73	dar	——

(74	xiaol	——
)	75	xiaor	——
#	76	jinhao	——

第二步，设计各部分的 LR 分析表

(1) 算术表达式的 LR 分析表

算术表达式文法拓广为文法 $G'[E]$ ：

$S' \rightarrow E$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow i$

算术表达式的 DFA 如图 2 所示：

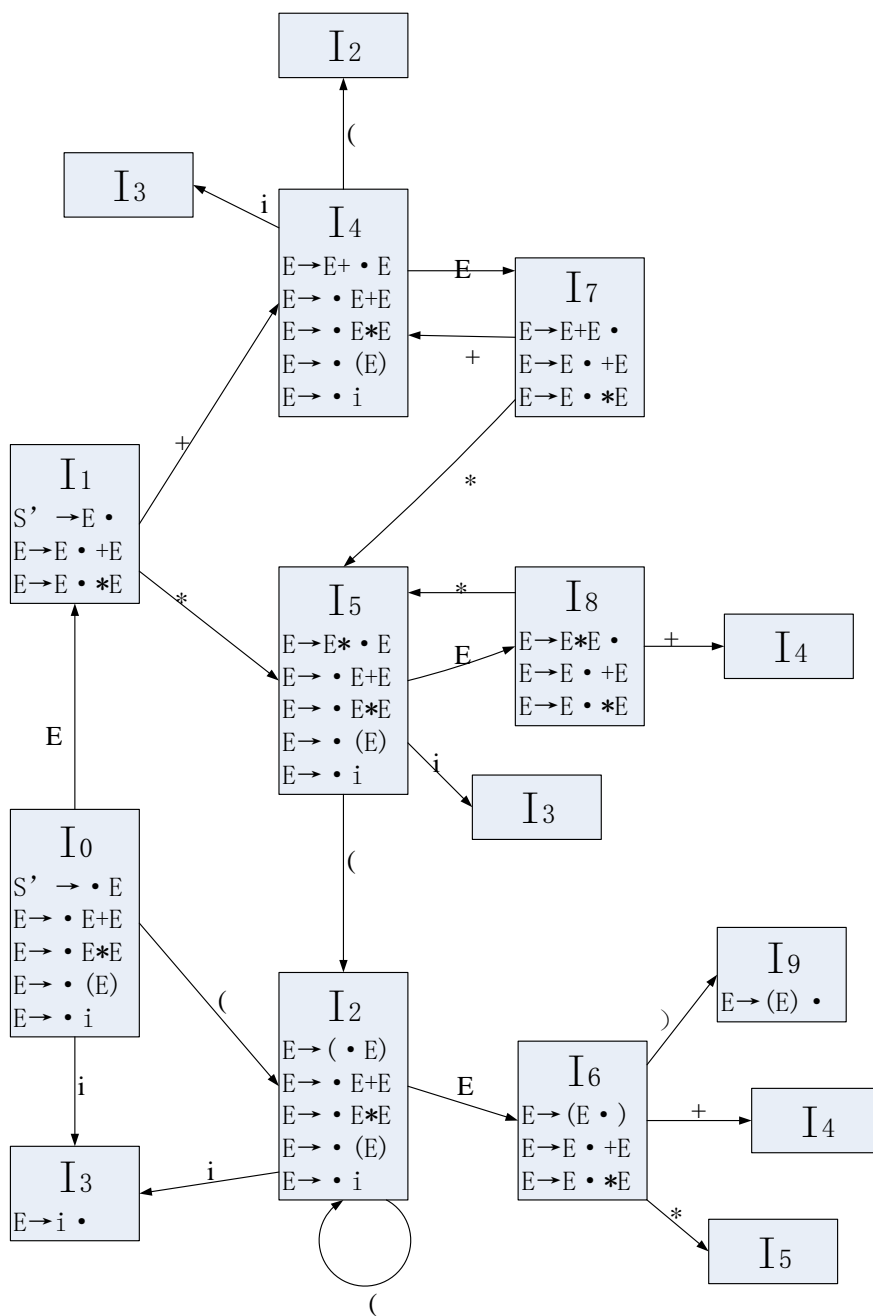


图 2 算术表达式的状态转换图

由此得到算术表达式的 SLR (1) 分析表如表 2:

表 2 算术表达式的 SLR (1) 表

状态	ACTION						GOTO
	i	+	*	()	#	E
0	S3	-2	-3	S2	-4	-5	1
1	-20	S4	S5	-20	-20	acc	-1
2	S3	-7	-8	S2	-9	-10	6
3	-20	r4	r4	-20	r4	r4	-1
4	S3	-11	-12	S2	-13	-14	7
5	S3	-15	-16	S2	-17	-18	8
6	-21	S4	S5	-21	S9	-19	-1
7	-20	r1	S5	-20	r1	r1	-1
8	-20	r2	r2	-20	r2	r2	-1
9	-21	r3	r3	-21	r3	r3	-1

出错信息编号及描述:

// -1: 表达式错误

// -2: 表达式不能以+开头

// -3: 表达式不能以*开头

// -4: 表达式不能以) 开头

// -5: 表达式只有一个分号, 没有其他内容

// -6: 缺少与) 匹配的 (

// -7: 左括号右边不能为+

// -8: 左括号右边不能为*

// -9: 小括号 () 之间没有任何内容

// -10: (出现在最右边
 // -11: 出现两个连续的+
 // -12: +后面出现*
 // -13: +后面出现)
 // -14: 表达式以+结尾
 // -15: *后面出现+
 // -16: 出现两个连续的*号
 // -17: *后面出现)
 // -18: 表达式以*结尾
 // -19: 左右括号数量不匹配!
 // -20: 缺少分号或结束符!
 // -21: 缺少运算符号!

(2) 布尔表达式的 LR 分析表

布尔表达式文法拓广为文法 $G'[B]$:

$S' \rightarrow B$

$B \rightarrow i$

$B \rightarrow i \text{ rop } i$

$B \rightarrow (B)$

$B \rightarrow \text{not } B$

$B \rightarrow AB$

$B \rightarrow 0B$

$A \rightarrow B \text{ and}$

$0 \rightarrow B \text{ or}$

布尔表达式的 DFA 如图 3:

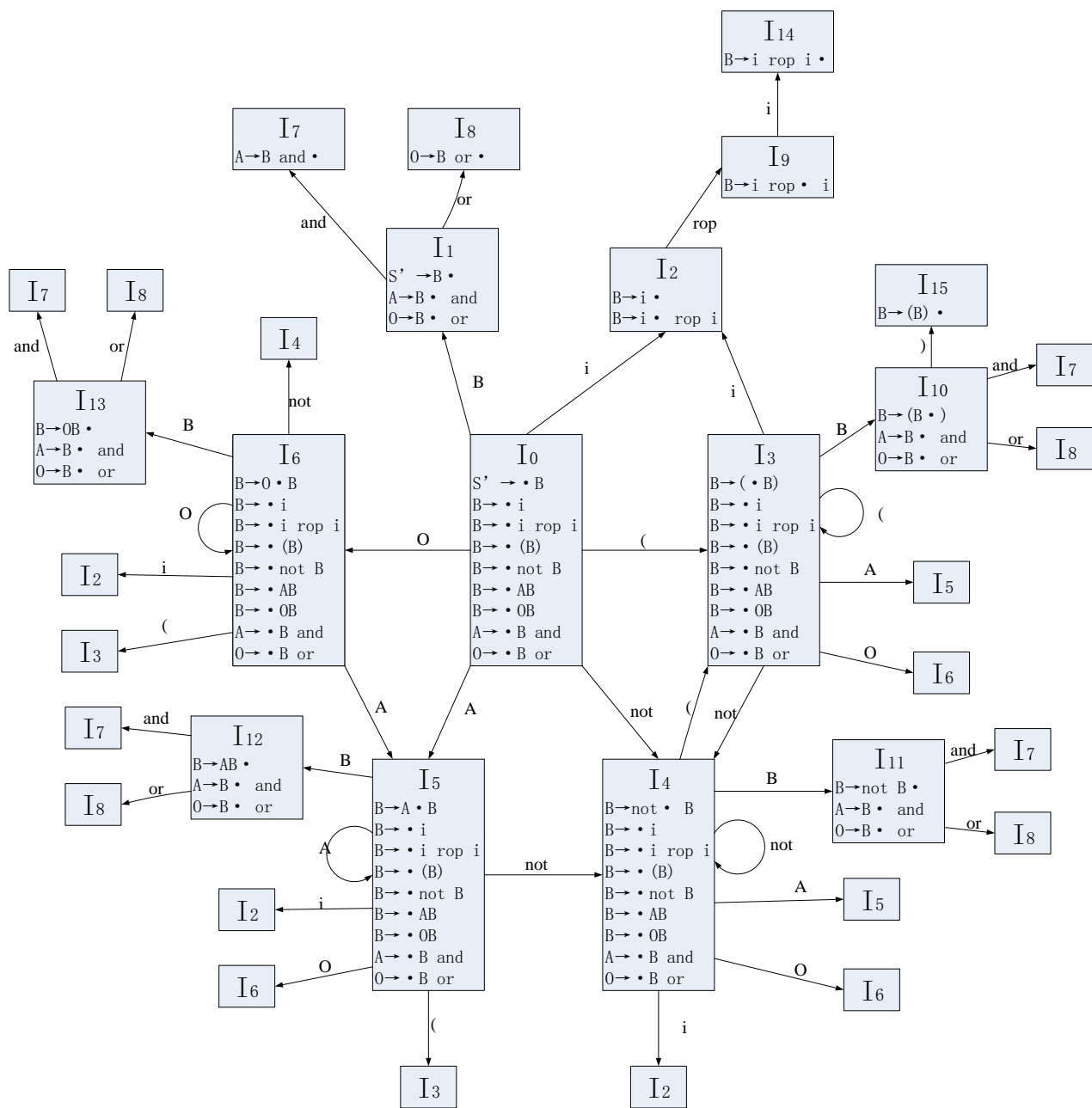


图 3 布尔表达式状态转换图

由此得到布尔表达式的 SLR (1) 分析表如表 3:

表 3 布尔表达式的 SLR (1) 分析表

状态	ACTION								GOTO		
	i	rop	()	not	and	or	#	B	A	O
0	S2	-22	S3	-4	S4	-23	-24		1	5	6
1						S7	S8	acc			
2		S9		r1		r1	r1	r1			
3	S2	-25	S3		S4				10	5	6
4	S2		S3		S4				11	5	6
5	S2		S3		S4				12	5	6
6	S2		S3		S4				13	5	6
7	r5		r5		r5						
8	r7		r7		r7						
9	14										
10				S15		S7	S8				
11				r4		S7	S8	r4			
12				r6		S7	S8	r6			
13				r8		S7	S8	r8			
14				r2		r2	r2	r2			
15				r3		r3	r3	r3			

出错信息编号及描述:

// -22: 比较符号 rop 出现在最左边

// -23: and 出现在最左边

// -24: or 出现在最左边

// -25: 比较符号 rop 没有左操作数

(3) 程序语句的 LR 分析表

程序语句的文法拓广为文法 $G'[B]$ 如下:

$$S' \rightarrow S$$
$$S \rightarrow \text{if } e \text{ } S \text{ else } S$$
$$S \rightarrow \text{while } e \text{ } S$$
$$S \rightarrow \{L\}$$
$$S \rightarrow a;$$
$$L \rightarrow SL$$
$$L \rightarrow S$$

程序语句的 DFA 如图 4:

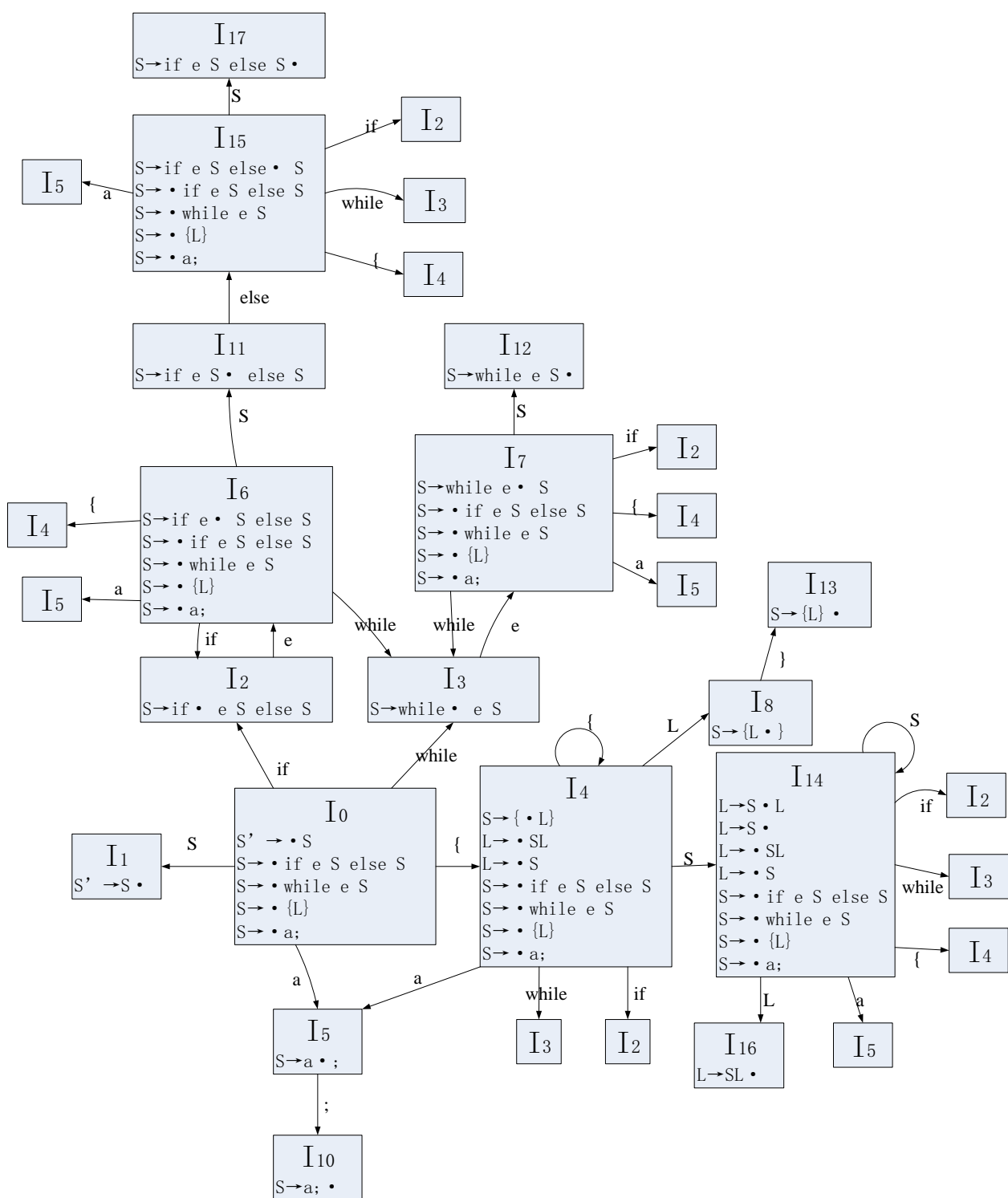


图 4 程序语句的状态转换图

由此得到程序语句的 SLR (1) 分析表如表 4:

表 4 程序语句的 SLR (1) 分析表

状态	ACTION									GOTO	
	if	else	while	a	;	e	{	}	#	S	L
0	S2	-30	S3	S5			S4	-31		1	
1									acc		
2						S6					
3						S7					
4	S2	-30	S3	S5			S4			14	8
5					S10						
6	S2		S3	S5			S4			11	
7	S2		S3	S5			S4			12	
8								S13			
9	S2		S3	S5			S4			14	
10	r4	r4	r4	r4			r4	r4	r4		
11	-32	S15	-32	-32	-32	-32	-32	-32	-32		
12	r2	r2	r2	r2			r2	r2	r2		
13	r3	r3	r3	r3			r3	r3	r3		
14	S2		S3	S5			S4	r6		14	16
15	S2	-30	S3	S5			S4			17	
16								r5			
17	r1	r1	r1	r1			r1	r1	r1		

出错信息编号及描述:

// -30: 没有与 else 匹配的 if 语句

// -31: 程序以 } 开头

// -32: 没有与 if 匹配的 else 语句

栈结构如下:

```

struct stackType{//分析栈
    int a;//状态栈
    aa b;//符号栈
    int c;//语义栈
} stack[100];

```

```

struct aa{
    int syl;//种别编码
    int pos;//存储在表中的位置
    int zhongjie;//是否是终结符
    int linecount;//在源代码中的行号
};

```

类函数功能描述如下：

```

class analyze
{
    string line;        //当前扫描行
    int i;              //当前扫描字符位置
    char ch;           //当前扫描字符
    int ccount;        //当前扫描到的词数
    int po_ch;         //当前登记了多少个标识符
    int po_nu = 0;     //当前登记了多少个数字
    int nowline;       //当前扫描的是代码中的第几行
    int count_err;     //当前扫描出的错误数
    char errs[60];      //储存错误信息
    int errsLine[60];  //每个错误在源代码中的行号
    string tab_ch[1000]; //标识符登记表
    int tab_nu[100];    //常数表
    ifstream in;        //待分析的文本
    ofstream out2;      //生成待分析的文本
    ofstream out;       //语义分析结果保存的文本
    int ista;           //语法分析的表达式的起始位置
    int iend;           //语法分析的表达式的结束位置
    int sp;            //栈顶指针 stack 下标

```



```

int ssp;//栈顶指针 sstack 下标
int ii;//当前要分析的符号在词法分析结果(buf 表)中的位置
int lr;//分析动作, 移进还是规约或是出错
int nxq;//下一个四元式的地址
ntab3 ntab3[200];//存储每个布尔表达式翻译成的两个四元式的
地址

int label;//ntab3 表下标
aa downline;//下划线
aa Et;//产生式左部

int sign; //sign=1 表明是算术表达式语句, sign=2 表明是循环语
句, sign=3 表明是控制语句

int newt;//新产生的临时变量的整数码
aa buf[3000];//词法分析表, 保存词法分析的结果
aa buf2[300];//保存算术或布尔表达式的输入字符串
stackType stack[100];//程序语句语法分析栈
stackType sstack[100];//算术或布尔表达式语法分析栈
int jj;//当前要分析的符号在 buf2 表中的位置
fouexp fexp[200];//四元式存储集
ll labelmark[10];//保存 if 或 while 一开头的那个状态
int labeltemp[10];//保存 if 和 else 中间那个跳过的状态
int pointmark;//labelmark 数组的当前下标
int pointtemp;//labeltemp 数组的当前下标
int ret;//算术表达式、布尔表达式分析子程序退出标记
int ret2;//程序语句退出标记
int subline;//源代码的错误行号

public:

analyze();//初始化各个变量
void stCifa();//开始进行词法分析
void stYufa();//开始进行语法分析

```

```

int findd(string m); //查找标识符 m 在标识符登记表中的位置
void identifier(); //识别字符串是标识符还是保留字
void number(); //识别数字
void backch(); //回退一个字符
void readch(); //读取一个字符
void scanOneLine(); //词法分析一行
void show(); //输出词法分析的结果
void showiden(); //输出标识符表
void shownum(); //输出常数表
void showZhuangtai(); //输出状态栈
void showFuhao(); //输出符号栈
void showYuyi(); //输出语义栈
void showYuyi(int yu); //输出语义栈
void showZhan(int lrr); //显示各个栈及规约、移进动作
string changeStackB(int sy, int zho); //根据种别编码回返相应的输入符号, 第十个参数标识是否是非终结符

string finsBuf(int q); //根据 buf 的下标返回对应的符号
void reset(); //重置语法分析栈
void reset2(); //重置语法分析栈
void show2(int st, int en); //输出词法分析结果中从位置 ista 到 iend 的元素

string finds(int tc); //根据种别编码返回符号
void stsend(); //语义分析时计算当前表达式的开始位置和结束位置

void show3(); //输出计算到当前这一步的计算结果
int conv(aa tem); //根据输入符号返回在程序语句 LR 分析表中的横向序号

int conv2(aa tem); //根据输入符号返回在算术表达式 LR 分析表中的横向序号

```

```

        int conv3(aa tem); //根据输入符号返回在布尔表达式 LR 分析表中的
        横向序号
        int test(int value); //判断当前输入字符是程序语句的话返回 0,
        否则返回 1 (布尔表达式或算术表达式)
        int newtemp(); //产生一个新的临时变量
        int lrparse(); //程序语句语法分析
        int lrparse2(); //算术表达式的分析
        int lrparse3(); //布尔表达式的分析
        int emit(string opl, aa arg1, aa arg2, int result1); //产生
        一个四元式
        int merge(int p1, int p2); //拉链函数
        void show4exp(); //输出四元式
        void backpatch(int p, int t); //返填函数
        ~analyze();
};

```

函数调用关系图如图 5 所示：

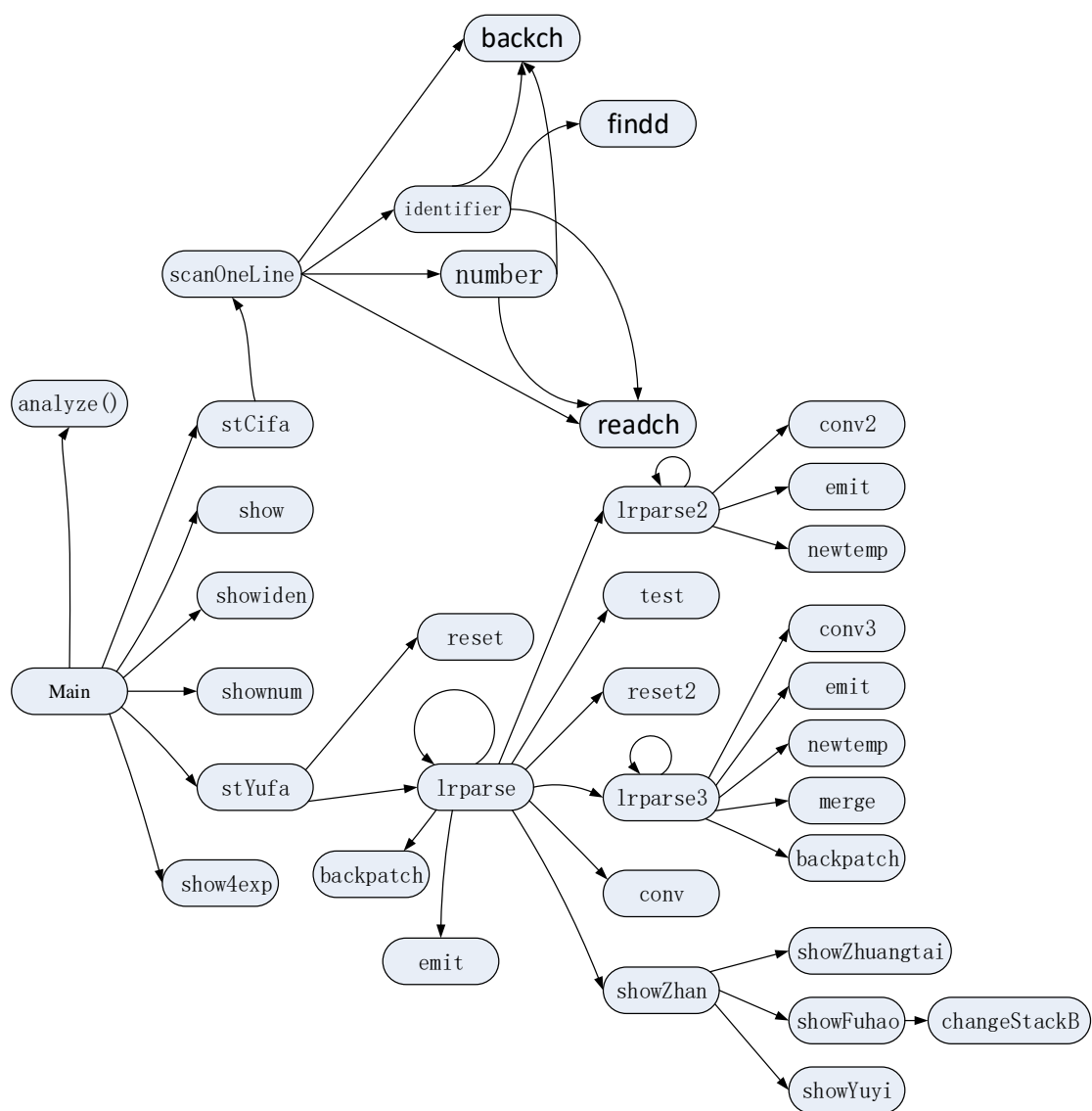


图 5 函数调用关系图

程序流程图如图 6 所示：

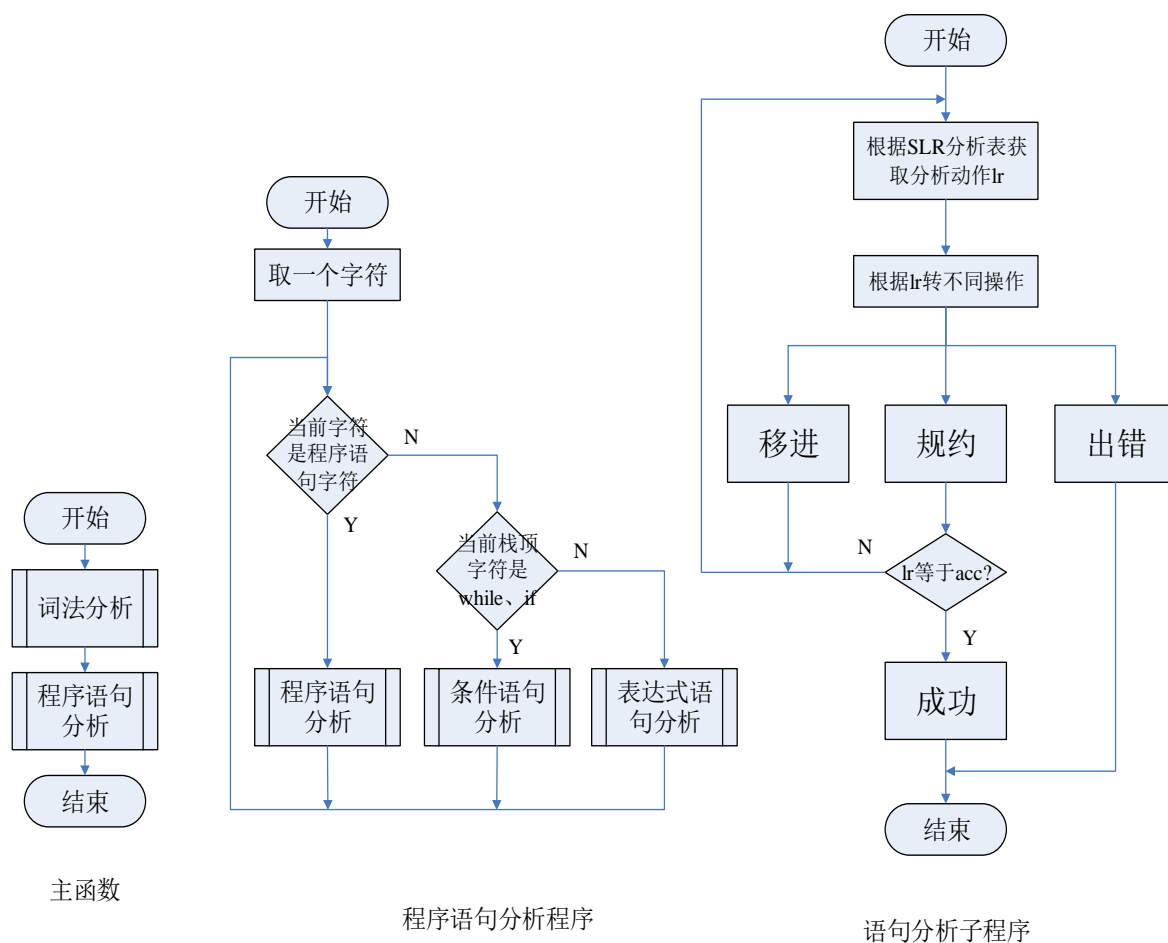


图 6 程序流程图

五、结果及测试分析

5.1 软件运行环境

开发环境： Visual Studio 2013

语言： C++

5.2 测试方法和测试数据

1. 测试方法说明

程序中如果有宏定义“#define iWantDefault”则测试的源程序始终为程序中所给的一段默认源代码，如果没有该宏定义，则需要事先在本地文件中编辑一段源程序，程序从这个文件中读取出待测试的源程序，实现方法如图 7：

```

#ifdef iWantDefault
    out2.open("E:\\hizengbiao2\\c.txt");
    if (!out2)
    {
        cerr << "文件打开失败! ";
    }
    {
        out2 << "while (a || b){ if (x < y){ while (c&&d){ k = k + 1; } } else{ if (m <
    }
    out2.close();
#endif

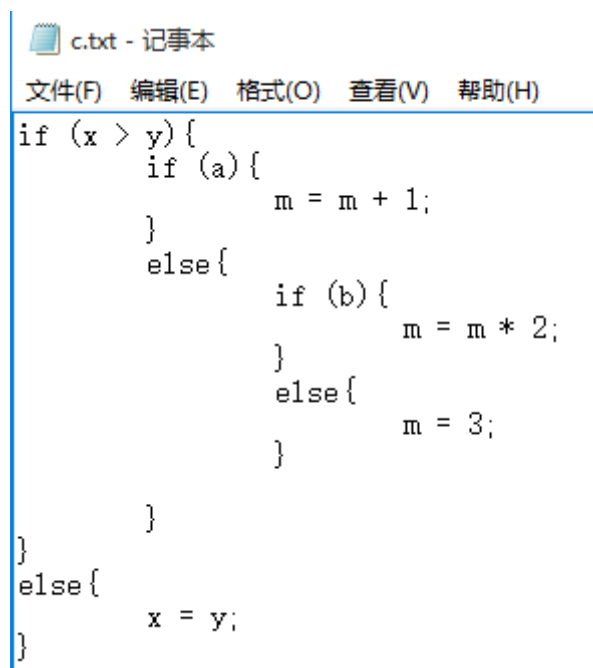
```

图 7 读取文件来源选择

2. 测试数据说明

测试用例 1，测试 if, else 控制语句：

测试程序语句中的 if 和 else 语句各个分支的跳转是否正确，如果 if 语句的条件满足则执行 if 后面的语句块，然后还应跳过对应的 else 后面的语句块；如果条件不满足则跳过 if 后面的语句块执行 else 后面的语句块。



```

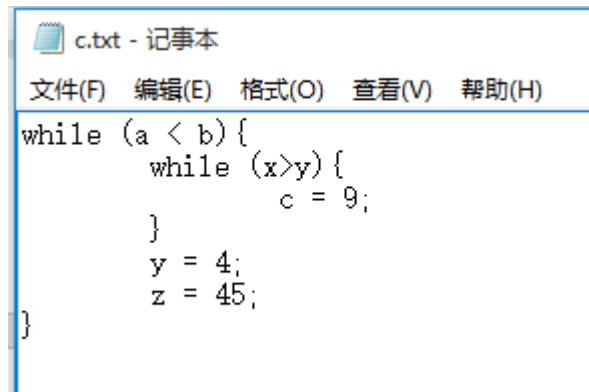
c.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
if (x > y) {
    if (a) {
        m = m + 1;
    }
    else {
        if (b) {
            m = m * 2;
        }
        else {
            m = 3;
        }
    }
}
else {
    x = y;
}

```

图 8 测试用例 1

测试用例 2，测试 while 循环语句：

测试 while 语句的各个跳转是否正确，如果 while 后面的条件满足则执行对应的语句块，然后继续返回来判断条件；如果条件不满足则执行 while 语句块后面的语句。

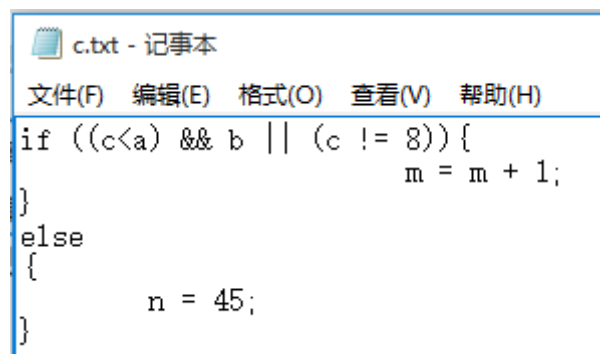


```
c.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
while (a < b) {
    while (x>y) {
        c = 9;
    }
    y = 4;
    z = 45;
}
```

图 9 测试用例 2

测试用例 3，测试布尔语句：

测试对于多个布尔条件组合起来使用能不能正确识别以及跳转是否正确，如果相邻的两个布尔条件是与的关系，第一个条件不满足就转 else 语句块，第一个条件如果满足则继续判断第二个条件；如果相邻的两个布尔条件是或的关系，第一个条件不满足则继续判断第二个条件，第一个条件如果满足则直接转 if 后面的语句块。

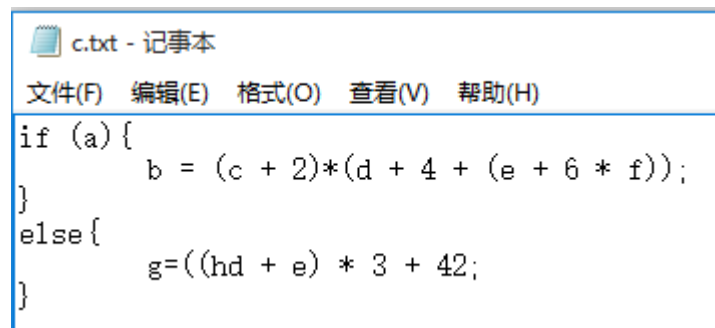


```
c.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
if ((c<a) && b || (c != 8)) {
    m = m + 1;
}
else
{
    n = 45;
}
```

图 10 测试用例 3

测试用例 4，测试算术表达式语句：

测试对于复杂的算术表达式程序能否正确生成其四元式，对于操作比较多的算术表达式，可采用中间变量来存储每步规约时的计算结果。



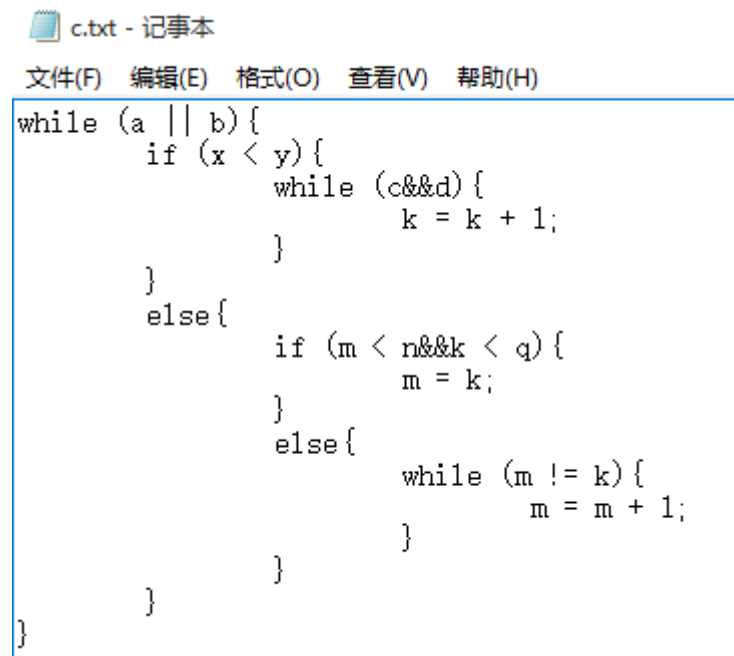
```
c.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

if (a) {
    b = (c + 2)*(d + 4 + (e + 6 * f));
}
else {
    g = (hd + e) * 3 + 42;
}
```

图 11 测试用例 4

测试用例 5，综合测试：

测试对于以上各种结构的组合，程序生成的四元式是否正确，各个分支、跳转等是否正确。



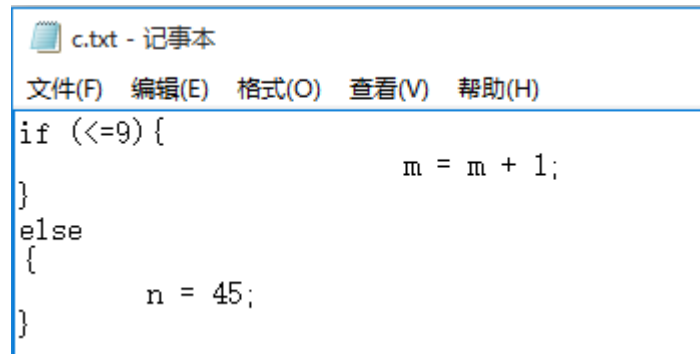
```
c.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

while (a || b) {
    if (x < y) {
        while (c && d) {
            k = k + 1;
        }
    }
    else {
        if (m < n && k < q) {
            m = k;
        }
        else {
            while (m != k) {
                m = m + 1;
            }
        }
    }
}
```

图 12 测试用例 5

测试用例 6，布尔表达式错误测试：

测试对于布尔表达式的错误，程序能否识别出来并给出错误位置和错误类型。测试用例中第 1 行代码中的布尔条件表达式有错误，缺少左操作数。



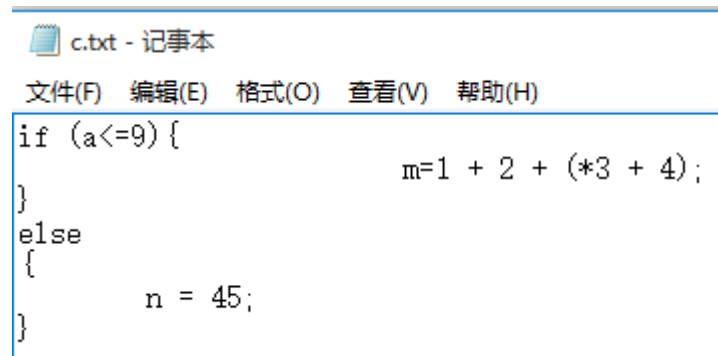
```
c.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

if (<=9) {
    m = m + 1;
}
else
{
    n = 45;
}
```

图 13 测试用例 6

测试用例 7，算术表达式错误测试 1：

测试对于算术表达式的错误，程序能否识别出来并给出错误位置和错误类型。测试用例中第 2 行代码中的算术表达式有错误，左括号右边出现一个乘号。



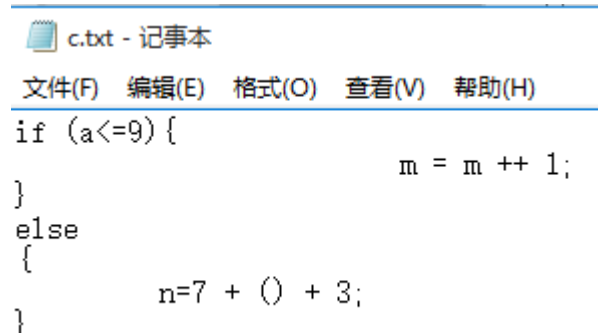
```
c.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

if (a<=9) {
    m=1 + 2 + (*3 + 4);
}
else
{
    n = 45;
}
```

图 14 测试用例 7

测试用例 8，算术表达式错误测试 2：

测试对于对于不同类型的算术表达式的错误，程序能否识别出来并给出错误位置和错误类型。测试用例中第 2 行代码中的算术表达式有错误，加号后面又出现一个加号；第 6 行代码的算术表达式中括号内没有任何内容。



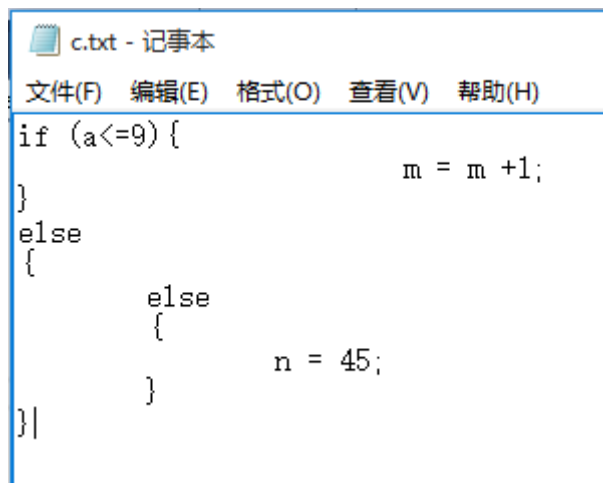
```
c.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

if (a<=9) {
    m = m ++ 1;
}
else
{
    n=7 + () + 3;
}
```

图 15 测试用例 8

测试用例 9，程序语句错误测试：

测试程序能否识别程序语句的错误，例中的程序语句的 else 中只有一个 else 语句块，很明显这是不对的，缺少与之对应的 if 语句块。



```
c.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
if (a<=9) {
                                m = m +1;
}
else
{
    else
    {
        n = 45;
    }
}|
```

图 16 测试用例 9

5.3 系统的运行结果及功能说明

测试用例 1：

```

词法分析识别如下（种别编码，自身值）：
(0, "if")
(74, "(")
(20, "x")
(50, ">")
(20, "y")
(75, ")")
(72, "{")
(0, "if")
(74, "(")
(20, "a")
(75, ")")
(72, "{")
(21, "3")
(20, "m")
(71, ".")
(56, "=")
(73, "}")
(20, "m")
(73, "}")
(52, "+")
(73, "}")
(21, "1")
(1, "else")
(71, ";")
(72, "{")
(20, "x")
(73, "}")
(56, "=")
(1, "else")
(72, "{")
(20, "y")
(0, "if")
(71, ".")
(74, "(")
(73, "}")
(20, "b")
(75, ")")
(72, "{")
(20, "m")
(56, "=")
(20, "m")
(54, "*")
(21, "2")
(71, ";")
(73, "}")
(1, "else")
(72, "{")
(20, "m")
(56, "=")
(0, 1)
(1, 2)
(2, 3)

```

标识符表如下：

```

( 0 , x )
( 1 , y )
( 2 , a )
( 3 , m )
( 4 , b )

```

常数表如下：

```

( 0 , 1 )
( 1 , 2 )
( 2 , 3 )

```

图 17 测试用例 1 的运行结果 1

分析栈如下:

状态栈	符号栈	动作)
0	#	s2
0 2	# if	s6
0 2 6	# if e	s4
0 2 6 4	# if e {	s2
0 2 6 4 2	# if e { if	s6
0 2 6 4 2 6	# if e { if e	s4
0 2 6 4 2 6 4	# if e { if e {	s5
0 2 6 4 2 6 4 5	# if e { if e { a	s10
0 2 6 4 2 6 4 5 10	# if e { if e { a ;	r4
用S->a;规约		
0 2 6 4 2 6 4 14	# if e { if e { S	r6
用L->S规约		
0 2 6 4 2 6 4 8	# if e { if e { L	s13
0 2 6 4 2 6 4 8 13	# if e { if e { L }	r3
用S->(L)规约		
0 2 6 4 2 6 11	# if e { if e S	s15
0 2 6 4 2 6 11 15	# if e { if e S else	s4
0 2 6 4 2 6 11 15 4	# if e { if e S else {	s2
0 2 6 4 2 6 11 15 4 2	# if e { if e S else { if	s6
0 2 6 4 2 6 11 15 4 2 6	# if e { if e S else { if e	s4
0 2 6 4 2 6 11 15 4 2 6 4	# if e { if e S else { if e {	s5
0 2 6 4 2 6 11 15 4 2 6 4 5	# if e { if e S else { if e { a	s10
0 2 6 4 2 6 11 15 4 2 6 4 5 10	# if e { if e S else { if e { a ;	r4
用S->a;规约		
0 2 6 4 2 6 11 15 4 2 6 4 14	# if e { if e S else { if e { S	r6
用L->S规约		
0 2 6 4 2 6 11 15 4 2 6 4 8	# if e { if e S else { if e { L	s13
0 2 6 4 2 6 11 15 4 2 6 4 8 13	# if e { if e S else { if e { L }	r3
用S->(L)规约		
0 2 6 4 2 6 11 15 4 2 6 11	# if e { if e S else { if e S	s15
0 2 6 4 2 6 11 15 4 2 6 11 15	# if e { if e S else { if e S else	s4
0 2 6 4 2 6 11 15 4 2 6 11 15 4	# if e { if e S else { if e S else {	s5
0 2 6 4 2 6 11 15 4 2 6 11 15 4 5	# if e { if e S else { if e S else { a	s10
万能五笔输入法 半 :况 13	# if e { if e S else { L }	r3

图 18 测试用例 1 的运行结果 2

0 2 6 4 2 6 11 15 4 2 6 11 15 4 5	# if e { if e S else { if e S else { a	s10
0 2 6 4 2 6 11 15 4 2 6 11 15 4 5 10	# if e { if e S else { if e S else { a ;	r4
用S->a;规约		
0 2 6 4 2 6 11 15 4 2 6 11 15 4 14	# if e { if e S else { if e S else { S	r6
用L->S规约		
0 2 6 4 2 6 11 15 4 2 6 11 15 4 8	# if e { if e S else { if e S else { L	s13
0 2 6 4 2 6 11 15 4 2 6 11 15 4 8 13	# if e { if e S else { if e S else { L }	r3
用S->(L)规约		
0 2 6 4 2 6 11 15 4 2 6 11 15 17	# if e { if e S else { if e S else S	r1
用S->if e S else S规约		
0 2 6 4 2 6 11 15 4 14	# if e { if e S else { S	r6
用L->S规约		
0 2 6 4 2 6 11 15 4 8	# if e { if e S else { L	s13
0 2 6 4 2 6 11 15 4 8 13	# if e { if e S else { L }	r3
用S->(L)规约		
0 2 6 4 2 6 11 15 17	# if e { if e S else S	r1
用S->if e S else S规约		
0 2 6 4 14	# if e { S	r6
用L->S规约		
0 2 6 4 8	# if e { L	s13
0 2 6 4 8 13	# if e { L }	r3
用S->(L)规约		
0 2 6 11	# if e S	s15
0 2 6 11 15	# if e S else	s4
0 2 6 11 15 4	# if e S else {	s5
0 2 6 11 15 4 5	# if e S else { a	s10
0 2 6 11 15 4 5 10	# if e S else { a ;	r4
用S->a;规约		
0 2 6 11 15 4 14	# if e S else { S	r6
用L->S规约		
0 2 6 11 15 4 8	# if e S else { L	s13
0 2 6 11 15 4 8 13	# if e S else { L }	r3
用S->(L)规约		
0 2 6 11 15 17	# if e S else S	r1
用S->if e S else S规约		
0 1	# S	acc

图 19 测试用例 1 的运行结果 3

输出源代码的中间代码（四元式）如下：

```

100      (j>      ,      x      ,      y      ,      102      )
101      (j      ,      _      ,      _      ,      114      )
102      (jnz     ,      a      ,      _      ,      104      )
103      (j      ,      _      ,      _      ,      107      )
104      (+      ,      m      ,      1      ,      T1      )
105      (=      ,      T1     ,      _      ,      m      )
106      (j      ,      _      ,      _      ,      113      )
107      (jnz     ,      b      ,      _      ,      109      )
108      (j      ,      _      ,      _      ,      112      )
109      (*      ,      m      ,      2      ,      T2      )
110      (=      ,      T2     ,      _      ,      m      )
111      (j      ,      _      ,      _      ,      113      )
112      (=      ,      3      ,      _      ,      m      )
113      (j      ,      _      ,      _      ,      115      )
114      (=      ,      y      ,      _      ,      x      )
115

```

图 20 测试用例 1 的运行结果 4

对照测试用例的程序代码分析图中的中间代码，首先比较 x 是否大于 y ，是则跳转到 102 执行 if 的语句块，即下一层的 if、else 语句；如果 $x < y$ 即 101，跳转到 114，执行 $y = x$ 语句，即 else 语句块，else 语句的上一条还应有一个跳过 else 语句块的跳转指令，即 113 (j, _, _, 115)。

测试用例 2:

(状态栈	符号栈	动作)
0	#	s3
0 3	# while	s7
0 3 7	# while e	s4
0 3 7 4	# while e {	s3
0 3 7 4 3	# while e { while	s7
0 3 7 4 3 7	# while e { while e	s4
0 3 7 4 3 7 4	# while e { while e {	s5
0 3 7 4 3 7 4 5	# while e { while e { a	s10
0 3 7 4 3 7 4 5 10	# while e { while e { a ;	r4
用S->a;规约		
0 3 7 4 3 7 4 14	# while e { while e { S	r6
用L->S规约		
0 3 7 4 3 7 4 8	# while e { while e { L	s13
0 3 7 4 3 7 4 8 13	# while e { while e { L }	r3
用S->{L} 规约		
0 3 7 4 3 7 12	# while e { while e S	r2
用S->while e S规约		
0 3 7 4 14	# while e { S	s5
0 3 7 4 14 5	# while e { S a	s10
0 3 7 4 14 5 10	# while e { S a ;	r4
用S->a;规约		
0 3 7 4 14 14	# while e { S S	s5
0 3 7 4 14 14 5	# while e { S S a	s10
0 3 7 4 14 14 5 10	# while e { S S a ;	r4
用S->a;规约		
0 3 7 4 14 14 14	# while e { S S S	r6
用L->S规约		
0 3 7 4 14 14 16	# while e { S S L	r5
用L->SL规约		
0 3 7 4 14 16	# while e { S L	r5
用L->SL规约		
0 3 7 4 8	# while e { L	s13
0 3 7 4 8 13	# while e { L }	r3
用S->{L} 规约		
0 3 7 12	# while e S	r2
用S->while e S规约		
0 1	# S	acc

万能五笔输入法 半 :马

图 21 测试用例 2 的运行结果：分析过程

```

输出源代码的中间代码（四元式）如下：
100      (j<    ,      a      ,      b      ,      102      )
101      (j      ,      _      ,      _      ,      109      )
102      (j>    ,      x      ,      y      ,      104      )
103      (j      ,      _      ,      _      ,      106      )
104      (=     ,      9      ,      _      ,      c      )
105      (j      ,      _      ,      _      ,      102      )
106      (=     ,      4      ,      _      ,      y      )
107      (=     ,      45     ,      _      ,      z      )
108      (j      ,      _      ,      _      ,      100      )
109

```

图 22 测试用例 2 的运行结果：中间代码

对于测试用例 2 中的代码，内层 while 循环结束后就首先跳转到这个循环的条件判断处，即 105 跳转到 102，内层的 while 语句块执行完后就执行该语句块后面的语句，即 103 内层 while 条件不满足时跳转到 106，执行 y=4，执行完 z=45 后继续跳转到外层 while 的条件判断，即 108 跳转到 100，外层 while 条件不满足时退出 while 循环，即 101 跳转到 109。

测试用例 3：

```

输出源代码的中间代码（四元式）如下：
100      (j<    ,      c      ,      a      ,      102      )
101      (j      ,      _      ,      _      ,      109      )
102      (jnz    ,      b      ,      _      ,      106      )
103      (j      ,      _      ,      _      ,      104      )
104      (j!=    ,      c      ,      8      ,      106      )
105      (j      ,      _      ,      _      ,      109      )
106      (+     ,      m      ,      1      ,      T1     )
107      (=     ,      T1     ,      _      ,      m      )
108      (j      ,      _      ,      _      ,      110     )
109      (=     ,      45     ,      _      ,      n      )
110
请按任意键继续. . .
万能五笔输入法 半 :

```

图 23 测试用例 3 的运行结果：中间代码

测试用例 3 测试的是布尔表达式，分析可知，当 $c < a$ 时由于后面是 $\&\&$ （与）操作，所以还应判断 $\&\&$ 后面的布尔操作数 (b)，即 100 跳转到 102；当 b 也是 true 时由于后面是 $\|\$ 操作（或），直接跳转到 106 执行 $m=m+1$ ；当 c 不大于 a 或 $c==8$ 时，即 if 的条件不满足，则跳转到 else 执行 $n=45$ ，即 101、105 跳转到 109。

测试用例 4:

```
输出源代码的中间代码（四元式）如下：
100      (jnz,      a      ,      _      ,      102      )
101      (j      ,      _      ,      _      ,      110      )
102      (+      ,      c      ,      2      ,      T1      )
103      (+      ,      d      ,      4      ,      T2      )
104      (*      ,      6      ,      f      ,      T3      )
105      (+      ,      e      ,      T3      ,      T4      )
106      (+      ,      T2      ,      T4      ,      T5      )
107      (*      ,      T1      ,      T5      ,      T6      )
108      (=      ,      T6      ,      _      ,      b      )
109      (j      ,      _      ,      _      ,      113      )
110      (+      ,      hd      ,      e      ,      T7      )
111      (*      ,      T7      ,      3      ,      T8      )
112      (+      ,      T8      ,      42      ,      T9      )
113
请按任意键继续. . . ■

万能五笔输入法 半 :
```

图 24 测试用例 4 的运行结果：中间代码

测试用例 4 测试的是布尔表达式，分析可知，对于第一个算术表达式，按优先级首先执行 $c+2$ ，结果保存在一个临时变量里，即 102: $T1=c+2$ ，后面的 $d+4+\dots$ 时首先执行 $d+4$ ，即 103: $T2=d+4$ ，下一步 $e+6*f$ 时先执行乘法，即 104: $T3=6*f$ ，接下来执行 105: $T4=e+T3$... 最后将计算的结果 $T6$ 保存在等号左边的变量 b 里，即 108 ($=, T6, _, b$)。

测试用例 5:

```
C:\Windows\system32\cmd.exe
输出源代码的中间代码（四元式）如下：
100      (jnz ,      a      ,      -      ,      104      )
101      (j      ,      -      ,      -      ,      102      )
102      (jnz ,      b      ,      -      ,      104      )
103      (j      ,      -      ,      -      ,      126      )
104      (j< ,      x      ,      y      ,      106      )
105      (j      ,      -      ,      -      ,      114      )
106      (jnz ,      c      ,      -      ,      108      )
107      (j      ,      -      ,      -      ,      113      )
108      (jnz ,      d      ,      -      ,      110      )
109      (j      ,      -      ,      -      ,      113      )
110      (+ ,      k      ,      1      ,      T1      )
111      (= ,      T1      ,      -      ,      k      )
112      (j      ,      -      ,      -      ,      106      )
113      (j      ,      -      ,      -      ,      125      )
114      (j< ,      m      ,      n      ,      116      )
115      (j      ,      -      ,      -      ,      120      )
116      (j< ,      k      ,      q      ,      118      )
117      (j      ,      -      ,      -      ,      120      )
118      (= ,      k      ,      -      ,      m      )
119      (j      ,      -      ,      -      ,      125      )
120      (j!= ,      m      ,      k      ,      122      )
121      (j      ,      -      ,      -      ,      125      )
122      (+ ,      m      ,      1      ,      T2      )
123      (= ,      T2      ,      -      ,      m      )
124      (j      ,      -      ,      -      ,      120      )
125      (j      ,      -      ,      -      ,      100      )
126
请按任意键继续. . .
万能五笔输入法 半 :见
```

图 25 测试用例 5 的运行结果：中间代码

该例子外层是 while 循环，里面是一个 if、else 结构，其中 if 的语句块是一个 while 循环，else 语句块又是一个 if、else 结构，再往里一层还有一个 while 语句。分析四元式，其中 100 到 125 是最外层 while 语句，125 跳转到 100 没有问题；120 到 124 是 else 语句块的最内层 while 语句，124 也要跳转到 120；113 处是内层的 if 语句执行结束，此时就跳过对应的 else 语句，但由于此时还属于最外层的 while 循环，所以是跳转到 125 而不是 126。

测试用例 6:

```

分析栈如下:
(状态栈      符号栈      动作)
0          #          s2
0 2        # if       s6

源代码中第1行布尔表达式有错误:      比较符号<=没有左操作数

0 2 6      # if e      s4
0 2 6 4      # if e {      s5
0 2 6 4 5      # if e { a      s10
0 2 6 4 5 10      # if e { a ;      r4
用S->a;规约
0 2 6 4 14      # if e { S      r6
用L->S规约
0 2 6 4 8      # if e { L      s13
0 2 6 4 8 13      # if e { L }      r3
用S->{L}规约
0 2 6 11      # if e S      s15
0 2 6 11 15      # if e S else      s4
0 2 6 11 15 4      # if e S else {      s5
0 2 6 11 15 4 5      # if e S else { a      s10
0 2 6 11 15 4 5 10      # if e S else { a ;      r4
用S->a;规约
0 2 6 11 15 4 14      # if e S else { S      r6
用L->S规约
0 2 6 11 15 4 8      # if e S else { L      s13
0 2 6 11 15 4 8 13      # if e S else { L }      r3
用S->{L}规约
0 2 6 11 15 17      # if e S else S      r1
用S->if e S else S规约
0 1          # S          acc

输出源代码的中间代码(四元式)如下:
100      (+      ,      m      ,      1      ,      T3      )
101      (=      ,      T1      ,      _      ,      T3      )
102      (j      ,      _      ,      _      ,      104      )
103      (=      ,      45      ,      _      ,      n      )
104
万能五笔输入法 半 : ■

```

图 26 测试用例 6 的运行结果

如图所示,当分析到 if 语句后面的布尔表达式时,程序给出了错误的位置(源代码第 1 行)和错误的原因(“<=”没有左操作数),并跳过该错误继续分析程序的其他语句。

测试用例 7:

```

(状态栈      符号栈      动作)
0          #          s2
0 2        # if       s6
0 2 6      # if e     s4
0 2 6 4          # if e {          s5

源代码中第2行算术表达式错误：左括号右边不能为*

0 2 6 4 5          # if e { a          s10
0 2 6 4 5 10       # if e { a ;        r4
用S->a;规约
0 2 6 4 14         # if e { S          r6
用L->S规约
0 2 6 4 8          # if e { L          s13
0 2 6 4 8 13       # if e { L }        r3
用S->{L}规约
0 2 6 11           # if e S            s15
0 2 6 11 15        # if e S else       s4
0 2 6 11 15 4      # if e S else {          s5
0 2 6 11 15 4 5    # if e S else { a          s10
0 2 6 11 15 4 5 10 # if e S else { a ;        r4
用S->a;规约
0 2 6 11 15 4 14   # if e S else { S          r6
用L->S规约
0 2 6 11 15 4 8    # if e S else { L          s13
0 2 6 11 15 4 8 13 # if e S else { L }        r3
用S->{L}规约
0 2 6 11 15 17     # if e S else S          r1
用S->if e S else S规约
0 1          # S          acc

输出源代码的中间代码（四元式）如下：
100      (j<= , a , 9 , 102 )
101      (j , _ , _ , 104 )
102      (+ , 1 , 2 , T1 )
103      (j , _ , _ , 105 )
104      (= , 45 , _ , n )
105
万能五笔输入法 半 :

```

图 27 测试用例 7 的运行结果

如图所示，当分析到 if 语句里的算术表达式时，程序给出了错误的位置（源代码第 2 行）和错误的原因（左括号右边出现一个不应该出现的乘号），算术表达式语句错误对程序语句的其他部分没有影响，故跳过该错误并继续分析程序的其他语句。

测试用例 8:

```
分析栈如下:
(状态栈      符号栈      动作)
0           #           s2
0 2         # if       s6
0 2 6       # if e     s4
0 2 6 4     # if e {   s5

源代码中第2行算术表达式错误: 出现两个连续的+

0 2 6 4 5   # if e { a   s10
0 2 6 4 5 10 # if e { a ;  r4
用S->a;规约
0 2 6 4 14   # if e { S   r6
用L->S规约
0 2 6 4 8    # if e { L   s13
0 2 6 4 8 13 # if e { L }  r3
用S->{L}规约
0 2 6 11     # if e S     s15
0 2 6 11 15  # if e S else s4
0 2 6 11 15 4 # if e S else { s5

源代码中第6行算术表达式错误: 小括号 ( ) 之间没有任何内容

0 2 6 11 15 4 5   # if e S else { a   s10
0 2 6 11 15 4 5 10 # if e S else { a ;  r4
用S->a;规约
0 2 6 11 15 4 14   # if e S else { S   r6
用L->S规约
0 2 6 11 15 4 8    # if e S else { L   s13
0 2 6 11 15 4 8 13 # if e S else { L }  r3
用S->{L}规约
0 2 6 11 15 17 # if e S else S   r1
用S->if e S else S规约
0 1           # S         acc

万能五笔输入法 半 :马
```

图 28 测试用例 8 的运行结果

如图所示,对于源代码中有多个算术表达式错误的情况,程序也能一一提示并跳过错误部分继续分析。

测试用例 9:

```
分析栈如下:
(状态栈      符号栈      动作)
0          #          s2
0 2        # if       s6
0 2 6      # if e     s4
0 2 6 4    # if e {    s5
0 2 6 4 5  # if e { a  s10
0 2 6 4 5 10 # if e { a ; r4
用S->a,规约
0 2 6 4 14  # if e { S  r6
用L->S规约
0 2 6 4 8   # if e { L  s13
0 2 6 4 8 13 # if e { L } r3
用S->{L}规约
0 2 6 11    # if e S    s15
0 2 6 11 15 # if e S else s4

源代码中第6行程序语句错误      没有与else匹配的if语句

输出源代码的中间代码(四元式)如下:
100      (j<= ,      a      ,      9      ,      102      )
101      (j      ,      _      ,      _      ,      105      )
102      (+      ,      m      ,      1      ,      T1      )
103      (=      ,      T1      ,      _      ,      m      )
104
请按任意键继续. . .
万能五笔输入法 半 :
```

图 29 测试用例 9 的运行结果

从图中可以看到, 程序给出了错误的位置(第 6 行)和错误信息(没有与 else 匹配的 if 语句), 由于程序的语句错误后当前栈顶的状态也就不对了, 故不继续进行后续的程序语句分析。

六、总结及心得体会

该系统能够处理由给定的语法规则书写的源代码, 该语法是以 C 语言为原型进行设计的, 具备基本的条件分支结构、循环结构和布尔表达式和算术表达式等。经过测试, 各种由上述结构的各种组合类型语句也能够识别, 并能够给出词法分析表、标识表、常数表、完整的分析过程和四元式代码等。在开始编写代码前足足花了两次课的时间设计相关的结构, 仔细考虑了各种可能出现的问题, 有了一个大体的框架以后才开始编写, 程序也经过多次修改, 不断完善, 用大量的例子

进行测试。但由于时间有限，有一部分问题则点到为止，没有进行深度地细化，我觉得掌握了相应的原理的方法即可。

在程序中也修正了一些在上机指导书的代码中存在的错误，比如说当一个 while 结构后面有一个算术表达式时，while 语句块执行结束后应进行什么操作取决于 while 语句块后面的一个字符，即这个算术表达式，也就是说要先把算术表达式识别出来后才能进行 while 语句块的下一步操作，比如说规约操作。但是 while 语句块最后有一个跳转回 while 的布尔条件判断的语句，该语句是在进行 while 语句块的规约时产生的，但此时算术表达式已经分析完毕，所以就导致了这个跳转语句的四元式出现在了 while 语句块后面的算术表达式之后，但这是不合理的。解决的方法是先根据栈和输入符号的状态综合考虑，预设一个当前输入符号，并从 SLR 分析表中取出操作指示，只有当当前是移进操作时才可以进一步分析布尔表达式和算术表达式，之后再进一步确定当前输入符号。

该程序是采用规范规约的方法进行设计的，所能识别的源程序依赖于所设计的文法，因此只有严格按照这个文法写的程序才能识别，否则会报错。代码出错时程序会提示错误位于源代码中的第几行以及是什么错误，对于算术表达式和布尔表达式，程序能够跳过错误的部分继续分析完整个源代码，因为程序语句分析、算术表达式分析和布尔表达式分析是独立开来的，分别用三个函数来写，算术表达式和布尔表达式出错了对于程序语句的分析影响不大。

另外这个系统的程序语句的产生式设计的不太好，有一些情况没有设计，因为一开始的时候我是想先随便设计一个简单的，如果运行成功了说明这个方法有效，再回来扩充。可是后来整个程序编写好了之后发现整个框架已经定型了，需要做的改动太大就放弃了。

采用 SLR 方法设计该系统首先要设计文法，然后画 DFA 图，再根据 DFA 写分析表，程序中根据文法设计代码结构。因此整个过程一定要细心，尤其越是靠前的步骤越要细心，宁可多检查几遍也要尽早发现错误，因为错误发现的越晚，后期需要做出的改动越大。在设计这个系统的过程中，我曾两次由于疏于检查，导致花费大量时间重新设计。一次是错把产生式 $S \rightarrow \text{if } e \text{ } S \text{ else } S$ 写成了 $S \rightarrow \text{if } e \text{ else } S$ ，还有一次是在画 DFA 时，其中的一个状态在输入一个符号后忘记把点往后移，两次都是在调试程序的时候才发现的，所以导致发现错误以后要回去重新

画 DFA 和分析表并修改程序。

通过此次课程设计，不仅加深了教材内容的理解，对词法分析、语法分析、语义分析和中间代码生成的原理、如何实现、涵义等有了更多的体会，增强了动手能力，对 SLR 分析方法的整个流程掌握的更加熟练，并且学到了很多处理和解决问题的经验和方法。

七、参考文献

- [1]胡元义. 编译原理教程（第三版）[M]. 西安电子科技大学出版社，2010. 10
- [2]胡元义. 编译原理教程（第三版）习题解析与上机指导[M]. 西安电子科技大学出版社，2012. 4