北京郵電大學

Beijing University of Posts and Telecommunications

北京邮电大学

计算机学院 (国家示范性软件学院)

语法分析程序的设计与实现

Author:

2019114514 班　田所浩二　2019114514

Course:
编译原理与技术

Supervisor:
刘辰副教授

2022 年 5 月 17 日

目录

1 实验简介

1.1 实验内容

- 编写语法分析程序, 实现对算术表达式的语法分析。要求所分析算数表达式已给出。
- 在对输入的算术表达式进行分析的过程中, 依次输出所采用的产生式。
- 编写 LL(1) 语法分析程序
- 编写语法分析程序实现自底向上的分析

1.2 实验目的

- 掌握语法分析方法。
- 熟悉 LL，LR 语法分析方法以及编程实现。
- 程序能完成对指定语言的语法分析。
- 了解语法分析过程中的错误处理方式。

1.3 实验环境

本实验在 MacOS 平台下，使用 C 语言编写实验程序代码，使用 clang 编译器编译。

2 LL(1) 语法分析程序

构造 LL(1) 语法分析程序的实验步骤由以下部分组成:

1. 构造 LL(1) 分析表
2. 分析过程详解
3. 错误处理

2.1 构造 LL(1) 分析表

2.1.1 LL(1) 语法应满足的条件

一个文法要能进行 LL（1）分析，那么这个文法应该满足：无二义性，无左递归，无左公因子。

实验中给出的文法含有左递归，所以第一步是消除左递归，结果如图1

2.1.2 构造 LL1 分析表的步骤

当文法满足条件后，再分别构造文法每个非终结符的 FIRST 和 FOLLOW 集合，然后根据 FIRST 和 FOLLOW 集合构造 LL（1）分析表，最后利用分析表，根据 LL(1) 语法分析构造一个分析器。

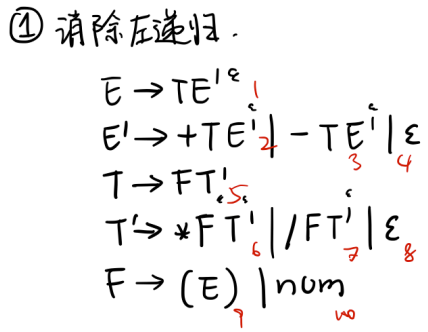


图 1: 消除左递归

2.1.3 构造 FIRST，FOLLOW 集

FIRST(A) 集合是非终结符号 A 的所有可能推导出的开头终结符或 组成的集合。称 FIRST(A) 为 A 的开始符号集或首符号集。

对于大部分文法而言，存在一个产生式存在多个候选式的情况，而选择哪一个候选式是不确定的，所以这就产生了回溯。回溯需要消耗大量的计算、存储空间，所以我们需要消除回溯。而消除回溯的其中一种方法叫作“预测”，即根据栈顶非终结符去预测后面的候选式，那预测方法就是求第一个非终结符，来判断是否和读头匹配，以达到预测的效果。

FOLLOW(A) 集合是所有紧跟 A 之后的终结符或 $ 所组成的集合（$ 是句尾的标志），称 FOLLOW(A) 是 A 的随符集。

当某一非终结符的产生式中含有空产生式时，它的非空产生式右部的开始符号集两两不相交，并与在推导过程中紧跟该非终结符右部可能出现的终结符集也不相交，则仍可构造确定的自顶向下分析。因此，引入了文法符号的后跟符号集合 FOLLOW。

FIRST, FOLLOW 集合的构造结果如图2

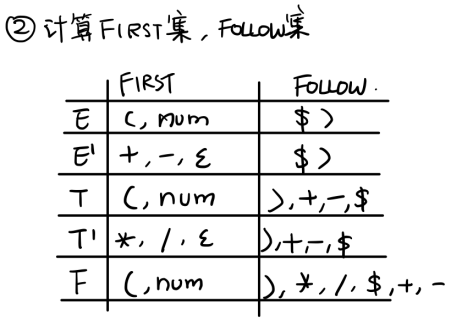


图 2: FIRST,FOLLOW 集合

2.1.4 构造分析表

至此，我们已经构造出了文法每个非终结符号的 FIRST，FOLLOW 集，接下来只需要根据一系列规则就可以简单的构造出 LL(1) 分析表了

设 M[A][a] 是一个二维数组，其中行 A 表示的是栈顶符号，a 表示的读头下的符号（A 为非终结符，a 为终结符），它们存放的是当前状态下所使用的候选式（或存放出错标志，指出 A 不该面临 a 的输入），称该数组 M 为文法的 LL(1) 分析表。

为了消除回溯，我们进行了 FIRST 集合和 FOLLOW 集合的求解。它们两个组合，达到了预测候选式的目的。为了使计算机比较好处理，把它们的预测结果统计成一张二维表。

构造 LL(1) 分析表需要经过如下步骤：

1. 对任意终结符号 $a \in First(A)$，将 $A \to a$ 填入 M[A,a]；
2. 如果存在 $\epsilon \in First(A)$，则对任意终结符号 $a \in Follow(A)$，将 $A \to a$ 填入 M[A,a]；
3. 将所有没有定义的 M[A,b] 设置为出错。

构造好的 LL(1) 分析表见图3

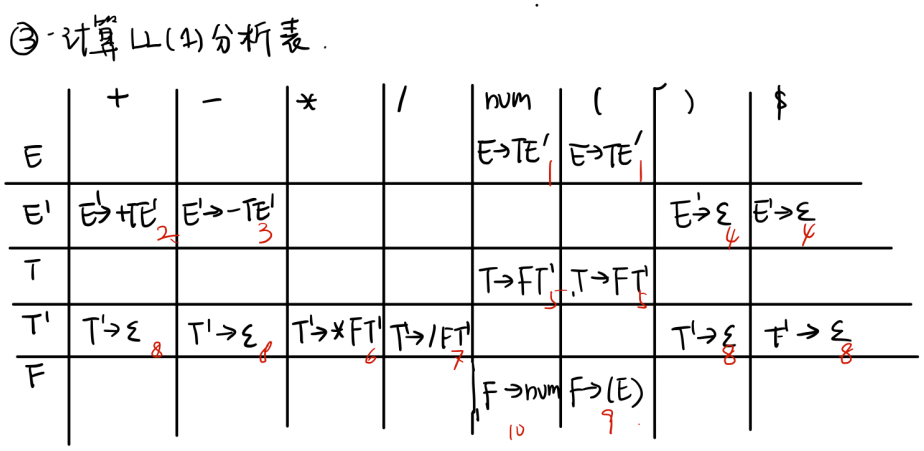


图 3: LL(1) 分析表

2.2 分析过程详解

预测分析程序的总控程序在任何时候都是按 STACK 栈顶符号 X 和当前的输入符号 a 行事的。如下图所示，对于任何 (X,a)，总控程序每次都执行下述三种可能的动作之一：

1. 若 X = a = '$'，则宣布分析成功，停止分析过程。
2. 若 X = a != '$'，则把 X 从 STACK 栈顶弹出，让 a 指向下一个输入符号。
3. 若 X 是一个非终结符，则查看分析表 M。
4. 若 M[X,a] 中存放着关于 X 的一个产生式，那么，先把 X 弹出 STACK 栈顶，然后把产生式的右部符号串按反序一一推进 STACK 栈（若右部符号为 ϵ，则意味着不推什么东西进栈）。
5. 若 M[X,a] 中存放着"出错标志"，则调用出错诊断程序 ERROR。

2.2.1 流程图

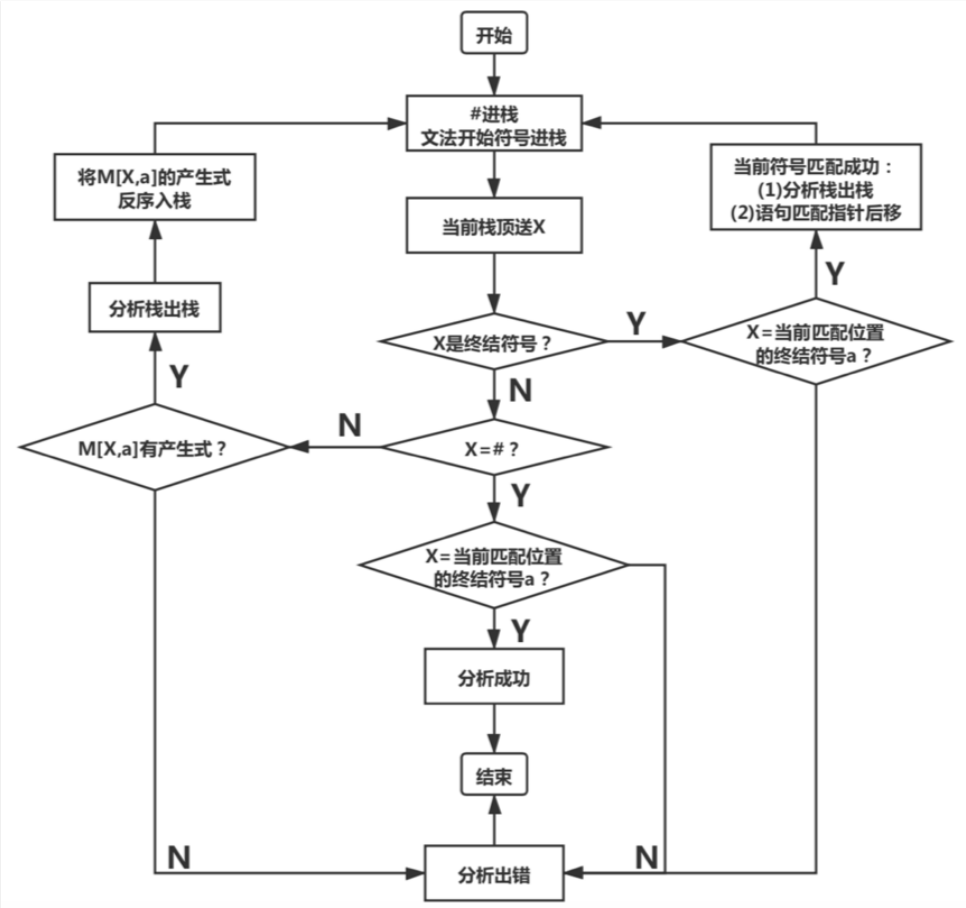


图 4: LL(1) 分析流程图

2.2.2 具体分析函数核心代码

```
int ll(int sym) {
    while(1) {
        // 取得栈顶符号
        int n = get();
        // 符号栈为空，输入栈也为空，接受
        if (n == -1 && sym == SYM_EOF)
            return STATUS_ACC;
        // 只有符号栈为空，出错
        else if (n == -1)
            return STATUS_ERR;
        if (n == sym) {
            printf("终结符出栈\n");
            pop(1);
            break;
        } else {
            int nentry = n2entry[n];
            int tentry = t2entry[sym];
```

```
            // 使用第ngen个生成式进行规约
            int ngen = LL[nentry][tentry];
            // 分析表中，没有对应项，即为错误
            if (ngen == 0) {
            return STATUS_ERR;
            }
            // 状态出栈
            pop(1);
            // 把生成式的符号反向入栈
            pusharr(gen[ngen].to, gen[ngen].len);
        }
    }
    return GOING;
}
```

代码块 1: LL(1) 分析过程代码

2.3 错误处理

发现错误，可以使用以下的错误处理方式:

1. 使用带有同步化信息的分析表
2. 不停弹栈，直到能够继续分析为止。

3 LR 语法分析程序

LR 分析法也是一种“移进—归约”的自底向上语法分析方法，其本质是规范归约【句柄作为可归约串】。其思想为一方面记住已移进和归约出的整个符号串，另一方面根据所用产生式推测未来可能碰到的输入符号。

构造 LR 语法分析程序由以下部分组成:

- 构造拓广文法
- 构造 LR(0) 有效项目族 DFA
- 如存在移进规约冲突，则计算每个非终结符的 FOLLOW 集合
- 如果用 FOLLOW 集合就可解决冲突，则为 SLR(1) 文法
- 如果不能解决冲突，则重新构造该文法的 LR(1) 有效项目族 DFA
- 最后根据 DFA 构造 LR 分析表

3.1 LR 分析表的构造

3.1.1 构造拓广文法

增加了一条右部为开始符号的产生式，就变成了拓广文法。这样可以创造出只有一个入口的文法。

构造出的拓广文法见图5

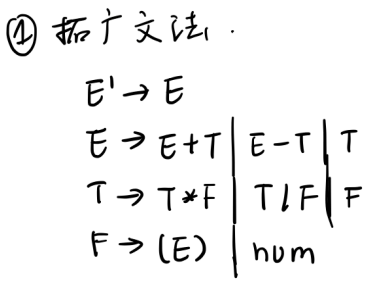


图 5: 拓广文法

3.1.2 构造 LR(0) 有效状态族 DFA

使用算法 4.5 构造文法 G 的 LR(0) 项目集规范族得到如图6的自动机

3.1.3 判断是否是 SLR(1) 文法

由图6可见，LR（0）项目集规范族存在移进规约冲突，但是通过 FOLLOW 集合就足够解决。所以文法是 SLR（1）文法。

3.1.4 构造 SLR(1) 分析表

使用算法 4.6 构造 SLR（1）分析表，得到如图7所示的分析表

3.2 分析过程

LR 分析方法：把“历史”以及“展望”综合抽象成状态；由栈顶的状态和现行的输入符号唯一确定每一步工作。

每一个符号对应一个状态，分析栈每次弹出一个符号，就要把对应的状态也弹出。然后 LR 分析程序会根据输入串在 LR 分析表中进行查找：是进行归约、移进还是报错操作。

LR 分析器实质上是一个带先进后出存储器（栈）的确定有限自动机，其核心部分是一张分析表，包括两部分：

（1）ACTION[s，a] 动作表，规定当状态 s 面临输入符号 a 时，应采取什么动作（移进、归约、接受、报错）【也就是告诉我们当栈顶状态为 s 时，输入的符号是 a 时，我们应该采取什么操作：归约、移进还是报错】

（2）GOTO[s，X] 状态转换表规定了状态 s 面对文法符号 X 时，下一状态是什么。【当归约完了后，要把规约后的非终结符压到栈里面的时候，跟新压入栈的这个非终结符所对应的状态是什么】

3.3 分析过程核心代码

```
int lr(int sym) {
  int goto_target;
  while (1) {
    int status = get();
    if (status == -1) {
      die("status stack empty");
```

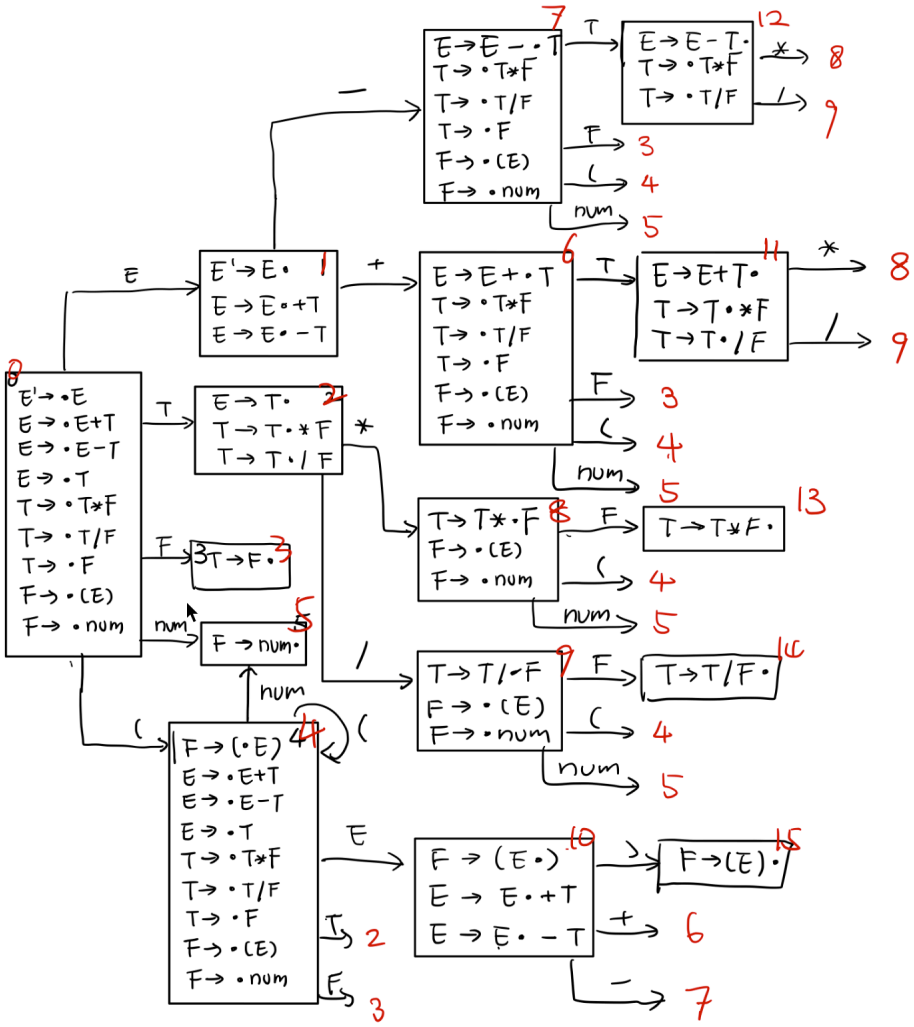
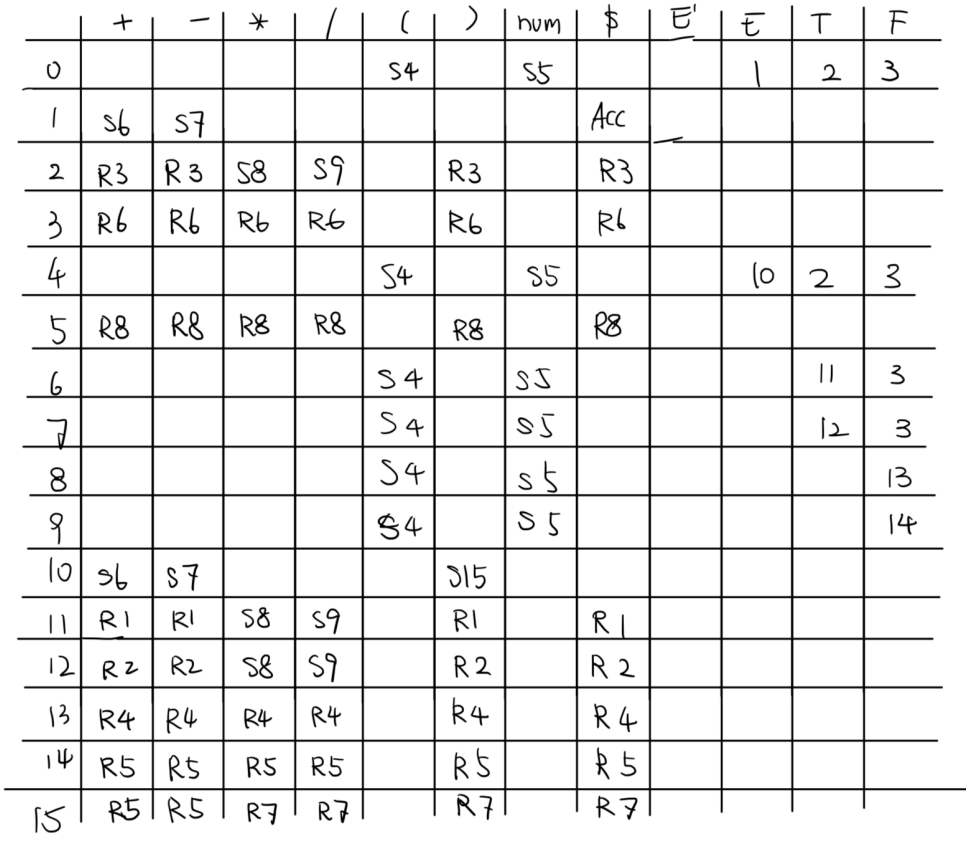


图 6: LR0 项目集规范族自动机



	+	−	*	/	(	)	num	$	E'	E	T	F
0					S4		S5			1	2	3
1	S6	S7						Acc				
2	R3	R3	S8	S9		R3		R3				
3	R6	R6	R6	R6		R6		R6				
4					S4		S5			10	2	3
5	R8	R8	R8	R8		R8		R8				
6					S4		S5				11	3
7					S4		S5				12	3
8					S4		S5					13
9					S4		S5					14
10	S6	S7				S15						
11	R1	R1	S8	S9		R1		R1				
12	R2	R2	S8	S9		R2		R2				
13	R4	R4	R4	R4		R4		R4				
14	R5	R5	R5	R5		R5		R5				
15	R5	R5	R7	R7		R7		R7				

图 7: SLR(1) 分析表

```
  return STATUS_ERR;
}
printstack(0);
printf("输入符号: %s ", getsymname(sym));
int symentry = symtoentry[sym];
struct action act = LR[status][symentry];
switch (act.type) {
case S:
  printf("移进, 到状态%d\n", act.go);
  push(act.go);
  // 主动退出函数, 直到下个符号被传入函数再次调用
  goto next_chr;
  break;
case R:
  printf("规约, 使用");
  printgen(gen[act.go]);
  /* printf("\n"); */
  if (pop(gen[act.go].len) == -1) {
    die("pop too much.");
    return STATUS_ERR;
  }
  // 根据状态栈出栈后栈顶元素与使用的生成式的左非终结符决定下个状态
  // GOTO表
```

```
      goto_target = gen[act.go].from;
      if ((status = get()) == -1) {
        die("stack empty");
        return STATUS_ERR;
      }
      symentry = symtoentry[goto_target];
      struct action act = LR[status][symentry];
      printf("GOTO: 转移到状态%d\n", act.go);
      push(act.go);
      break;
    case ACC:
      return STATUS_ACC;
      break;
    case ACTION_ERR:
      return STATUS_ERR;
      break;
    default:
      exit(1);
    }
  }
next_chr:
  return GOING;
}
```

3.4 流程图

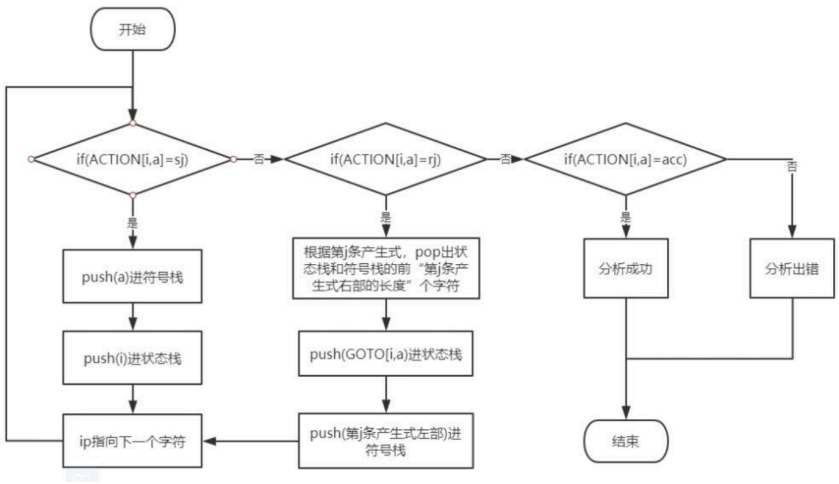


图 8: SLR(1) 分析流程图

3.5 错误处理

错误恢复策略：恐慌模式错误恢复、短语层次错误恢复。

3.5.1 恐慌模式错误恢复

1. 从栈顶向下扫描，直到发现某个状态 si，它有一个对应于某个非终结符 A 的 GOTO 目标，可以认为从这个 A 推导出的串中包含错误；
2. 然后丢弃 0 个或多个输入符号，直到发现一个可能合法地跟在 A 之后的符号 a 为止；
3. 之后将 $s_{i+1} = GOTO(si, A)$ 压入栈中，继续进行正常的语法分析。

3.5.2 短语层次错误恢复

1. 检查 LR 分析表中的每一个报错条目，并根据语言的使用方法来决定程序员所犯的何种错误最有可能引起这个语法错误；
2. 然后构造出适当的恢复过程。

4 实验总结

通过这次实验，我:

- 掌握了语法分析方法。
- 熟悉 LL，LR 语法分析方法以及编程实现。
- 程序能完成对指定语言的语法分析。
- 了解了语法分析过程中的错误处理方式。

之前对于 LR 语法分析中的一些细节理解的还不够深刻，比如为什么要有拓广文法等，但是通过这次实验，我结合实践更牢固的掌握了词法分析相关知识

4.1 实验结果

实验样例见5.1

测试命令:

```
$ ./ll < test1.txt > ll.txt
$ ./lr < test1.txt > lr.txt
```

ll.txt

```
栈:$ E          输入字符:NUM 生成式: E -> T E'
栈:$ E' T          输入字符:NUM 生成式: T -> F T'
栈:$ E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T'          输入字符:* 生成式: T' -> * F T'
栈:$ E' T' F *          输入字符:* 终结符出栈
栈:$ E' T' F          输入字符:( 生成式: F -> ( E )
栈:$ E' T' ) E (          输入字符:( 终结符出栈
栈:$ E' T' ) E          输入字符:NUM 生成式: E -> T E'
栈:$ E' T' ) E' T          输入字符:NUM 生成式: T -> F T'
栈:$ E' T' ) E' T' F          输入字符:NUM 生成式: F -> NUM
```

```
栈:$ E' T' ) E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T' ) E' T'          输入字符:+ 生成式: T' -> 空
栈:$ E' T' ) E'          输入字符:+ 生成式: E' -> + T E'
栈:$ E' T' ) E' T +          输入字符:+ 终结符出栈
栈:$ E' T' ) E' T          输入字符:( 生成式: T -> F T'
栈:$ E' T' ) E' T' F          输入字符:( 生成式: F -> ( E )
栈:$ E' T' ) E' T' ) E (          输入字符:( 终结符出栈
栈:$ E' T' ) E' T' ) E          输入字符:NUM 生成式: E -> T E'
栈:$ E' T' ) E' T' ) E' T          输入字符:NUM 生成式: T -> F T'
栈:$ E' T' ) E' T' ) E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' ) E' T' ) E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T' ) E' T' ) E' T'          输入字符:* 生成式: T' -> * F T'
栈:$ E' T' ) E' T' ) E' T' F *          输入字符:* 终结符出栈
栈:$ E' T' ) E' T' ) E' T' F          输入字符:( 生成式: F -> ( E )
栈:$ E' T' ) E' T' ) E' T' ) E (          输入字符:( 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E          输入字符:NUM 生成式: E -> T E'
栈:$ E' T' ) E' T' ) E' T' ) E' T          输入字符:NUM 生成式: T -> F T'
栈:$ E' T' ) E' T' ) E' T' ) E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' ) E' T' ) E' T' ) E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T'          输入字符:- 生成式: T' -> 空
栈:$ E' T' ) E' T' ) E' T' ) E'          输入字符:- 生成式: E' -> - T E'
栈:$ E' T' ) E' T' ) E' T' ) E' T -          输入字符:- 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T          输入字符:( 生成式: T -> F T'
栈:$ E' T' ) E' T' ) E' T' ) E' T' F          输入字符:( 生成式: F -> ( E )
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E (          输入字符:( 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E          输入字符:NUM 生成式: E -> T E'
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T          输入字符:NUM 生成式: T -> F T'
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T'          输入字符:/ 生成式: T' -> / F T'
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' F /          输入字符:/ 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T'          输入字符:+ 生成式: T' -> 空
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E'          输入字符:+ 生成式: E' -> + T E'
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T +          输入字符:+ 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T          输入字符:( 生成式: T -> F T'
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' F          输入字符:( 生成式: F -> ( E )
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E (          输入字符:( 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E          输入字符:NUM 生成式: E -> T E'
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T          输入字符:NUM 生成式: T -> F T'
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T'          输入字符:- 生成式: T' -> 空
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E'          输入字符:- 生成式: E' -> - T E'
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T -          输入字符:- 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T          输入字符:NUM 生成式: T -> F T'
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T'          输入字符:* 生成式: T' -> * F T'
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T' F *          输入字符:* 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T'          输入字符:/ 生成式: T' -> / F T'
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T' F /          输入字符:/ 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E' T'          输入字符:) 生成式: T' -> 空
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' ) E'          输入字符:) 生成式: E' -> 空
```

```
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T' )          输入字符:) 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E' T'          输入字符:) 生成式: T' -> 空
栈:$ E' T' ) E' T' ) E' T' ) E' T' ) E'          输入字符:) 生成式: E' -> 空
栈:$ E' T' ) E' T' ) E' T' ) E' T' )          输入字符:) 终结符出栈
栈:$ E' T' ) E' T' ) E' T' ) E' T'          输入字符:) 生成式: T' -> 空
栈:$ E' T' ) E' T' ) E' T' ) E'          输入字符:) 生成式: E' -> 空
栈:$ E' T' ) E' T' ) E' T' )          输入字符:) 终结符出栈
栈:$ E' T' ) E' T' ) E' T'          输入字符:) 生成式: T' -> 空
栈:$ E' T' ) E' T' ) E'          输入字符:) 生成式: E' -> 空
栈:$ E' T' ) E' T' )          输入字符:) 终结符出栈
栈:$ E' T' ) E' T'          输入字符:) 生成式: T' -> 空
栈:$ E' T' ) E'          输入字符:) 生成式: E' -> 空
栈:$ E' T' )          输入字符:) 终结符出栈
栈:$ E' T'          输入字符:/ 生成式: T' -> / F T'
栈:$ E' T' F /          输入字符:/ 终结符出栈
栈:$ E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T'          输入字符:* 生成式: T' -> * F T'
栈:$ E' T' F *          输入字符:* 终结符出栈
栈:$ E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T'          输入字符:- 生成式: T' -> 空
栈:$ E'          输入字符:- 生成式: E' -> - T E'
栈:$ E' T -          输入字符:- 终结符出栈
栈:$ E' T          输入字符:NUM 生成式: T -> F T'
栈:$ E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' NUM          输入字符:NUM 终结符出栈
栈:$ E' T'          输入字符:+ 生成式: T' -> 空
栈:$ E'          输入字符:+ 生成式: E' -> + T E'
栈:$ E' T +          输入字符:+ 终结符出栈
栈:$ E' T          输入字符:NUM 生成式: T -> F T'
栈:$ E' T' F          输入字符:NUM 生成式: F -> NUM
栈:$ E' T' NUM          输入字符:NUM 终结符出栈

栈:$ E' T'          输入字符:$ 生成式: T' -> 空
栈:$ E'          输入字符:$ 生成式: E' -> 空
栈:$          输入字符:$ LL(1) 分析成功! 分析过程如上
```

---

lr.txt

```
栈:$ 0 输入符号: NUM 移进, 到状态 5
栈:$ 0 5 输入符号: * 规约, 使用 F -> NUM GOTO: 转移到状态 3
栈:$ 0 3 输入符号: * 规约, 使用 T -> F GOTO: 转移到状态 2
栈:$ 0 2 输入符号: * 移进, 到状态 8
栈:$ 0 2 8 输入符号: ( 移进, 到状态 4
栈:$ 0 2 8 4 输入符号: NUM 移进, 到状态 5
栈:$ 0 2 8 4 5 输入符号: + 规约, 使用 F -> NUM GOTO: 转移到状态 3
栈:$ 0 2 8 4 3 输入符号: + 规约, 使用 T -> F GOTO: 转移到状态 2
栈:$ 0 2 8 4 2 输入符号: + 规约, 使用 E -> T GOTO: 转移到状态 10
栈:$ 0 2 8 4 10 输入符号: + 移进, 到状态 6
栈:$ 0 2 8 4 10 6 输入符号: ( 移进, 到状态 4
栈:$ 0 2 8 4 10 6 4 输入符号: NUM 移进, 到状态 5
栈:$ 0 2 8 4 10 6 4 5 输入符号: * 规约, 使用 F -> NUM GOTO: 转移到状态 3
栈:$ 0 2 8 4 10 6 4 3 输入符号: * 规约, 使用 T -> F GOTO: 转移到状态 2
栈:$ 0 2 8 4 10 6 4 2 输入符号: * 移进, 到状态 8
栈:$ 0 2 8 4 10 6 4 2 8 输入符号: ( 移进, 到状态 4
栈:$ 0 2 8 4 10 6 4 2 8 4 输入符号: NUM 移进, 到状态 5
栈:$ 0 2 8 4 10 6 4 2 8 4 5 输入符号: - 规约, 使用 F -> NUM GOTO: 转移到状态 3
栈:$ 0 2 8 4 10 6 4 2 8 4 3 输入符号: - 规约, 使用 T -> F GOTO: 转移到状态 2
```

```
栈:$ 0 2 8 4 10 6 4 2 8 4 2 输入符号: - 规约, 使用 E -> T GOTO: 转移到状态 10
栈:$ 0 2 8 4 10 6 4 2 8 4 10 输入符号: - 移进, 到状态 7
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 输入符号: ( 移进, 到状态 4
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 输入符号: NUM 移进, 到状态 5
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 5 输入符号: / 规约, 使用 F -> NUM GOTO: 转移到状态 3
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 3 输入符号: / 规约, 使用 T -> F GOTO: 转移到状态 2
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 2 输入符号: / 移进, 到状态 9
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 2 9 输入符号: NUM 移进, 到状态 5
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 2 9 5 输入符号: + 规约, 使用 F -> NUM GOTO: 转移到状态 14
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 2 9 14 输入符号: + 规约, 使用 T -> T / F GOTO: 转移到状态 2
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 2 输入符号: + 规约, 使用 E -> T GOTO: 转移到状态 10
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 输入符号: + 移进, 到状态 6
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 输入符号: ( 移进, 到状态 4
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 输入符号: NUM 移进, 到状态 5
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 5 输入符号: - 规约, 使用 F -> NUM GOTO: 转移到状态 3
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 3 输入符号: - 规约, 使用 T -> F GOTO: 转移到状态 2
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 2 输入符号: - 规约, 使用 E -> T GOTO: 转移到状态 10
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 输入符号: - 移进, 到状态 7
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 7 输入符号: NUM 移进, 到状态 5
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 7 5 输入符号: * 规约, 使用 F -> NUM GOTO: 转移到状态 3
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 7 3 输入符号: * 规约, 使用 T -> F GOTO: 转移到状态 12
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 7 12 输入符号: * 移进, 到状态 8
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 7 12 8 输入符号: NUM 移进, 到状态 5
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 7 12 8 5 输入符号: / 规约, 使用 F -> NUM GOTO: 转移到状态 13
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 7 12 8 13 输入符号: / 规约, 使用 T -> T * F GOTO: 转移到状态 12
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 7 12 输入符号: / 移进, 到状态 9
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 7 12 9 输入符号: NUM 移进, 到状态 5
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 7 12 9 5 输入符号: ) 规约, 使用 F -> NUM GOTO: 转移到状态 14
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 7 12 9 14 输入符号: ) 规约, 使用 T -> T / F GOTO: 转移到状态 12
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 7 12 输入符号: ) 规约, 使用 E -> E - T GOTO: 转移到状态 10
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 输入符号: ) 移进, 到状态 15
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 4 10 15 输入符号: ) 规约, 使用 F -> ( E ) GOTO: 转移到状态 3
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 3 输入符号: ) 规约, 使用 T -> F GOTO: 转移到状态 11
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 6 11 输入符号: ) 规约, 使用 E -> E + T GOTO: 转移到状态 10
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 输入符号: ) 移进, 到状态 15
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 4 10 15 输入符号: ) 规约, 使用 F -> ( E ) GOTO: 转移到状态 3
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 3 输入符号: ) 规约, 使用 T -> F GOTO: 转移到状态 12
栈:$ 0 2 8 4 10 6 4 2 8 4 10 7 12 输入符号: ) 规约, 使用 E -> E - T GOTO: 转移到状态 10
栈:$ 0 2 8 4 10 6 4 2 8 4 10 输入符号: ) 移进, 到状态 15
栈:$ 0 2 8 4 10 6 4 2 8 4 10 15 输入符号: ) 规约, 使用 F -> ( E ) GOTO: 转移到状态 13
栈:$ 0 2 8 4 10 6 4 2 8 13 输入符号: ) 规约, 使用 T -> T * F GOTO: 转移到状态 2
栈:$ 0 2 8 4 10 6 4 2 输入符号: ) 规约, 使用 E -> T GOTO: 转移到状态 10
栈:$ 0 2 8 4 10 6 4 10 输入符号: ) 移进, 到状态 15
栈:$ 0 2 8 4 10 6 4 10 15 输入符号: ) 规约, 使用 F -> ( E ) GOTO: 转移到状态 3
栈:$ 0 2 8 4 10 6 3 输入符号: ) 规约, 使用 T -> F GOTO: 转移到状态 11
栈:$ 0 2 8 4 10 6 11 输入符号: ) 规约, 使用 E -> E + T GOTO: 转移到状态 10
栈:$ 0 2 8 4 10 输入符号: ) 移进, 到状态 15
栈:$ 0 2 8 4 10 15 输入符号: / 规约, 使用 F -> ( E ) GOTO: 转移到状态 13
栈:$ 0 2 8 13 输入符号: / 规约, 使用 T -> T * F GOTO: 转移到状态 2
栈:$ 0 2 输入符号: / 移进, 到状态 9
栈:$ 0 2 9 输入符号: NUM 移进, 到状态 5
栈:$ 0 2 9 5 输入符号: * 规约, 使用 F -> NUM GOTO: 转移到状态 14
栈:$ 0 2 9 14 输入符号: * 规约, 使用 T -> T / F GOTO: 转移到状态 2
栈:$ 0 2 输入符号: * 移进, 到状态 8
栈:$ 0 2 8 输入符号: NUM 移进, 到状态 5
栈:$ 0 2 8 5 输入符号: - 规约, 使用 F -> NUM GOTO: 转移到状态 13
栈:$ 0 2 8 13 输入符号: - 规约, 使用 T -> T * F GOTO: 转移到状态 2
栈:$ 0 2 输入符号: - 规约, 使用 E -> T GOTO: 转移到状态 1
栈:$ 0 1 输入符号: - 移进, 到状态 7
```

```
栈:$ 0 1 7 输入符号: NUM 移进, 到状态 5
栈:$ 0 1 7 5 输入符号: + 规约, 使用 F -> NUM GOTO: 转移到状态 3
栈:$ 0 1 7 3 输入符号: + 规约, 使用 T -> F GOTO: 转移到状态 12
栈:$ 0 1 7 12 输入符号: + 规约, 使用 E -> E - T GOTO: 转移到状态 1
栈:$ 0 1 输入符号: + 移进, 到状态 6
栈:$ 0 1 6 输入符号: NUM 移进, 到状态 5

栈:$ 0 1 6 5 输入符号: $ 规约, 使用 F -> NUM GOTO: 转移到状态 3
栈:$ 0 1 6 3 输入符号: $ 规约, 使用 T -> F GOTO: 转移到状态 11
栈:$ 0 1 6 11 输入符号: $ 规约, 使用 E -> E + T GOTO: 转移到状态 1
栈:$ 0 1 输入符号: $ SLR(1) 分析成功! 分析过程如上
```

5 附录

5.1 测试样例

test1.txt

```
1*(1+(1*(2-(4/3+(5-9*3/2)))))/5*3-2+5
```

5.2 LL1 分析程序源代码

ll.c

```
#include "symbols.h"
#include "stack.h"
#include <stdlib.h>

const int t2entry[128] = {
['+'] = 0,
['-'] = 1,
['*'] = 2,
['/'] = 3,
[NUM] = 4,
['('] = 5,
[')'] = 6,
[0]   = 7,
};

const int n2entry[128] = {
[E]  = 0,
[E_] = 1,
[T]  = 2,
[T_] = 3,
[F]  = 4
};

struct generator gen[11] = {
    {},
    {E, {T, E_}, 2},
    {E_, {'+', T, E_}, 3},
    {E_, {'-', T, E_}, 3},
    {E_, {}, 0},
    {T, {F, T_}, 2},
    {T_, {'*', F, T_}, 3},
```

```
    {T_, {'/', F, T_}, 3},
    {T_, {}, 0},
    {F, {'(', E, ')'}, 3},
    {F, {NUM}, 1},
};

int LL[5][8] = {
    {0,0,0,0,1,1,0,0},
    {2,3,0,0,0,0,4,4},
    {0,0,0,0,5,5,0,0},
    {8,8,6,7,0,0,8,8},
    {0,0,0,0,10,9,0,0},
};

void printgen(struct generator gen) {
    printf("%s -> ", getsymname(gen.from));
    if (gen.len == 0)
        printf(" 空");
    for (int i = 0; i != gen.len; i++)
        printf("%s ", getsymname(gen.to[i]));
}

int ll(int sym) {
    while(1) {
        printstack(1);
        printf("\t 输入字符:%s ", getsymname(sym));
        int n = get();
        // 符号栈为空，输入栈也为空，接受
        if (n == -1 && sym == SYM_EOF)
            return STATUS_ACC;
        // 只有符号栈为空，出错
        else if (n == -1)
            return STATUS_ERR;
        if (n == sym) {
            printf(" 终结符出栈\n");
            pop(1);
            break;
        } else {
            int nentry = n2entry[n];
            int tentry = t2entry[sym];
            // 使用第 ngen 个生成式进行规约
            int ngen = LL[nentry][tentry];
            // 分析表中，没有对应项，即为错误
            if (ngen == 0) {
                return STATUS_ERR;
            }
            pop(1);
            printf(" 生成式: ");
            printgen(gen[ngen]);
            printf("\n");
            // 把生成式的符号反向入栈
            pusharr(gen[ngen].to, gen[ngen].len);
        }
    }
    return GOING;
}

int main() {
    push(E);
```

```
    int lex;
    int res = GOING;
    while(res == GOING) {
        lex = getsym();
        res = ll(lex);
        if (res == STATUS_ERR)
            die(" 分析失败, LL(1) 分析无法继续进行");
    }
    printf("LL(1) 分析成功! 分析过程如上\n");
}
```

5.3 SLR1 分析程序源代码

lr.c

```
#include "symbols.h"
#include "stack.h"

struct generator gen[9] = {
{E_, {E           }, 1 },
{E,  {E, '+', T   }, 3 },
{E,  {E, '-', T   }, 3 },
{E,  {T           }, 1 },
{T,  {T, '*', F   }, 3 },
{T,  {T, '/', F   }, 3 },
{T,  {F           }, 1 },
{F,  {'(', E, ')' }, 3 },
{F,  {NUM         }, 1 },
};

const int symtoentry[128] = {
    ['+'] = 0,
    ['-'] = 1,
    ['*'] = 2,
    ['/'] = 3,
    ['('] = 4,
    [')'] = 5,
    [NUM] = 6,
    [0]   = 7,
    [E_]  = 8,
    [E]   = 9,
    [T]   = 10,
    [F]   = 11,
};

struct action LR[16][12] = {
      //+     -      *      /      (      )       NUM    $      E'  E       T       F
/*0 */{{},    {},    {},    {},    {S,4}, {},     {S,5}, {},    {}, {G,1},  {G,2},  {G,3}},
/*1 */{{S,6}, {S,7}, {},    {},    {},    {},     {},    {ACC}, {}, {},     {},     {}},
/*2 */{{R,3}, {R,3}, {S,8}, {S,9}, {},    {R,3},  {},    {R,3}, {}, {},     {},     {}},
/*3 */{{R,6}, {R,6}, {R,6}, {R,6}, {},    {R,6},  {},    {R,6}, {}, {},     {},     {}},
/*4 */{{},    {},    {},    {},    {S,4}, {},     {S,5}, {},    {}, {G,10}, {G,2},  {G,3}},
/*5 */{{R,8}, {R,8}, {R,8}, {R,8}, {},    {R,8},  {},    {R,8}, {}, {},     {},     {}},
/*6 */{{},    {},    {},    {},    {S,4}, {},     {S,5}, {},    {}, {},     {G,11}, {G,3}},
/*7 */{{},    {},    {},    {},    {S,4}, {},     {S,5}, {},    {}, {},     {G,12}, {G,3}},
/*8 */{{},    {},    {},    {},    {S,4}, {},     {S,5}, {},    {}, {},     {},     {G,13}},
/*9 */{{},    {},    {},    {},    {S,4}, {},     {S,5}, {},    {}, {},     {},     {G,14}},
/*10*/{{S,6}, {S,7}, {},    {},    {},    {S,15}, {},    {},    {}, {},     {},     {}},
```

```
/*11*/{{R,1}, {R,1}, {S,8}, {S,9}, {},    {R,1},  {},    {R,1}, {}, {},     {},     {}},
/*12*/{{R,2}, {R,2}, {S,8}, {S,9}, {},    {R,2},  {},    {R,2}, {}, {},     {},     {}},
/*13*/{{R,4}, {R,4}, {R,4}, {R,4}, {},    {R,4},  {},    {R,4}, {}, {},     {},     {}},
/*14*/{{R,5}, {R,5}, {R,5}, {R,5}, {},    {R,5},  {},    {R,5}, {}, {},     {},     {}},
/*15*/{{R,5}, {R,5}, {R,7}, {R,7}, {},    {R,7},  {},    {R,7}, {}, {},     {},     {}},
};

void printgen(struct generator gen) {
    printf("%s -> ", getsymname(gen.from));
    if (gen.len == 0)
        printf(" 空");
    for (int i = 0; i != gen.len; i++)
        printf("%s ", getsymname(gen.to[i]));
}

int lr(int sym) {
    int goto_target;
    while(1) {
        int status = get();
        if (status == -1) {
            die("status stack empty");
            return STATUS_ERR;
        }
        printstack(0);
        printf(" 输入符号: %s ", getsymname(sym));
        int symentry = symtoentry[sym];
        struct action act = LR[status][symentry];
        switch(act.type) {
            case S:
                printf(" 移进, 到状态%d\n", act.go);
                push(act.go);
                // 主动退出函数, 直到下个符号被传入函数再次调用
                goto next_chr;
                break;
            case R:
                printf(" 规约, 使用");
                printgen(gen[act.go]);
                /* printf("\n"); */
                if(pop(gen[act.go].len) == -1) {
                    die("pop too much.");
                    return STATUS_ERR;
                }
                // 根据状态栈出栈后栈顶元素与使用的生成式的左非终结符决定下个状态
                // GOTO 表
                goto_target = gen[act.go].from;
                if((status = get()) == -1) {
                    die("stack empty");
                    return STATUS_ERR;
                }
                symentry = symtoentry[goto_target];
                struct action act = LR[status][symentry];
                printf("GOTO: 转移到状态%d\n", act.go);
                push(act.go);
                break;
            case ACC:
                return STATUS_ACC;
                break;
            case ACTION_ERR:
                return STATUS_ERR;
```

```
                break;
            default:
                exit(1);
        }
    }
next_chr:
    return GOING;
}

int main() {
    push(0);
    int lex;
    int res = GOING;
    while((lex = yylex()) != ERR && res == GOING) {
        res = lr(lex);
        if (res == STATUS_ERR) {
            die(" 分析失败, SLR(1) 分析无法继续进行");
            exit(1);
        }
    }
    if (res == STATUS_ACC)
        printf("SLR(1) 分析成功! 分析过程如上\n");
};
```