编译原理

LL1 语法分析器实验报告

编译环境： Microsoft Visual Studio 2019

Windows SDK **版本：** 10.0

平台工具集： Visual Studio 2019（v142）

语言： C++

1、需求：LL1 语法分析程序的设计与实现。

题目：语法分析程序的设计与实现。

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。

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

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

$F \rightarrow (E) \mid \text{num}$

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。

(1) 编程实现算法 4.2，为给定文法自动构造预测分析表。

(2) 编程实现算法 4.1，构造 LL(1)预测分析程序。

拓展需求：还能自动够造 LL1 文法的 first 集和 follow 集，为 LL1 文法自动构造预测分析表。

2、核心算法：

①整个程序的模型如图所示：

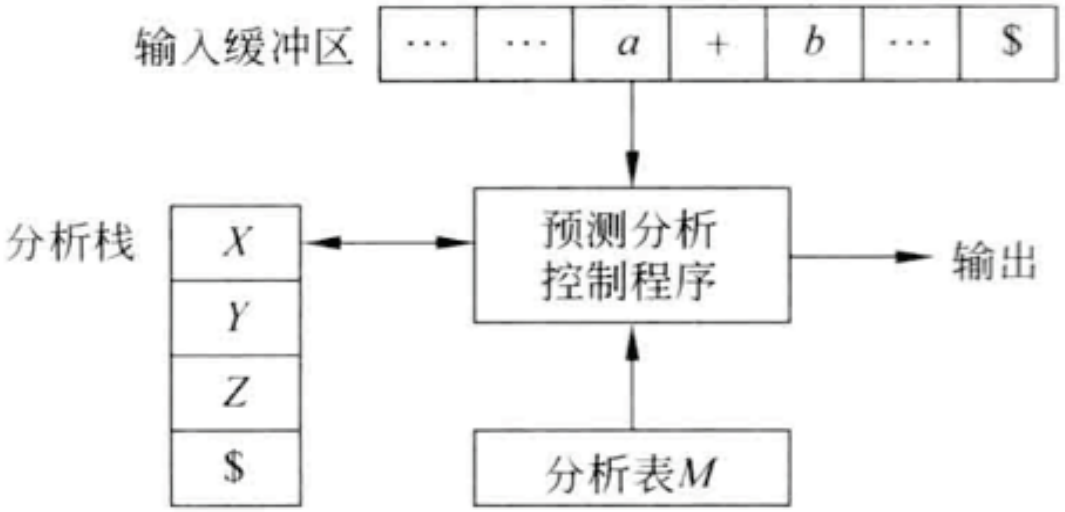


图 4-8　非递归预测分析程序的模型

分析栈、分析表的数据结构详见【3、变量和函数】。

②first 集的构造

使用深度优先遍历递归的方法，当遍历完所有的非终结符的产生式右部，first 集构造完成。详见注释。

```cpp
void DFS(int x)//为构造first集深度优先遍历PF，递归调用
{
    if (used[x])
        return;
    used[x] = 1;
    string& left = PF_vector[x].left;
    set<string>& right = PF_vector[x].right;
    set<string>::iterator it = right.begin();
    for (; it != right.end(); it++)//对于非终结符的产生式的所有右部
    {
        for (int i = 0; i < it->length(); i++)//对于这个右部的每个字符
        {
            if (!isupper(it->at(i)))//如果不是大写字母，即为终结符，直接加入到first集中
            {
                first[left].insert(it->at(i));
                break;//已经是终结符，不必再深度搜索，跳出
            }
            else//如果是大写字母，即为非终结符
            {
                int y;
                y = PF_map[it->substr(i, 1)] - 1;
                string& tleft = PF_vector[y].left;
                DFS(y);//递归调用
                set<char>& temp = first[tleft];
                set<char>::iterator it_point = temp.begin();
                for (; it_point != temp.end(); it_point++)
                {
                    if (*it_point!='~')//把所有不为空的first集加入
                    {
                        first[left].insert(*it_point);
                    }
                }
                set<char>::iterator it_temp = temp.find('~');//查找first集中有没有空
                if (it_temp!=temp.end())//如果存在空
                {
                    if (i == it->length() - 1)//并且该符号是当前产生式的最后一个符号，把空加入到待求first集中
                    {
                        first[left].insert('~');
                    }
                    //如果该符号不是当前产生式的最后一个符号，则继续递归查找
                }
                else//如果不存在空，可以停止递归，直接跳出
                    break;
            }
        }
    }
}
```

```cpp
void first_construction()//构造并输出first集
{
    memset(used, 0, sizeof(used));
    for (int i = 0; i < PF_vector.size(); i++)
        DFS(i);//输出构造好的first集
    printf("----------------------FIRST集----------------------\n");
    map<string, set<char> >::iterator it = first.begin();
    for (; it != first.end(); it++)
    {
        printf("%s:{", it->first.c_str());
        set<char>& temp = it->second;
        set<char>::iterator it_temp = temp.begin();
        bool flag = false;
        for (; it_temp != temp.end(); it_temp++)
        {
            if (flag) printf(",");
            printf("%c", *it_temp);
            flag = true;
        }
        printf("}\n");
    }
}
```

③follow 集的构造

不能采用深度优先遍历的方法，否则求给出的测试样例【其它文法 1】会无限循环，因为非终结符之间的 follow 集是互相依赖的，因而采用了另一种方法解决了这个问题。就是无限循环求所有非终结符的 follow 集，直到所有的 follow 集都不再增大为止，跳出循环。详见注释。

```
void add_follow(const string& str1, const string& str2)//将str1的follow集加入到str2的follow集中
{
    set<char>& from = follow[str1];
    set<char>& to = follow[str2];
    set<char>::iterator it = from.begin();
    for (; it != from.end(); it++)
        to.insert(*it);
}
```

```
void follow_construction()//构造并输出follow集
{
    follow[PF_vector[0].left].insert('$');//首先将'$'加入到起始符的follow集中，这里默认第一个输入的产生式的左部为起始符
    while (true)//一直循环求follow集，直到所有的非终结符的follow集都没有变化为止
    {
        bool not_finished = false;//使用一个bool类型的变量纪录follow集是否变大
        for (int i = 0; i < PF_vector.size(); i++)//遍历所有的非终结符
        {
            string& left = PF_vector[i].left;
            set<string>& right = PF_vector[i].right;
            set<string>::iterator it = right.begin();
            for (; it != right.end(); it++)//遍历所有非终结符的产生式
            {
                bool flag = true;//用来判断是否要将产生式左边非终结符的follow集加入到当前查找的非终结符的follow集中
                const string& str = *it;
                for (int j = it->length() - 1; j >= 0; j--)//遍历整个产生式右部，且从后往前看
                {
                    if (isupper(str[j]))//是非终结符
                    {
                        int x = PF_map[it->substr(j, 1)] - 1;
                        if (flag)//判断是否要将产生式左边非终结符的follow集加入到当前查找的非终结符的follow集中
                        {
                            int num_old = follow[it->substr(j, 1)].size();
                            add_follow(left, it->substr(j, 1));
                            if (!PF_vector[x].right.count("~"))//如果当前的非终结符的产生式中不含空，说明所有它前面的非终结符均不可能将
                                flag = false;                  //产生式左边非终结符的follow集加入到当前查找的非终结符的follow集中
                            int num_new = follow[it->substr(j, 1)].size();
```

```
                            int num_new = follow[it->substr(j, 1)].size();
                            if (num_new > num_old)//如果follow集没有变化，说明follow集已经构造完成
                                not_finished = true;
                        }
                        for (int k = j + 1; k < it->length(); k++)
                        {
                            if (isupper(str[k]))//是非终结符
                            {
                                string id;
                                id = it->substr(k, 1);
                                set<char>& from = first[id];
                                set<char>& to = follow[it->substr(j, 1)];

                                set<char>::iterator it_temp = from.begin();
                                int num_old = follow[it->substr(j, 1)].size();
                                for (; it_temp != from.end(); it_temp++)//把待求非终结符的后面的非终结符的first集中不为空的终结符加入待求非终结符的follow集中
                                    if (*it_temp != '~')
                                        to.insert(*it_temp);
                                int num_new = follow[it->substr(j, 1)].size();
                                if (num_new > num_old) //如果follow集没有变化，说明follow集已经构造完成
                                    not_finished = true;

                                if (!PF_vector[PF_map[id] - 1].right.count("~"))//如果待求非终结符的后面的非终结符的first集中没有空，就不必往后看，直接跳出
                                    break;
                            }
                            else//是终结符
                            {
                                int num_old = follow[it->substr(j, 1)].size();
                                follow[it->substr(j, 1)].insert(str[k]);//直接将待求非终结符后面的终结符加入到待求非终结符的follow集中
                                int num_new = follow[it->substr(j, 1)].size();
                                if (num_new > num_old)//如果follow集没有变化，说明follow集已经构造完成
                                    not_finished = true;
                                break;//因为待求非终结符后面是终结符，不必再往后看，直接跳出
                            }
                        }
                    }
                    else//是终结符
                        flag = false;//如果中途遇到终结符，就不可能将产生式左边非终结符的follow集加入到当前查找的非终结符的follow集中
                }
            }
        }
        if (!not_finished)//如果所有的非终结符的follow集都没有变化，说明所有的follow集已经构造完成，跳出while循环
            break;
    }
    //输出构造好的follow集
```

```
    //输出构造好的follow集
    printf("----------------------FOLLOW集----------------------\n");
    map<string, set<char> >::iterator it = follow.begin();
    for (; it != follow.end(); it++)
    {
        printf("%s:{", it->first.c_str());
        set<char>& temp = it->second;
        set<char>::iterator it_temp = temp.begin();
        bool flag = false;
        for (; it_temp != temp.end(); it_temp++)
        {
            if (flag) printf(",");
            printf("%c", *it_temp);
            flag = true;
        }
        printf("}\n");
    }
}
```

④预测分析表的构造算法是利用课本上的算法 4.2

算法 4.2 预测分析表的构造方法

输入：文法 G。

输出：文法 G 的预测分析表 M。

方法：

```
for(文法 G 的每一个产生式 A→α) {
    for(每个终结符号 a ∈FIRST(α)) 把 A→α 放入表项 M[A,a]中;
    if( ε ∈FIRST(α))
        for (每个 b ∈FOLLOW(A)) 把 A→α 放入表项 M[A,b]中;
    };
for( 所有无定义的表项 M[A,a] )  标上错误标志。
```

⑤预测分析控制程序算法是利用课本上的算法：

```
do{
      令 X 是栈顶文法符号,a 是 ip 所指向的输入符号;
      if( X 是终结符号或$)
            if( X==a ) {从栈顶弹出 X;ip 前移一个位置;};
            else  error();
      else                                              /* X 是非终结符号 */
          if( M[X,a]=X→Y1Y2…Yk){
            从栈顶弹出 X;
            依次把 Yk,Yk-1,…,Y2,Y1 压入栈;              /* Y1 在栈顶 */
            输出产生式 X→Y1Y2…Yk;
          };
          else  error();
} while (X!=$)                                          /* 栈非空,分析继续 */
```

但是该算法有点小瑕疵，就是最后while循环的出口判断条件有问题，不应该是只是【栈顶文法符号X不为‘$’】，而应为【栈顶文法符号X不为‘$’&&所指向的输入符号a不为‘$’】。

输入形式：

先输入文法的产生式个数，然后输入产生式，该文法必须为LL1文法（消除左递归和左公因子），然后输入1或0，决定是否要进行分析，然后再输入要分析的字符串，该字符串需要用i代表数字，以‘$’结尾。

输入的文法默认第一个输入的产生式的左部为起始符，非终结符仅为一个大写字母，终结符仅为一个小写字母和各种符号，且用~代替 epsilon,且输入的文法必须满足 LL1 文法的条件。输入的要分析的字符串需要用 i 代表数字，以‘$’结尾。我认为没有必要把 i 也就是题目中的 num 细化为数字，因为这些操作已经在词法分析中完成了，语法分析只需完成分析类似于这种字符串的输入是否被接受就可以了。

输出形式：

先输出该文法的first集和follow集，再输出该文法的LL1预测分析表，最后输出用户输入的待分析字符串的分析过程，
详见【5、测试样例】所示：

3、变量和函数：

类：

```
class PF//Production formula,产生式的类
{
public:
    string left;//产生式的左部
    set<string> right;//产生式的右部
    PF(char s[])//构造函数，确定产生式左部
    {
        left = s;
```

```
    }
    void insert(char s[])//插入产生式右部的函数
    {
        right.insert(s);
    }
};
```

全局变量：

```
vector<PF> PF_vector;//产生式
```

使用一个 vector 数组，每个元素为 PF 类的对象，来存放输入的产生式。

```
map<string, set<char> > first;//first集
map<string, set<char> > follow;//follow集
```

使用了map哈希key的类型为string，存放产生式的左部，value的类型为char的set,用来存放对应key的first集和follow集。

```
vector<map<char, string>> predict_table;//LL1分析表
```

使用一个vector数组，每个元素为一个map，该vector的每一个map分别对应PF_vector中每个产生式的左部的非终结符所在的预测分析表的行，map的key为终结符，value为对应表项中的产生式的右部

```
vector<char> A;//分析栈
vector<char> B;//剩余串
map<string, int> PF_map;//存储每个非终结符对应的编号，key为非终结符，value为编号
vector<char> letter;//所有的终结符,在构造预测分析表的时候创建完毕
int B_point = 0, input_len = 0;//B_point 为输入串指针，input_len 为输入串长度
```

函数：

first 集的构造：

```
void first_construction() //构造并输出 first 集
void DFS(int x)//为构造 first 集深度优先遍历 PF，递归调用
```

follow 集的构造：

```
void follow_construction()//构造并输出 follow 集
void add_follow(const string& str1, const string& str2)//将 str1 的 follow 集加入到 str2 的 follow 集中
```

预测分析表的构造：

```
void predict_table_construction()//构造并输出 LL1 分析表
bool check_first(const string& str, char ch)//检查 ch 是否属于 str 的 FIRST 集合
bool check_follow(const string& str, char ch)//检查 ch 是否属于 str 的 FOLLOW 集合
```

预测分析控制程序：

```
void analyse()//预测分析控制程序
void print_A()//输出分析栈
void print_B()//输出剩余串
```

主函数：

```
int main()
```

PS：关于每个函数内部详细实现详见 main.cpp 中写好了详细的注释，我几乎对每一步操作的目的和函数的目的都写了详细的注释。

4、详细代码：

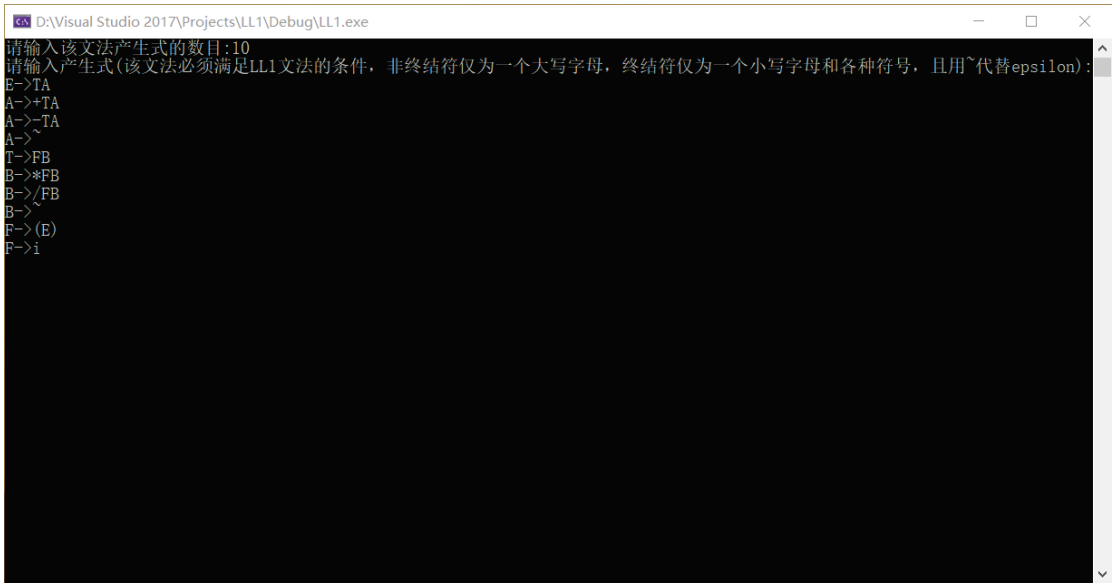
详见附件源.cpp，在此处不再赘述。

5、测试样例：

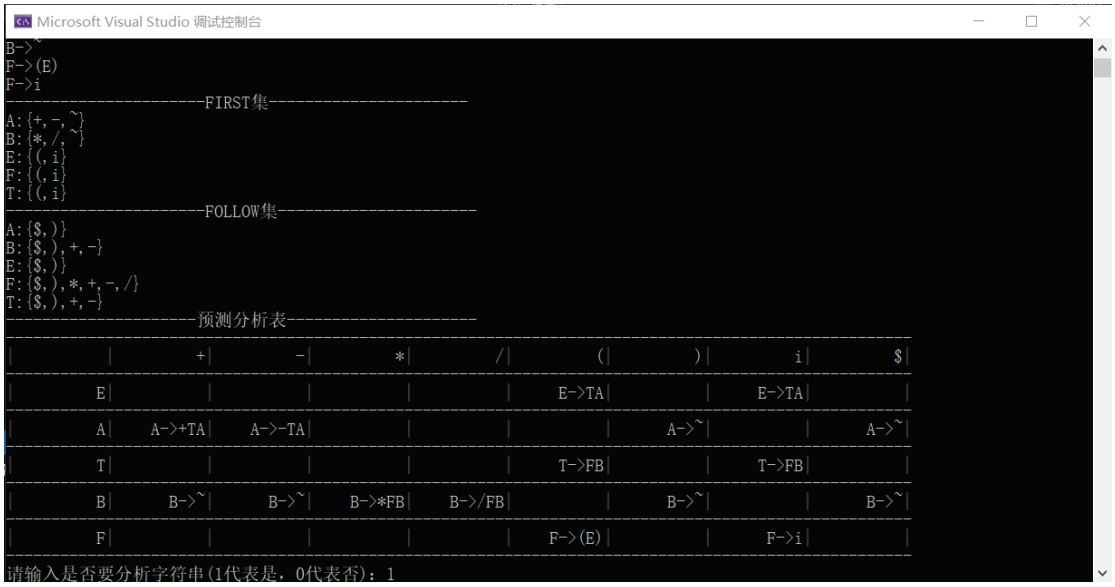
我写的这个语法分析程序普适性较高，不仅对课本上的算数运算文法能实现求first集和follow集，自动生成预测分析表，以及对字符串进行分析，还对几乎所有的LL1文法都可以实现，只要输入的文法按照【2.核心算法】中的【输入形式：】的要求即可。在这里我已课本上的算数运算文法为例，其余各种样例输入和输出截图详见附件，其它文法1为follow集循环依赖文法（求follow集不可以用递归求，否则该文法会无限循环），其它文法2为起始符可推空文法（如果不按严格算法4.2构造分析表，也就是作业中把A->~产生式加入到A的follow集所对应的终结符的表项中的做法其实是不准确的，而应把A可以推空的产生式加入到A的follow集所对应的终结符的表项中，否则就会导致无法识别某些字符串的错误，例如其它文法2就是无法识别空串的错误，但该文法确实可以识别空串）。

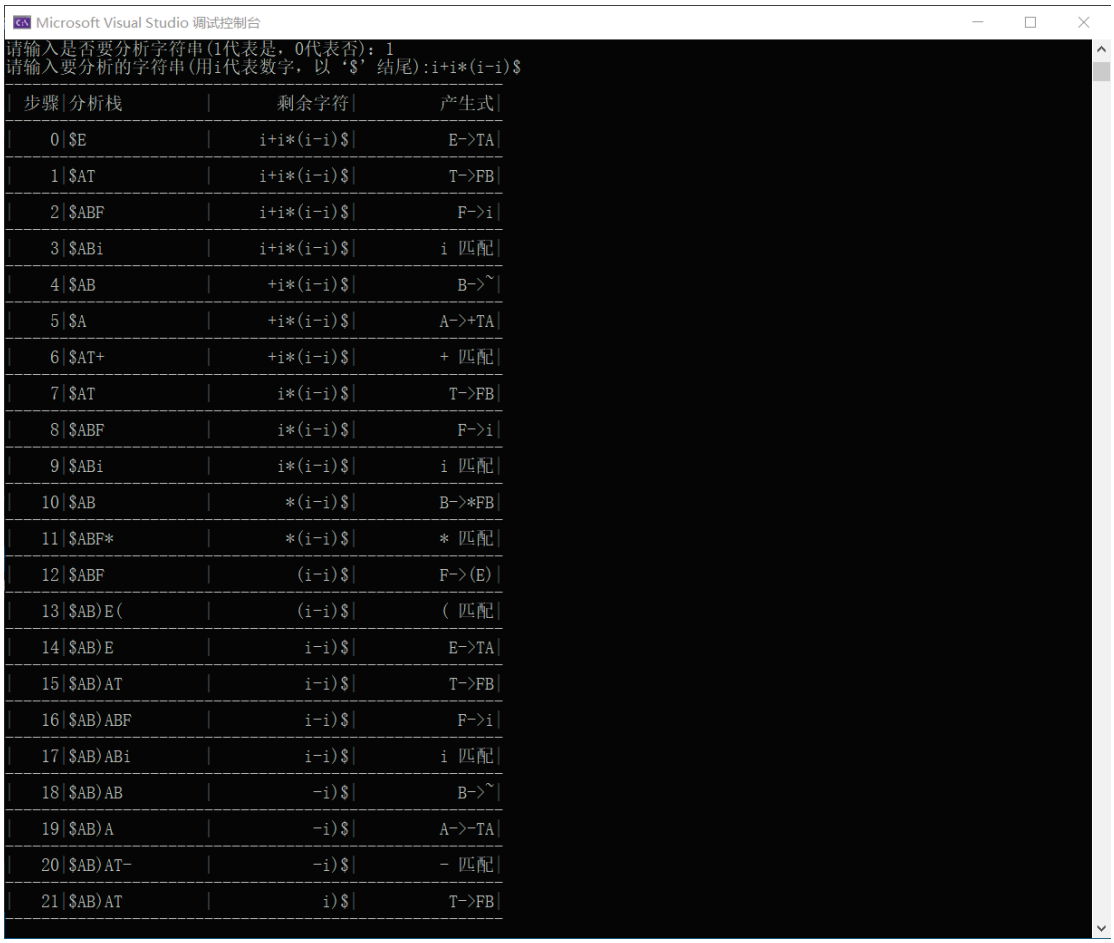
1、 输入

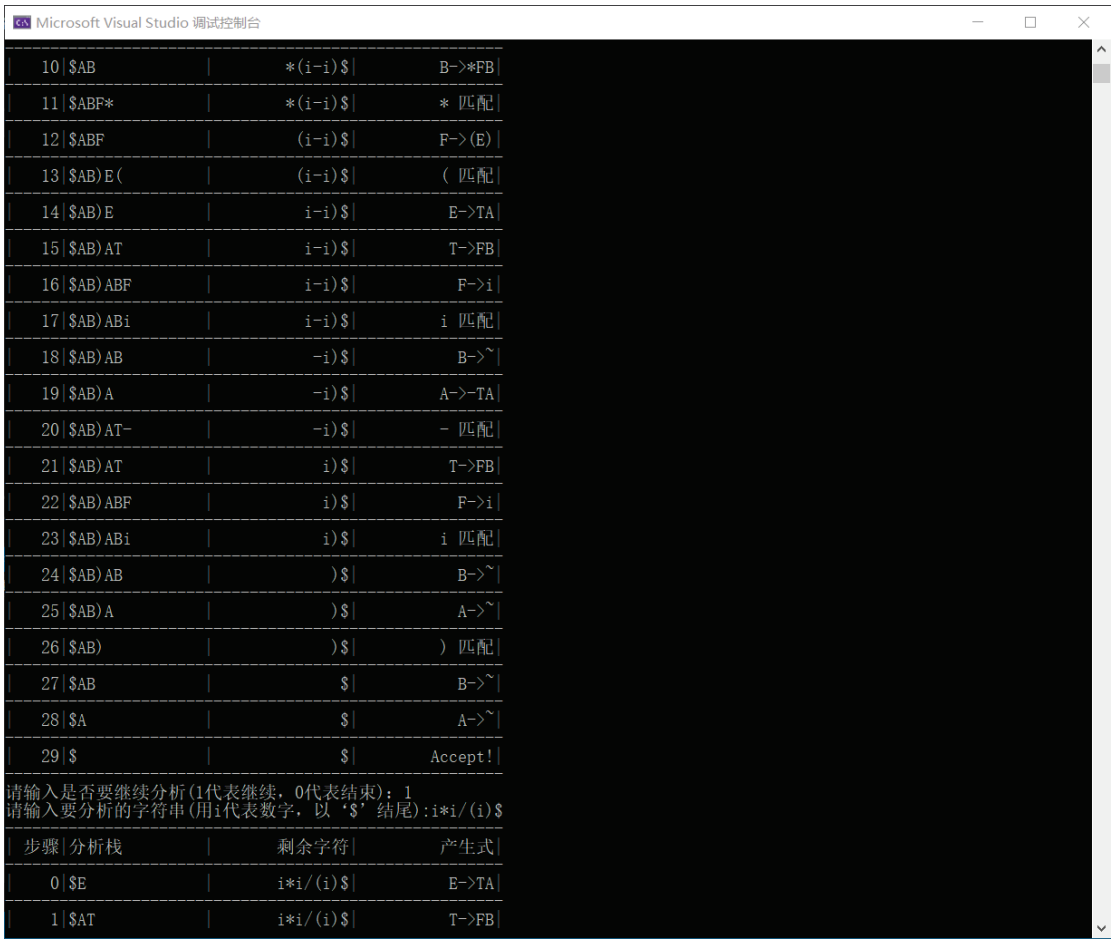
```
10
E->TA
A->+TA
A->-TA
A->~
T->FB
B->*FB
B->/FB
B->~
F->(E)
F->i
1
i+i*(i-i)$
1
i*i/(i)$
1
i/(i+i)-$
1
i**(i-i)$
1
*i-i$
0
```

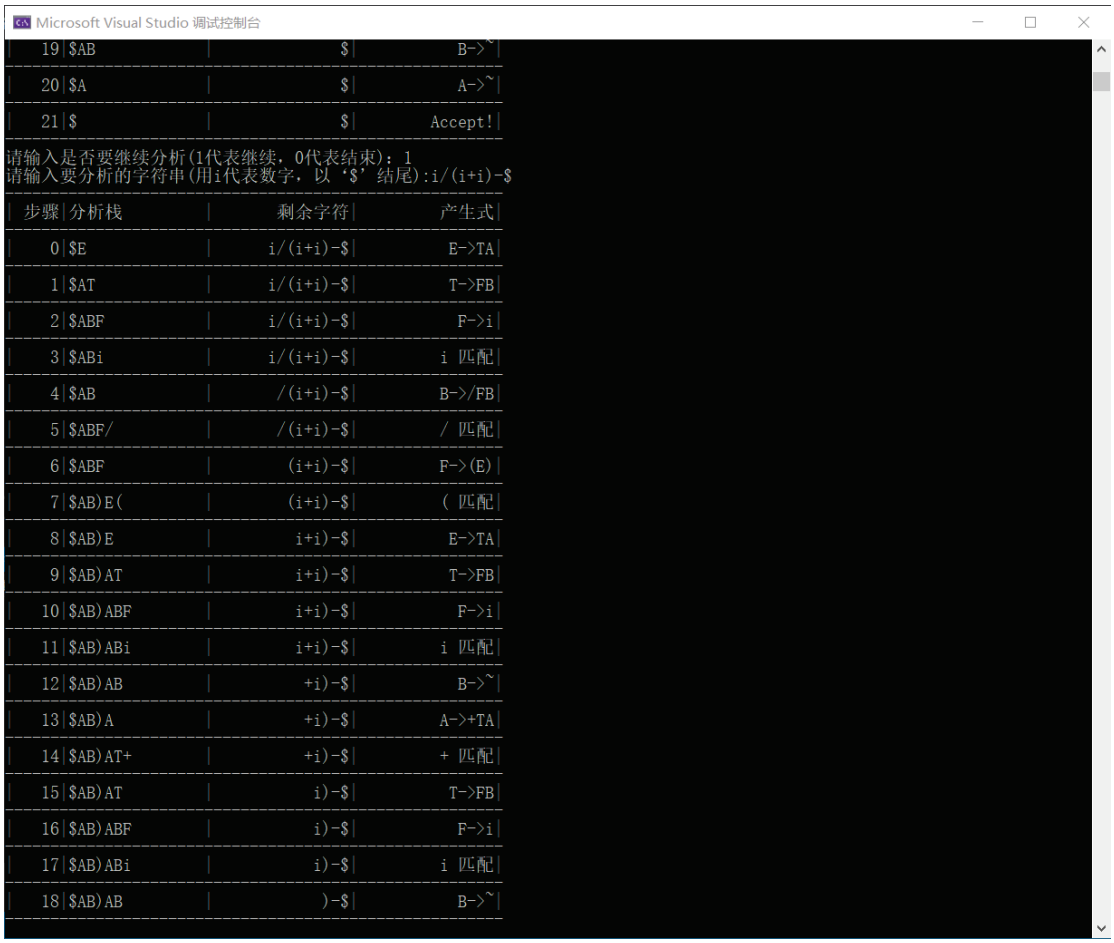

D:\Visual Studio 2017\Projects\LL1\Debug\LL1.exe
请输入该文法产生式的数目:10
请输入产生式(该文法必须满足LL1文法的条件，非终结符仅为一个大写字母，终结符仅为一个小写字母和各种符号，且用~代替epsilon):
E->TA
A->+TA
A->-TA
A->~
T->FB
B->*FB
B->/FB
B->~
F->(E)
F->i

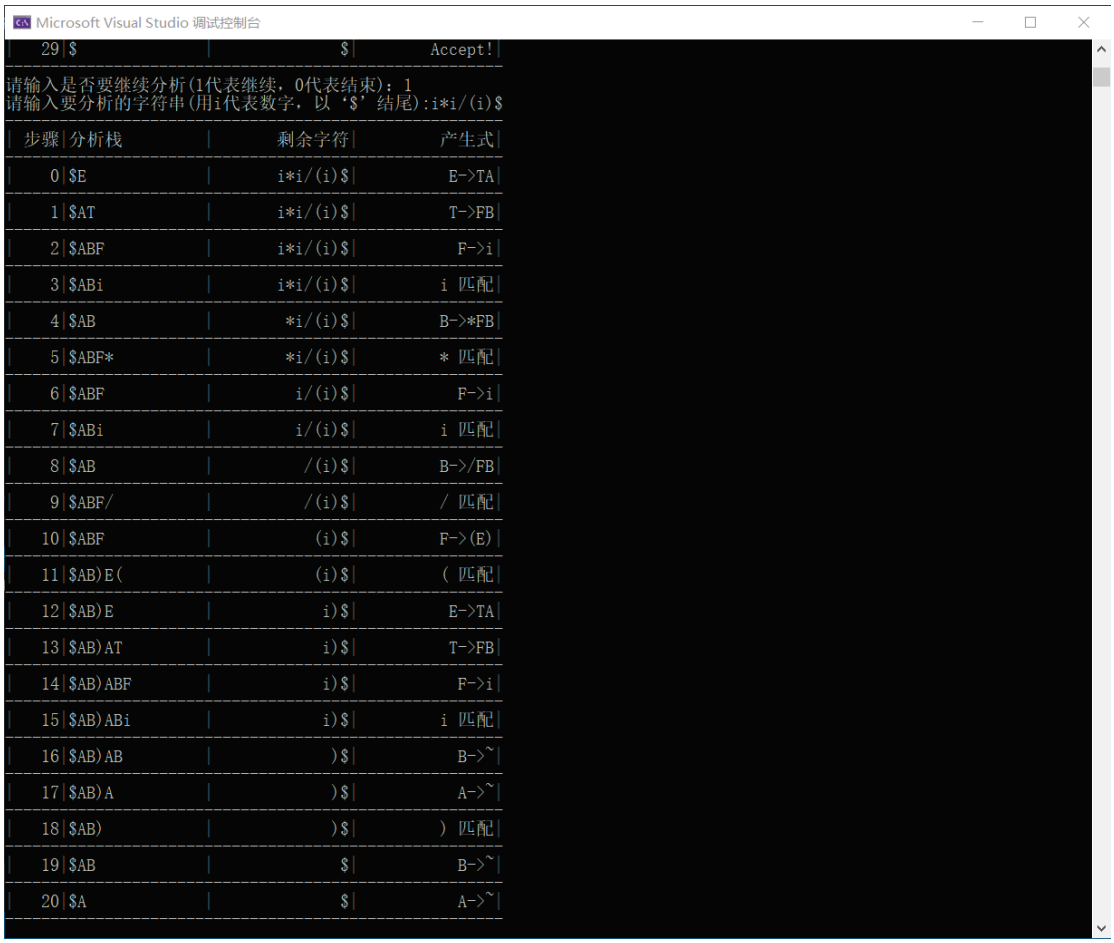
2、 输出：

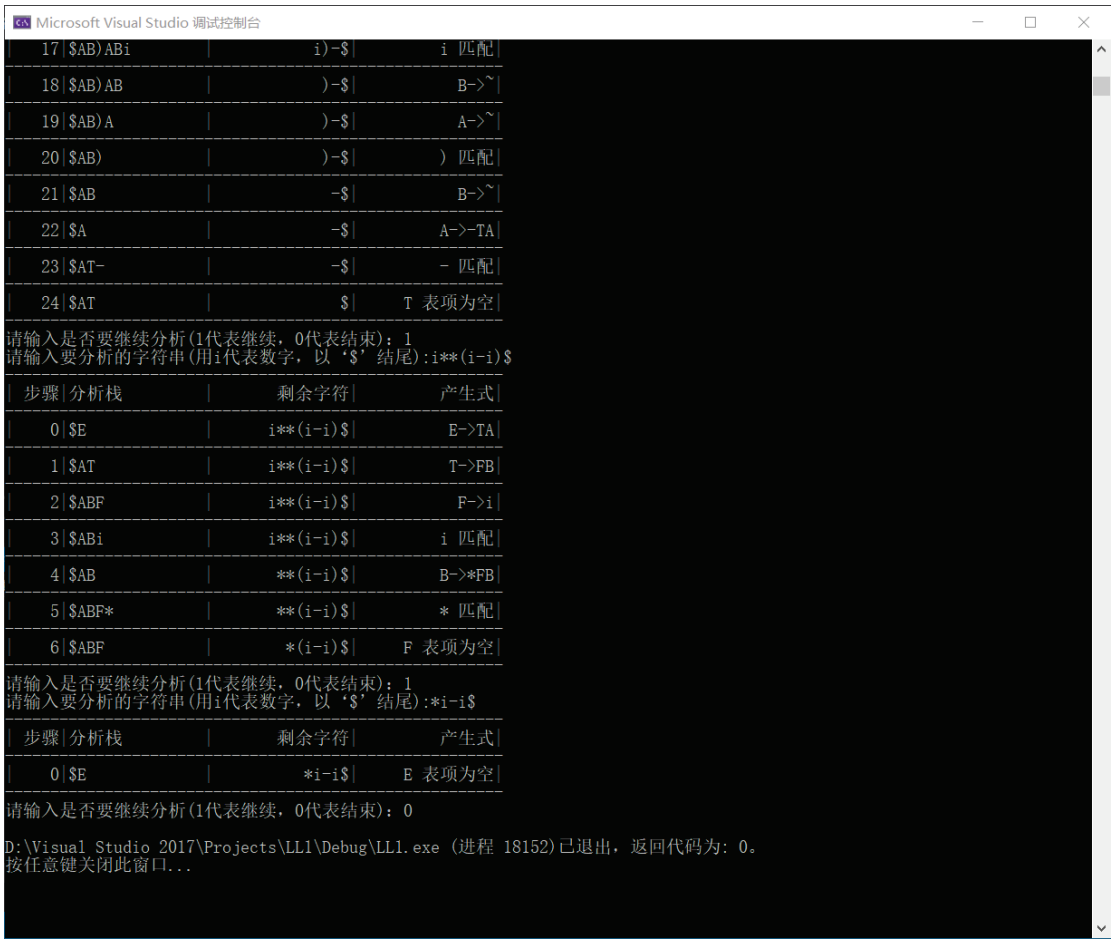

Microsoft Visual Studio 调试控制台
B->~
F->(E)
F->i
----------FIRST集----------
A:{+,-,~}
B:{*,/,~}
E:{(,i}
F:{(,i}
T:{(,i}
----------FOLLOW集----------
A:{$,)}
B:{$,),+,-}
E:{$,)}
F:{$,),*,+,-,/}
T:{$,),+,-}
----------预测分析表----------
	+	-	*	/	(	)	i	$
E					E->TA		E->TA	
A	A->+TA	A->-TA				A->~		A->~
T					T->FB		T->FB	
B	B->~	B->~	B->*FB	B->/FB		B->~		B->~
F					F->(E)		F->i	
请输入是否要分析字符串(1代表是，0代表否)：1


Microsoft Visual Studio 调试控制台
请输入是否要分析字符串(1代表是，0代表否)：1
请输入要分析的字符串(用i代表数字，以‘$’结尾):i+i*(i-i)$
步骤	分析栈	剩余字符	产生式
0	$E	i+i*(i-i)$	E->TA
1	$AT	i+i*(i-i)$	T->FB
2	$ABF	i+i*(i-i)$	F->i
3	$ABi	i+i*(i-i)$	i 匹配
4	$AB	+i*(i-i)$	B->^
5	$A	+i*(i-i)$	A->+TA
6	$AT+	+i*(i-i)$	+ 匹配
7	$AT	i*(i-i)$	T->FB
8	$ABF	i*(i-i)$	F->i
9	$ABi	i*(i-i)$	i 匹配
10	$AB	*(i-i)$	B->*FB
11	$ABF*	*(i-i)$	* 匹配
12	$ABF	(i-i)$	F->(E)
13	$AB)E(	(i-i)$	(匹配
14	$AB)E	i-i)$	E->TA
15	$AB)AT	i-i)$	T->FB
16	$AB)ABF	i-i)$	F->i
17	$AB)ABi	i-i)$	i 匹配
18	$AB)AB	-i)$	B->^
19	$AB)A	-i)$	A->-TA
20	$AB)AT-	-i)$	- 匹配
21	$AB)AT	i)$	T->FB


Microsoft Visual Studio 调试控制台
10	$AB	*(i-i)$	B->*FB
11	$ABF*	*(i-i)$	* 匹配
12	$ABF	(i-i)$	F->(E)
13	$AB)E(	(i-i)$	(匹配
14	$AB)E	i-i)$	E->TA
15	$AB)AT	i-i)$	T->FB
16	$AB)ABF	i-i)$	F->i
17	$AB)ABi	i-i)$	i 匹配
18	$AB)AB	-i)$	B->ˆ
19	$AB)A	-i)$	A->-TA
20	$AB)AT-	-i)$	- 匹配
21	$AB)AT	i)$	T->FB
22	$AB)ABF	i)$	F->i
23	$AB)ABi	i)$	i 匹配
24	$AB)AB	)$	B->ˆ
25	$AB)A	)$	A->ˆ
26	$AB)	)$	) 匹配
27	$AB	$	B->ˆ
28	$A	$	A->ˆ
29	$	$	Accept!
请输入是否要继续分析(1代表继续，0代表结束)：1
请输入要分析的字符串(用i代表数字，以‘$’结尾):i*i/(i)$
步骤	分析栈	剩余字符	产生式
0	$E	i*i/(i)$	E->TA
1	$AT	i*i/(i)$	T->FB


Microsoft Visual Studio 调试控制台
19	$AB	$	B->`
20	$A	$	A->`
21	$	$	Accept!
请输入是否要继续分析(1代表继续，0代表结束)：1
请输入要分析的字符串(用i代表数字，以‘$’结尾):i/(i+i)-$
步骤	分析栈	剩余字符	产生式
0	$E	i/(i+i)-$	E->TA
1	$AT	i/(i+i)-$	T->FB
2	$ABF	i/(i+i)-$	F->i
3	$ABi	i/(i+i)-$	i 匹配
4	$AB	/(i+i)-$	B->/FB
5	$ABF/	/(i+i)-$	/ 匹配
6	$ABF	(i+i)-$	F->(E)
7	$AB)E(	(i+i)-$	(匹配
8	$AB)E	i+i)-$	E->TA
9	$AB)AT	i+i)-$	T->FB
10	$AB)ABF	i+i)-$	F->i
11	$AB)ABi	i+i)-$	i 匹配
12	$AB)AB	+i)-$	B->`
13	$AB)A	+i)-$	A->+TA
14	$AB)AT+	+i)-$	+ 匹配
15	$AB)AT	i)-$	T->FB
16	$AB)ABF	i)-$	F->i
17	$AB)ABi	i)-$	i 匹配
18	$AB)AB	)-$	B->`


Microsoft Visual Studio 调试控制台
| 29|$ | $| Accept!|
请输入是否要继续分析(1代表继续，0代表结束)：1
请输入要分析的字符串(用i代表数字，以‘$’结尾):i*i/(i)$
步骤	分析栈	剩余字符	产生式
0	$E	i*i/(i)$	E->TA
1	$AT	i*i/(i)$	T->FB
2	$ABF	i*i/(i)$	F->i
3	$ABi	i*i/(i)$	i 匹配
4	$AB	*i/(i)$	B->*FB
5	$ABF*	*i/(i)$	* 匹配
6	$ABF	i/(i)$	F->i
7	$ABi	i/(i)$	i 匹配
8	$AB	/(i)$	B->/FB
9	$ABF/	/(i)$	/ 匹配
10	$ABF	(i)$	F->(E)
11	$AB)E(	(i)$	(匹配
12	$AB)E	i)$	E->TA
13	$AB)AT	i)$	T->FB
14	$AB)ABF	i)$	F->i
15	$AB)ABi	i)$	i 匹配
16	$AB)AB	)$	B->~
17	$AB)A	)$	A->~
18	$AB)	)$	) 匹配
19	$AB	$	B->~
20	$A	$	A->~



6、实验总结

LL1 语法分析器实验让我更深入的了解了求 first 集，求 follow 集，构造预测分析表和分析字符串的各种算法，也让我发现了我曾经忽略的部分细节，如样例中其它文法 2 构造分析表时的错误，还提高了我的编程能力，总之让我受益匪浅。