北京邮电大学《编译原理与技术》课程实验报告

<table>
<tr><td>实验名称</td><td colspan="2">语法分析程序的设计与实现</td><td>学院</td><td>计算机</td><td>指导教师</td><td>王雅文</td></tr>
<tr><td>班 级</td><td>班内序号</td><td colspan="2">学 号</td><td>学生姓名</td><td colspan="2">成绩</td></tr>
<tr><td>2020211323</td><td>11</td><td colspan="2">2020211429</td><td>陈宇</td><td colspan="2"></td></tr>
<tr><td></td><td></td><td colspan="2"></td><td></td><td colspan="2"></td></tr>
<tr><td></td><td></td><td colspan="2"></td><td></td><td colspan="2"></td></tr>
<tr><td>实验内容</td><td colspan="6">编写语法分析程序，实现对算数表达式的语法分析。要求所分析算数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</td></tr>
<tr><td>学生实验报告</td><td colspan="6">(详见“实验报告和源程序”册)</td></tr>
<tr><td>课程设计成绩评定</td><td colspan="6">评语:

成绩:

指导教师签名:

年 月 日</td></tr>
</table>

注：评语要体现每个学生的工作情况，可以加页。

语法分析程序的设计分析

一、实验内容以及要求

1.1 实验内容

编写语法分析程序，实现对算数表达式的与语法分析。要求所分析算数表达式由如下文法产生。

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

1.2 实验要求

在对输入的算数表达式进行分析的过程中，依次输出所采用的的产生式。

方法 1：编写递归调用程序实现自顶向下分析。

方法 2：编写 LL(1)语法分析程序，要求如下。

(1) 编写实现算法 4.2，为给定文法自动构造预测分析表。

(2) 编写实现算法 4.1，构造 LL(1)预测分析程序。

方法 3：编写 LR(1)语法分析程序实现自底向上分析，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) 编写实现算法 4.3，构造 LR 分析程序。

方法 4：利用 YACC 自动生成语法分析程序，调用 LEX 自动生成的词法分析程序。

二、递归调用程序设计说明

2.1 设计思路

首先，将给定文法修改为不含左递归的等价文法，为每一个非终结符构造一张对应的状态转移图。

然后，依据状态转移图为每个非终结符构造分析函数，若经过非终结符号标记的边，调用该非终结符的分析函数；若经过终结符标记的边，对比当前输入符号，若相同转移至下一状态，否则沿ε-边转移或调用错误分析程序。

错误分析程序采用不断跳过输入符号直到遇到期望的终结符的方式。

2.2 数据结构

常量

```
//记号类型
enum token
{
```

```
    PLUS,
    MINUS,
    MULTIPLY,
    DIVIDE,
    LEFT_PARENTHESIS,
    RIGHT_PARENTHESIS,
    NUM,
};
```

2.3 函数设计

```
int forward_pointer(ifstream& token_stream, int& char_num)
```
功能：读取一个记号
参数：ifstream& token_stream - 记号输入流
　　int& char_num - 字符数
返回：读取的记号类型

```
int error(ifstream& token_stream, int& char_num, const string&
description, const set<token>& expect_tokens)
```
功能：错误处理，不断跳过输入记号直到遇到其中一个期望的记号
参数：ifstream& token_stream - 记号输入流
　　int& char_num - 字符数
　　const string& description - 错误描述
　　const set<token>& expect_tokens - 期望的记号集
返回：遇到的期望的记号

```
void procE(int& current_token, ifstream& token_stream, int& char_num)
```
功能：非终结符 E 分析函数
参数：int& current_token - 当前输入记号
　　ifstream& token_stream - 记号输入流
　　int& char_num - 字符数

```
void procT(int& current_token, ifstream& token_stream, int& char_num)
```
功能：非终结符 T 分析函数
参数：int& current_token - 当前输入记号
　　ifstream& token_stream - 记号输入流
　　int& char_num - 字符数

```
void procF(int& current_token, ifstream& token_stream, int& char_num)
```
功能：非终结符 F 分析函数
参数：int& current_token - 当前输入记号
　　ifstream& token_stream - 记号输入流
　　int& char_num - 字符数

```
void syntax_analysis(ifstream& token_stream)
```

功能：对输入记号流进行语法分析，依次输出所采用的的生成式
参数：`ifstream& token_stream` - 记号输入流

2.4 详细设计

首先，消除左递归得到如下等价文法：

E->TE’

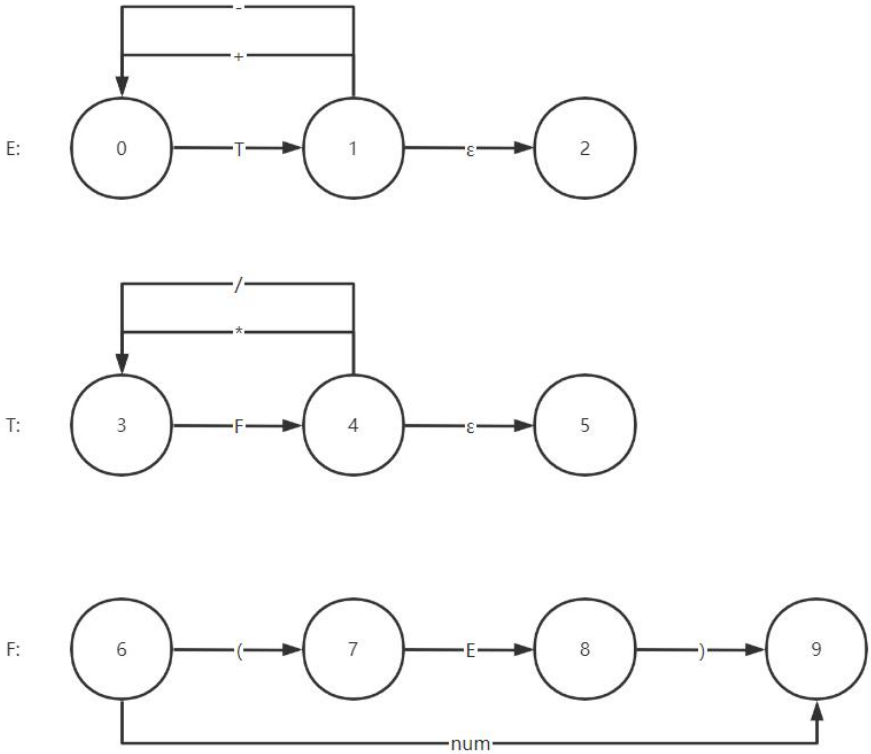
E’->+TE’|-TE’|ε

T->FT’

T’->*FT’|/FT’|ε

F->(E)|num

然后，为每个非终结符构造状态转移图，并简化后得到如下状态转移图：



依据该状态转移图即可构造出每个非终结符的分析函数。若经过非终结符号标记的边，调用该非终结符的分析函数；若经过终结符标记的边，对比当前输入符号，若相同转移至下一状态，否则沿ε-边转移或调用错误分析程序。

由于 E 和 T 在状态 1 和 4 后都具有ε-边，因此最后不需要调用错误分析程序，而 F 状态 6 和 8 都不具有ε-边，因此若输入符号与期望的符号不符，需要调用错误处理程序，不断跳过输入符号直到遇到其中一个期望的终结符。

代码实现请参考源程序。

三、LL(1)语法分析程序设计说明

3.1 设计思路

由于需要依据输入文法自动生成分析表，因此需要设计数据结构存储符号和产生式。为了方便查询，具体数据均采用数组索引代替，同时在需要区别非终结符索引和终结符索引的地方，将非终结符索引改为其相反数以示区别。

在进行语法分析前，依次构造非终结符和产生式右边的 first 集，非终结符的 follow 集，分析表。

构建完分析表后，定义分析栈并将结束符和初始非终结符压入栈中，读入一个输入记号后开始分析。若栈顶为结束符，跳出循环，若输入记号流还没有分析完输出错误信息和剩余记号，否则输出接受输入表达式。若栈顶为终结符，且当期输入符号与栈顶符号相同，弹出栈顶并读取下一个输入记号，否则进行错误处理，直接弹出栈顶。若栈顶为非终结符，查看分析表对应表项，若为产生式，将产生式右边符号串反向依次压入栈顶，否则进行错误处理，依据表项错误原因弹出栈顶或跳过输入记号。

3.2 数据结构

class grammar

存储文法终结符、非终结符、生成式以及该文法的分析表，实现语法分析功能。

3.2.1 结构体

```
typedef struct grammar_production
{
   int left; //non_terminal_symbols 索引
   vector<int> right; //符号索引序列, 负数为非终结符索引
}production;
```

3.2.2 常量成员

```
static const int EMPTY = -1;
static const int SYNCH = -2;
```

3.2.3 非常量成员

```
vector<char> non_terminal_symbols; //0 号位不使用
vector<char> terminal_symbols;     //0 号位不使用
vector<production> productions;
int initial_symbol_index;

//以下数据结构使用数组索引代替具体数据
vector<set<int>> first_set;  //索引为非终结符，内容为终结符集合，0 号位不使用
vector<set<int>> follow_set; //索引为非终结符，内容为终结符集合，0 号位不使用
vector<set<int>> production_first_set;//索引为生成式，内容为终结符集合
```

```
vector<vector<int>> analyse_table; //索引为[非终结符(0 号位不使用)][终结符(0 号位表示 END_SYMBOL)]，内容为生成式
```

3.2.4 成员函数

```
bool add_non_terminal_symbol(const char non_terminal_symbol)
```
功能：加入一个非终结符
参数：const char non_terminal_symbol - 需要加入的非终结符
返回：若该非终结符与某一终结符相同返回 false，否则返回 true

```
bool add_terminal_symbol(const char terminal_symbol)
```
功能：加入一个终结符
参数：const char terminal_symbol - 需要加入的终结符
返回：若该终结符与某一非终结符相同返回 false，否则返回 true

```
bool set_initial_symbol(const char initial_symbol);
```
功能：设置初始非终结符
参数：const char initial_symbol - 给定初始非终结符
返回：若该初始非终结符与不在已加入的非终结符集里返回 false，否则返回 true

```
bool add_production(const char left, const string& right)
```
功能：加入一个文法生成式
参数：const char left - 文法生成式左边非终结符
const string& right - 文法生成式右边符号串
返回：生成式中含有不在已加入非终结符号集和终结符号集的符号返回 false，否则返回 true

```
void construct_first_set()
```
功能：构建非终结符号和生成式右边符号串的 first 集

```
void construct_follow_set()
```
功能：依据非终结符的 first 集构建 follow 集

```
bool construct_analysis_table()
```
功能：依据非终结符的 follow 集和生成式右边符号串的 first 集构建分析表
返回：若存在多重表项，表示该文法非 LL(1)文法，返回 false，否则返回 true

```
int forward_pointer(ifstream& token_stream, int& char_num)
```
功能：读取一个记号
参数：ifstream& token_stream - 记号输入流
int& char_num - 字符数
返回：读取的记号类型

```
void error(ifstream& token_stream, int& current_token, int& char_num, vector<int>& stack, const int& cause)
```

功能：错误处理，依据错误原因选择弹出栈顶或跳过一个输入记号
参数：ifstream& token_stream - 记号输入流
int& current_token - 当前输入记号
int& char_num - 字符数
vector<int>& stack - 分析栈
const int& cause - 错误原因，可能为 EMPTY 或 SYNCH

void syntax_analysis(ifstream& token_stream)
功能：对输入记号流进行语法分析，依次输出所采用的的生成式
参数：ifstream& token_stream - 记号输入流

3.3 详细设计

首先，将文法非终结符和终结符存入数组中，之后所有使用到符号的地方都将使用数组索引代替具体符号，部分需区别非终结符和终结符索引的地方，将非终结符索引取反以示区别。用数组索引代替具体符号方便符号的统一管理，若简单修改程序，具体符号可以设计为任意字符串。

（1）构建非终结符和文法生成式右边符号串的 first 集

具体实现为：对每一个文法生成式，依次分析生成式右边符号串中的符号，若为终结符，将终结符加入 first 集并终止分析；

若为非终结符，将该非终结符的 first 集加入符号串 first 集，若该非终结符 first 集含有ε，继续分析下一个符号，否则停止分析。

若分析完符号串最后一个符号还未停止分析，说明生成式右边可以推导出ε，将ε加入符号串 first 集。最后将生成式右边符号串 first 集加入生成式左边非终结符的 first 集。

重复以上过程直到 first 集不再变化为止。

（2）构建非终结符的 follow 集

为了进一步优化教科书中算法，可以设置一个数据结构 follow_symbol，若在分析过程中 FOLLOW(A)需要加入 FOLLOW(B)，则将 A 加入 follow_symbol[B]。

算法具体实现为：首先将结束符 END_SYMBOL 加入初始非终结符的 follow 集，然后分析所有生成式右边符号串中每一个非终结符 A，生成该非终结符右边符号串的 first 集，将非ε元素加入 A 的 follow 集。

若该非终结符是生成式最后一个符号，或非终结符右边符号串的 first 集含有ε，将生成式左边非终结符 B 的 follow 集加入 A 的 follow 集，并将符 B 加入 A 的 follow_symbol 集。

之后循环遍历每个非终结符的 follow_symbol 集，将非终结符 A 的 follow_symbol 集内的每一个非终结符 B 的 follow 集加入 A 的 follow 集，直到 follow 集不再发生改变。

（3）构造分析表

具体实现为：遍历生成式右边符号串的 first 集，对于每一个 first 集中的元素 a，将该生成式加入到表项 M[A,a]中，其中 A 非生成式左边非终结符。

若 first 集中含有ε，对于 A 的 follow 集中每一个元素 b，将该生成式加入表项 M[A,b]。若存在多重表项说明该文法非 LL(1)文法，构造分析表失败。

构建完分析表后，定义分析栈并将结束符和初始非终结符压入栈中，读入一个输入

记号后开始分析。若栈顶为结束符，跳出循环，若输入记号流还没有分析完输出错误信息和剩余记号，否则输出接受输入表达式。若栈顶为终结符，且当期输入符号与栈顶符号相同，弹出栈顶并读取下一个输入记号，否则进行错误处理，直接弹出栈顶。若栈顶为非终结符，查看分析表对应表项，若为产生式，将产生式右边符号串反向依次压入栈顶，否则进行错误处理，依据表项错误原因弹出栈顶或跳过输入记号。

代码请参考源程序。

四、LR(1)语法分析程序设计说明

4.1 设计思路

首先，构造能够识别文法所有活前缀的 DFA，构造出 LR(1)分析表，并将分析表以二维数组存为常量。

语法分析过程中需要依据当前状态和当前输入记号查询分析表，获取应该进行的动作，若为移进动作，移动至下一个状态并读取下一个输入记号，若为规约动作，依据规约生成式右边符号串的符号数量，将栈中相同数量的状态弹出，并依据生成式左边的非终结符和当前栈顶转态查询分析表，将下一状态压入栈中。若为错误，调用错误处理程序，依据错误原因进行错误处理。若为接受，表示语法分析完成，接受输入算数表达式。

4.2 数据结构

4.2.1 结构体

```
//分析表中分析动作
typedef struct action
{
   char operation;
   int index;
}action;
```

4.2.1 常量

```
//非终结符和终结符
enum symble
{
   PLUS,
   MINUS,
   MULTIPLY,
   DIVIDE,
   LEFT_PARENTHESIS,
   RIGHT_PARENTHESIS,
   NUM,
   END,
   E,
   T,
   F,
```

```
};

//分析动作类型
enum operation
{
    EMPTY,
    SHIFT,
    REDUCE,
    ACCEPT,
    ERROR
};

//错误类型
enum error_type
{
    MISS_OP_OBJECT = 1,
    UNMATCHED_PARENTHESES,
    MISS_OP_SYMBLE,
    MISS_RIGHT_PARENTHESIS,
};

//LR(1)分析表，将 go 表并入 ACTION 表中，动作均为 SHIFT
const action ACTION[30][11] = {
{{ERROR, 1},{ERROR, 1},{ERROR, 1},{ERROR, 1},{SHIFT, 5},{ERROR,
2},{SHIFT, 1},{ERROR, 1},{SHIFT, 3},{SHIFT, 4},{SHIFT, 2}},
...
}

//文法生成式
const vector<string> productions = {
    "",
    "S->E",
    "E->E+T",
    "E->E-T",
    "E->T",
    "T->T*F",
    "T->T/F",
    "T->F",
    "F->(E)",
    "F->num",
};
```

4.3 函数设计

```
int forward_pointer(ifstream& token_stream, int& char_num)
```

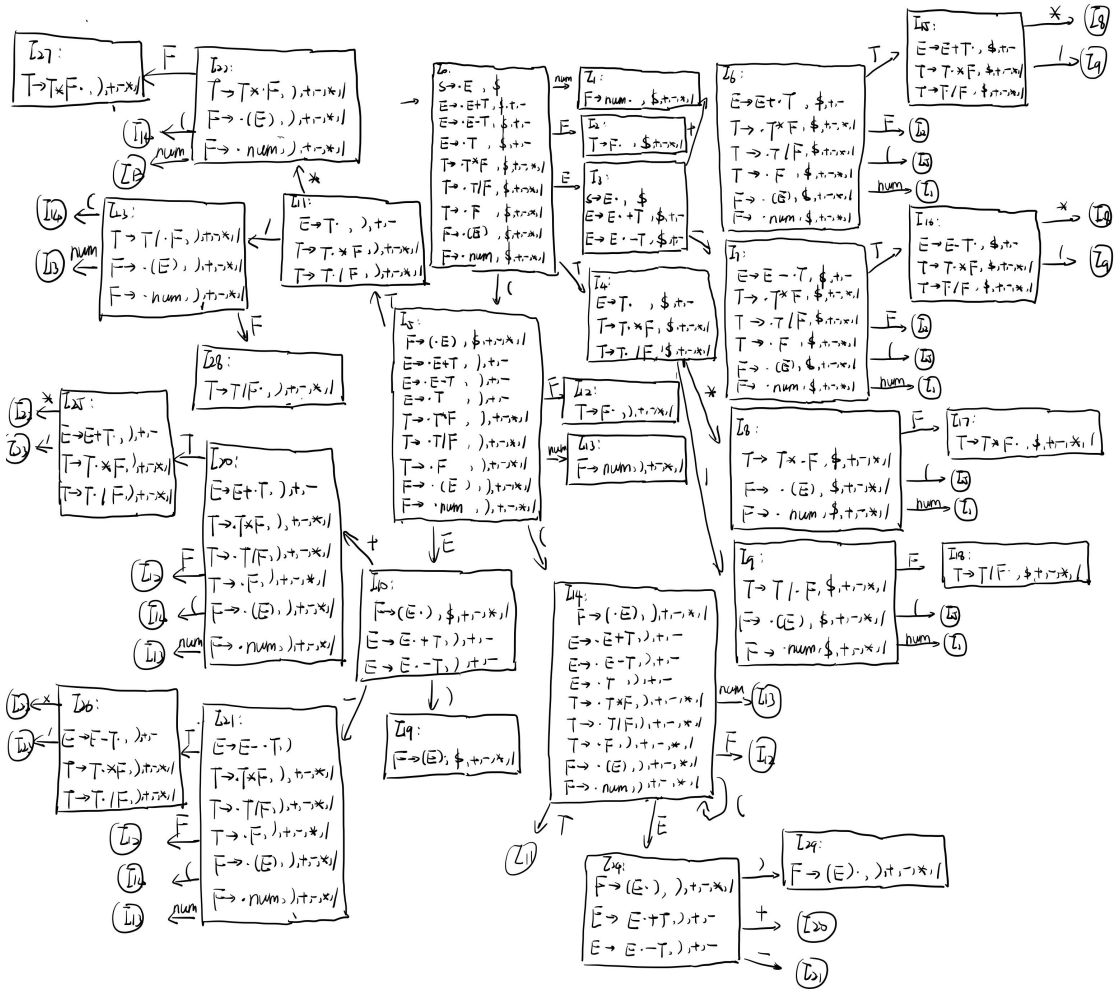
功能：读取一个记号
参数：ifstream& token_stream - 记号输入流
int& char_num - 字符数
返回：读取的记号类型

```
void error(ifstream& token_stream, int& current_token, int& char_num,
vector<int>& stack, const char& error_type)
```
功能：依据错误类型进行错误处理
参数：ifstream& token_stream - 记号输入流
int& current_token - 当前输入记号
int& char_num - 字符数
vector<int>& stack - 状态栈
const char& error_type - 错误类型

```
void syntax_analysis(ifstream& token_stream)
```
功能：对输入记号流进行语法分析，依次输出所采用的的生成式
参数：ifstream& token_stream - 记号输入流

4.4 详细设计

首先，构造能够识别文法所有活前缀的 DFA，DFA 如下：



然后，依据 DFA 构造分析表，分析表如下：

	action								goto		
	+	−	*	/	(	)	num	$	E	T	F
0					S5		S1		3	4	2
1	R9	R9	R9	R9				R9			
2	R7	R7	R7	R7				R7			
3	S6	S7						ACC			
4	R4	R4	S8	S9				R4			
5					S14		S13		10	11	12
6					S5		S1			15	12
7					S5		S1			16	12
8					S5		S1				17
9					S5		S1				18
10	S20	S21				S19					
11	R4	R4	S22	S23		R4					
12	R7	R7	R7	R7		R7					
13	R9	R9	R9	R9		R9					
14					S14		S13		24	11	12
15	R2	R2	S8	S9				R2			
16	R3	R3	S8	S9				R3			
17	R5	R5	R5	R5				R5			
18	R6	R6	R6	R6				R6			
19	R8	R8	R8	R8				R8			
20					S14		S13			25	12
21					S14		S13			26	12
22					S14		S13				27
23					S14		S13				28
24	S20	S21				S29					
25	R2	R2	S22	S23		R2					
26	R3	R3	S22	S23		R3					
27	R5	R5	R5	R5		R5					
28	R6	R6	R6	R6		R6					
29	R8	R8	R8	R8		R8					

依据算数表达式的特性，向表中空缺表项填入错误类型，修改后分析表如下：

	action								goto		
	+	−	*	/	(	)	num	$	E	T	F
0	e1	e1	e1	e1	S5	e2	S1	e1	3	4	2
1	R4	R4	R4	R4	R4	R4	R4	R4			
2	R7	R7	R7	R7	R7	R7	R7	R7			
3	S6	S7			e3	e2	e3	ACC			
4	R4	R4	S8	S9	R4	R4	R4	R4			
5	e1	e1	e1	e1	S14	e2	S13	e1	10	11	12
6	e1	e1	e1	e1	S5	e2	S1	e1		15	12
7	e1	e1	e1	e1	S5	e2	S1	e1		16	12
8	e1	e1	e1	e1	S5	e2	S1	e1			17
9	e1	e1	e1	e1	S5	e2	S1	e1			18
10	S20	S21			e3	S19	e3	e4			
11	R4	R4	S22	S23	R4	R4	R4	R4			
12	R7	R7	R7	R7	R7	R7	R7	R7			
13	R9	R9	R9	R9	R9	R9	R9	R9			
14	e1	e1	e1	e1	S14	e2	S13	e1	24	11	12
15	R2	R2	S8	S9	R2	R2	R2	R2			
16	R3	R3	S8	S9	R3	R3	R3	R3			
17	R5	R5	R5	R5	R5	R5	R5	R5			
18	R6	R6	R6	R6	R6	R6	R6	R6			
19	R8	R8	R8	R8	R8	R8	R8	R8			
20	e1	e1	e1	e1	S14	e2	S13	e1		25	12
21	e1	e1	e1	e1	S14	e2	S13	e1		26	12
22	e1	e1	e1	e1	S14	e2	S13	e1			27
23	e1	e1	e1	e1	S14	e2	S13	e1			28
24	S20	S21			e3	S29	e3	e4			
25	R2	R2	S22	S23	R2	R2	R2	R2			
26	R3	R3	S22	S23	R2	R3	R2	R2			
27	R5	R5	R5	R5	R5	R5	R5	R5			
28	R6	R6	R6	R6	R6	R6	R6	R6			
29	R8	R8	R8	R8	R8	R8	R8	R8			

e1：期待的输入符号为运算对象，即“(”或“num”，但当前输入符号却是运算符号或结束符。给出诊断信息“缺少运算对象”，对于栈顶状态 0，6，7，8，9 将状态 1 压入栈顶，对于栈顶状态 5，14，20，21，22，23 将状态 13 压入栈顶。

e2：期待的输入符号为运算对象，即“(”或“num”，但当前输入符号却是“）”。给出诊断信息“括号不匹配”，跳过当前输入记号，读入下一个输入记号。

e3：期待的输入符号为运算符号或“)”，但当前输入符号却是运算对象，即“(”或“num”。给出诊断信息“缺少运算符号”，对于栈顶状态 3 将状态 6 压入栈顶，对于栈顶状态 10，24 将状态 20 压入栈顶。

e4：期待的输入符号为运算符号或“)”，但当前输入符号为结束符。给出诊断信息“缺少右括号”，对于栈顶状态 10 将状态 19 压入栈顶，对于栈顶状态 24 将状态 29 压入栈顶。

对于状态 1,2,4,11,12,13,15,16,17,18,19,25,26,27,28,29 的空缺表项均填入规约动作。这样，在检查除错误前可以规约出一个或多个非终结符，但错误在移进下一个输入符号前仍然会被捕获。

由于在进入状态 3,10,24 前需要规约一个非终结符 E 出来，而所有含有规约出 E 的规约项目的状态遇到*或/符号时都会进行移进动作而不会进行规约，也就是说规约出 E 后当前输入符号不可能为*或/，那么状态 3,10,24 不可能出现遇到*或/的情况，表项为空也不会影响错误处理。

获得分析表后将表项依次输入 ACTION 二维数组中，其中 go 表并入 ACTION 表，动作均为移进动作。

语法分析时只需要依据当前状态和当前输入记号查询分析表，获取应该进行的动作，若为移进动作，移动至下一个状态并读取下一个输入记号，若为规约动作，依据规约生成式右边符号串的符号数量，将栈中相同数量的状态弹出，并依据生成式左边的非终结符和当前栈顶转态查询分析表，将下一状态压入栈中。若为错误，调用错误处理程序，依据错误原因进行错误处理。若为接受，表示语法分析完成，接受输入算数表达式。

五、YACC 生成语法分析程序

5.1 编写 LEX 源程序

```
%{
    #include"y.tab.h"
%}
digit   [0-9]
%%
{digit}+ {return DIGIT;}
"+"      {return '+';}
"-"      {return '-';}
"*"      {return '*';}
"/"      {return '/';}
"("      {return '(';}
")"      {return ')';}
"\n"     {return '\n';}
%%
```

LEX 生成的词法分析程序将作为语法分析程序的子过程，每调用一次就返回一个记号。DIGIT 定义由 YACC 生成的 y.tab.h 内定义，表示数字。

5.2 编写 YACC 源程序

```
%{
    #include<stdio.h>
    #include<ctype.h>
    #include "lex.yy.c"
%}
%token DIGIT
%%
line:expr'\n'   {printf("ACCEPT\n"); return 0;}
;
expr:expr'+'term    {printf("E->E+T\n");}
    |expr'-'term    {printf("E->E-F\n");}
    |term           {printf("E->T\n");}
    ;
term:term'*'factor  {printf("T->T*F\n");}
    |term'/'factor  {printf("T->T/F\n");}
    |factor         {printf("T->F\n");}
    ;
factor:'('expr')'   {printf("F->(E)\n");}
    |DIGIT          {printf("F->num\n");}
    ;
%%
main()
{
    return yyparse();
}
void yyerror(char* s)
{
    printf("%s\n", s);
}
```

在 YACC 源程序中定义了记号 DIGIT，由 LEX 生成的词法分析程序识别返回。YACC 生成的语法分析程序每规约出一个文法生成式就调用对应的语义动作，输出采用的文法生成式，直到一行末尾结束分析。

六、程序测试

6.1 测试案例

（1）无错误案例

(n+(n-n)*(n+n))/(n-n)

（2）含错误案例

(n+(n-)*(n+n)))/(nn-n

其中 n 表示 num。

6.2 测试结果

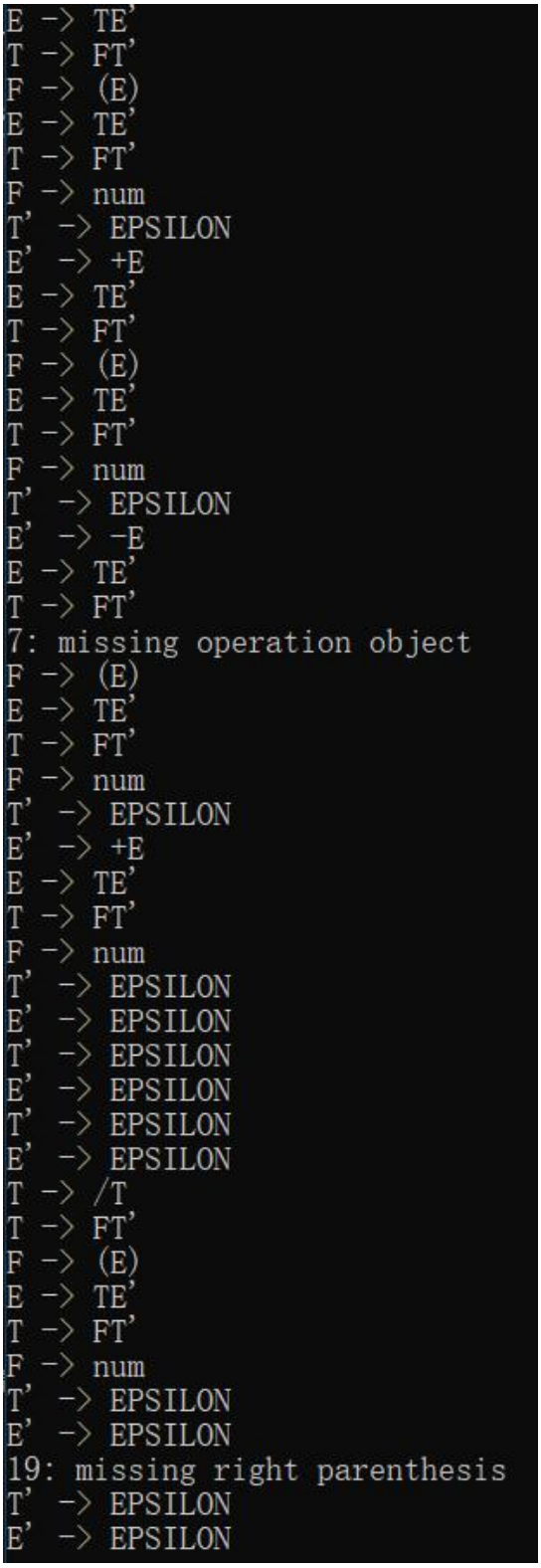
6.2.1 递归调用程序测试结果

将测试用例写入同目录下的 token_stream.txt 文件内，执行语法分析程序。对于无错误案例，程序输出采用的产生式如下：

```
E -> TE'
T -> FT'
F -> (E)
E -> TE'
T -> FT'
F -> num
T' -> EPSILON
E' -> +E
E -> TE'
T -> FT'
F -> (E)
E -> TE'
T -> FT'
F -> num
T' -> EPSILON
E' -> -E
E -> TE'
T -> FT'
F -> num
T' -> EPSILON
E' -> EPSILON
T -> *T
T -> FT'
F -> (E)
E -> TE'
T -> FT'
F -> num
T' -> EPSILON
E' -> +E
E -> TE'
T -> FT'
F -> num
T' -> EPSILON
E' -> EPSILON
T' -> EPSILON
E' -> EPSILON
T -> /T
T -> FT'
F -> (E)
E -> TE'
T -> FT'
F -> num
T' -> EPSILON
E' -> -E
E -> TE'
T -> FT'
F -> num
T' -> EPSILON
E' -> EPSILON
T' -> EPSILON
E' -> EPSILON
```

经检验，程序输出正确。

对于含错误案例，程序输出采用的产生式如下：



在第 7 个符号处，期望的符号为“n”或“(”，但当前输入符号为“)”，错误处理程序跳过两个字符，直到在第 9 符号处发现符号“(”，程序从此处继续分析。

在第 19 个字符处，期望的符号为“)”，但当前输入符号为“n”，错误处理程序跳过三个字符后读取到文件结束符，依旧未找到需要的符号，返回文件结束符，程序以文件结束符作为当前输入符号继续分析。

6.2.2 LL(1)语法分析程序测试结果

首先，将文法修改为无左递归等价文法：

E->TE’

A->+TA|-TA|ε

T>FB

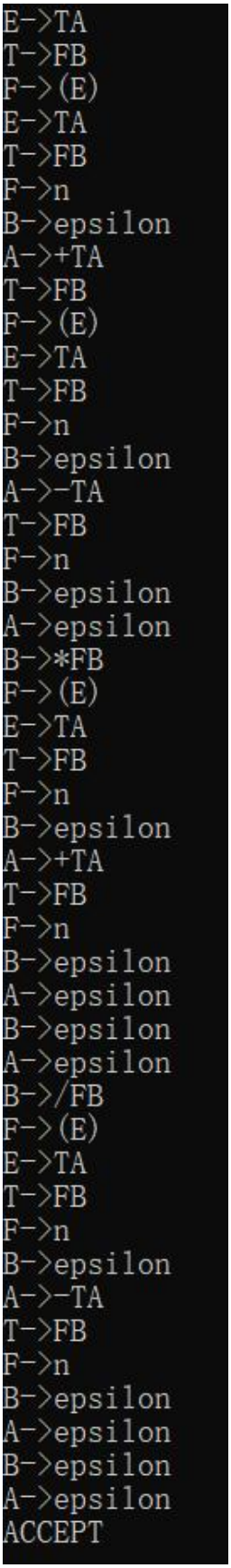
B->*FB|/FB|ε

F->(E)|n

依据提示，输入文法给程序：

```
How many non-terminal symbols are there: 5
non-terminal symbols:
   [1] E
   [2] A
   [3] T
   [4] B
   [5] F
How many terminal symbols are there: 7
terminal symbols:
   [1] +
   [2] -
   [3] *
   [4] /
   [5] (
   [6] )
   [7] n
How many productions are there: 10
Note: the grammar productions should be left-recursion-eliminated
Note: if the right symbol string is epsilon, just input nothing and press the enter key
grammar productions:
   [1]
       left: E
       right: TA
   [2]
       left: A
       right: +TA
   [3]
       left: A
       right: -TA
   [4]
       left: A
       right:
   [5]
       left: T
       right: FB
   [6]
       left: B
       right: *FB
   [7]
       left: B
       right: /FB
   [8]
       left: B
       right:
   [9]
       left: F
       right: (E)
   [10]
       left: F
       right: n
initial non-terminal symbol: E
```

随后，将测试用例写入同目录下的 token_stream.txt 文件内，执行语法分析程序，程序依次输出采用的产生式如下：



由于和递归下降程序均采用自顶向下分析的方式，LL(1)采用的产生式和递归下降采用的产生式相同。

对于含错误案例，程序输出采用的产生式如下：

```
E->TA
T->FB
F->(E)
E->TA
T->FB
F->n
B->epsilon
A->+TA
T->FB
F->(E)
E->TA
T->FB
F->n
B->epsilon
A->-TA
7: pop state 'T'
A->epsilon
B->*FB
F->(E)
E->TA
T->FB
F->n
B->epsilon
A->+TA
T->FB
F->n
B->epsilon
A->epsilon
B->epsilon
A->epsilon
B->epsilon
A->epsilon
The stack is empty , but there are still tokens not analysed
rest tokens: )/(nn-n
```

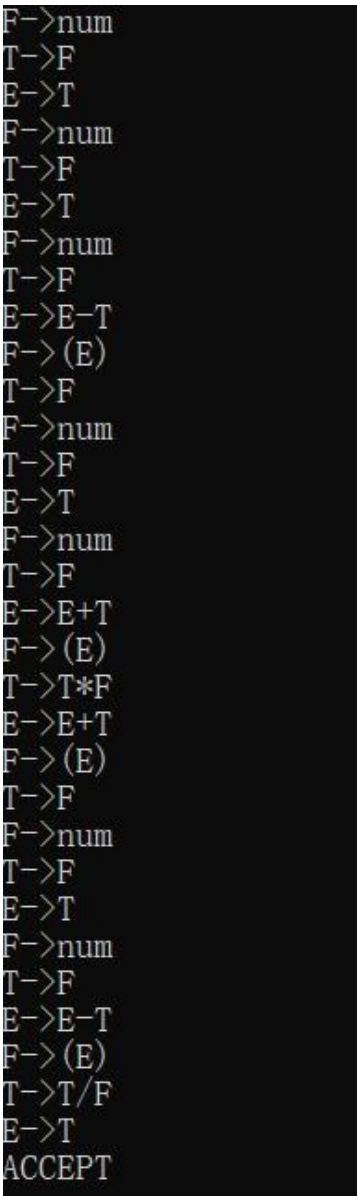
在第 7 个符号处，由于符号“)”属于非终结符 T 的 follow 集，因此 M[T,)]=SYNCH，需要弹出符号 T。

当分析完 14 个符号后，依据分析表，非终结符 A 和 B 遇到“)”会采用ε生成式，弹出栈内 A 和 B 后，最终栈顶只留下终结符，但还有未分析完的字符串“)/nn-n”。

6.2.3 LR(1)语法分析程序测试结果

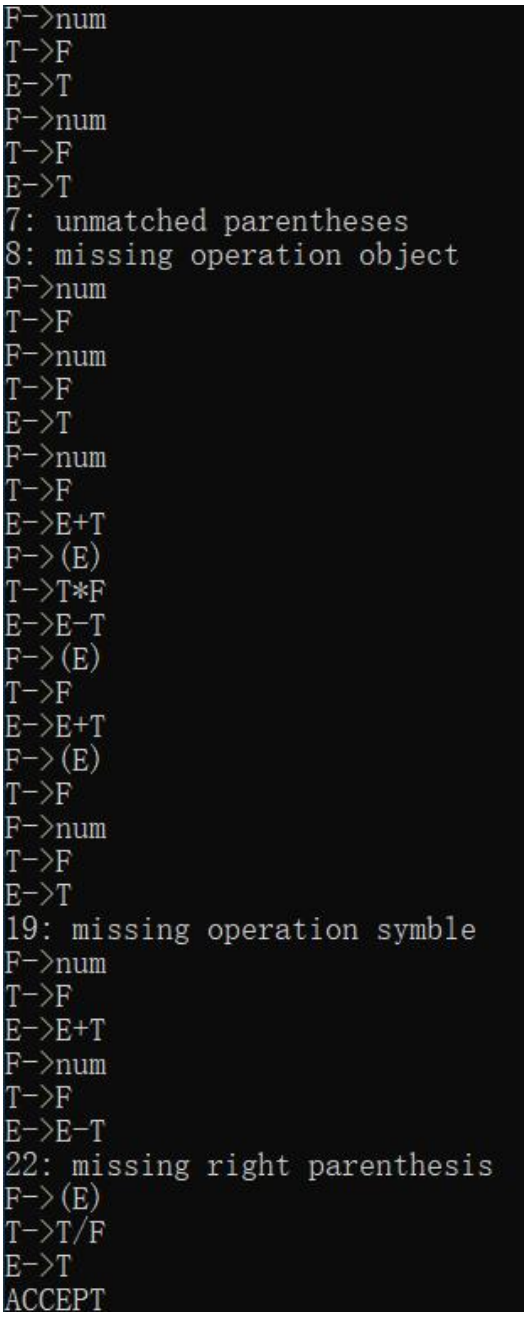
将测试用例写入同目录下的 token_stream.txt 文件内，执行语法分析程序。

对于无错误案例，程序输出采用的产生式如下：



经检验，程序输出正确。

对于含错误案例，程序输出采用的产生式如下：



由于不需要对文法进行去除左递归处理，文法生成式更少，分析树更加简单。同时，LR(1)能够实现更加详细的错误处理。

在第 7 个符号处，查询分析表后得到{ERROR, 2}，表示期待的输入符号为运算对象，即“(”或“num”，但当前输入符号却是“）”。给出诊断信息“括号不匹配”，跳过当前输入记号，读入下一个输入记号。

在第 8 个符号处，由于跳过了“)”，相当于“+”后面跟着的符号变为了“*”，查询分析表后得到{ERROR, 1}，表示期待的输入符号为运算对象，即“(”或“num”，但当前输入符号却是运算符号或结束符。给出诊断信息“缺少运算对象”，对于栈顶状态 0，6，7，8，9 将状态 1 压入栈顶，对于栈顶状态 5，14，20，21，22，23 将状态 13 压入栈顶。

由于之前跳过了一个“)”，第 14 个符号处多出来的“)”，正好匹配上了前面的“(”，程序不会报错。

在第 19 个符号处，查询分析表后得到{ERROR, 3}，表示期待的输入符号为运算符号或“)”，但当前输入符号却是运算对象，即“(”或“num”。给出诊断信息“缺少运算符号”，对于栈顶状态 3 将状态 6 压入栈顶，对于栈顶状态 10，24 将状态 20 压入栈顶。

在第 22 个符号处，当前符号已经读到结束符，查询分析表后得到{ERROR, 4}，表示期待的输入符号为运算符号或“)”，但当前输入符号为结束符。给出诊断信息“缺少右括号”，对于栈顶状态 10 将状态 19 压入栈顶，对于栈顶状态 24 将状态 29 压入栈顶。

6.2.4 YACC 自动生成语法分析程序测试结果

由于 YACC 自动生成的语法分析程序直接调用了 LEX 自动生成的词法分析程序，因此可以直接识别真正的算数表达式而非记号流。

测试用例可以直接使用以下算数表达式：

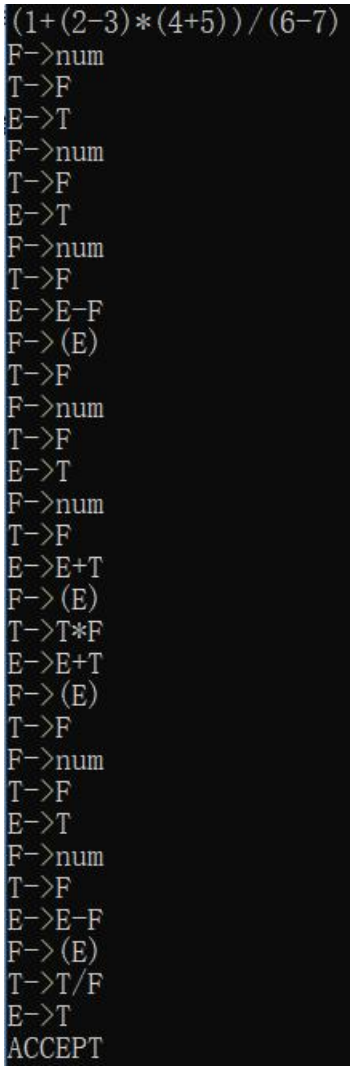
（1）无错误案例

(1+(2-3)*(4+5))/(6-7)

（2）含错误案例

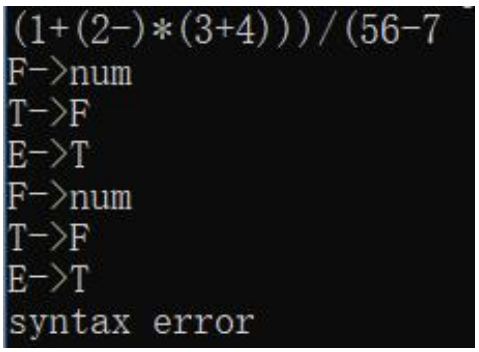
(1+(2-)*(3+4)))/(56-7

对于无错误案例，程序输出采用的产生式如下：



由于 YACC 生成的是 LALR(1)语法分析程序，在输入符号串无错误的情况下，采用的文法生成式和 LR(1)语法分析程序相同。经检验，程序输出正确。

对于含错误案例，程序输出采用的产生式如下：



由于编写的 YACC 源程序较为简单，没有自定义错误处理，因此程序在遇到第 7 个符号处错误时程序就会停止分析。

七、实验总结

本次课程实验中，本人同时使用递归下降，LL(1)，LR(1)，YACC 自动生成四种方法实现了语法分析。代码均由本人一人完成，耗费了不少时间，在实验过程中也遇到了不少的问题。

第一个问题是 LL(1)语法分析程序中的文法该如何存储，存储文法生成式时如果直接存储符号串会在后续的分析中为了识别非终结符和终结符浪费过多的计算。使用数组索引代替具体符号是我能够想出的最优解决方案，同时在文法符号串中分别用正数和负数代表终结符和非终结符也能够更为简单的区分两种符号。

第二个问题就是构造 first 集，follow 集和分析表，构造 first 集和 follow 集都需要获得符号串的 first 集，分析符号串的 first 过程中在遇到非终结符时不只需要将其 first 集加入，还要判断其 first 集中是否包含ε来确定是否还需要分析下一个符号。这一点在最开始被我忽略了，导致一直无法发现程序错误，最后再多次检查书中算法后发现了此错误。

第三个问题就是 LR(1)分析表的构建，在构建 DFA 时，由于不仔细导致缺少状态或缺少向前看符号，前前后后修改了好多次。

第四个问题就是 LR(1)分析表的存储，一开始由于没有考虑错误处理，考虑到二维数组中含有过多的空表项，就想使用 map 来存储稀疏矩阵。但后来填补上错误处理的表项后，发现还是使用二维数组存储最好。

总的来说，从本次试验中学习到不少东西，对于语法分析的各种方法有了更深的理解，同时也提升了自己的代码水平。

不过还有几点不足，首先没有实现 LL(1)语法分析程序中的自动去除左递归，其次没有实现 LR(1)语法分析程序的自动生成识别所有活前缀的 DFA 和分析表，最后 YACC 程序没有实现错误处理，这些遗憾都将在之后时间充裕的时候补充实现，