

一、实验题目

二、实验要求

三、方法1实现

3.1 改写文法

3.2 为每个非终结符号构造转换图

3.3 转换图的化简

3.4 构造预测分析程序

3.5 程序测试

四、方法2实现

4.0 数据结构

4.1 消除文法的左递归

4.2 提取文法的左公因子

4.3 构造文法的FIRST集

4.4 构造文法的FOLLOW集

4.5 构造文法的预测分析表（算法4.2）

4.6 构造LL(1)预测分析程序（算法4.1）

4.7 程序测试

五、方法3实现

5.0 要求

5.1 拓广文法

5.2 构造LR(0)项目集

5.3 构造LR(0)项目集族与GO数组

5.4 构造SLR(1)分析表

5.5 构造LR分析控制程序

5.6 构造图形化界面，打印分析过程与语法树

六、方法4实现

6.0 要求

6.1 YACC的概念

6.2 YACC语法结构

符号

定义段

规则段

用户子例程段

动作

6.3 YACC对冲突的处理

6.4 Lex的概念

6.5 Lex 的常规表达式

6.6 Lex编程

- 6.6.1 全局声明举例
- 6.6.2 模式举例
- 6.6.3 补充的 C 函数举例
- 6.7 Lex的使用
- 6.8 Lex程序设计
 - 6.8.1 Lex模式设计
 - 6.8.2 辅助变量与函数设计
 - 6.8.3 完整代码
- 6.9 编写YACC代码
- 6.10 YACC与Lex综合使用
- 6.11 程序测试

一、实验题目

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

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid num$$

二、实验要求

在对输入的算术表达式进行分析的过程中，依次输出所采用的产生式。

- 方法1：编写递归调用程序实现自顶向下的分析。
- 方法2：编写LL(1)语法分析程序，要求如下。（必做）
 - a. 编程实现算法4.2，为给定文法 自动构造预测分析表。
 - b. 编程实现算法4.1，构造LL(1)预测分析程序。
- 方法3：编写语法分析程序实现自底向上的分析，要求如下。（必做）
 - a. 构造识别该文法所有活前缀的DFA。
 - b. 构造该文法的LR分析表。
 - c. 编程实现算法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。

三、方法1实现

3.1 改写文法

添加非终结符 E', T' ，消除左递归：

$$E \rightarrow TE'$$

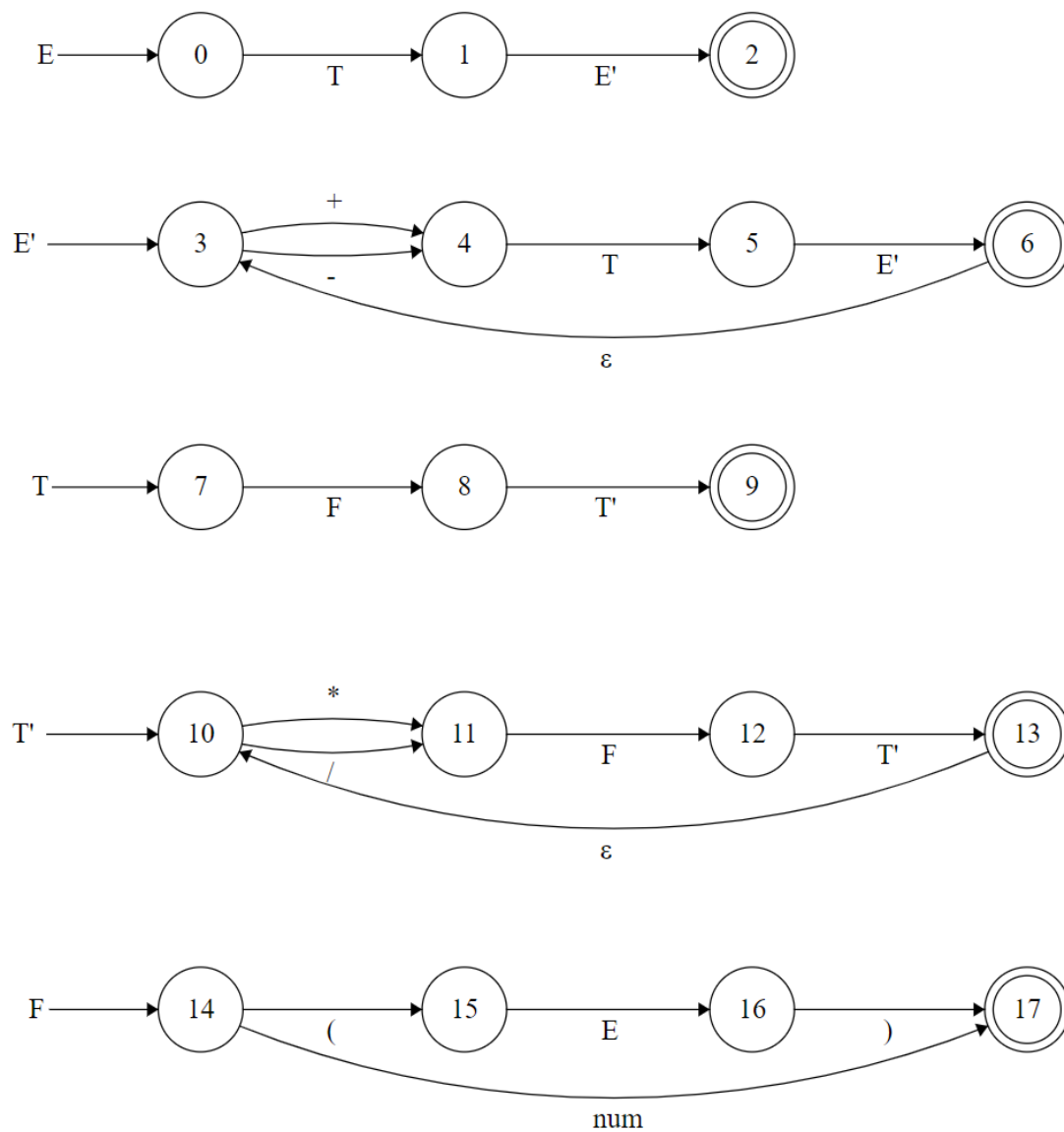
$$E' \rightarrow +TE' \mid -TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid /FT' \mid \varepsilon$$

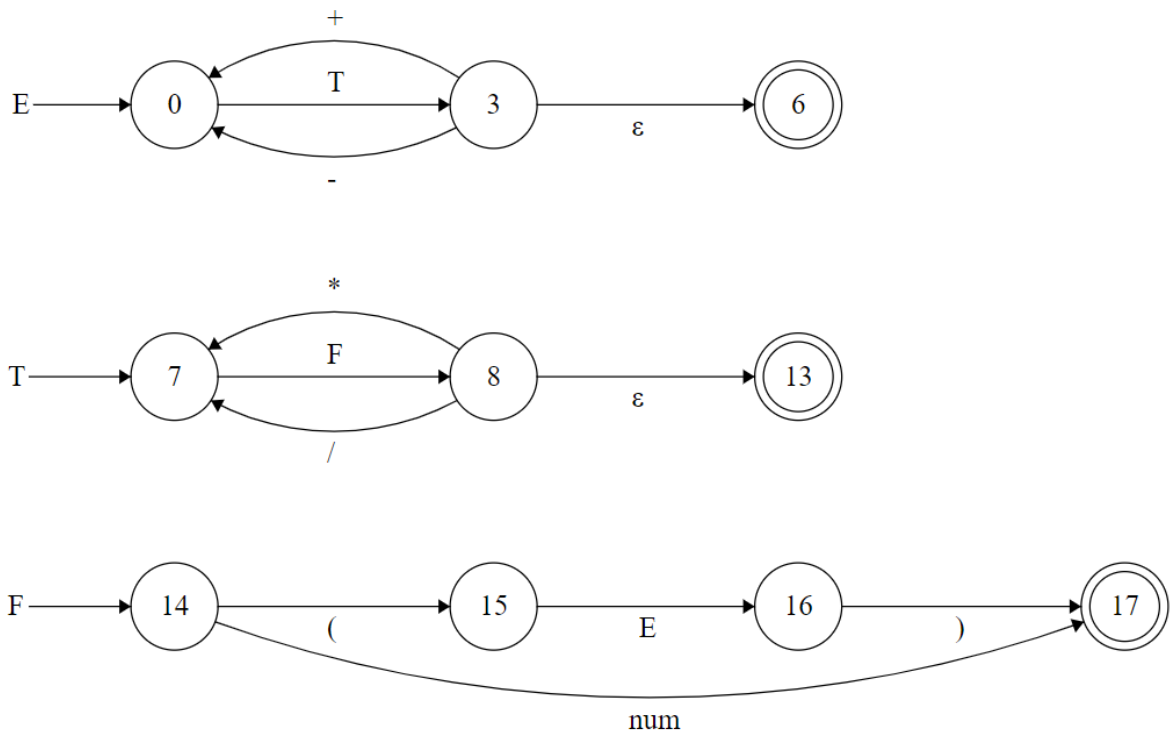
$$F \rightarrow (E) \mid num$$

3.2 为每个非终结符号构造转换图



3.3 转换图的化简

用代入法进行化简，得到：



3.4 构造预测分析程序

对于E:

```
void procE()
{
    procT();
    if (S[point] == '+' || S[point] == '-') {
        point++;
        procE();
    }
}
```

对于T:

```

void proct()
{
    procF();
    if (S[point] == '*' || S[point] == '/') {
        point++;
        proct();
    }
}

```

对于F:

```

void procF() {
    if (S[point] == '(')
    {
        point++;
        procE();
        if (S[point] == ')')
            point++;
        else
            end(0);
    }
    else if (S[point] >= '0' && S[point] <= '9')
    {
        while (S[point] >= '0' && S[point] <= '9')
            point++;
    }
    else
        end(0);
}

```

完整代码：

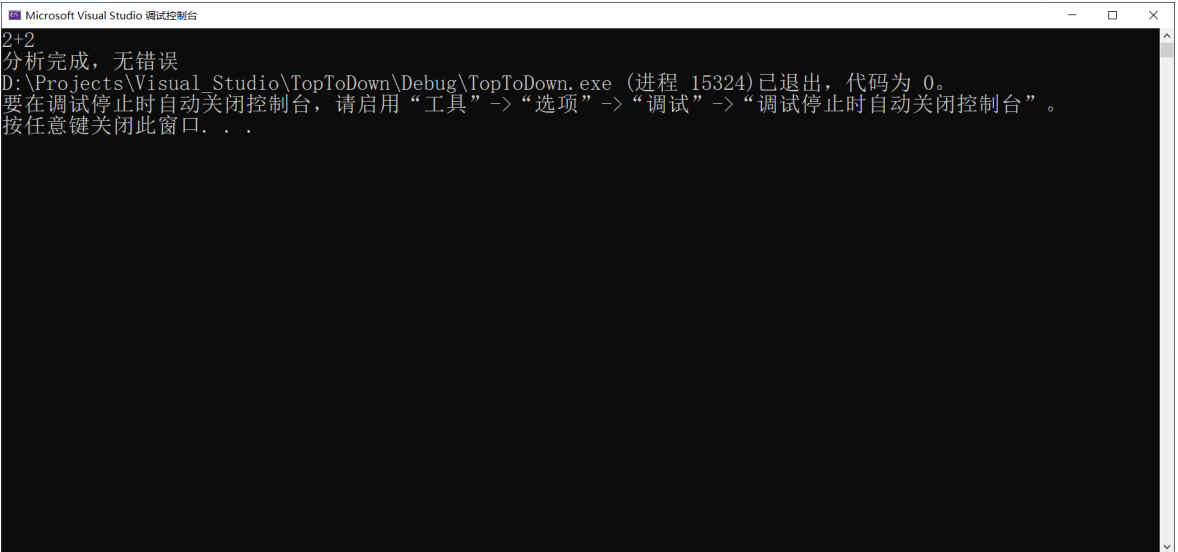
```
#include <iostream>
#include <string>
using namespace std;
string S;
int point = 0;
void procE();
void procF();
void procT();
void end(int result)
{
    if (result == 1)
        cout << "分析完成，无错误";
    else
    {
        cout << "发生错误!" << endl;
        cout << "错误出现在字符串'" << S.substr(0, point) <<
"'末尾处" << endl;
    }
    exit(0);
}
void procE()
{
    procT();
    if (S[point] == '+' || S[point] == '-') {
        point++;
        procE();
    }
}
void procT()
{
    procF();
    if (S[point] == '*' || S[point] == '/') {
        point++;
        procT();
    }
}
```

```
void procF() {
    if (S[point] == '(')
    {
        point++;
        procE();
        if (S[point] == ')')
            point++;
        else
            end(0);
    }
    else if (S[point] >= '0' && S[point] <= '9')
    {
        while (S[point] >= '0' && S[point] <= '9')
            point++;
    }
    else
        end(0);
}

int main()
{
    cin >> S;
    S += '$';
    procE();
    if (S[point] == '$')
        end(1);
    else
        end(0);
}
```

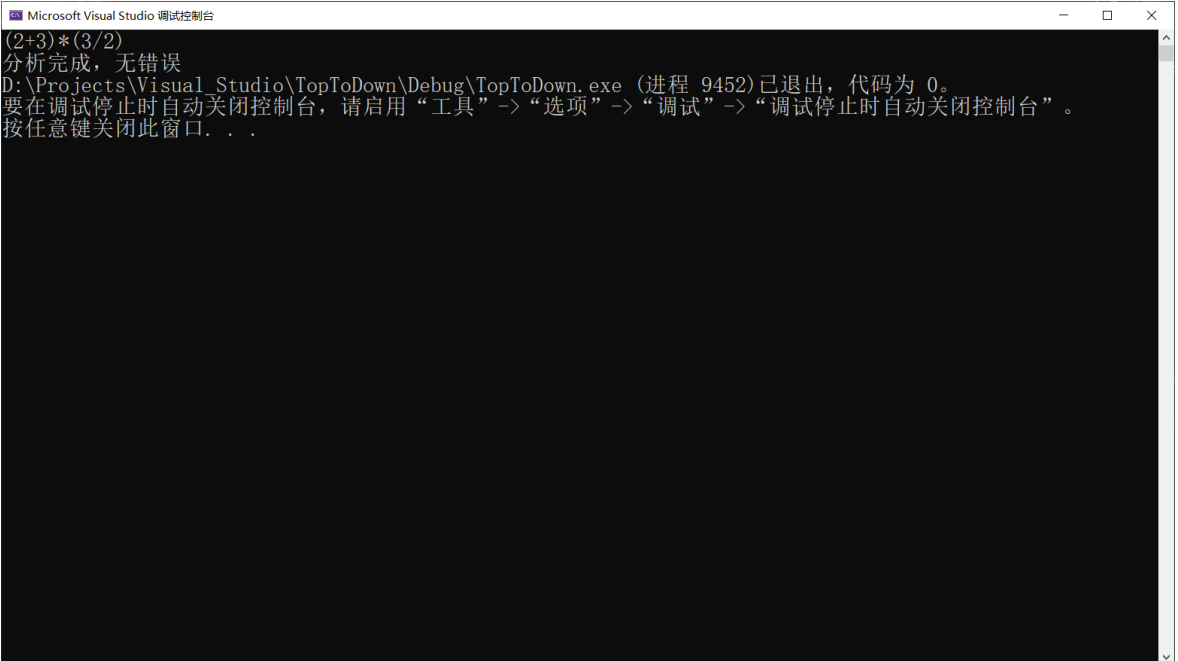

3.5 程序测试

1. $2+2$



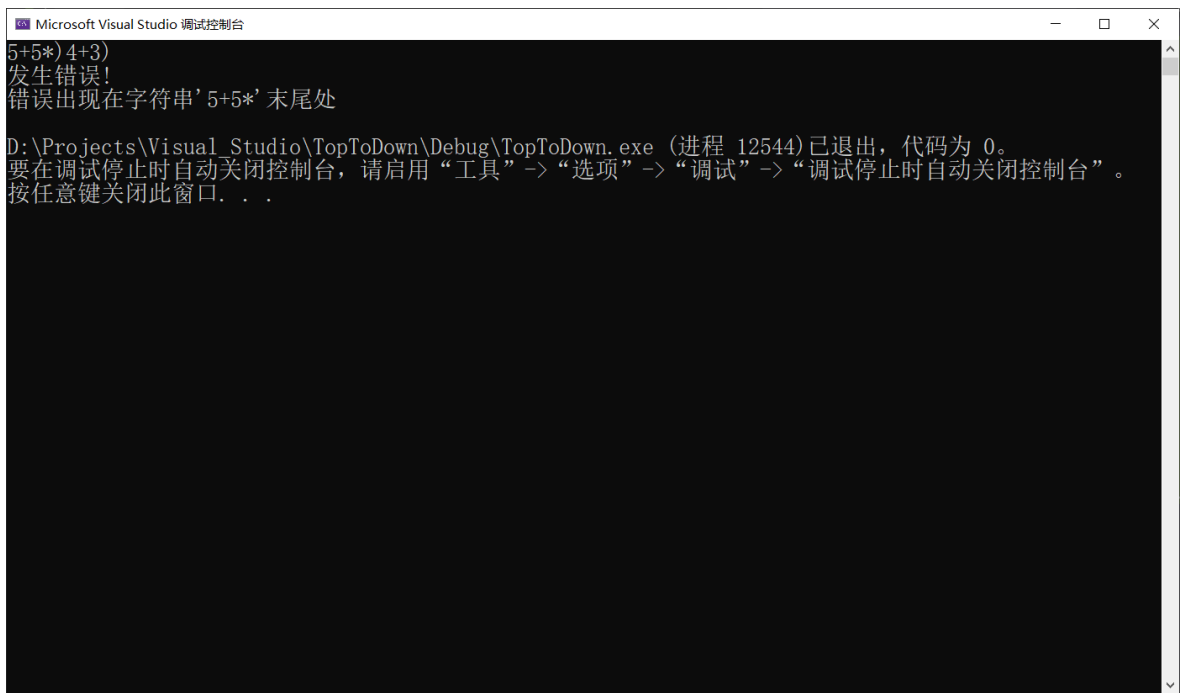
```
Microsoft Visual Studio 调试控制台
2+2
分析完成，无错误
D:\Projects\Visual_Studio\TopToDown\Debug\TopToDown.exe (进程 15324) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。...
```

2. $(2+3) * (3/2)$



```
Microsoft Visual Studio 调试控制台
(2+3)*(3/2)
分析完成，无错误
D:\Projects\Visual_Studio\TopToDown\Debug\TopToDown.exe (进程 9452) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。...
```

3. $5+5 *)4+3)$



四、方法2实现

4.0 数据结构

```
string BeginSym;  
set <string> termin, non_termin;  
map <string, set<string> > first, follow;  
set <string> created_first, created_follow;  
vector<string> boostsplit(const string & input);  
struct Proce;  
vector<Proce> proce;  
map <string, vector<Proce> > proce_index;  
map <string, map<string, Proce> > Table;  
stack<string> Stack;  
struct Proce  
{  
    string ori;
```

```

string left;
vector<string> right;
Proce() {};
Proce(const string & _p)
{
    ori = _p;
    vector<string> syms = boostsplit(_p);
    left = syms[0];
    int l = syms.size();
    FOR(i, 2, l - 1)
        right.push_back(syms[i]);
    proce_index[left].push_back(*this);
}
};

```

4.1 消除文法的左递归

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid -TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid /FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid num$$

4.2 提取文法的左公因子

本文法中不存在左公因子，无需提取

4.3 构造文法的FIRST集

对任意文法符号串，其FIRST集为其可推导出的开头终结符号集合。

构造任意文法符号 X 的FIRST集 $FIRST(X)$ 可遍历 X 的产生式并利用如下规则：

(1) 若 $X \in VT$ ，则 $FIRST(X) = \{X\}$ ；

(2) 若 $X \in VN$ ，且有

$X \rightarrow a \dots (a \in VT \vee a = \varepsilon)$ ，则 $FIRST(X) = FIRST(X) \cup \{a\}$ ；

(3) 若有 $X \rightarrow Y_1 Y_2 \dots Y_k Y_{k+1} \dots$ ，且有 $\varepsilon \in FIRST(Y_i) (i = 1, \dots, k)$ ，

则 $FIRST(X) = FIRST(X) \cup \{a \in FIRST(Y_i) | a \neq \varepsilon\}$ ；

若 $\forall Y_i, \varepsilon \in FIRST(Y_i)$ ， $FIRST(X) = FIRST(X) \cup \varepsilon$ 。

```
void CreateFirst(const string & sym)
{
    vector<Proce> & Pv = proce_index[sym];

    for (vector<Proce>::iterator it = Pv.begin(); it !=
Pv.end(); ++it)
    {
```

```

    Proce &P = (*it);
    string left = P.left;
    bool all_epsilon = true;
    for (vector<string>::iterator j = P.right.begin(); j
!= P.right.end(); j++)
    {

        if (termin.find(*j) != termin.end())
        {
            first[left].insert(*j);
            break;
        }

        if (created_first.find(*j) ==
created_first.end())
        {
            CreateFirst(*j);
        }

        first[left].insert(first[*j].begin(),
first[*j].end());

        if (first[*j].find("epsilon") ==
first[*j].end())
        {
            all_epsilon = false;
            break;
        }
        else
            first[left].erase("epsilon");

        if (all_epsilon)
            first[left].insert("epsilon");
    }
    created_first.insert(sym);
}

```

求解本题文法的FIRST集如下：

```
Microsoft Visual Studio 调试控制台
E: ( num
E': + - epsilon
F: ( num
T: ( num
T': * / epsilon

D:\Projects\Visual_Studio\LL(1)\Debug\LL(1).exe (进程 8884) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。 . . .
```

非终结符号	FIRST
E	$\{ (, num \}$
E'	$\{ +, -, \epsilon \}$
T	$\{ (, num \}$
T'	$\{ *, /, \epsilon \}$
F	$\{ (, num \}$

4.4 构造文法的FOLLOW集

对任意非终结符，其FOLLOW集是该文法所有句型中紧跟在其后的终结符或结尾符号\$的集合。

构造任意文法符号 X 的FOLLOW集 $FOLLOW(X)$ 可遍历文法所有产生式并利用如下规则：

(1) 若 X 是起始符号，则 $FOLLOW(X) = FOLLOW(X) \cup \{\$ \}$;

(2) 若有 $A \rightarrow \cdots X\beta$ ，则

$FOLLOW(X) = FOLLOW(X) \cup \{a \in FIRST(\beta) | a \neq \varepsilon\}$;

(3) 若有 $A \rightarrow \cdots X$ ，或有 $A \rightarrow \cdots X\beta$ 且 $\varepsilon \in FIRST(\beta)$ ，

则 $FOLLOW(X) = FOLLOW(X) \cup FOLLOW(A)$ 。

```
void CreateFollow()
{
    follow[BeginSym].insert("$");
    bool changed = true;
    while (changed)
    {
        changed = false;
        for (vector<Proce>::iterator it1 = proce.begin();
it1 != proce.end(); it1++)           //遍历所有产生式
        {
            Proce& P = *it1;
            for (vector<string>::iterator it2 =
P.right.begin(); it2 != P.right.end(); it2++)
            {
                string x = *it2;
                int old_size = follow[x].size();
                bool all_epsilon = true;
```

```

        for (vector<string>::iterator it3 = it2 + 1;
it3 != P.right.end() && all_epsilon; it3++)
        {
            string b = *it3;
            bool show_epsilon = false;
            for (set<string>::iterator it4 =
first[b].begin(); it4 != first[b].end(); it4++)
            {
                string a = *it4;
                if (a != "epsilon")
                    follow[X].insert(a);
                else
                    show_epsilon = true;
            }
            all_epsilon &= show_epsilon;
        }
        if (all_epsilon)
            for (set<string>::iterator it5 =
follow[P.left].begin(); it5 != follow[P.left].end(); it5++)
                follow[X].insert(*it5);
        if (follow[X].size() > old_size)
            changed = true;
    }
}
}
}

```

求解本题文法的FOLLOW集如下：


```
Microsoft Visual Studio 调试控制台
First:
E: ( num
E': + - epsilon
F: ( num
T: ( num
T': * / epsilon
Follow:
E: $ )
E': $ )
F: $ ) * + - /
T: $ ) + -
T': $ ) + -

D:\Projects\Visual_Studio\LL(1)\Debug\LL(1).exe (进程 22724) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。...
```

非终结符号	FIRST	FOLLOW
E	$\{ (, num \}$	$\{), \$ \}$
E'	$\{ +, -, \epsilon \}$	$\{), \$ \}$
T	$\{ (, num \}$	$\{ +, -,), \$ \}$
T'	$\{ +, -, \epsilon \}$	$\{ +, -,), \$ \}$
F	$\{ (, num \}$	$\{ +, -, *, /,), \$ \}$

4.5 构造文法的预测分析表（算法4.2）

使用 算法4.2 预测分析表的构造方法，其基本流程如下：

输入：文法G

输出：文法G的预测分析表M

伪代码：

```
for (文法G的每个产生式  $A \rightarrow \alpha$ ) {  
  
    for (每个终结符号  $a \in FIRST(\alpha)$ )  
  
        把  $A \rightarrow \alpha$  放入  $M[A, a]$  中;  
  
    if ( $\epsilon \in FIRST(\alpha)$ )  
  
        for (任何  $b \in FOLLOW(A)$ )  
  
            把  $A \rightarrow \alpha$  放入  $M[A, b]$  中;  
  
};  
for (所有无定义的  $M[A, a]$ ) 标上错误标志。
```

```
void CreateFollow()  
{  
    follow[BeginSym].insert("$");  
    bool changed = true;  
    while (changed)  
    {  
        changed = false;  
        for (vector<Proce>::iterator it1 = proce.begin();  
it1 != proce.end(); it1++)    //遍历所有产生式  
        {  
            Proce& P = *it1;  
            for (vector<string>::iterator it2 =  
P.right.begin(); it2 != P.right.end(); it2++)  
            {  
                string x = *it2;  
                int old_size = follow[x].size();  
                bool all_epsilon = true;
```

```

        for (vector<string>::iterator it3 = it2 + 1;
it3 != P.right.end() && all_epsilon; it3++)
        {
            string b = *it3;
            bool show_epsilon = false;
            for (set<string>::iterator it4 =
first[b].begin(); it4 != first[b].end(); it4++)
            {
                string a = *it4;
                if (a != "epsilon")
                    follow[X].insert(a);
                else
                    show_epsilon = true;
            }
            all_epsilon &= show_epsilon;
        }
        if (all_epsilon)
            for (set<string>::iterator it5 =
follow[P.left].begin(); it5 != follow[P.left].end(); it5++)
                follow[X].insert(*it5);
        if (follow[X].size() > old_size)
            changed = true;
    }
}
}
}

```

得到结果：

```
Microsoft Visual Studio 调试控制台
First:
E: ( num
E': + - epsilon
F: ( num
T: ( num
T': * / epsilon
Follow:
E: $ )
E': $ )
F: $ ) * + - /
T: $ ) + -
T': $ ) + -
M[E, (] = E -> T E'
M[E, num] = E -> T E'
M[E, $] = E' -> epsilon
M[E, )] = E' -> epsilon
M[E, +] = E' -> + T E'
M[E, -] = E' -> - T E'
M[F, (] = F -> ( E )
M[F, num] = F -> num
M[T, (] = T -> F T'
M[T, num] = T -> F T'
M[T, $] = T' -> epsilon
M[T, )] = T' -> epsilon
M[T, *] = T' -> * F T'
M[T, +] = T' -> epsilon
M[T, -] = T' -> epsilon
M[T, /] = T' -> / F T'

D:\Projects\Visual_Studio\LL(1)\Debug\LL(1).exe (进程 17592) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

4.6 构造LL(1)预测分析程序（算法4.1）

```
void Prase(string w)
{
    w += "$";
    int pointer = 0;
    while (!Stack.empty())
        stack.pop();
    Stack.push("$");
    Stack.push(BeginSym);
    while (!Stack.empty())
    {
        string x = Stack.top();
        string a = w.substr(pointer, 1);
        if (w[pointer] >= '0' && w[pointer] <= '9')
        {
            a = "num";
            while (w[pointer + 1] >= '0' && w[pointer + 1]
<= '9') pointer++;

```

```

    }
    if (termin.find(X) != termin.end())
    {
        if (X == a)
        {
            stack.pop();
            pointer++;
        }
        else
        {
            error(w, pointer);
            return;
        }
    }
    else
    {
        if (Table.find(X) != Table.end() &&
Table[X].find(a) != Table[X].end())
        {
            stack.pop();
            Proce P = Table[X][a];
            int length = P.right.size();
            for (int i = length - 1; i >= 0; i--)
            {
                string Y = P.right[i];
                if (Y != "epsilon")
                    stack.push(Y);
            }
            cout << P.ori << endl;
        }
        else
        {
            error(w, pointer);
            return;
        }
    }
}
}

```

```

        cout << "分析完成! " << endl;
    }

    void error(string w, int pointer)
    {
        cout << "出现错误! 错误出现在字符串'" << w.substr(0, pointer)
        << "'结尾处" << endl;
    }

```

4.7 程序测试

输入 1+4

```

1+4
E -> T E'
T -> F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> num
T' -> epsilon
E' -> epsilon
分析完成!

```

输入 4*4+3+(3+244)/2

```
4*4+3+(3+244)/2
E -> T E'
T -> F T'
F -> num
T' -> * F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> num
T' -> epsilon
E' -> epsilon
T' -> / F T'
F -> num
T' -> epsilon
E' -> epsilon
分析完成!
```

输入4+88)

```
4+88)
E -> T E'
T -> F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> num
T' -> epsilon
E' -> epsilon
出现错误! 错误出现在字符串'4+88' 结尾处
```

检测到错误，并打印错误信息。

五、方法3实现

5.0 要求

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。

拓展：使用图形化界面画出语法分析树

5.1 拓广文法

加入起始非终结符 S 与文法 $S \rightarrow E$ ，将起始符号改为 S


```

void Extend()
{
    non_termin.insert("S");
    Proce P = Proce("S -> " + BeginSym);
    BeginSym = "S";
    proce.push_back(P);
    set<Item> its;
    its.insert(Item(P, 0));
    ItemSet its0 = ItemSet(its);
    its0.extend();
    item_sets.push_back(its0);
}

```

5.2 构造LR(0)项目集

在LL(1)文法中的产生式结构体**Struct Proce**的基础上，添加圆点位置**dot**，成为LR项目**Item**

```

struct Item
{
    Proce pro;
    int dot, tot_len, belong;
    Item() {};
    Item(Proce _p, int _dot)
    {
        pro = _p;
        dot = _dot;
        tot_len = _p.right.size();
    }
    bool operator< (const Item& rhs) const
    {
        return (pro == rhs.pro && dot < rhs.dot) || pro <
rhs.pro;
    }
}

```

```

    }
    bool operator== (const Item& rhs) const
    {
        return pro == rhs.pro && dot == rhs.dot;
    }

};

```

重载小于号与等于号，便于其排序与判断集合相同。

项目集合构成项目集 `ItemSet`，其包含一个 `Item` 类型的 `std::set`，存储项目集合。还包含了一个构造闭包的方法 `extend`

```

struct ItemSet
{
    set<Item> item_set;
    ItemSet() {};
    ItemSet(set<Item> _item_set) : item_set(_item_set) {};
    void extend()
    {
        int ori_num = 0;
        while (item_set.size() > (UINT)ori_num)
        {
            ori_num = item_set.size();
            for (set<Item>::iterator it1 = item_set.begin();
it1 != item_set.end(); it1++)
            {
                Item I = *it1;
                Proce Pa = I.pro;

```

```

        if ((UINT)I.dot < Pa.right.size() &&
non_termin.find(Pa.right[I.dot]) != non_termin.end())
        {

            string B = Pa.right[I.dot];

            for (vector<Proce>::iterator it2 =
proce_index[B].begin(); it2 != proce_index[B].end(); it2++)
            {

                Proce Pb = *it2;
                Item new_i = Item(Pb, 0);

                item_set.insert(new_i);

            }

        }

    }

}

bool operator==(const ItemSet& rhs) const
{
    if (item_set.size() != rhs.item_set.size()) return
false;
    for (set<Item>::iterator it1 = item_set.begin(); it1
!= item_set.end(); it1++)
    {
        if (rhs.item_set.find(*it1) ==
rhs.item_set.end()) return false;
    }
    return true;
}

};

```

5.3 构造LR(0)项目集族与GO数组

用一个map存储项目集间的转移关系：

```
map<string, int>go[1000];
```

在构造项目集族时，利用了广度优先搜索（BFS）。维护一个还未计算Go数组的项目集族队列Q。初始Q只存在项目集 I_0 （ I_0 为 $S \rightarrow \cdot E$ 构造出的闭包）开始，考虑输入所有可能的符号产生的新项目集，判断是否已经存在该集合，若不存在，则将其加入队列尾部。

```
void CreateGo()
{
    queue<int> Q;
    Q.push(0);
    while (!Q.empty())
    {
        ItemSet IS = item_sets[Q.front()];
        Q.pop();

        vector<ItemSet>::iterator it0 =
find(item_sets.begin(), item_sets.end(), IS);
        int id = it0 - item_sets.begin();

        for (set<string>::iterator it1 = termin.begin(); it1
!= termin.end(); it1++)
        {
```

```

        string a = *it1;
        set<Item> its;
        for (set<Item>::iterator it2 =
IS.item_set.begin(); it2 != IS.item_set.end(); it2++)
        {
            Item I = *it2;
            Proce P = I.pro;
            if ((UINT)I.dot < P.right.size() &&
P.right[I.dot] == a)
                its.insert(Item(P, I.dot + 1));
        }
        if (its.empty()) continue;
        ItemSet n_IS(its);
        n_IS.extend();
        vector<ItemSet>::iterator it3;
        for (it3 = item_sets.begin(); it3 !=
item_sets.end(); it3++) if (*it3 == n_IS) break;
        int n_id = it3 - item_sets.begin();
        if (it3 == item_sets.end())
        {
            item_sets.push_back(n_IS);
            Q.push(n_id);
            go[id][a] = n_id;
        }
        else
        {
            go[id][a] = n_id;
        }
    }

    for (set<string>::iterator it1 = non_termin.begin();
it1 != non_termin.end(); it1++)
    {
        string a = *it1;
        set<Item> its;
        for (set<Item>::iterator it2 =
IS.item_set.begin(); it2 != IS.item_set.end(); it2++)
        {

```

```

        Item I = *it2;
        Proce P = I.pro;
        if ((UINT)I.dot < P.right.size() &&
P.right[I.dot] == a)
            its.insert(Item(P, I.dot + 1));
    }
    if (its.empty()) continue;
    ItemSet n_IS(its);
    n_IS.extend();
    vector<ItemSet>::iterator it3;
    for (it3 = item_sets.begin(); it3 !=
item_sets.end(); it3++) if (*it3 == n_IS) break;
    int n_id = it3 - item_sets.begin();
    if (it3 == item_sets.end())
    {
        item_sets.push_back(n_IS);
        Q.push(n_id);
        go[id][a] = n_id;
    }
    else
    {
        go[id][a] = n_id;
    }
}
}
}

```

针对本题的文法，构造出的项目集组与go数组为：

```

I0:
E->·E+T
E->·E-T
E->·T

```

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{num}$

$S \rightarrow \cdot E$

$T \rightarrow \cdot F$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot T / F$

I1:

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot E - T$

$E \rightarrow \cdot T$

$F \rightarrow \cdot (E)$

$F \rightarrow (\cdot E)$

$F \rightarrow \cdot \text{num}$

$T \rightarrow \cdot F$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot T / F$

I2:

$F \rightarrow \text{num} \cdot$

I3:

$E \rightarrow E \cdot + T$

$E \rightarrow E \cdot - T$

$S \rightarrow E \cdot$

I4:

$T \rightarrow F \cdot$

I5:

$E \rightarrow T \cdot$

$T \rightarrow T \cdot * F$

$T \rightarrow T \cdot / F$

I6:

$E \rightarrow E \cdot + T$

$E \rightarrow E \cdot - T$

$F \rightarrow (E \cdot)$

I7:

$E \rightarrow E+ \cdot T$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{num}$

$T \rightarrow \cdot F$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot T / F$

I8:

$E \rightarrow E- \cdot T$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{num}$

$T \rightarrow \cdot F$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot T / F$

I9:

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{num}$

$T \rightarrow T * \cdot F$

I10:

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{num}$

$T \rightarrow T / \cdot F$

I11:

$F \rightarrow (E) \cdot$

I12:

$E \rightarrow E+T \cdot$

$T \rightarrow T \cdot * F$

$T \rightarrow T \cdot / F$

I13:

$E \rightarrow E-T \cdot$

$T \rightarrow T \cdot * F$

$T \rightarrow T \cdot / F$

I14:

$T \rightarrow T * F \cdot$

I15:

$T \rightarrow T / F \cdot$

$GO[0, (] = 1$

$GO[0, \text{num}] = 2$

$GO[0, E] = 3$

$GO[0, F] = 4$

$GO[0, T] = 5$

$GO[1, (] = 1$

$GO[1, \text{num}] = 2$

$GO[1, E] = 6$

$GO[1, F] = 4$

$GO[1, T] = 5$

$GO[3, +] = 7$

$GO[3, -] = 8$

$GO[5, *] = 9$

$GO[5, /] = 10$

$GO[6,)] = 11$

$GO[6, +] = 7$

$GO[6, -] = 8$

$GO[7, (] = 1$

$GO[7, \text{num}] = 2$

$GO[7, F] = 4$

$GO[7, T] = 12$

$GO[8, (] = 1$

```
GO[8 , num] = 2
```

```
GO[8 , F] = 4
```

```
GO[8 , T] = 13
```

```
GO[9 , (] = 1
```

```
GO[9 , num] = 2
```

```
GO[9 , F] = 14
```

```
GO[10 , (] = 1
```

```
GO[10 , num] = 2
```

```
GO[10 , F] = 15
```

```
GO[12 , *] = 9
```

```
GO[12 , /] = 10
```

```
GO[13 , *] = 9
```

```
GO[13 , /] = 10
```

5.4 构造SLR(1)分析表

由于本项目集族存在移进-规约冲突（如 I_12 ）：

I_{12} :

$E \rightarrow E+T \cdot$

$T \rightarrow T \cdot *F$

$T \rightarrow T \cdot /F$

于是尝试构造SLR(1)分析表。

SLR(1)分析表的构造方式为:

1. 构造拓广文法 G' 的LR(0)项目集规范族 $C = \{I_0, I_1, \dots, I_n\}$ 。
2. 对于状态 i （对应于项目集 I_i 的状态）的分析动作如下
 - a. 若 $A \rightarrow \alpha \cdot a\beta \in I_i$ ，且 $go(I_i, a) = I_j$ ，则置 $action[i, a] = Sj$
 - b. 若 $A \rightarrow \alpha \cdot \in I_i$ ，则对所有 $a \in FOLLOW(A)$ ，置 $action[i, a] = R \quad A \rightarrow \alpha$
 - c. 若 $S \rightarrow S' \cdot \in I_i$ ，则置 $action[i, \$] = ACC$ ，表示分析成功
3. 若 $go(I_i, A) = I_j$ ，A为非终结符号，则置 $goto[i, A] = j$
4. 分析表中凡不能用规则(2)、(3)填入信息的空白表项，均置为出错标志error。
5. 分析程序的初态是包含项目 $S \rightarrow S'$ 的有效项目集所对应的状态

Action的存储类型为:

```
struct ActionOp
{
    string type;
    int val;
    ActionOp() {}
    ActionOp(string _type, int _val) : type(_type),
    val(_val) {}
};
```

Action与Goto存储在:

```
vector <map<string, ActionOp> > Action;  
vector <map<string, int> > Goto;
```

构造SLR(1)分析表:

```
void CreateTable()  
{  
    Action.resize(item_sets.size());  
    Goto.resize(item_sets.size());  
    FOR(i, 0, (int)(item_sets.size() - 1))  
    {  
        ItemSet IS = item_sets[i];  
        for (set<Item>::iterator it1 = IS.item_set.begin();  
it1 != IS.item_set.end(); it1++)  
        {  
            Item I = *it1;  
            Proce P = I.pro;  
            if ((UINT)I.dot < P.right.size())  
            {  
                string a = P.right[I.dot];  
                if (termin.find(a) != termin.end())  
                    Action[i][a] = ActionOp("SHIFT", go[i]  
[a]);  
                else  
                    Goto[i][a] = go[i][a];  
            }  
            else  
            {  
                string A = P.left;  
                if (A == BeginSym)  
                    Action[i]["$"] = ActionOp("ACC", 0);  
                else  
                {
```

```

        for (set<string>::iterator it2 =
follow[A].begin(); it2 != follow[A].end(); it2++)
        {
            string a = *it2;
            int pid = P.id;
            Action[i][a] = ActionOp("REDUCE",
pid);
        }
    }
}
}
}
}

```

构造结果:

```

Action[0 , (] = S1
Action[0 , num] = S2
Action[1 , (] = S1
Action[1 , num] = S2
Action[2 , $] = R7
Action[2 , )] = R7
Action[2 , *] = R7
Action[2 , +] = R7
Action[2 , -] = R7
Action[2 , /] = R7
Action[3 , $] = ACC
Action[3 , +] = S7
Action[3 , -] = S8
Action[4 , $] = R5
Action[4 , )] = R5
Action[4 , *] = R5
Action[4 , +] = R5

```

Action[4 , -] = R5
Action[4 , /] = R5
Action[5 , \$] = R2
Action[5 ,)] = R2
Action[5 , *] = S9
Action[5 , +] = R2
Action[5 , -] = R2
Action[5 , /] = S10
Action[6 ,)] = S11
Action[6 , +] = S7
Action[6 , -] = S8
Action[7 , (] = S1
Action[7 , num] = S2
Action[8 , (] = S1
Action[8 , num] = S2
Action[9 , (] = S1
Action[9 , num] = S2
Action[10 , (] = S1
Action[10 , num] = S2
Action[11 , \$] = R6
Action[11 ,)] = R6
Action[11 , *] = R6
Action[11 , +] = R6
Action[11 , -] = R6
Action[11 , /] = R6
Action[12 , \$] = R0
Action[12 ,)] = R0
Action[12 , *] = S9
Action[12 , +] = R0
Action[12 , -] = R0
Action[12 , /] = S10
Action[13 , \$] = R1
Action[13 ,)] = R1
Action[13 , *] = S9
Action[13 , +] = R1
Action[13 , -] = R1
Action[13 , /] = S10
Action[14 , \$] = R3

```
Action[14 , )] = R3
Action[14 , *] = R3
Action[14 , +] = R3
Action[14 , -] = R3
Action[14 , /] = R3
Action[15 , $] = R4
Action[15 , )] = R4
Action[15 , *] = R4
Action[15 , +] = R4
Action[15 , -] = R4
Action[15 , /] = R4
```

```
Goto[0 , E] = 3
Goto[0 , F] = 4
Goto[0 , T] = 5
Goto[1 , E] = 6
Goto[1 , F] = 4
Goto[1 , T] = 5
Goto[7 , F] = 4
Goto[7 , T] = 12
Goto[8 , F] = 4
Goto[8 , T] = 13
Goto[9 , F] = 14
Goto[10 , F] = 15
```

经检验，构造出的SLR（1）分析表不存在冲突。

5.5 构造LR分析控制程序

输入：文法 G 的一张分析表和一个输入符号串 w

输出：若 $w \in L(G)$ ，得到 w 的自底向上的分析，否则报错

方法：开始时，初始状态 S_0 在栈顶， $w\$$ 在输入缓冲区中。

```
置 ip 指向 w$ 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；
        推进ip，使它指向下一个输入符号；
    };

    else if (action[S, a]==reduce by A→b) {

        从栈顶弹出 |β| 个符号；    // 令S'是现在的栈顶状态

        把 A 和 goto[S', A]分别压入符号栈和状态栈；

        输出产生式 A→β；

    };

    else if (action[S, a]==accept) return;

    else error();

} while(1).
```



```

void Prase(string w)
{
    int step = 0;
    w += "$";
    int pointer = 0;
    stack<int> state_stack;
    stack<string> sym_stack;
    state_stack.push(0);
    sym_stack.push(BeginSym);
    while (!state_stack.empty())
    {
        cout << setw(3) << left << ++step << " ";
        print_stack_state(state_stack);
        cout << setw(10) << right << w.substr(pointer) << "
";

        int S = state_stack.top();
        string a = w.substr(pointer, 1);
        if (w[pointer] >= '0' && w[pointer] <= '9')
        {
            a = "num";
            while (w[pointer + 1] >= '0' && w[pointer + 1]
<= '9') pointer++;
        }
        if (Action[S][a].type == "SHIFT")
        {
            cout << "移进" << a << endl;
            cout << setw(4) << " ";
            print_stack_sym(sym_stack);
            state_stack.push(Action[S][a].val);
            sym_stack.push(a);
            pointer++;
        }
        else if (Action[S][a].type == "REDUCE")
        {
            Proce P = proce[Action[S][a].val];
            cout << "使用产生式" << P.ori << "规约" << endl;

```

```

        cout << setw(4) << " ";
        print_stack_sym(sym_stack);
        FOR(i, 1, (int)P.right.size())
        {
            state_stack.pop();
            sym_stack.pop();
        }
        sym_stack.push(P.left);
        int n_state = state_stack.top();
        state_stack.push(Goto[n_state][P.left]);

    }
    else if (Action[S][a].type == "ACC")
    {
        cout << "分析完成" << endl;
        cout << setw(4) << " ";
        print_stack_sym(sym_stack);
        return;
    }
    else
        error(w, pointer);

}
}

```

5.6 构造图形化界面，打印分析过程与语法树

1、符号串1+2+3

请输入文法

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

输入完毕

显示分析过程

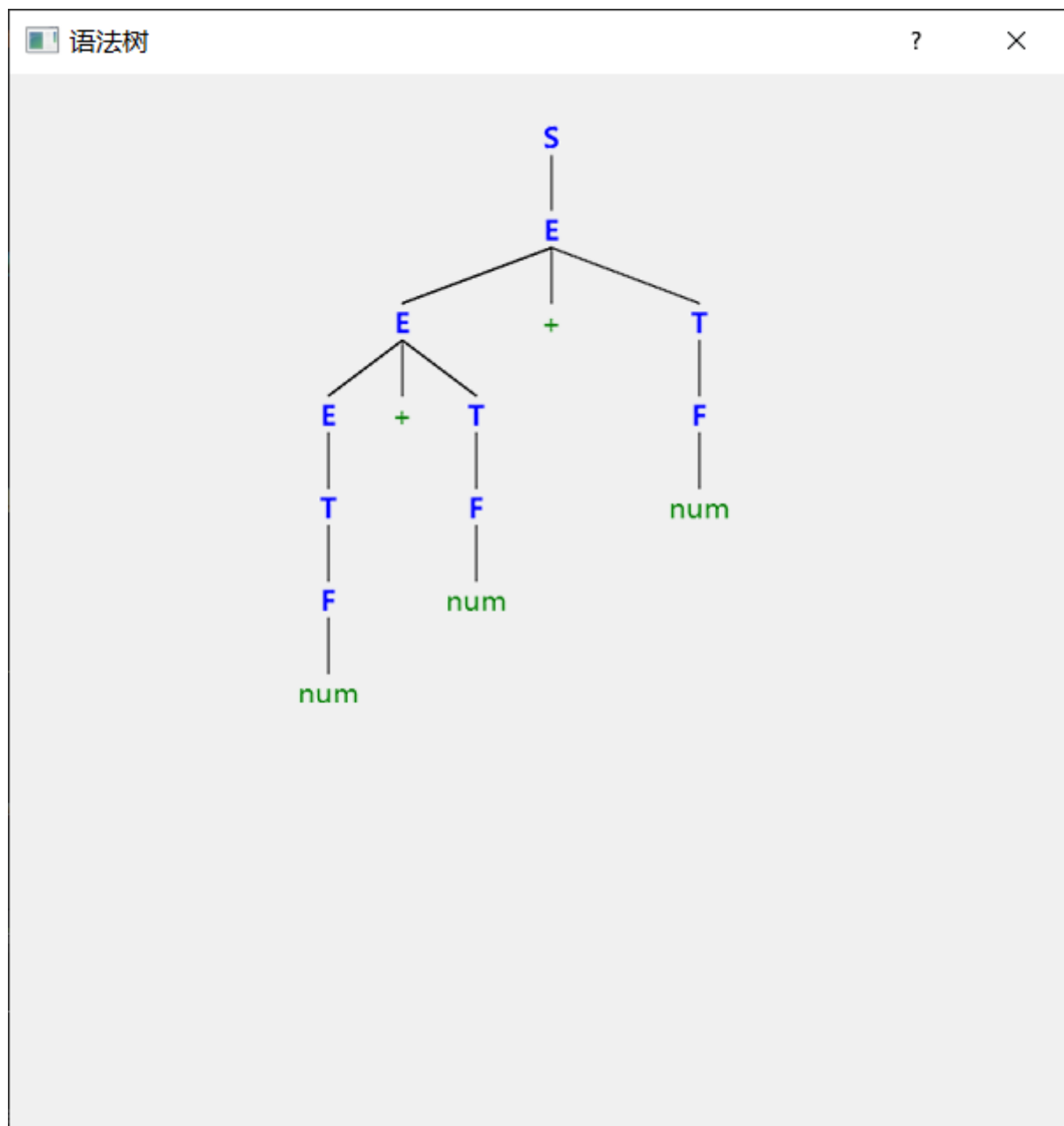
显示语法树

请输入单词串

1+2+3

输入完毕

步骤	栈	剩余符号	动作
1	0 S	1+2+3\$	移进num
2	0 2 S num	+2+3\$	使用产生式F \rightarrow num规约
3	0 4 S F	+2+3\$	使用产生式T \rightarrow F规约
4	0 5 S T	+2+3\$	使用产生式E \rightarrow T规约
5	0 3 S E	+2+3\$	移进+
6	0 3 7 S E +	2+3\$	移进num
7	0 3 7 2 S E + num	+3\$	使用产生式F \rightarrow num规约
8	0 3 7 4 S E + F	+3\$	使用产生式T \rightarrow F规约
9	0 3 7 12 S E + T	+3\$	使用产生式E \rightarrow E + T规约
10	0 3 S E	+3\$	移进+
11	0 3 7 S E +	3\$	移进num
12	0 3 7 2 S E + num	\$	使用产生式F \rightarrow num规约
13	0 3 7 4 S E + F	\$	使用产生式T \rightarrow F规约
14	0 3 7 12 S E + T	\$	使用产生式E \rightarrow E + T规约
15	0 3 S E	\$	分析完成



2、符号串 $2^*(3+4)/7$

请输入文法

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

输入完毕

显示分析过程

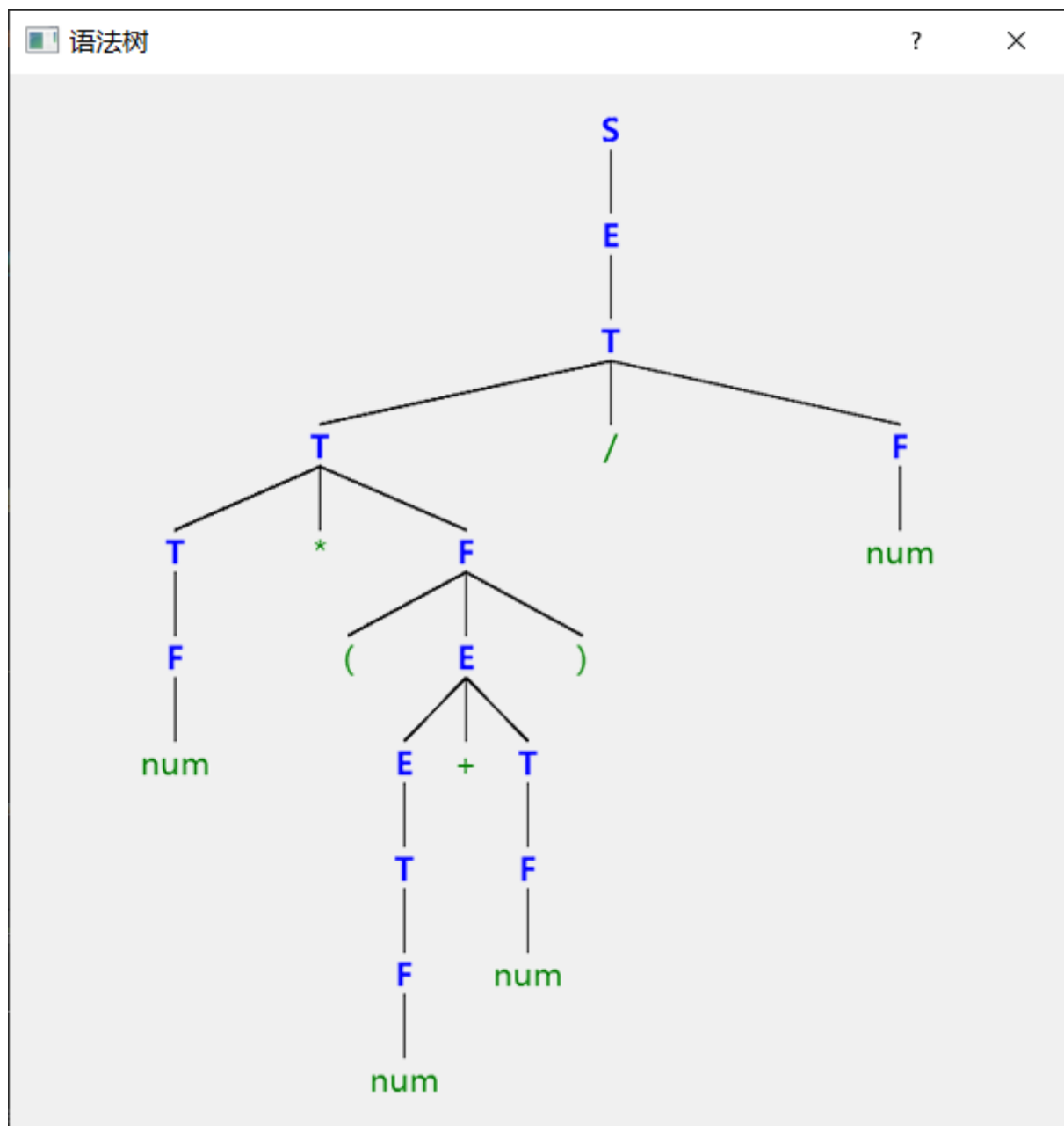
显示语法树

请输入单词串

 $2*(3+4)/7$

输入完毕

步骤	栈	剩余符号	动作
1	0	2*(3+4)/7\$	移进num
2	0 2 S num	*(3+4)/7\$	使用产生式F → num规约
3	0 4 S F	*(3+4)/7\$	使用产生式T → F规约
4	0 5 S T	*(3+4)/7\$	移进*
5	0 5 9 S T *	(3+4)/7\$	移进(
6	0 5 9 1 S T * (3+4)/7\$	移进num
7	0 5 9 1 2 S T * (num	+4)/7\$	使用产生式F → num规约
8	0 5 9 1 4 S T * (F	+4)/7\$	使用产生式T → F规约
9	0 5 9 1 5 S T * (T	+4)/7\$	使用产生式E → T规约
10	0 5 9 1 6 S T * (E	+4)/7\$	移进+
11	0 5 9 1 6 7 S T * (E +	4)/7\$	移进num
12	0 5 9 1 6 7 2 S T * (E + num)/7\$	使用产生式F → num规约
13	0 5 9 1 6 7 4 S T * (E + F)/7\$	使用产生式T → F规约
14	0 5 9 1 6 7 12 S T * (E + T)/7\$	使用产生式E → E + T规约
15	0 5 9 1 6 S T * (E)/7\$	移进)
16	0 5 9 1 6 11 S T * (E)	/7\$	使用产生式F → (E)规约
17	0 5 9 14 S T * F	/7\$	使用产生式T → T * F规约
18	0 5 S T	/7\$	移进/
19	0 5 10 S T /	7\$	移进num
20	0 5 10 2 S T / num	\$	使用产生式F → num规约
21	0 5 10 15 S T / F	\$	使用产生式T → T / F规约
22	0 5 S T	\$	使用产生式E → T规约
23	0 3 S E	\$	分析完成



六、方法4实现

6.0 要求

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。

6.1 YACC的概念

yacc(Yet Another Compiler Compiler)，是Unix/Linux上一个用来生成编译器的编译器（编译器代码生成器）。

使用巴克斯范式(BNF)定义语法，能处理上下文无关文法(context-free)。出现在每个产生式左边(left-hand side: lhs)的符号是非终端符号，出现在产生式右边(right-hand side: rhs)的符号有非终端符号和终端符号，但终端符号只出现在右端。

yacc是开发编译器的一个有用的工具,采用LR（1）（实际上是LALR(1)）语法分析方法。

LR(k)分析方法是1965年Knuth提出的，括号中的k（ $k \geq 0$ ）表示向右查看输入串符号的个数。LR分析法正视图给出一种能根据当前分析栈中的符号串和向右顺序查看输入串的k个符号就可唯一确定分析器的动作是移进还是规约和用哪个产生式规约。

这种方法具有分析速度快，能准确，即使地指出出错的位置，它的主要缺点是对于一个使用语言文法的分析器的构造工作量相当大，k愈大构造愈复杂，实现比较困难。

一个LR分析器有3个部分组成：

- 总控程序，也可以称为驱动程序。

对所有的LR分析器总控程序都是相同的。

- 分析表或分析函数。

不同的文法分析表将不同，同一个文法采用的LR分析器不同时，分析表也不同，分析表又可分为动作(ACTION)表和状态转换(GOTO)表两个部分。

- 分析栈，包括文法符号栈和相应的状态栈。

它们均是先进后出栈。分析器的动作由栈顶状态和当前输入符号所决定。

6.2 YACC语法结构

yacc语法包括三部分：定义段、规则段和用户子例程段

```
...定义段...
```

```
%%
```

```
...规则段...
```

```
%%
```

```
...用户子例程段...
```

各部分由以两个百分号开头的行分开，尽管某一个部分可以为空，但是前两部分是必须的，第三部分和前面的百分号可以省略。

符号

yacc 语法由符号组成，即语法的“词”。符号是一串不以数字开头的字母、数字、句点和下划线。符号`error`专用于错误恢复，另外，yacc对任何符号都不会附加“先验”的意义。

由词法分析程序产生的符号叫做终结符号或者标记。定义在规则左侧的叫做非终结符号或者非终结。标记也可能是字面上引用的字符，通常遵循约定：标记大写，非终结符号小写。

定义段

定义段包括文字块，逐字拷贝到生成的C文件开头部分的C代码，通常包括声明和`#include`行。可能有`%union %start %token %type %left %right` 和 `%nonassoc`声明。

也可以包含普通的C语言风格的注释，所有这些都是可选的，在简单的语法分析程序中，定义段可能完全是空的。

规则段

规则段由语法规则和包括C代码的动作组成。

规则中目标或非终端符放在左边，后跟一个冒号（:），然后是产生式的右边，之后是对应的动作（用{}包含）。如：

用户子例程段

`yacc` 将用户子例程段的内容完全拷贝到C文件中，通常这部分包括从动作调用的例程。

该部分是函数部分。当`yacc`解析出错时，会调用函数`yyerror()`，用户可自定义函数的实现。

`main`函数是调用`yacc`解析入口函数`yyparse()`。如：

动作

动作是yacc与在语法中规则相符时执行的C代码，动作一定是C复合语句。

动作有4种可能：

- 移进：

当 $S_j = \text{GOTO}[S_i, a]$ 成立，则把 S_j 移入到状态栈，把 a 移入到文法符号栈。
其中 i, j 表示状态号。

- 规约：

当在栈顶形成句柄为 β 时，则用 β 归约为相应的非终结符 A ，即当文法中有 $A \rightarrow \beta$ 的产生式，而 β 的长度为 r （即 $|\beta| = r$ ），则从状态栈和文法符号栈中自栈顶向下去掉 r 个符号，即栈指针 SP 减去 r 。并把 A 移入文法符号栈内，再把满足 $S_j = \text{GOTO}[S_i, A]$ 的状态移进状态栈，其中 S_i 为修改指针后的栈顶状态。

- 接受acc：

当规约到文法符号栈只剩文法的开始符号 S 时，并且输入符号串已结束即当前输入符是‘#’，则为分析成功。

- 报错：

当遇到状态栈顶为某一状态下出现不该遇到的文法符号时，则报错，说明输入串不是该文法能接受的句子。

6.3 YACC对冲突的处理

当存在二义性时，LALR 算法将会出现语法分析动作冲突。对于 YACC 来说，默认在解决规约 / 规约冲突时，选择规则中前面那个冲突进行分析；在解决移入 / 规约冲突时总是选择移入。

6.4 Lex的概念

Lex 是一种生成扫描器的工具。扫描器是一种识别文本中的词汇模式的程序。这些词汇模式（或者常规表达式）在一种特殊的句子结构中定义，这个我们一会儿就要讨论。

一种匹配的常规表达式可能会包含相关的动作。这一动作可能还包括返回一个标记。当 **Lex** 接收到文件或文本形式的输入时，它试图将文本与常规表达式进行匹配。它一次读入一个输入字符，直到找到一个匹配的模式。如果能够找到一个匹配的模式，**Lex** 就执行相关的动作（可能包括返回一个标记）。另一方面，如果没有可以匹配的常规表达式，将会停止进一步的处理，**Lex** 将显示一个错误消息。

Lex 和 **C** 是强耦合的。一个 *.lex* 文件（**Lex** 文件具有 *.lex* 的扩展名）通过 **lex** 公用程序来传递，并生成 **C** 的输出文件。这些文件被编译为词法分析器的可执行版本。

6.5 Lex 的常规表达式

字符	含义
A-Z, 0-9, a-z	构成了部分模式的字符和数字。
.	匹配任意字符，除了 \n 。
-	用来指定范围。例如： A-Z 指从 A 到 Z 之间的所有字符。
[]	一个字符集合。匹配括号内的 任意 字符。如果第一个字符是 ^ 那么它表示否定模式。例如： [abC] 匹配 a , b , 和 C 中的任何一个。
*	匹配 0 个或者多个上述的模式。
+	匹配 1 个或者多个上述模式。
?	匹配 0 个或 1 个上述模式。
\$	作为模式的最后一个字符匹配一行的结尾。
{ }	指出一个模式可能出现的次数。 例如： A{1,3} 表示 A 可能出现 1 次或 3 次。

字符	含义
*	用来转义元字符。同样用来覆盖字符在此表中定义的特殊意义，只取字符的本意。
^	否定。
	表达式间的逻辑或。
"<一些符号">	字符的字面含义。元字符具有。
/	向前匹配。如果在匹配的模版中的“/”后跟有后续表达式，只匹配模版中“/”前面的部分。如：如果输入 A01，那么在模版 A0/1 中的 A0 是匹配的。
()	将一系列常规表达式分组。

6.6 Lex编程

Lex 编程可以分为三步：

1. 以 Lex 可以理解的格式指定模式相关的动作。
2. 在这一文件上运行 Lex，生成扫描器的 C 代码。
3. 编译和链接 C 代码，生成可执行的扫描器。

一个 Lex 程序分为三个段：第一段是 C 和 Lex 的全局声明，第二段包括模式（C 代码），第三段是补充的 C 函数。这些段以%%来分界。

6.6.1 全局声明举例

```
%{
    int tot_token = 0;
    int tot_error=0;
    int col = 0,line= 1;
%}
```

这里声明了四个整型变量`tot_token,tot_error,col,line`，分别代表符号总数，错误总数，列号，行号。

6.6.2 模式举例

<code>letter</code>	<code>[A-Za-z]</code>
<code>digit</code>	<code>[0-9]</code>

`letter`的模式为`[A-Za-z]`，即所有的大写或小写字母

`digit`的模式为`[0-9]`，即0~9之间任意数字

6.6.3 补充的 C 函数举例

```
void addcol(int cnt) {  
    col += cnt;  
}
```

我在这里定义了一个列数增加函数，作用是根据当前识别出的符号的长度增加当前列数

6.7 Lex的使用

编写好Lex的程序后，将其保存为.l后缀的文件。首先要使用flex生成lex语法对应的c文件：

```
C:\Users\HTY\iCloudDrive\iCloud~QReader~MarginStudy\编译原理与技术\实验>flex lex.l
```

生成的文件名为lex.yy.c

然后将这个c文件用GNU编译

```
C:\Users\HTY\iCloudDrive\iCloud~QReader~MarginStudy\编译原理与技术\实验>gcc lex.yy.c -o lex.exe
```

如果编译无误，将会生成lex.exe可执行文件

最后，输入测试程序文件名调用lex.exe即可

```
C:\Users\HTY\iCloudDrive\iCloud~QReader~MarginStudy\编译原理与技术\实验>lex.exe <test.c >out.txt
```

结果保存在out.txt中

6.8 Lex程序设计

考虑在”词法分析“实验中设计的九类符号

- 1、标识符 IDENTIFIER
- 2、关键字 KEYWORD
- 3、常量字符 CONSTANT_CHAR
- 4、常量字符串 CONSTANT_STRING
- 5、常整数 CONSTANT_INT
- 6、常实数 CONSTANT_REAL

7、操作符号 SIGNAL

8、行注释 LINE_COMMENT

9、换行符 LINE_BREAK

6.8.1 Lex模式设计

以上九种模式对应的正则表达式为：

1. 标识符 IDENTIFIER

标识符的特征为有字母开头的、字母数字组成的字符串

```
{letter}({letter}|{digit})*
```

2. 关键字 KEYWORD

这里直接枚举了所有C语言关键字：

```
"char"|"int"|"long"|"float"|"double"|"void"|"unsigned"|"signed"|"const"|"static"|"extern"|"struct"|"union"|"typedef"|"sizeof"|"if"|"else"|"do"|"while"|"for"|"switch"|"case"|"default"|"continue"|"break"|"goto"
```

3. 常量字符 CONSTANT_CHAR

考虑普通字符与转义字符两种情况，普通字符为两个单引号之间包裹一个非单引号、非反斜杠、非转义字符的字符，可以用 `[^'\n]` 表示这个集合；转义字符为反斜杠后跟一个可以被转义的字符

```
'[^'\n]|'\\[0nrtvabf\\'\\"\\']'
```

4. 常量字符串 CONSTANT_STRING

```
\"[^\"]*\"
```

5. 常整数 CONSTANT_INT

```
{digit}+
```

6. 常实数 CONSTANT_REAL

```
{digit}*(\\. {digit}+)?(e|E[+\\-]?{digit}+)?
```

7. 操作符号 SIGNAL

这里直接枚举了所有C语言的合法符号：
"="|"+"|"++|"+="|"-"|"--|"-"
="|"*="|"/="|"/="|"%"|"%=|"&"|"&="|"|"|"="|"^"|"^="|"~"|"
<<"|<="|>>"|>="|"&&"|"&&="|"|"|"|"="|"!"|<"|"
<="|"=="|"!="|>"|>="|##"|"?"|","|:"|;"|."|"*"|"-"|"
(")|"|"["|"]"|"{"|"}"

8. 行注释 LINE_COMMENT

行注释的开头两个/符号需要被转义

(\\/. *\\n)

9. 换行符 LINE_BREAK

\\n

6.8.2 辅助变量与函数设计

```
void addline() { //换行时行数加1
    line ++;
}
void clearcol() { //换行时列数置1
    col = 1;
}
void addcol(int cnt) { //识别完一个符号后列数加上这个符号的长度
    col += cnt;
}
void addword() { //识别完一个符号符号总数加1
    tot_token ++;
}
```

6.8.3 完整代码

```
%{  
    int tot_token = 0;  
    int charCounter=0;  
    int col =1,line=1;  
    void addline();  
    void addcol(int);  
    void clearcol();  
    void addword();  
%}  
  
letter                [A-Za-z_]  
digit                 [0-9]  
LINE_COMMENT          (\\/\\. *\\n)  
CONSTANT_CHAR         '[^']*'|'\\'[0nrtvabf\\'\"\\\']'  
CONSTANT_STRING       \"[^\"]*\"  
  
IDENTIFIER            {letter}({letter}|{digit})*  
KEYWORD                 
    "char"|"int"|"long"|"float"|"double"|"void"|"unsigned"|"sig  
ned"|"const"|"static"|"extern"|"struct"|"union"|"typedef"|"s  
izeof"|"if"|"else"|"do"|"while"|"for"|"switch"|"case"|"defau  
lt"|"continue"|"break"|"goto"  
CONSTANT_INT          {digit}+  
CONSTANT_REAL         {digit}*\\. {digit}+)?(e|E[+-]?  
{digit}+)?  
SIGNAL                "="|"+"|"++"|"+="|"-"|--"|"--  
="|"*="|"/"|"/="|% "|"%= "&" "&="|" "| |= "|^"|^="|~"|"  
<< "|<=> "> ">=" "&&" "&&="|" | | |= "|! "|< "|  
<=" |==" |!=" |> "|>=" |##" |?" |," |: ":" ;" |"." |"*" |"-> "| "  
(" |")" | "[" | "]" | "{" | "}"  
LINE_BREAK            \\n  
  
%%  
  
{LINE_COMMENT} {  
    printf ("< LINE_COMMENT , \\\");  
    int i = 0;
```

```

while (i < yyleng - 1)
    printf ("%c", yytext[i++]);
printf ("\\" > in ( %d , %d)\n", line, col);
addcol(yyleng);
addword();
}

{CONSTANT_CHAR} {
    printf("< LINE_COMMENT , %s > in ( %d , %d)\n",yytext,
line, col);
    addcol(yyleng);
    addword();
}

{CONSTANT_STRING} {
    printf("< LINE_STRING , %s > in ( %d , %d)\n",yytext,
line, col);
    addcol(yyleng);
    addword();
}

{KEYWORD} {
    printf("< KEYWORD , \"%s\" > in ( %d , %d)\n",yytext,
line, col);
    addcol(yyleng);
    addword();
}

{IDENTIFIER} {
    printf("< IDENTIFIER , \"%s\" > in ( %d , %d)\n",yytext,
line, col);
    addcol(yyleng);
    addword();
}

{CONSTANT_INT} {
    printf("< CONSTANT_INT , \"%s\" > in ( %d ,
%d)\n",yytext, line, col);
    addcol(yyleng);
    addword();
}

```

```

{CONSTANT_REAL} {
    printf("< CONSTANT_REAL , \"%s\" > in ( %d ,
%d)\n",yytext, line, col);
    addcol(yyvaleng);
    addword();
}

{SIGNAL} {
    printf("< SIGNAL , \"%s\" > in ( %d , %d)\n",yytext,
line, col);
    addcol(yyvaleng);
    addword();
}

{LINE_BREAK} {
    addline();
    clearcol();
}

%%

int main(void)
{
    printf ("开始分析\n");
    yylex();
    printf("统计结果: 共 %d 行, 共 %d 个单词\n",line,
tot_token);
    return 0;
}

int yywrap() {
    return 1;
}

void addline() {
    line ++;
}

void addcol(int cnt) {
    col += cnt;
}

void clearcol() {

```

```

        col = 1;
    }
    void addword()    {
        tot_token ++;
    }

```

6.9 编写YACC代码

```

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

%token NUM

%left '+' '-'
%left '*' '/'

%%
S      : E                {printf("Ans = %d\n", $1);}
      ;
E      : E '+' T          {$$ = $1+$3; printf("E->E+T\n");}
      | E '-' T          {$$ = $1-$3; printf("E->E-T\n");}
      | T                {$$ = $1; printf("E->T\n");}
      ;
T      : T '*' F          {$$ = $1*$3; printf("T->T*F\n");}
      | T '/' F          {$$ = $1/$3; printf("T->T/F\n");}
      | F                {$$ = $1; printf("T->F\n");}
      ;
F      : '(' E ')'        {$$ = $2; printf("F->(E)\n");}
      | NUM              {$$ = $1; printf("F->NUM\n");}

%%

```

```
int main(){
    return yyparse();
}
```

（这里使用简化版词法分析）：

```
%{
# include "y.tab.h"
%}

%%
[0-9]+ {yyval = atoi(yytext);return NUM;}
[-/+*()^\n] {return yytext[0];}
[ \t\r\n]+ {/*ignore all space*/}
. {return 0;}
%%
int yywrap(void){
return 1;
}
```

6.10 YACC与Lex综合使用

1. 生成词法文件

```
D:\Programs\ToolProgram\GnuWin32\bin>flex lex.l
```

2. 生成语法文件

```
D:\Programs\ToolProgram\GnuWin32\bin>bison -dyv yacc.y
```

3. 联合编译

```
D:\Programs\ToolProgram\GnuWin32\bin>gcc lex.yy.c y.tab.c -o cal
```

4. 使用目标文件运算

```
D:\Programs\ToolProgram\GnuWin32\bin>cal
1+2
F->NUM
T->F
E->T
F->NUM
T->F
E->E+T
Ans = 3
```

6.11 程序测试

$3*(2+6)/4$

```
D:\Programs\ToolProgram\GnuWin32\bin>cal
3*(2+6)/4
F->NUM
T->F
F->NUM
T->F
E->T
F->NUM
T->F
E->E+T
F->(E)
T->T*F
F->NUM
T->T/F
E->T
Ans = 6
```

$((5+3)*5)+7-(9/3)$


```
D:\Programs\ToolProgram\GnuWin32\bin>cal
(( (5+3)*5)+7-(9/3))
F->NUM
T->F
E->T
F->NUM
T->F
E->E+T
F->(E)
T->F
F->NUM
T->T*F
E->T
F->(E)
T->F
E->T
F->NUM
T->F
E->E+T
F->NUM
T->F
F->NUM
T->T/F
E->T
F->(E)
T->F
E->E-T
F->(E)
T->F
E->T
Ans = 44
```