UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Proyecto 1

"Analizador léxico"

Compiladores

Grupo 03

Integrantes:

Alcantar Correa Vianey

Sánchez Rosas Alexis Alejandro

Profesor: M.C. LAURA SANDOVAL MONTAÑO

Fecha de entrega: 28/Marzo/2023

Índice

Índice 1

Objetivo 2

Descripción del problema 2

Propuesta de solución y partes del desarrollo del sistema 4

- Lenguaje lex 4
- Expresiones regulares 5
  - Identificadores 5
  - Constantes cadenas 7
  - Constante carácter 8
  - Constantes numéricas reales 8
- Catálogos 8
- Tabla de símbolos 10
- Tablas de literales 11
- Manejo de archivos 13
- Generando tokens 14
- Funciones extra 14
  - Convertir hexadecimales a decimales……

Indicaciones de como correr el programa 16

- Ilustración de proceso de ejecución 17

Conclusiones 20

Alcantar Correa Vianey 20

Sánchez Rosas Alexis Alejandro 20

Objetivo

Elaborar un analizador léxico en lex/flex que reconozca los componentes léxicos pertenecientes a las 10 clases descritas en clase.

Descripción del problema

El analizador léxico es un pilar fundamental en el funcionamiento de un compilador, esté, se encarga de recibir el código fuente y leer su contenido para dar comienzo al proceso que compete la materia.

Sus principales tareas consisten en:

1. Crea las tablas. Las de tipo catálogo, las llena; a las otras, sólo define su estructura.
2. Lee carácter por carácter del programa fuente hasta formar una cadena que sea identificada como un componente léxico, como un error o el carácter de fin de archivo.
3. Si se trata de un componente léxico, dependiendo de la clase a la que pertenezca, realiza lo siguiente:
    - Si es un identificador, lo busca en la tabla de símbolos. Si lo encuentra, entonces toma la posición del identificador en la tabla de símbolos, la cual va a ser el campo “valor” del token. Si el identificador no está en la tabla de símbolos, lo coloca ahí y guarda, en el campo “valor” del token, la posición donde fue colocado. Finalmente genera el token con los campos clase (identificador) y valor (posición en la tabla de símbolos), y se lo entrega al analizador sintáctico.
    - Si es una cadena o una constante, la busca en la tabla de literales. Y sigue el mismo procedimiento del identificador, sólo que utiliza la tabla de literales. El token que genera tendrá como clase: cadena o tipo de constante(entera, real, etc), y como valor, la posición dentro de la tabla de literales.
    - Si es palabra reservada, busca en qué posición, del catálogo correspondiente, se encuentra y la guarda en el campo “valor” del token. Crea el token correspondiente con clase (pal. reservada) y valor (posición en el catálogo) y se lo entrega al analizador sintáctico.
    - Para las otras clases de componentes léxicos, que están definidas en catálogos, se realiza el mismo procedimiento de las palabras reservadas. Crea el token con la clase correspondiente (operador relacional, lógico, aritmético, etc.) y valor (posición en el catálogo correspondiente) y se lo entrega al analizador sintáctico.

- Se regresa al paso 2.

4. Si es una cadena o carácter no reconocido envía un mensaje de error y se regresa al paso 2

5. Si es el carácter es el fin de archivo (EOF), termina el proceso de análisis léxico

Las **clases** propuestas para la construcción del analizador son las siguientes:

Clase	Descripción
0	Identificadores. Sólo letras minúsculas y _ Inicia con letra minúscula y con tamaño mínimo de dos.
1	Símbolos especiales ( ) { } ; , [ ]
2	Operadores relacionales (ver tabla).
3	Operador de asignación ->
4	Palabras reservadas (ver tabla).
5	Operadores aritméticos + - * / %
6	Constantes numéricas enteras. En base 10 (secuencia de dígitos del 0-9 sin 0's a la izquierda, en base 8 (inicien con 0 y le sigan dígitos del 0 al 7) y en base hexadecimal (inicien con 0x o 0X y le sigan los dígitos 0 al 9, 'a' a la 'f', y 'A' a la 'F')
7	Constantes cadenas. Encerradas entre comillas (") cualquier secuencia de caracteres que no contenga salto de línea.
8	Constante carácter. Entre apóstrofos (')
9	Constantes numéricas reales. Ejemplos: 73.0, .85, 10.2, 4. No aceptados: . , 12

Valor	Palabra reservada
0	ent
1	finsi
2	haz
3	mientras
4	large
5	para
6	real
7	regresa
8	si
9	simb
10	sino

Valor	Op. relacional
0	=
1	/=
2	>=
3	>
4	<
5	<=

Además nuestro analizador cuenta con ciertas consideraciones que lo diferencian de un analizador que sigue todas las reglas arriba descritas.

- Existen catálogos para las palabras reservadas, los operadores relacionales y el operador de asignación. Los símbolos especiales y operadores aritméticos no cuentan con sus propias tablas.
- Nuestro analizador no contará con un carácter de fin de archivo.
- Se tendrán 2 tablas de literales una para constantes reales y otra para constantes cadenas.
- Al terminar su ejecución deben mostrarse las tablas, los tokens y los errores léxicos.

Propuesta de solución y partes del desarrollo del sistema

Lenguaje lex

La estructura básica de un programa en Lex consta de tres secciones principales, que se definen mediante etiquetas especiales:

1. Sección de definiciones: esta sección comienza con %{ y termina con %}. En esta sección, se pueden definir macros, prototipos de funciones, variables globales, etc. Esta sección se copia directamente en el archivo de salida generado por el programa de Lex.

2. Sección de reglas: esta sección es donde se especifican las reglas de análisis léxico. Las reglas se escriben en una forma de expresión regular, seguida de una acción en C que se ejecutará cuando la expresión regular coincida con una entrada. Cada regla se separa de las demás con %%.

3. Sección de código C: esta sección es donde se pueden incluir funciones y variables definidas por el usuario en C, así como código C que se ejecutará antes o después del análisis léxico. Esta sección también se copia directamente en el archivo de salida generado por el programa de Lex.

La estructura general de un programa en Lex se ve así:

```
%{
  /* Sección de definiciones */
%}
/* Sección de reglas */
%%
/* Sección de código C */
%%
/* Funciones auxiliares y definiciones adicionales */
```

En donde “%%” se utiliza para separar las tres secciones principales de un programa Lex.
El primer "%%" indica el final de la sección de definiciones y el comienzo de la sección de reglas. El segundo "%%" indica el final de la sección de reglas y el comienzo de la sección de código C. En resumen, los símbolos "%%"

Expresiones regulares

Se define una expresión regular para cada clase que reconocerá el analizador.

Identificadores

[a-z]([a-z]|_)*

Esta expresión regular utiliza los siguientes elementos:

[a-z]: Representa cualquier carácter alfabético en minúscula, es decir, de la 'a' a la 'z'.

(): Paréntesis que agrupan los elementos de la expresión regular.

|: Barra vertical que indica alternativa, es decir, puede ser uno de los dos elementos separados por la barra.

*: Asterisco que indica que el elemento anterior puede aparecer cero o más veces.

En conjunto, esta expresión regular indica que la cadena de entrada debe comenzar con una letra minúscula y puede ser seguida por cero o más apariciones de letras minúsculas o guiones bajos ('_') en cualquier orden.

Por ejemplo, algunas cadenas válidas para esta expresión regular son: "a", "variable", "mi_variable", "una_variable_mas". Mientras que algunas cadenas inválidas son: "1a", "variable!", "Variable", "var-iable".

Símbolos especiales

[(){};,[]]

Explicación de la expresión regular:

[‘y’] delimitan el conjunto de caracteres que la expresión regular debe coincidir.

() {} , ; y [] son los caracteres dentro del conjunto.

Por lo tanto, esta expresión regular coincide con cualquier cadena que contenga cualquiera de los caracteres (,), {, }, ,, ;, [o].

Operadores relacionales

=|\/=|>=|>|<|<=

Explicación de la expresión regular:

“=” coincide con el carácter igual.

“\/”= coincide con el carácter de barra inclinada hacia adelante (/) seguido del carácter igual para formar la operación "no igual".

“>=” coincide con el operador mayor o igual.

“>“ coincide con el operador mayor que.

“<” coincide con el operador menor que.

“<=” coincide con el operador menor o igual.

Por lo tanto, esta expresión regular coincide con cualquiera de los operadores de comparación =, /=, >=, >, < y <=.

Operador de asignación

“->”

Explicación de la expresión regular:
“->” coincide con el operador de flecha, que se utiliza en muchos lenguajes de programación para acceder a un miembro o método de una estructura o clase.
Por lo tanto, esta expresión regular coincide con cualquier cadena que contenga el operador de flecha ->.

Palabras reservadas

“ent|finsi|haz|mientras|large|para|real|regresa|si|simb|sino”

Explicación de la expresión regular:
ent, finsi, haz, mientras, large, para, real, regresa, si, simb y sino son las palabras reservadas del lenguaje que se quieren reconocer.
Por lo tanto, esta expresión regular coincide con cualquiera de las palabras reservadas ent, finsi, haz, mientras, large, para, real, regresa, si, simb y sino.
Operadores aritméticos

“[+\-*\/%]”

Explicación de la expresión regular:
“[ ]” delimita el conjunto de caracteres que la expresión regular debe coincidir.
“+, -, *, / y %” son los caracteres dentro del conjunto.
Por lo tanto, esta expresión regular coincide con cualquiera de los operadores aritméticos +, -, *, / y %.

Constantes numéricas enteras (decimal,octal y hexadecimal)

Decimal:

“{dig}{digcero}*”

Explicación de la expresión regular:
“{dig}” es una expresión regular que coincide con cualquier dígito decimal (0 a 9).

“{digcero}” es una expresión regular que coincide con cualquier dígito decimal (0 a 9) o el carácter 0.

“*” indica que se pueden repetir cero o más veces los caracteres que coinciden con {digcero}.

Por lo tanto, esta expresión regular coincide con cualquier cadena que comience con un dígito decimal (0 a 9) seguido de cero o más dígitos decimales o el carácter 0

Octal:

“0[0-7]+”

Explicación de la expresión regular:

0 coincide con el carácter cero.

[0-7] es una expresión regular que coincide con cualquier dígito octal (0 a 7).

+ indica que se pueden repetir una o más veces los caracteres que coinciden con [0-7].

Por lo tanto, esta expresión regular coincide con cualquier cadena que comience con un cero seguido de uno o más dígitos octales (0 a 7).

Hexadecimal:

“0[xX][0-9a-fA-F]+”

Explicación de la expresión regular:

“0” coincide con el carácter cero.

“[xX]” coincide con cualquiera de los caracteres x o X.

“[0-9a-fA-F]” es una expresión regular que coincide con cualquier dígito hexadecimal (0 a 9, a a f o A a F).

“+” indica que se pueden repetir una o más veces los caracteres que coinciden con [0-9a-fA-F].

Por lo tanto, esta expresión regular coincide con cualquier cadena que comience con un cero, seguido de un x o X, y luego uno o más dígitos hexadecimales (0 a 9, a a f o A a F). En resumen, esta expresión regular reconoce números hexadecimales con un cero inicial.

Constantes cadenas

\"[^\"]*\"

Explicación de la expresión regular:

\" coincide con las comillas dobles (").

[^\"]* Es una expresión regular que coincide con cero o más caracteres que no son comillas dobles (").

\" coincide con las comillas dobles (").

Por lo tanto, esta expresión regular coincide con cualquier cadena que esté encerrada entre comillas dobles ("), y que no contenga comillas dobles (") en su interior. En resumen, *esta expresión regular reconoce cadenas de caracteres que están entrecomilladas.*

Constante carácter

\'[^\']*\'

Explicación de la expresión regular:
"\' " coincide con las comillas simples (').
" [^\']* " es una expresión regular que coincide con cero o más caracteres que no son comillas simples (').

" \' " coincide con las comillas simples (').

Por lo tanto, esta expresión regular coincide con cualquier cadena que esté encerrada entre comillas simples ('), y que no contenga comillas simples (') en su interior. En resumen, *esta expresión regular reconoce cadenas de caracteres que están entre comillas por comillas simples.*

Constantes numéricas reales

" 0\.[0-9]+ "

Explicación de la expresión regular:

" 0 " coincide con el carácter cero.

" \. " coincide con el punto (.). El backslash \ se usa para escapar el punto y que sea tratado literalmente, ya que el punto es un metacaracter que coincide con cualquier caracter.

" [0-9]+ " es una expresión regular que coincide con uno o más dígitos numéricos.

Por lo tanto, esta expresión regular coincide con cualquier cadena que comience con un cero seguido de un punto y uno o más dígitos numéricos. En resumen, esta expresión regular reconoce números decimales que empiezan con cero antes del punto decimal.

Catálogos

Los catálogos son las tablas estáticas del compilador, estas son determinadas por el mismo analizador léxico por lo que se cree conveniente colocarlos en arreglos de cadenas de caracteres a las que

simplemente se accede buscando coincidencias con los datos del arreglo. Tenemos un arreglo para las palabras reservadas, otra para los operadores relacionales y una última para los operadores

```
41  char  *resWordTable[] = {"ent","finsi","haz","mientras","large","para","real","regresa","si","simb","sino"};
42  char  *relOpTable[] = {"=","/=",">=",">","<","<="};
43  char  *asigOpTable [] = {"->"};
```

Cada vez que el analizador detecta coincidencia con alguna de estas 3 clases se manda a llamar a la función writeToken que recibe 2 números enteros y se encarga de generar los tokens para el archivo de salida. Además el segundo argumento de la función se obtiene del valor de retorno de otra función que varía dependiendo la clase de la cadena identificada.

```
108  {palres}       { writeToken(4, resWordIndex(yytext)); }
109  {oprelacional} { writeToken(2, relIndex(yytext)); }
110  {opasignacion} { writeToken(3, asigIndex(yytext)); }
```

Para el operador relacional se compara la cadena de entrada con todas y cada una de las cadenas almacenadas en el arreglo relOpTable, cuando se encuentra una coincidencia la función retorna la posición para correspondiente en la tabla de operadores relacionales. Cabe destacar que este ciclo se repite 6 veces pues solo existen 6 operadores relacionales en nuestro lenguaje a aceptar.

```
220  //Función para obtener el valor del operador en su tabla
221  int relIndex(char* yytext()) {
222        // 6 = tamaño de tabla de operadores relacionales
223      for (int i = 0; i <= 6; i++) {
224          // Compara la cadena que entra a la función con cada cadena del arreglo de operadores relacionales
225          if (strcmp(yytext, relOpTable[i]) == 0) {
226              return i;
227          }
228      }
229      return -1;
230  }
```

Ocurre exactamente lo mismo con la función para el operador de asignación pero esta tabla solamente cuenta con un elemento por lo que siempre se retorna la primera posición del arreglo a la función writeToken.

```
244  //Función para obtener el valor de operador de asignacion en su tabla
245  int asigIndex(char* yytext()){
246    for (int i = 0; i <= 10; i++) {
247          // Compara la cadena que entra a la función con cada cadena del arreglo de operadores de asignacion
248          if (strcmp(yytext, asigOpTable[i]) == 0) {
249              return i;
250          }
251      }
252      return -1;
253  }
```

Finalmente se tiene la función para obtener el valor de las palabras reservadas, la lógica de la función es la misma pero en esta ocasión el ciclo for itera 10 veces para revisar coincidencias con cada una de las 10 palabras reservadas almacenadas en nuestro catálogo.

```
232  //Función para obtener el valor de la palabra reservada en su tabla
233  int resWordIndex(char* yytext()){
234        // 10 = tamaño de tabla de palabras reservadas
235      for (int i = 0; i <= 10; i++) {
236          // Compara la cadena que entra a la función con cada cadena del arreglo de palabras reservadas
237          if (strcmp(yytext, resWordTable[i]) == 0) {
238              return i;
239          }
240      }
241      return -1;
242  }
```

Tabla de símbolos

Para la construcción de la tabla de símbolos se utiliza una estructura que almacena los 3 datos que solicita el programa, el primero es la posición del dato dentro de la tabla, después un arreglo de cadenas que almacena el nombre del identificador y finalmente un valor entero que indicará el tipo del identificador, este último será utilizado más adelante por el analizador sintáctico.

```
27  typedef struct{
28      int pos;
29      char* name;
30      int type ;
31  }identificador;
32
```

Además también se asigna un arreglo dinámico para almacenar los datos de la tabla, este arreglo se crea inicialmente de tamaño 100 pero más adelante se modifica su tamaño en caso de necesitarlo.

```
identificador *symbolTable;
```

```
symbolTable    = (identificador *)malloc(100*sizeof(identificador));
```

Cada vez que el analizador detecta un identificador el programa llamará a 2 funciones writeToken que se encarga de crear el token para el archivo de salida, está función además recibe el valor de la clase y el valor del identificador como argumentos. El valor del identificador se obtiene con el retorno de la función addToSymbTab.

```
107   {identificador} { writeToken(0, addToSymbTab(yytext)); }
```

La función addToSymbTab es de tipo entero y recibe la cadena de caracteres leída por el analizador, esta función se encarga de copiar la cadena de caracteres que recibe la función y la almacena en un nuevo arreglo de cadenas utilizando la función strcpy. Posteriormente se hace una búsqueda comparando elemento por elemento dentro de la tabla de símbolos, si existe alguna coincidencia la función simplemente encontrará la posición del identificador dentro de la tabla y posteriormente la regresará como el valor de retorno de la función.

```
int addToSymbTab (char* yytext()){
    // Asigna el inicio de un bloque de memoria dinámica al apuntador ptr
  char *ptr = calloc(strlen(yytext) + 1, sizeof(char));
  // Copia el contenido de yytext a ptr
  strcpy(ptr, yytext);
     for (int i = 0; i < pointerTS; i++) {
    // Guarda en el apuntador name el nombre del identificador que se encuentra en la posición i del arreglo
     char *name = symbolTable[i].name;
    // Compara name con el apuntador ptr en caso de haber coincidencia
      if (strcmp(name, ptr) == 0) {
    // Retorna el valor del índice que corresponde a la posición en la tabla de símbolos para este identificador
          symbolTable[i].pos = i;
          return i;
      }
    }
    if (pointerTS == sizeTS) {
        sizeTS *= 2;
        stringTable = realloc(stringTable, sizeof(char*) * sizeTS);
        if (stringTable == NULL) {
            printf("Error: memoria insuficiente.\n");
            exit(1);
        }
    }

    symbolTable[pointerTS].name = ptr;
    symbolTable[pointerTS].type = -1;
    pointerTS++;

    // Retorna el valor del índice que corresponde a la posición en la tabla de símbolos para este identificador
    return pointerTS - 1;
}
```

Si no encuentra coincidencia en el arreglo entonces se hará una comparación entre el tamaño de la cadena y el tamaño disponible en el arreglo dinámico, en caso de que no haya espacio suficiente el tamaño del arreglo dinámico se redimensiona al doble del original utilizando la función realloc. Finalmente se guarda el nuevo elemento dentro de la tabla de símbolos utilizando el arreglo de cadenas que se declara dentro de la función y además se establece el atributo type del dato con un valor de -1

Tablas de literales

Las tablas de literales para cadenas y constantes reales se almacenarán en arreglos dinámicos, estos arreglos tendrán un tamaño inicial de 100 y podrán reasignar su tamaño destinado de memoria según sea necesario utilizando la misma estrategia empleada en la tabla de símbolos.

```
40   float   *realTable;
41   char  **stringTable;
```

Cada vez que se encuentre una constante cadena o una constante flotante se manda a llamar dos funciones, una para escribir los tokens y otra para agregar las constantes a sus respectivas tablas de símbolos.

```
{constcadena}   { addToStringTable(yytext); writeToken(7, pointerTLString-1); }
{constreal}     { addToRealTable(atof(yytext)); writeToken(9,pointerTR-1); }
```

Para las constantes cadenas primero se crea una copia del arreglo que lee el analizador léxico y posteriormente se verifica si la cadena a insertar tiene espacio disponible dentro de la tabla de literales, en caso de no contar con espacio disponible este se duplica haciendo uso de la función realloc, una vez que se ha asegurado el espacio dentro de la tabla se manda a insertar la cadena de caracteres reconocida en la tabla de literales para finalmente incrementar el valor de la variable que corresponde al índice de la siguiente cadena, este valor modificado además es utilizado por la función WriteToken para encontrar el valor de la cadena dentro de su respectiva tabla y poder generar así el token correctamente.

```
// Función para agregar una cadena a la tabla de cadenas
void addToStringTable(char* yytext){
    // Asigna memoria dinámicamente para la cadena
    char *ptr = (char*)malloc(sizeof(char) * (strlen(yytext) + 1));
    // Copia el contenido de la cadena 'yytext' en la cadena recién asignada
    strcpy(ptr, yytext);
    // Verifica si el índice actual es igual al tamaño de la tabla de cadenas
    if(pointerTLString == sizeTLString){
        // Si es así, duplica el tamaño de la tabla
        sizeTLString *= 2;
        // Usa 'realloc()' para asignar la nueva cantidad de memoria y mantener los valores previos de la tabla
        stringTable = realloc(stringTable, sizeof(char*) * sizeTLString);
    }
    // Almacena el puntero de la cadena recién asignada en la tabla de cadenas
    stringTable[pointerTLString] = ptr;
    // Incrementa el índice para la próxima cadena
    pointerTLString++;
}
```

Cuando se va a añadir una constante real a su tabla se encuentra el desafío de convertir la cadena de caracteres a un número flotante. C cuenta con una función de nombre atof que nos permite realizar está tarea de forma sencilla, entonces la función addToRealTable simplemente verifica el espacio disponible dentro de nuestro arreglo dinámico y en caso de tener espacio suficiente inserta el nuevo elemento en su respectiva tabla de literales, modifica además el valor de la variable que nos sirve de referencia para la ubicación de la cadena recién ingresada.

```
314  //Función para agregar constante real a su respectiva tabla
315  void addToRealTable(float yytext){
316      if(pointerTR == sizeTR){
317          sizeTR *= 2;
318          stringTable = realloc(stringTable, sizeof(char*) * sizeTR);
319      }
320      realTable[pointerTR] = yytext;
321      pointerTR++;
322  }
```

Cabe destacar que a diferencia de la tabla de símbolos en estas tablas de literales no se verifica si el elemento a insertar se encuentra ya dentro de la tabla, esté solo es insertado.

Manejo de archivos

El programa lee un archivo de entrada especificado por el primer argumento de línea de comando, se comienza abriendo el archivo de entrada especificado por el primer argumento de línea de comando usando la función fopen(). Luego, comprueba si se proporcionó exactamente un argumento de línea de comando. Si no se proporcionó exactamente un argumento, el programa imprime un mensaje de error y sale. Si se proporcionó exactamente un argumento, el programa continúa.

```
51   FILE* inFile = fopen(argv[1], "r");
52   // Comprueba si se proporcionó exactamente un argumento de línea de comando
53   if (argc != 2) {
54     printf("Uso: %s <archivo_entrada>\n", argv[0]);
55     return 1;
56   }
```

Se abren tres archivos de salida: "salida.txt", "token.txt" y "tablas.txt" usando la función fopen().

A continuación, el programa comprueba si se pudo abrir el archivo de entrada. Si no se pudo abrir el archivo de entrada, el programa imprime un mensaje de error y sale. Si se pudo abrir el archivo de entrada, el programa configura el archivo de entrada para que sea el que se usará con el analizador léxico (Flex) usando la variable yyin.

```
357  // Abre el archivo de salida "salida.txt" en modo de escritura
358  outFile = fopen("salida.txt","w");
359  tokenFile = fopen("token.txt","w");
360  tableFile = fopen("tablas.txt","w");
361  // Comprueba si se pudo abrir el archivo de entrada
362  if (inFile == NULL) {
363    printf("No se puede abrir el archivo: %s\n", argv[1]);
364    return 1;
365  }
366  // Configura el archivo de entrada para que sea el que se usará con el analizador léxico (Flex)
367  yyin = inFile;
```

Generando tokens

Durante todo el programa se habla de la función que genera tokens, la función en sí misma es bastante sencilla. Consta únicamente de un switch que varía sus acciones dependiendo la clase que detecte el analizador. Dentro de la función se va escribiendo en 2 archivos uno de salida que almacena mensajes que pueden resultar muy ilustrativos para el mejor entendimiento de todo el programa en general y un archivo que se destina a guardar los tokens que genera todo el analizador sin información más que la necesaria.

```
124 void writeToken(int class, int value) {
125     switch(class){
126         case 0:   //Identificadores
127             fprintf(outFile,"identificador: %s con Token: ( %d,%d )\n",yytext,class,value );
128             break;
129         case 1:   //Simbolo especial
130             fprintf(outFile,"Simbolo especial: '%s' con Token: ( %d,%d )\n",yytext,class,value );
131             break;
132         case 2:   //Operador relacional
133             fprintf(outFile,"Operador relacional: %s con Token: ( %d,%d )\n",yytext,class,value );
134             break;
135         case 3:   //Operador de asignacion (caso especial, 2 caracteres y solo un elemento en la tabla)
136             fprintf(outFile,"Operador de asignacion: %s con Token: ( %d,%d )\n",yytext,class,value );
137             break;
138         case 4:   //Palabras reservadas
139             fprintf(outFile,"Palabra reservada: %s con Token: ( %d,%d )\n",yytext,class,value );
140             break;
141         case 5:   //Operadores aritmeticos
142             fprintf(outFile,"Operador aritmetico: %s con Token: ( %d,%d )\n",yytext,class,value );
143         break;
144     case 6:   //Constantes numericas enteras
145             fprintf(outFile,"Constante numerica entera: %s con Token: ( %d,%d )\n",yytext,class,value );
146       break;
147         case 7:   //Constantes cadenas
148             fprintf(outFile,"Constante cadena: %s con Token: ( %d,%d )\n",yytext,class,value );
149             break;
150         case 8:   //Constantes caracter
151             fprintf(outFile,"Constante caracter: %s con Token: ( %d,%d )\n",yytext,class,value );
152         break;
153       case 9:   //Constante real
154             fprintf(outFile,"Constante numerica real: %s con Token: ( %d,%d )\n",yytext,class,value );
155       break;
156         default:
157            break;
158     }
159         fprintf(tokenFile,"%d,%d\n",class,value);
160 }
```

Funciones extra

El programa cuenta además con funcionalidades que pueden ser o no necesarias para cumplir con todos los rubros especificados en la documentación. En este apartado se mencionan algunas de ellas.

Convertir caracteres a ASCII

```
198 int toASCII(char* yytext()){
199   char *aux = calloc(strlen(yytext) + 1, sizeof(char));
200   int ret = 0;
201   if (aux == NULL) {
202     printf("Error: memoria insuficiente.\n");
203     exit(1);
204   }
205
206   // Copia el contenido de yytext a aux
207   strcpy(aux, yytext);
208   if(strlen(yytext)>1){
209     fprintf (outFile,"Convirtiendo caracter %s a su ASCII ---> %d\n",aux,(int)aux[1]);
210     ret= (int)aux[1];
211     free(aux);
212     return ret;
213   }
214   fprintf (outFile,"Convirtiendo caracter %s a su ASCII ---> %d\n",aux,(int)aux[0]);
215   ret= (int)aux[0];
216   free(aux);
217   return ret;
218 }
```

El programa cuenta con una función para, según se requiera, convertir un carácter a su equivalente en ASCII y retornarlo como entero. La función es capaz de convertir caracteres y caracteres entre comillas simples.

Convertir octales a decimales

```
274 // Convierte un número octal representado como una cadena de caracteres en su equivalente decimal y lo devuelve como un entero
275 int convertOctalToDecimal(char * yytext) {
276
277     // Crea una nueva cadena de caracteres para almacenar el número octal sin el primer carácter "0"
278     char *ptr = malloc(sizeof(char) * (strlen(yytext) - 1));
279     for(int i = 0; i < strlen(yytext)-1; i++){
280         ptr[i] = yytext[i+1];
281     }
282     // Convierte la cadena de caracteres en un entero
283     int n = atoi(ptr);
284     // Inicializa las variables que se usarán en el proceso de conversión a decimal
285     int p = 0, decimal = 0, r;
286     // Convierte el número octal en decimal utilizando el método de división sucesiva por 10
287     while(n>0){
288         // Calcula el residuo de la división de n entre 10
289         r = n % 10;
290         // Divide n entre 10
291         n = n / 10;
292         // Agrega el residuo al número decimal multiplicado por 8 elevado a la potencia p
293         decimal = decimal + r * pow(8, p);
294         // Incrementa la potencia de 8
295         ++p;
296     }
297     // Devuelve el número decimal resultante
298     fprintf(outFile,"Convirtiendo %s en base octal a base decimal -> %d\n",yytext,decimal);
299     return decimal;
300 }
```

El programa requiere tener la capacidad de convertir números octales a decimales para almacenarlos en la tabla de constantes enteras que se encuentra en sistema decimal, el programa debe además ignorar un primer 0 a la izquierda que se encuentra en todos los números octales que reconoce nuestro lenguaje.

Convertir hexadecimales a decimales

```
302 int convertHexaToDecimal(char* yytext){
303   // reserva memoria para la cadena c
304     char *ptr = malloc(sizeof(char) * (strlen(yytext) - 2));
305     // recorre la cadena yytext para obtener los dígitos hexadecimales a convertir
306     for(int i = 0; i < strlen(yytext)-2; i++){
307         ptr[i] = yytext[i+2];
308     }
309     // imprime un mensaje con el número hexadecimal a convertir y llama a la función strtol para convertirlo en decimal
310     fprintf(outFile,"Convirtiendo %s en base hexa a base decimal -> %d\n",yytext,(int)strtol(ptr, NULL, 16));
311     // retorna el resultado de la conversión a decimal
312     return (int)strtol(ptr, NULL, 16);
313 }
```

El programa necesita de igual forma ser capaz de convertir números hexadecimales a decimales, esto lo consigue haciendo uso de la función strlol y de esta manera se puede almacenar también los números hexadecimales en la tabla de constantes enteras. Cabe destacar que el programa debe ignorar los primeros 2 caracteres de cada número hexadecimal pues estos forzosamente deben estar compuestos de un 0x al inicio de sus cadenas para poder ser reconocidos como hexadecimales por nuestro lenguaje.

Indicaciones de como correr el programa

Para correr el programa es necesario utilizar una terminal con bash en linux. Para las pruebas de este proyecto se utilizó la distribución más reciente de ubuntu.

Primero se debe utilizar en la línea de comandos:

$flex AnalizadorLexico.l

Es importante resaltar que la terminación del código fuente debe ser “.l” . Tras ejecutar este comando se creará un archivo escrito en c con nombre lex.yy.c.

Ahora se debe utilizar en la línea de comandos:

$gcc lex.yy.c - lfl -lm

El parámetro -lm se utiliza para enlazar la biblioteca matemática estándar de C, que proporciona una serie de funciones matemáticas como trigonométricas, exponenciales, de raíz cuadrada y logarítmicas, entre otras. Al agregar este parámetro a la línea de comandos de compilación, se asegura que el programa tenga acceso a estas funciones matemáticas.

El parámetro -lfl se utiliza para enlazar la biblioteca Flex, que es un generador de analizadores léxicos utilizado para la generación de código en C para el análisis de texto y el procesamiento de archivos de texto. Al agregar este parámetro a la línea de comandos de compilación, se asegura que el programa tenga acceso a las funciones de la biblioteca Flex necesarias para el análisis léxico del texto.

Cómo no especificamos el nombre del archivo de salida mediante el parámetro -o se le asigna un nombre predeterminada a.out.

Entonces finalmente para ejecutar el programa se utilizará la siguiente línea en la consola de comandos:

$./a.out <nombreArchivo.txt>

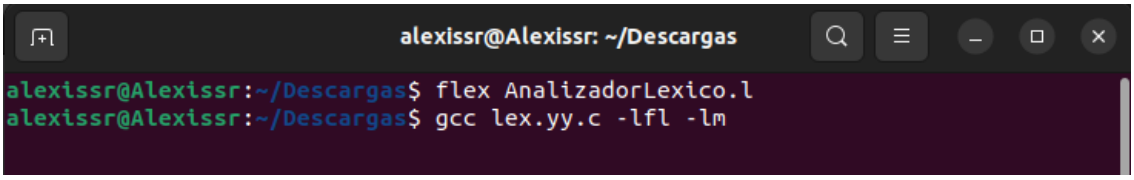
Es importante mencionar que el archivo que pasa por el analizador debe encontrarse en la misma carpeta que el archivo a.out para poder ejecutar el analizador con la anterior línea. También debemos aclarar que la terminación del archivo debe incluirse en caso de existir pero el programa acepta archivos de texto sin terminación.

Una vez ejecutado el programa se crean 3 archivos de texto.

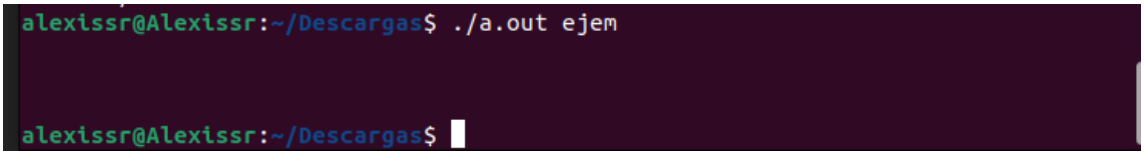
- Salida.txt: contiene toda la información de la ejecución del programa para una visualización cómoda de su ejecución.
- Token.txt: contiene una lista de tokens separados entre sí por un salto de línea.
- Tablas.txt: contiene una impresión estética de la tabla de símbolos, la de cadenas y la de reales.

Ilustración de proceso de ejecución

1.- Escribir comandos en la misma ubicación del código fuente del analizador léxico.



2.- Enseguida se ejecuta el programa utilizando un archivo de texto sin extensión ejemplo



3.- El archivo de entrada tiene el siguiente contenido:

```
ent main(){
        char caracter    -> '$';
        char cadena      -> "Archivos"
        ent diez         -> 10;
        large pi         -> 3.1416;

        large resultado_op;

        resultado_op = diez + pi;

        regresa resultado_op;

}
```

4- Se generan 3 archivos de texto

4.1.-Tablas.txt

```
*** Tabla de Simbolos ***
-------------------------
     0 | main   |     -1
     1 | char   |     -1
     2 | caracter       |     -1
     3 | cadena |     -1
     4 | diez   |     -1
     5 | pi     |     -1
     6 | resultado_op   |     -1
*** Tabla de Cadenas ***
-------------------------
     0 | "Archivos"
*** Tabla de Reales ***
-------------------------
     0 | 3.14
```

4.2.-Token.txt

```
4,0
0,0
1,40
1,41
1,123
0,1
0,2
3,0
8,36
1,59
0,1
0,3
3,0
7,0
4,0
0,4
3,0
6,10
1,59
4,4
0,5
3,0
9,0
1,59
4,4
0,6
1,59
0,6
2,0
0,4
5,43
0,5
```

4.3.-Salidas.txt

```
Palabra reservada: ent con Token: ( 4,0 )
identificador: main con Token: ( 0,0 )
Convirtiendo caracter ( a su ASCII ---> 40
Simbolo especial: '(' con Token: ( 1,40 )
Convirtiendo caracter ) a su ASCII ---> 41
Simbolo especial: ')' con Token: ( 1,41 )
Convirtiendo caracter { a su ASCII ---> 123
Simbolo especial: '{' con Token: ( 1,123 )
identificador: char con Token: ( 0,1 )
identificador: caracter con Token: ( 0,2 )
Operador de asignacion: -> con Token: ( 3,0 )
Convirtiendo caracter '$' a su ASCII ---> 36
Constante caracter: '$' con Token: ( 8,36 )
Convirtiendo caracter ; a su ASCII ---> 59
Simbolo especial: ';' con Token: ( 1,59 )
identificador: char con Token: ( 0,1 )
identificador: cadena con Token: ( 0,3 )
Operador de asignacion: -> con Token: ( 3,0 )
Constante cadena: "Archivos" con Token: ( 7,0 )
Palabra reservada: ent con Token: ( 4,0 )
identificador: diez con Token: ( 0,4 )
Operador de asignacion: -> con Token: ( 3,0 )
Constante numerica entera: 10 con Token: ( 6,10 )
Convirtiendo caracter ; a su ASCII ---> 59
Simbolo especial: ';' con Token: ( 1,59 )
Palabra reservada: large con Token: ( 4,4 )
identificador: pi con Token: ( 0,5 )
Operador de asignacion: -> con Token: ( 3,0 )
Constante numerica real: 3.1416 con Token: ( 9,0 )
Convirtiendo caracter ; a su ASCII ---> 59
Simbolo especial: ';' con Token: ( 1,59 )
Palabra reservada: large con Token: ( 4,4 )
```

5.- Archivos utilizados y generados.

Conclusiones

Alcantar Correa Vianey

En conclusión, el proyecto de elaborar un analizador léxico en Flex que reconociera los componentes léxicos pertenecientes a las 10 clases descritas en clase fue una tarea desafiante pero muy gratificante. La herramienta resultante es de vital importancia para poder continuar con la realización de un compilador, ya que es el pilar sobre el que se construirán los demás componentes del compilador. El proceso de programación del analizador léxico implicó el uso de la expresión regular y la programación en C, así como la configuración y ejecución en Linux. A pesar de los desafíos, se pudo resolver exitosamente el proyecto y obtener un analizador léxico funcional que puede reconocer las diferentes clases de componentes léxicos. En definitiva, este proyecto nos ha permitido adquirir un conocimiento más profundo sobre el proceso de análisis léxico y las herramientas necesarias para llevarlo a cabo

Sánchez Rosas Alexis Alejandro

La realización del analizador léxico en Flex con nuestro propio lenguaje programado en C como proyecto en equipo resultó ser una tarea desafiante pero muy valiosa. El desarrollo del programa involucró la definición de las clases de tokens necesarias para nuestro lenguaje, la creación de expresiones regulares y reglas de análisis léxico para reconocer cada uno de ellos. Aunque hubo dificultades a lo largo del camino, estas se pudieron resolver gracias a la paciencia y dedicación del equipo. El resultado final es una herramienta capaz de cumplir con todos los rubros solicitados y además es una herramienta indispensable para continuar con la realización de un compilador, ya que el analizador léxico es el primer elemento del compilador en recibir el código fuente y es sobre el cual se construye el resto del compilador. En resumen, el objetivo principal del proyecto se logró: elaborar un analizador léxico en Lex/Flex que reconoce los componentes léxicos pertenecientes a las 10 clases descritas en clase. Ahora tenemos una base sólida para seguir avanzando en el desarrollo del compilador y estamos muy satisfechos con los resultados obtenidos.