编译原理实验报告

语法分析程序的设计与实现

姓名：胡敏臻

班级：2019211307

学号：2019211424

日期：2021/12/1

1 概述

1.1 实验内容

编写语法分析程序，实现对算术表达式的语法分析。要求所分析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；

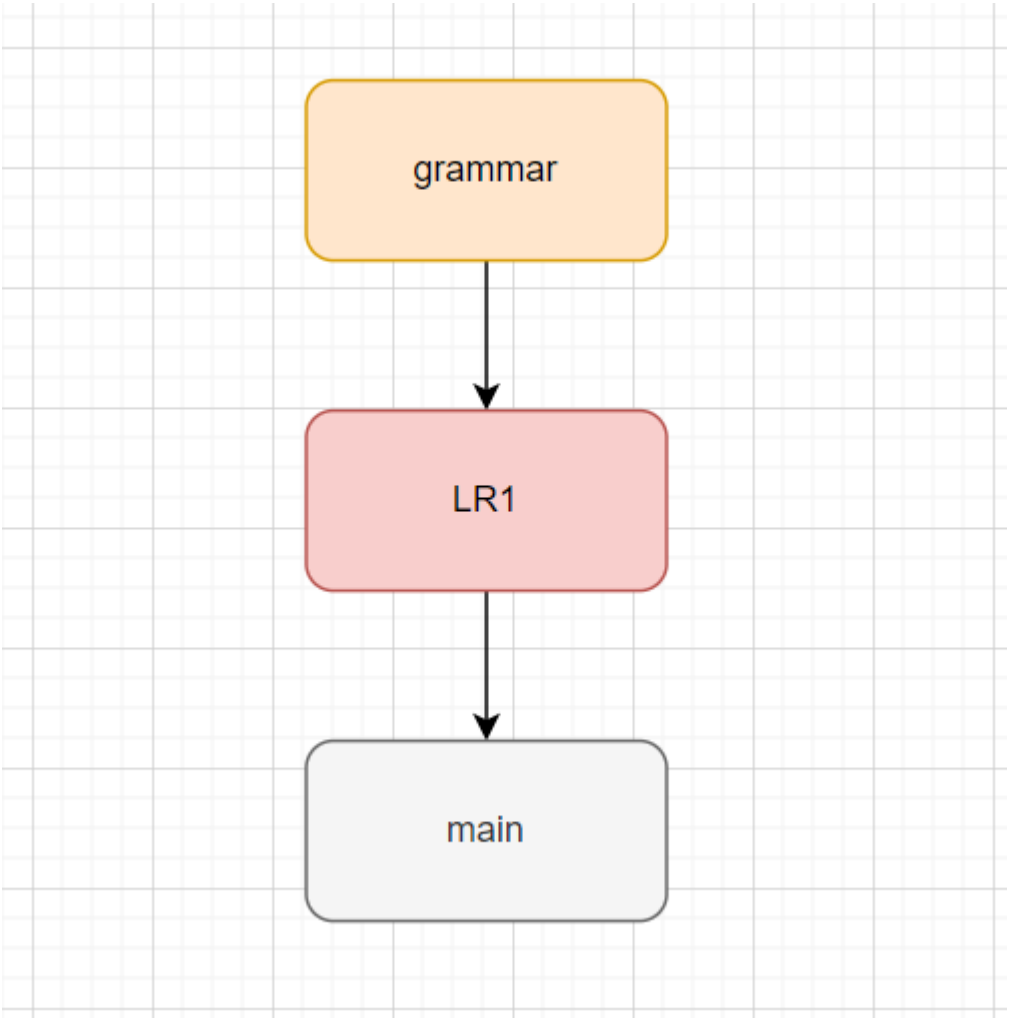
(2)构造该文法的 LR 分析表；

(3)编程实现算法 4.3，构造 LR 分析程序。

1.2 实验环境

Visual Studio 2019

2 总体设计



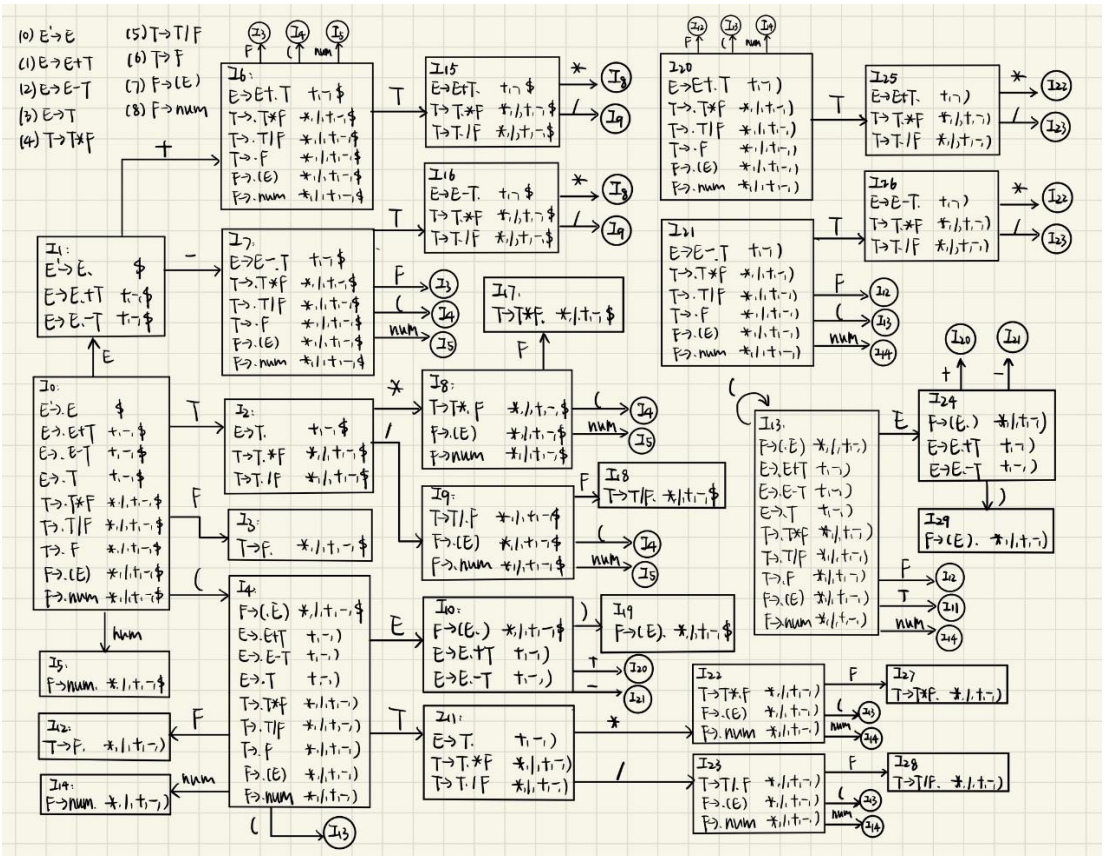
grammar:存储要识别的文法，记录其 first 集与 follow 集

LR1:计算该文法的 LR（1）项目集规范族，构造具有活前缀的 dfa,计算该文法的分析表并且实现 4.3 算，分析输入的字符串

main:输入字符串并调用 lr1 识别

3 活前缀 dfa

活前缀 dfa 图如下展示，**共有 30 个状态。**



4 实现说明

4.1 grammar 类实现

```
class grammar
{
public:
    std::string S;//开始符号
    std::vector<std::string> terminal; //终结符
    std::vector<std::string> non_ter;  //非终结符
    std::vector<std::vector<std::vector<std::string>>> all_gra; //所有文法关系

    int len_non; //非终结符的长度
    int len_ter; //终结符长度
    std::vector<std::vector<std::string>> first; //first集
    std::vector<std::vector<std::string>> follow; //follow集

public:
    grammar();
    void read_grammer();
    void test_print_grammar(); //测试打印此时文法
    void get_first(); //得到first集
```

```
    void print_first(); //打印first集
    void get_follow(); //得到follow集
    void print_follow(); //打印follow集
};
```

因为在本次实验中，grammar 类的实现不是重点，因此在 grammar 中，我们直接赋予我们需要的内容。

赋予题目要求的文法：

```
grammar::grammar() {
    S = "E";
    terminal = { "+", "-", "*", "/", "(", ")", "num" };
    non_ter = { "E", "T", "F" };
    all_gra.resize(non_ter.size());
    all_gra[0].resize(3);
    all_gra[0][0] = { "E","+","T" };
    all_gra[0][1] = { "E","-","T" };
    all_gra[0][2] = { "T" };
    all_gra[1].resize(3);
    all_gra[1][0] = { "T","*","F" };
    all_gra[1][1] = { "T","/","F" };
    all_gra[1][2] = { "F" };
    all_gra[2].resize(2);
    all_gra[2][0] = { "(","E",")" };
    all_gra[2][1] = { "num" };
    len_non = non_ter.size();
    len_ter = terminal.size();
    first.resize(len_non);
    follow.resize(len_non);
}
```

直接赋予求出的 first 集与 follow 集。

```
void grammar::get_first() {
    first[0] = { "(","num" };
    first[1] = { "(","num" };
    first[2] = { "(","num" };
}
```

```
//计算follow集
void grammar::get_follow() {
    follow[0] = { "+","-",")","$" };
    follow[1] = { "*","/","+","-",")","$" };
    follow[2] = { "*","/","+","-",")","$" };
}
```

4.2 LR1 类的实现

本次实验，主要任务就在 LR1 类的实现，LR1 类的实现较为复杂，设计如下：

```
class LR1
{
    struct Item
    {
        int lefti, rightj;  //标记文法的左边与右边
        int dotPos;         //记录点的位置
        std::vector<std::string> tail;   //记录小尾巴
    };

    struct State
    {
        int serial;
        std::vector<Item> items;           //记录关系式
        std::unordered_map<std::string, int> trans;  //转移关系
    };

    std::vector<State> states;    //记录所有项目集规范族
    std::vector<std::vector<std::string>>
reduces{{"1","2","3"},{"4","5","6"},{"7","8"},{"0"}}; //记录待约项的编号
    std::vector<std::vector<std::string>> Action;  //Action分析表
    std::vector<std::vector<std::string>> Goto;    //Goto分析表
    grammar gra;
    int count = 0;

    std::vector<int> s_states; //状态栈
    std::vector<std::string> s_symbols; //字符栈
    std::vector<std::string> parse_s; //分析字符串

public:
    auto my_find(std::vector<Item>::iterator beg, std::vector<Item>::iterator end,
int pos); //找到是否已存在item
    int is_ter(std::string a); //判断终结符
    int is_unter(std::string a); //判断非终结符
    void extension_grammar(); //扩展文法
    void init_i0();  //初始化第一个状态
    void get_items(std::vector<Item>& items, Item item); //得到每个状态集的项目集
    void get_closure(std::vector<std::string>& sstring, std::vector<std::string>&
first); //求first闭包
    void find_after(std::vector<Item>& items,std::vector<std::string>& after); //找
到需要转移的情况
    int is_same_items(std::vector<Item>& items, std::vector<State>& state);  //看此
状态在state中是否包含
    void get_states(); //计算状态
    void push_dot(std::vector<Item> old_item, std::vector<Item>& new_item,
```

```
std::string a); //求解移动点的状态
    void get_table();//构造分析表
    void test_print_items(std::vector<Item>& items);
    void test_print_states(std::vector<State>& states);
    void test_print_table();

    void parse_string(std::string s); //预处理字符串
    void parse(std::string w); //分析字符串
    void print_process(int ip); //打印过程

};
```

首先我们设计了 Item 结构，用来记录每一个项目，包括对应产生式、点的位置、向前搜索符集合。

```
    struct Item
    {
        int lefti, rightj;  //标记文法的左边与右边
        int dotPos;         //记录点的位置
        std::vector<std::string> tail;   //记录小尾巴
    };
```

接着我们设计了 State 的结构，记录了状态的序号，此状态的项目集，以及这个项目的转移关系。

```
    struct State
    {
        int serial; //序号
        std::vector<Item> items;           //记录关系式
        std::unordered_map<std::string, int> trans;  //转移关系
    };
```

第一步：求拓广文法

首先，我们需要拓广文法，在此时我们需要将文法的开始符号更新为“E’ ”，并且加入“E’ ->E”的产生式。根据需要此时我们需要更新终结符的集合，加入“$”。

```
void LR1::extension_grammar() {
    gra.S = "E'";
    int len = gra.all_gra.size();
    gra.all_gra.resize(len + 1);
    gra.all_gra[len].resize(1);
    gra.all_gra[len][0] = { "E"};
    gra.non_ter.push_back("E'");
    gra.terminal.push_back("$");
}
```

第二步：计算一个状态之中完整的项目

书上给出了计算一个文法的闭包的算法，如下展示：

算法 4.7 closure(*I*)的构造过程。

输入：项目集合 *I*。

输出：集合 *J*＝closure(*I*)。

方法：

```
J=I;
do {
    J_new=J;
    for(J_new 中的每一个项目[A→α·Bβ,a] 和文法 G 的每个产生式 B→η)
        for (FIRST(βa)中的每一个终结符号 b)
            if([B→·η,b]∉J)  把[B→·η,b]加入 J;
} while (J_new!=J);
```

在我们的实现中，get_items 对应于求闭包的运算。

在该类中通过函数 void get_items(std::vector<Item>& items, Item item)，调用时需要传入目前判断的项目和所有项目集。若判断下一个字符是终结符则直接返回。若是非终结符分为两种情况，判断的字符为产生式的最后一个，则直接继承其向前搜索符；若不是最后一个，则需要计算之后字符的 first 集，调用 void get_closure(std::vector<std::string>& sstring, std::vector<std::string>& first)函数，之后将 first 集加入其中，若 first 集为空，则需要加入生成式的小尾巴。

get_closure(std::vector<std::string>& sstring, std::vector<std::string>& first)函数的实现如下：

```
void LR1::get_closure(std::vector<std::string>& sstring , std::vector<std::string>& first) {
    int flag = 0;
    for (int i = 0; i < sstring.size(); i++) {
        if (is_ter(sstring[i]) != -1) { //如果是终结符
            first.push_back(sstring[i]);
            return;
        }
        else {   //如果是非终结符
            auto it=find(gra.non_ter.begin(), gra.non_ter.end(), sstring[i]) ;
            int pos = distance(gra.terminal.begin(), it);
            flag = 0;  //标记是否有空
            for (int j = 0; j < gra.first[pos].size(); j++) {
                if (gra.first[pos][i] == "epsilon") {
                    flag = 1;
                }
                first.push_back(gra.first[pos][j]);
            }
            if (flag == 0) {
                return;
            }
        }
    }
    if (flag == 1) {
        first.push_back("epsilon");
    }
}
```

那么实现了这个之后，我们就可以得到 I0 的状态，先给 item 一个初始文法“E’ ->E”，在这个基础上我们可以得到所有状态，并且可以得到此时的状态。

I0 的状态实现如下：

```
void LR1::init_i0() {
    gra.get_first();
    gra.get_follow();
    extension_grammar();
    Item item;
    item.lefti= gra.all_gra.size()-1;
    item.rightj = 0;
    item.dotPos = 0;
    item.tail.push_back( "$");
    std::vector<Item> items;
    items.push_back(item);
    for (int i = 0; i < items.size(); i++) {
        get_items(items, items[i]);
    }
    //test_print_items(items);
    State state0;
    state0.items = items;
    state0.serial = count;
    states.push_back(state0);
}
```

第三步：求出所有状态

书上给出了求取 LR（1）项目集规范族的算法，如下展示：

算法 4.8 构造文法 G 的 LR(1)项目集规范族。

输入：文法 G。

输出：G 的 LR(1)项目集规范族。

方法：

构造文法 G 的拓广文法 G'；

```
C={closure({[S'→ · S,$]})};
do
    for(C 中的每一个项目集 I 和每一个文法符号 X)
        if((go(I,X)≠Φ)并且(go(I,X)∉C)) 把 go(I,X)加入 C 中;
while (有新项目集加入 C 中);
```

这里 closure({[$S'\rightarrow \cdot S, \$$]})是活前缀 ε 的有效项目集。

此时我们已经求出了初始状态，那么我们接下来的任务便很清晰了。我们需要在求出所有状态的时候，还需要记录各个状态的转换关系。首先，我们通过 void find_after(std::vector<Item>& items,std::vector<std::string>& after) 函数求出此时需要转换的字符。

```
//查找点之后的字符
void LR1::find_after(std::vector<Item>& items, std::vector<std::string>& after) {
    for (int i = 0; i < items.size(); i++) {
        if (items[i].dotPos < gra.all_gra[items[i].lefti][items[i].rightj].size()) {
            std::string temp = gra.all_gra[items[i].lefti][items[i].rightj][items[i].dotPos];
            if (find(after.begin(), after.end(), temp) == after.end()) {
                after.push_back(temp);
            }
        }
    }
}
```

然后通过调用 void push_dot(std::vector<Item> old_item, std::vector<Item>&

new_item, std::string a)函数来向后移动点的位置。该函数实现如下：

```
//求解移动点的状态
void LR1:: push_dot(std::vector<Item> old_item, std::vector<Item>& new_item, std::string a) {
    for (int i = 0; i < old_item.size(); i++) {
        if (old_item[i].dotPos < gra.all_gra[old_item[i].lefti][old_item[i].rightj].size()) {
            if (gra.all_gra[old_item[i].lefti][old_item[i].rightj][old_item[i].dotPos] == a) {
                Item item;
                item.lefti = old_item[i].lefti;
                item.rightj = old_item[i].rightj;
                item.tail = old_item[i].tail;
                item.dotPos = old_item[i].dotPos + 1;
                new_item.push_back(item);
            }
        }
    }
}
```

接着求取移动后点的所有项目，判断此时的项目有没有重复，若有重复则转移函数记录为该状态的序号；反之，则新建状态添加。总的 get_states()函数如下：

```
void LR1::get_states() {
    init_i0();
    for (int i = 0; i < states.size(); i++) {
        std::vector<std::string> after;
        find_after(states[i].items, after);   //找到之后可以转换的字符
        for (int j = 0; j < after.size(); j++) { //遍历转换字符
            std::vector<Item> new_items;
            push_dot(states[i].items, new_items, after[j]); //移动点形成新的items
            for (int k = 0; k < new_items.size(); k++) { //生成总的新的items
                get_items(new_items, new_items[k]);
            }
            int flag = is_same_items(new_items, states);
            if (flag == -1) {   //如果此时没有相同的
                count++;
                State astate;
                astate.items=new_items;
                astate.serial = count;
                states.push_back(astate);
                states[i].trans[after[j]] = count;
            }
            else {
                states[i].trans[after[j]] = flag;
            }
        }
        std::cout << "I" << i << "\n";
        test_print_items(states[i].items);
    }
}
```

通过以上的操作，我们可以成功得到所有状态了，详细状态结果已经存在 states.txt 文件中。因为共有 30 个状态，因此不在此展示。

第四步：求分析表

分析表的求取在书上也给出了算法，如下：

算法 4.9 构造文法 G 的 LR(1)分析表。

输入：文法 G。

输出：文法 G 的 LR(1)分析表(包括 action 表和 goto 表两部分)。

方法：

(1) 构造文法 G 的拓广文法 G'；

(2) 构造 G'的 LR(1)项目集规范族 $C=\{I_0, I_1, \cdots, I_n\}$。

(3) 对于状态 i(代表项目集 I_i)，分析动作如下：

- 若$[A\to\alpha\cdot a\beta, b]\in I_i$，且 $go(I_i, a)=I_j$，则置 action$[i,a]=Sj$。
- 若$[A\to\alpha\cdot, a]\in I_i$，且 $A\neq S'$，则置 action$[i,a]=Rj$，这里 j 是产生式 $A\to\alpha$ 的编号。
- 若$[S'\to S\cdot, \$]\in I_i$，则置 action$[i, \$]=$ACC。

(4) 若对非终结符号 A，有 $go(I_i, A)=I_j$，则置 goto$[i,A]=j$。

(5) 分析表中凡是不能用规则(3)和规则(4)填入信息的空白表项，均置上出错标志 error。

(6) 分析程序的初态是包括$[S'\to\cdot S, \$]$的有效项目集所对应的状态。

在上面的步骤中，我们已经求出了所有的状态，并且也已经知道他们的转移关系，那么我们接下来只要根据转移关系求取状态分析表集合。状态分析表分为 Action 表和 Goto 表，用 std::vector<std::vector<std::string>>的格式进行存储。横坐标为其状态序号，纵坐标为其在终结符和非终结符中的位置。 需要注意的是，我们需要在其向前搜索符集中进行规约操作。具体实现如下：

```
//构造分析表
void LR1::get_table() {
    Action.resize(states.size());
    Goto.resize(states.size());
    for (int i = 0; i < states.size(); i++) {
        Action[i].resize(gra.terminal.size());
        Goto[i].resize(gra.non_ter.size()-1);
        for (int j = 0; j < states[i].items.size(); j++) { //查找规约项
            if (states[i].items[j].dotPos == gra.all_gra[states[i].items[j].lefti][states[i].items[j].rightj].size()) {
                if (states[i].items[j].lefti == 3 && states[i].items[j].rightj == 0) { //如果此时是扩展项目
                    Action[i][gra.terminal.size() - 1] = "ACC";
                }
                else {
                    for (int k = 0; k < states[i].items[j].tail.size(); k++) {  //有小尾巴在的地方规约
                        auto it = find(gra.terminal.begin(), gra.terminal.end(), states[i].items[j].tail[k]);
                        if (it == gra.terminal.end()) {
                            std::cout << "error!";
                        }
                        else {
                            int pos = distance(gra.terminal.begin(), it);
                            Action[i][pos] = "R"+(reduces[states[i].items[j].lefti][states[i].items[j].rightj]);

                        }
                    }
                }
            }
        }
```

```
        for (auto j = states[i].trans.begin(); j != states[i].trans.end(); j++) {
            std::string temp;
            temp = (*j).first;
            auto it = find(gra.terminal.begin(), gra.terminal.end(), temp);
            if (it != gra.terminal.end()) { //如果转移是终结符
                int pos = distance(gra.terminal.begin(), it);
                Action[i][pos] = "S" + std::to_string((*j).second);
            }
            else { // 如果转移是非终结符
                it = find(gra.non_ter.begin(), gra.non_ter.end(), temp);
                int pos = distance(gra.non_ter.begin(), it);
                Goto[i][pos] = std::to_string((*j).second);
            }
        }
    }
    test_print_table();
}
```

最后我们可以得到的分析表结果为：

	Action								Goto	
	+	-	*	/	(	)	num	$	E	T	F
0					S4		S5		1	2	3
1	S6	S7						ACC			
2	R3	R3	S8	S9				R3			
3	R6	R6	R6	R6				R6			
4					S13		S14		10	11	12
5	R8	R8	R8	R8				R8			
6					S4		S5			15	3
7					S4		S5			16	3
8					S4		S5				17
9					S4		S5				18
10	S20	S21				S19					
11	R3	R3	S22	S23		R3					
12	R6	R6	R6	R6		R6					
13					S13		S14		24	11	12
14	R8	R8	R8	R8		R8					
15	R1	R1	S8	S9				R1			
16	R2	R2	S8	S9				R2			
17	R4	R4	R4	R4				R4			
18	R5	R5	R5	R5				R5			
19	R7	R7	R7	R7				R7			
20					S13		S14			25	12
21					S13		S14			26	12
22					S13		S14				27
23					S13		S14				28
24	S20	S21				S29					
25	R1	R1	S22	S23		R1					
26	R2	R2	S22	S23		R2					
27	R4	R4	R4	R4		R4					
28	R5	R5	R5	R5		R5					
29	R7	R7	R7	R7		R7					

第五步：分析字符串

LR 分析程序的设计思路如下：

算法 4.3 LR 分析程序。

输入：文法 G 的一张分析表和一个输入符号串 ω。

输出：若 $\omega \in L(G)$，得到 ω 的自底向上的分析，否则报错。

方法：

首先初始化，将初始状态 S_0 入栈，将 $\omega\$$ 存入输入缓冲区中；并置 ip 指向 $\omega\$$ 的第一个符号

```
do {
    令 S 是栈顶状态，a 是 ip 所指向的符号；
    if(action[S,a]=shift S′) {
        把 a 和 S′分别压入符号栈和状态栈的栈顶；
        推进 ip,使它指向下一个输入符号；
    };
    else if (action[S,a]=reduce by A→β) {
            从栈顶弹出|β|个符号；
            令 S′是当前栈顶状态，把 A 和 goto[S′,A]分别压入符号栈和状态栈的栈顶；
            输出产生式 A→β;
        };
        else if (action[S,a]=accept) return;
              else error();
} while(1);
```

同理，我们还是需要先预处理字符串，因为在数字类型中有整数，小数，科学计数等形式，我们都要将其识别为 num，因此我们利用词法分析时的 dfa 转换图。同时为了方便标识这是 num 类型，我们在记录其值的时候在前面加上字符‘0’标识这是 num 类型，这样就不用构造新的变量了。具体实现在 parse_string(std::string s)函数中，这部分内容与之前

类似，因此代码就不在此贴出，具体内容见附件中的代码。

预处理之后，我们继续进行字符串分析，在分析的过程中，我们还需要将此时状态栈和符号栈都记录下来，并且还需要打印此时的动作。

```
void LR1::parse(std::string w) {
    parse_string(w + "$");

    s_states.push_back(0);
    s_symbols.push_back("~");

    int s = 0;
    int ip = 0;
    std::string a;
    std::string action_str;
    do {
        print_process(ip);
        action_str = "";
        s = s_states[s_states.size()-1];
        a = parse_s[ip];
        if (a[0] == '0') {  //如果此时是num
            a = "num";
        }
        int pos = is_ter(a);
        if (pos == -1) {
            std::cout << "error!";
        }
        else {
            std::cout << std::setw(20);
            std::string temp = Action[s][pos];
            if (temp == "ACC") {
                action_str = "ACCEPT\n";
                std::cout << action_str;
                return;
            }
```

```
            else if (temp[0] == 'S') { //此时需要shift
                int t_state = 0;
                for (int i = 1; i < temp.size(); i++) {
                    t_state = t_state * 10 + temp[i] - '0';
                }
                s_states.push_back(t_state);
                s_symbols.push_back(a);
                ip++;
                std::cout << "Shift " << t_state << "\n";
            }
            else if (temp[0] == 'R') { //此时需要规约
                int t_re = 0;
                for (int i = 1; i < temp.size(); i++) {
                    t_re = t_re * 10 + temp[i] - '0';
                }
                int pi = (t_re - 1) / 3;
                int pj = (t_re - 1) % 3;
                std::cout << "Reduce by " << gra.non_ter[pi] << "->";
                for (int i = 0; i < gra.all_gra[pi][pj].size(); i++) {
                    std::cout << gra.all_gra[pi][pj][i];
                    s_states.pop_back();
                    s_symbols.pop_back();
                }
                std::cout << "\n";
                int pos1 = s_states[s_states.size() - 1];
                std::string tempc = gra.non_ter[pi];
                s_symbols.push_back(tempc);
                int pos2 = is_unter(tempc);
```

```
                int pos2 = is_unter(tempc);
                if (pos2 == -1) {
                    std::cout << "Error!";
                }
                else {
                    int t_state = 0;
                    for (int j = 0; j < Goto[pos1][pos2].size(); j++) {
                        t_state = t_state * 10 + Goto[pos1][pos2][j] - '0';
                    }
                    s_states.push_back(t_state);
                }

            }
            else {
                std::cout << "Error!\n";
                return;
            }
        }
    } while (1);
```

5 测试样例

5.1 测试样例 1

输入：2+3

输出：

```
请输入要规约的字符串
2+3
 0
 -          2+3$               Shift 5
 0  5
 - num          +3$           Reduce by F->num
 0  3
 -  F           +3$           Reduce by T->F
 0  2
 -  T           +3$           Reduce by E->T
 0  1
 -  E           +3$               Shift 6
 0  1  6
 -  E  +           3$              Shift 5
 0  1  6  5
 -  E  + num           $          Reduce by F->num
 0  1  6  3
 -  E  +  F           $          Reduce by T->F
 0  1  6 15
 -  E  +  T           $          Reduce by E->E+T
 0  1
 -  E           $              ACCEPT
```

5.2 测试样例 2

输入：5+2*(3/4)

输出：

```
请输入要规约的字符串
5+2*(3/4)
 0
 -         5+2*(3/4)$              Shift 5
 0  5
 - num          +2*(3/4)$         Reduce by F->num
 0  3
 -  F           +2*(3/4)$         Reduce by T->F
 0  2
 -  T           +2*(3/4)$         Reduce by E->T
 0  1
 -  E           +2*(3/4)$             Shift 6
 0  1  6
 -  E  +          2*(3/4)$            Shift 5
 0  1  6  5
 -  E  + num          *(3/4)$         Reduce by F->num
 0  1  6  3
 -  E  +  F           *(3/4)$         Reduce by T->F
 0  1  6 15
 -  E  +  T           *(3/4)$             Shift 8
 0  1  6 15  8
 -  E  +  T  *           (3/4)$            Shift 4
 0  1  6 15  8  4
 -  E  +  T  *  (           3/4)$           Shift 14
 0  1  6 15  8  4 14
 -  E  +  T  *  ( num          /4)$         Reduce by F->num
 0  1  6 15  8  4 12
 -  E  +  T  *  (  F           /4)$         Reduce by T->F
 0  1  6 15  8  4 11
 -  E  +  T  *  (  T           /4)$             Shift 23
 0  1  6 15  8  4 11 23
 -  E  +  T  *  (  T  /           4)$            Shift 14
 0  1  6 15  8  4 11 23 14
 -  E  +  T  *  (  T  / num         )$          Reduce by F->num
 0  1  6 15  8  4 11 23 28
 -  E  +  T  *  (  T  /  F          )$          Reduce by T->T/F
```

```
0  1  6 15  8  4 11
-  E  +  T  *  (  T          /4)$              Shift 23
0  1  6 15  8  4 11 23
-  E  +  T  *  (  T  /          4)$               Shift 14
0  1  6 15  8  4 11 23 14
-  E  +  T  *  (  T  / num         )$          Reduce by F->num
0  1  6 15  8  4 11 23 28
-  E  +  T  *  (  T  /  F         )$          Reduce by T->T/F
0  1  6 15  8  4 11
-  E  +  T  *  (  T          )$          Reduce by E->T
0  1  6 15  8  4 10
-  E  +  T  *  (  E          )$              Shift 19
0  1  6 15  8  4 10 19
-  E  +  T  *  (  E  )          $          Reduce by F->(E)
0  1  6 15  8 17
-  E  +  T  *  F          $          Reduce by T->T*F
0  1  6 15
-  E  +  T          $          Reduce by E->E+T
0  1
-  E          $             ACCEPT
```

5.3 测试样例 3

输入：5.7+14e8

输出：

```
请输入要规约的字符串
5.7+14e8
 0
 -        5.7+14e8$              Shift 5
 0  5
 - num          +14e8$          Reduce by F->num
 0  3
 -  F          +14e8$          Reduce by T->F
 0  2
 -  T          +14e8$          Reduce by E->T
 0  1
 -  E          +14e8$              Shift 6
 0  1  6
 -  E  +       14e8$              Shift 5
 0  1  6  5
 -  E  + num          $          Reduce by F->num
 0  1  6  3
 -  E  +  F          $          Reduce by T->F
 0  1  6 15
 -  E  +  T          $          Reduce by E->E+T
 0  1
 -  E          $             ACCEPT
```

5.4 测试样例 4

输入：7+1.3e2*(5.4/10)-7E30

输出：

```
7+1.3e2*(5.4/10)-7E30
0
-          7+1.3e2*(5.4/10)-7E30$              Shift 5
0  5
- num          +1.3e2*(5.4/10)-7E30$          Reduce by F->num
0  3
-  F          +1.3e2*(5.4/10)-7E30$          Reduce by T->F
0  2
-  T          +1.3e2*(5.4/10)-7E30$          Reduce by E->T
0  1
-  E          +1.3e2*(5.4/10)-7E30$              Shift 6
0  1  6
-  E  +      1.3e2*(5.4/10)-7E30$              Shift 5
0  1  6  5
-  E  + num          *(5.4/10)-7E30$          Reduce by F->num
0  1  6  3
-  E  +  F          *(5.4/10)-7E30$          Reduce by T->F
0  1  6 15
-  E  +  T          *(5.4/10)-7E30$              Shift 8
0  1  6 15  8
-  E  +  T  *          (5.4/10)-7E30$              Shift 4
0  1  6 15  8  4
-  E  +  T  *  (        5.4/10)-7E30$              Shift 14
0  1  6 15  8  4 14
-  E  +  T  *  ( num          /10)-7E30$          Reduce by F->num
0  1  6 15  8  4 12
-  E  +  T  *  (  F          /10)-7E30$          Reduce by T->F
0  1  6 15  8  4 11
-  E  +  T  *  (  T          /10)-7E30$              Shift 23
0  1  6 15  8  4 11 23
-  E  +  T  *  (  T  /         10)-7E30$              Shift 14
0  1  6 15  8  4 11 23 14
-  E  +  T  *  (  T  / num          )-7E30$          Reduce by F->num
```

```
0  1  6 15  8  4 11 23 28
-  E  +  T  *  (  T  /  F          )-7E30$          Reduce by T->T/F
0  1  6 15  8  4 11
-  E  +  T  *  (  T          )-7E30$          Reduce by E->T
0  1  6 15  8  4 10
-  E  +  T  *  (  E          )-7E30$              Shift 19
0  1  6 15  8  4 10 19
-  E  +  T  *  (  E  )          -7E30$          Reduce by F->(E)
0  1  6 15  8 17
-  E  +  T  *  F          -7E30$          Reduce by T->T*F
0  1  6 15
-  E  +  T          -7E30$          Reduce by E->E+T
0  1
-  E          -7E30$              Shift 7
0  1  7
-  E  -       7E30$              Shift 5
0  1  7  5
-  E  - num          $          Reduce by F->num
0  1  7  3
-  E  -  F          $          Reduce by T->F
0  1  7 16
-  E  -  T          $          Reduce by E->E-T
0  1
-  E          $             ACCEPT
```

5.5 测试样例 5（错误情况）

输入：7.8+3((2-8)

输出：

```
请输入要规约的字符串
7.8+3((2-8)
 0
 -       7.8+3((2-8)$              Shift 5
 0  5
 - num         +3((2-8)$          Reduce by F->num
 0  3
 -  F         +3((2-8)$          Reduce by T->F
 0  2
 -  T         +3((2-8)$          Reduce by E->T
 0  1
 -  E         +3((2-8)$              Shift 6
 0  1  6
 -  E  +         3((2-8)$             Shift 5
 0  1  6  5
 -  E  + num          ((2-8)$            Error!
```

6 实验总结

本次实验，题目中其实只硬性要求了需要编程写分析程序，但是在实验过程中，同时完成了活前缀 dfa 的编写和 LR(1)分析表的编写，加大了实验的难度，但同时对自己也是一种挑战。并且选择的 LR 文法是 LR(1)分析文法，因此还需要考虑向前搜索符。

在做实验的时候，一步一步的完成实验过程，在设计没有明确之时，先把每一步完成，在这一次实验中，先完成了一个状态的所有项目的生成，再考虑怎么求该状态的转移，接着考虑怎么求转移后的状态，再不断去深化细节，比如怎么判断两个状态集是否相同等，然后一步步的完成该实验。最后将所有状态正确求出，并且记录了状态转移的情况。这样在一步步写分析表和分析程序的时候就变得更加容易一些。分析程序同样需要用到识别 num 的 dfa 过程，其实这是我们一开始写的词法分析。并且在画整个 dfa 活前缀图的时候为了画的好看一点也画了好久。

总的来说，本次实验还是非常有收获的，与之前写的 LL（1）文法进行了区别。就比如一开始写的时候没有注意还进行了消除左递归的操作，在写完之后对这个地方印象就更加深刻了。并且对每一个过程和步骤有了更深的了解。是一次十分有收获的实验。