**实验二 语法分析程序**

**57119108 吴桐**

**一、实验目的**

通过本实验的编程实践，了解语法分析过程，根据构造的LR(1)表分析输入语句的正确性，是否符合规定的语法规则，深度理解语法分析程序设计的原理和构造方法，对编译的基本概念、原理和方法有完整的和清楚的理解。

**二、实验内容**

用C或C++语言实现LR(1)语法分析。输入字符串流，通过预先设定的上下文无关文法及其生成的LR(1) Parsing Table，通过自下而上的规约，判断该语句是否合法，并打印出规约过程；若遇到错误则显示Error。

**三、实验设计思路**

首先选定分析使用的语法规则，即加减乘除的算数表达式：

(0) E’🡪E

(1) E🡪E+F

(2) E🡪E-F

(3) E🡪F

(4) F🡪F\*G

(5) F🡪F/G

(6) F🡪G

(7) G🡪H

(8) G🡪(E)

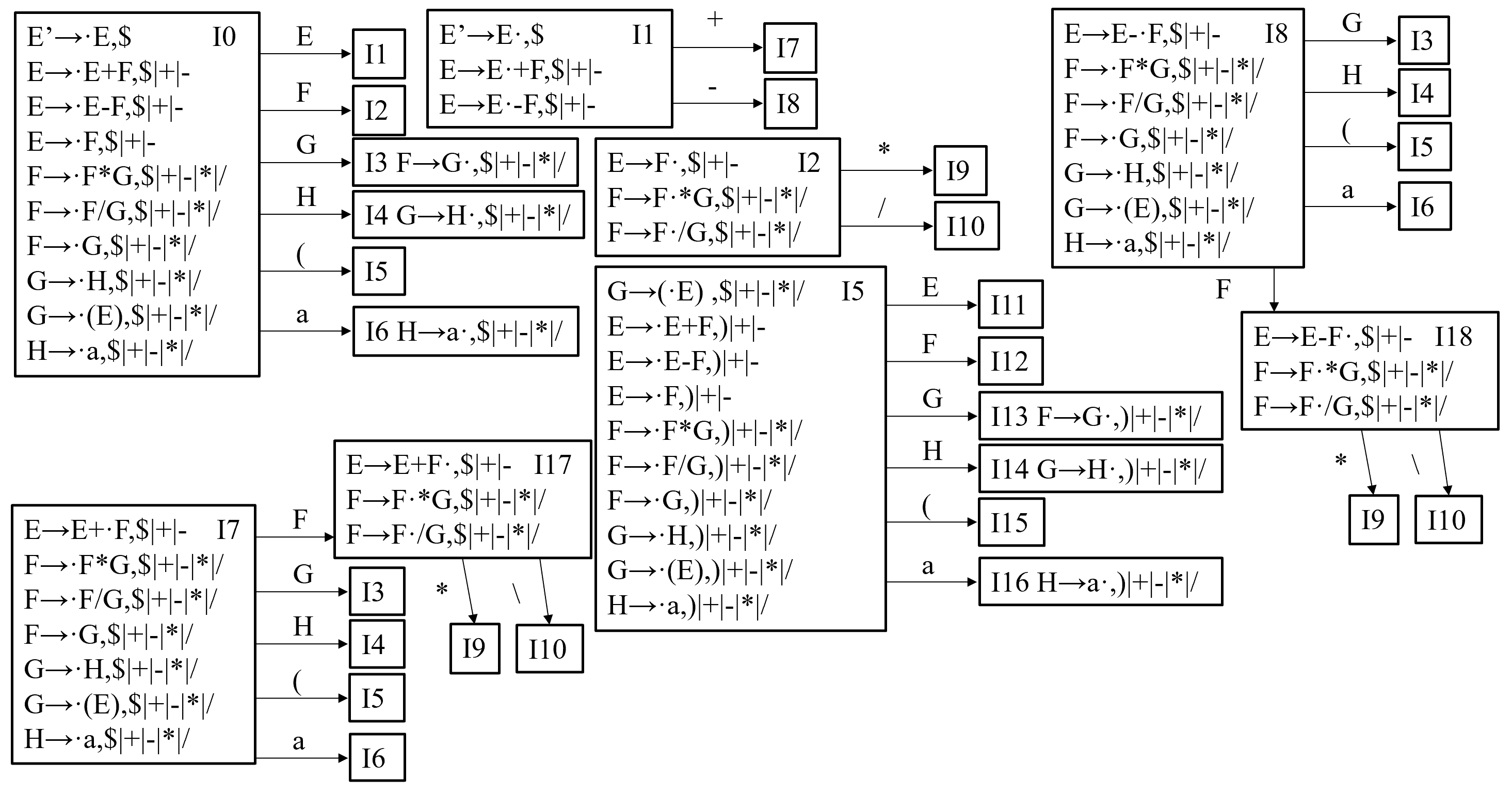
(9) H🡪a

根据以上语法规则构造DFA，之后根据DFA构造LR(1)分析表。编写程序按照预先设定的上下文无关文法及其生成的LR(1) Parsing Table，通过自下而上的规约，判断该语句是否合法。

在程序中设置状态栈S和符号栈B两个栈，根据读头项根据状态栈顶和读头项查找ACTION表，若是Sn（在程序中用+n表示），当前状态n入状态栈S，读头项入符号栈B，读头项向后移一位；若是Rn（在程序中用-n表示），则进行规约，将相应产生式的右部从符号栈弹出，以及对应的状态也弹出，然后将产生式左部入符号栈，并根据Parsing Table记录的GOTO部分，将相应的状态入栈。直到进行产生式0的规约后，表示该表达式符合预设语法，分析完成。

**四、DFA及LR(1) Parsing Table的设计**

根据语法规则构造的DFA如图1所示。相应的LR(1) Parsing Table如表1所示。



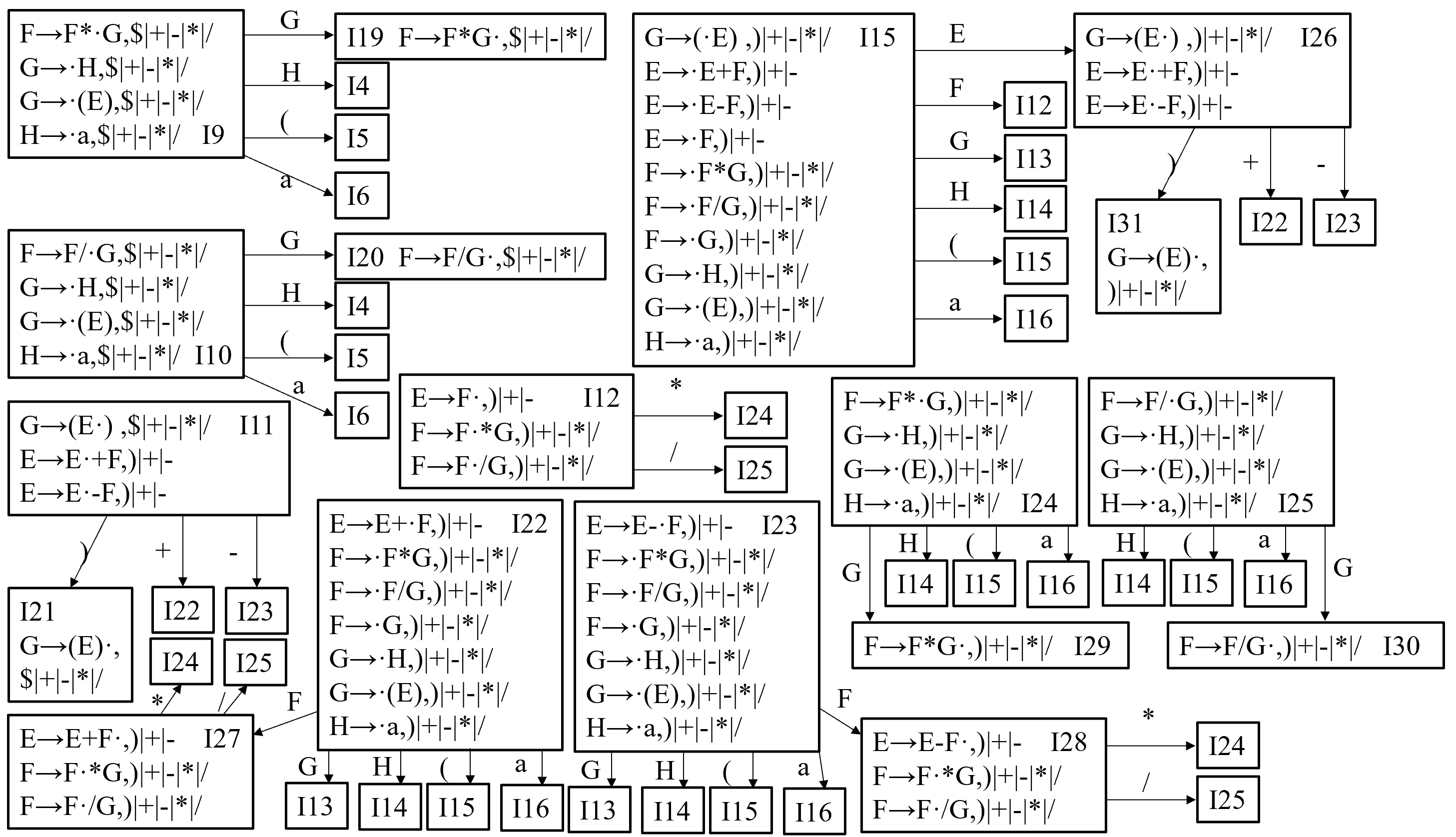


图1 语法分析DFA

表1 LR(1)分析表

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| States | ACTION | | | | | | | | GOTO | | | |
| + | - | \* | / | ( | ) | a | $ | E | F | G | H |
| 0 |  |  |  |  | S5 |  | S6 |  | 1 | 2 | 3 | 4 |
| 1 | S7 | S8 |  |  |  |  |  | acc |  |  |  |  |
| 2 | R3 | R3 | S9 | S10 |  |  |  | R3 |  |  |  |  |
| 3 | R6 | R6 | R6 | R6 |  |  |  | R6 |  |  |  |  |
| 4 | R7 | R7 | R7 | R7 |  |  |  | R7 |  |  |  |  |
| 5 |  |  |  |  | S15 |  | S16 | R9 | 11 | 12 | 13 | 14 |
| 6 | R9 | R9 | R9 | R9 |  |  |  | R9 |  |  |  |  |
| 7 |  |  |  |  | S5 |  | S6 |  |  | 17 | 3 | 4 |
| 8 |  |  |  |  | S5 |  | S6 |  |  | 18 | 3 | 4 |
| 9 |  |  |  |  | S5 |  | S6 |  |  |  | 19 | 4 |
| 10 |  |  |  |  | S5 |  | S6 |  |  |  | 20 | 4 |
| 11 | S22 | S23 |  |  |  | S21 |  |  |  |  |  |  |
| 12 | R3 | R3 | S24 | S25 |  | R3 |  |  |  |  |  |  |
| 13 | R6 | R6 | R6 | R6 |  | R6 |  |  |  |  |  |  |
| 14 | R7 | R7 | R7 | R7 |  | R7 |  |  |  |  |  |  |
| 15 |  |  |  |  | S15 |  | S16 |  | 26 | 12 | 13 | 14 |
| 16 | R9 | R9 | R9 | R9 |  | R9 |  |  |  |  |  |  |
| 17 | R1 | R1 | S9 | S10 |  |  |  | R1 |  |  |  |  |
| 18 | R2 | R2 | S9 | S10 |  |  |  | R2 |  |  |  |  |
| 19 | R4 | R4 | R4 | R4 |  |  |  | R4 |  |  |  |  |
| 20 | R5 | R5 | R5 | R5 |  |  |  | R5 |  |  |  |  |
| 21 | R8 | R8 | R8 | R8 |  |  |  | R8 |  |  |  |  |
| 22 |  |  |  |  | S15 |  | S16 |  |  | 27 | 13 | 14 |
| 23 |  |  |  |  | S15 |  | S16 |  |  | 28 | 13 | 14 |
| 24 |  |  |  |  | S15 |  | S16 |  |  |  | 29 | 14 |
| 25 |  |  |  |  | S15 |  | S16 |  |  |  | 30 | 14 |
| 26 | S22 | S23 |  |  |  | S31 |  |  |  |  |  |  |
| 27 | R1 | R1 | S24 | S25 |  | R1 |  |  |  |  |  |  |
| 28 | R2 | R2 | S24 | S25 |  | R2 |  |  |  |  |  |  |
| 29 | R4 | R4 | R4 | R4 |  | R4 |  |  |  |  |  |  |
| 30 | R5 | R5 | R5 | R5 |  | R5 |  |  |  |  |  |  |
| 31 | R8 | R8 | R8 | R8 |  | R8 |  |  |  |  |  |  |

**五、核心算法描述**

init函数进行初始化，为分析做好准备。

void init() {

str[len++] = '$'; //输入字符串结尾设为‘$’

topS = 0; //状态栈栈顶指针

topB = 0; //符号栈栈顶指针

po = 0; //读头项

S[topS++] = 0; //初始状态为0

B[topB++] = '$'; //初始字符为‘$’

}

程序的主体为scaner函数，该函数依据预先设定的上下文无关文法及其生成的LR(1) Parsing Table，对输入字符串进行规约分析。

设置状态栈S和符号栈B两个栈，根据读头项根据状态栈顶和读头项查找ACTION表，若是Sn（在程序中用+n表示），当前状态n入状态栈S，读头项入符号栈B，读头项向后移一位；若是Rn（在程序中用-n表示），则进行规约，将相应产生式的右部从符号栈弹出，以及对应的状态也弹出，然后将产生式左部入符号栈，并根据Parsing Table记录的GOTO部分，将相应的状态入栈。直到进行产生式0的规约后，表示该表达式符合预设语法，分析完成。

void scaner() {

while (1) {

int act = ACTION[S[topS - 1]][changeToNum(str[po])]; //根据状态栈顶和读头项查找ACTION表

if (act == 100) { //分析成功

printf("Success!\n");

break;

}

if (act > 0) { //Shift

S[topS++] = act; //状态n入状态栈S

B[topB++] = str[po]; //读头项入符号栈B

po++; //读头项后移

}

else if (act < 0) { //Reduce

act = -act;

for(int k = 0; k < rightSize[act]; ++k) { //按照产生式的右部长度进行弹栈

topS--; //对应的状态从状态栈弹出

topB--; //产生式的右部从符号栈弹出

}

B[topB++] = leftB[act]; //将产生式左部入符号栈

int state = GOTO[S[topS-1]][changeToNum(leftB[act])]; //根据GOTO表确定当前状态

S[topS++] = state; //当前状态如状态栈

prt(); //输出规约串

}

else { //ACTION表中无动作，说明出现错误

printf("Error!\n");

break;

}

}

}

在程序中，有一个辅助函数changeToNum，该函数可以将终结符或非终结符转换为数字，对应于ACTION和GOTO表的索引。之所以出现这个辅助函数，主要是因为程序中设定的LR(1) Parsing Table的存储方式，使用的是最简单的整型数组。

当每次进行规约后，调用输出函数prt，依次输出符号栈中的元素和读头项之后未处理的字符。

void prt() {

printf("=> ");

for (int i = 1; i < topS; ++i) { //输出符号栈中的元素

printf("%c", B[i]);

}

for (int i = po; i < len-1; i++) { //输出读头项之后未处理的字符

printf("%c", str[i]);

}

printf("\n");

}

其余参数及函数设置见源代码parser.c。

**六、测试用例**

输入用例如图2所示，相应的输出如图3所示。可见程序能实现对算术表达式的正确规约分析，输出规约串。

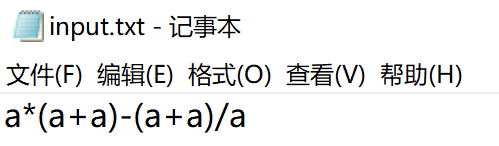


图2 输入用例

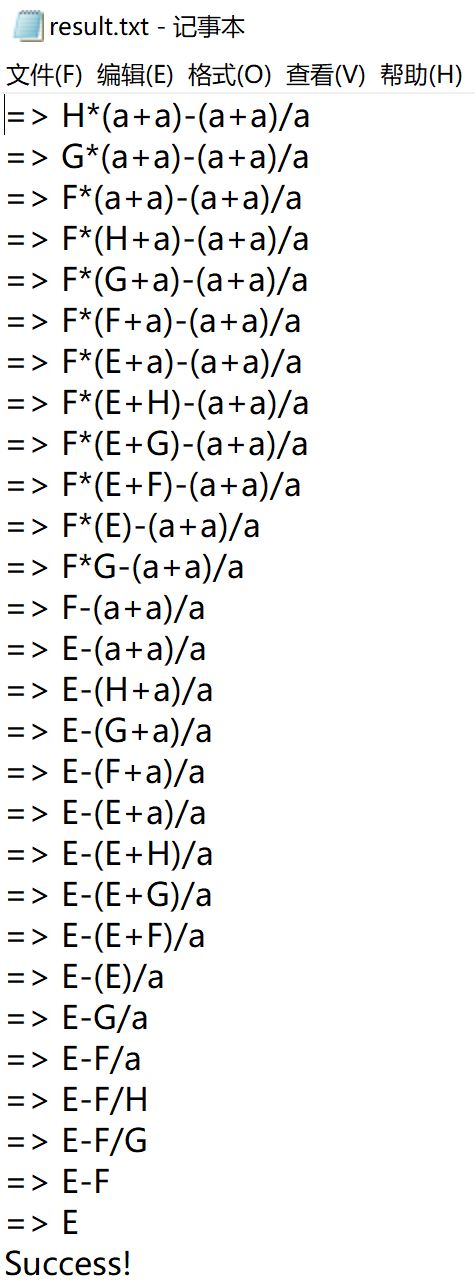


图3 输出用例

**七、出现的问题及解决办法**

本次实验的思路比较明确，大部分时间都花在了DFA和LR(1) Parsing Table的构造上。在实验过程中，也遇到了一些实现上的问题，例如：如何存储ACTION表和GOTO表，分析过程的实现，以及如何输出规约串等。

针对ACTION表和GOTO表的存储问题，我选择了最简单的方法：将表格用整型数组嵌入程序中，作为内部预设参数。我将所有的终结符与非终结符转换为数字（利用changeToNum函数），这样就可以直接将其作为索引查找表格。设计辅助函数略微增加了程序的复杂度。

也可以使用map将符号与Sn/Rn直接关联，这种方法更容易理解，但是map所需要的存储空间和查找时间远高于普通的整型数组查找，程序实现也更加困难。除此之外，还有自动生成表格、附加文件输入表格等方式，每种方式都有各自优点，但是显然不如整型数组方便高效。因此，在综合考虑各种方法的优缺点后，我确定了现在的解决方案。

不同的Parsing Table存储方式，会导致不同的分析过程实现。由于程序中设定的LR(1) Parsing Table的存储方式为整形数组，为了方便查表，我设置了专门的辅助函数changeToNum，该函数可以将终结符或非终结符转换为数字，对应于ACTION和GOTO表的索引。

规约串的输出，一开始我没有什么思路，但是在仔细观察符号栈和数据流的结构后，发现只要依次输出符号栈中的元素和读头项之后未处理的字符即可。只有进行规约操作后才需要调用输出函数。

**八、实验总结**

通过本次实验，我深入了解了语法分析的过程，加深了对编译器的工作原理的理解，利用编程解决实际问题，获得了编程经验。这次实验还有可以优化的地方，如将表格作为外部附加文件输入，或者直接自动生成表格，或者编写一个完整的Yacc，实现根据语法自动地生成语法分析程序。总的来说，本次实验从DFA设计到LR(1) Parsing Table的构造，再到具体程序实现，都需要极大的耐心和细心，让我体验到科研学习过程的严谨认真，从中也学习到了新的编程实践知识。