

河北大学

《编译原理实验三》

实验报告

学 院 网络空间安全与计算机学院

专 业 计算机科学与技术

学 号 20201202075

姓 名 myp

指导教师 刘海博

2023年 5月 12 日

目录

[1 概述 2](#_Toc134893212)

[1.1目的与意义 2](#_Toc134893213)

[1.2主要完成的任务 2](#_Toc134893214)

[1.3使用的开发工具 2](#_Toc134893215)

[1.4解决的主要问题 2](#_Toc134893216)

[2.使用的基本概念和原理 2](#_Toc134893217)

[2.1 FIRST集 2](#_Toc134893218)

[2.2 FOLLOW集 2](#_Toc134893219)

[2.3 SELECT集 3](#_Toc134893220)

[2.4 LL(1)文法 3](#_Toc134893221)

[2.5 表驱动LL(1)分析程序 3](#_Toc134893222)

[3 总体设计 4](#_Toc134893223)

[3.1 技术路线 4](#_Toc134893224)

[3.2总体结构 4](#_Toc134893225)

[3.3模块划分 4](#_Toc134893226)

[4 详细设计 5](#_Toc134893227)

[4.1 函数及参数设计 5](#_Toc134893228)

[5 编码实现 7](#_Toc134893229)

[5.1源代码及注释 7](#_Toc134893230)

[5.2开发环境 27](#_Toc134893231)

[5.3解决的难点 27](#_Toc134893232)

[6 测试和试运行 27](#_Toc134893233)

[6.1测试方法 27](#_Toc134893234)

[6.2测试用例 27](#_Toc134893235)

[7. 总结 28](#_Toc134893236)

# 1 概述

## 1.1目的与意义

设计、编制、调试一个典型的语法分析程序，实现对词法分析程序所提供的单词序列进行语法检查和结构分析，进一步掌握常用的语法分析方法

## 1.2主要完成的任务

(1)根据LL(1)分析法编写一个语法分析程序，输入文法，由程序自动生成文法的预测分析表。

(2)所开发的程序可适用于不同的文法和任意输入串，且能判断该文法是否为LL(1)文法。

(3) 有运行实例。对于输入的一个文法和一个单词串，所编制的语法分析程序应能正确地判断，此单词串是否为该文法的句子，并要求输出分析过程。

## 1.3使用的开发工具

Visual Studio Code

## 1.4解决的主要问题

（1）对输入的文法的相关信息进行处理，并根据输入的文法来求得每个文法的FIRST集，FOLLOW集和SELECT集。

（2）对具有相同产生式左部的SELECT集求交集，判断是否为LL(1)文法，若不是程序结束。

（3）构造该文法的预测分析表并输出。

（4）输入符号串，判定该符号串是否为该文法的句子。

（5）对过程中出现的一些错误能够及时报错。

# 2.使用的基本概念和原理

## 2.1 FIRST集

定义：设G=(VT,VN,P,S)是一个上下文无关文法。



若，则规定。

## 2.2 FOLLOW集

定义：设G=(VT,VN,P,S)是一个上下文无关文法，，S是开始符号



若且，则规定。

## 2.3 SELECT集

定义：一个产生式的选择符号集SELECT。给定上下文无关文法的产生式，，，若，则。如果，则。

## 2.4 LL(1)文法

含义：第一个L表明自顶向下分析是从左到右扫描输入串。第二个L表明分析过程中将用最左推导，1表明只需向右查看一个字符便可决定如何推导，即选择哪个产生式进行推导。

判定：一个上下文无关文法是LL(1)文法的充分必要条件是，对每个非终结符A的两个不同产生式，，满足，其中α，β不同时能

## 2.5 表驱动LL(1)分析程序

概述：一个表驱动的LL(1)分析程序由预测分析程序、先进后出栈和预测分析表3个部分组成，其中只有预测分析表与文法有关，而分析表又可用一个矩阵M表示。矩阵的元素M[A,a]中的下标A表示非终结符，a表示终结符或句子括号#，矩阵元素M[A,a]中的内容是一条关于A的产生式，表明当用非终结符A向下推导时，面临输入符a时所采取的候选产生式。当元素内容无产生式时，则表明用A为左部向下推导时遇到了不该出现的符号，因此内容为转为出错处理的信息。

构造预测分析表：

对每个非终结符或‘#’号用a表示。

若

把所有无意义的M[A,a]标上出错标记。

为了使表简化，其产生式的左部可以不写入表中，表中空白处为出错。

# 3 总体设计

## 3.1 技术路线

本程序采用的编程语言是面向过程的编程语言（c），通过设计程序对输入的内容进行处理，得到最终结果并输出。

## 3.2总体结构

下图给出了程序的总体结构以及各模块间的消息。程序按照下图的顺序运行，同时一些模块间存在调用关系，在下图也有体现。具体内容请看下面的模块划分部分。

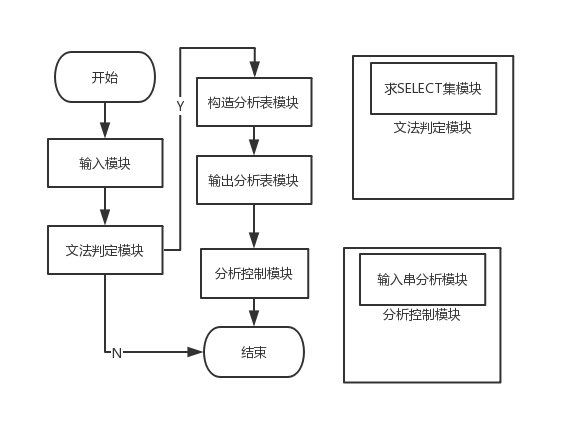


图3.1 系统层次图

## 3.3模块划分

3.3.1 输入模块

功能描述：输入文法的产生式个数、内容、终结符、非终结符、产生式、开始符等，并将其存入系统，以便其他模块调用。

3.3.2 求SELECT集模块

功能描述：根据文法产生式，求出FIRST集、FOLLOW集，推导出各个产生式的SELECT集。

3.3.3判定LL(1)文法模块

功能描述：将含有相同左部的产生式找出来，判断它们的SELECT集有没有交集。若都没有交集，说明文法为LL（1）文法，否则则不是，程序结束。

3.3.4 构造预测分析表模块

功能描述：对分析表进行初始化，将‘#’存入终结符集合，然后分析表用行表示对应的非终结符，用列表示对应的终结符，将对应的产生式是第几条放入分析表。

3.3.5 输出预测分析表模块

功能描述：首先输出第一行（终结符和‘#’）,然后依次输出非终结符及其对应的产生式，无产生式的地方输出error。

3.3.6 控制分析模块

功能描述：可用于人机交互，首先调用输入串分析模块，处理一次符号串，然后选择是否继续输入符号串，若否，程序结束；若是，则继续调用输入串分析模块。

3.3.7 输入串分析模块

功能描述：首先输入输入串，然后一边处理输入串，一边输出处理过程。若出错，则报错程序结束。否则，程序继续运行，若输入串有不匹配，说明该输入串不是该文法的句子，程序结束。若全部匹配，则说明该输入串是该文法的句子，程序结束。

# 4 详细设计

## 4.1 函数及参数设计

（1）全局变量设计

int count = 0; /\*分解的产生式的个数\*/

int number; /\*所有终结符和非终结符的总数\*/

char start; /\*开始符号\*/

char termin[50]; /\*终结符号\*/

char non\_ter[50]; /\*非终结符号\*/

char allSymbols[50]; /\*所有符号\*/

char left[50]; /\*左部\*/

char right[50][50]; /\*右部\*/

char first[50][50], follow[50][50]; /\*各产生式右部的FIRST和左部的FOLLOW集合\*/

char firstSetSingle[50][50]; /\*所有单个符号的FIRST集合\*/

char select[50][50]; /\*各单个产生式的SELECT集合\*/

char firstFlags[50], followFlags[50]; /\*记录各符号的FIRST和FOLLOW是否已求过\*/

char empty[20]; /\*记录可直接推出^的符号\*/

char tempFirstSet[50]; /\*求FOLLOW时存放某一符号串的FIRST集合\*/

int validity = 1; /\*表示输入文法是否有效\*/

int isLL1 = 1; /\*表示输入文法是否为LL(1)文法\*/

int parseTable[20][20]; /\*分析表\*/

char choose; /\*用户输入时使用\*/

char emptyTemp[20]; /\*求\_emp()时使用\*/

char followTemp[20]; /\*求FOLLOW集合时使用\*/

（2）函数设计

char c() //得到一个不是非终结符的符号

void non\_re(char \*point) // 分解不含有左递归的产生式

void recur(char \*point) // 分解含有左递归的产生式

char readGrammar (char \*t, char \*n, char \*left, char right[50][50])// 读入一个文法

void merge(char \*d, char \*s, int type)// 将单个符号或符号串并入另一符号串

void emp(char c)// 求所有能直接推出^的符号

int \_emp(char c)// 求某一符号能否推出‘ ^ ’

void computeSingleFirst (int i) //求单个符号的FIRST

void FIRST(int i, char \*p) //求各产生式右部的FIRST

void FOLLOW(int i)// 求各产生式左部的FOLLOW

int isLL1Grammar () //判断读入文法是否为一个LL(1)文法

void constructParsingTable ()构造分析表M

int judge(); // 判断读入的文法是否正确

int in(char c,char \*p); //判断一个字符是否在指定字符串中

void menu(); //一个用户调用函数

void syntax(); // 总控算法

int main(); //主函数

# 5 编码实现

## 5.1源代码及注释

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int count = 0; /\*分解的产生式的个数\*/

int number; /\*所有终结符和非终结符的总数\*/

char start; /\*开始符号\*/

char termin[50]; /\*终结符号\*/

char non\_ter[50]; /\*非终结符号\*/

char allSymbols[50]; /\*所有符号\*/

char left[50]; /\*左部\*/

char right[50][50]; /\*右部\*/

char first[50][50], follow[50][50]; /\*各产生式右部的FIRST和左部的FOLLOW集合\*/

char firstSetSingle[50][50]; /\*所有单个符号的FIRST集合\*/

char select[50][50]; /\*各单个产生式的SELECT集合\*/

char firstFlags[50], followFlags[50]; /\*记录各符号的FIRST和FOLLOW是否已求过\*/

char empty[20]; /\*记录可直接推出^的符号\*/

char tempFirstSet[50]; /\*求FOLLOW时存放某一符号串的FIRST集合\*/

int validity = 1; /\*表示输入文法是否有效\*/

int isLL1 = 1; /\*表示输入文法是否为LL(1)文法\*/

int parseTable[20][20]; /\*分析表\*/

char choose; /\*用户输入时使用\*/

/\*\*emptyTemp数组是用于辅助计算FOLLOW集合的临时数组。在FOLLOW函数中，它存储了待求的非终结符的FOLLOW集合的临时结果。具体来说，emptyTemp数组在以下几个地方起到了作用：

在每次递归调用FOLLOW函数之前，将followTemp数组的内容复制到emptyTemp数组中，作为临时存储。

在计算某个非终结符的FOLLOW集合时，将结果存储在followTemp数组中。

在计算某个产生式右部的FIRST集合时，需要将emptyTemp数组清空，以便存储新的FIRST集合结果。

通过使用emptyTemp数组，可以在计算FOLLOW集合的过程中临时存储结果，避免数据混乱或重复计算的问题。

\*\*/

char emptyTemp[20]; /\*求\_emp()时使用\*/

char followTemp[20]; /\*求FOLLOW集合时使用\*/

/\*\*判断一个字符是否在指定字符串中\*\*/

int in(char c, char \*p)

{

int i;

if (strlen(p) == 0)

return (0);

for (i = 0;; i++)

{

if (p[i] == c)

return (1); /\*若在，返回1\*/

if (i == strlen(p))

return (0); /\*若不在，返回0\*/

}

}

/\*\*得到一个不是非终结符的符号\*\*/

char c()

{

char c = 'A';

while (in(c, non\_ter) == 1)

c++;

return (c);

}

/\*\*分解含有左递归的产生式\*\*/

void recur(char \*point)

{ /\*完整的产生式在point[]中\*/

int j, m = 0, n = 3, k;

char temp[20], ch;

ch = c(); /\*得到一个非终结符\*/

k = strlen(non\_ter);

non\_ter[k] = ch;

non\_ter[k + 1] = '\0';

for (j = 0; j <= strlen(point) - 1; j++)

{

if (point[n] == point[0])

{ /\*如果‘|’后的首符号和左部相同\*/

for (j = n + 1; j <= strlen(point) - 1; j++)

{

while (point[j] != '|' && point[j] != '\0')

temp[m++] = point[j++];

left[count] = ch;

memcpy(right[count], temp, m);

right[count][m] = ch;

right[count][m + 1] = '\0';

m = 0;

count++;

if (point[j] == '|')

{

n = j + 1;

break;

}

}

}

else

{ /\*如果‘|’后的首符号和左部不同\*/

left[count] = ch;

right[count][0] = '^';

right[count][1] = '\0';

count++;

for (j = n; j <= strlen(point) - 1; j++)

{

if (point[j] != '|')

temp[m++] = point[j];

else

{

left[count] = point[0];

memcpy(right[count], temp, m);

right[count][m] = ch;

right[count][m + 1] = '\0';

printf(" count=%d ", count);

m = 0;

count++;

}

}

left[count] = point[0];

memcpy(right[count], temp, m);

right[count][m] = ch;

right[count][m + 1] = '\0';

count++;

m = 0;

}

}

}

/\*\*分解不含有左递归的产生式\*\*/

void non\_re(char \*point)

{

int m = 0, j;

char temp[20];

for (j = 3; j <= strlen(point) - 1; j++)

{

if (point[j] != '|')

temp[m++] = point[j];

else

{

left[count] = point[0];

memcpy(right[count], temp, m);

right[count][m] = '\0';

m = 0;

count++;

}

}

left[count] = point[0];

memcpy(right[count], temp, m);

right[count][m] = '\0';

count++;

m = 0;

}

/\*\*读入一个文法\*\*/

char readGrammar(char \*t, char \*n, char \*left, char right[50][50])

{

char vn[50], vt[50];

char s;

char p[50][50];

int i, j, k;

printf("\n请输入文法的非终结符号串：");

scanf("%s", vn);

getchar();

i = strlen(vn);

memcpy(n, vn, i);

n[i] = '\0';

printf("请输入文法的终结符号串：");

scanf("%s", vt);

getchar();

i = strlen(vt);

memcpy(t, vt, i);

t[i] = '\0';

printf("请输入文法的开始符号：");

scanf("%c", &s);

getchar();

printf("请输入文法产生式的条数：");

scanf("%d", &i);

getchar();

for (j = 1; j <= i; j++)

{

printf("请输入文法的第%d条（共%d条）产生式：", j, i);

scanf("%s", p[j - 1]);

getchar();

}

for (j = 0; j <= i - 1; j++)

if (p[j][1] != '-' || p[j][2] != '>')

{

printf("\ninput error!");

validity = 0;

return ('\0');

} /\*检测输入错误\*/

for (k = 0; k <= i - 1; k++)

{ /\*分解输入的各产生式\*/

if (p[k][3] == p[k][0])

recur(p[k]);

else

non\_re(p[k]);

}

return (s);

}

/\*\*将单个符号或符号串并入另一符号串\*\*/

void merge(char \*d, char \*s, int type)

{ /\*d是目标符号串，s是源串，type＝1，源串中的‘ ^ ’一并并入目串；type＝2，源串中的‘ ^ ’不并入目串\*/

int i, j;

for (i = 0; i <= strlen(s) - 1; i++)

{

if (type == 2 && s[i] == '^')

;

else

{

for (j = 0;; j++)

{

if (j < strlen(d) && s[i] == d[j])

break;

if (j == strlen(d))

{

d[j] = s[i];

d[j + 1] = '\0';

break;

}

}

}

}

}

/\*\*求所有能直接推出^的符号\*\*/

void emp(char c)

{ /\*即求所有由‘ ^ ’推出的符号\*/

char temp[10];

int i;

for (i = 0; i <= count - 1; i++)

{

if (right[i][0] == c && strlen(right[i]) == 1)

{

temp[0] = left[i];

temp[1] = '\0';

merge(empty, temp, 1);

emp(left[i]);

}

}

}

/\*\*求某一符号能否推出‘ ^ ’\*\*/

/\*\*函数接收一个字符作为参数，用于在文法的产生式中查找。

将输入的字符添加到emptyTemp数组中，用于跟踪已经处理过的符号。

如果输入的字符已经在empty数组中（表示可以推导出空字符串），直接返回1。

遍历文法的所有产生式，找到左部等于输入字符的产生式。

如果找到符合条件的产生式，计算其右部的长度。

针对右部的长度和右部符号的类型（终结符、非终结符或空），进行不同的处理。

a. 如果右部只包含一个符号，且该符号可以推导出空字符串，返回1。

b. 如果右部只包含一个终结符，返回0。

c. 对于其他情况：

遍历右部的所有符号，检查这些符号是否在emptyTemp数组中。如果找到一个在emptyTemp中的符号，设置mark为1，然后继续检查下一个产生式。

如果没有找到在emptyTemp中的符号，对右部的每个符号递归调用\_emp()函数，将每个符号的结果相乘得到整个右部的结果。

如果整个右部的结果为0，表示当前产生式不能推导出空字符串，继续检查下一个产生式；如果结果为1，表示当前产生式可以推导出空字符串，返回1。

如果遍历完所有产生式都没有找到可以推导出空字符串的情况，返回0。\*/

int \_emp(char c)

{ /\*若能推出，返回1；否则，返回0\*/

int i, j, k, result = 1, mark = 0;

char temp[20];

temp[0] = c;

temp[1] = '\0';

merge(emptyTemp, temp, 1);

if (in(c, empty) == 1)

return (1);

for (i = 0;; i++)

{

if (i == count)

return (0);

if (left[i] == c) /\*找一个左部为c的产生式\*/

{

j = strlen(right[i]); /\*j为右部的长度\*/

if (j == 1 && in(right[i][0], empty) == 1)

return (1);

else if (j == 1 && in(right[i][0], termin) == 1)

return (0);

else

{

for (k = 0; k <= j - 1; k++)

if (in(right[i][k], emptyTemp) == 1)

mark = 1;

if (mark == 1)

continue;

else

{

for (k = 0; k <= j - 1; k++)

{

result \*= \_emp(right[i][k]);

temp[0] = right[i][k];

temp[1] = '\0';

merge(emptyTemp, temp, 1);

}

}

}

if (result == 0 && i < count)

continue;

else if (result == 1 && i < count)

return (1);

}

}

}

/\*\*判断读入的文法是否正确\*\*/

int judge()

{

int i, j;

for (i = 0; i <= count - 1; i++)

{

if (in(left[i], non\_ter) == 0)

{ /\*若左部不在非终结符中，报错\*/

printf("\nerror1!");

validity = 0;

return (0);

}

for (j = 0; j <= strlen(right[i]) - 1; j++)

{

if (in(right[i][j], non\_ter) == 0 && in(right[i][j], termin) == 0 && right[i][j] != '^')

{ /\*若右部某一符号不在非终结符、终结符中且不为‘ ^ ’，报错\*/

printf("\nerror2!");

validity = 0;

return (0);

}

}

}

return (1);

}

/\*\*求单个符号的FIRST\*\*/

/\*\*函数接收一个参数i，表示符号在所有输入符号中的序号。

从allSymbols数组中获取这个符号。

调用emp函数来计算空集。

检查这个符号是否是终结符：

a. 如果是，将其添加到相应的firstSetSingle数组中。

如果这个符号是非终结符：

a. 遍历所有的产生式，找到左部等于这个非终结符的产生式。

b. 如果产生式的右部的第一个符号是终结符或者空，将它添加到firstSetSingle数组中。

c. 如果产生式的右部的第一个符号是非终结符，且这个非终结符不等于当前的符号（避免自我循环）：

找到这个非终结符在allSymbols数组中的序号。

如果这个非终结符的FIRST集还没计算过，递归调用computeSingleFirst函数来计算它，然后将结果合并到当前符号的firstSetSingle数组中。

遍历产生式右部的所有符号，如果一个符号的\_emp值为1（表示它可以推导出空），且这个符号不是右部的最后一个符号，计算下一个符号的FIRST集并将结果合并到当前符号的firstSetSingle数组中。

如果一个符号的\_emp值为1，且这个符号是右部的最后一个符号，将空添加到当前符号的firstSetSingle数组中。

最后，设置firstFlags数组中相应的值为'1'，表示这个符号的FIRST集已经计算过。

\*/

void computeSingleFirst(int i)

{ /\*i为符号在所有输入符号中的序号\*/

char c, temp[20];

int j, k, m;

c = allSymbols[i];

char ch = '^';

emp(ch);

if (in(c, termin) == 1) /\*若为终结符\*/

{

firstSetSingle[i][0] = c;

firstSetSingle[i][1] = '\0';

}

else if (in(c, non\_ter) == 1) /\*若为非终结符\*/

{

for (j = 0; j <= count - 1; j++)

{

if (left[j] == c)

{

if (in(right[j][0], termin) == 1 || right[j][0] == '^')

{

temp[0] = right[j][0];

temp[1] = '\0';

merge(firstSetSingle[i], temp, 1);

}

else if (in(right[j][0], non\_ter) == 1)

{

if (right[j][0] == c)

continue;

for (k = 0;; k++)

if (allSymbols[k] == right[j][0])

break;

if (firstFlags[k] == '0')

{

computeSingleFirst(k);

firstFlags[k] = '1';

}

merge(firstSetSingle[i], firstSetSingle[k], 2);

for (k = 0; k <= strlen(right[j]) - 1; k++)

{

emptyTemp[0] = '\0';

if (\_emp(right[j][k]) == 1 && k < strlen(right[j]) - 1)

{

for (m = 0;; m++)

if (allSymbols[m] == right[j][k + 1])

break;

if (firstFlags[m] == '0')

{

computeSingleFirst(m);

firstFlags[m] = '1';

}

merge(firstSetSingle[i], firstSetSingle[m], 2);

}

else if (\_emp(right[j][k]) == 1 && k == strlen(right[j]) - 1)

{

temp[0] = '^';

temp[1] = '\0';

merge(firstSetSingle[i], temp, 1);

}

else

break;

}

}

}

}

}

firstFlags[i] = '1';

}

/\*\*求各产生式右部的FIRST\*/

/\*\*函数接收两个参数：i表示要计算的产生式的索引，p为要计算FIRST集的字符串。

计算字符串的长度。

如果字符串长度为1：

a. 如果字符串中的唯一字符是空（'^'），将空添加到first数组（如果i >= 0）或tempFirstSet数组（如果i < 0）中。

b. 否则，查找这个字符在allSymbols数组中的位置，将对应的firstSetSingle数组拷贝到first数组（如果i >= 0）或tempFirstSet数组（如果i < 0）中。

如果字符串长度大于1：

a. 查找第一个字符在allSymbols数组中的位置，将对应的firstSetSingle数组合并到first数组（如果i >= 0）或tempFirstSet数组（如果i < 0）中。

b. 遍历字符串中的每个字符，如果字符的\_emp值为1（表示它可以推导出空），并且这个字符不是字符串的最后一个字符，将下一个字符的FIRST集合并到first数组（如果i >= 0）或tempFirstSet数组（如果i < 0）中。

c. 如果字符的\_emp值为1，并且这个字符是字符串的最后一个字符，将空添加到first数组（如果i >= 0）或tempFirstSet数组（如果i < 0）中。

d. 如果字符的\_emp值为0（表示它不能推导出空），则停止遍历。\*/

void FIRST(int i, char \*p)

{

int length;

int j, k, m;

char temp[20];

length = strlen(p);

if (length == 1) /\*如果右部为单个符号\*/

{

if (p[0] == '^')

{

if (i >= 0)

{

first[i][0] = '^';

first[i][1] = '\0';

}

else

{

tempFirstSet[0] = '^';

tempFirstSet[1] = '\0';

}

}

else

{

for (j = 0;; j++)

if (allSymbols[j] == p[0])

break;

if (i >= 0)

{

memcpy(first[i], firstSetSingle[j], strlen(firstSetSingle[j]));

first[i][strlen(firstSetSingle[j])] = '\0';

}

else

{

memcpy(tempFirstSet, firstSetSingle[j], strlen(firstSetSingle[j]));

tempFirstSet[strlen(firstSetSingle[j])] = '\0';

}

}

}

else /\*如果右部为符号串\*/

{

for (j = 0;; j++)

if (allSymbols[j] == p[0])

break;

if (i >= 0)

merge(first[i], firstSetSingle[j], 2);

else

merge(tempFirstSet, firstSetSingle[j], 2);

for (k = 0; k <= length - 1; k++)

{

emptyTemp[0] = '\0';

if (\_emp(p[k]) == 1 && k < length - 1)

{

for (m = 0;; m++)

if (allSymbols[m] == right[i][k + 1])

break;

if (i >= 0)

merge(first[i], firstSetSingle[m], 2);

else

merge(tempFirstSet, firstSetSingle[m], 2);

}

else if (\_emp(p[k]) == 1 && k == length - 1)

{

temp[0] = '^';

temp[1] = '\0';

if (i >= 0)

merge(first[i], temp, 1);

else

merge(tempFirstSet, temp, 1);

}

else if (\_emp(p[k]) == 0)

break;

}

}

}

/\*\*求各产生式左部的FOLLOW\*\*/

/\*获取要计算的非终结符c。

将非终结符c添加到followTemp数组中，以避免循环递归。

如果非终结符c是开始符号，将'#'添加到对应的follow数组中。

遍历所有产生式，找到右部含有非终结符c的产生式。

a. 找到非终结符c在产生式右部的位置k。

b. 找到产生式左部非终结符在allSymbols数组中的位置m。

c. 如果非终结符c在产生式右部的最后一个位置：

i. 如果产生式左部非终结符已经在followTemp数组中，将follow[m]合并到follow[i]，并继续下一次循环。

ii. 如果产生式左部非终结符的FOLLOW集尚未计算，调用FOLLOW(m)计算它的FOLLOW集，并将followFlags[m]置为'1'。

iii. 将follow[m]合并到follow[i]。

d. 如果非终结符c不在产生式右部的最后一个位置：

i. 检查非终结符c后面的符号串是否可以推导出空，将结果存储在result变量中。

ii. 如果result为1，表示非终结符c后面的符号串可以推导出空：

1) 如果产生式左部非终结符已经在followTemp数组中，将follow[m]合并到follow[i]，并继续下一次循环。

2) 如果产生式左部非终结符的FOLLOW集尚未计算，调用FOLLOW(m)计算它的FOLLOW集，并将followFlags[m]置为'1'。

3) 将follow[m]合并到follow[i]。

iii. 提取非终结符c后面的符号串，调用FIRST(-1, temp)计算它的FIRST集，将结果合并到follow[i]。

将followFlags[i]置为'1'，表示非终结符c的FOLLOW集已经计算。

\*/

void FOLLOW(int i)

{

int j, k, m, n, result = 1;

char c, temp[20];

c = non\_ter[i]; /\*c为待求的非终结符\*/

temp[0] = c;

temp[1] = '\0';

merge(followTemp, temp, 1);

if (c == start)

{ /\*若为开始符号\*/

temp[0] = '#';

temp[1] = '\0';

merge(follow[i], temp, 1);

}

for (j = 0; j <= count - 1; j++)

{

if (in(c, right[j]) == 1) /\*找一个右部含有c的产生式\*/

{

for (k = 0;; k++)

if (right[j][k] == c)

break; /\*k为c在该产生式右部的序号\*/

for (m = 0;; m++)

if (allSymbols[m] == left[j])

break; /\*m为产生式左部非终结符在所有符号中的序号\*/

if (k == strlen(right[j]) - 1)

{ /\*如果c在产生式右部的最后\*/

if (in(allSymbols[m], followTemp) == 1)

{

merge(follow[i], follow[m], 1);

continue;

}

if (followFlags[m] == '0')

{

FOLLOW(m);

followFlags[m] = '1';

}

merge(follow[i], follow[m], 1);

}

else

{ /\*如果c不在产生式右部的最后\*/

for (n = k + 1; n <= strlen(right[j]) - 1; n++)

{

emptyTemp[0] = '\0';

result \*= \_emp(right[j][n]);

}

if (result == 1)

{ /\*如果右部c后面的符号串能推出^\*/

if (in(allSymbols[m], followTemp) == 1)

{ /\*避免循环递归\*/

merge(follow[i], follow[m], 1);

continue;

}

if (followFlags[m] == '0')

{

FOLLOW(m);

followFlags[m] = '1';

}

merge(follow[i], follow[m], 1);

}

for (n = k + 1; n <= strlen(right[j]) - 1; n++)

temp[n - k - 1] = right[j][n];

temp[strlen(right[j]) - k - 1] = '\0';

FIRST(-1, temp);

merge(follow[i], tempFirstSet, 2);

}

}

}

followFlags[i] = '1';

}

/\*判断读入文法是否为一个LL(1)文法\*\*/

/\*初始化相关变量，如first、follow、firstSetSingle、select等数组以及firstFlags和followFlags标志数组。

对所有输入符号，计算它们的单个符号FIRST集合。

对所有产生式的右部，计算它们的FIRST集合。

计算所有非终结符的FOLLOW集合。

对所有产生式，计算它们的SELECT集合。如果产生式右部的符号串可以推导出空，将FOLLOW集合合并到SELECT集合中。

判断输入文法是否为LL(1)文法。为此，遍历所有产生式，并合并相同左部的产生式的SELECT集合。如果合并后的SELECT集合长度小于原SELECT集合长度之和，说明存在重叠，输入的文法不是LL(1)文法。否则，输入的文法是LL(1)文法。

\*/

int isLL1Grammar()

{

int i, j, length, result = 1;

char temp[50];

for (j = 0; j <= 49; j++)

{ /\*初始化\*/

first[j][0] = '\0';

follow[j][0] = '\0';

firstSetSingle[j][0] = '\0';

select[j][0] = '\0';

tempFirstSet[j] = '\0';

temp[j] = '\0';

firstFlags[j] = '0';

followFlags[j] = '0';

}

for (j = 0; j <= strlen(allSymbols) - 1; j++)

computeSingleFirst(j); /\*求单个符号的FIRST集合\*/

printf("\nfirstSetSingle");

for (j = 0; j <= strlen(allSymbols) - 1; j++)

printf("%c:%s ", allSymbols[j], firstSetSingle[j]);

printf("\nempty:%s", empty);

printf("\nemp:");

for (j = 0; j <= strlen(allSymbols) - 1; j++)

printf("%d ", \_emp(allSymbols[j]));

for (i = 0; i <= count - 1; i++)

FIRST(i, right[i]); /\*求FIRST\*/

printf("\n");

for (j = 0; j <= strlen(non\_ter) - 1; j++)

{ /\*求FOLLOW\*/

if (followTemp[j] == 0)

{

followTemp[0] = '\0';

FOLLOW(j);

}

}

printf("\nfirst:");

for (i = 0; i <= count - 1; i++)

printf("%s ", first[i]);

printf("\nfollow:");

for (i = 0; i <= strlen(non\_ter) - 1; i++)

printf("%s ", follow[i]);

for (i = 0; i <= count - 1; i++)

{ /\*求每一产生式的SELECT集合\*/

memcpy(select[i], first[i], strlen(first[i]));

select[i][strlen(first[i])] = '\0';

for (j = 0; j <= strlen(right[i]) - 1; j++)

result \*= \_emp(right[i][j]);

if (strlen(right[i]) == 1 && right[i][0] == '^')

result = 1;

if (result == 1)

{

for (j = 0;; j++)

if (allSymbols[j] == left[i])

break;

merge(select[i], follow[j], 1);

}

}

printf("\nselect:");

for (i = 0; i <= count - 1; i++)

printf("%s ", select[i]);

memcpy(temp, select[0], strlen(select[0]));

temp[strlen(select[0])] = '\0';

for (i = 1; i <= count - 1; i++)

{ /\*判断输入文法是否为LL(1)文法\*/

length = strlen(temp);

if (left[i] == left[i - 1])

{

merge(temp, select[i], 1);

if (strlen(temp) < length + strlen(select[i]))

return (0);

}

else

{

temp[0] = '\0';

memcpy(temp, select[i], strlen(select[i]));

temp[strlen(select[i])] = '\0';

}

}

return (1);

}

/\*\*构造分析表M\*\*\*/

void constructParsingTable()

{

int i, j, k, m;

for (i = 0; i <= 19; i++)

for (j = 0; j <= 19; j++)

parseTable[i][j] = -1;

i = strlen(termin);

termin[i] = '#'; /\*将#加入终结符数组\*/

termin[i + 1] = '\0';

for (i = 0; i <= count - 1; i++)

{

for (m = 0;; m++)

if (non\_ter[m] == left[i])

break; /\*m为产生式左部非终结符的序号\*/

for (j = 0; j <= strlen(select[i]) - 1; j++)

{

if (in(select[i][j], termin) == 1)

{

for (k = 0;; k++)

if (termin[k] == select[i][j])

break; /\*k为产生式右部终结符的序号\*/

parseTable[m][k] = i;

}

}

}

}

/\*\*总控算法\*\*/

/\*读取输入的字符串，并在末尾添加结束符'#'。

初始化分析栈S，将结束符'#'和文法的开始符号放入栈中。

初始化输入字符串的当前位置为j。

获取当前位置的输入字符ch。

进入一个无限循环，分析输入字符串：

a. 如果栈顶的符号属于终结符集合：

i. 如果栈顶符号与当前输入字符不匹配，输出错误信息并返回。

ii. 如果栈顶符号是结束符'#'，说明输入字符串被成功分析，输出结果并返回。

iii. 否则，从栈中弹出栈顶符号，并将输入字符串的位置向前移动一个字符。

b. 否则，栈顶符号是非终结符：

i. 查找栈顶非终结符在non\_ter数组中的序号i。

ii. 查找当前输入字符在termin数组中的序号k。如果找不到，输出词法错误并返回。

iii. 如果预测分析表parseTable[i][k]的值为-1，说明发生了语法错误，输出错误信息并返回。

iv. 否则，将预测分析表parseTable[i][k]对应的产生式右部替换栈顶的非终结符。如果产生式右部是空串（'^'），直接弹出栈顶符号；否则，将产生式右部的符号逆序压入栈中。

c. 输出当前的栈S和未处理的输入字符串。\*/

void syntax()

{

int i, j, k, m, n, p, q;

char ch;

char S[50], str[50];

printf("请输入该文法的句型：");

scanf("%s", str);

getchar();

i = strlen(str);

str[i] = '#';

str[i + 1] = '\0';

S[0] = '#';

S[1] = start;

S[2] = '\0';

j = 0;

ch = str[j];

while (1)

{

if (in(S[strlen(S) - 1], termin) == 1)

{

if (S[strlen(S) - 1] != ch)

{

printf("\n该符号串不是文法的句型！");

return;

}

else if (S[strlen(S) - 1] == '#')

{

printf("\n该符号串是文法的句型.");

return;

}

else

{

S[strlen(S) - 1] = '\0';

j++;

ch = str[j];

}

}

else

{

for (i = 0;; i++)

if (non\_ter[i] == S[strlen(S) - 1])

break;

for (k = 0;; k++)

{

if (termin[k] == ch)

break;

if (k == strlen(termin))

{

printf("\n词法错误！");

return;

}

}

if (parseTable[i][k] == -1)

{

printf("\n语法错误！");

return;

}

else

{

m = parseTable[i][k];

if (right[m][0] == '^')

S[strlen(S) - 1] = '\0';

else

{

p = strlen(S) - 1;

q = p;

for (n = strlen(right[m]) - 1; n >= 0; n--)

S[p++] = right[m][n];

S[q + strlen(right[m])] = '\0';

}

}

}

printf("\nS:%s str:", S);

for (p = j; p <= strlen(str) - 1; p++)

printf("%c", str[p]);

printf(" ");

}

}

/\*\*一个用户调用函数\*\*/

void menu()

{

syntax();

printf("\n是否继续？(y or n):");

scanf("%c", &choose);

getchar();

while (choose == 'y')

{

menu();

}

}

/\*\*主函数\*\*/

int main()

{

int i, j;

start = readGrammar(termin, non\_ter, left, right); /\*读入一个文法\*/

printf("count=%d", count);

printf("\nstart:%c", start);

strcpy(allSymbols, non\_ter);

strcat(allSymbols, termin);

printf("\nallSymbols:%s", allSymbols);

printf("\nnon\_ter:%s", non\_ter);

printf("\ntermin:%s", termin);

printf("\nright:");

for (i = 0; i <= count - 1; i++)

printf("%s ", right[i]);

printf("\nleft:");

for (i = 0; i <= count - 1; i++)

printf("%c ", left[i]);

if (validity == 1)

validity = judge();

printf("\nvalidity=%d", validity);

if (validity == 1)

{

printf("\n文法有效");

isLL1 = isLL1Grammar();

printf("\nisLL1=%d", isLL1);

if (isLL1 == 0)

printf("\n该文法不是一个LL1文法！");

else

{

constructParsingTable();

printf("\n");

for (i = 0; i <= 19; i++)

for (j = 0; j <= 19; j++)

if (parseTable[i][j] >= 0)

printf("parseTable[%d][%d]=%d ", i, j, parseTable[i][j]);

printf("\n");

menu();

}

}

return 0;

}

## 5.2开发环境

运用Visual C++ 6.0软件编写c语言程序。

## 5.3解决的难点

1. 程序无法输入ε。
2. 求解SELECT集合。
3. 输出完整的分析表。
4. 将分析过程的步骤、分析栈、剩余输入串、匹配/接收全部输出。

# 6 测试和试运行

## 6.1测试方法

首先，进行对功能模块进行划分，明确功能测试的人员负责情况。其次对各个模块进行测试。黑盒测试也称功能测试或数据驱动测试，它是在已知产品所应具有的功能，通过测试来检测每个功能是否都能正常使用，在测试时，把程序看作一个不能打开的黑盒子，在完全不考虑程序内部结构和内部特性的情况下，测试者在程序接口进行测试，它只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数锯而产生正确的输出信息，并且保持外部信息（如数据库或文件）的完整性。黑盒测试方法主要有等价类划分、边值分析、因—果图、错误推测等，主要用于软件确认测试。黑盒测试着力于程序外部结构、不考虑内部逻辑结构、针对软件界面和软件功能进行测试。“黑盒法是穷举输入测试，只有把所有可能的输入都作为测试情况使用，才能以这种方法查出程序中所有的错误。实际上测试情况有无穷多个，人们不仅要测试所有合法的输入，而且还要对那些不合法但是可能的输入进行测试。

## 6.2测试用例

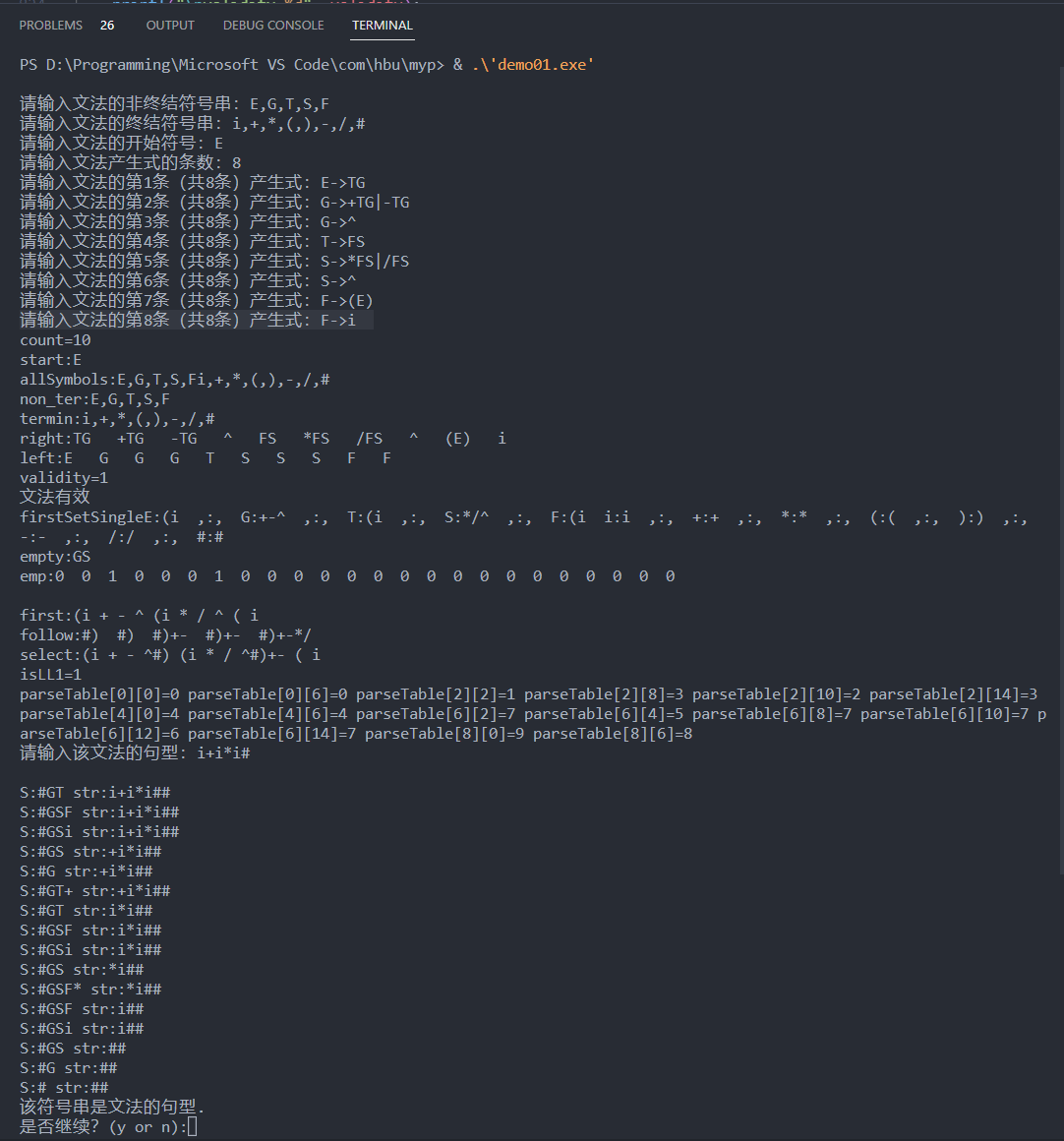


图6.2 测试结果5

# 7. 总结

1、通过这次实验，加强了我动手、思考和解决问题的能力。我选择的是题目三：表驱动LL(1)语法分析程序。通过这几天的实验，我完成了以下内容：对输入的文法的相关信息进行处理，产生first集和follow集；对具有相同产生式左部的SELECT集求交集，判断是否为LL(1)文法，若不是程序结束；构造该文法的预测分析表并输出；输入符号串，判定该符号串是否为该文法的句子；对过程中出现的一些错误能够及时报错。

2.通过这次实验,我懂得了理论与实际相结合是很重要的，只有把所学的理论知识与实践相结合起来，从理论中得出结论，才能真正为社会服务，从而提高自己的实际动手能力和独立思考的能力。在实验中，难免会遇到过各种各样的问题，不过，我也可以从中发现了自己的不足，发现自己对以前所学过的知识理解得不够深刻，掌握得不够牢固。所以，此次实验我受益匪浅。