编译原理语法分析实验报告

一、实验题目与要求

> 题目一：认识Unix/Linux下的语法分析器，并完成简单规则的书写
>
> 题目二：利用JavaCC生成能够解析正整数加法、减法、乘法的解析器，在题目一的基础上实现初步代码化
>
> 题目三：分别使用LL(1)和LR(1)分析文法，设计并实现一个C/C++语言的语法分析程序

实验一的要求与解析：

1.用户编程要求

根据注释提示，在右侧编辑器（begin和end注释间）补充代码，实现加法(+)、减法(-)、乘法(*)、除法(/)、乘方(^)以及取负运算(n)！

对于本关，你只需要考虑写好规则段的（后缀表达式）定义即可。

图1 实验一编程要求

由题目要求（图1）可知，我们需要实现对于逆波兰表达式，基本四则运算、乘方以及取负的辨别，最后需要计算出该表达式的结果

2.YACC/Bison语法结构（实验二三略去）

1)语法规则

通过处理输入的规则段来确立分析规则，与先前我们了解的LEX/FLEX语法结构类似，其语法被分为三个部分，用 `%%` 进行分离，如图2所示：

```
/* 定义段 */
%{
C/C++头文件、全局文件、全局变量、类型定义
词法分析器yylex（采用lex进行词法分析）和错误打印函数
%}
Bison声明区间。定义之后用到的终结符、非终结符、操作符优先级
%%
/* 规则段 */
Bison语法规则定义
%%
/* 用户子程序段 */
C/C++代码 需要定义prologue区域函数，或者其他代码，生成的c/c++文件会完全拷贝这份代码。
```

图2 YACC/Bison语法结构示例

1)定义段：

这一部分一般是一些声明及选项设置等。C/C++语言的全局变量、头文件以及本次实验我们不需要考虑的词法分析器等放在 `%{ %}` 之间，这一部分的内容会被直接复制到输出文件的开头部分；

2)规则段：

为一系列匹配模式和动作，模式一般使用正则表达式书写，动作部分为C/C++代码：`模式1 {动作1(C代码)}`，在输入和 `模式 1` 匹配的时候，执行动作部分的代码。

由于Bison中使用的是BNF范式来描述产生式，这里先简单介绍一下BNF范式。BNF规定是推导规则(产生式)的集合，写为：`<符号> ::= <使用符号的表达式>` 。这里的 <符号> 是非终结符，而表达式由一个符号序列，或用指示选择的竖杠'|'分隔的多个符号序列构成，每个符号序列整体都是左端的符号的一种可能的替代。从未在左端出现的符号叫做终结符。

BNF类似一种数学游戏：从起始标志开始，给出替换先前符号的规则。BNF语法定义的语言是一个字符串集合，可以按照下述规则书写生产式规则，形式如下：`symbol := alternative1 | alternative2 ...`

3)用户子程序段：

仅含 `C` 代码，会被原样复制到输出文件中，一般这里定义一些辅助函数等，如动作代码中使用到的辅助函数。

2)符号定义规则

```
%token NUM
%nonassoc '<' /*表示该终结符无结合性 不能出现a<b<c*/
%left '+' '-' /*左结合 后面接操作符 下方的操作符比上方的优先级高*/
%left '*' '/'
%right NEG  NEG表示非
%right '^'
```

图3 终结符与非终结符定义规则

在图3当中，Token用于定义终结符，type定义非终结符；此外，与本实验相关性最大的操作符，也属于终结符，以left/right标识属于哪一类操作符，在本实验第五条“实验遇到的问题与思考”小项当中，有系统的介绍。

3.测试程序示例

测试输入：`4 5 +`，`3 4 ^`，`2 7 + 3 /`，`16 4 / 12 * n`；

预期输出：

```
9
81
3
-48
```

图4 实验一测试程序示例

如图4所示，我们可以从中得到输出格式，即该逆波兰表达式的计算结果，翻看测试集的输入样例，则是以 `!` 作为结尾，那么我们在用户子程序段实现对这一输入的处理。

4.实验通过截图

```
%{
  #include <ctype.h>
  #include <stdio.h>
  #include <math.h>
  int yylex (void);
  void yyerror (char const *);
%}


%define api.value.type {double}
%token NUM
%token NEG
%left '+' '-'        /* 左结合，低优先级 */
%left '*' '/'        /* 左结合，中等优先级 */
%right '^'           /* 右结合，最高优先级 */
%left NEG           /* 左结合，取反操作 */

%%

/* Grammar rules and actions follow.  */

input:
  %empty
| input line
;


line:
  '\n'
| exp '\n'      { printf ("%.10g\n", $1); }
;


exp:
  NUM           { $$ = $1; }
  | exp exp '+'  { $$ = $1 + $2; }
  | exp exp '-'  { $$ = $1 - $2; }
  | exp exp '*'  { $$ = $1 * $2; }
  | exp exp '/'  {
      if ($2 == 0) {
        yyerror("除数不能为零");
        $$ = 0;
```

```
48          } else {
49            $$ = $1 / $2;
50          }
51        }
52      | exp exp '^'  { $$ = pow($1, $2); }
53      | exp NEG { $$ = -$1; }
54    ;
55
56
57    %%
58
59    /* The lexical analyzer returns a double floating point
60       number on the stack and the token NUM, or the numeric code
61       of the character read if not a number.  It skips all blanks
62       and tabs, and returns 0 for end-of-input.  */
63
64
65    int yylex (void)
66    {
67      int c;
68
69      /* Skip white space.  */
70      while ((c = getchar ()) == ' ' || c == '\t')
71        continue;
72
73      /* Process numbers.  */
74      if (c == '.' || isdigit (c))
75        {
76          ungetc (c, stdin);
77          scanf ("%lf", &yylval);
78          return NUM;
79        }
80      /* 处理取反操作符 'n' */
81      if (c == 'n' || c == 'N')
82        return NEG;
83
84      /* Return end-of-input.  */
85      if (c == EOF)
86        return 0;
87      if (c == '!')
88        return 0;
89      /* Return a single char.  */
90      return c;
91    }
92
93    int main (int argc, char** argv)
94    {
95      return yyparse();
96    }
97
98
99    /* Called by yyparse on error.  */
100   void yyerror (char const *s)
101   {
102     fprintf (stderr, "%s\n", s);
103   }
```

图5 笔者编写的实验一程序（红框即遇到的问题）

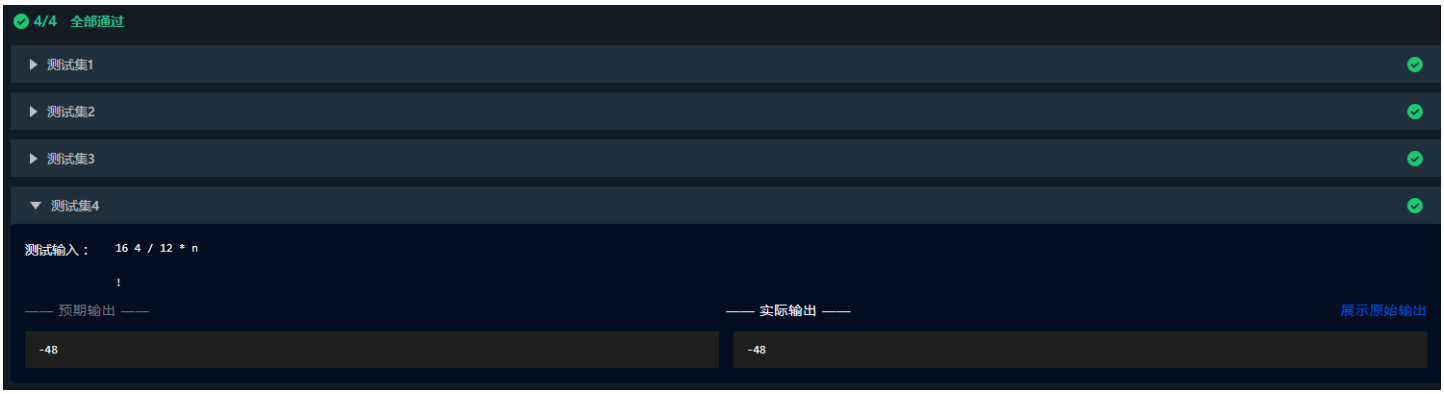


图6 实验一通过截图

笔者代码如图5所示，实验结果如图6所示。

5.实验遇到的问题与思考

1.关联操作符的分类问题

左关联和右关联操作符的区别尚不明晰，搜查有关资料，笔者总结如下：

左关联操作符（Left-associative operators）

左关联操作符是指在表达式中，操作符将从左到右结合操作数。这意味着当你有一个包含多个相同操作符的表达式时，最左边的操作会首先被执行。加法的例子如下：

- 表达式 `a + b + c`，会先计算 `a + b`，然后将结果与 `c` 相加。

右关联操作符（Right-associative operators）

右关联操作符是指在表达式中，操作符将从右到左结合操作数。这意味着当你有一个包含多个相同操作符的表达式时，最右边的操作会首先被执行。赋值操作符（`=`）的例子如下：

- 表达式 `a = b = c`，在C语言等许多编程语言中，会先计算 `b = c`，然后将结果赋值给 `a`。

区别

- **结合方向**：左关联操作符从左到右结合，而右关联操作符从右到左结合。
- **执行顺序**：在没有括号改变运算顺序的情况下，左关联操作符会先执行最左边的操作，而右关联操作符会先执行最右边的操作。

这里我们可以注意到，取反操作左右关联影响并不大，我们取其为左关联操作符；乘方按照人类的一般计算习惯，我们取其为右关联操作符。

2.取反的识别问题

此外示例给出的取反标记为 `NEG`，即negative的三位首字母，而在测试集中取反的输入标记为 `n`，这样会出现报错信息： `syntex error`。笔者在程序中进行了修正，程序读入 `n/N` 之后会返回 `NEG`，成功解决了这个问题。

实验二的要求与解析：

1.用户编程要求

根据提示，在右侧编辑器补充代码，完成能够解析正整数加法(+)、减法(-)、乘法(*)的解析器。

```
options {
    JavaCC的选项
}

PARSER_BEGIN(解析器类名)
package 包名;
import 库名;

public class 解析器类名 {
    任意的Java代码
}
PARSER_END(解析器类名)

扫描器的描述

解析器的描述
```

图7 实验二编程要求以及JavaCC语法结构

JavaCC的源文件是*.jj文件，其语法结构如图7所示，我们需要根据按照JavaCC的语法规则和代码结构，构造可以识别并解析正整数加减法以及乘法构成的中缀表达式的语法分析器。

2.编程样例

```
options {
    STATIC = false;
}

PARSER_BEGIN(Adder)
import java.io.*;

class Adder {
    public static void main(String[] args) {
        /*从命令行参数中读取待解析的字符串*/
        for (String arg : args) {
            try {
                System.out.println(evaluate(arg));
            } catch (ParseException ex) {
                System.err.println(ex.getMessage());
            }
        }
    }
    /*定义计算的方法*/
    public static long evaluate(String src) throws ParseException {
        Reader reader = new StringReader(src);
        return new Adder(reader).expr();
    }
}
PARSER_END(Adder)
/*忽视空格和换行*/
SKIP: { <[" ", "\t", "\r", "\n"]> }

TOKEN: {
    <INTEGER: (["0"-"9"])+>
}
/*定义规则*/
long expr():
{
    Token x, y;
}
{
    x=<INTEGER> "+" y=<INTEGER> <EOF>
    {
        return Long.parseLong(x.image) + Long.parseLong(y.image);
    }
}
```

图8 实验二JavaCC编程样例

如图8所示，我们以此样例框架构造我们需要的语法分析器，需要注意的是，本实验需要在最后输出表达式的计算结果，那么就需要一个中间变量时刻存储这一结果，这部分应当在“定义规则”部分实现。

3.测试程序示例

测试输入： 4+5 12+7+9 27-9 2+7*9-2*6

预期输出： 9 28 18 53

图9 实验二测试程序示例

如图9，我们可以得知输入结束符号为EOF，直接输出中缀表达式的计算结果。

4.实验通过截图

```
/* JavaCC 小测试 */
/* 功能：实现一个能够进行加法(+)，减法(-)，乘法(*)的计算器 */
/* 说明：在下面的begin和end之间添加代码，已经实现了简单的加法(+)，你需要完成剩下的部分，加油吧！ */
/* 提示： */

options {
    STATIC = false;
}

PARSER_BEGIN(Calc)
import java.io.*;

class Calc {
    public static void main(String[] args) {
        for (String arg : args) {
            try {
                System.out.println(evaluate(arg));
            } catch (ParseException ex) {
                System.err.println(ex.getMessage());
            }
        }
    }

    public static long evaluate(String src) throws ParseException {
        Reader reader = new StringReader(src);
        return new Calc(reader).expr();
    }
}
PARSER_END(Calc)
/* begin */
SKIP: { <[" ", "\t", "\r", "\n"]> }

TOKEN: {
    < INTEGER: (["0"-"9"])+ >
    | < PLUS: "+" >
    | < MINUS: "-" >
    | < MULTIPLY: "*" >
}

/* 解析表达式，支持加法和减法 */
long expr() :
{
    long value;
}
{
    value = term() (
        ( <PLUS> { value += term(); } )
      | ( <MINUS> { value -= term(); } )
```

```
    )*
    <EOF>
    {
        return value;
    }
}

/* 解析项，支持乘法 */
long term() :
{
    long value;
}
{
    value = factor() (
        ( <MULTIPLY> { value *= factor(); } )
    )*
    {
        return value;
    }
}

/* 解析因子，这里仅支持整数 */
long factor() :
{
    Token x;
}
{
    x = <INTEGER>
    {
        return Long.parseLong(x.image);
    }
}
 /* end */
```

图10 笔者编写的实验二程序

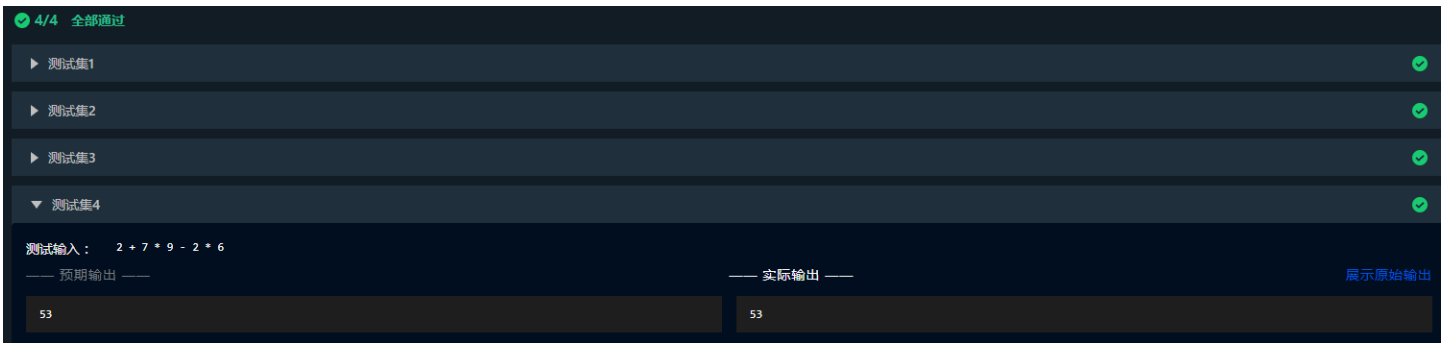


图11 实验二通过截图

笔者代码如图10所示，实验结果如图11所示。

5.实验遇到的问题与思考

1.题目要求错误的问题

题目要求原先为“实现一个能够进行加法(+)，减法(-)，**乘方(^)**的计算器”，在此时已经被笔者本地更正为“实现一个能够进行加法(+)，减法(-)，**乘法(*)**的计算器”

2.中间变量的确定

针对前文所说的中间变量存储表达式值的问题，我们选定value变量为中间变量，在 `expr` 和 `term` 函数中被定义和使用。

3.JavaCC语法的认识

笔者给出对JavaCC常用解析规则的认识：

`expr`、`term` 和 `factor` 是三个主要的解析规则，它们分别对应不同的解析级别：

1. **factor**：最低级别的解析规则，用于解析单个因子。在本实验中，`factor` 规则被定义为解析一个整数（`INTEGER`），并将其转换为 `long` 类型的值。

```
long factor() {
    Token x;
    <INTEGER> x = <INTEGER>;
    {
        return Long.parseLong(x.image);
    }
}
```

2. **term**：用于解析项，由一个或多个因子组成，这些因子可以通过乘法操作符连接。`term` 规则调用 `factor` 规则来获取因子的值，并在遇到乘法操作符时将因子的值相乘。

```
long term() {
    long value;
    value = factor();
    (<MULTIPLY> { value *= factor(); })*
    return value;
}
```

3. **expr**：最高级规则，用于解析整个表达式。它由一个或多个项组成，这些项可以通过加法或减法操作符连接。`expr` 规则调用 `term` 规则来获取项的值，并在遇到加法或减法操作符时更新表达式的值。

```
long expr() {
    long value;
    value = term();
    (<PLUS> { value += term(); })*
    (<MINUS> { value -= term(); })*
    return value;
}
```

实验三(1)的要求与解析：

1.用户编程要求

为方便代码评判，本关只考虑第一次遇到错误就报错并停止分析的错误处理方式。
同学们可以下去自己尝试考虑加入对应的错误信息，以及如何进行错误恢复，以便继续分析完整的输入串来获取可能更多的错误信息。

编写一个LL(1)语法分析程序，能对算数表达式进行语法分析。

要求:
在给定的消除左递归的文法G'的基础上：

编号	产生式
1	E -> TA
2	A -> +TA
3	A -> -TA
4	A -> ε
5	T -> FB
6	B -> *FB
7	B -> /FB
8	B -> ε
9	F -> (E)
10	F -> num

1. 编程实现算法4.2，为给定文法自动构造预测分析表。
2. 编程实现算法4.1，构造LL(1)预测分析程序。

图12 实验三(1)编程要求

实验要求如图12，我们可以得知，题目希望我们实现一个基于LL(1)文法的预测分析表，以此来识别或能被接收或不被接收的符号串，输出识别过程的预测分析程序；此外任务还提示我们可以在完成任务之外，额外选配一些功能，为我们留足了**发挥空间**。

2.平台环境说明

编译器版本：gcc7.3.0
OS版本：Debian GNU/Linux 9

图13 实验三(1)环境说明

由图13我们可以得知，平台的实验环境为gcc7+Linux9的结合，也不采用非标准库，故正常编写C++语言程序即可。

3.LL(1)预测分析表

	E	T	A	F	B
FIRST	(,num	(,num	+,-,ε	(,num	*,/,ε
FOLLOW	$,)	$,),+,-	$,)	$,),+,-*,/	$,),+,-

	+	-	*	/	(	)	num	$
E					E → TA		E → TA	
T					T → FB		T → FB	
A	A → +TA	A → -TA				A → ε		A → ε
F					F → (E)		F → num	
B	B → ε	B → ε	B → *FB	B → /FB		B → ε		B → ε

图14 实验三(1)LL(1)FIRST&FOLLOW&预测分析表

针对C语言的C11标准，先按照八种单词符号类型进行初步识别（关键字、标识符、运算符、分隔符、字符常量、字符串常量、数值常量以及错误），在每种类型的分支下，针对每一个范围内的字符做判断和细分识别，遇到复杂的函数和状态变量，就先封装再细化，最终构造出如上图的自动机。

4.测试程序示例

1）输入格式

从标准输入（使用 cin/scanf 等输出）读入数据。

输入仅包含1行：

- 第1行输入为一个算术表达式。构成该算术表达式的字符有：{'n'， '+'， '-'， '*'， '/'， '('， ')'}。

2）输出格式

输出到标准输出（使用 cout/printf 等输出）中。

输出包括若干行LL(1)分析过程：

假设输出有n行，则第i行(1⩽i⩽n)表示分析进行到第i步，它的输出包含**制表符 `\t` 分隔**的三个部分：

- **分析栈**：以 $ 符号表示栈底的字符串(左侧为栈底；由终结符和非终结符构成)
- **输入串栈**：以 $ 符号表示栈底的字符串(右侧为栈底)
- **分析动作**：产生式编号 或 match 或 error 或 accept(表示当前步骤应执行的动作)

4）简单测试样例：

图15 实验三(1)样例输入

```
$E      n*(n+$  1
$AT     n*(n+$  5
$ABF    n*(n+$  10
$ABn    n*(n+$  match
$AB     *(n+$   6
$ABF*   *(n+$   match
$ABF    (n+$    9
$AB)E(  (n+$    match
$AB)E   n+$     1
$AB)AT  n+$     5
$AB)ABF n+$     10
$AB)ABn n+$     match
$AB)AB  +$      8
$AB)A   +$      2
$AB)AT+ +$      match
$AB)AT  $       error
```

图16 实验三(1)样例输出

5）简单样例分析：

样例的输入中，包含一行或合法或不合法的符号串。

样例的第一列输出中，包含了分析栈当前状态下的所有符号，最右端为栈顶。

第二列输出为输入串栈当前状态下的所有符号，最右端为栈底。

第三列输出则是程序的动作依据，或是推导所凭借的产生式编号，或是分析栈与输入串栈栈顶匹配，或是最终成功接收/无法接收而报错。

5.实验通过截图

源代码可见第四部分的最后一点

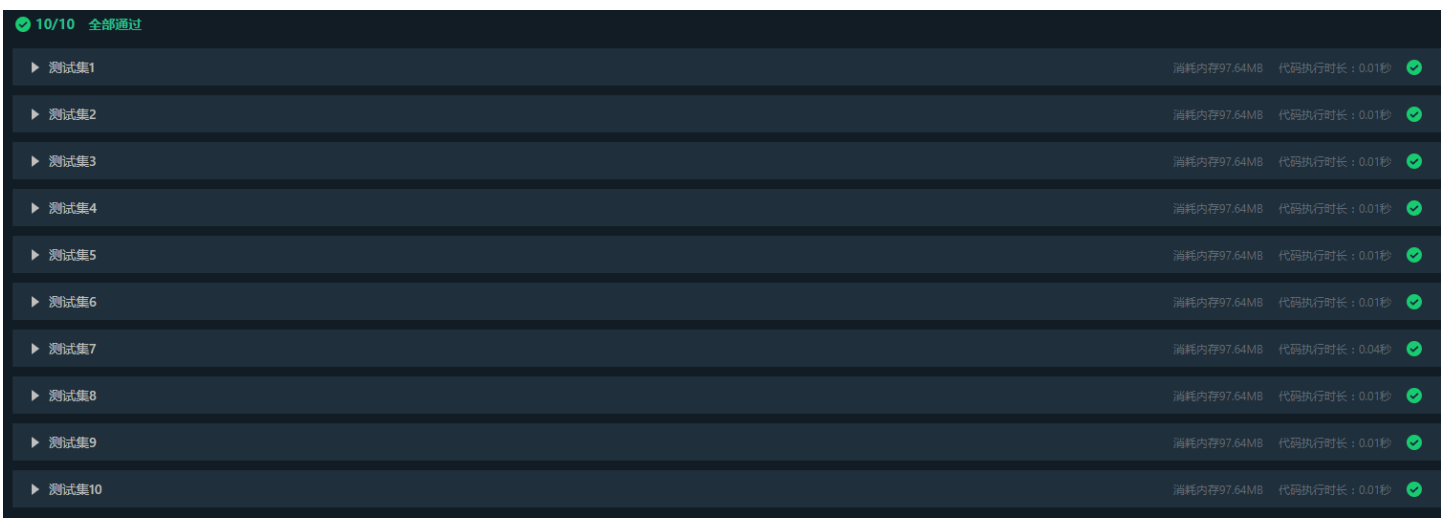


图17 实验三(1)通过截图

实验三(2)的要求与解析：

1.用户编程要求

为方便代码评判，本关只考虑第一次遇到错误就报错并停止分析的错误处理方式。
同学们可以下去自己尝试考虑加入对应的错误信息，以及如何进行错误恢复，以便继续分析完整的输入串来获取可能更多的错误信息。

编写一个语法分析程序，能对算数表达式进行LR(1)语法分析。

要求:
对给定文法的产生式进行如下编号后：

编号	产生式
0	E' -> E
1	E -> E+T
2	E -> E-T
3	E -> T
4	T -> T*F
5	T -> T/F
6	T -> F
7	F -> (E)
8	F -> num

1. 编程实现构造该文法的LR(1)分析表。
2. 编程实现算法4.3，构造LR(1)分析程序。

图18 实验三(2)编程要求

实验要求如图18，我们可以得知，题目希望我们实现一个基于LR(1)文法的预测分析表，以此来识别或能被接收或不被接收的符号串，输出识别过程的预测分析程序。

2.平台环境说明

编译器版本：gcc7.3.0
OS版本：Debian GNU/Linux 9

图19 实验三(2)环境说明

由图19我们可以得知，平台的实验环境为gcc7+Linux9的结合，也不采用非标准库，故正常编写C++语言程序即可。

3.LR(1)预测分析表

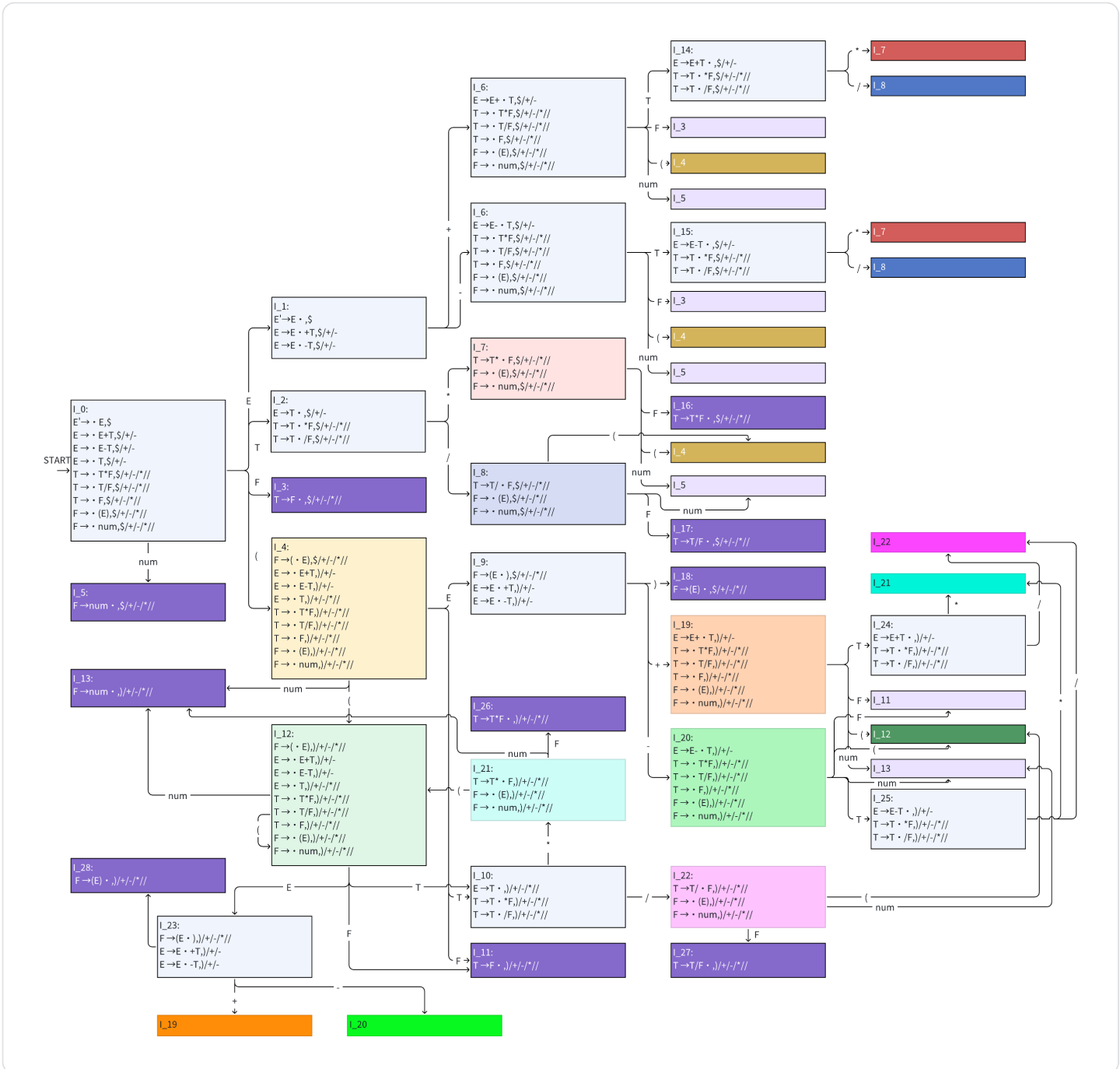


图20 实验三(2)LR(1)项目集规范族

针对给定文法，我们先按照有效项目集的构造方法进行初步构建，在每种活前缀串输入的情况下，针对每一个项目集进行分支构造，直到遇到归约项目集为止，则识别完成一个句柄，依此类推直到无法推出新的有效项目集，最终构造出如图20的LR(1)项目集规范族。

4.测试程序示例

1）输入格式

从标准输入（使用 cin/scanf 等输出）读入数据。

输入仅包含1行：

- 第1行输入为一个算术表达式。构成该算术表达式的字符有：{'n'， '+'， '-'， '*'， '/'， '('， ')'}。

2）输出格式

输出到标准输出（使用 cout/printf 等输出）中。

输出包括若干行LR(1)分析过程：

假设输出有n行，则第i行(1⩽i⩽n)表示分析进行到第i步，它的输出包含**制表符** `\t` **分隔**的三个部分：

- **分析动作**：归约使用的产生式编号 或 shift 或 error 或 accept(表示当前步骤应执行的动作)

注：规定第1步的分析过程为：根据分析栈中只有状态0、输入串栈顶为第一个输入字符，来产生分析动作。

4）简单测试样例：

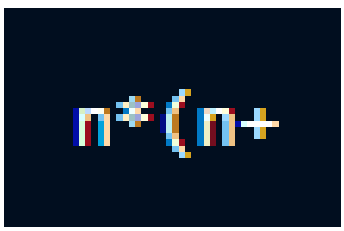


图21 实验三(2)样例输入

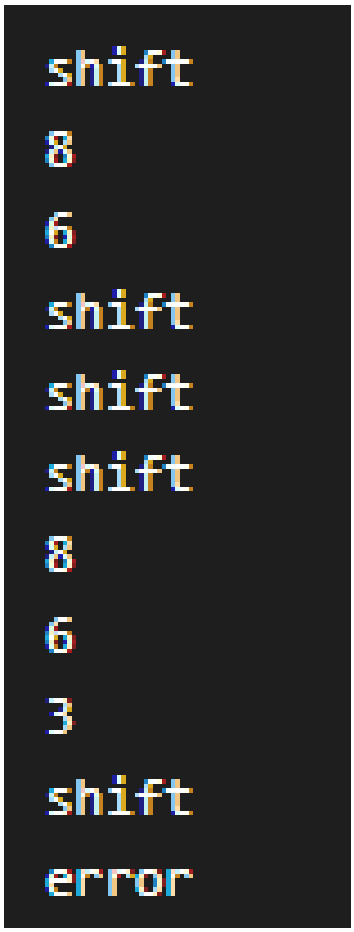


图22 实验三(2)样例输出

5）简单样例分析：

样例的输入中，包含一行或合法或不合法的符号串。

样例的输出中，则是程序的动作依据，或是归约所凭借的产生式编号，或是分析栈与输入串栈栈顶匹配，或是最终成功接收/无法接收而报错。

5.实验通过截图

源代码可见第四部分的最后一点

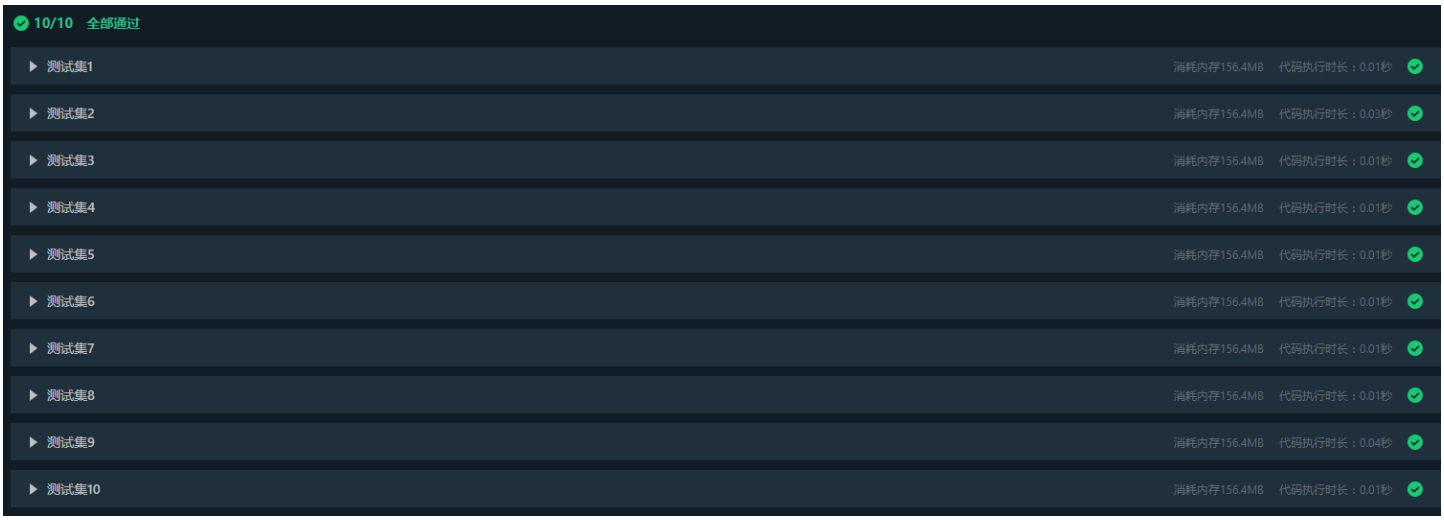


图23 实验三(2)通过截图

二、程序设计构思

> 💡 **目标一：完成面向特定文法的基本语法分析**
>
> **目标二：完成扩展功能，尝试更通用的识别算法**

1.题目解析

设计并实现给定文法的语法分析程序，要求如下。

1. 可以识别出给定文法，构建生成式对应的存储表。
2. 可以根据存储表构建出状态转换表（项目集规范族）。
3. 可以利用栈结合状态转换表处理读入的符号串。
4. 检查符号串是否存在错误，如果有错便停止处理返回error。
5. 如果没有错误，读到结束符号$后返回accept。

2.本地实验环境

- Microsoft Windows 11
- Microsoft Visual Studio Community 2022 (64 位) v17.8.3
- Microsoft .NET Framework v4.8.04161

3.模块设计

1）模块划分

LL(1)预测分析程序较为简单，直接使用main函数存储、处理实现，详情见源代码一节；设计细节见本章节。

LR(1)分析程序各模块及其关系如图24所示，其中绿色代表函数的入口，红色代表出口，紫色代表主要动作转换，黄色代表辅助函数：

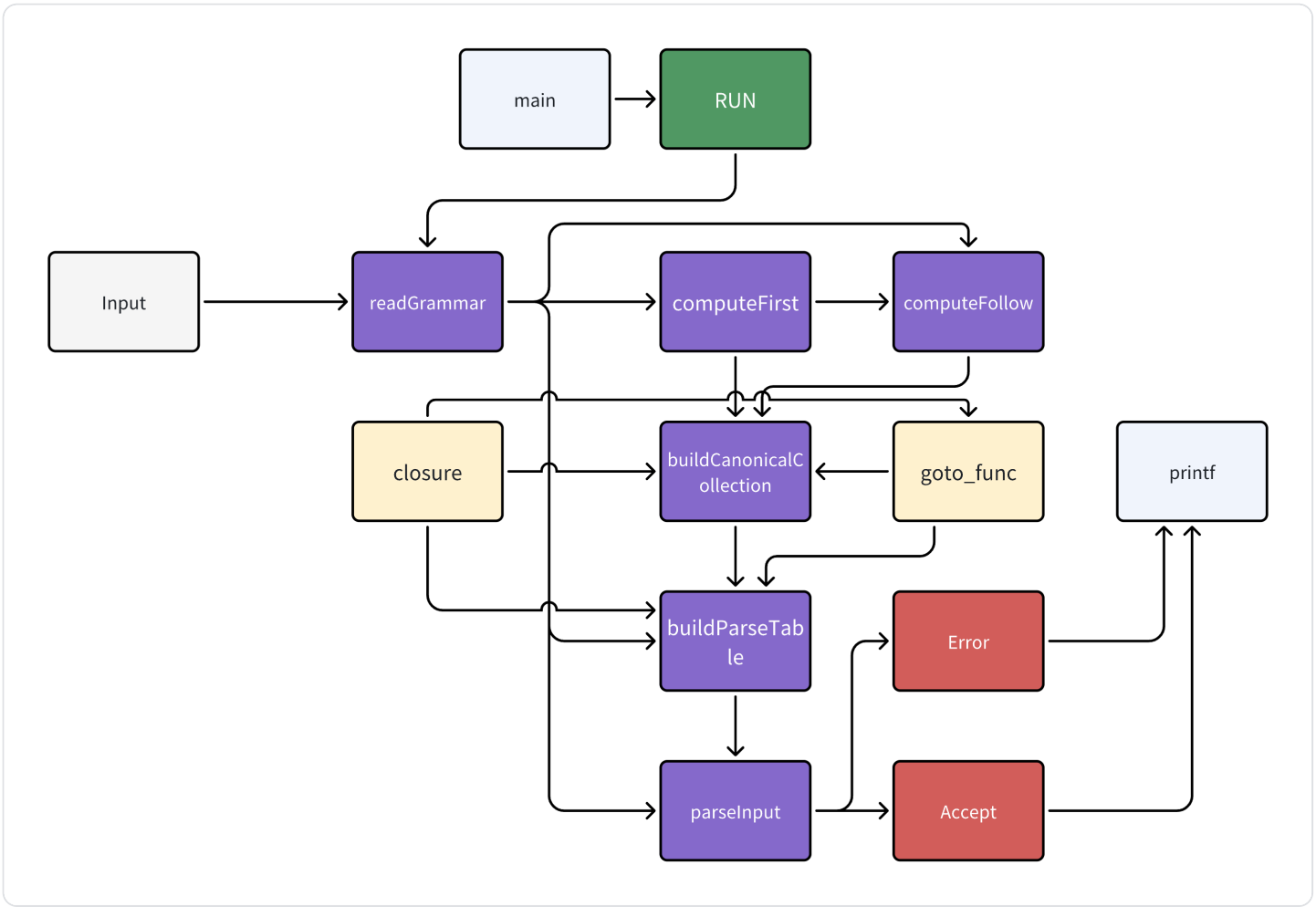


图24 LR(1)模块设计以及调用关系

2）输入关键字

1.StartSymbol，起始符号

第一行输入为起始符号

2.Non-terminals，非终结符

第二行输入，用户可以自行输入

```
{'E','T','A','F','B'}
```

3.Terminals，终结符

第三行输入，用户可以自行输入，其中ε为了方便读取与辅助函数识别，改为e

`{'n','+','-','*','/','(',')',e}`

4.ProductionAmounts，产生式数量

第四行输入，用户自行输入

5.ProfuctionList，产生式列表

第五行输入，可以输入[ProductionAmounts]行，用户自行输入

注意：可以输入带"|"产生式，但是LL(1)算法中需要消除左递归

6.PreString，待解析的输入字符串

最后一行输入，用户自行输入，为任意输入，不合法会报错

3）数据结构定义

1.文法结构定义

```
// LL(1)文法结构
方法 initializeGrammar():
        // 定义非终结符和终结符
        nonTerminals = { ?  }
        terminals = { ?  }

        // 定义开始符号
        startSymbol = ?

        // 定义产生式，编号从1开始
        添加产生式 productions 加入 Production(编号，左侧符号，右侧符号元组)
```

```
//LR(1)文法结构
结构体 Grammar:
    字符串 startSymbol          // 开始符号
    字符串列表 nonTerminals     // 非终结符集合
    字符串列表 terminals        // 终结符集合
    产生式列表 productions       // 产生式列表
```

2.生成式存储结构定义

```
// LL(1)生成式右端
结构体 Production:
```

```
    整数 id                  // 产生式编号
    字符串 lhs               // 左部非终结符
    字符串列表 rhs           // 右部符号串
```

```
// LR(1)生成式右端
结构体 Production:
    字符串 lhs               // 左部非终结符
    字符串列表 rhs           // 右部符号串
```

3.LR(1)项目集存储结构定义（或者说是ε-NFA表）

```
结构体 LR1Item:
    字符串 lhs
    字符串列表 rhs
    整数 dot                 // 点的位置
    字符串 lookahead         // 向前看符号

    方法 比较(other):
        返回 (lhs, rhs, dot, lookahead) 比较 other 的对应字段
```

4.解析表存储结构定义

```
//LL(1)解析表构造方法
方法 buildParseTable():
        对于 每个产生式 prod 在 productions:
            A = prod.lhs
            alpha = prod.rhs

            // 计算 First(α)
            // 对于 a ∈ First(α) - {e}, M[A,a] = prod.id
            // 如果 e ∈ First(α), 对于 b ∈ Follow(A), M[A,b] = prod.id
```

```
//LR(1)解析表构造方法
函数 BuildParseTable(Grammar grammar, 映射<string, 集合<string>> FIRST, 映射<string, 集合<string>> FOLLOW) -> 映射<string, 映射<string, 整数>>:
    初始化 映射<string, 映射<string, 整数>> parseTable

    对于 每个产生式 prod 在 grammar.productions:
        A = prod.lhs
```

```
        alpha = prod.rhs

        // 计算 First(α)
        // 对于 a ∈ First(α) - {e}, M[A,a] = prod.id
        // 如果 e ∈ First(α), 对于 b ∈ Follow(A), M[A,b] = prod.id

    返回 parseTable
```

4.扩展识别伪代码

LL(1)识别

```
// 定义 LL(1) 解析器类
类 LL1Parser:
    // 成员变量
    产生式列表 productions
    非终结符集合 nonTerminals
    终结符集合 terminals
    字符串 startSymbol

    // FIRST 和 FOLLOW 集合
    映射 First
    映射 Follow

    // 解析表：非终结符 -> (终结符 -> 产生式编号)
    映射 parseTable

    // 初始化文法
    方法 initializeGrammar():
        // 定义非终结符和终结符
        nonTerminals = { ? }
        terminals = { ? }

        // 定义开始符号
        startSymbol = ?

        // 定义产生式，编号从1开始
        添加产生式 productions 加入 Production(编号, 左侧符号, 右侧符号元组)

    // 计算 FIRST 集合
    方法 computeFirst():
        // 初始化终结符的 FIRST 集合
        对于 每个终结符 t 在 terminals:
```

```
            如果 t ≠ "e":
                FIRST[t] 加入 t

        // 'e' 的 FIRST 集合

        // 初始化非终结符的 FIRST 集合为空
        对于 每个非终结符 nt 在 nonTerminals:
            FIRST[nt] = 空集合

        // 迭代直到 FIRST 集合不再变化
        循环直到 no changes

    // 计算 FOLLOW 集合
    方法 computeFollow():
        // 初始化 FOLLOW 集合为空
        对于 每个非终结符 nt 在 nonTerminals:
            FOLLOW[nt] = 空集合

        // 开始符号的 FOLLOW 集加入 $

        // 迭代直到 FOLLOW 集合不再变化
        循环直到 no changes

    // 构造解析表
    方法 buildParseTable():
        对于 每个产生式 prod 在 productions:
            A = prod.lhs
            alpha = prod.rhs

            // 计算 First(α)
            // 对于 a ∈ First(α) - {e}, M[A,a] = prod.id
            // 如果 e ∈ First(α), 对于 b ∈ Follow(A), M[A,b] = prod.id

    // 解析输入字符串
    方法 parseInput():
        // 读取待解析的输入字符串
        // 初始化解析栈
        // 解析循环
        循环直到 parseStack 为空:

        // 检查是否接受

    // 运行解析器
    方法 run():
        调用 initializeGrammar()
        调用 computeFirst()
        调用 computeFollow()
```

```
        调用 buildParseTable()
        调用 parseInput()

    // 辅助函数：判断是否是终结符
    方法 isTerminal(sym):
        返回 sym 在 terminals 中

    // 辅助函数：计算一串符号的 First 集合
    方法 computeFirstOfString(symbols):

        返回 result

    // 辅助函数：分词函数，将输入字符串转化为终结符序列
    方法 tokenize(input):

        返回 tokens

// 主程序
    创建 LL1Parser 实例 parser
    调用 parser.run()
```

LR(1)识别

```
// 辅助函数：分割字符串
函数 Split(string s, 字符 char delim) -> 字符串列表:
    初始化 空的字符串列表 tokens
    初始化 空的字符串 token
    创建字符串流 ss 从 s
    当 从 ss 获取一行到 token 时:
        如果 token 不为空:
            添加 token 到 tokens
    返回 tokens

// 辅助函数：去除字符串首尾空白
函数 Trim(string s) -> string:
    去除 s 的左侧空白字符
    去除 s 的右侧空白字符
    返回 去除后的字符串

// 读取文法
函数 ReadGrammar() -> Grammar:

    返回 grammar

// 计算 FIRST 集合
```

```
函数 ComputeFirst(Grammar grammar) -> 映射<string, 集合<string>>:
    初始化 映射<string, 集合<string>> FIRST

    // 初始化终结符的 FIRST 集合
    对于 每个终结符 t 在 grammar.terminals:
        如果 t ≠ "e":
            将 t 添加到 FIRST[t]

    // 初始化非终结符的 FIRST 集合为空
    对于 每个非终结符 nt 在 grammar.nonTerminals:
        FIRST[nt] = 空集合

    当 changed 为 true:
        设置 changed 为 false
        对于 每个产生式 prod 在 grammar.productions:
            A = prod.lhs
            alpha = prod.rhs

            // 计算 First(α)
            // 将 First(α) - {e} 加入 First(A)
            // 如果 e ∈ First(α), 将 e 加入 First(A)

    返回 FIRST

// 计算一串符号的 FIRST 集合
函数 ComputeFirstOfString(字符串列表 symbols, 映射<string, 集合<string>> FIRST) -
> 集合<string>:

    返回 result

// 计算 FOLLOW 集合
函数 ComputeFollow(Grammar grammar, 映射<string, 集合<string>> FIRST) -> 映射
<string, 集合<string>>:
    初始化 映射<string, 集合<string>> FOLLOW
    // 初始化所有非终结符的 FOLLOW 集合为空
    // 将开始符号的 FOLLOW 集合加入 $


    当 changed 为 true:
        设置 changed 为 false
        对于 每个产生式 prod 在 grammar.productions:
            A = prod.lhs
            alpha = prod.rhs
            对于 每个索引 j 从 0 到 alpha.size() - 1:
                B = alpha[j]
                如果 B 是非终结符:
                    beta = alpha 从 j+1 到结束的子串
```

```
                    如果 beta 为空:
                        // 将 FOLLOW(A) 加入 FOLLOW(B)

                    否则:
                        // 计算 First(β)
                        // 将 First(β) - {e} 加入 FOLLOW(B)
                        // 如果 e ∈ First(β)，将 FOLLOW(A) 加入 FOLLOW(B)
    返回 FOLLOW

// 构造解析表
函数 BuildParseTable(Grammar grammar, 映射<string, 集合<string>> FIRST, 映射<string, 集合<string>> FOLLOW) -> 映射<string, 映射<string, 整数>>:
    初始化 映射<string, 映射<string, 整数>> parseTable

    对于 每个产生式 prod 在 grammar.productions:
        A = prod.lhs
        alpha = prod.rhs

        // 计算 First(α)
        // 对于 a ∈ First(α) - {e}, M[A,a] = prod.id
        // 如果 e ∈ First(α), 对于 b ∈ Follow(A), M[A,b] = prod.id

    返回 parseTable

// 解析输入字符串
函数 ParseInput(Grammar grammar, 映射<string, 映射<string, 整数>> parseTable, string inputStr):
    // 分词：将输入字符串拆分为终结符列表
    // 初始化解析栈

    当 parseStack 不为空:
        // 构造堆栈字符串
        // 构造剩余输入字符串
        // 获取栈顶符号

        // 检查是否接受

    如果 accept 为 false:
        输出 "Parsing failed."

// 主程序
    // 读取文法
    // 计算 FIRST 和 FOLLOW 集合
    // 构造解析表
    // 读取待解析的输入字符串
    读取一行到字符串 inputStr
    调用 ParseInput(grammar, parseTable, inputStr)
```

5.扩展识别函数设计

1）读入函数

功能：读取并初始化文法规则，包括开始符号、非终结符、终结符以及产生式。

设计思路：

- **输入开始符号**：用户首先输入文法的开始符号。
- **输入非终结符和终结符**：通过读取用户输入的字符串，将非终结符和终结符存储在相应的集合中。
- **输入产生式**：用户指定产生式的数量后，逐行输入每个产生式。每个产生式可能包含多个右部，通过 `|` 分隔。
- **处理空串**：将表示空串的符号（如 `e` 或 `ε`）特别处理，以便在后续计算中正确识别和应用。

2）FIRST集合计算函数

功能：计算文法中每个符号的 FIRST 集合。

设计思路：

- **初始化**：为所有终结符设置其自身为 FIRST 集合的唯一元素。非终结符的 FIRST 集合初始化为空。
- **迭代更新**：通过迭代所有产生式，逐步添加符号的 FIRST 集合，直到没有更多变化为止。
- **处理空串**：如果一个产生式的右部可以推导出空串（`e`），则将 `e` 添加到相应非终结符的 FIRST 集合中。
- **确保终结符和非终结符的区分**：通过不同的处理方式确保终结符和非终结符的 FIRST 集合正确计算。

3）FOLLOW集合计算函数

功能：计算文法中每个非终结符的 FOLLOW 集合。

设计思路：

- **初始化**：所有非终结符的 FOLLOW 集合初始化为空，开始符号的 FOLLOW 集合包含结束符 `$`。
- **迭代更新**：通过遍历所有产生式，依据文法规则（如 $A \rightarrow \alpha B \beta$）的要求，逐步添加符号的 FOLLOW 集合，直到不再有变化。
- **结合 FIRST 集合**：利用已计算的 FIRST 集合来确定 FOLLOW 集合中的元素，特别是在处理右部存在多个符号的情况时。

4）LL(1)解析表构建函数

功能：基于计算得到的 FIRST 和 FOLLOW 集合，构建 LL(1) 解析表。

设计思路：

- **填充解析表**：对于每个产生式 A → α，遍历 FIRST(α) 中的每个终结符 a，将产生式编号填入解析表的 M[A,a] 位置。
- **处理空串**：如果 `e` 存在于 FIRST(α) 中，则将 FOLLOW(A) 中的每个终结符 b 也填入解析表的 M[A,b] 位置。
- **冲突检测**：在填充解析表时，检测同一表项是否被多个产生式填充，以确保文法是 LL(1) 的。如果发现冲突，程序会报告错误并终止，提示文法不适合 LL(1) 解析器。

5）LL(1)解析动作输出函数

功能：根据构建的解析表，解析用户输入的字符串，并输出解析过程的每一步动作。

设计思路：

- **分词**：将输入字符串拆分为终结符序列，并在末尾添加结束符 `$`。
- **初始化栈**：解析栈初始化为 `["$", startSymbol]`。
- **解析循环**：
  - **栈顶符号与当前输入符号比较**：
    - **终结符匹配**：如果栈顶符号是终结符且与当前输入符号匹配，执行 `match` 操作，弹出栈顶符号并移动输入指针。
    - **非终结符替换**：如果栈顶符号是非终结符，查找解析表 M[A,a] 中对应的产生式，应用产生式（弹出非终结符并将产生式右部逆序压入栈）。
  - **接受与错误**：当栈和输入均为 `$` 时，接受输入。否则，如果无法匹配或应用产生式，输出 `error` 并终止解析。
- **输出格式**：每一步输出当前栈内容、剩余输入和所执行的动作，格式为 `Stack\tInput\tAction`。

6）给定项集计算函数

功能：计算给定项集的闭包，生成新的项集。

设计思路：

- **初始项集**：开始时，闭包包括初始项集 `I`。

- **扩展项集**：对于闭包中每个项目，如果点（`.`）前面是非终结符，则根据其产生式添加新的项目，考虑向前看符号。
- **迭代**：持续添加新的项目，直到闭包不再变化，确保所有可能的项都被包含。

7）项集推理函数

功能：计算在给定项集 `I` 上迁移符号 `X` 后的新项集。

设计思路：

- **移动点**：对于项集 `I` 中每个项目，如果点前面是符号 `X`，则将点向右移动一位，生成新的项目。
- **闭包**：对移动后的项目集应用闭包操作，确保所有相关项目都被包含。

8）项目集规范族构建函数

- **功能**：构建 LR(1) 项集规范族，即所有可能的项集。
- **设计思路**：
  - **初始项集**：创建初始项集 `C0`，包括扩展文法的开始产生式和向前看符号 `$`。
  - **广度优先搜索（BFS）**：使用 BFS 遍历所有可能的迁移，生成并编号所有项集。
  - **记录迁移**：在遍历过程中，记录每个符号的迁移状态，填充 Action 和 Goto 表。

9）LR(1)解析表构建函数

功能：基于构建的项目集规范族，生成 Action 和 Goto 表，用于解析过程中的决策。

设计思路：

- **遍历项集**：对于每个项集，检查其中的项目类型（移进、归约或接受）。
- **填充 Action 表**：
  - **移进动作**：如果项目的点前面是终结符，则对应的 Action 表填入 `shift` 操作和目标状态。
  - **归约动作**：如果项目的点已到达右端，则根据向前看符号填入 `reduce` 操作和产生式编号。
  - **接受动作**：如果项目是扩展文法的开始产生式且点已到达右端，填入 `accept` 操作。
- **填充 Goto 表**：对于非终结符的迁移，记录相应的状态转移。
- **冲突检测**：在填充过程中，检测 Action 表是否存在冲突（同一表项有多种操作），确保文法是 LR(1) 的。

10）LR(1)解析动作输出函数

功能：根据生成的 Action 和 Goto 表，解析用户输入的字符串，并输出解析过程的动作序列。

设计思路：

- **分词**：将输入字符串转换为终结符序列，并在末尾添加结束符 `$`。
- **初始化栈**：解析栈初始化为 `[0]`，表示初始状态。
- **解析循环**：
  - **当前状态和输入符号**：获取栈顶状态和当前输入符号。
  - **查找 Action 表**：根据当前状态和输入符号，查找对应的操作（shift、reduce 或 accept）。
  - **执行操作**：
    - **Shift**：将目标状态压入栈，移动输入指针，记录 `shift` 动作。
    - **Reduce**：根据产生式编号，弹出相应数量的栈元素，查找 Goto 表，压入新的状态，记录归约动作。
    - **Accept**：记录 `accept` 动作，解析成功结束。
  - **错误处理**：如果 Action 表中不存在对应的操作，记录 `error` 动作并终止解析。
- **输出格式**：每一步输出执行的动作，按照指定的格式（如 `shift`,产生式编号,`accept`），每个动作占一行。

6.扩展语法分析器输出

1）LL(1)输出

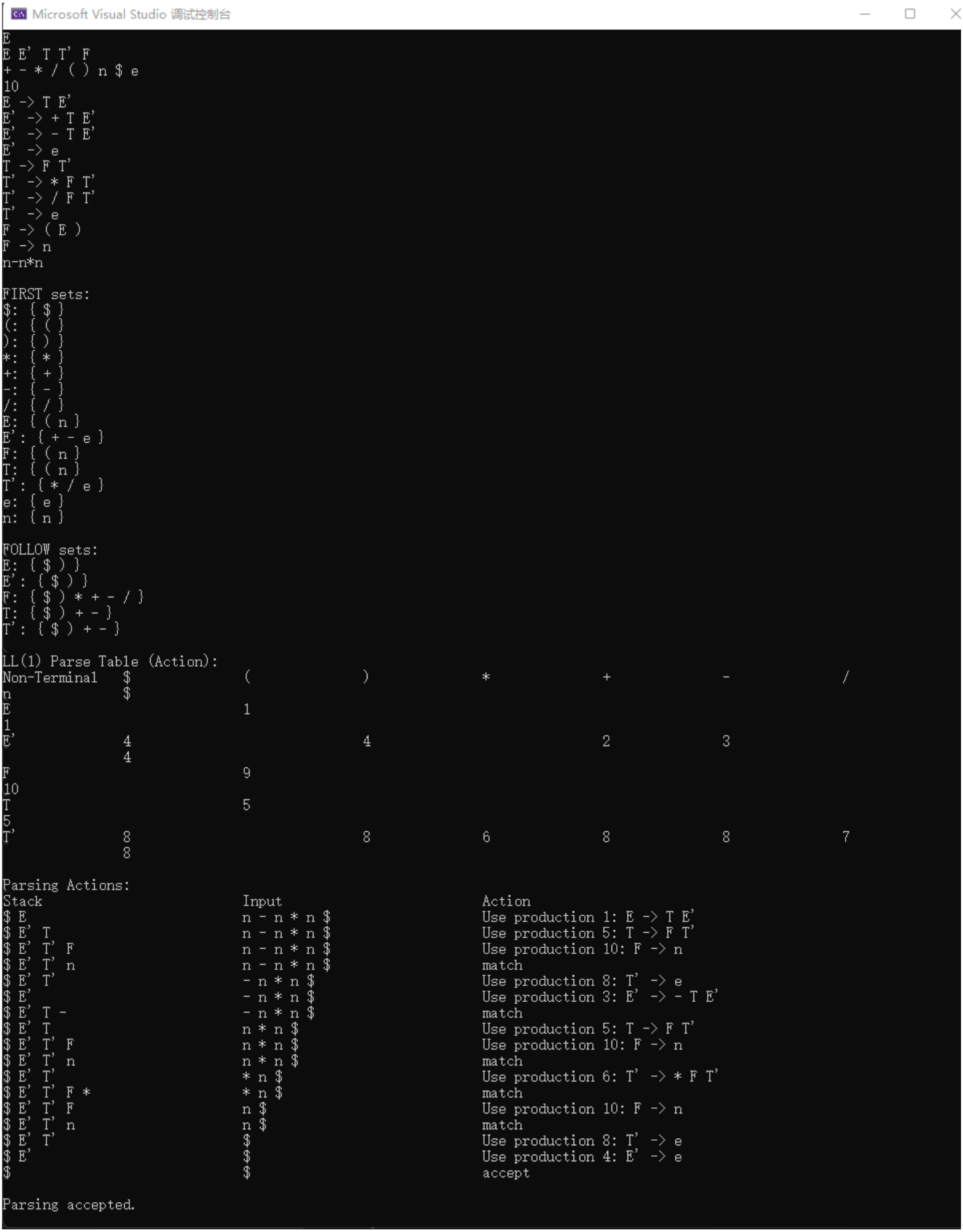


图25 LL(1)针对输入"n-n*n"输出的预测分析表与预测分析步骤

2）LR(1)输出

E' E T F
+ - * / ( ) n $ e
9
E' -> E
E -> E + T
E -> E - T
E -> T
T -> T * F
T -> T / F
T -> F
F -> (E)
F -> n
n-n*n

Action Table:

State	$	(	)	*	+	-	/	n	$
0		shift 1						shift 5	
1		shift 6						shift 10	
2	accept				shift 11		shift 12		accept
3	reduce 7				reduce 7		reduce 7	reduce 7	reduce 7
4	reduce 4				shift 13		reduce 4	reduce 4	shift 14
5	reduce 9				reduce 9		reduce 9	reduce 9	reduce 9
6		shift 6						shift 10	
7			shift 16			shift 17		shift 18	
8			reduce 7		reduce 7		reduce 7	reduce 7	
9			reduce 4		shift 19		reduce 4	reduce 4	shift 20
10			reduce 9		reduce 9		reduce 9	reduce 9	
11		shift 1						shift 5	
12		shift 1						shift 5	
13		shift 1						shift 5	
14		shift 1						shift 5	
15			shift 25			shift 17		shift 18	
16	reduce 8				reduce 8		reduce 8	reduce 8	reduce 8
17		shift 6						shift 10	
18		shift 6						shift 10	
19		shift 6						shift 10	
20		shift 6						shift 10	
21	reduce 2				shift 13		reduce 2	reduce 2	shift 14
22	reduce 3				shift 13		reduce 3	reduce 3	shift 14
23	reduce 5				reduce 5		reduce 5	reduce 5	reduce 5
24	reduce 6				reduce 6		reduce 6	reduce 6	reduce 6
25			reduce 8		reduce 8		reduce 8	reduce 8	
26			reduce 2		shift 19		reduce 2	reduce 2	shift 20
27			reduce 3		shift 19		reduce 3	reduce 3	shift 20
28			reduce 5		reduce 5		reduce 5	reduce 5	reduce 5
29			reduce 6		reduce 6		reduce 6	reduce 6	reduce 6

Goto Table:

State	E	E'	F	T
0	2		3	4
1	7		8	9
2				
3				
4				
5				
6	15		8	9
7				
8				
9				
10				
11			3	21
12			3	22
13			23	
14			24	
15				
16				
17			8	26
18			8	27
19			28	
20			29	
21				
22				
23				
24				
25				
26				
27				
28				
29				

Parsing Actions:
shift
8
6
3
shift
shift
8
6

图26 LR(1)针对输入"n-n*n"输出的预测分析表与预测分析步骤

三、词法分析实验总结

在本次实验中，我亲自编写了基于LL(1)和LR(1)文法的两个C++语法分析程序。这次实践让我对语法分析的基本流程以及LL(1)与LR(1)两种不同解析方法有了更深入的理解。同时，我也加深了对预测分析表构建、FIRST和FOLLOW集合计算以及项目集规范族等关键知识点的掌握。

在代码实现过程中，我遇到了不少需要细心处理的细节问题。刚开始编写LL(1)和LR(1)解析器时，使用简单的数组结构来记录预测分析表导致数据管理变得混乱，特别是在处理Action和Goto表时，记录方式不够合理，甚至出现了内存泄漏的问题。通过多次调试和优化，我最终决定采用C++的STL容器（如 `map` 和 `set` ）来作为主要的数据存储结构。这不仅简化了数据的管理，还通过建立动作表的映射关系，成功解决了之前遇到的所有bug。

在LL(1)解析器的实现过程中，算法设计是重点，尤其是FIRST和FOLLOW集合的计算部分。虽然理论上LL(1)解析器需要消除左递归，但由于时间和复杂度的限制，我在此次实验中并未实现左递归的消除。因此，在使用该解析器时，需要确保输入的文法本身不包含左递归。这一限制提醒我，未来的工作中需要进一步完善解析器，以支持更广泛的文法形式。

相比之下，LR(1)解析器的实现更注重数据结构的设计和算法的准确实现，特别是在处理LR(1)项目和项目集时，必须选择合适的存储结构以保证计算的准确性和效率。借助C++11的语法特性，如自动类型推断（ `auto` ）和基于范围的for循环，我大大简化了代码，提高了程序的可读性和可维护性。这些现代C++特性有效减少了编程复杂度，使得算法流程更加清晰明了

然而，我也意识到程序中仍存在许多需要改进的地方。例如，错误处理机制尚不够全面，无法覆盖所有可能的解析错误情况；在处理较大规模输入时，DFA（确定性有限自动机）的求解过程可能不够稳定，容易导致程序中断；LR(1)解析器中的状态机实现较为复杂，状态转换和产生式规则的硬编码方式可能降低了代码的可读性和可维护性。由于实验时间的限制，这些问题在本次实验中尚未得到充分解决。

总体而言，本次实验不仅加深了我对语法分析相关知识的理解，还显著提升了我的C++编程技能。在实现LL(1)和LR(1)解析器的过程中，我学会了如何选择和优化数据结构，如何将理论知识应用于实际代码中，以及如何通过调试和测试不断改进程序的性能和稳定性。这些收获将为我未来在编译原理和相关领域的学习与研究提供宝贵的经验和技能支持。

四、（附录）测试报告

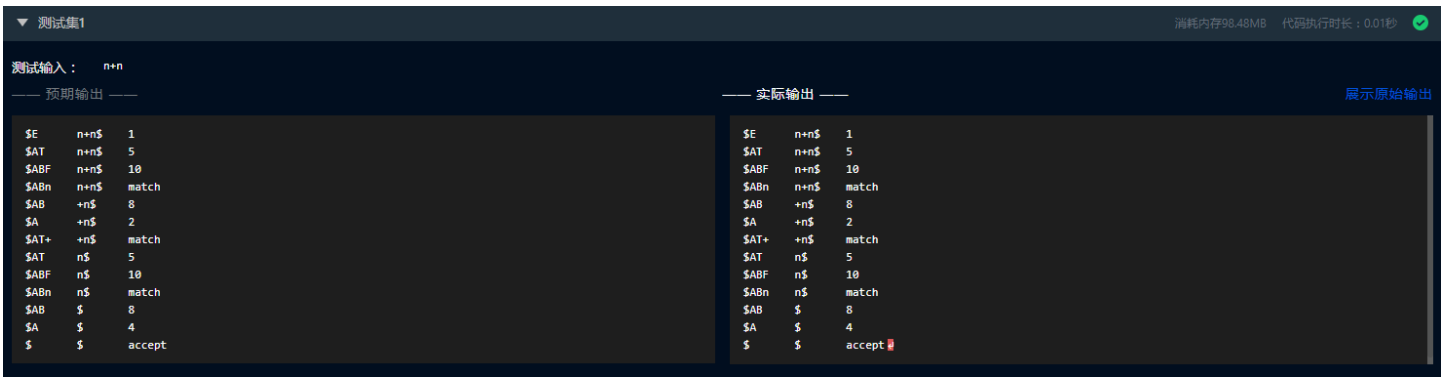
此部分为实验详情，属于实验报告的一部分。

在这里，笔者扒出样例输入与输出，并逐一在本地测试，出错则写明错误原因和修改思路，最终提交。

LL(1)识别

1.合法的加法表达式

1.运行结果与截图

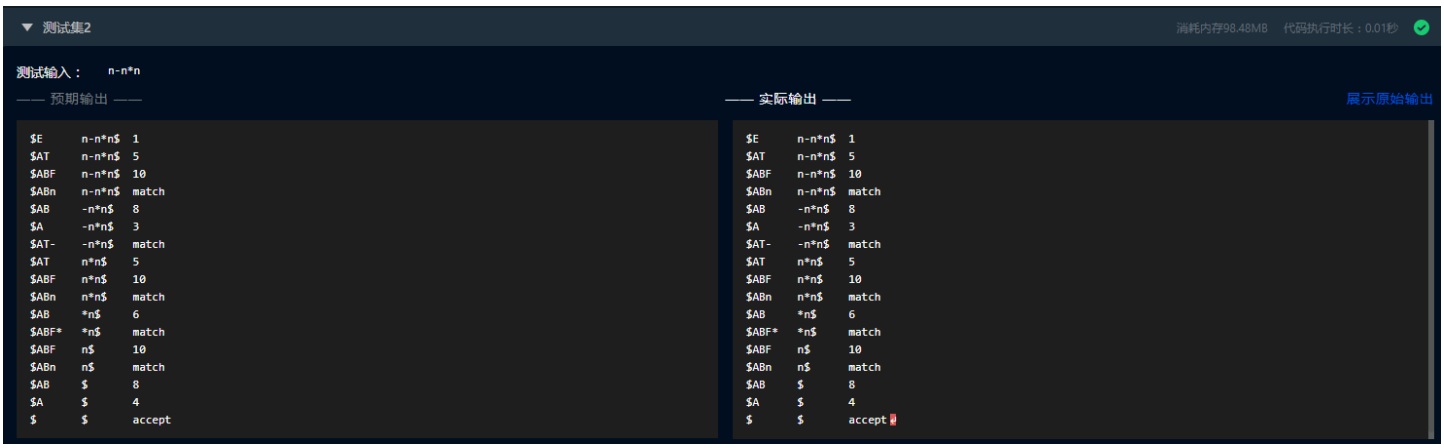


2.分析说明

加法表达式完全可以识别

2.合法的减法乘法表达式

1.运行结果与截图

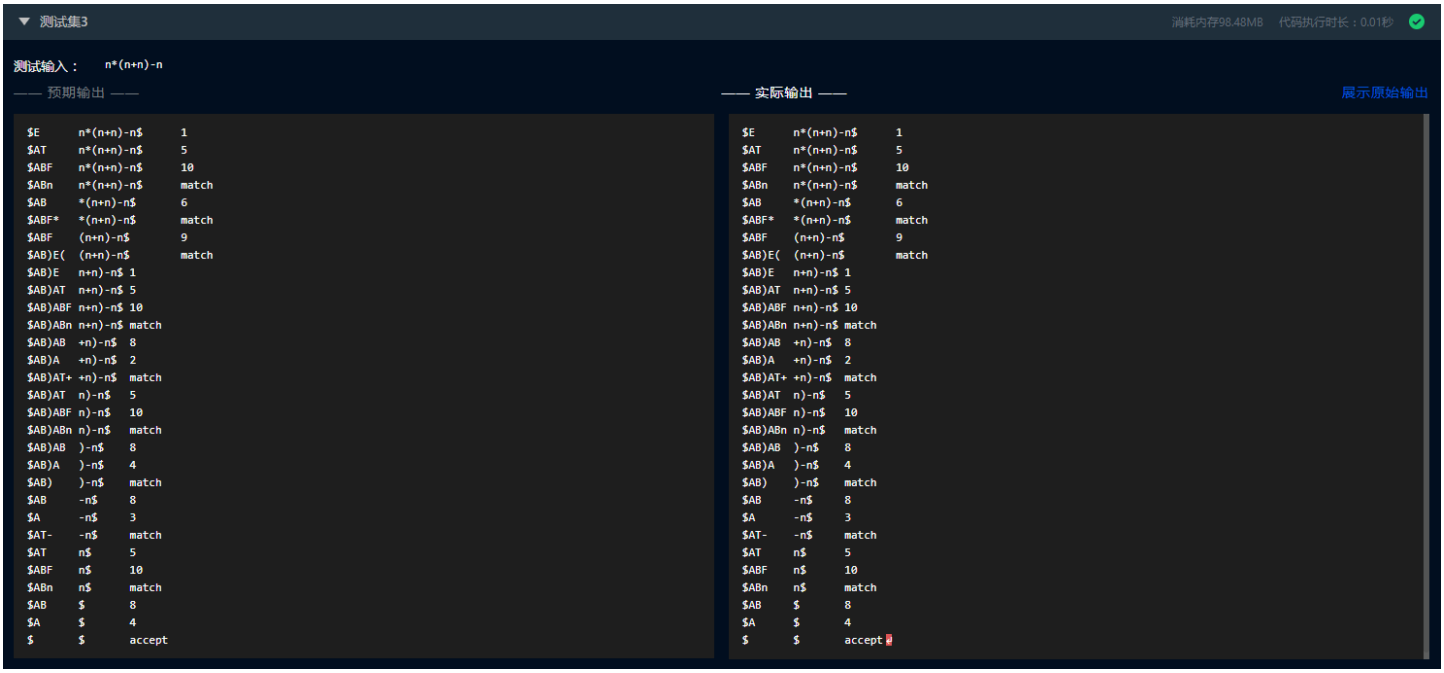


2.分析说明

不同优先级的运算完全可以识别

3.合法的带括号乘法表达式

1.运行结果与截图

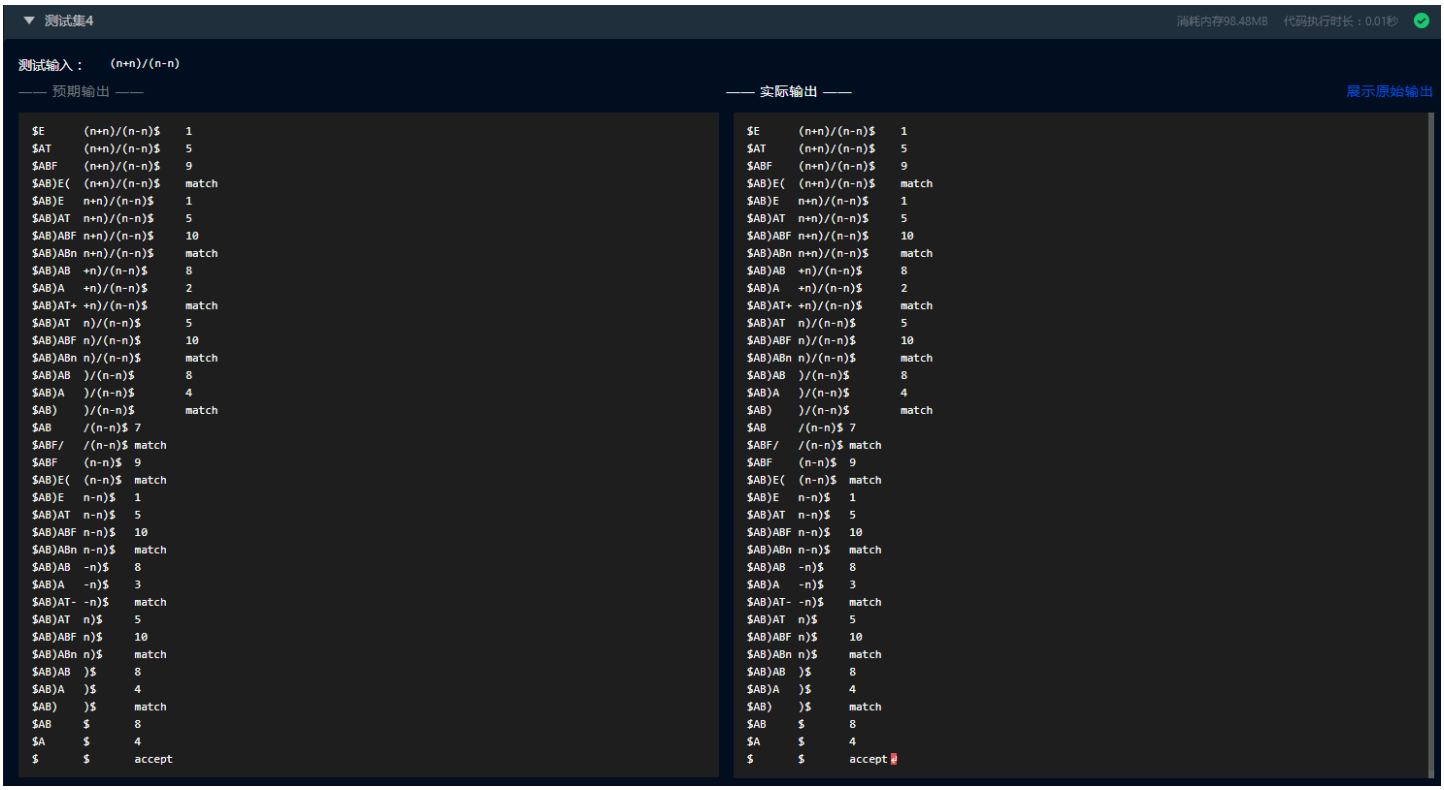


2.分析说明

低优先级运算加入括号成为优先级最高的运算，完全可以识别

4.合法的带括号除法表达式

1.运行结果与截图

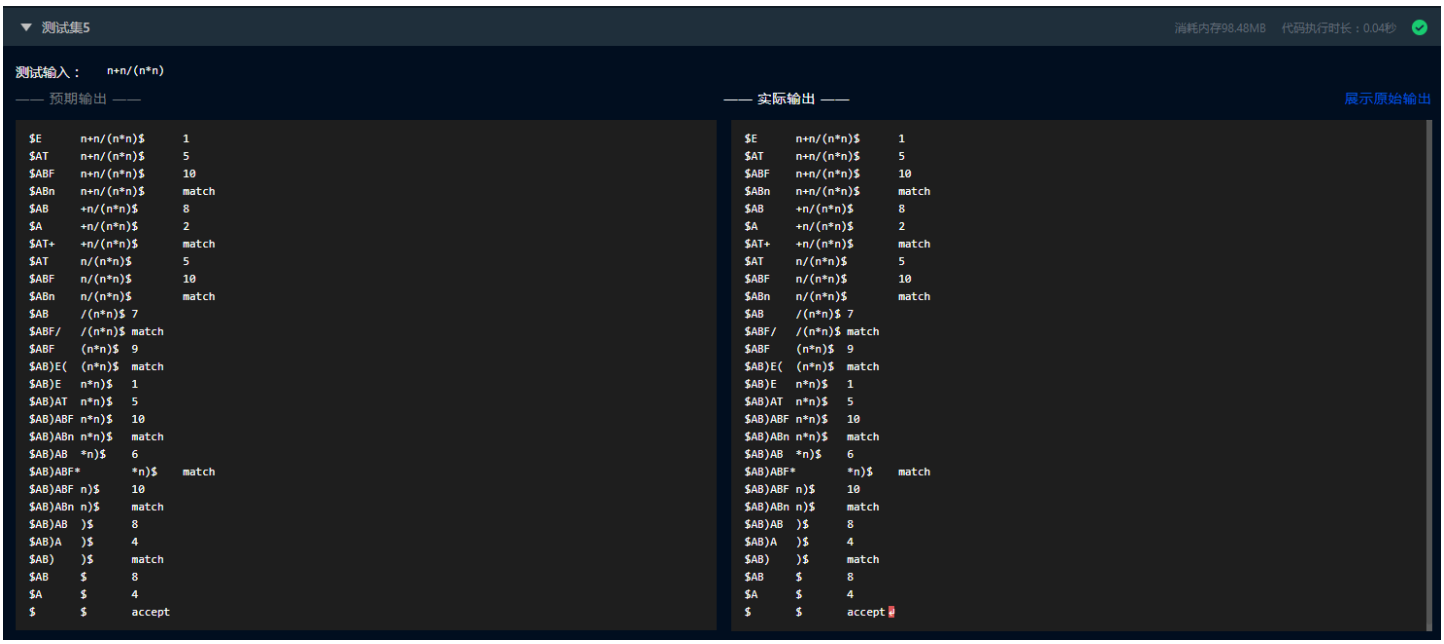


2.分析说明

除法运算完全可以识别

5.较为复杂的合法运算

1.运行结果与截图

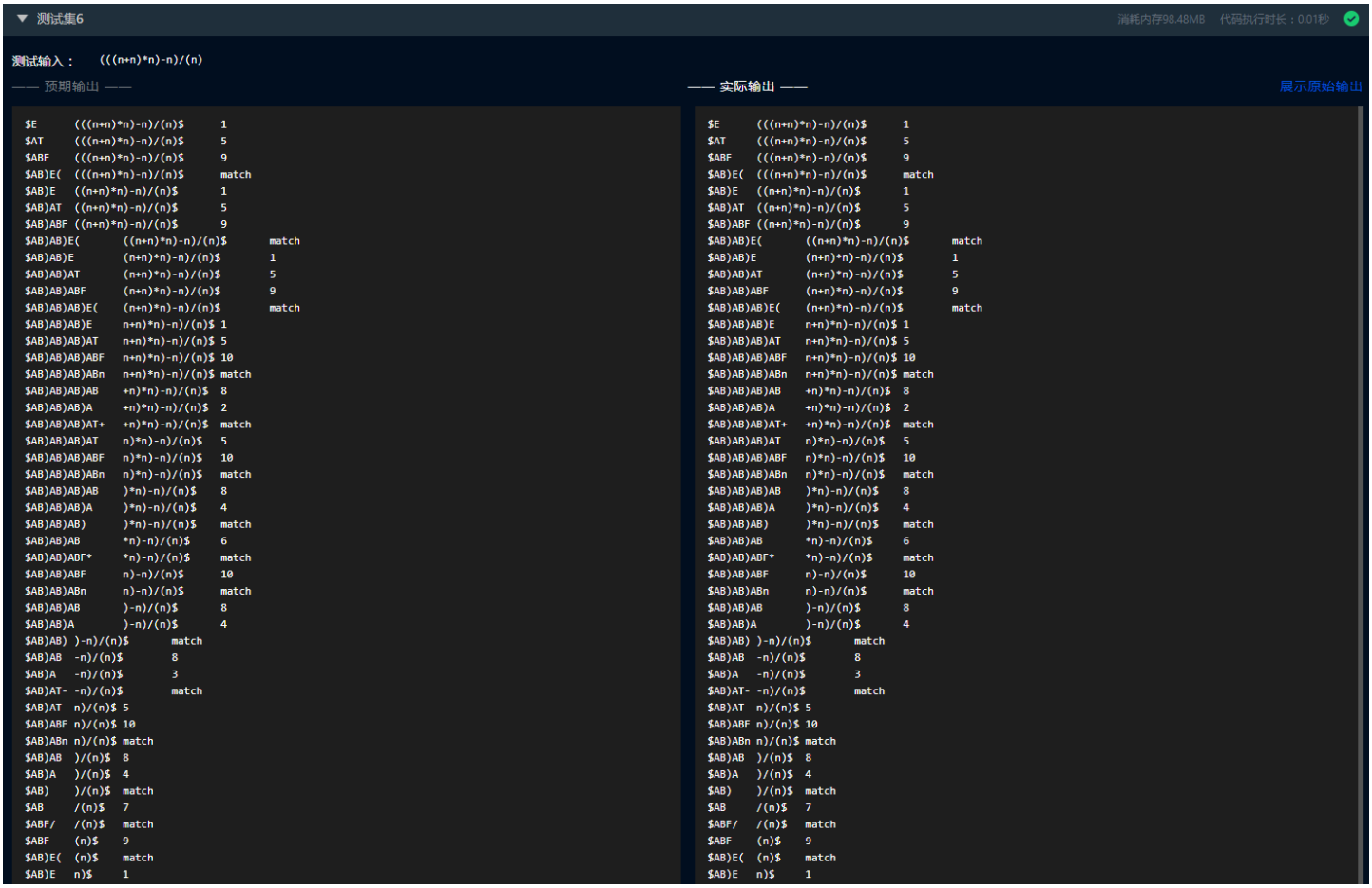


2.分析说明

较为复杂的运算完全可以识别

6.复杂的合法四则运算

1.运行结果与截图

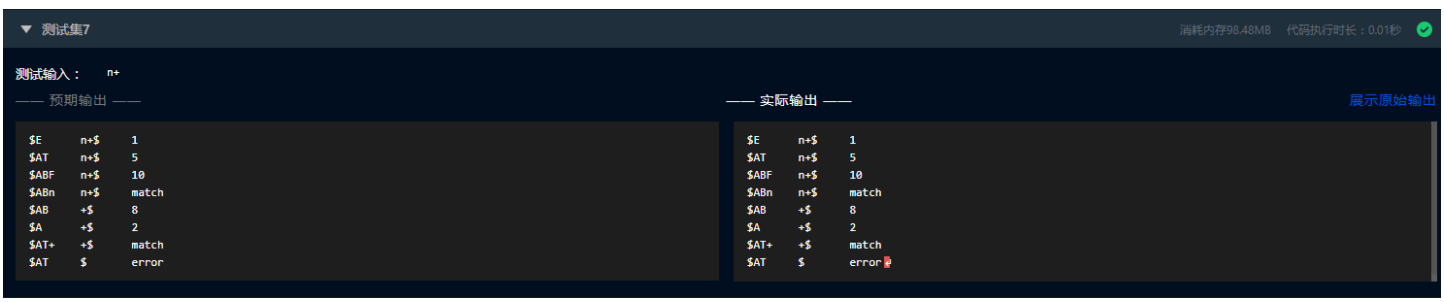


2.分析说明

复杂的四则运算以及其中的优先级完全可以识别

7.不合法的双目运算符错误

1.运行结果与截图

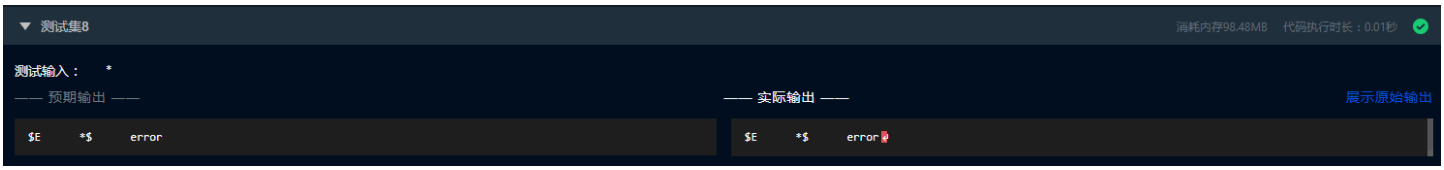


2.分析说明

双目运算符未闭合完全可以识别

8.不合法的双目运算符错误

1.运行结果与截图



2.分析说明

即使只有一个双目运算符也是错误的，可以识别

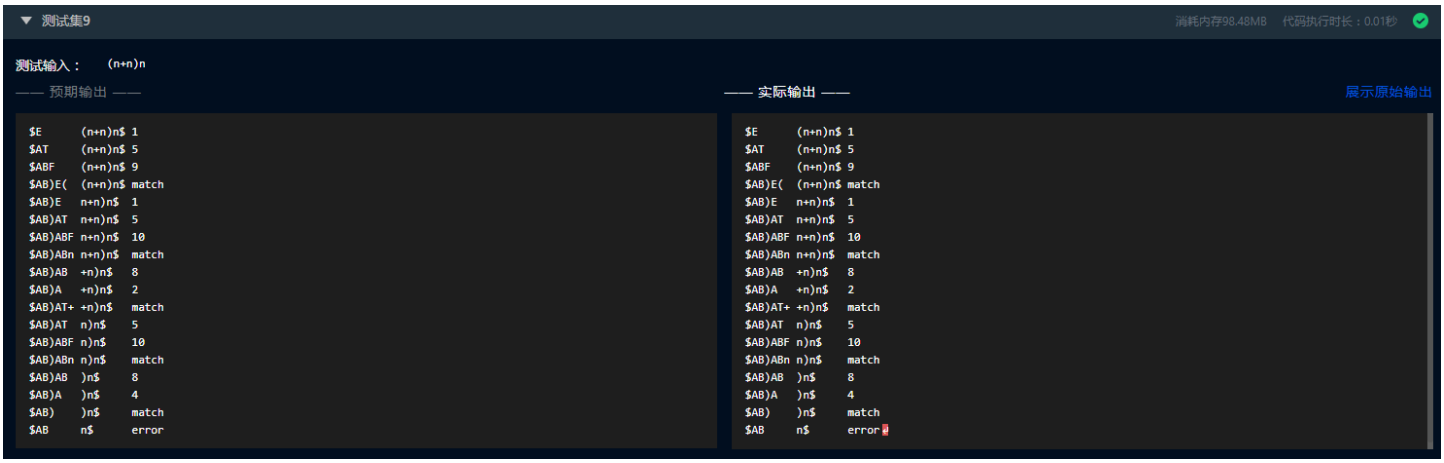
9.不合法的乘法省略错误

此题目曾出错

在最开始，我认为乘法省略是可以被允许的，在设计产生式识别的时候还刻意把这个问题优化掉了。

原来的方案是可以在任意地方插入一个 `"*"` 号，让算法更加灵活。在多次修改之后，询问gpt4o模型，获悉原来在产生式的位置就规定无法识别乘法省略的可能，没看题了属于是。

1.运行结果与截图

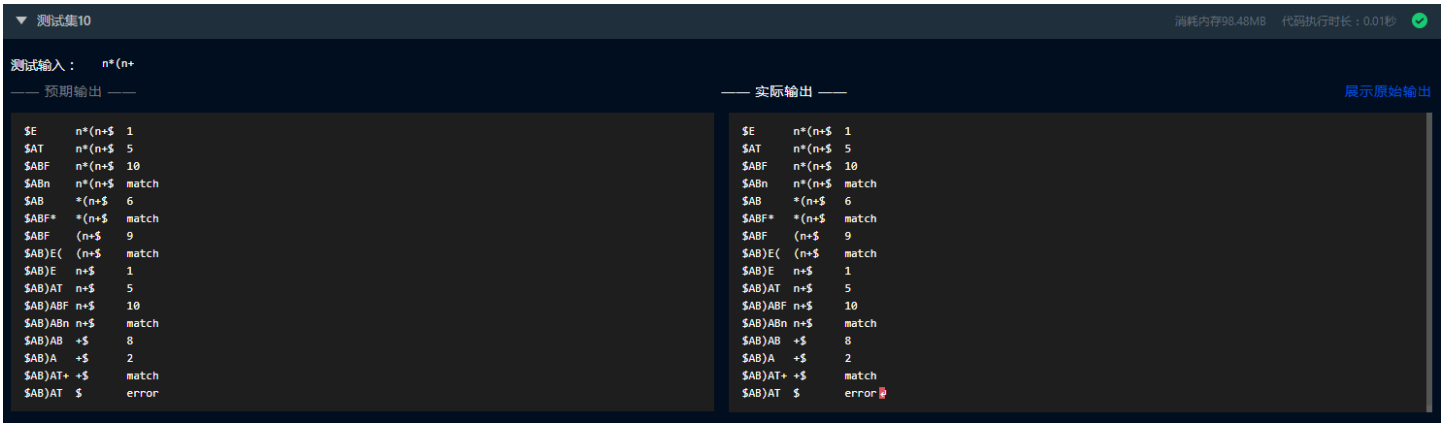


2.分析说明

乘法符号缺失完全可以识别

10.不合法的未闭合括号与双目运算符错误

1.运行结果与截图



2.分析说明

未闭合括号以及其他错误完全可以识别

LR(1)识别

11.合法的加法表达式

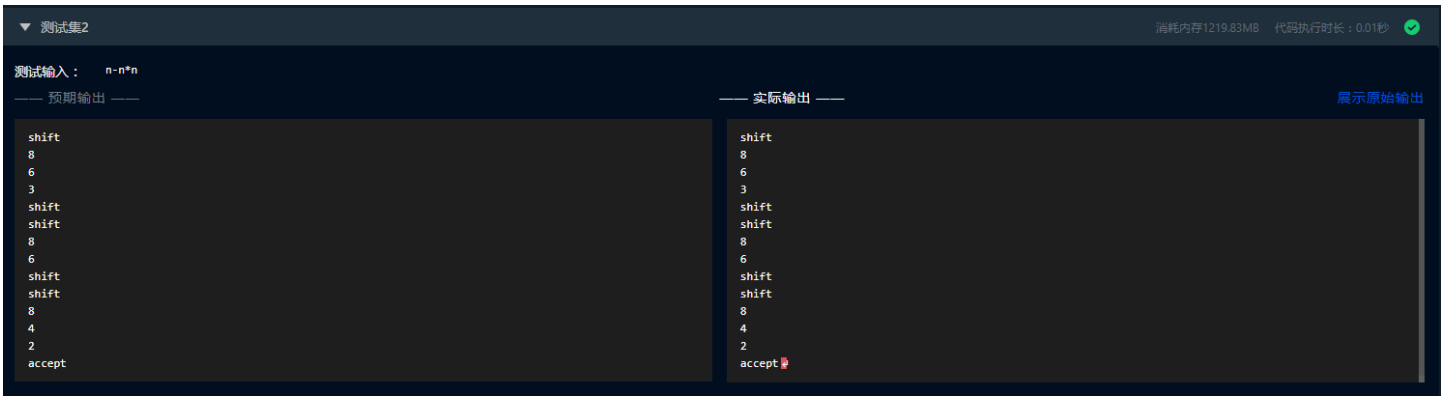
1.运行结果与截图

2.分析说明

加法表达式完全可以识别

12.合法的减法乘法表达式

1.运行结果与截图

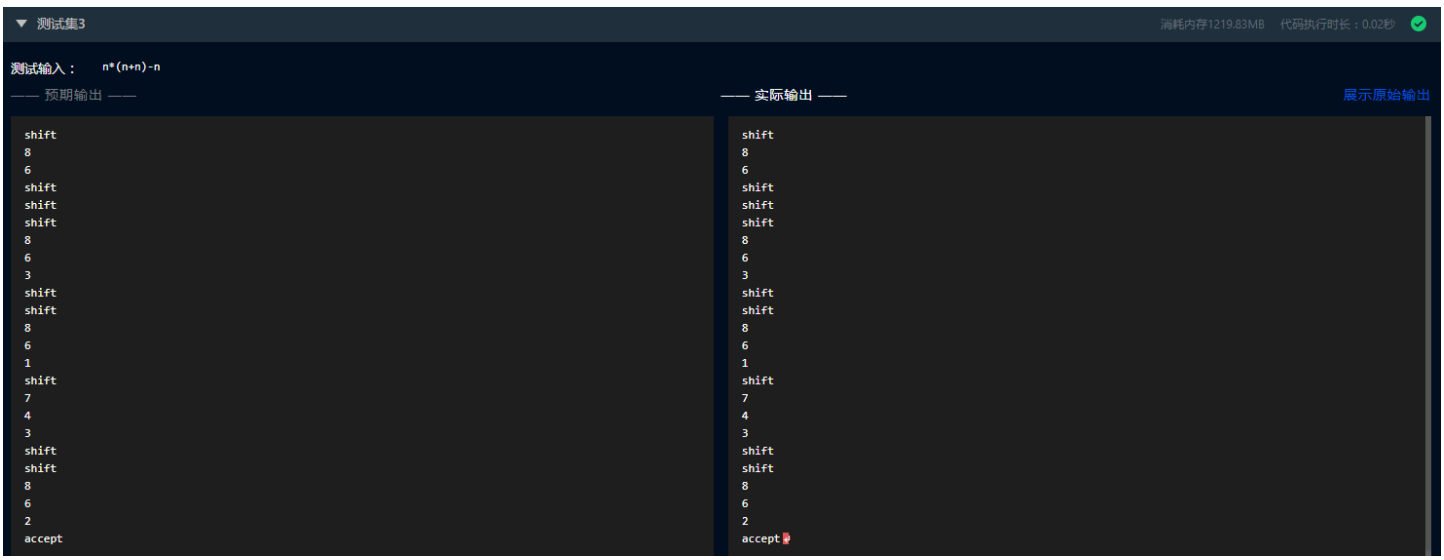


2.分析说明

不同优先级的运算完全可以识别

13.合法的带括号乘法表达式

1.运行结果与截图

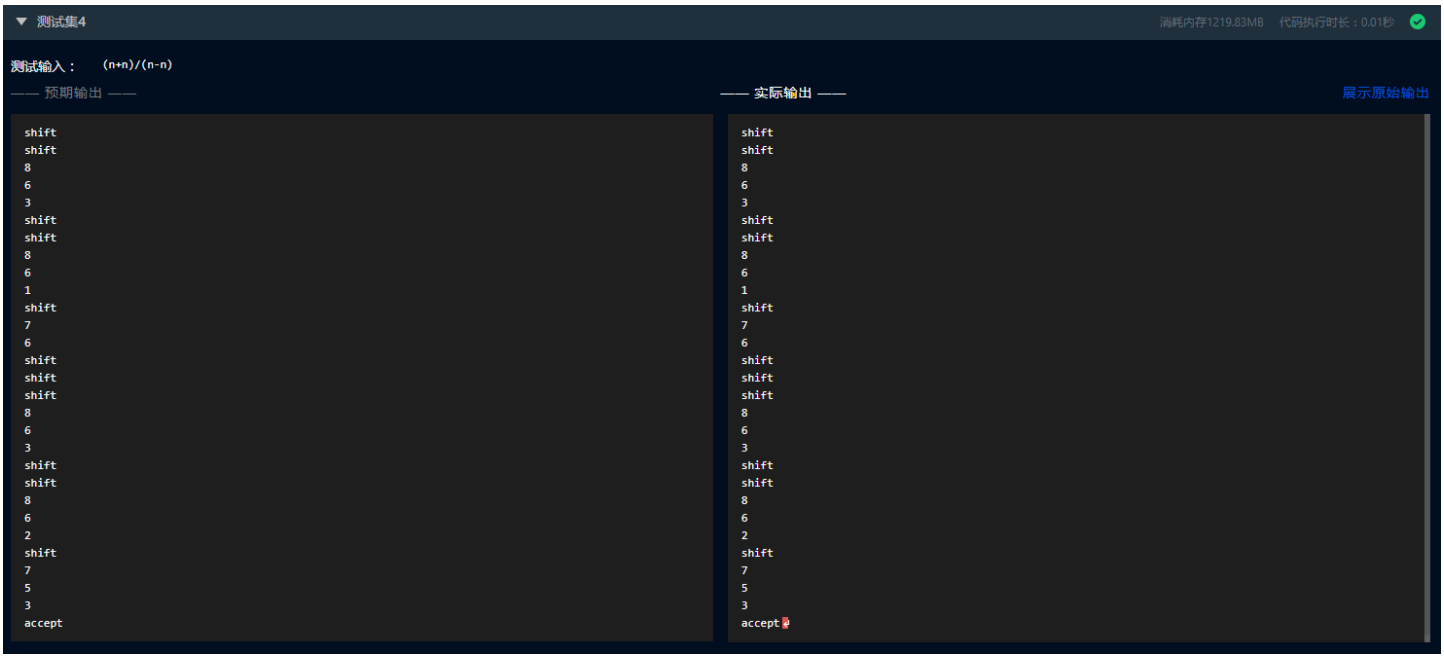


2.分析说明

低优先级运算加入括号成为优先级最高的运算，完全可以识别

14.合法的带括号除法表达式

1.运行结果与截图

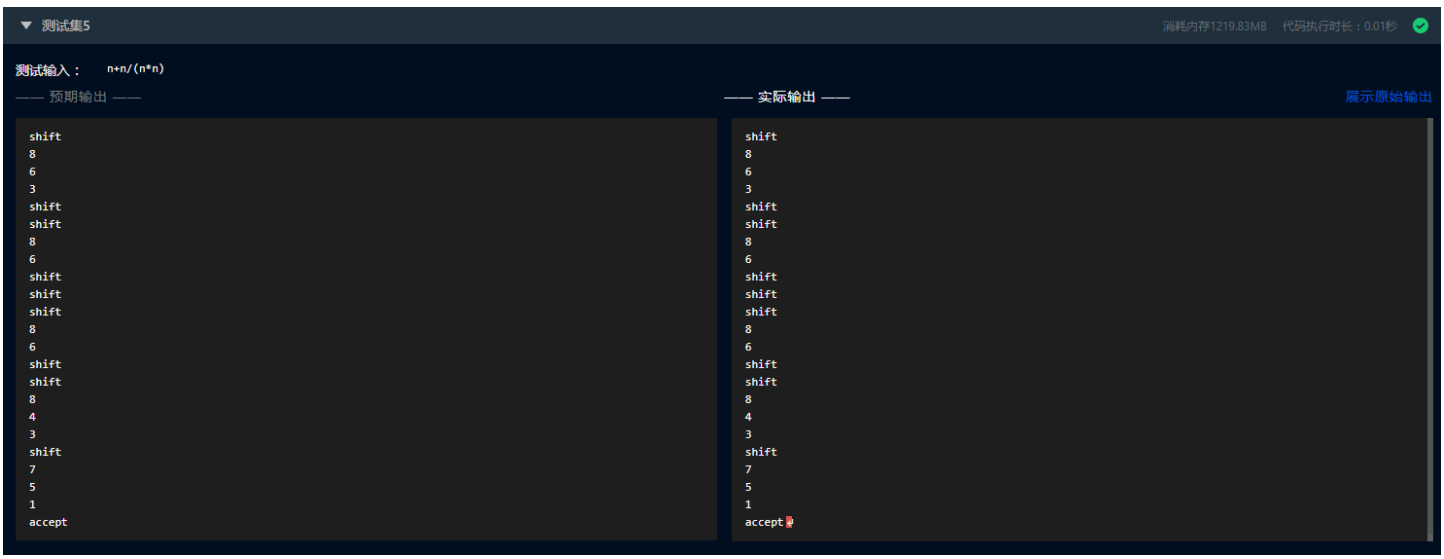


2.分析说明

除法运算完全可以识别

15.较为复杂的合法运算

1.运行结果与截图

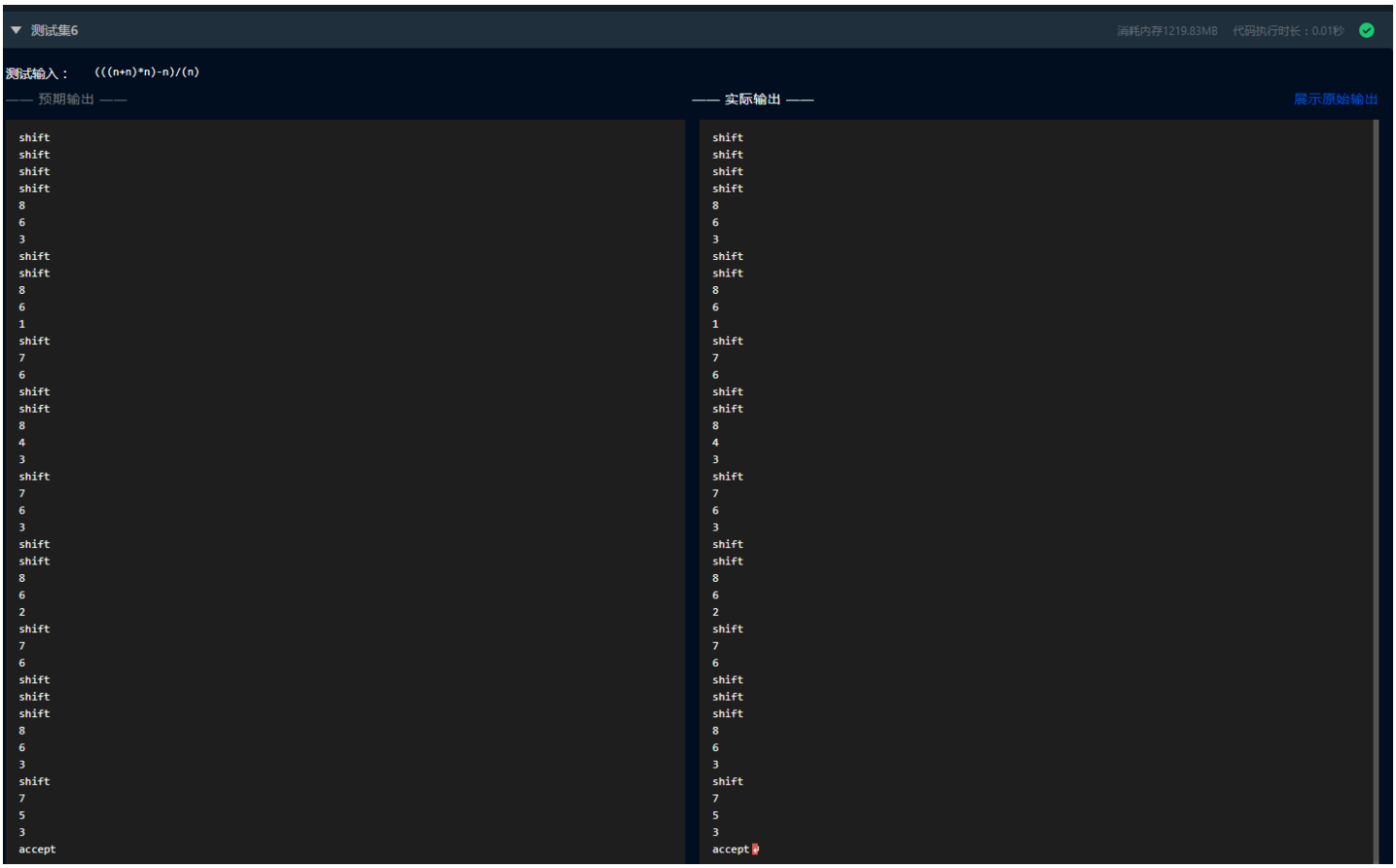


2.分析说明

较为复杂的运算完全可以识别

16.复杂的合法四则运算

1.运行结果与截图



2.分析说明

复杂的四则运算以及其中的优先级完全可以识别

17.不合法的双目运算符错误

1.运行结果与截图

2.分析说明

双目运算符未闭合完全可以识别

18.不合法的双目运算符错误

1.运行结果与截图

2.分析说明

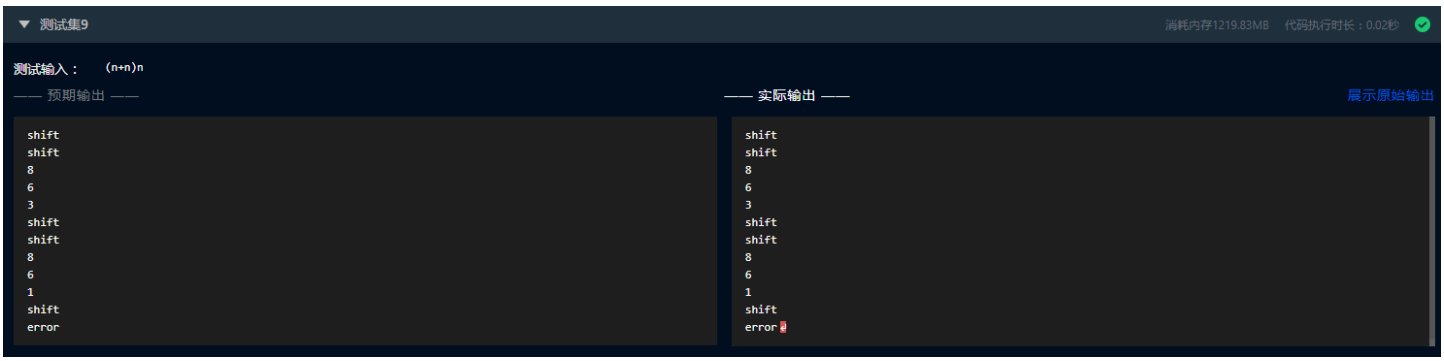
即使只有一个双目运算符也是错误的，可以识别

19.不合法的乘法省略错误

此题目曾出错

详情参考四、9，错误原因相同

1.运行结果与截图

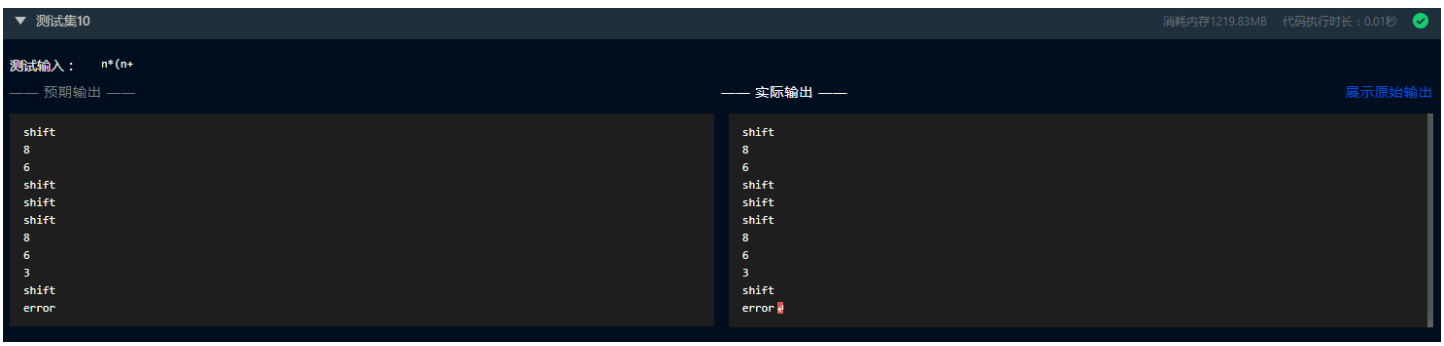


2.分析说明

乘法符号缺失完全可以识别

20.不合法的未闭合括号与双目运算符错误

1.运行结果与截图



2.分析说明

未闭合括号以及其他错误完全可以识别

21.源代码

LL(1)实验代码

```cpp
#include <bits/stdc++.h>
using namespace std;

// 定义产生式
struct Production {
    int id; // 产生式编号
    string lhs; // 左部非终结符
    vector<string> rhs; // 右部符号串

    Production(int identifier, string left, vector<string> right)
        : id(identifier), lhs(left), rhs(right) {}
};

// LL1 解析器类
class LL1Parser {
private:
    vector<Production> productions;                 // 产生式列表
    set<string> nonTerminals;                       // 非终结符集合
    set<string> terminals;                          // 终结符集合
    string startSymbol;                             // 开始符号

    // First 和 Follow 集合
    map<string, set<string>> First;
    map<string, set<string>> Follow;

    // 分析表：非终结符 -> (终结符 -> 产生式编号)
    map<string, map<string, int>> parseTable;

public:
    // 初始化文法
    void initializeGrammar() {
        // 定义非终结符和终结符
        nonTerminals = {"E", "A", "T", "B", "F"};
        terminals = {"+", "-", "*", "/", "(", ")", "n", "$", "e"};

        // 定义开始符号
        startSymbol = "E";

```

```
        // 定义产生式，编号从1开始
        productions.emplace_back(1, "E", vector<string>{"T", "A"});
        productions.emplace_back(2, "A", vector<string>{"+", "T", "A"});
        productions.emplace_back(3, "A", vector<string>{"-", "T", "A"});
        productions.emplace_back(4, "A", vector<string>{"e"});
        productions.emplace_back(5, "T", vector<string>{"F", "B"});
        productions.emplace_back(6, "B", vector<string>{"*", "F", "B"});
        productions.emplace_back(7, "B", vector<string>{"/", "F", "B"});
        productions.emplace_back(8, "B", vector<string>{"e"});
        productions.emplace_back(9, "F", vector<string>{"(", "E", ")"});
        productions.emplace_back(10, "F", vector<string>{"n"});
    }

    // 计算 First 集合
    void computeFirst() {
        // 初始化终结符的 First 集合
        for (auto& t : terminals) {
            if (t != "e") // 'e' 作为特殊符号单独处理
                First[t].insert(t);
        }
        // 'e' 的 FIRST 集合
        First["e"].insert("e");

        // 初始化非终结符的 First 集合为空
        for (auto& nt : nonTerminals) {
            First[nt] = set<string>();
        }

        bool changed = true;
        while (changed) {
            changed = false;
            for (auto& prod : productions) {
                string A = prod.lhs;
                vector<string> alpha = prod.rhs;

                // 计算 First(α)
                set<string> firstAlpha = computeFirstOfString(alpha);

                // 将 First(α) - {e} 加入 First(A)
                for (auto& sym : firstAlpha) {
                    if (sym != "e" && First[A].find(sym) == First[A].end()) {
                        First[A].insert(sym);
                        changed = true;
                    }
                }

                // 如果 e ∈ First(α)，将 e 加入 First(A)
```

```
                if (firstAlpha.find("e") != firstAlpha.end()) {
                    if (First[A].find("e") == First[A].end()) {
                        First[A].insert("e");
                        changed = true;
                    }
                }
            }
        }

    }

    // 计算 Follow 集合
    void computeFollow() {
        // 初始化 Follow 集合为空
        for (auto& nt : nonTerminals) {
            Follow[nt] = set<string>();
        }
        // 开始符号的 Follow 集加入 $
        Follow[startSymbol].insert("$");

        bool changed = true;
        while (changed) {
            changed = false;
            for (auto& prod : productions) {
                string A = prod.lhs;
                vector<string> alpha = prod.rhs;
                for (int j = 0; j < (int)alpha.size(); j++) {
                    string B = alpha[j];
                    if (nonTerminals.find(B) != nonTerminals.end()) {
                        // β = α[j+1 ... end]
                        vector<string> beta(alpha.begin() + j + 1,
alpha.end());
                        if (beta.empty()) {
                            // 如果 β 为空，则将 Follow(A) 加入 Follow(B)
                            for (auto& f : Follow[A]) {
                                if (Follow[B].find(f) == Follow[B].end()) {
                                    Follow[B].insert(f);
                                    changed = true;
                                }
                            }
                        }
                        else {
                            // 计算 First(β)
                            set<string> firstBeta = computeFirstOfString(beta);

                            // 将 First(β) - {e} 加入 Follow(B)
                            for (auto& sym : firstBeta) {
```

```
                                if (sym != "e" && Follow[B].find(sym) ==
Follow[B].end()) {
                                    Follow[B].insert(sym);
                                    changed = true;
                                }
                            }

                            // 如果 e ∈ First(β)，则将 Follow(A) 加入 Follow(B)
                            if (firstBeta.find("e") != firstBeta.end()) {
                                for (auto& f : Follow[A]) {
                                    if (Follow[B].find(f) == Follow[B].end()) {
                                        Follow[B].insert(f);
                                        changed = true;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }

    }

    // 构造分析表
    void buildParseTable() {
        for (auto& prod : productions) {
            string A = prod.lhs;
            vector<string> alpha = prod.rhs;

            // 计算 First(α)
            set<string> firstAlpha = computeFirstOfString(alpha);

            // 对于 a ∈ First(α) - {e}, M[A,a] = prod.id
            for (auto& a : firstAlpha) {
                if (a != "e") {
                    if (parseTable[A].find(a) != parseTable[A].end()) {
                        // 检查是否有冲突
                        cerr << "Parse table conflict at M[" << A << "," << a
<< "] between productions "
                             << parseTable[A][a] << " and " << prod.id << endl;
                        exit(1);
                    }
                    parseTable[A][a] = prod.id;
                }
            }

```

```
177             // 如果 e ∈ First(α), 对于 b ∈ Follow(A), M[A,b] = prod.id
178             if (firstAlpha.find("e") != firstAlpha.end()) {
179                 for (auto& b : Follow[A]) {
180                     if (parseTable[A].find(b) != parseTable[A].end()) {
181                         // 检查是否有冲突
182                         cerr << "Parse table conflict at M[" << A << "," << b
    << "] between productions "
183                              << parseTable[A][b] << " and " << prod.id << endl;
184                         exit(1);
185                     }
186                     parseTable[A][b] = prod.id;
187                 }
188             }
189         }
190     }
191 
192     // 解析输入字符串
193     void parseInput() {
194         // 读取待解析的输入字符串
195         string inputStr;
196         cin >> inputStr;
197 
198         // 分词：将输入字符串按符号匹配终结符
199         vector<string> inputTokens = tokenize(inputStr);
200         inputTokens.push_back("$"); // 末尾加入 $
201 
202         // 初始化解析栈
203         vector<string> parseStack;
204         parseStack.push_back("$");
205         parseStack.push_back(startSymbol);
206 
207         int ip = 0; // 输入指针
208         bool accept = false;
209 
210         while (!parseStack.empty()) {
211             // 构造堆栈字符串
212             string stackStr = "";
213             for (auto& s : parseStack) {
214                 stackStr += s;
215             }
216 
217             // 构造剩余输入字符串
218             string remainingInput = "";
219             for (int i = ip; i < (int)inputTokens.size(); i++) {
220                 remainingInput += inputTokens[i];
221             }
222 
```

```
223             // 获取栈顶符号
224             string X = parseStack.back();
225             string a = inputTokens[ip];
226 
227             // 检查是否接受
228             if (X == "$" && a == "$") {
229                 cout << stackStr << "\t" << remainingInput << "\t" <<
    "accept\n";
230                 accept = true;
231                 break;
232             }
233 
234             // 如果 X 是终结符
235             if (isTerminal(X)) {
236                 if (X == a) {
237                     // match
238                     cout << stackStr << "\t" << remainingInput << "\t" <<
    "match\n";
239                     parseStack.pop_back();
240                     ip++;
241                 }
242                 else {
243                     // error
244                     cout << stackStr << "\t" << remainingInput << "\t" <<
    "error\n";
245                     break;
246                 }
247             }
248             else { // X 是非终结符
249                 // 查找 M[X, a]
250                 if (parseTable.find(X) != parseTable.end() &&
    parseTable[X].find(a) != parseTable[X].end()) {
251                     int prodNum = parseTable[X][a];
252                     Production& prod = productions[prodNum - 1]; // 产生式编号从1
    开始
253                     // 输出使用的产生式编号
254                     cout << stackStr << "\t" << remainingInput << "\t" <<
    prodNum << "\n";
255                     // 弹出栈顶
256                     parseStack.pop_back();
257                     // 将产生式右部逆序压栈（如果不是 e）
258                     if (!(prod.rhs.size() == 1 && prod.rhs[0] == "e")) {
259                         for (int i = prod.rhs.size() - 1; i >= 0; --i) {
260                             parseStack.push_back(prod.rhs[i]);
261                         }
262                     }
263                 }
```

```
                else {
                    // error
                    cout << stackStr << "\t" << remainingInput << "\t" << "error\n";
                    break;
                }
            }
        }

        if (!accept) {

        }
    }

    // 运行解析器
    void run() {
        initializeGrammar();
        computeFirst();
        computeFollow();
        buildParseTable();
        parseInput();
    }

private:
    // 辅助函数：判断是否是非终结符
    bool isNonTerminal(const string& sym) {
        return nonTerminals.find(sym) != nonTerminals.end();
    }

    // 辅助函数：判断是否是终结符
    bool isTerminal(const string& sym) {
        return terminals.find(sym) != terminals.end();
    }

    // 辅助函数：去除字符串首尾空白
    string trim(const string& s) {
        string result = s;
        // 左
        result.erase(result.begin(), find_if(result.begin(), result.end(), [](int ch) {
            return !isspace(ch);
        }));
        // 右
        result.erase(find_if(result.rbegin(), result.rend(), [](int ch) {
            return !isspace(ch);
        }).base(), result.end());
        return result;
```

```
309     }
310
311     // 辅助函数：按分隔符分割字符串
312     vector<string> split(const string& s, char delimiter) {
313         vector<string> tokens;
314         string token;
315         stringstream ss(s);
316         while (getline(ss, token, delimiter)) {
317             if (!token.empty())
318                 tokens.push_back(token);
319         }
320         return tokens;
321     }
322
323     // 计算一串符号的 First 集合
324     set<string> computeFirstOfString(const vector<string>& symbols) {
325         set<string> result;
326         bool epsilonFound = true;
327         for (auto& sym : symbols) {
328             for (auto& f : First[sym]) {
329                 if (f != "e") {
330                     result.insert(f);
331                 }
332             }
333             if (First[sym].find("e") == First[sym].end()) {
334                 epsilonFound = false;
335                 break;
336             }
337         }
338         if (epsilonFound) {
339             result.insert("e");
340         }
341         return result;
342     }
343
344     // 分词函数：将输入字符串转化为终结符序列
345     vector<string> tokenize(const string& input) {
346         vector<string> tokens;
347         size_t pos = 0;
348         while (pos < input.size()) {
349             bool matched = false;
350             string matchedToken = "";
351             // 尝试匹配多字符终结符（当前文法无多字符终结符，保留此逻辑以便扩展）
352             for (auto& t : terminals) {
353                 if (t.size() > 1 && pos + t.size() <= input.size()) {
354                     if (input.substr(pos, t.size()) == t) {
355                         if (t.size() > matchedToken.size()) {
```

```
                            matchedToken = t;
                        }
                    }
                }
            }
            if (matchedToken != "") {
                tokens.push_back(matchedToken);
                pos += matchedToken.size();
                matched = true;
            }
            if (!matched) {
                // 如果没有匹配到多字符终结符，则按单字符处理
                string token(1, input[pos]);
                tokens.push_back(token);
                pos++;
            }
        }
        return tokens;
    }
};

int main() {
    LL1Parser parser;
    parser.run();
    return 0;
}
```

LR(1)实验代码

```
#include <bits/stdc++.h>
using namespace std;

// 定义产生式
struct Production {
    int id; // 产生式编号
    string lhs; // 左部非终结符
    vector<string> rhs; // 右部符号串

    Production(int identifier, string left, vector<string> right)
        : id(identifier), lhs(left), rhs(right) {
    }
};

// 定义LR(1)项目
struct LR1Item {
```

```
    string lhs;
    vector<string> rhs;
    int dot; // 点的位置
    string lookahead;
    int prodId; // 产生式编号

    LR1Item(string left, vector<string> right, int d, string la, int pid)
        : lhs(left), rhs(right), dot(d), lookahead(la), prodId(pid) {
    }

    bool operator<(const LR1Item& other) const {
        if (lhs != other.lhs)
            return lhs < other.lhs;
        if (rhs != other.rhs)
            return rhs < other.rhs;
        if (dot != other.dot)
            return dot < other.dot;
        if (lookahead != other.lookahead)
            return lookahead < other.lookahead;
        return prodId < other.prodId;
    }

    bool operator==(const LR1Item& other) const {
        return lhs == other.lhs && rhs == other.rhs && dot == other.dot &&
lookahead == other.lookahead && prodId == other.prodId;
    }
};

// 语法分析器类
class LR1Parser {
private:
    vector<Production> productions;
    set<string> nonTerminals;
    set<string> terminals;
    string startSymbol;

    // First 和 Follow 集合
    map<string, set<string>> First;
    map<string, set<string>> Follow;

    // 项目集规范族
    vector<set<LR1Item>> C;

    // 分析表
    // Action 表：状态 -> (终结符 -> Action)
    // Action 的值为 "shift X", "reduce Y", "accept"
    map<int, map<string, string>> Action;
```

```
63
64      // Goto 表：状态 -> (非终结符 -> 状态)
65      map<int, map<string, int>> GotoTable;
66
67  public:
68      // 初始化文法
69      void initializeGrammar() {
70          // 定义非终结符和终结符
71          nonTerminals = {"E'", "E", "T", "F"};
72          terminals = {"+", "-", "*", "/", "(", ")", "n", "$"};
73
74          // 定义开始符号
75          startSymbol = "E'";
76
77          // 定义产生式
78          // 编号从0开始
79          productions.emplace_back(0, "E'", vector<string>{"E"});
80          productions.emplace_back(1, "E", vector<string>{"E", "+", "T"});
81          productions.emplace_back(2, "E", vector<string>{"E", "-", "T"});
82          productions.emplace_back(3, "E", vector<string>{"T"});
83          productions.emplace_back(4, "T", vector<string>{"T", "*", "F"});
84          productions.emplace_back(5, "T", vector<string>{"T", "/", "F"});
85          productions.emplace_back(6, "T", vector<string>{"F"});
86          productions.emplace_back(7, "F", vector<string>{"(", "E", ")"});
87          productions.emplace_back(8, "F", vector<string>{"n"});
88      }
89
90      // 计算 First 集合
91      void computeFirst() {
92          // 初始化终结符的 First 集合
93          for(auto &t : terminals){
94              if(t != "e") // 'e' 作为特殊符号单独处理（当前文法无epsilon）
95                  First[t].insert(t);
96          }
97
98          // 初始化非终结符的 First 集合为空
99          for(auto &nt : nonTerminals){
100             First[nt] = set<string>();
101         }
102
103         bool changed = true;
104         while(changed){
105             changed = false;
106             for(auto &prod : productions){
107                 string A = prod.lhs;
108                 vector<string> alpha = prod.rhs;
109
```

```
                // 计算 First(α)
                set<string> firstAlpha = computeFirstOfString(alpha);

                // 将 First(α) - {e} 加入 First(A)
                for(auto &sym : firstAlpha){
                    if(sym != "e" && First[A].find(sym) == First[A].end()){
                        First[A].insert(sym);
                        changed = true;
                    }
                }

                // 如果 e ∈ First(α)，将 e 加入 First(A)
                if(firstAlpha.find("e") != firstAlpha.end()){
                    if(First[A].find("e") == First[A].end()){
                        First[A].insert("e");
                        changed = true;
                    }
                }
            }
        }
    }

    // 计算 Follow 集合
    void computeFollow() {
        // 初始化 Follow 集合为空
        for(auto &nt : nonTerminals){
            Follow[nt] = set<string>();
        }
        // 开始符号的 Follow 集加入 $
        Follow[startSymbol].insert("$");

        bool changed = true;
        while(changed){
            changed = false;
            for(auto &prod : productions){
                string A = prod.lhs;
                vector<string> alpha = prod.rhs;
                for(int i = 0; i < (int)alpha.size(); i++){
                    string B = alpha[i];
                    if(nonTerminals.find(B) != nonTerminals.end()){
                        // β = α[i+1 ... end]
                        vector<string> beta(alpha.begin() + i + 1,
alpha.end());
                        if(beta.empty()){
                            // 如果 β 为空，则将 Follow(A) 加入 Follow(B)
                            for(auto &f : Follow[A]){
                                if(Follow[B].find(f) == Follow[B].end()){
```

```
                                    Follow[B].insert(f);
                                    changed = true;
                                }
                            }
                        }
                        else{
                            // 计算 First(β)
                            set<string> firstBeta = computeFirstOfString(beta);

                            // 将 First(β) - {e} 加入 Follow(B)
                            for(auto &sym : firstBeta){
                                if(sym != "e" && Follow[B].find(sym) ==
    Follow[B].end()){
                                    Follow[B].insert(sym);
                                    changed = true;
                                }
                            }

                            // 如果 e ∈ First(β)，则将 Follow(A) 加入 Follow(B)
                            if(firstBeta.find("e") != firstBeta.end()){
                                for(auto &f : Follow[A]){
                                    if(Follow[B].find(f) == Follow[B].end()){
                                        Follow[B].insert(f);
                                        changed = true;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    // 闭包操作
    set<LR1Item> closure(const set<LR1Item>& I) {
        set<LR1Item> closureSet = I;
        bool changed = true;

        while(changed){
            changed = false;
            set<LR1Item> newItems;
            for(auto &item : closureSet){
                if(item.dot < (int)item.rhs.size()){
                    string B = item.rhs[item.dot];
                    if(nonTerminals.find(B) != nonTerminals.end()){
                        // β = item.rhs[item.dot +1 ... end]
```

```
                        vector<string> beta(item.rhs.begin() + item.dot +1,
item.rhs.end());
                        // a = item.lookahead
                        string a = item.lookahead;
                        // FIRST(β a)
                        vector<string> symbols = beta;
                        symbols.push_back(a);
                        set<string> firstBetaA = computeFirstOfString(symbols);
                        for(auto &prod : productions){
                            if(prod.lhs == B){
                                for(auto &la : firstBetaA){
                                    if(la == "e") continue; // 通常不添加
lookahead 为 e 的项目
                                    LR1Item newItem(prod.lhs, prod.rhs, 0, la,
prod.id);
                                    if(closureSet.find(newItem) ==
closureSet.end() && newItems.find(newItem) == newItems.end()){
                                        newItems.insert(newItem);
                                        changed = true;
                                    }
                                }
                            }
                        }
                    }
                }
            }
            closureSet.insert(newItems.begin(), newItems.end());
        }

        return closureSet;
    }

    // 迁移操作
    set<LR1Item> goto_func(const set<LR1Item>& I, const string& X) {
        set<LR1Item> J;
        for(auto &item : I){
            if(item.dot < (int)item.rhs.size() && item.rhs[item.dot] == X){
                LR1Item movedItem = item;
                movedItem.dot += 1;
                J.insert(movedItem);
            }
        }
        return closure(J);
    }

    // 构建项目集规范族
    void buildCanonicalCollection() {
```

```
245         // 初始项集 C0 = closure({ E' -> . E, $ })
246         set<LR1Item> C0;
247         // 生产式 0: E' -> E
248         C0.emplace(LR1Item(productions[0].lhs, productions[0].rhs, 0, "$",
    productions[0].id));
249         set<LR1Item> closureC0 = closure(C0);
250         C.push_back(closureC0);
251 
252         // 使用 BFS 构建项目集
253         queue<int> q;
254         q.push(0);
255 
256         while(!q.empty()){
257             int i = q.front();
258             q.pop();
259             set<string> symbols;
260             for(auto &item : C[i]){
261                 if(item.dot < (int)item.rhs.size()){
262                     symbols.insert(item.rhs[item.dot]);
263                 }
264             }
265 
266             for(auto &X : symbols){
267                 set<LR1Item> gotoI = goto_func(C[i], X);
268                 if(gotoI.empty()) continue;
269 
270                 // Check if gotoI already exists in C
271                 int j = -1;
272                 for(int k = 0; k < (int)C.size(); ++k){
273                     if(C[k] == gotoI){
274                         j = k;
275                         break;
276                     }
277                 }
278 
279                 if(j == -1){
280                     C.push_back(gotoI);
281                     j = C.size() -1;
282                     q.push(j);
283                 }
284 
285                 // 填充 Action 和 Goto 表
286                 if(terminals.find(X) != terminals.end()){
287                     Action[i][X] = "shift " + to_string(j);
288                 }
289                 else if(nonTerminals.find(X) != nonTerminals.end()){
290                     GotoTable[i][X] = j;
```

```
                }
            }
        }
    }

    // 构建分析表
    void buildParseTable() {
        for(int i = 0; i < (int)C.size(); ++i){
            for(auto &item : C[i]){
                if(item.dot < (int)item.rhs.size()){
                    string a = item.rhs[item.dot];
                    if(terminals.find(a) != terminals.end()){
                        // 查找 goto(Ci, a)
                        set<LR1Item> gotoSet = goto_func(C[i], a);
                        if(!gotoSet.empty()){
                            // 查找状态 j
                            int j = -1;
                            for(int k = 0; k < (int)C.size(); ++k){
                                if(C[k] == gotoSet){
                                    j = k;
                                    break;
                                }
                            }
                            if(j != -1){
                                Action[i][a] = "shift " + to_string(j);
                            }
                        }
                    }
                }
                else{
                    if(item.lhs != productions[0].lhs){
                        // A -> α ., a
                        // Action[i, a] = reduce prod.id
                        Action[i][item.lookahead] = "reduce " +
to_string(item.prodId);
                    }
                    else{
                        // E' -> E ., $
                        if(item.lookahead == "$"){
                            Action[i][item.lookahead] = "accept";
                        }
                    }
                }
            }
        }
    }

```

```
337     // 解析输入字符串并输出分析过程
338     void parseInput() {
339         // 读取待解析的输入字符串
340         string inputStr;
341         cin >> inputStr;
342 
343         // 分词：将输入字符串按符号匹配终结符
344         vector<string> inputTokens = tokenize(inputStr);
345         inputTokens.push_back("$"); // 末尾加入 $
346 
347         // 初始化解析栈
348         vector<int> parseStack;
349         parseStack.push_back(0);
350 
351         int ip = 0; // 输入指针
352         bool accept = false;
353 
354         // 存储输出动作
355         vector<string> actions;
356 
357         while(true){
358             int state = parseStack.back();
359             string a = inputTokens[ip];
360 
361             // 查找 Action[state][a]
362             if(Action.find(state) != Action.end() && Action[state].find(a) !=
    Action[state].end()){
363                 string action = Action[state][a];
364                 if(action.substr(0,5) == "shift"){
365                     actions.push_back("shift");
366                     // 获取状态 j
367                     int j = stoi(action.substr(6));
368                     parseStack.push_back(j);
369                     ip++;
370                 }
371                 else if(action.substr(0,6) == "reduce"){
372                     // 获取生产式编号
373                     int prodId = stoi(action.substr(7));
374                     actions.push_back(to_string(prodId));
375                     Production prod = productions[prodId];
376                     // 弹出 rhs.size() 个状态
377                     for(size_t k = 0; k < prod.rhs.size(); ++k){
378                         if(!parseStack.empty())
379                             parseStack.pop_back();
380                         else{
381                             // Stack underflow
382                             // actions.push_back("error");
```

```
                            break;
                        }
                    }
                    // 获取当前状态
                    if(parseStack.empty()){
                        // Stack is empty after reduction
                        break;
                    }
                    int currentState = parseStack.back();
                    // Goto[currentState][A] = j
                    if(GotoTable.find(currentState) != GotoTable.end() &&
GotoTable[currentState].find(prod.lhs) != GotoTable[currentState].end()){
                        int j = GotoTable[currentState][prod.lhs];
                        parseStack.push_back(j);
                    }
                    else{
                        // Goto table entry not found
                        // actions.push_back("error");
                        break;
                    }
                }
                else if(action == "accept"){
                    actions.push_back("accept");
                    accept = true;
                    break;
                }
            }
            else{
                // 查找 Action[state][a] 不存在，解析错误
                actions.push_back("error");
                break;
            }
        }

        // 输出动作
        for(auto &act : actions){
            cout << act << "\n";
        }

        if(!accept){

        }
    }

    // 辅助函数：判断是否是非终结符
    bool isNonTerminal(const string& sym){
        return nonTerminals.find(sym) != nonTerminals.end();
```

```
429     }
430 
431     // 辅助函数：去除字符串首尾空白
432     string trim(const string& s){
433         string result = s;
434         // 左
435         result.erase(result.begin(), find_if(result.begin(), result.end(), []
    (int ch) {
436             return !isspace(ch);
437         }));
438         // 右
439         result.erase(find_if(result.rbegin(), result.rend(), [](int ch) {
440             return !isspace(ch);
441         }).base(), result.end());
442         return result;
443     }
444 
445     // 辅助函数：按分隔符分割字符串
446     vector<string> split(const string& s, char delimiter){
447         vector<string> tokens;
448         string token;
449         stringstream ss(s);
450         while(getline(ss, token, delimiter)){
451             if(!token.empty())
452                 tokens.push_back(token);
453         }
454         return tokens;
455     }
456 
457     // 计算一串符号的 First 集合
458     set<string> computeFirstOfString(const vector<string>& symbols){
459         set<string> result;
460         bool epsilonFound = true;
461         for(auto &sym : symbols){
462             for(auto &f : First[sym]){
463                 if(f != "e"){
464                     result.insert(f);
465                 }
466             }
467             if(First[sym].find("e") == First[sym].end()){
468                 epsilonFound = false;
469                 break;
470             }
471         }
472         if(epsilonFound){
473             result.insert("e");
474         }
```

```
        return result;
    }

    // 将符号串转为字符串（用于映射）
    string stringify(const vector<string>& symbols){
        string s;
        for(auto &sym : symbols){
            s += sym + " ";
        }
        return s;
    }

    // 分词函数：将输入字符串转化为终结符序列
    vector<string> tokenize(const string& input){
        vector<string> tokens;
        size_t pos = 0;
        while(pos < input.size()){
            bool matched = false;
            string matchedToken = "";
            // 尝试匹配多字符终结符（当前文法无多字符终结符，保留此逻辑以便扩展）
            for(auto &t : terminals){
                if(t.size() > 1 && pos + t.size() <= input.size()){
                    if(input.substr(pos, t.size()) == t){
                        if(t.size() > matchedToken.size()){
                            matchedToken = t;
                        }
                    }
                }
            }
            if(matchedToken != ""){
                tokens.push_back(matchedToken);
                pos += matchedToken.size();
                matched = true;
            }
            if(!matched){
                // 如果没有匹配到多字符终结符，则按单字符处理
                string token(1, input[pos]);
                tokens.push_back(token);
                pos++;
            }
        }
        return tokens;
    }

    // 运行解析器
    void run(){
        initializeGrammar();
```

```
        computeFirst();
        computeFollow();
        buildCanonicalCollection();
        buildParseTable();
        parseInput();
    }
};

int main(){
    LR1Parser parser;
    parser.run();
    return 0;
}
```

LL(1)扩展代码

```
#include <bits/stdc++.h>
#include <iomanip>
using namespace std;

// 定义产生式
struct Production {
    string lhs;                 // 左部非终结符
    vector<string> rhs;         // 右部产生式

    Production() {}
    Production(string left, vector<string> right) : lhs(left), rhs(right) {}
};

// LL1 解析器类
class LL1Parser {
private:
    vector<Production> productions;              // 产生式列表
    set<string> nonTerminals;                    // 非终结符集合
    set<string> terminals;                       // 终结符集合
    string startSymbol;                          // 开始符号

    // First 和 Follow 集合
    map<string, set<string>> First;
    map<string, set<string>> Follow;

    // 分析表：非终结符 -> (终结符 -> 产生式编号)
    map<string, map<string, int>> parseTable;

public:
```

```
    // 读取文法规则
    void readGrammar() {
        // 输入开始符号
        cin >> startSymbol;

        // 输入非终结符集合
        string line;
        getline(cin, line); // 读取剩余的换行符
        getline(cin, line);
        vector<string> ntTokens = split(line, ' ');
        nonTerminals = set<string>(ntTokens.begin(), ntTokens.end());

        // 输入终结符集合（包括 'e' 作为 ε）
        getline(cin, line);
        vector<string> tTokens = split(line, ' ');
        terminals = set<string>(tTokens.begin(), tTokens.end());
        terminals.insert("e"); // 将 'e' 作为终结符添加

        // 输入产生式数量
        int P;
        cin >> P;
        getline(cin, line); // 读取剩余的换行符

        // 读取产生式
        for (int i = 0; i < P; ++i) {
            getline(cin, line);
            line = trim(line);
            size_t arrow = line.find("->");
            if (arrow == string::npos) {
                cerr << "Invalid production format: " << line << endl;
                exit(1);
            }
            string lhs = trim(line.substr(0, arrow));
            string rhsPart = trim(line.substr(arrow + 2));
            vector<string> alternatives = split(rhsPart, '|');
            for (auto& alt : alternatives) {
                alt = trim(alt);
                if (alt == "e") { // 将 'e' 视为 ε
                    productions.emplace_back(lhs, vector<string>{ "e" });
                }
                else {
                    vector<string> symbols = split(alt, ' ');
                    productions.emplace_back(lhs, symbols);
                }
            }
        }

```

```
77          // 读取待分析的输入字符串
78          cin >> line;
79          inputString = line;
80      }
81
82      // 计算 First 集合
83      void computeFirst() {
84          // 初始化终结符的 First 集合
85          for (auto& t : terminals) {
86              if (t != "e") // 'e' 作为特殊符号单独处理
87                  First[t].insert(t);
88          }
89          // 'e' 的 FIRST 集合
90          First["e"].insert("e");
91
92          // 初始化非终结符的 First 集合为空
93          for (auto& nt : nonTerminals) {
94              First[nt] = set<string>();
95          }
96
97          bool changed = true;
98          while (changed) {
99              changed = false;
100             for (int i = 0; i < productions.size(); ++i) {
101                 Production& prod = productions[i];
102                 string A = prod.lhs;
103                 vector<string> alpha = prod.rhs;
104
105                 // 计算 First(alpha)
106                 set<string> firstAlpha = computeFirstOfString(alpha);
107
108                 // 将 First(alpha) - {e} 加入 First(A)
109                 for (auto& sym : firstAlpha) {
110                     if (sym != "e" && First[A].find(sym) == First[A].end()) {
111                         First[A].insert(sym);
112                         changed = true;
113                     }
114                 }
115
116                 // 如果 e ∈ First(alpha)，将 e 加入 First(A)
117                 if (firstAlpha.find("e") != firstAlpha.end()) {
118                     if (First[A].find("e") == First[A].end()) {
119                         First[A].insert("e");
120                         changed = true;
121                     }
122                 }
123             }
```

```
124         }
125 
126         // 打印 FIRST 集合
127         cout << "\nFIRST sets:\n";
128         for (auto& pair : First) {
129             cout << pair.first << ": { ";
130             for (auto& sym : pair.second) {
131                 cout << sym << " ";
132             }
133             cout << "}\n";
134         }
135     }
136 
137     // 计算 Follow 集合
138     void computeFollow() {
139         // 初始化 Follow 集合为空
140         for (auto& nt : nonTerminals) {
141             Follow[nt] = set<string>();
142         }
143         // 开始符号的 Follow 集加入 $
144         Follow[startSymbol].insert("$");
145 
146         bool changed = true;
147         while (changed) {
148             changed = false;
149             for (int i = 0; i < productions.size(); ++i) {
150                 Production& prod = productions[i];
151                 string A = prod.lhs;
152                 vector<string> alpha = prod.rhs;
153                 for (int j = 0; j < (int)alpha.size(); j++) {
154                     string B = alpha[j];
155                     if (nonTerminals.find(B) != nonTerminals.end()) {
156                         // β = alpha[j+1 ... end]
157                         vector<string> beta(alpha.begin() + j + 1,
    alpha.end());
158                         if (beta.empty()) {
159                             // 如果 β 为空，则将 Follow(A) 加入 Follow(B)
160                             for (auto& f : Follow[A]) {
161                                 if (Follow[B].find(f) == Follow[B].end()) {
162                                     Follow[B].insert(f);
163                                     changed = true;
164                                 }
165                             }
166                         }
167                         else {
168                             // 计算 First(beta)
169                             set<string> firstBeta = computeFirstOfString(beta);
```

```
170
171                             // 将 First(beta) - {e} 加入 Follow(B)
172                             for (auto& sym : firstBeta) {
173                                 if (sym != "e" && Follow[B].find(sym) ==
    Follow[B].end()) {
174                                     Follow[B].insert(sym);
175                                     changed = true;
176                                 }
177                             }
178
179                             // 如果 e ∈ First(beta)，则将 Follow(A) 加入
    Follow(B)
180                             if (firstBeta.find("e") != firstBeta.end()) {
181                                 for (auto& f : Follow[A]) {
182                                     if (Follow[B].find(f) == Follow[B].end()) {
183                                         Follow[B].insert(f);
184                                         changed = true;
185                                     }
186                                 }
187                             }
188                         }
189                     }
190                 }
191             }
192         }
193
194         // 打印 FOLLOW 集合
195         cout << "\nFOLLOW sets:\n";
196         for (auto& pair : Follow) {
197             cout << pair.first << ": { ";
198             for (auto& sym : pair.second) {
199                 cout << sym << " ";
200             }
201             cout << "}\n";
202         }
203     }
204
205     // 构造分析表
206     void buildParseTable() {
207         for (int i = 0; i < (int)productions.size(); ++i) {
208             Production& prod = productions[i];
209             string A = prod.lhs;
210             vector<string> alpha = prod.rhs;
211
212             // 计算 First(alpha)
213             set<string> firstAlpha = computeFirstOfString(alpha);
214
```

```
215             // 对于 a ∈ First(alpha) - {e}, M[A,a] = i+1
216             for (auto& a : firstAlpha) {
217                 if (a != "e") {
218                     if (parseTable[A].find(a) != parseTable[A].end()) {
219                         // 检查是否有冲突
220                         cerr << "Parse table conflict at M[" << A << "," << a
    << "] between productions "
221                              << parseTable[A][a] << " and " << (i + 1) << endl;
222                         exit(1);
223                     }
224                     parseTable[A][a] = i + 1; // 1-based indexing
225                 }
226             }
227 
228             // 如果 e ∈ First(alpha), 对于 b ∈ Follow(A), M[A,b] = i+1
229             if (firstAlpha.find("e") != firstAlpha.end()) {
230                 for (auto& b : Follow[A]) {
231                     if (parseTable[A].find(b) != parseTable[A].end()) {
232                         // 检查是否有冲突
233                         cerr << "Parse table conflict at M[" << A << "," << b
    << "] between productions "
234                              << parseTable[A][b] << " and " << (i + 1) << endl;
235                         exit(1);
236                     }
237                     parseTable[A][b] = i + 1;
238                 }
239             }
240         }
241 
242         // 不在 buildParseTable() 中打印解析表，而是使用单独的函数
243     }
244 
245     // 打印预测分析表
246     void printParseTable() {
247         // 收集所有终结符（包括 $）
248         vector<string> termList;
249         for (const auto& t : terminals) {
250             if (t != "e") // 'e' 作为特殊符号单独处理
251                 termList.push_back(t);
252         }
253         termList.push_back("$"); // 添加结束符
254 
255         // 收集所有非终结符
256         vector<string> ntList(nonTerminals.begin(), nonTerminals.end());
257 
258         // 打印 Action 表
259         cout << "\nLL(1) Parse Table (Action):\n";
```

```
260         // 打印表头
261         cout << left << setw(15) << "Non-Terminal";
262         for (const auto& term : termList) {
263             cout << left << setw(15) << term;
264         }
265         cout << "\n";
266 
267         // 打印每个非终结符的行
268         for (const auto& A : ntList) {
269             cout << left << setw(15) << A;
270             for (const auto& term : termList) {
271                 if (parseTable.find(A) != parseTable.end() &&
    parseTable[A].find(term) != parseTable[A].end()) {
272                     cout << left << setw(15) << parseTable[A][term];
273                 }
274                 else {
275                     cout << left << setw(15) << "";
276                 }
277             }
278             cout << "\n";
279         }
280 
281         // 如果需要，可以分开 Action 和 Goto 表
282         // 但在LL(1)中，Goto表通常不需要，因为解析表已经包含了所有必要的信息
283     }
284 
285     // 解析输入字符串
286     void parseInput() {
287         // 分词：将输入字符串按字符拆分
288         vector<string> inputTokens = tokenize(inputString);
289         inputTokens.push_back("$"); // 末尾加入 $
290 
291         // 初始化解析栈
292         vector<string> parseStack;
293         parseStack.push_back("$");
294         parseStack.push_back(startSymbol);
295 
296         int ip = 0; // 输入指针
297         bool accept = false;
298 
299         // 输出表头
300         cout << "\nParsing Actions:\n";
301         cout << left << setw(30) << "Stack" << setw(30) << "Input" <<
    "Action\n";
302 
303         while (!parseStack.empty()) {
304             // 构造堆栈字符串
```

```
305             string stackStr = "";
306             for (auto& s : parseStack) {
307                 stackStr += s + " ";
308             }
309 
310             // 构造剩余输入字符串
311             string inputStr = "";
312             for (int i = ip; i < (int)inputTokens.size(); i++) {
313                 inputStr += inputTokens[i] + " ";
314             }
315 
316             // 获取栈顶符号
317             string X = parseStack.back();
318             string a = inputTokens[ip];
319 
320             // 检查是否接受
321             if (X == "$" && a == "$") {
322                 cout << left << setw(30) << stackStr << setw(30) << inputStr
    << "accept\n";
323                 accept = true;
324                 break;
325             }
326 
327             // 如果 X 是终结符
328             if (isTerminal(X)) {
329                 if (X == a) {
330                     // match
331                     cout << left << setw(30) << stackStr << setw(30) <<
    inputStr << "match\n";
332                     parseStack.pop_back();
333                     ip++;
334                 }
335                 else {
336                     // error
337                     cout << left << setw(30) << stackStr << setw(30) <<
    inputStr << "error\n";
338                     break;
339                 }
340             }
341             else { // X 是非终结符
342                 // 查找 M[X, a]
343                 if (parseTable.find(X) != parseTable.end() &&
    parseTable[X].find(a) != parseTable[X].end()) {
344                     int prodNum = parseTable[X][a];
345                     // 检查生产式编号是否有效
346                     if (prodNum <= 0 || prodNum > (int)productions.size()) {
```

```
                        cerr << "Error: Invalid production number " << prodNum
<< " for M[" << X << "," << a << "].\n";
                        cout << left << setw(30) << stackStr << setw(30) <<
inputStr << "error\n";
                        break;
                    }
                    Production& prod = productions[prodNum - 1]; // 1-based
indexing

                    // 输出使用的产生式编号
                    cout << left << setw(30) << stackStr << setw(30) <<
inputStr << "Use production " << prodNum << ": " << prod.lhs << " -> ";
                    for (const auto& sym : prod.rhs) {
                        cout << sym << " ";
                    }
                    cout << "\n";

                    // 弹出栈顶
                    parseStack.pop_back();

                    // 将产生式右部逆序压栈（如果不是 e）
                    if (!(prod.rhs.size() == 1 && prod.rhs[0] == "e")) {
                        for (int i = prod.rhs.size() - 1; i >= 0; --i) {
                            parseStack.push_back(prod.rhs[i]);
                        }
                    }
                }
                else {
                    // error
                    cout << left << setw(30) << stackStr << setw(30) <<
inputStr << "error\n";
                    break;
                }
            }
        }

        if (accept) {
            cout << "\nParsing accepted.\n";
        }
        else {
            cout << "\nParsing failed.\n";
        }
    }

    // 运行解析器
    void run() {
        readGrammar();
```

```
389         computeFirst();
390         computeFollow();
391         buildParseTable();
392         printParseTable(); // 输出预测分析表
393         parseInput();
394     }
395 
396 private:
397     string inputString; // 待分析的输入字符串
398 
399     // 辅助函数：判断是否是非终结符
400     bool isNonTerminal(const string& sym) {
401         return nonTerminals.find(sym) != nonTerminals.end();
402     }
403 
404     // 辅助函数：判断是否是终结符
405     bool isTerminal(const string& sym) {
406         return terminals.find(sym) != terminals.end();
407     }
408 
409     // 辅助函数：去除字符串首尾空白
410     string trim(const string& s) {
411         string result = s;
412         // 去除左侧空白
413         result.erase(result.begin(), find_if(result.begin(), result.end(), []
    (int ch) {
414             return !isspace(ch);
415             }));
416         // 去除右侧空白
417         result.erase(find_if(result.rbegin(), result.rend(), [](int ch) {
418             return !isspace(ch);
419             }).base(), result.end());
420         return result;
421     }
422 
423     // 辅助函数：按分隔符分割字符串
424     vector<string> split(const string& s, char delimiter) {
425         vector<string> tokens;
426         string token;
427         stringstream ss(s);
428         while (getline(ss, token, delimiter)) {
429             if (!token.empty())
430                 tokens.push_back(token);
431         }
432         return tokens;
433     }
434 
```

```
    // 计算一串符号的 First 集合
    set<string> computeFirstOfString(const vector<string>& symbols) {
        set<string> result;
        bool epsilonFound = true;
        for (auto& sym : symbols) {
            for (auto& f : First[sym]) {
                if (f != "e") {
                    result.insert(f);
                }
            }
            if (First[sym].find("e") == First[sym].end()) {
                epsilonFound = false;
                break;
            }
        }
        if (epsilonFound) {
            result.insert("e");
        }
        return result;
    }

    // 分词函数：将输入字符串转化为终结符序列
    vector<string> tokenize(const string& input) {
        vector<string> tokens;
        // 假设所有终结符都是单字符，或者特定多字符符号
        // 这里假设终结符都是单字符
        for (char c : input) {
            string token(1, c);
            tokens.push_back(token);
        }
        return tokens;
    }
};

int main() {
    LL1Parser parser;
    parser.run();
    return 0;
}
```

LR(1)扩展代码

```
#include <bits/stdc++.h>
using namespace std;

```

```cpp
// 定义产生式
struct Production {
    int id; // 产生式编号
    string lhs; // 左部非终结符
    vector<string> rhs; // 右部符号串

    Production(int identifier, string left, vector<string> right)
        : id(identifier), lhs(left), rhs(right) {
    }
};

// 定义LR(1)项目
struct LR1Item {
    string lhs;
    vector<string> rhs;
    int dot; // 点的位置
    string lookahead;
    int prodId; // 产生式编号

    LR1Item(string left, vector<string> right, int d, string la, int pid)
        : lhs(left), rhs(right), dot(d), lookahead(la), prodId(pid) {
    }

    bool operator<(const LR1Item& other) const {
        if (lhs != other.lhs)
            return lhs < other.lhs;
        if (rhs != other.rhs)
            return rhs < other.rhs;
        if (dot != other.dot)
            return dot < other.dot;
        if (lookahead != other.lookahead)
            return lookahead < other.lookahead;
        return prodId < other.prodId;
    }

    bool operator==(const LR1Item& other) const {
        return lhs == other.lhs && rhs == other.rhs && dot == other.dot &&
lookahead == other.lookahead && prodId == other.prodId;
    }
};

// 语法分析器类
class LR1Parser {
private:
    vector<Production> productions;
    set<string> nonTerminals;
    set<string> terminals;
```

```
50      string startSymbol;
51
52      // First 和 Follow 集合
53      map<string, set<string>> First;
54      map<string, set<string>> Follow;
55
56      // 项目集规范族
57      vector<set<LR1Item>> C;
58
59      // 分析表
60      // Action 表：状态 -> (终结符 -> Action)
61      // Action 的值为 "shift X", "reduce Y", "accept"
62      map<int, map<string, string>> Action;
63
64      // Goto 表：状态 -> (非终结符 -> 状态)
65      map<int, map<string, int>> GotoTable;
66
67  public:
68      // 读取文法规则
69      void readGrammar() {
70          // 输入开始符号
71          cin >> startSymbol;
72
73          // 输入非终结符集合
74          string line;
75          getline(cin, line); // 读取剩余的换行符
76          getline(cin, line);
77          vector<string> ntTokens = split(line, ' ');
78          nonTerminals = set<string>(ntTokens.begin(), ntTokens.end());
79
80          // 输入终结符集合（包括 'e' 作为 ε）
81          getline(cin, line);
82          vector<string> tTokens = split(line, ' ');
83          terminals = set<string>(tTokens.begin(), tTokens.end());
84          terminals.insert("e"); // 将 'e' 作为终结符添加
85
86          // 输入产生式数量
87          int P;
88          cin >> P;
89          getline(cin, line); // 读取剩余的换行符
90
91          // 读取产生式
92          for (int i = 0; i < P; ++i) {
93              getline(cin, line);
94              line = trim(line);
95              size_t arrow = line.find("->");
96              if (arrow == string::npos) {
```

```
                cerr << "Invalid production format: " << line << endl;
                exit(1);
            }
            string lhs = trim(line.substr(0, arrow));
            string rhsPart = trim(line.substr(arrow + 2));
            vector<string> alternatives = split(rhsPart, '|');
            for (auto& alt : alternatives) {
                alt = trim(alt);
                if (alt == "e") { // 将 'e' 视为 ε
                    productions.emplace_back(i + 1, lhs, vector<string>{ "e"
});
                }
                else {
                    vector<string> symbols = split(alt, ' ');
                    productions.emplace_back(i + 1, lhs, symbols);
                }
            }
        }

        // 读取待分析的输入字符串
        cin >> line;
        inputString = line;
    }

    // 计算 First 集合
    void computeFirst() {
        // 初始化终结符的 First 集合
        for (auto& t : terminals) {
            if (t != "e") // 'e' 作为特殊符号单独处理
                First[t].insert(t);
        }
        // 'e' 的 FIRST 集合
        First["e"].insert("e");

        // 初始化非终结符的 First 集合为空
        for (auto& nt : nonTerminals) {
            First[nt] = set<string>();
        }

        bool changed = true;
        while (changed) {
            changed = false;
            for (auto& prod : productions) {
                string A = prod.lhs;
                vector<string> alpha = prod.rhs;

                // 计算 First(alpha)
```

```
143                 set<string> firstAlpha = computeFirstOfString(alpha);
144 
145                 // 将 First(alpha) - {e} 加入 First(A)
146                 for (auto& sym : firstAlpha) {
147                     if (sym != "e" && First[A].find(sym) == First[A].end()) {
148                         First[A].insert(sym);
149                         changed = true;
150                     }
151                 }
152 
153                 // 如果 e ∈ First(alpha)，将 e 加入 First(A)
154                 if (firstAlpha.find("e") != firstAlpha.end()) {
155                     if (First[A].find("e") == First[A].end()) {
156                         First[A].insert("e");
157                         changed = true;
158                     }
159                 }
160             }
161         }
162 
163         // Debug: printFirstFollow();
164     }
165 
166     // 计算 Follow 集合
167     void computeFollow() {
168         // 初始化 Follow 集合为空
169         for (auto& nt : nonTerminals) {
170             Follow[nt] = set<string>();
171         }
172         // 开始符号的 Follow 集加入 $
173         Follow[startSymbol].insert("$");
174 
175         bool changed = true;
176         while (changed) {
177             changed = false;
178             for (auto& prod : productions) {
179                 string A = prod.lhs;
180                 vector<string> alpha = prod.rhs;
181                 for (int i = 0; i < (int)alpha.size(); i++) {
182                     string B = alpha[i];
183                     if (nonTerminals.find(B) != nonTerminals.end()) {
184                         // β = alpha[i+1 ... end]
185                         vector<string> beta(alpha.begin() + i + 1,
    alpha.end());
186                         if (beta.empty()) {
187                             // 如果 β 为空，则将 Follow(A) 加入 Follow(B)
188                             for (auto& f : Follow[A]) {
```

```
                                if (Follow[B].find(f) == Follow[B].end()) {
                                    Follow[B].insert(f);
                                    changed = true;
                                }
                            }
                        }
                        else {
                            // 计算 First(beta)
                            set<string> firstBeta = computeFirstOfString(beta);

                            // 将 First(beta) - {e} 加入 Follow(B)
                            for (auto& sym : firstBeta) {
                                if (sym != "e" && Follow[B].find(sym) ==
Follow[B].end()) {
                                    Follow[B].insert(sym);
                                    changed = true;
                                }
                            }

                            // 如果 e ∈ First(beta)，则将 Follow(A) 加入
Follow(B)
                            if (firstBeta.find("e") != firstBeta.end()) {
                                for (auto& f : Follow[A]) {
                                    if (Follow[B].find(f) == Follow[B].end()) {
                                        Follow[B].insert(f);
                                        changed = true;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }

        // Debug: printFirstFollow();
    }

    // 闭包操作
    set<LR1Item> closure(const set<LR1Item>& I) {
        set<LR1Item> closureSet = I;
        bool changed = true;

        while (changed) {
            changed = false;
            set<LR1Item> newItems;
            for (auto& item : closureSet) {
```

```
                if (item.dot < (int)item.rhs.size()) {
                    string B = item.rhs[item.dot];
                    if (nonTerminals.find(B) != nonTerminals.end()) {
                        // β = item.rhs[item.dot +1 ... end]
                        vector<string> beta(item.rhs.begin() + item.dot + 1,
item.rhs.end());
                        // a = item.lookahead
                        string a = item.lookahead;
                        // FIRST(beta a)
                        vector<string> symbols = beta;
                        symbols.push_back(a);
                        set<string> firstBetaA = computeFirstOfString(symbols);

                        for (auto& prod : productions) {
                            if (prod.lhs == B) {
                                for (auto& la : firstBetaA) {
                                    if (la == "e") continue; // 在 LR(1) 项目中,
通常不添加 lookahead 为 ε 的项目
                                    LR1Item newItem(prod.lhs, prod.rhs, 0, la,
prod.id);
                                    if (closureSet.find(newItem) ==
closureSet.end() && newItems.find(newItem) == newItems.end()) {
                                        newItems.insert(newItem);
                                        changed = true;
                                    }
                                }
                            }
                        }
                    }
                }
                closureSet.insert(newItems.begin(), newItems.end());
            }

            return closureSet;
        }

        // 迁移操作
        set<LR1Item> goto_func(const set<LR1Item>& I, const string& X) {
            set<LR1Item> J;
            for (auto& item : I) {
                if (item.dot < (int)item.rhs.size() && item.rhs[item.dot] == X) {
                    LR1Item movedItem = item;
                    movedItem.dot += 1;
                    J.insert(movedItem);
                }
            }
```

```
277         return closure(J);
278     }
279 
280     // 构建项目集规范族
281     void buildCanonicalCollection() {
282         // 初始项集 C0 = closure({ S' -> . S, $ })
283         set<LR1Item> C0;
284         // 生产式 1: S' -> E
285         C0.emplace(LR1Item(productions[0].lhs, productions[0].rhs, 0, "$",
    productions[0].id));
286         set<LR1Item> closureC0 = closure(C0);
287         C.push_back(closureC0);
288 
289         // 使用 BFS 构建项目集
290         queue<int> q;
291         q.push(0);
292 
293         while (!q.empty()) {
294             int i = q.front();
295             q.pop();
296             set<string> symbols;
297             for (auto& item : C[i]) {
298                 if (item.dot < (int)item.rhs.size()) {
299                     symbols.insert(item.rhs[item.dot]);
300                 }
301             }
302 
303             for (auto& X : symbols) {
304                 set<LR1Item> gotoI = goto_func(C[i], X);
305                 if (gotoI.empty()) continue;
306 
307                 // Check if gotoI already exists in C
308                 int j = -1;
309                 for (int k = 0; k < (int)C.size(); ++k) {
310                     if (C[k] == gotoI) {
311                         j = k;
312                         break;
313                     }
314                 }
315 
316                 if (j == -1) {
317                     C.push_back(gotoI);
318                     j = C.size() - 1;
319                     q.push(j);
320                 }
321 
322                 // 填充 Action 和 Goto 表
```

```
                if (terminals.find(X) != terminals.end()) {
                    Action[i][X] = "shift " + to_string(j);
                }
                else if (nonTerminals.find(X) != nonTerminals.end()) {
                    GotoTable[i][X] = j;
                }
            }
        }
    }

    // 构建分析表
    void buildParseTable() {
        for (int i = 0; i < (int)C.size(); ++i) {
            for (auto& item : C[i]) {
                if (item.dot < (int)item.rhs.size()) {
                    string a = item.rhs[item.dot];
                    if (terminals.find(a) != terminals.end()) {
                        // 查找 goto(Ci, a)
                        set<LR1Item> gotoSet = goto_func(C[i], a);
                        if (!gotoSet.empty()) {
                            // 查找状态 j
                            int j = -1;
                            for (int k = 0; k < (int)C.size(); ++k) {
                                if (C[k] == gotoSet) {
                                    j = k;
                                    break;
                                }
                            }
                            if (j != -1) {
                                Action[i][a] = "shift " + to_string(j);
                            }
                        }
                    }
                }
                else {
                    if (item.lhs != productions[0].lhs) {
                        // A -> α ., a
                        // Action[i, a] = reduce prod.id
                        Action[i][item.lookahead] = "reduce " +
to_string(item.prodId);
                    }
                    else {
                        // S' -> S ., $
                        if (item.lookahead == "$") {
                            Action[i][item.lookahead] = "accept";
                        }
                    }
```

```
                }
            }
        }
    }

    // 解析输入字符串并输出分析过程
    void parseInput() {
        // 分词：将输入字符串按字符拆分
        vector<string> inputTokens = tokenize(inputString);
        inputTokens.push_back("$"); // 末尾加入 $

        // 初始化解析栈
        vector<int> parseStack;
        parseStack.push_back(0);

        int ip = 0; // 输入指针
        bool accept = false;

        // 存储输出动作
        vector<string> actions;

        while (true) {
            int state = parseStack.back();
            string a = inputTokens[ip];

            // 查找 Action[state][a]
            if (Action.find(state) != Action.end() && Action[state].find(a) != Action[state].end()) {
                string action = Action[state][a];
                if (action.substr(0, 5) == "shift") {
                    actions.push_back("shift");
                    // 获取状态 j
                    int j = stoi(action.substr(6));
                    parseStack.push_back(j);
                    ip++;
                }
                else if (action.substr(0, 6) == "reduce") {
                    // 获取生产式编号
                    int prodId = stoi(action.substr(7));
                    actions.push_back(to_string(prodId-1));
                    // 注意：prodId 是从1开始的
                    if (prodId <= 0 || prodId > (int)productions.size()) {
                        cerr << "Error: Invalid production ID " << prodId << ".\n";
                        break;
                    }
```

```
413                     Production prod = productions[prodId - 1]; // 产生式编号从1开始
414 
415                     // 弹出 rhs.size() 个状态
416                     for (size_t k = 0; k < prod.rhs.size(); ++k) {
417                         if (!parseStack.empty())
418                             parseStack.pop_back();
419                         else {
420                             cerr << "Error: Stack underflow during reduction.\n";
421                             break;
422                         }
423                     }
424                     // 获取当前状态
425                     if (parseStack.empty()) {
426                         cerr << "Error: Stack is empty after reduction.\n";
427                         break;
428                     }
429                     int currentState = parseStack.back();
430                     // Goto[currentState][A] = j
431                     if (GotoTable.find(currentState) != GotoTable.end() && GotoTable[currentState].find(prod.lhs) != GotoTable[currentState].end()) {
432                         int j = GotoTable[currentState][prod.lhs];
433                         parseStack.push_back(j);
434                     }
435                     else {
436                         cerr << "Error: Goto table entry not found for state " << currentState << " and non-terminal " << prod.lhs << ".\n";
437                         break;
438                     }
439                 }
440                 else if (action == "accept") {
441                     actions.push_back("accept");
442                     accept = true;
443                     break;
444                 }
445             }
446             else {
447                 // 查找 Action[state][a] 不存在，解析错误
448                 actions.push_back("error");
449                 break;
450             }
451         }
452 
453         // 输出动作
454         cout << "Parsing Actions:\n";
455         for (auto& act : actions) {
```

```
456             cout << act << "\n";
457         }
458 
459         if (accept) {
460             cout << "Parsing accepted.\n";
461         }
462         else {
463             cout << "Parsing failed.\n";
464         }
465     }
466 
467     // 辅助函数：判断是否是非终结符
468     bool isNonTerminal(const string& sym) {
469         return nonTerminals.find(sym) != nonTerminals.end();
470     }
471 
472     // 辅助函数：去除字符串首尾空白
473     string trim(const string& s) {
474         string result = s;
475         // 去除左侧空白
476         result.erase(result.begin(), find_if(result.begin(), result.end(), []
    (int ch) {
477             return !isspace(ch);
478             }));
479         // 去除右侧空白
480         result.erase(find_if(result.rbegin(), result.rend(), [](int ch) {
481             return !isspace(ch);
482             }).base(), result.end());
483         return result;
484     }
485 
486     // 辅助函数：按分隔符分割字符串
487     vector<string> split(const string& s, char delimiter) {
488         vector<string> tokens;
489         string token;
490         stringstream ss(s);
491         while (getline(ss, token, delimiter)) {
492             if (!token.empty())
493                 tokens.push_back(token);
494         }
495         return tokens;
496     }
497 
498     // 计算一串符号的 First 集合
499     set<string> computeFirstOfString(const vector<string>& symbols) {
500         set<string> result;
501         bool epsilonFound = true;
```

```
        for (auto& sym : symbols) {
            for (auto& f : First[sym]) {
                if (f != "e") {
                    result.insert(f);
                }
            }
            if (First[sym].find("e") == First[sym].end()) {
                epsilonFound = false;
                break;
            }
        }
        if (epsilonFound) {
            result.insert("e");
        }
        return result;
    }

    // 将符号串转为字符串（用于映射）
    string stringify(const vector<string>& symbols) {
        string s;
        for (auto& sym : symbols) {
            s += sym + " ";
        }
        return s;
    }

    // 分词函数：将输入字符串转化为终结符序列
    vector<string> tokenize(const string& input) {
        vector<string> tokens;
        // 假设所有终结符都是单字符
        for (char c : input) {
            string token(1, c);
            tokens.push_back(token);
        }
        return tokens;
    }

    // 打印项目集规范族（调试用）
    void printCanonicalCollection() {
        cout << "\nCanonical Collection of LR(1) Items:\n";
        for (int i = 0; i < (int)C.size(); ++i) {
            cout << "C" << i << ":\n";
            for (auto& item : C[i]) {
                cout << "  " << item.lhs << " -> ";
                for (int j = 0; j < (int)item.rhs.size(); ++j) {
                    if (j == item.dot)
                        cout << ". ";
```

```
549                     cout << item.rhs[j] << " ";
550                 }
551                 if (item.dot == (int)item.rhs.size())
552                     cout << ". ";
553                 cout << ", " << item.lookahead << "\n";
554             }
555             cout << "\n";
556         }
557     }
558 
559     // 打印 First 和 Follow 集合（调试用）
560     void printFirstFollow() {
561         cout << "\nFIRST sets:\n";
562         for (auto& pair : First) {
563             cout << pair.first << ": { ";
564             for (auto& sym : pair.second) {
565                 cout << sym << " ";
566             }
567             cout << "}\n";
568         }
569 
570         cout << "\nFOLLOW sets:\n";
571         for (auto& pair : Follow) {
572             cout << pair.first << ": { ";
573             for (auto& sym : pair.second) {
574                 cout << sym << " ";
575             }
576             cout << "}\n";
577         }
578     }
579 
580     // 打印分析表（改进版）
581     void printParseTable() {
582         // 打印 Action 表
583         cout << "\nAction Table:\n";
584 
585         // 收集所有终结符（包括 $）
586         vector<string> termList;
587         for (const auto& t : terminals) {
588             if (t != "e") // 'e' 作为特殊符号单独处理
589                 termList.push_back(t);
590         }
591         termList.push_back("$"); // 添加结束符
592 
593         // 打印表头
594         cout << "State\t";
595         for (auto& term : termList) {
```

```
596             cout << term << "\t";
597         }
598         cout << "\n";
599 
600         // 打印每个状态的 Action 表项
601         for (int i = 0; i < (int)C.size(); ++i) {
602             cout << i << "\t";
603             for (auto& term : termList) {
604                 if (Action.find(i) != Action.end() && Action[i].find(term) != Action[i].end()) {
605                     cout << Action[i][term] << "\t";
606                 }
607                 else {
608                     cout << "\t";
609                 }
610             }
611             cout << "\n";
612         }
613 
614         // 打印 Goto 表
615         cout << "\nGoto Table:\n";
616 
617         // 收集所有非终结符
618         vector<string> ntList(nonTerminals.begin(), nonTerminals.end());
619 
620         // 打印表头
621         cout << "State\t";
622         for (auto& nt : ntList) {
623             cout << nt << "\t";
624         }
625         cout << "\n";
626 
627         // 打印每个状态的 Goto 表项
628         for (int i = 0; i < (int)C.size(); ++i) {
629             cout << i << "\t";
630             for (auto& nt : ntList) {
631                 if (GotoTable.find(i) != GotoTable.end() && GotoTable[i].find(nt) != GotoTable[i].end()) {
632                     cout << GotoTable[i][nt] << "\t";
633                 }
634                 else {
635                     cout << "\t";
636                 }
637             }
638             cout << "\n";
639         }
640     }
```

```
641
642     // 运行解析器
643     void run() {
644         readGrammar();
645         computeFirst();
646         computeFollow();
647         buildCanonicalCollection();
648         buildParseTable();
649         printParseTable(); // 输出解析表
650         parseInput();
651     }
652
653 private:
654     string inputString; // 待分析的输入字符串
655 };
656
657 int main() {
658     LR1Parser parser;
659     parser.run();
660     return 0;
661 }
```