```
/* Compilador del Lenguaje Micro (Fischer)       */
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define NUMESTADOS 15
#define NUMCOLS 13
#define TAMLEX 32+1
#define TAMNOM 20+1
/******************Declaraciones Globales*************************/
FILE * in;
typedef enum {
 INICIO, FIN, LEER, ESCRIBIR, ID, CONSTANTE, PARENIZQUIERDO, PARENDERECHO, PUNTOYCOMA,
COMA, ASIGNACION, SUMA, RESTA, FDT, ERRORLEXICO
} TOKEN;
typedef struct {
    char identifi[TAMLEX];
    TOKEN t;      /* t=0, 1, 2, 3 Palabra Reservada, t=ID=4 Identificador (ver enum) */
    } RegTS;
RegTS TS[1000] = { {"inicio", INICIO}, {"fin", FIN}, {"leer", LEER}, {"escribir", ESCRIBIR}, {"$", 99} };

typedef struct{
    TOKEN clase;
    char nombre[TAMLEX];
    int valor;
    } REG_EXPRESION;

char buffer[TAMLEX];
TOKEN tokenActual;
int flagToken = 0;

/*********************Prototipos de Funciones************************/
TOKEN scanner(); // el scanner
int columna(int c);
int estadoFinal(int e);
void Objetivo(void); // del PAS
void Programa(void);
void ListaSentencias(void);
void Sentencia(void);
void ListaIdentificadores(void);
void Identificador(REG_EXPRESION * presul);
void ListaExpresiones(void);
void Expresion(REG_EXPRESION * presul);
void Primaria(REG_EXPRESION * presul);
void OperadorAditivo(char * presul);

REG_EXPRESION ProcesarCte(void);
REG_EXPRESION ProcesarId(void);
```

```
char * ProcesarOp(void);
void Leer(REG_EXPRESION in);
void Escribir(REG_EXPRESION out);
REG_EXPRESION GenInfijo(REG_EXPRESION e1, char * op, REG_EXPRESION e2);

void Match(TOKEN t);
TOKEN ProximoToken();
void ErrorLexico();
void ErrorSintactico();
void Generar(char * co, char * a, char * b, char * c);
char * Extraer(REG_EXPRESION * preg);
int Buscar(char * id, RegTS * TS, TOKEN * t);
void Colocar(char * id, RegTS * TS);
void Chequear(char * s);
void Comenzar(void);
void Terminar(void);
void Asignar(REG_EXPRESION izq, REG_EXPRESION der);

/**************************Programa Principal************************/
int main(int argc, char * argv[])
{
 TOKEN tok;
 char nomArchi[TAMNOM];
 int l;
/**************************Se abre el Archivo Fuente*****************/
// verifica errores posibles
if ( argc == 1 ) {
  printf("Debe ingresar el nombre del archivo fuente (en lenguaje Micro) en la linea de
comandos\n");  return -1;  }// no puso nombre de archivo fuente
 if ( argc != 2 ) {
  printf("Numero incorrecto de argumentos\n");  return -1;  }//los argumentos deben ser 2
 strcpy(nomArchi, argv[1]);
 l = strlen(nomArchi);
 if ( l > TAMNOM ) {
  printf("Nombre incorrecto del Archivo Fuente\n"); return -1; }
// requiere para compilar un archivo de extensión.m archivo.m
 if ( nomArchi[l-1] != 'm' || nomArchi[l-2] != '.' ) {
 printf("Nombre incorrecto del Archivo Fuente\n"); return -1; }
 if ( (in = fopen(nomArchi, "r") ) == NULL )  {
  printf("No se pudo abrir archivo fuente\n");  return -1;//no pudo abrir archivo
 }
/**************************Inicio Compilacion*************************/
 Objetivo();
/**************************Se cierra el Archivo Fuente*****************/
 fclose(in);
 return 0;
}
```

/**********Procedimientos de Analisis Sintactico (PAS) *****************/
/**********Procedimientos de Analisis Sintactico (PAS) *****************/

```
void Objetivo(void)
{
 /* <objetivo> -> <programa> FDT #terminar */

 Programa();
 Match(FDT);
 Terminar();
}
```

```
void Programa(void)
{
 /* <programa> -> #comenzar INICIO <listaSentencias> FIN */
 Comenzar();//de inicio semántico en caso de corresponder
 Match(INICIO);
 ListaSentencias();
 Match(FIN);
}
```

```
void ListaSentencias(void)
{
 /* <listaSentencias> -> <sentencia> {<sentencia>} */
 Sentencia();
 while ( 1 ) {// se repite hasta que retorna al no encontrar sentencia {<sentencia>}
  switch ( ProximoToken() )  {
          case ID : case LEER : case ESCRIBIR :
          Sentencia();
          break;
          default : return; //si no es sentencia termina la funcion
    }// fin del switch
  }// fin del while
}// fin funcion
```

```
void Sentencia(void) {
 TOKEN tok = ProximoToken();
 REG_EXPRESION izq, der;
//typedef struct{  TOKEN clase;  char nombre[TAMLEX];  int valor;  } REG_EXPRESION;
 switch ( tok )  {
  case ID :          /* <sentencia> -> ID := <expresion> #asignar ; */
   Identificador(&izq);
   Match(ASIGNACION);
   Expresion(&der);
   Asignar(izq, der); //genera instrucción de asignacion
   Match(PUNTOYCOMA);
   break;
  case LEER :      /* <sentencia> -> LEER ( <listaIdentificadores> ) */
   Match(LEER);
   Match(PARENIZQUIERDO);
   ListaIdentificadores();
   Match(PARENDERECHO);
   Match(PUNTOYCOMA);
   break;
  case ESCRIBIR :           /* <sentencia> -> ESCRIBIR ( <listaExpresiones> ) */
   Match(ESCRIBIR);
   Match(PARENIZQUIERDO);
   ListaExpresiones();
   Match(PARENDERECHO);
   Match(PUNTOYCOMA);
   break;
  default : return;
 }
}
```

```
void ListaIdentificadores(void) {
 /* <listaIdentificadores> -> <identificador> #leer_id {COMA <identificador> #leer_id} */
 TOKEN t;
 REG_EXPRESION reg;
 Identificador(&reg);
 Leer(reg);
 for ( t = ProximoToken(); t == COMA; t = ProximoToken() ) {
  Match(COMA);
  Identificador(&reg);
  Leer(reg);
 }
}
```

```
void Identificador(REG_EXPRESION * presul) {
 /* <identificador> -> ID #procesar_id */
 Match(ID);
 *presul = ProcesarId();//rutina semantica
}
```

```
void ListaExpresiones(void) {
 /* <listaExpresiones> -> <expresion> #escribir_exp {COMA <expresion> #escribir_exp} */
 TOKEN t;
 REG_EXPRESION reg;
 Expresion(&reg);
 Escribir(reg);
 for ( t = ProximoToken(); t == COMA; t = ProximoToken() ) {
          Match(COMA);
          Expresion(&reg);
          Escribir(reg);
  }
}
```

```
void Expresion(REG_EXPRESION * presul){
 /* <expresion> -> <primaria> { <operadorAditivo> <primaria> #gen_infijo } */
 REG_EXPRESION operandoIzq, operandoDer;
 char op[TAMLEX];
 TOKEN t;
 Primaria(&operandoIzq);
 for ( t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken() ) {
          OperadorAditivo(op);
          Primaria(&operandoDer);
          operandoIzq = GenInfijo(operandoIzq, op, operandoDer);
   }
   *presul = operandoIzq;
}
```

```
void Primaria(REG_EXPRESION * presul) {
 TOKEN tok = ProximoToken();
 switch ( tok ) {
  case ID :          /* <primaria> -> <identificador> */
   Identificador(presul);    break;
  case CONSTANTE :       /* <primaria> -> CONSTANTE #procesar_cte */
   Match(CONSTANTE);   *presul = ProcesarCte();    break;
  case PARENIZQUIERDO :          /* <primaria> -> PARENIZQUIERDO <expresion> PARENDERECHO
   Match(PARENIZQUIERDO);    Expresion(presul);
   Match(PARENDERECHO);    break;
  default : return;
 }
}
```

```
void OperadorAditivo(char * presul) {
 /* <operadorAditivo> -> SUMA #procesar_op | RESTA #procesar_op */
 TOKEN t = ProximoToken();
 if ( t == SUMA || t == RESTA )  {
  Match(t);
  strcpy(presul, ProcesarOp());
 }  else
  ErrorSintactico(t);
}
```

/**********Rutinas Semanticas**********/

```
REG_EXPRESION ProcesarCte(void)
{
 /* Convierte cadena que representa numero a entero y construye un registro semantico */
 REG_EXPRESION reg;
 reg.clase = CONSTANTE;
 strcpy(reg.nombre, buffer);
 sscanf(buffer, "%d", &reg.valor);
 return reg;
}
```

```
REG_EXPRESION ProcesarId(void) {
 /* Declara ID y construye el correspondiente registro semantico */
 REG_EXPRESION reg;
 Chequear(buffer); //function auxiliar
 reg.clase = ID;
 strcpy(reg.nombre, buffer);
 return reg;
}
```

```
char * ProcesarOp(void) {
 /* Declara OP y construye el correspondiente registro semantico */
 return buffer;
}
```

```
void Leer(REG_EXPRESION in) {
 /* Genera la instruccion para leer */
 Generar("Read", in.nombre, "Entera", "");
}
```

```
void Escribir(REG_EXPRESION out) {
 /* Genera la instruccion para escribir */
 Generar("Write", Extraer(&out), "Entera", "");
}
```

```
REG_EXPRESION GenInfijo(REG_EXPRESION e1, char * op, REG_EXPRESION e2){
 /* Genera la instruccion para una operacion infija y construye un registro semantico con el
resultado */
 REG_EXPRESION reg;
 static unsigned int numTemp = 1;
 char cadTemp[TAMLEX] ="Temp&";
 char cadNum[TAMLEX];
 char cadOp[TAMLEX];
 if ( op[0] == '-' ) strcpy(cadOp, "Restar");
 if ( op[0] == '+' ) strcpy(cadOp, "Sumar");
 sprintf(cadNum, "%d", numTemp);
 numTemp++;
 strcat(cadTemp, cadNum);
 if ( e1.clase == ID) Chequear(Extraer(&e1));
 if ( e2.clase == ID) Chequear(Extraer(&e2));
 Chequear(cadTemp);
 Generar(cadOp, Extraer(&e1), Extraer(&e2), cadTemp);
 strcpy(reg.nombre, cadTemp);
 return reg;
}
```

/***************Funciones Auxiliares*********************************/

```
void Match(TOKEN t) {
 if ( !(t == ProximoToken()) ) ErrorSintactico();
 flagToken = 0;
}
```

```
TOKEN ProximoToken() {
 if ( !flagToken )  {
  tokenActual = scanner();
  if ( tokenActual == ERRORLEXICO ) ErrorLexico();
  flagToken = 1;
  if ( tokenActual == ID )   {
   Buscar(buffer, TS, &tokenActual);
  }
 }
 return tokenActual;
}
```

```
void ErrorLexico() {
 printf("Error Lexico\n");
}
```

```
void ErrorSintactico() {
 printf("Error Sintactico\n");
}
```

```
void Generar(char * co, char * a, char * b, char * c) {
 /* Produce la salida de la instruccion para la MV por stdout */
 printf("%s %s%c%s%c%s\n", co, a, ',', b, ',', c);
}
```

```
char * Extraer(REG_EXPRESION * preg) {
 /* Retorna la cadena del registro semantico */
 return preg->nombre;
}
```

```
int Buscar(char * id, RegTS * TS, TOKEN * t) {
 /* Determina si un identificador esta en la TS */
 int i = 0;
 while ( strcmp("$", TS[i].identifi) ) {
  if ( !strcmp(id, TS[i].identifi) )  {
   *t = TS[i].t;
   return 1;
  }
  i++;
 }
 return 0;
}
```

```
void Colocar(char * id, RegTS * TS){
 /* Agrega un identificador a la TS */
 int i = 4;
 while ( strcmp("$", TS[i].identifi) ) i++;
  if ( i < 999 ) {
  strcpy(TS[i].identifi, id );
  TS[i].t = ID;
  strcpy(TS[++i].identifi, "$" );
 }
}
```

```
void Chequear(char * s){
 /* Si la cadena No esta en la Tabla de Simbolos la agrega,
    y si es el nombre de una variable genera la instruccion */
 TOKEN t;
 if ( !Buscar(s, TS, &t) ) {
  Colocar(s, TS);
  Generar("Declara", s, "Entera", "");
 }
}
```

```
void Comenzar(void) {
 /* Inicializaciones Semanticas */
}
```

```
void Terminar(void) {
 /* Genera la instruccion para terminar la ejecucion del programa */
 Generar("Detiene", "", "", "");
}
```

```
void Asignar(REG_EXPRESION izq, REG_EXPRESION der){
 /* Genera la instruccion para la asignacion */
 Generar("Almacena", Extraer(&der), izq.nombre, "");
}
```

```
/*************************Scanner*************************************/
TOKEN scanner()
{
 int tabla[NUMESTADOS][NUMCOLS] =
            L  D  +  -  (  )  ,  ;  :  =  EOF ´ ´ OTRO
0        {{ 1,  3 ,  5 ,  6 ,  7  ,  8  ,  9 , 10, 11, 14, 13,  0, 14 },
 1       {  1,  1 ,  2 ,  2 ,  2  ,  2  ,  2 ,  2 ,  2 ,  2 ,  2 ,  2 },
 2 ID    { 14, 14, 14, 14, 14, 14, 14, 14, 14 , 14, 14, 14, 14 },
3         {  4,  3 ,  4 ,  4 ,  4  ,  4  ,  4 ,  4 ,  4 ,  4,  4,  4 },
4 CTE    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
5 +      { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
6 -      { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
7 (      { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
8 )      { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
9 ,      { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
10 ;     { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
11       { 14, 14, 14, 14, 14, 14, 14, 14, 14, 12, 14, 14, 14 },
12 ASIG{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
13 fdt   { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
14 Err   { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 } };
 int car;
 int col;
```

```
int estado = 0;
int i = 0;
do {
 car = fgetc(in);
 col = columna(car);
 estado = tabla[estado][col];
 if ( col != 11 )  { //si es espacio no lo agrega al buffer
  buffer[i] = car;
  i++;
 }
}
while ( !estadoFinal(estado) && !(estado == 14) );
buffer[i] = '\0'; //complete la cadena
switch ( estado )
{
 case 2 : if ( col != 11 ){            //si el carácter espureo no es blanco…
              ungetc(car, in);         // lo retorna al flujo
              buffer[i-1] = '\0';
             }
             return ID;
 case 4 : if ( col != 11 )  {
              ungetc(car, in);
              buffer[i-1] = '\0';
             }
        return CONSTANTE;
 case 5 : return SUMA;
 case 6 : return RESTA;
 case 7 : return PARENIZQUIERDO;
 case 8 : return PARENDERECHO;
 case 9 : return COMA;
 case 10 : return PUNTOYCOMA;
 case 12 : return ASIGNACION;
 case 13 : return FDT;
 case 14 : return ERRORLEXICO;
}
 return 0;
}
```

```
int estadoFinal(int e){
 if ( e == 0 || e == 1 || e == 3 || e == 11 || e == 14 ) return 0;
 return 1;
}
```

```
int columna(int c){
 if ( isalpha(c) ) return 0;
 if ( isdigit(c) ) return 1;
 if ( c == '+' ) return 2;
 if ( c == '-' ) return 3;
 if ( c == '(' ) return 4;
 if ( c == ')' ) return 5;
 if ( c == ',' ) return 6;
 if ( c == ';' ) return 7;
 if ( c == ':' ) return 8;
 if ( c == '=' ) return 9;
 if ( c == EOF ) return 10;
 if ( isspace(c) ) return 11;
 return 12;
}
```

/**********Fin Scanner**********/