编译原理与技术

LR 语法分析程序设计报告

姓　　名：刘立敏

学　　号：2018211398

学　　院：计算机学院

专　　业：计算机科学与技术

班　　级：2018211308

指导老师：张玉洁

目录

1、实验内容

编写语法分析程序，实现对算术表达式的语法分析。要求所分析的算术表达式由如下的文法产生：

E → E+T ｜ E - T ｜ T

T → T*F ｜ T/F ｜ F

F → (E) ｜ num

编写语法分析程序实现自底向上的分析，要求如下：

（1）构造识别该文法所有活前缀的 DFA

（2）构造该文法的 LR 分析表。

（3）编程实现算法 4.3，构造 LR 分析程序。

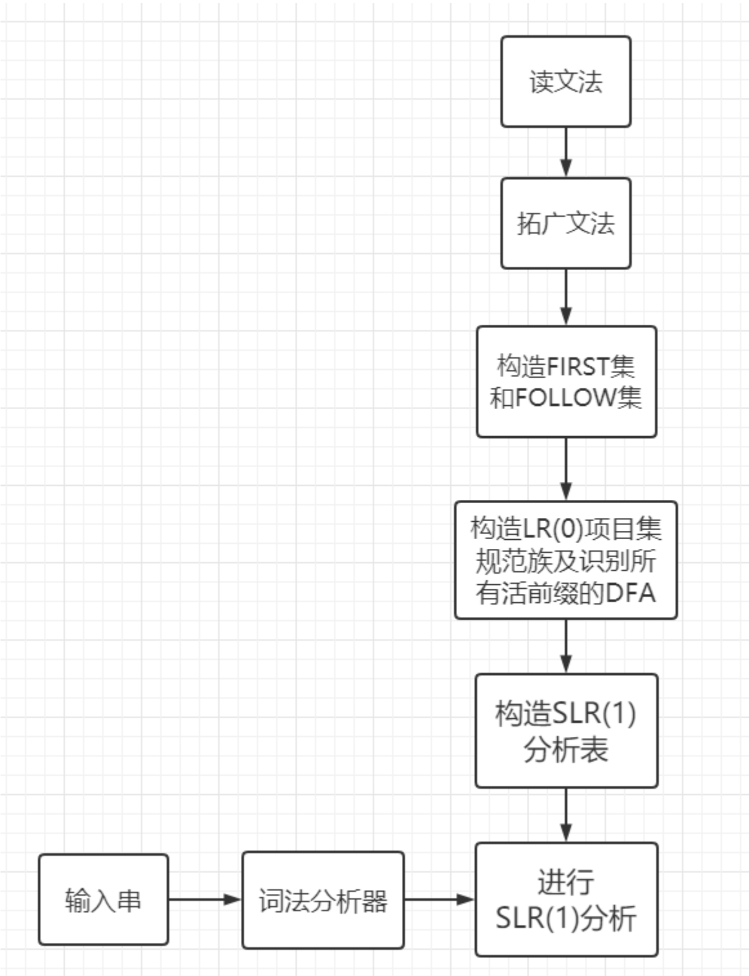
2、实验环境

编程语言：C++

集成开发环境：Visual Studio 2019

3、总体设计

程序总体的流程图如下：



3.1、拓广文法

引入新的非终结符 E’,将原文法拓广为

（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

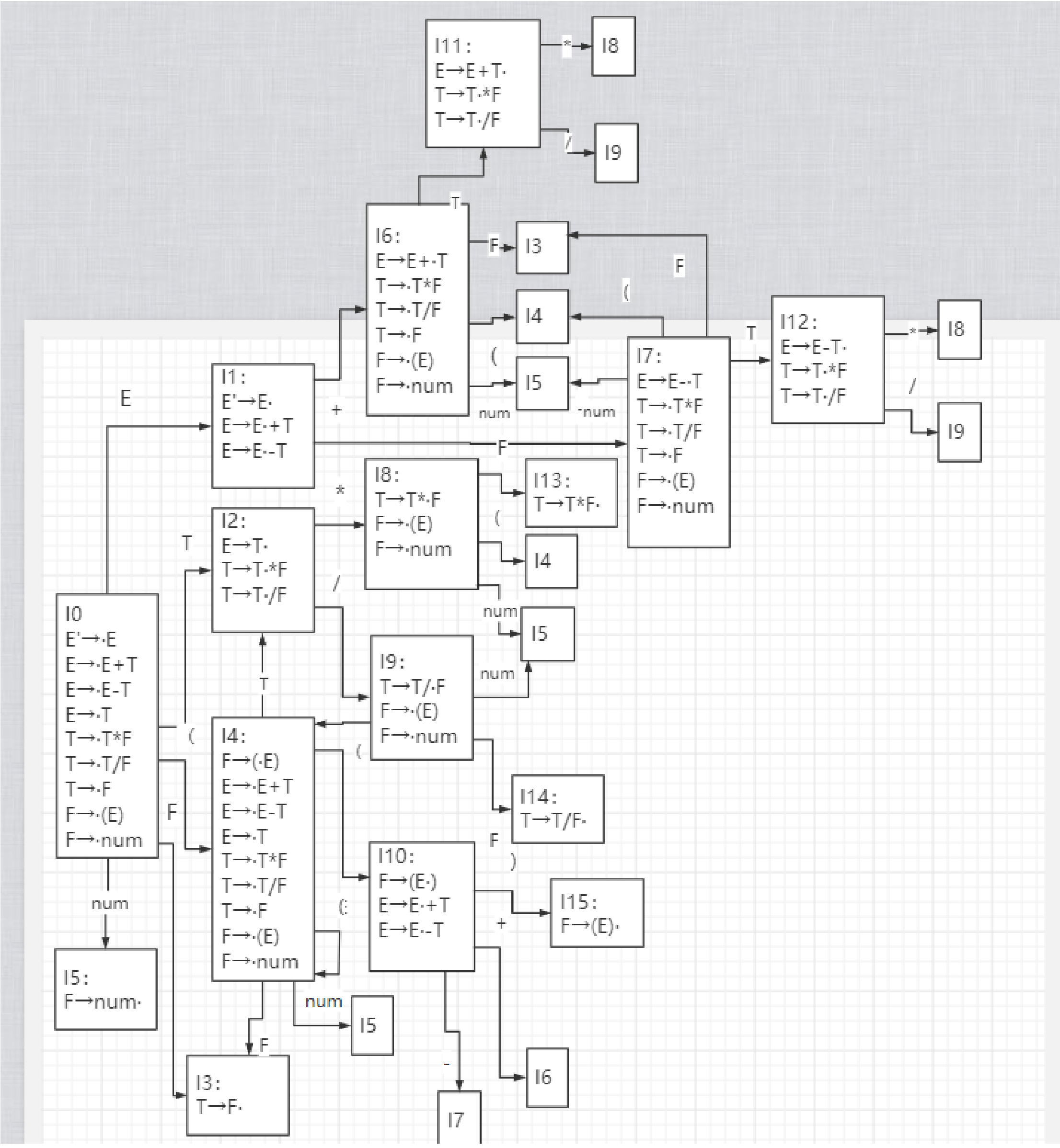
（7）F → (E)

（8）F → num

3.2、计算 FIRST 集和 FOLLOW 集

	FIRST	FOLLOW
E’	(num	$
E	(num	$ + -)
T	(num	$ + -) * /
F	(num	$ + -) * /

3.3、构造 LR(0)项目集规范族及识别所有活前缀的 DFA



3.4、构造 SLR(1)分析表

action 表

	+	-	*	/	(	)	num	$
0					s4		s5	
1	s6	s7						acc
2	r3	r3	s8	s9		r3		r3
3	r6	r6	r6	r6		r6		r6

4					s4		s5	
5	r8	r8	r8	r8		r8		r8
6					s4		s5	
7					s4		s5	
8					s4		s5	
9					s4		s5	
10	s6	s7				s15		
11	r1	r1	s8	s9		r1		r1
12	r2	r2	s8	s9		r2		r2
13	r4	r4	r4	r4		r4		r4
14	r5	r5	r5	r5		r5		r5
15	r7	r7	r7	r7		r7		r7

goto 表

	E	T	F
0	1	2	3
1			
2			
3			
4	10	2	3
5			
6		11	3
7		12	3
8			13

9			14
10			
11			
12			
13			
14			
15			

3.5、构造 SLR(1)分析程序

SLR(1)分析程序使用一个符号栈，一个状态栈和一个输入串。分析时，根据当前栈顶符号和输入串首字符查询 SLR(1)分析表，以确定下一步进行的动作。

4、详细设计

4.1、输入

输入分为两部分，一是文法输入，二是字符串输入。两者均采用文件读取的方式（grammer.txt 和 input.txt）。

文法采用课本上给出的文法：

```
#nonterminals
E T F

#terminals
+ - * / ( ) num

#startSymbol
E

#productions
E -> E + T | E - T | T
T -> T * F | T / F | F
F -> ( E ) | num
```

输入的字符串在文件 input.txt 中。程序读取字符串后，将其交给词法分析器处理，得到 token 流。然后语法分析器再利用 SLR(1)分析表分析该 token 流。

4.2、输出

控制台方式输出。

输出分为三部分：拓广后的文法、SLR(1)分析表、SLR(1)分析过程。

示例如下：

```
Extended Grammer:
E' -> E
E -> E+T
E -> E-T
E -> T
T -> T*F
T -> T/F
T -> F
F -> (E)
F -> num
```

SLR(1) Analysis Table:

	+	-	*	/	(	)	num	$	E	T	F
0					s4		s5		1	2	3
1	s6	s7						acc			
2	r3	r3	s8	s9		r3		r3			
3	r6	r6	r6	r6		r6		r6			
4					s4		s5		10	2	3
5	r8	r8	r8	r8		r8		r8			
6					s4		s5			11	3
7					s4		s5			12	3
8					s4		s5				13
9					s4		s5				14
10	s6	s7				s15					
11	r1	r1	s8	s9		r1		r1			
12	r2	r2	s8	s9		r2		r2			
13	r4	r4	r4	r4		r4		r4			
14	r5	r5	r5	r5		r5		r5			
15	r7	r7	r7	r7		r7		r7			

```
Analysis:

1:
State Stack:   0
Symbol Stack:  $
Input String:  1 $
action:        shift 5

2:
State Stack:   0 5
Symbol Stack:  $ num
Input String:  $
action:        reduce F -> num

3:
State Stack:   0 3
Symbol Stack:  $ F
Input String:  $
action:        reduce T -> F

4:
State Stack:   0 2
Symbol Stack:  $ T
Input String:  $
action:        reduce E -> T

5:
State Stack:   0 1
Symbol Stack:  $ E
Input String:  $
action:        accept
```

4.3、文法

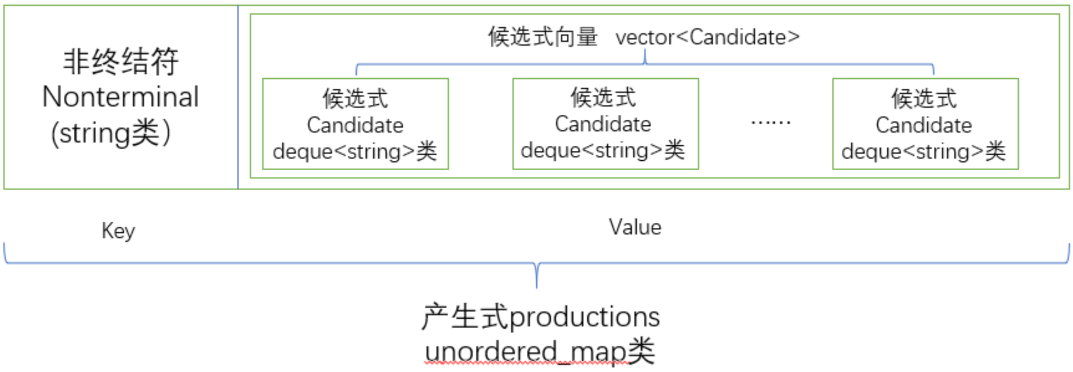
对于一个文法，我们需要存储它的非终结符、终结符、起始符和产生式信息。非终结符一般为单个的大写字母，但考虑到可以形如”E’”(修改文法后导致带有一撇)，因此每个非终结符用 C++的 string 类存储，所有的非终结符组成一个 string 类的容器 vector。终结符和起始符同理。

产生式的数据结构相对比较复杂。一个产生式的左边是非终结符，右边是（一个或多个）候选式。我们先定义候选式的结构如下：

```
typedef deque<string> Candidate;
```

一个候选式可能由多个终结符或非终结符组成，每个符号是一个 string 类，因此候选式的结构被定义为一个由 string 类元素构成的双端队列 deque，之所以选择双端队列，是因为便于我们加入或删除队列头部或尾部的元素。

一个非终结符对应多个候选式，这些候选式组成一个容器 vector<Candidate>，作为非终结符(key)对应的值（value)。因此产生式的整体结构是一个 map，为保证顺序，我们选用 unordered_map 来替代 map。



综上，文法存储结构为：

```
typedef string Nonterminal,Terminal;
typedef deque<string> Candidate;
class Parser
{
private:
    vector<Nonterminal>nonterminals;                    //非终结符
    vector<Terminal>terminals;                          //终结符
    Nonterminal startSymbol;                            //起始符
    unordered_map<Nonterminal,vector<Candidate>>productions;//产生式
    ……
};
```

4.4、SLR(1)分析表

分析表的数据结构如下所示：

map<pair<int, string>, string> action;　　　　　　//action 表

map<pair<int, string>, string> goTo;　　　　　　　//goto 表

可以看到，分析表被拆分成为两部分，其中 action 表中的 pair<int,string>是代表当前状态（即状态栈顶元素）和当前输入串的首字符，两者决定了一个 string（采取的动作）。

分析表的构建函数为：

void Parser::constructTable()

4.5、SLR(1)分析程序

```
stack<string>symbolStack;        //符号栈
stack<string>stateStack;         //状态栈
deque<Token>inputString;         //输入串
```

分析程序用到了以上三个数据结构。

在分析过程中，我们用 bool 型变量 acc 和 err 标记是否分析完毕或产生错误。分析的每一步里，我们取出状态栈顶的元素和输入串的首字符，根据分析表找到对应的动作。

动作对应四种情况：

①acc：表示分析完毕，置变量 acc 为 true，退出分析。

②空：分析表中没有对应表项，说明分析出错，输入串不是该 SLR(1)文法的句子。

③s：移进项目，将输入串首字符移进符号栈，同时进行状态转移。

④r:归约项目，对符号栈中的对应元素进行归约，同时更新状态栈。

分析程序如下：

```
void Parser::analyze(deque<Token> input)
{
    cout << "\n\nAnalysis:\n";

    bool acc = false;           //标记是否分析完毕
    bool err = false;           //标记是否出错

    stateStack.push("0");       //初始化栈和双端队列
    symbolStack.push("$");
    inputString = input;
    Token token;
    token.str = "$";
    inputString.push_back(token);

    int cnt = 0;                //步骤数
```

```
while (!acc && !err)      //分析未结束，未出错
{
    cout << endl<< ++cnt << ":" << endl;
    printStack();         //打印当前栈内容
    printInputString();       //打印当前输入串内容

    //从分析表中获取所需采取的动作
    int num1 = stoi(stateStack.top());
    if (inputString.front().tokenType == NUMERIC_CONSTANTS)
        inputString.front().str = "num";
    string act = action[make_pair(num1, inputString.front().str)];

    if (act == "acc")         //接受
    {
        acc = true;
        cout << setw(15) << "action: "<<"accept\n";
    }
    else if (act == "")           //错误
    {
        err = true;
        cout << setw(15) << "action: "<<"error\n";
        return;
    }
    else if (act[0] == 's')       //移进
    {
        int state = stoi(act.substr(1));
        stateStack.push(act.substr(1));
        string s = inputString.front().str;
        symbolStack.push(s);
        inputString.pop_front();
        cout << setw(15) << "action:  " <<"shift "<< state << endl;
    }
    else if(act[0]=='r')      //归约
    {
        int num = stoi(act.substr(1));
        auto tmp = num2pro[num];
        cout << setw(15) << "action: "<<"reduce " << tmp.first << " -> ";
        for (auto r : tmp.second)
            cout << r;
        cout << endl;

        int size = tmp.second.size();
        while (size--)
        {
            stateStack.pop();
            symbolStack.pop();
        }
        symbolStack.push(tmp.first);
```

```
            stateStack.push(goTo[make_pair(stoi(stateStack.top()), symbolStack.top())]);
        }
    }
}
```

5、测试

样例 1：1+2

```
1:
State Stack:    0
Symbol Stack:   $
Input String:   1 + 2 $
action:         shift 5

2:
State Stack:    0 5
Symbol Stack:   $ num
Input String:   + 2 $
action:         reduce F -> num

3:
State Stack:    0 3
Symbol Stack:   $ F
Input String:   + 2 $
action:         reduce T -> F

4:
State Stack:    0 2
Symbol Stack:   $ T
Input String:   + 2 $
action:         reduce E -> T

5:
State Stack:    0 1
Symbol Stack:   $ E
Input String:   + 2 $
action:         shift 6

6:
State Stack:    0 1 6
Symbol Stack:   $ E +
Input String:   2 $
action:         shift 5

7:
State Stack:    0 1 6 5
```

```
Symbol Stack:  $ E + num
Input String:  $
action:        reduce F -> num

8:
State Stack:   0 1 6 3
Symbol Stack:  $ E + F
Input String:  $
action:        reduce T -> F

9:
State Stack:   0 1 6 11
Symbol Stack:  $ E + T
Input String:  $
action:        reduce E -> E+T

10:
State Stack:   0 1
Symbol Stack:  $ E
Input String:  $
action:        accept
```

样例 2：3*4+3-5/6

```
1:
State Stack:   0
Symbol Stack:  $
Input String:  3 * 4 + 3 - 5 / 6 $
action:        shift 5

2:
State Stack:   0 5
Symbol Stack:  $ num
Input String:  * 4 + 3 - 5 / 6 $
action:        reduce F -> num

3:
State Stack:   0 3
Symbol Stack:  $ F
Input String:  * 4 + 3 - 5 / 6 $
action:        reduce T -> F

4:
State Stack:   0 2
Symbol Stack:  $ T
Input String:  * 4 + 3 - 5 / 6 $
action:        shift 8

5:
```

```
State Stack:   0 2 8
Symbol Stack:  $ T *
Input String:  4 + 3 - 5 / 6 $
action:        shift 5

6:
State Stack:   0 2 8 5
Symbol Stack:  $ T * num
Input String:  + 3 - 5 / 6 $
action:        reduce F -> num

7:
State Stack:   0 2 8 13
Symbol Stack:  $ T * F
Input String:  + 3 - 5 / 6 $
action:        reduce T -> T*F

8:
State Stack:   0 2
Symbol Stack:  $ T
Input String:  + 3 - 5 / 6 $
action:        reduce E -> T

9:
State Stack:   0 1
Symbol Stack:  $ E
Input String:  + 3 - 5 / 6 $
action:        shift 6

10:
State Stack:   0 1 6
Symbol Stack:  $ E +
Input String:  3 - 5 / 6 $
action:        shift 5

11:
State Stack:   0 1 6 5
Symbol Stack:  $ E + num
Input String:  - 5 / 6 $
action:        reduce F -> num

12:
State Stack:   0 1 6 3
Symbol Stack:  $ E + F
Input String:  - 5 / 6 $
action:        reduce T -> F

13:
State Stack:   0 1 6 11
```

```
Symbol Stack:  $ E + T
Input String:  - 5 / 6 $
action:        reduce E -> E+T

14:
State Stack:   0 1
Symbol Stack:  $ E
Input String:  - 5 / 6 $
action:        shift 7

15:
State Stack:   0 1 7
Symbol Stack:  $ E -
Input String:  5 / 6 $
action:        shift 5

16:
State Stack:   0 1 7 5
Symbol Stack:  $ E - num
Input String:  / 6 $
action:        reduce F -> num

17:
State Stack:   0 1 7 3
Symbol Stack:  $ E - F
Input String:  / 6 $
action:        reduce T -> F

18:
State Stack:   0 1 7 12
Symbol Stack:  $ E - T
Input String:  / 6 $
action:        shift 9

19:
State Stack:   0 1 7 12 9
Symbol Stack:  $ E - T /
Input String:  6 $
action:        shift 5

20:
State Stack:   0 1 7 12 9 5
Symbol Stack:  $ E - T / num
Input String:  $
action:        reduce F -> num

21:
State Stack:   0 1 7 12 9 14
Symbol Stack:  $ E - T / F
```

Input String: $
action: reduce T -> T/F

22:
State Stack: 0 1 7 12
Symbol Stack: $ E - T
Input String: $
action: reduce E -> E-T

23:
State Stack: 0 1
Symbol Stack: $ E
Input String: $
action: accept

样例 3：(((1))*(((1))))

1:
State Stack: 0
Symbol Stack: $
Input String: (((1)) * (((1)))) $
action: shift 4

2:
State Stack: 0 4
Symbol Stack: $ (
Input String: ((1)) * (((1)))) $
action: shift 4

3:
State Stack: 0 4 4
Symbol Stack: $ ((
Input String: (1)) * (((1)))) $
action: shift 4

4:
State Stack: 0 4 4 4
Symbol Stack: $ (((
Input String: 1)) * (((1)))) $
action: shift 5

5:
State Stack: 0 4 4 4 5
Symbol Stack: $ (((num
Input String:)) * (((1)))) $
action: reduce F -> num

6:
State Stack: 0 4 4 4 3

```
Symbol Stack:  $ ( ( ( F
Input String:  ) ) * ( ( ( 1 ) ) ) ) $
action:        reduce T -> F

7:
State Stack:   0 4 4 4 2
Symbol Stack:  $ ( ( ( T
Input String:  ) ) * ( ( ( 1 ) ) ) ) $
action:        reduce E -> T

8:
State Stack:   0 4 4 4 10
Symbol Stack:  $ ( ( ( E
Input String:  ) ) * ( ( ( 1 ) ) ) ) $
action:        shift 15

9:
State Stack:   0 4 4 4 10 15
Symbol Stack:  $ ( ( ( E )
Input String:  ) * ( ( ( 1 ) ) ) ) $
action:        reduce F -> (E)

10:
State Stack:   0 4 4 3
Symbol Stack:  $ ( ( F
Input String:  ) * ( ( ( 1 ) ) ) ) $
action:        reduce T -> F

11:
State Stack:   0 4 4 2
Symbol Stack:  $ ( ( T
Input String:  ) * ( ( ( 1 ) ) ) ) $
action:        reduce E -> T

12:
State Stack:   0 4 4 10
Symbol Stack:  $ ( ( E
Input String:  ) * ( ( ( 1 ) ) ) ) $
action:        shift 15

13:
State Stack:   0 4 4 10 15
Symbol Stack:  $ ( ( E )
Input String:  * ( ( ( 1 ) ) ) ) $
action:        reduce F -> (E)

14:
State Stack:   0 4 3
Symbol Stack:  $ ( F
```

```
Input String:  * ( ( ( 1 ) ) ) ) $
action:        reduce T -> F

15:
State Stack:   0 4 2
Symbol Stack:  $ ( T
Input String:  * ( ( ( 1 ) ) ) ) $
action:        shift 8

16:
State Stack:   0 4 2 8
Symbol Stack:  $ ( T *
Input String:  ( ( ( 1 ) ) ) ) $
action:        shift 4

17:
State Stack:   0 4 2 8 4
Symbol Stack:  $ ( T * (
Input String:  ( ( 1 ) ) ) ) $
action:        shift 4

18:
State Stack:   0 4 2 8 4 4
Symbol Stack:  $ ( T * ( (
Input String:  ( 1 ) ) ) ) $
action:        shift 4

19:
State Stack:   0 4 2 8 4 4 4
Symbol Stack:  $ ( T * ( ( (
Input String:  1 ) ) ) ) $
action:        shift 5

20:
State Stack:   0 4 2 8 4 4 4 5
Symbol Stack:  $ ( T * ( ( ( num
Input String:  ) ) ) ) $
action:        reduce F -> num

21:
State Stack:   0 4 2 8 4 4 4 3
Symbol Stack:  $ ( T * ( ( ( F
Input String:  ) ) ) ) $
action:        reduce T -> F

22:
State Stack:   0 4 2 8 4 4 4 2
Symbol Stack:  $ ( T * ( ( ( T
Input String:  ) ) ) ) $
```

```
action:        reduce E -> T

23:
State Stack:   0 4 2 8 4 4 4 10
Symbol Stack:  $ ( T * ( ( ( E
Input String:  ) ) ) ) $
action:        shift 15

24:
State Stack:   0 4 2 8 4 4 4 10 15
Symbol Stack:  $ ( T * ( ( ( E )
Input String:  ) ) ) $
action:        reduce F -> (E)

25:
State Stack:   0 4 2 8 4 4 3
Symbol Stack:  $ ( T * ( ( F
Input String:  ) ) ) $
action:        reduce T -> F

26:
State Stack:   0 4 2 8 4 4 2
Symbol Stack:  $ ( T * ( ( T
Input String:  ) ) ) $
action:        reduce E -> T

27:
State Stack:   0 4 2 8 4 4 10
Symbol Stack:  $ ( T * ( ( E
Input String:  ) ) ) $
action:        shift 15

28:
State Stack:   0 4 2 8 4 4 10 15
Symbol Stack:  $ ( T * ( ( E )
Input String:  ) ) $
action:        reduce F -> (E)

29:
State Stack:   0 4 2 8 4 3
Symbol Stack:  $ ( T * ( F
Input String:  ) ) $
action:        reduce T -> F

30:
State Stack:   0 4 2 8 4 2
Symbol Stack:  $ ( T * ( T
Input String:  ) ) $
action:        reduce E -> T
```

31:
State Stack: 0 4 2 8 4 10
Symbol Stack: $ (T * (E
Input String:)) $
action: shift 15

32:
State Stack: 0 4 2 8 4 10 15
Symbol Stack: $ (T * (E)
Input String:) $
action: reduce F -> (E)

33:
State Stack: 0 4 2 8 13
Symbol Stack: $ (T * F
Input String:) $
action: reduce T -> T*F

34:
State Stack: 0 4 2
Symbol Stack: $ (T
Input String:) $
action: reduce E -> T

35:
State Stack: 0 4 10
Symbol Stack: $ (E
Input String:) $
action: shift 15

36:
State Stack: 0 4 10 15
Symbol Stack: $ (E)
Input String: $
action: reduce F -> (E)

37:
State Stack: 0 3
Symbol Stack: $ F
Input String: $
action: reduce T -> F

38:
State Stack: 0 2
Symbol Stack: $ T
Input String: $
action: reduce E -> T

39:
State Stack: 0 1
Symbol Stack: $ E
Input String: $
action: accept

测试 4（括号未闭合）：(((3.14))

1:
State Stack: 0
Symbol Stack: $
Input String: (((3.14)) $
action: shift 4

2:
State Stack: 0 4
Symbol Stack: $ (
Input String: ((3.14)) $
action: shift 4

3:
State Stack: 0 4 4
Symbol Stack: $ ((
Input String: (3.14)) $
action: shift 4

4:
State Stack: 0 4 4 4
Symbol Stack: $ (((
Input String: 3.14)) $
action: shift 5

5:
State Stack: 0 4 4 4 5
Symbol Stack: $ (((num
Input String:)) $
action: reduce F -> num

6:
State Stack: 0 4 4 4 3
Symbol Stack: $ (((F
Input String:)) $
action: reduce T -> F

7:
State Stack: 0 4 4 4 2
Symbol Stack: $ (((T
Input String:)) $
action: reduce E -> T

8:
State Stack: 0 4 4 4 10
Symbol Stack: $ (((E
Input String:)) $
action: shift 15

9:
State Stack: 0 4 4 4 10 15
Symbol Stack: $ (((E)
Input String:) $
action: reduce F -> (E)

10:
State Stack: 0 4 4 3
Symbol Stack: $ ((F
Input String:) $
action: reduce T -> F

11:
State Stack: 0 4 4 2
Symbol Stack: $ ((T
Input String:) $
action: reduce E -> T

12:
State Stack: 0 4 4 10
Symbol Stack: $ ((E
Input String:) $
action: shift 15

13:
State Stack: 0 4 4 10 15
Symbol Stack: $ ((E)
Input String: $
action: reduce F -> (E)

14:
State Stack: 0 4 3
Symbol Stack: $ (F
Input String: $
action: reduce T -> F

15:
State Stack: 0 4 2
Symbol Stack: $ (T
Input String: $
action: reduce E -> T

16:
State Stack: 0 4 10
Symbol Stack: $ (E
Input String: $
action: error

样例 5（缺少运算数）：(6+1-)*3+4

1:
State Stack: 0
Symbol Stack: $
Input String: (6 + 1 -) * 3 + 4 $
action: shift 4

2:
State Stack: 0 4
Symbol Stack: $ (
Input String: 6 + 1 -) * 3 + 4 $
action: shift 5

3:
State Stack: 0 4 5
Symbol Stack: $ (num
Input String: + 1 -) * 3 + 4 $
action: reduce F -> num

4:
State Stack: 0 4 3
Symbol Stack: $ (F
Input String: + 1 -) * 3 + 4 $
action: reduce T -> F

5:
State Stack: 0 4 2
Symbol Stack: $ (T
Input String: + 1 -) * 3 + 4 $
action: reduce E -> T

6:
State Stack: 0 4 10
Symbol Stack: $ (E
Input String: + 1 -) * 3 + 4 $
action: shift 6

7:
State Stack: 0 4 10 6
Symbol Stack: $ (E +
Input String: 1 -) * 3 + 4 $

action: shift 5

8:
State Stack: 0 4 10 6 5
Symbol Stack: $ (E + num
Input String: -) * 3 + 4 $
action: reduce F -> num

9:
State Stack: 0 4 10 6 3
Symbol Stack: $ (E + F
Input String: -) * 3 + 4 $
action: reduce T -> F

10:
State Stack: 0 4 10 6 11
Symbol Stack: $ (E + T
Input String: -) * 3 + 4 $
action: reduce E -> E+T

11:
State Stack: 0 4 10
Symbol Stack: $ (E
Input String: -) * 3 + 4 $
action: shift 7

12:
State Stack: 0 4 10 7
Symbol Stack: $ (E -
Input String:) * 3 + 4 $
action: error

6、总结

6.1、程序实现的效果

该 LR 语法分析程序实现了题目所要求的全部功能：根据给出的文法构造 DFA 和分析表，然后对输入串进行 SLR(1)分析。

6.2、设计亮点和缺点

亮点在于数据结构、类的定义和接口设计得比较合理，各模块划分清晰。缺点在于算法的实现上。算法有些地方不够优化，复杂度比较高。后续我会尽力优化的改进。

6.3、实验心得

有了前两次实验的基础，本次LR语法分析的设计相对而言比较轻松。通过这次实验，我已经能熟练掌握LR语法分析中的每一个步骤，大大加深了我对语法分析的理解。通过亲自动手实践，我明白了LR语法分析所需要注意的各种细节。总之，这次实验给我带来了很大的收获。今后我也将进行更加深入的学习，尽可能地将所学知识用于以后的学习和工作当中去。