scaner.l: Utilizando la Herramienta Flex, se genera un Escaner, el cual se encargará de reconocer patrones léxicos del archivo con el codigo micro.

%{%}: Se llaman bibliotecas y prototipos.

#include <stdio.h>
Biblioteca que agrega funciones de manejo de entradas y salidas
#include <stdlib.h>
La utilizamos para EXIT_FAILURE
#include <string.h>
Biblioteca que agrega funciones de manejo de strings

- yylex(): función principal del analizador léxico flex,analizar una entrada de texto y devolver tokens al analizador sintáctico
- yyerror(const char mensaje): Funcion de Bison, (definida en Parser), utilizada para reportar errores de sintaxis. En el scanner se utiliza para indicar los errores lexicos.
- yylineno: Variable Global utilizada en flex utilizado para rastrear el numero de linea del archivo en el que se encuentra el analizador lexico .
- yylexerrs:Variable global utilizada por flex encargada de seguir el numero de errores lexicos durante el analisis del archivo.

Definiciones: Nombra a Determinadas Expresiones Regulares para simplificar el análisis y comprensión del código, y aclarar otras Expresiones. En esta se Define:

- Digito: Todo numero del 0 al 9, se reconoce como DIGITO.
- Letra: Toda letra de “a” a “z” y de “A” a “Z”, se reconoce como LETRA.
- Constante: Toda cadena que comienza con un DIGITO y Finaliza o continua con otro DIGITO, se reconoce como CONST.
- Identificador: Toda cadena que comienza con LETRA y finaliza o continua con LETRA o DIGITO, se reconoce como IDENTIFICADOR.
- Error: Asegura que el scanner al leer un Patron que comienza con DIGITO y finaliza, o continua con un DIGITO o LETRA lo reconozca como error o ID_ERROR.

%%Reglas%%: A partir de Expresiones Regulares(Patron) y las Definiciones se indican los Tokens a Reconocer del Archivo a Analizar, y las Acciones a Realizar una vez reconocidos.

Patrón o Definicion	Tipo	Acción
:=	Operador	Retorna el Token ASIGNACION
;	Carácter Puntuación	Retorna el Token PUNTOYCOMA
,	Carácter Puntuación	Retorna el Token COMA
inicio	Palabra Reservada	Retorna el Token INICIO
leer	Palabra Reservada	Retorna el Token LEER
escribir	Palabra Reservada	Retorna el Token ESCRIBIR
fin	Palabra Reservada	Retorna el Token FIN
+	Operador	Retorna el Token SUMA
-	Operador	Retorna el Token RESTA
)	Carácter Puntuación	Retorna el Token PARENDERECHO
(	Carácter Puntuación	Retorna el Token PARENIZQUIERDO
IDENTIFICADOR	Identificador no digito	yylval.cadena = strdup(yytext) Retorna el Token ID
CONST	Constante	yylval.num =atoi(yytext) Retorna el Token CONSTANTE

ID_ERROR	-	Aumenta conteo de errores Lexicos Indica Error de identificador no valido
\n		aumenta conteo yylineo
[\tEOF]		No realiza accion, ignora espacio en blanco, tabulaciones y final de Archivo
.		Aumenta conteo de errores Lexicos Indica Error de caracter invalido

- yylval: Variable global utilizada para almacenar un valor asociado al Token reconocido.
    - yylval.cadena: Almacena el nombre (IDENTIFICADOR) del token ID analizado.
    - yylval.num: Almacena el valor (CONST) del token CONSTANTE analizado.
- strdup(): Duplica una cadena de caracteres, utilizada para duplicar lo leido por yytext, almacenado en yylval.cadena.
- yytext: variable global de Flex, contiene el texto con la Expresion Regular que el analizador lexico acaba de reconocer.
- atoi(): Convierte cadena de caracteres que representa un numero entero en dicho valor pero en int.
- sprintf():Formatea y almacena una cadena en un buffer, en un formato específico y lo almacena en una cadena de caracteres. En este caso lo almacena en mensajeDeError

Codigo: En el caso del Trabajo simplemente se Utiliza la Función yywrap(). Dicha función es la encargada de finalizar el análisis léxico una vez alcanzado el final del Archivo.

parserConListaIDs.y

```
%{
#include <stdio.h>
```
Biblioteca que agrega funciones de manejo de entradas y salidas
```
#include <stdlib.h>
```
La utilizamos para EXIT_FAILURE
```
#include <string.h>
```
Biblioteca que agrega funciones de manejo de strings

```
extern char *yytext;
```
Puntero a una cadena de caracteres que contiene el valor del lexema actual.

```
extern int yyleng;
```
Guarda la longitud del token almacenado en yytext

```
extern int yylex(void);
```
función principal del analizador léxico flex,analizar una entrada de texto y devolver tokens al analizador sintáctico

```
extern void yyerror(char*);
```
Es una función que maneja los errores sintácticos y léxicos durante el análisis. Recibe como parámetro un mensaje de error (char*) que describe el problema

```
void asignarIds(char* nombre, int valor);
```
Función para asignar valores a los IDs que no están definidos

```
extern int yylineno;
```
contiene el número de la línea actual que está siendo procesada por el analizador léxico. Actualizada automáticamente por flex

```
extern int yylexerrs;
```
Es una variable global que almacena la cantidad de errores léxicos que han ocurrido durante la ejecución del analizador léxico.

```
extern FILE* yyin;
```
yyin es un puntero a un archivo, que es el archivo de entrada que flex está analizando.

```
struct Identificador{
  char* nombre;
  int valor;
};
```
Estructura que define como se componen los IDs:(nombreID,valorID)

```
struct Identificador listaIdentificadores[80];
int cantidadIdentificadores = 0;
%}
```
listaIdentificadores está compuesta por IDs y tiene como máximo 80. La inicializamos vacía

```
int lineaActual = 0;
int erroresLexicos = 0;
int erroresSintacticos = 0;
int errorID = 0;
int errorTotal= 0;

typedef enum {
    CORRECTO,
    ERROR,
    ERROR_MEMORIA
} Estado;

Estado estadoActual = CORRECTO;
%}
```
Las secciones entre %{ %} en Bison delimitan código en C que se inserta tal cual en el archivo C generado por Bison. Todo lo que esté dentro de estas delimitaciones se copia directamente al inicio del archivo de salida que Bison genera, antes de las funciones del analizador léxico y sintáctico.
```
%union {
  char* cadena;
  int num;
}
```
Esta estructura define todos los tipos de datos posibles que utilizaremos para las reglas gramaticales (Cadenas de caracteres y números enteros)

```
%token ASIGNACION PUNTOYCOMA RESTA SUMA PARENIZQUIERDO PARENDERECHO INICIO FIN LEER
ESCRIBIR COMA
%token <cadena> ID
%token <num> CONSTANTE
```
Se definen los tipos de tokens posibles: IDs, constantes numericas y palabras reservadas ya definidas en el scanner.

```
programa: INICIO listaSentencias FIN
;
```
Un programa en micro siempre debe iniciar con INICIO, luego tiene una lista de sentencias y debe terminar con FIN
```
listaSentencias: listaSentencias sentencia
| sentencia
;
```
Una lista de sentencias puede ser una única sentencia o un conjunto de sentencias (que se genera por la recursividad en esta definición)

```
sentencia: instruccion PUNTOYCOMA
```
Una sentencia es una instrucción seguida de un punto y coma
```
| error{
        yyclearin;
        yyerrok;
```

```
}
 ;
```
Nota de nico: Aparentemente si este error lo vuelo a la mierda el compi no lee el codigo completo --(solo si hay un error de sentencia sin punto y coma). Del manejo de este error se encarga el propio bison

```
instruccion: ID ASIGNACION expresion {
char* nombre = $<cadena>1;
int valor = $<num>3;
asignarIds(nombre, valor);
}
| LEER PARENIZQUIERDO listaIds PARENDERECHO {
            printf("Lee %s\n", $<cadena>3);
}
| ESCRIBIR PARENIZQUIERDO listaExpresiones PARENDERECHO{
           printf("Escribe %d\n", $<num>3);
}
;
```
Una instrucción es un LEER/ESCRIBIR con sus respectivas gramáticas o una asignación de una expresión a un ID

Cuando lee o escribe lo hace con el tercer argumento que encuentra (por el $<algo>3)
Lo mismo pasa para asignar ID, le asigna el número que está en el tercer argumento a la cadena que está en el primer argumento.

```
listaIds: listaIds COMA ID
{
           strcat($<cadena>1, ",");
           strcat($<cadena>1, $<cadena>3); }
| ID
;
```
Para hacer la lista de IDs primero pone una coma al final de la lista y después de esto le pone el ID

```
listaExpresiones: listaExpresiones COMA expresion
| expresion
;

expresion: primaria
| expresion SUMA primaria
{$<num>$ = $<num>1 + $<num>3;}
| expresion RESTA primaria
{$<num>$ = $<num>1 - $<num>3;} ;
```

Bueno esto es lo mismo que antes xd

```
primaria: ID
{ char* nombre = $<cadena>1;
            int i;
           for (i = 0; i < cantidadIdentificadores; i++) {
                      if (strcmp(listaIdentificadores[i].nombre, nombre) == 0) {
                      $<num>$ = listaIdentificadores[i].valor;
                       break;
           }
```
Si el ID ya está en la lista le asigna el valor guardado

```
}
if (i == cantidadIdentificadores) {
char mensajeDeError[100];
sprintf(mensajeDeError, "La variable %s no ha sido definida con ningun valor", nombre);
yyerror(mensajeDeError);
errorID++;
}
}
```

Si termina de recorrer toda la lista y no lo encuentra salta error ya que no se definió anteriormente

```
| CONSTANTE
| PARENIZQUIERDO expresion PARENDERECHO {
$<num>$ = $<num>2; }
| error {
--Salta al siguiente punto y coma
            yyclearin;
            yyerrok;
}
;
```

Si primaria no coincide con ninguno de los anteriores va a error. Del manejo de este se encarga el propio bison

```
%%

int main(int argc, char** argv) {
   // Validación de argumentos
   if (argc == 1) {
      printf("Debe ingresar el nombre del archivo fuente (en lenguaje Micro) en la línea de comandos\n");
      return -1;
   } else if (argc != 2) {
      printf("Número incorrecto de argumentos\n");
      return -1;
   }
```

Tenemos que pasar dos argumentos en argv (que es una cadena de cadenas(ponele que una matriz)) el primero es el ejecutable y el otro es el archivo micro. Esta cantidad de argumentos se almacena en argc. De ahí las comprobaciones que hace.

```
   // Carga del nombre del archivo
   char filename[50];
   sprintf(filename, "%s", argv[1]);
   int longitudArchivo = strlen(filename);
```

sprintf():Formatea y almacena una cadena en un buffer, en un formato específico y lo almacena en una cadena de caracteres. En este caso lo almacena en filename

```
   // Verificación de la extensión del archivo
   if (argv[1][longitudArchivo - 1] != 'm' || argv[1][longitudArchivo - 2] != '.') {
      printf("Extensión incorrecta (debe ser .m)\n");
      return EXIT_FAILURE;
   }
```

Verificación por medio de los últimos dos caracteres que sea “.m”

```
   // Apertura del archivo
   yyin = fopen(filename, "r");
   if (yyin == NULL) {
```

```
        perror("Error al abrir el archivo");
        return EXIT_FAILURE;
    }

    // Análisis sintáctico
    switch (yyparse()) {
        case 0: estadoActual = CORRECTO; break;
        case 1: estadoActual = ERROR; break;
        case 2: estadoActual = ERROR_MEMORIA; break;
    }

    // Verificación de errores
    if ((errorTotal || yylexerrs || errorID) && estadoActual != ERROR_MEMORIA)
        estadoActual = ERROR;

    // Mensajes según el estado de compilación
    switch (estadoActual) {
        case CORRECTO:
            printf("\nProceso de compilación terminó exitosamente, código correcto sintácticamente\n");
            break;
        case ERROR:
            printf("\nErrores en la compilación\n");
            break;
        case ERROR_MEMORIA:
            printf("\nNo hay memoria suficiente\n");
            break;
    }
```

Esto lo implementamos con un switch para poder recorrer el programa entero por más de que haya algún error en la compilación y así poder devolver la cantidad total de errores. Una vez pasa al case ERROR imprimirá que hubieron errores en la compilación.(empieza en CORRECTO)

```
    // Cálculo de errores
    erroresSintacticos = errorTotal - erroresLexicos - errorID;
    printf("\nErrores sintácticos: %i\tErrores léxicos: %i\tErrores totales: %i\n", erroresSintacticos, erroresLexicos,
errorTotal);

    // Cierre del archivo
    fclose(yyin);
    return 0;
}

// Manejo de errores
void yyerror(char *s) {
    if (lineaActual != yylineno) {
        lineaActual = yylineno;
        fprintf(stderr, "ERROR: %s en la línea %d\n", s, yylineno);
        erroresLexicos = yylexerrs;
        errorTotal++;
    } else if (erroresLexicos != yylexerrs) {
        fprintf(stderr, "ERROR: %s en la línea %d\n", s, yylineno);
        erroresLexicos = yylexerrs;
        errorTotal++;
    }
}
```

Si la línea actual es diferente a la última línea que arrojó un error actualiza la línea actual con la línea donde ocurrió el error (Para evitar el bucle que se nos formaba) e imprime el mensaje de error en la salida estándar de error (stderr)
Actualiza la cantidad de errores léxicos a partir de la variable global e incrementa el contador de errores totales

```
// Asignación de identificadores
void asignarIds(char* nombre, int valor) {
    int i;
    // Busca si el identificador ya existe
    for (i = 0; i < cantidadIdentificadores; i++) {
        if (strcmp(listaIdentificadores[i].nombre, nombre) == 0) {
            listaIdentificadores[i].valor = valor;
            break;
        }
    }

    // Si no existe, agrega uno nuevo
    if (i == cantidadIdentificadores) {
        listaIdentificadores[cantidadIdentificadores].nombre = nombre;
        listaIdentificadores[cantidadIdentificadores].valor = valor;
        cantidadIdentificadores++;
    }
}
```

Aca se hace la comprobación de que no exista el ID. De no existir se guarda el nombre junto con el valor del identificador en la lista de IDs (Para esta asignación es que utilizamos el struct Identificador)