01

#include <stdio.h>  
/\*  
 \*First(S)={a, b, d, e, ε}  
 \*First(H)={e, ε}  
 \*First(K)={d, ε}  
 \*First(L)={e}  
 \*First(M)={b, d, ε}  
 \*Follow(S)={#, o}  
 \*Follow(H)={#, o, f}  
 \*Follow(K)={#, e, o}  
 \*Follow(L)={o, a, b, d, e}  
 \*Follow(M)={#, e, o}  
 \*/  
char str[100];  
int length;  
int count;  
char stack[100];  
int top;  
void error() {  
 printf(“\t\t不接受!\n”);  
}  
int matcher() {  
 switch(stack[top-1]) {  
 case ‘S’:  
 if(str[count] == ‘b’ || str[count] == ‘d’ ||  
 str[count] == ‘e’ || str[count] == ‘o’ ||  
 str[count] == ‘#’) {  
 top–;  
 stack[top++] = ‘H’;  
 stack[top++] = ‘M’;  
 printf(“\t\tS->MH\n”);  
 } else if(str[count] == ‘a’) {  
 top–;  
 stack[top++] = ‘a’;  
 printf(“\t\tS->a\n”);  
 } else {  
 return -1;  
 }  
 break;  
 case ‘H’:  
 if(str[count] == ‘f’ || str[count] == ‘o’ ||  
 str[count] == ‘#’) {  
 top–;  
 printf(“\t\tH->ε\n”);  
 } else if(str[count] == ‘e’) {  
 top–;  
 stack[top++] = ‘o’;  
 stack[top++] = ‘S’;  
 stack[top++] = ‘L’;  
 printf(“\t\tH->LSo\n”);  
 } else {  
 return -1;  
 }  
 break;  
 case ‘K’:  
 if(str[count] == ‘e’ || str[count] == ‘o’ || str[count] == ‘#’) {  
 top–;  
 printf(“\t\tK->ε\n”);  
 } else if(str[count] == ‘d’) {  
 top–;  
 stack[top++] = ‘L’;  
 stack[top++] = ‘M’;  
 stack[top++] = ‘d’;  
 printf(“\t\tK->dML\n”);  
 } else {  
 return -1;  
 }  
 break;  
 case ‘L’:  
 if(str[count] == ‘e’) {  
 top–;  
 stack[top++] = ‘f’;  
 stack[top++] = ‘H’;  
 stack[top++] = ‘e’;  
 printf(“\t\tL->eHf\n”);  
 } else {  
 return -1;  
 }  
 break;  
 case ‘M’:  
 if(str[count] == ‘d’ || str[count] == ‘e’ || str[count] == ‘o’ || str[count] == ‘#’) {  
 top–;  
 stack[top++] = ‘K’;  
 printf(“\t\tM->K\n”);  
 } else if(str[count] == ‘b’) {  
 top–;  
 stack[top++] = ‘M’;  
 stack[top++] = ‘L’;  
 stack[top++] = ‘b’;  
 printf(“\t\tM->bLM\n”);  
 } else {  
 return -1;  
 }  
 break;  
 default:  
 return -1;  
 }  
 return 0;  
}  
void print() {  
 int i;  
 for(i = 0; i < top; i++) {  
 printf(“%c”, stack[i]);  
 }  
 printf(“\t\t”);  
 for(i = 0; i < count; i++) {  
 printf(“ “);  
 }  
 for(i = count; i < length; i++) {  
 printf(“%c”, str[i]);  
 }  
}  
int main() {  
 printf(“\ta\tb\td\te\to\tf\t#\n”);  
 printf(“S\tS->a\tS->MH\tS->MH\tS->MH\tS->MH\t\tS->MH\n”);  
 printf(“H\t\t\t\tH->LSo\tH->ε\tH->ε\tH->ε\n”);  
 printf(“K\t\t\tK->dML\tK->ε\tK->ε\t\tK->ε\n”);  
 printf(“L\t\t\t\tL->eHf\t\t\t\n”);  
 printf(“M\t\tM->bLM\tM->K\tM->K\tM->K\t\tM->K\n”);  
 while(1) {  
 printf(“\n\n”);  
 char c=’ ‘;  
 int i = 0;  
 while(c!=’#’ && c!=’$’) {  
 scanf(“%c”, &c);  
 str[i++] = c;  
 }  
 getchar();  
 length = i;  
 top = 0;  
 stack[top++] = ‘#’;  
 stack[top++] = ‘S’;  
 count = 0;  
 printf(“栈\t\t输入串\t\t匹配或使用的产生式\n”);  
 while(top-1 != 0 || count != length-1) {  
 if(stack[top-1] == str[count]) {  
 print();  
 printf(“\t\t’%c’匹配\n”, str[count]);  
 top–;  
 count++;  
 } else {  
 print();  
 if(matcher() == -1) {  
 error();  
 break;  
 }  
 }  
 }  
 if(top-1 == 0 && count == length-1) {  
 print();  
 printf(“\t\t接受\n”);  
 }  
 }  
 return 0;  
}

02 词法分析

#include <stdio.h>  
#include <string.h>  
char str[80],token[8];  
char ch;  
int syn,p,m=0,n,sum=0;  
double small = 0;  
char \*tab[10] = {“begin”,“end”,“var”,“integer”,“real”,“if”,“then”,“read”,“write”,“while”};  
/\*\*  
 \*整数 1  
 \*小数 2  
 \*字符串 3

- begin 10 end 11 var 12 integer 13 real 14

- if 15 then 16 read 17 write 18 while 19

- - 20 - 21 \* 22 / 23 := 24 = 25

- >26 < 27 >= 28 <= 29

- , 30 . 31 ; 32 ( 33 ) 34

- { 35 } 36 : 37  
\*/  
void scaner() {  
for(n=0; n<8; n++) token[n]=NULL;  
ch=str[p++];  
while(ch==’ ‘) {  
 ch=str[p];  
 p++;  
}  
//关键字和变量名识别  
if((ch>=’a’&&ch<=’z’)||(ch>=’A’&&ch<=’Z’)) {  
 m=0;  
 while((ch>=’0’&&ch<=’9’)||(ch>=’a’&&ch<=’z’)||(ch>=’A’&&ch<=’Z’)) {  
 token[m++]=ch;  
 ch=str[p++];  
 }  
 token[m++]=’\0’;  
 p–;  
 syn=3;  
 for(n=0; n<10; n++)  
 if(strcmp(token,tab[n])==0) {  
 syn=10+n;  
 break;  
 }  
} else if((ch>=’0’&&ch<=’9’)) { //整数和小数识别  
 sum=0;  
 while((ch>=’0’&&ch<=’9’) || ch==’.’) {  
 if(ch==’.’) {  
 ch=str[p++];  
 double rate = 0.1;  
 small = sum;  
 while((ch>=’0’&&ch<=’9’)) {  
 small = small + (ch-‘0’)*rate;*  
 *rate = rate*0.1;  
 ch=str[p++];  
 }  
 p–;  
 syn=2;  
 break;  
 }  
 sum=sum\*10+ch-‘0’;  
 ch=str[p++];  
 }  
 if(syn!=2) {  
 p–;  
 syn=1;  
 }  
} else switch(ch) { //运算符等其它符号识别  
 case’<’:  
 m=0;  
 token[m++]=ch;  
 ch=str[p++];  
 if(ch==’=’) {  
 syn=29;  
 token[m++]=ch;  
 } else {  
 syn=27;  
 p–;  
 }  
 break;  
 case’>’:  
 m=0;  
 token[m++]=ch;  
 ch=str[p++];  
 if(ch==’=’) {  
 syn=28;  
 token[m++]=ch;  
 } else {  
 syn=26;  
 p–;  
 }  
 break;  
 case’:’:  
 m=0;  
 token[m++]=ch;  
 ch=str[p++];  
 if(ch==’=’) {  
 syn=24;  
 token[m++]=ch;  
 } else {  
 syn=37;  
 p–;  
 }  
 break;  
 case’+’:  
 syn=20;  
 token[0]=ch;  
 break;  
 case’-‘:  
 syn=21;  
 token[0]=ch;  
 break;  
 case’\*’:  
 syn=22;  
 token[0]=ch;  
 break;  
 case’/’:  
 syn=23;  
 token[0]=ch;  
 break;  
 case’=’:  
 syn=25;  
 token[0]=ch;  
 break;  
 case’(‘:  
 syn=33;  
 token[0]=ch;  
 break;  
 case’)’:  
 syn=34;  
 token[0]=ch;  
 break;  
 case’{’:  
 syn=35;  
 token[0]=ch;  
 break;  
 case’}’:  
 syn=36;  
 token[0]=ch;  
 break;  
 case’,’:  
 syn=30;  
 token[0]=ch;  
 break;  
 case’.’:  
 syn=31;  
 token[0]=ch;  
 break;  
 case’;’:  
 syn=32;  
 token[0]=ch;  
 break;  
 case’\n’: //去掉换行  
 syn=-2;  
 break;  
 case’#’:  
 syn=0;  
 token[0]=ch;  
 break;  
 default:  
 if(ch != ‘#’)  
 syn=-1;  
 break;  
 }  
}  
int main() {  
printf(“请输入字符串:\n”);  
while(1) {  
 p=0;  
 do {  
 scanf(“%c”, &ch);  
 str[p++]=ch;  
 } while(ch!=’#’);  
 p=0;  
 do {  
 scaner();  
 switch(syn) {  
 case 1:  
 printf(“(%d,%d)\n”,syn,sum);  
 break;  
 case 2:  
 printf(“(%d,%f)\n”,syn,small);  
 break;  
 case -1:  
 printf(“Error %c\n”, ch);  
 break;  
 case -2:  
 break;  
 default:  
 if(token[0] != ‘#’)  
 printf(“(%d,%s)\n”,syn,token);  
 break;  
 }  
 } while (syn!=0);  
 printf(“请再次输入：\n”);  
}  
}

实验三 递归下降分析法

**一、实验内容**

用高级语言实现递归下降分析程序。使用输入串i\*(i+i)，输出分析栈中所有内容，并给出分析结果。

**二、前期准备**

2.1 递归下降分析法原理

自顶向下分析就是从文法的开始符出发并寻找出这样一个推导序列：推导出的句子恰好为输入符号串；或者说，能否从根节点出发向下生长出一颗语法树，其叶节点组成的句子恰好为输入符号串。显然，语法树的每一步生长（每一步推导）都以能否与输入符号串匹配为准，如果最终句子得到识别，则证明输入符号串为该文法的一个句子；否则，输入符号串不是该文法的句子。

递归下降分析法是一种自顶向下的分析方法，文法的每个非终结符对应一个递归过程（函数）。分析过程就是从文法开始符出发执行一组递归过程（函数），这样向下推到直到推出句子；或者说从根节点出发，自顶向下为输入串寻找一个最左匹配序列，建立一棵语法树。

2.2 要实现的文法

已知要实现的文法如下，可以观察到该文法G[E]中包含直接左递归：

G[E]: E->E+T|T

T->T\*F|F

F->(E)|i

为了实现确定的自顶向下分析，要求文法满足下述两个条件：

（1） 文法不含左递归，即不存在这样的非终结符A：有A->A……存在；

（2） 无回溯，对文法的任一非终结符号，当其产生式右部有多个候选式可供选择时，各候选式所推出的终结符号串的首字符集合要两两不相交。

  因为文法如果包含左递归和回溯在文法分析的时候就可能会做大量无用的工作，导致分析效率降低。

  所以首先需要对该文法消除左递归和回溯，得到如下文法G’[E]：

G’[E]: E->TE’

E’->+TE’|ε

T->FT’

T’->\*FT’|ε

F->(E)|i

另外为了方便编写代码，所以将文法表示符替换成便于书写的单个大写字母，得到新的G[E]文法如下：

G[E]: E->TG

G->+TG|ε

T->FS

S->\*FS|ε

F->(E)|i

2.3 需要的函数

在不含左递归和回溯的条件下，我们就能够构造一个不带回溯的自顶向下的分析程序，这个分析程序是由一组递归过程组成的，每个过程对应文法的一个非终结符。这样的一个分析程序称为递归下降分析器。分析书上给出的伪代码首先可以确定的函数有以下几个：

（1）void E(); //E–>TG

（2）void G(); //G–>+TG|ε

（3）void T(); //T–>FS

（4）void S(); //S–>\*FS|ε

（5）void F(); //F–>(E)|i

（6）void err(); //提示错误信息

然后实验要求对于分析过程中的分析栈进行输出，所以我们使用一个字符串类型的变量stackStr来模拟分析栈，并添加vector类型的链表记录每一步的分析结果，以供输出。在此基础上就需要添加压栈等相关函数如下：

（1）int check();//验证是否已经到栈底

（2）void push(string pre, string value);//将字符串存入输出栈

**三、分析过程**

3.1  递归下降分析法设计思想及算法

为文法G[E]的每个非终结符号E构造一个递归过程，不妨命名为E。E的产生式的右部指出这个过程的代码结构，在匹配时：

（1）若是终结符a，则继续对照，若匹配则输入串向前进一个符号；否则出错。

（2）若是非终结符号A，则调用与此非终结符对应的过程。当A的右部有多个产生式时，可用选择结构实现。

  具体为：

  1）对于每个非终结符号 E->A|B|,|Z 处理的方法如下：

E( )

{

ch=s[i];

if(ch 可能是 A 的首字符 ) 处理 A 的程序部分 ;

else if(ch 可能是 B 的首字符 ) 处理 B 的程序部分 ;

,

else error()

}

2）对于每个右部 A->x1x2,xn的处理架构如下：

处理 x1 的程序；

,

处理 xn 的程序；

3）如果右部为空，则不处理。  
 4）对于右部中的每个符号 xi

① 如果 xi 为终结符号：

if(xi == 当前的符号 )

{

NextChar() ； //NextChar 为前进一个字符函数

return;

}

else 出错处理2;

② 如果 xi 为非终结符号，直接调用相应的过程 xi()

LL(1)分析法

一、实验目的

1.能够判断一文法是否为LL(1)文法；

2.能构造LL(1)语法分析表

3.用程序实现LL(1)的语法分析过程

二、实验内容

已知文法G[S]：

S→MH|a

H→LSo|ε

K→dML|ε

L→eHf

M→K|bLM

 三．**实验要求：**

1．给出文法的FIRST集和FOLLOW集(程序实现）。

2．构造该文法的语法分析表。

3．根据语法分析表模拟LL（1）语法分析过程。

4．设计相应的例子进行测试，输入所给句子的语法分析过程。

ade#

aedh#

算符优先文法

一、实验目的

1.能够判断一文法是否为算符优先文法；

2.能构造算符优先文法分析表

3.用程序实现算符优先文法分析过程

二、实验内容

文法G(E):

E->E+T|T

T->T\*F|F

F->(E)|i

三、实验要求

1．给出文法的FIRSTVT集和LASTVT集(程序实现）。

2．构造该文法的语法分析表。

3．根据语法分析表模拟算符优先文法分析过程。

i+i\*i#

i\*i+a#