

 UNIVERSIDAD PRIVADA DEL NORTE	FLEX	Fecha: 12/10/2022
--	------	-------------------

Programa de estudio	Experiencia curricular	Sesión
INGENIERÍA DE SISTEMAS COMPUTACIONALES	COMPILEDORES Y LENGUAJES DE PROGRAMACIÓN	9

I. OBJETIVO

Utilizar el programa FLEX para generar el analizador léxico de un lenguaje de programación de alto nivel.

II. MATERIALES Y MÉTODOS

2.1. Materiales

Computador y el entorno de desarrollo integrado (IDE) Dev-C++ para programar en lenguaje C/C++ (<https://sourceforge.net/projects/orwelldevcpp/>). También FLEX (será proporcionado por el profesor)

2.2. Metodología

Los estudiantes elaboran programas en FLEX para generar el analizador léxico de un lenguaje de programación de alto nivel, haciendo uso de las herramientas Dev-C++ y FLEX y siguiendo la explicación del docente en base a ejemplos propuestos.

III. REFERENCIA BIBLIOGRAFÍA

Aho, A., Lam, M., Sethi, R. y Ullman, J. (2008). Compiladores. Principios, técnicas y herramientas. España: Editorial Addison Wesley Iberoamericana.

IMPLEMENTACIÓN AUTOMÁTICA DE ANALIZADORES LÉXICOS

Ejemplo:

Utilizando FLEX, generar el analizador léxico de un LP que considera los token que se identifican en el Paso 1.

Paso 1: Identificar los tokens

- Números enteros: digit0+
- Números reales: digit0+.digit0+
- Identificador: letra(letra|digit0)*
- Cadena: "(letra|digit0|blanco)+"
- Asignador: =
- Suma: +
- Por: *
- Punto y coma: ;

Ejemplo de programa fuente:

```
Val1 = 15 + 82;
Res = Val1 * 3;
```

Paso 2: Usando un editor de texto escribir la especificación FLEX de las expresiones regulares y guardarla. Se debe guardar el archivo con extensión l. (Ejemplo, lex1.l).

```
%{
#include <stdio.h>
#include <conio.h>
#define IDENT 300
#define ENTERO 302
#define REAL 304
#define CADENA 306
%}

%option noyywrap
%option yylineno
ignora " "|\t|\n
letra [a-zA-Z]
digito [:digit:]

%%

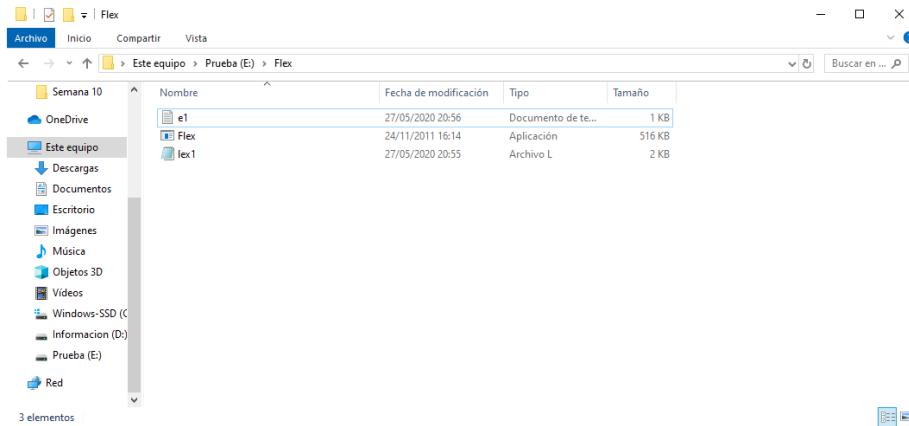
{ignora}*          {}
{digito}+          {return(ENTERO);}
{digito}+.{digito}+ {return(REAL);}
\"{letra}|{digito}| " "+\"
{letra}{letra}|{digito})*   {return(CADENA);}
"+"                {return('+');}
"*"                {return('*');}
 "("                {return('('); }
 ")"                {return(')'); }
 "="                {return('='); }
 ";"                {return(';'); }
 .                  {printf("LINEA %d: ERROR LEXICO\n",yylineno);}

%%

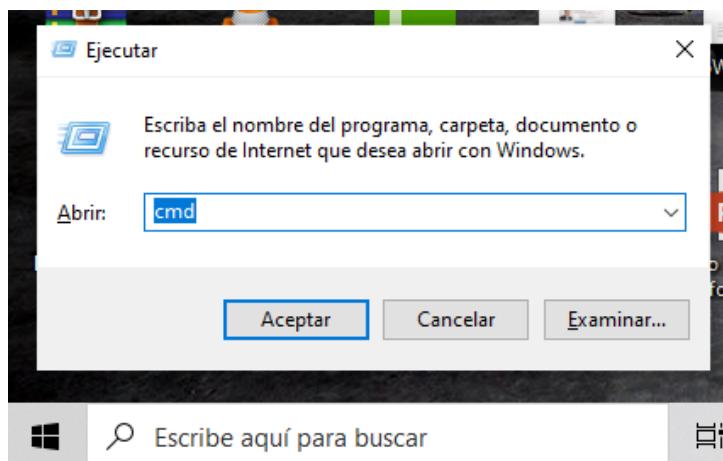
int main()
{
char NomArch[30];
int token;
printf("Ingrese archivo fuente a procesar:");
gets(NomArch);
yyin=fopen(NomArch,"rt");
if (yyin==NULL)
    printf("No se puede abrie el archivo %s",NomArch);
else
    while((token=yylex())>0)
        printf("TOKEN = %d  LEXEMA = %s \n",token,yytext);
fclose(yyin);
getch();
return(0);
}
```

Paso 3: Usando Flex, generar el código del analizador léxico.

1. Crear una carpeta en una unidad (Ejm. en la unidad E; crear la carpeta FLEX)
2. En la carpeta creada, colocar el programa Flex, el archivo con la especificación Flex (lex1.l) y el ejemplo de programa fuente (e1.txt) que se va a procesar.



3. Ejecutar el símbolo del sistema.



4. Ubicarse en la carpeta creada (FLEX) y verificar que los archivos estén en la carpeta

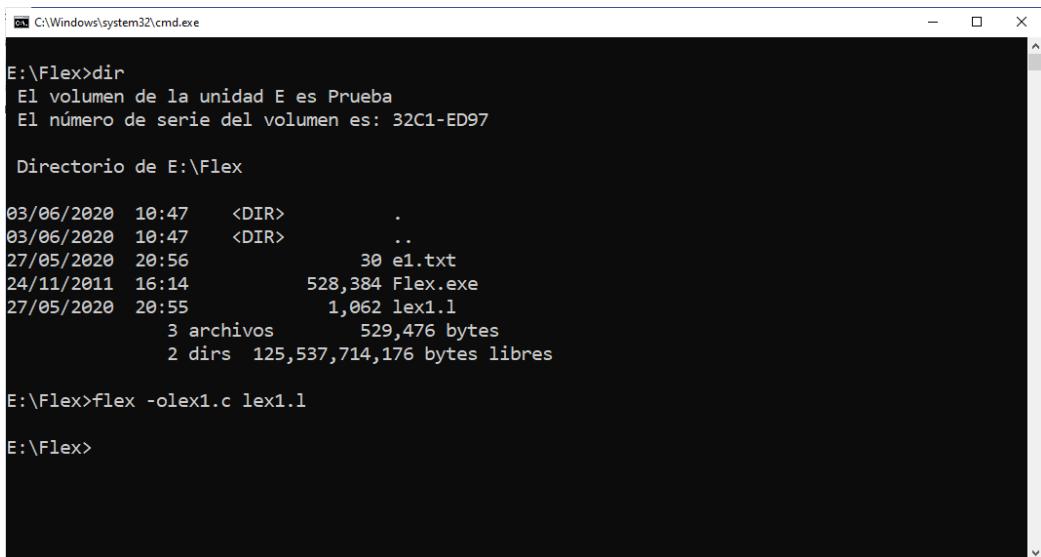
```
C:\Windows\system32\cmd.exe
E:\Flex>dir
El volumen de la unidad E es Prueba
El número de serie del volumen es: 32C1-ED97

Directorio de E:\Flex

03/06/2020 10:47    <DIR>      .
03/06/2020 10:47    <DIR>      ..
27/05/2020 20:56            30 e1.txt
24/11/2011 16:14        528,384 Flex.exe
27/05/2020 20:55        1,062 lex1.l
                  3 archivos       529,476 bytes
                  2 dirs  125,537,714,176 bytes libres

E:\Flex>
```

5. Escribir ***Flex -olex1.c lex1.l*** para generar el archivo lex1.c que contiene el analizador léxico generado a partir de la especificación flex lex1.l. (o significa output)



```
C:\Windows\system32\cmd.exe
E:\Flex>dir
El volumen de la unidad E es Prueba
El n mero de serie del volumen es: 32C1-ED97

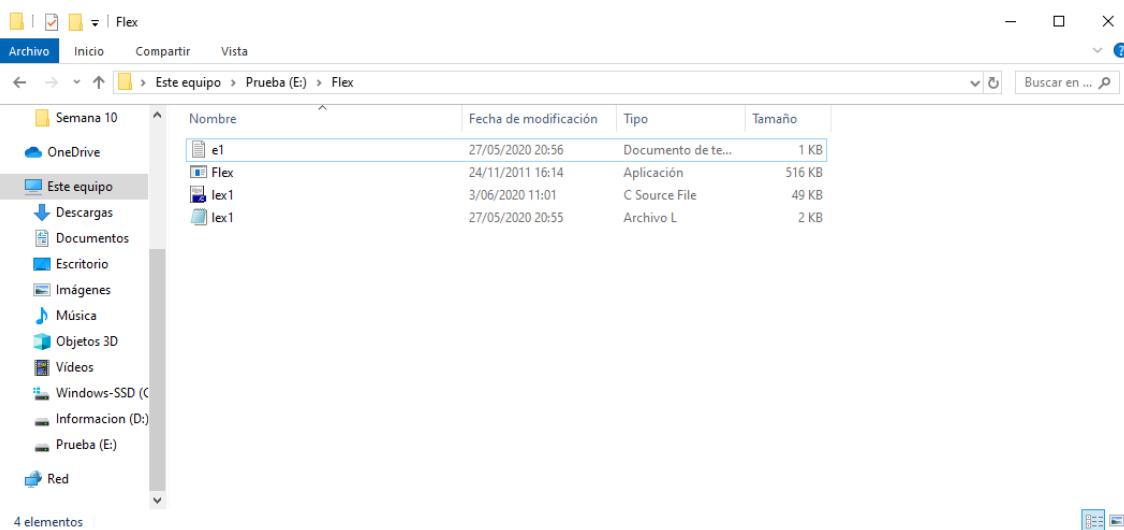
Directorio de E:\Flex

03/06/2020 10:47    <DIR>      .
03/06/2020 10:47    <DIR>      ..
27/05/2020 20:56            30 e1.txt
24/11/2011 16:14        528,384 Flex.exe
27/05/2020 20:55        1,062 lex1.l
                3 archivos       529,476 bytes
                2 dirs   125,537,714,176 bytes libres

E:\Flex>flex -olex1.c lex1.l

E:\Flex>
```

6. En la carpeta creada (FLEX) aparece generado el archivo lex1.c. Cargar dicho archivo con Dev C++ y listo para ejecutarlo.



Ejercicios propuestos:

Usando flex, generar un analizador léxico para los lenguajes de programación que consideran los siguientes tokens:

1. Tokens:

- Identificador: \$letra(letra/digito)*
- Entero: digito+
- Real : digito+.digito+
- Asignador: =
- Mas: +
- Por: *
- Menor o igual que: <=
- Menor que: <
- Cadena: <<(letra|digito|blanco)+>>

2. Tokens:

- Números enteros: digito+
- Números reales: digito+.digito+
- Identificador: letra(letra|digito)*
- Cadena: <(letra|digito|blanco)*>
- Leer: READ
- Escribir: WRITE
- Si: SI
- Mientras: WHILE
- Asignador: =
- Suma: +
- Por: *
- Mayor que: >
- Mayor o igual que: >=
- Igual a: ==
- Punto y coma: ;
- Paréntesis izq.: (
- Paréntesis der.:)

Además, se ignora los comentarios definidos por la ER: /(letra|digito|blanco)+/

	FLEX	Fecha: 19/10/2022
---	------	-------------------

Programa de estudio	Experiencia curricular	Sesión
INGENIERÍA DE SISTEMAS COMPUTACIONALES	COMPILEDORES Y LENGUAJES DE PROGRAMACIÓN	10

I. OBJETIVO

Utilizar el programa FLEX para generar el analizador léxico de un lenguaje de programación de alto nivel.

II. MATERIALES Y MÉTODOS

2.1. Materiales

Computador y el entorno de desarrollo integrado (IDE) Dev-C++ para programar en lenguaje C/C++ (<https://sourceforge.net/projects/orwelldevcpp/>). También FLEX (será proporcionado por el profesor)

2.2. Metodología

Los estudiantes elaboran programas en FLEX para generar el analizador léxico de un lenguaje de programación de alto nivel, haciendo uso de las herramientas Dev-C++ y FLEX y siguiendo la explicación del docente en base a ejemplos propuestos.

III. REFERENCIA BIBLIOGRAFÍA

Aho, A., Lam, M., Sethi, R. y Ullman, J. (2008). Compiladores. Principios, técnicas y herramientas. España: Editorial Addison Wesley Iberoamericana.

IMPLEMENTACIÓN AUTOMÁTICA DE ANALIZADORES LÉXICOS

Ejemplo:

Utilizando FLEX, generar el analizador léxico de un LP que considera los token que se identifican en el Paso 1.

Paso 1: Identificar los tokens

- Números enteros: digit0+
- Números reales: digit0+.digit0+
- Identificador: letra(letra|digit0)*
- Cadena: "(letra|digit0|blanco)+"
- Asignador: =
- Suma: +
- Por: *
- Punto y coma: ;

Ejemplo de programa fuente:

```
Val1 = 15 + 82;
Res = Val1 * 3;
```

Paso 2: Usando un editor de texto escribir la especificación FLEX de las expresiones regulares y guardarla. Se debe guardar el archivo con extensión l. (Ejemplo, lex1.l).

```
%{
#include <stdio.h>
#include <conio.h>
#define IDENT 300
#define ENTERO 302
#define REAL 304
#define CADENA 306
%}

%option noyywrap
%option yylineno
ignora " "|\t|\n
letra [a-zA-Z]
digito [:digit:]

%%

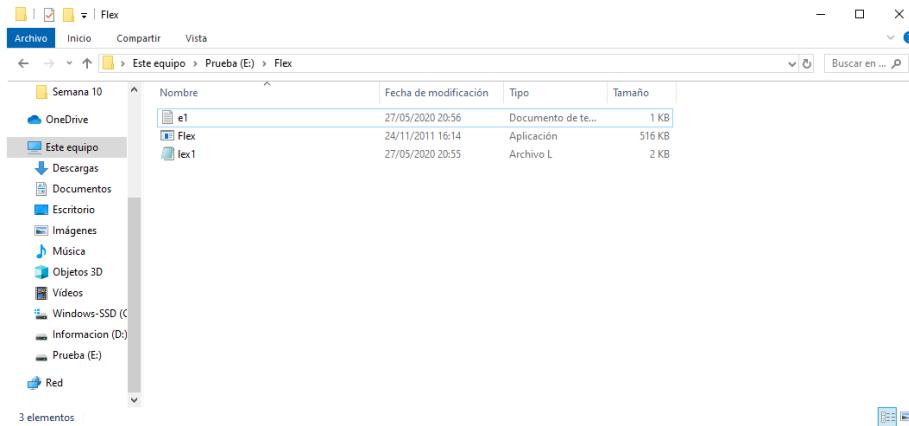
{ignora}*          {}
{digito}+          {return(ENTERO);}
{digito}+.{digito}+ {return(REAL);}
\"{letra}|{digito}| " "+\"
{letra}{letra}|{digito})*   {return(CADENA);}
"+"                {return('+');}
"*"                {return('*');}
 "("                {return('('); }
 ")"                {return(')'); }
 "="                {return('='); }
 ";"                {return(';'); }
 .                  {printf("LINEA %d: ERROR LEXICO\n",yylineno);}

%%

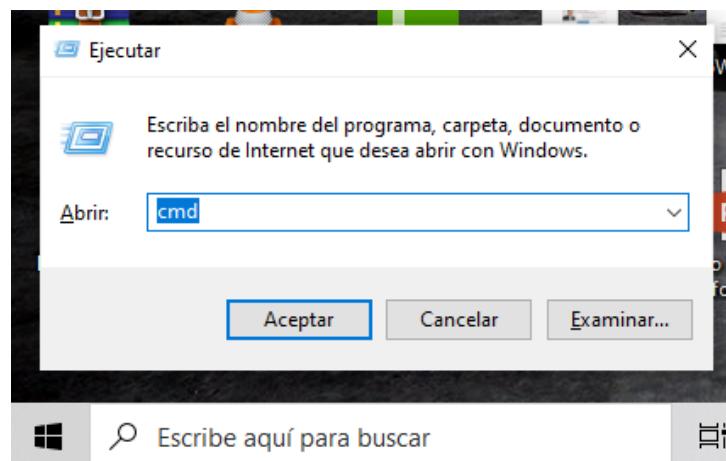
int main()
{
char NomArch[30];
int token;
printf("Ingrese archivo fuente a procesar:");
gets(NomArch);
yyin=fopen(NomArch,"rt");
if (yyin==NULL)
    printf("No se puede abrie el archivo %s",NomArch);
else
    while((token=yylex())>0)
        printf("TOKEN = %d  LEXEMA = %s \n",token,yytext);
fclose(yyin);
getch();
return(0);
}
```

Paso 3: Usando Flex, generar el código del analizador léxico.

1. Crear una carpeta en una unidad (Ejm. en la unidad E; crear la carpeta FLEX)
2. En la carpeta creada, colocar el programa Flex, el archivo con la especificación Flex (lex1.l) y el ejemplo de programa fuente (e1.txt) que se va a procesar.



3. Ejecutar el símbolo del sistema.



4. Ubicarse en la carpeta creada (FLEX) y verificar que los archivos estén en la carpeta

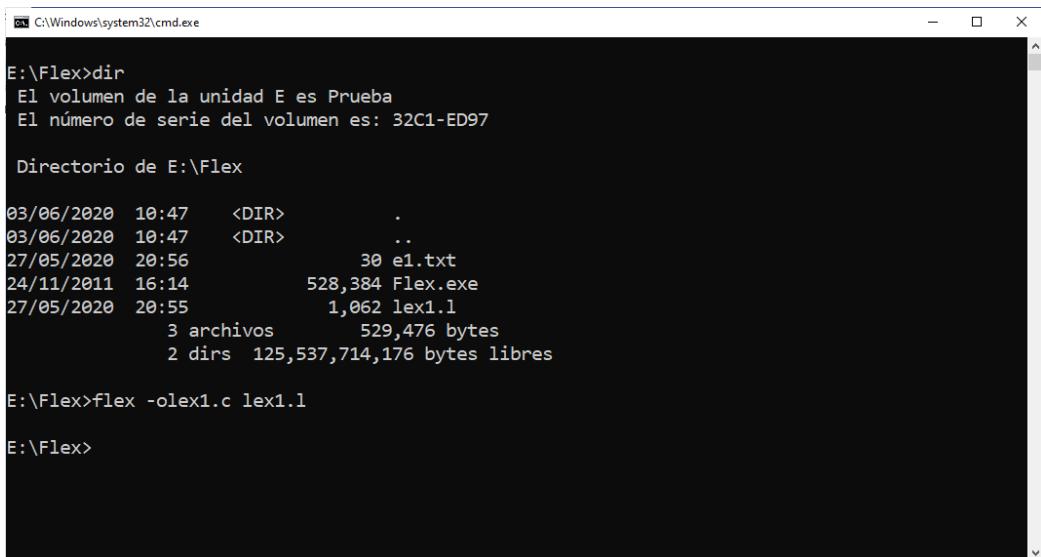
```
C:\Windows\system32\cmd.exe
E:\Flex>dir
El volumen de la unidad E es Prueba
El número de serie del volumen es: 32C1-ED97

Directorio de E:\Flex

03/06/2020 10:47    <DIR>      .
03/06/2020 10:47    <DIR>      ..
27/05/2020 20:56            30 e1.txt
24/11/2011 16:14        528,384 Flex.exe
27/05/2020 20:55       1,062 lex1.l
                  3 archivos      529,476 bytes
                  2 dirs  125,537,714,176 bytes libres

E:\Flex>
```

5. Escribir ***Flex -olex1.c lex1.l*** para generar el archivo lex1.c que contiene el analizador léxico generado a partir de la especificación flex lex1.l. (o significa output)



```
C:\Windows\system32\cmd.exe
E:\Flex>dir
El volumen de la unidad E es Prueba
El n mero de serie del volumen es: 32C1-ED97

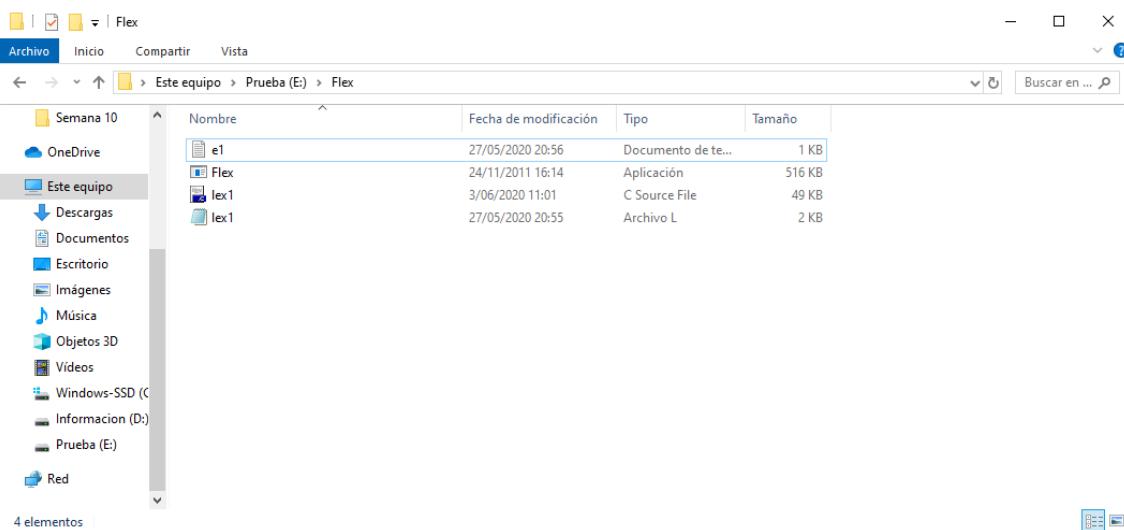
Directorio de E:\Flex

03/06/2020 10:47    <DIR>      .
03/06/2020 10:47    <DIR>      ..
27/05/2020 20:56            30 e1.txt
24/11/2011 16:14        528,384 Flex.exe
27/05/2020 20:55        1,062 lex1.l
                3 archivos       529,476 bytes
                2 dirs   125,537,714,176 bytes libres

E:\Flex>flex -olex1.c lex1.l

E:\Flex>
```

6. En la carpeta creada (FLEX) aparece generado el archivo lex1.c. Cargar dicho archivo con Dev C++ y listo para ejecutarlo.



Ejercicios propuestos:

Usando flex, generar un analizador léxico para los lenguajes de programación que consideran los siguientes tokens:

1. Tokens:

- Identificador: \$letra(letra/digito)*
- Entero: digito+
- Real : digito+.digito+
- Asignador: =
- Mas: +
- Por: *
- Menor o igual que: <=
- Menor que: <
- Cadena: <<(letra|digito|blanco)+>>

2. Tokens:

- Números enteros: digito+
- Números reales: digito+.digito+
- Identificador: letra(letra|digito)*
- Cadena: <(letra|digito|blanco)*>
- Leer: READ
- Escribir: WRITE
- Si: SI
- Mientras: WHILE
- Asignador: =
- Suma: +
- Por: *
- Mayor que: >
- Mayor o igual que: >=
- Igual a: ==
- Punto y coma: ;
- Paréntesis izq.: (
- Paréntesis der.:)

Además, se ignora los comentarios definidos por la ER: /(letra|digito|blanco)+/

Programa de estudio	Experiencia curricular	Sesión
INGENIERÍA DE SISTEMAS COMPUTACIONALES	COMPILEDORES Y LENGUAJES DE PROGRAMACIÓN	11

I. OBJETIVO

Utilizar el programa BISON para generar el analizador sintáctico de un lenguaje de programación de alto nivel.

II. MATERIALES Y MÉTODOS

2.1. Materiales

Computador y el entorno de desarrollo integrado (IDE) Dev-C++ para programar en lenguaje C/C++ (<https://sourceforge.net/projects/orweldevcpp/>). También BISON (será proporcionado por el profesor)

2.2. Metodología

Los estudiantes elaboran programas en BISON para generar el analizador sintáctico de un lenguaje de programación de alto nivel, haciendo uso de las herramientas Dev-C++ y BISON y siguiendo la explicación del docente en base a ejemplos propuestos.

III. REFERENCIA BIBLIOGRAFÍA

Aho, A., Lam, M., Sethi, R. y Ullman, J. (2008). Compiladores. Principios, técnicas y herramientas. España: Editorial Addison Wesley Iberoamericana.

INTRODUCCIÓN A BISON

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción para una gramática independiente del contexto (en realidad de una subclase de éstas, las LALR) en un programa en C que analiza esa gramática. Es compatible al 100% con Yacc, una herramienta clásica de Unix para la generación de analizadores léxicos, pero es un desarrollo diferente realizado por GNU bajo licencia GPL. Todas las gramáticas escritas apropiadamente para Yacc deberían funcionar con Bison sin ningún cambio. Usándolo junto a Flex esta herramienta permite construir compiladores de lenguajes.

Un fuente de Bison (normalmente un fichero con extensión .y) describe una gramática. El ejecutable que se genera indica si un fichero de entrada dado pertenece o no al lenguaje generado por esa gramática. La forma general de una gramática de Bison es la siguiente:

```
%{
declaraciones en C
%
Declaraciones de Bison
%%
Reglas gramaticales
%%
Código C adicional
```

Los '%%', '%{' y '%}' son signos de puntuación que aparecen en todo archivo de gramática de Bison para separar las secciones.

Las declaraciones en C pueden definir tipos y variables utilizadas en las acciones. Puede también usar comandos del preprocesador para definir macros que se utilicen ahí, y utilizar #include para incluir archivos de cabecera que realicen cualquiera de estas cosas.

Las declaraciones de Bison declaran los nombres de los símbolos terminales y no terminales, y también podrían describir la precedencia de operadores y los tipos de datos de los valores semánticos de varios símbolos.

Las reglas gramaticales son las **producciones** de la gramática, que además pueden llevar asociadas acciones, código en C, que se ejecutan cuando el analizador encuentra las reglas correspondientes.

El código C adicional puede contener cualquier código C que deseé utilizar. A menudo suele ir la definición del analizador léxico yylex, más subrutinas invocadas por las acciones en las reglas gramaticales. En un programa simple, todo el resto del programa puede ir aquí.

Símbolos, terminales y no terminales

Los **símbolos terminales** de la gramática se denominan en Bison **tokens** y deben declararse en la sección de definiciones. Por convención se suelen escribir los tokens en mayúsculas y los **símbolos no terminales** en minúsculas.

Los nombres de los símbolos pueden contener letras, dígitos (no al principio), subrayados y puntos. Los puntos tienen sentido únicamente en no-terminales.

Hay tres maneras de escribir símbolos terminales en la gramática. Aquí se describen las dos más usuales:

- Un **token declarado** se escribe con un identificador, de la misma manera que un identificador en C. Por convención, debería estar todo en mayúsculas. Cada uno de estos nombres debe definirse con una declaración de %token.
- Un **token de carácter** se escribe en la gramática utilizando la misma sintaxis usada en C para las constantes de un carácter; por ejemplo, '+' es un tipo de token de carácter. Un tipo de token de carácter no necesita ser declarado a menos que necesite especificar el tipo de datos de su valor semántico, asociatividad, o precedencia. Por convención, un token de carácter se utiliza únicamente para representar un token consistente en ese carácter en particular.

Sintaxis de las reglas gramaticales (producciones)

Una regla gramatical de Bison tiene la siguiente forma general:

resultado: componentes...
;

donde *resultado* es el símbolo no terminal que describe esta regla y *componentes* son los diversos símbolos terminales y no terminales que están reunidos por esta regla. Por ejemplo,

exp: exp '+' exp
;

dice que dos agrupaciones de tipo exp, con un token '+' en medio, puede combinarse en una agrupación mayor de tipo exp.

Los espacios en blanco en las reglas son significativos únicamente para separar símbolos. Puede añadir tantos espacios en blanco extra como desee.

Distribuidas en medio de los componentes pueden haber *acciones* que determinan la semántica de la regla. Una acción tiene el siguiente aspecto:

{sentencias en C}

Normalmente hay una única acción que sigue a los componentes.

Se pueden escribir por separado varias reglas para el mismo *resultado* o pueden unirse con el carácter de barra vertical ‘|’ así:

```
Resultado : componentes-regla1...
           | componentes-regla2...
           ...
           ;
```

Estas aún se consideran reglas distintas incluso cuando se unen de esa manera. Si los *componentes* en una regla están vacíos, significa que *resultado* puede concordar con la cadena vacía (en notación formal sería ϵ). Por ejemplo, aquí aparece cómo definir una secuencia separada por comas de cero o más agrupaciones exp:

```
expseq: /* vacío */
       | expseq1
       ;
expseq1: exp
       | expseq1 ',' exp
       ;
```

Es habitual escribir el comentario ‘/* vacío */’ en cada regla sin componentes.

Ejemplo:

Considerando la siguiente gramática de libre contexto generar el analizador sintáctico usando BISON. Y además implementar el analizador léxico.

GLC

```
Asignacion   : IDENT '=' Expresion ';'
              | IDENT '=' Expresion ';' Asignacion
              ;
Expresion    : Expresion '+' Termino
              | Termino
              ;
Termino      : Termino '*' Factor
              | Factor
              ;
Factor        : ENTERO
              | IDENT
              | '('Expresion')'
              ;
```

Tokens compuestos:

Identificador: letra(letra|digito)*

Entero: digito+

Tokens simples:

Asignador: =

Suma: +

Por: *

Paréntesis izquierdo: (

Paréntesis derecho:)

Punto y coma: ;

SOLUCION:

Paso 1: Escribir el programa en BISON. Usando un editor de texto escribir la especificación BISON de la GLC. Incluir el programa en C del analizador léxico. Se debe guardar el archivo con extensión y. (Ejemplo, parser.y)

```
%{  
#include <stdio.h>  
#include <string.h>  
#include <ctype.h>  
  
int yystopparser=0;  
%}  
  
%token IDENT ENTERO  
%start Asignacion  
  
%%  
  
/*GRAMATICA DE LIBRE CONTEXTO*/  
  
Asignacion : IDENT '=' Expresion ';' | IDENT '!=' Expresion ';' Asignacion ;  
Expresion : Expresion '+' Termino | Termino ;  
Termino : Termino '*' Factor | Factor ;  
Factor : ENTERO | IDENT | '('Expresion')' ;  
%%
```

/*ANALIZADOR LEXICO*/

```
FILE *Fuente;  
int error=0;  
int yylex(void)  
{  
int c;  
c=fgetc(Fuente);  
  
while (isspace(c))  
    c=fgetc(Fuente);  
  
if (isdigit(c))  
{  
    c=fgetc(Fuente);  
    while (isdigit(c))  
        c=fgetc(Fuente);  
    ungetc(c,Fuente);
```

```

    return(ENTERO);
}

if (isalpha(c))
{
    c=fgetc(Fuente);
    while(isalpha(c) || isdigit(c))
        c=fgetc(Fuente);
    ungetc(c,Fuente);
    return(IDENT);
}

if (strchr("=/*+;()",c))
    return(c);
if (c==EOF)
    return(0);
yyerror("Error Lexico");

}

/*MANEJADOR DE ERRORES*/

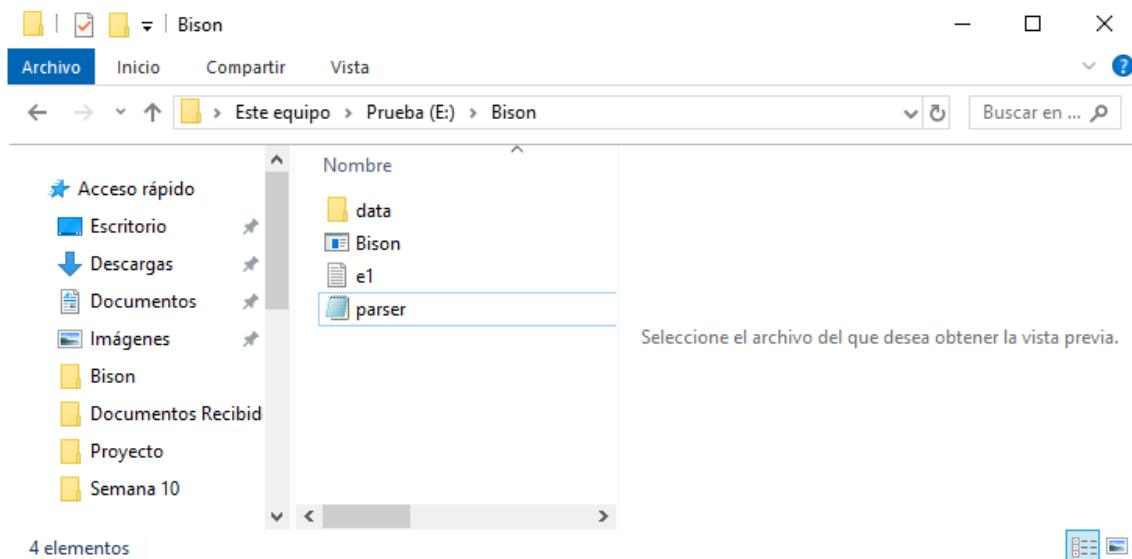
yyerror(char *s)
{
printf("%s\n",s);
error=1;
}

/*FUNCION PRINCIPAL*/
int main(void)
{
char NomArch[30];
printf("Ingrese archivo a procesar:");
gets(NomArch);
Fuente=fopen(NomArch,"rt");
if (Fuente==NULL)
    printf("Error: Archivo no se puede abrir");
else
    yyparse();
if (error==0)
    printf("Programa OK");
fclose(Fuente);
getch();
}

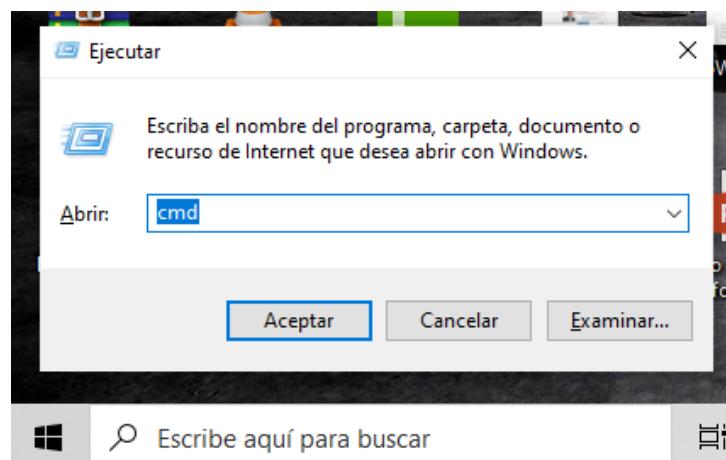
```

Paso 3: Usando BISON, generar el código del analizador sintáctico.

1. Crear una carpeta en una unidad (Ejm. en la unidad E:, crear la carpeta BISON)
2. En la carpeta creada, colocar el programa BISON, el archivo con la especificación BISON (parser.y) y el ejemplo de programa fuente (e1.txt) que se va a procesar.



3. Ejecutar el símbolo del sistema.



4. Ubicarse en la carpeta creada (BISON) y verificar que los archivos estén en la carpeta, usando el comando **dir**

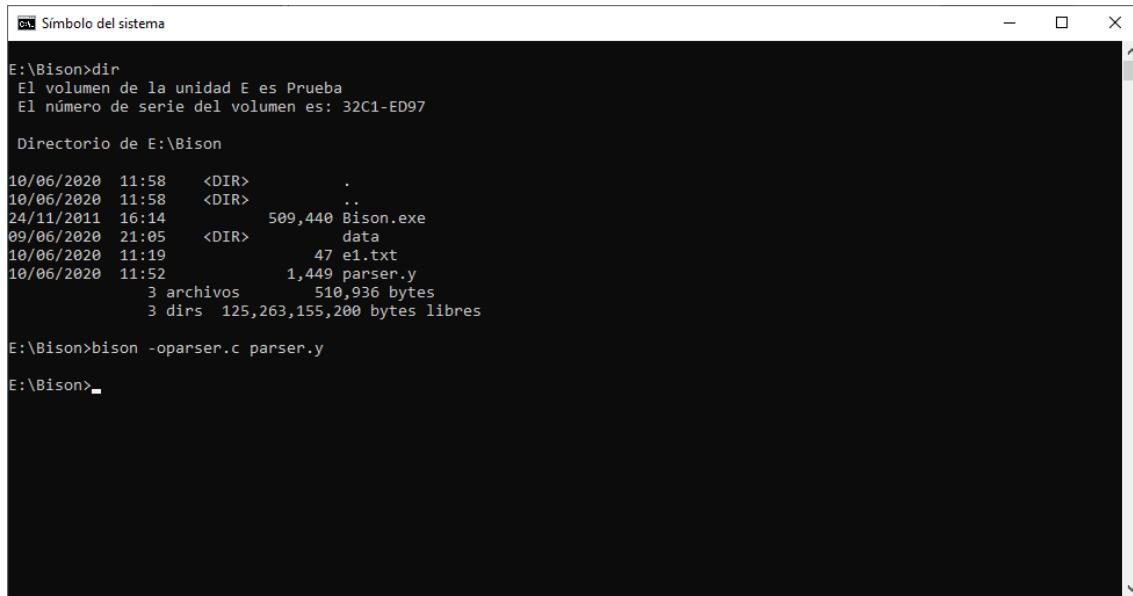
```
E:\Bison>dir
El volumen de la unidad E es Prueba
El número de serie del volumen es: 32C1-ED97

Directorio de E:\Bison

10/06/2020 12:01    <DIR>      .
10/06/2020 12:01    <DIR>      ..
24/11/2011 16:14  509,440 Bison.exe
09/06/2020 21:05    <DIR>      data
10/06/2020 11:19        47 e1.txt
10/06/2020 12:01     47,224 parser.c
10/06/2020 11:52       1,449 parser.y
                           4 archivos   558,160 bytes
                           3 dirs  125,263,106,048 bytes libres

E:\Bison>
```

- Escribir **Bison -oparser.c parser.y** para generar el archivo parser.c que contiene el analizador sintáctico generado a partir de la especificación BISON parser.y (o significa output)



```

Símbolo del sistema

E:\Bison>dir
El volumen de la unidad E es Prueba
El número de serie del volumen es: 32C1-ED97

Directorio de E:\Bison

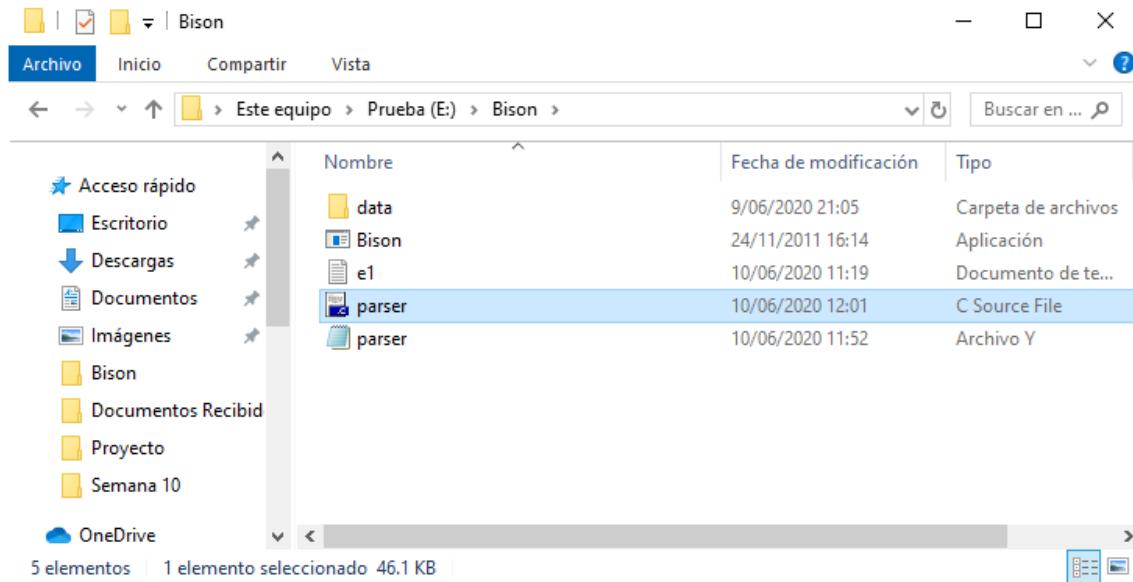
10/06/2020 11:58    <DIR>      .
10/06/2020 11:58    <DIR>      ..
24/11/2011 16:14    509,440 Bison.exe
09/06/2020 21:05    <DIR>      data
10/06/2020 11:19        47 e1.txt
10/06/2020 11:52    1,449 parser.y
            3 archivos     510,936 bytes
            3 dirs   125,263,155,200 bytes libres

E:\Bison>bison -oparser.c parser.y

E:\Bison>_

```

- En la carpeta creada (BISON) aparece generado el archivo parser.c. Cargar dicho archivo con Dev C++ y listo para ejecutarlo.



Ejercicios propuestos:

Usando BISON, generar un analizador sintáctico para los siguientes ejercicios:

- Modificar el ejemplo, con lo siguiente: (está resaltado de color verde lo agregado)

GLC

Asignacion	:	IDENT '=' Expresion ';'
		IDENT '=' Expresion ';' Asignacion
	;	
Expresion	:	Expresion '+' Termino
		Expresion '-' Termino

```

    | Termino
    ;
Termino   : Termino '*' Factor
    | Termino '/' Factor
    | Factor
    ;
Factor    : ENTERO
    | IDENT
    | REAL
    | '('Expresion')'
    ;

```

Tokens compuestos:

Identificador: letra(letra|digito)*

Entero: digito+

Real: digito+.digito+

Tokens simples:

Asignador: =

Suma: +

Por: *

Paréntesis izquierdo: (

Paréntesis derecho:)

Punto y coma: ;

Menos: -

Entre: /

2. Lo siguiente:

GLC	
Programa	: Sentencia
	Programa Sentencia
	;
Sentencia	: Asignacion
	Leer
	Escribir
	;
Asignacion	: IDENT '=' Expresion ';'
	IDENT '=' Expresion ';' Asignacion
	;
Expresion	: Expresion '+' Termino
	Termino
	;
Termino	: Termino '*' Factor
	Factor
	;
Factor	: ENTERO
	IDENT
	'('Expresion')'
	;
Leer	: READ '(' IDENT ')';
	;

Escribir : WRITE '(' Expresion ')'
;

Tokens compuestos:

Identificador: letra(letra|digito)*

Entero: digito+

Leer: READ

Escribir: WRITE

Tokens simples:

Asignador: =

Suma: +

Por: *

Paréntesis izquierdo: (

Paréntesis derecho:)

Punto y coma: ;

Programa de estudio	Experiencia curricular	Sesión
INGENIERÍA DE SISTEMAS COMPUTACIONALES	COMPILADORES Y LENGUAJES DE PROGRAMACIÓN	13

I. OBJETIVO

Utilizar el programa FLEX y BISON para generar el analizador léxico-sintáctico de un lenguaje de programación de alto nivel.

II. MATERIALES Y MÉTODOS

2.1. Materiales

Computador y el entorno de desarrollo integrado (IDE) Dev-C++ para programar en lenguaje C/C++ (<https://sourceforge.net/projects/orweldevcpp/>). También FLEX y BISON (serán proporcionado por el profesor)

2.2. Metodología

Los estudiantes elaboran programas en FLEX y BISON para generar el analizador léxico-sintáctico de un lenguaje de programación de alto nivel, haciendo uso de las herramientas Dev-C++, FLEX y BISON y siguiendo la explicación del docente en base a ejemplos propuestos.

III. REFERENCIA BIBLIOGRAFÍA

Aho, A., Lam, M., Sethi, R. y Ullman, J. (2008). Compiladores. Principios, técnicas y herramientas. España: Editorial Addison Wesley Iberoamericana.

ANALIZADOR LEXICO-SINTÁCTICO USANDO FLEX Y BISON

Ejemplo:

Considerando la siguiente gramática de libre contexto generar el analizador léxico-sintáctico usando FLEX y BISON.

GLC

```

Asignacion   : IDENT '=' Expresion ';'
              | IDENT '=' Expresion ';' Asignacion
              ;
Expresion    : Expresion '+' Termino
              | Termino
              ;
Termino      : Termino '*' Factor
              | Factor
              ;
Factor        : ENTERO
              | IDENT
              | '('Expresion ')'
              ;

```

Tokens compuestos:

Identificador: letra(letra|digito)*

Entero: digito+

Tokens simples:

Asignador: =

Suma: +

Por: *

Paréntesis izquierdo: (

Paréntesis derecho:)

Punto y coma: ;

SOLUCION:

Paso 1: Crear una carpeta en una unidad (Ejm. en la unidad E:, crear la carpeta COMPI). En estarán todos los archivos.

Paso 2: Escribir el programa en BISON. Usando un editor de texto escribir la especificación BISON de la GLC. En la carpeta COMPI se debe guardar el archivo con extensión y. (Ejemplo, parser.y)

```
%{  
int yystopparser=0;  
%}  
  
%token IDENT ENTERO  
%start Asignacion  
  
%%
```

/*GRAMATICA DE LIBRE CONTEXTO*/

```
Asignacion : IDENT '=' Expresion ';' | IDENT '=' Expresion ';' Asignacion ;  
Expresion : Expresion '+' Termino | Termino ;  
Termino : Termino '*' Factor | Factor ;  
Factor : ENTERO | IDENT | '('Expresion')' ;
```

Paso 3: Escribir el programa en FLEX. Usando un editor de texto escribir la especificación FLEX de las expresiones regulares de los tokens del lenguaje. En la carpeta COMPI se debe guardar el archivo con extensión l. (Ejemplo, lexico.l).

```

%{
#include <stdio.h>
#include <conio.h>

#include "e:\compi\parser.h"
%}

%option noyywrap
%option yylineno

ignora " "|\t|\n
letra [A-Za-z]
dígito [0-9]

%%

{ignora}*
{dígito}+
{letra}({letra}|{dígito})*
"="
"+"
"**"
 "("
 ")"
";"
.

; }

{return ENTERO;}
{return IDENT;}
{return ('=');}
{return ('+');}
{return ('*');}
{return ('(');}
{return (')');}
{return (';');}
{printf("En linea %d: ERROR: Carácter ilegal\n", yylineno);}

%%

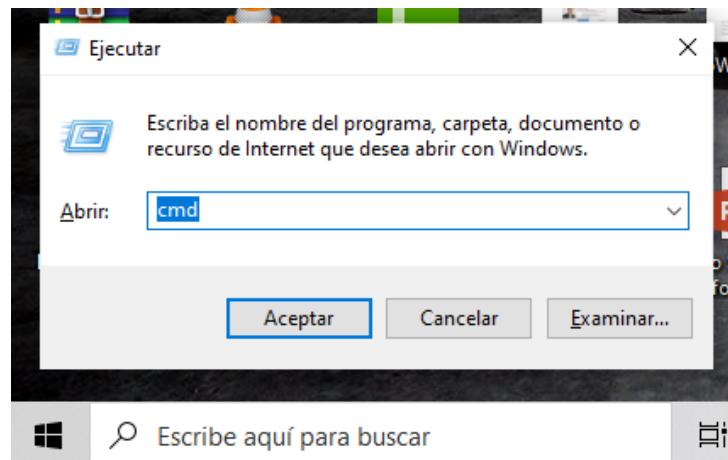
yyerror(char * msg)
{
    printf ("%s\n", msg);
}

/*FUNCION PRINCIPAL*/
int error=0;
int main(void)
{
    char NomArch[30];
    printf("Ingrese archivo a procesar:");
    gets(NomArch);
    yyin=fopen(NomArch,"rt");
    if (yyin==NULL)
        printf("Error: Archivo no se puede abrir");
    else
        yyparse();
    if (error==0)
        printf("Programa OK");
    fclose(yyin);
    return(0);
}

```

Paso 4: Usando Flex y Bison, generar el analizador léxico y el analizador sintáctico

1. Ejecutar el símbolo del sistema.



2. Ubicarse en la carpeta creada (COMPI) y verificar que los archivos estén en la carpeta, usando el comando **dir**

```
E:\Compi>dir
El volumen de la unidad E es Prueba
El n mero de serie del volumen es: 32C1-ED97

Directorio de E:\Compi

17/06/2020 16:34    <DIR>      .
17/06/2020 16:34    <DIR>      ..
24/11/2011 16:14    509,440 Bison.exe
17/06/2020 15:58    <DIR>      data
24/11/2011 16:14    528,384 Flex.exe
17/06/2020 16:32            887 lexico.l
21/11/2016 21:17            15 p.txt
17/06/2020 16:08            398 parser.y
                           5 archivos     1,039,124 bytes
                           3 dirs   125,258,153,984 bytes libres

E:\Compi>
```

3. Escribir en el símbolo del sistema:

Flex -olexico.c lexico.l para generar el archivo lexico.c que contiene el analizador léxico generado a partir de la especificación flex lexico.l. (o significa output)
Luego,

Bison -d -oparser.c parser.y para generar el archivo parser.c que contiene el analizador sintáctico y el archivo parser.h que contiene los c digos num ricos de los tokens compuestos, ambos generados a partir de la especificaci n BISON parser.y (d es para generar parser.h)

```

Símbolo del sistema
E:\Compi>dir
El volumen de la unidad E es Prueba
El número de serie del volumen es: 32C1-ED97

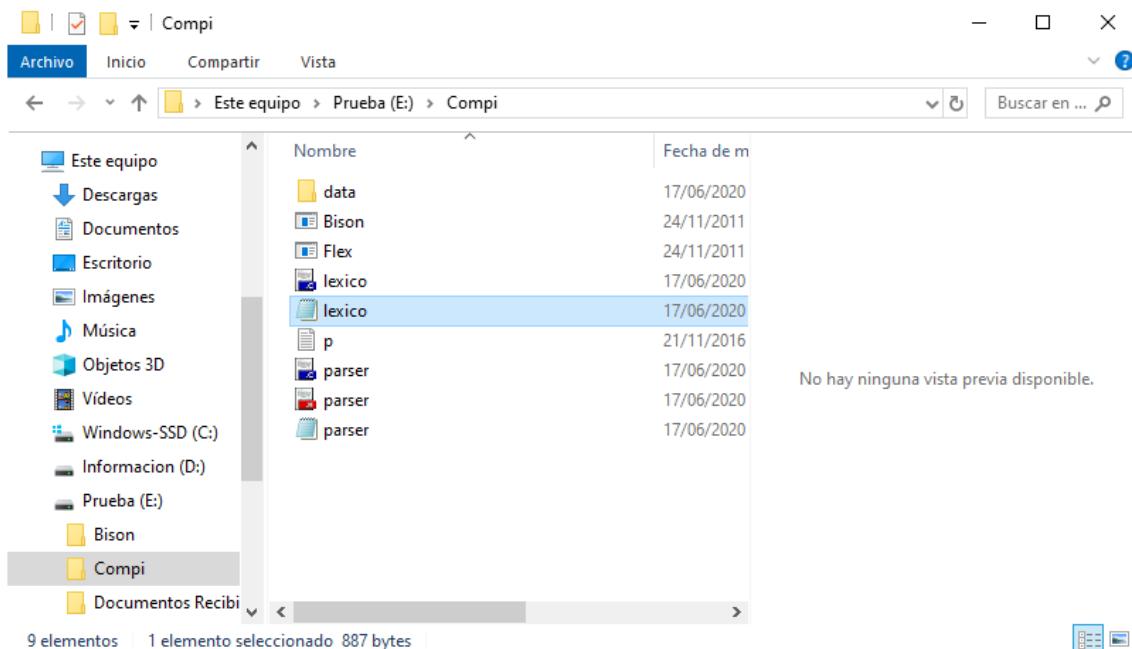
Directorio de E:\Compi

17/06/2020 16:34    <DIR>      .
17/06/2020 16:34    <DIR>      ..
24/11/2011 16:14    509,440 Bison.exe
17/06/2020 15:58    <DIR>      data
24/11/2011 16:14    528,384 Flex.exe
17/06/2020 16:32    887 lexico.l
21/11/2016 21:17    15 p.txt
17/06/2020 16:08    398 parser.y
              5 archivos     1,039,124 bytes
              3 dirs   125,258,153,984 bytes libres

E:\Compi>flex -olexico.c lexico.l
E:\Compi>bison -d -oparser.c parser.y
E:\Compi>_

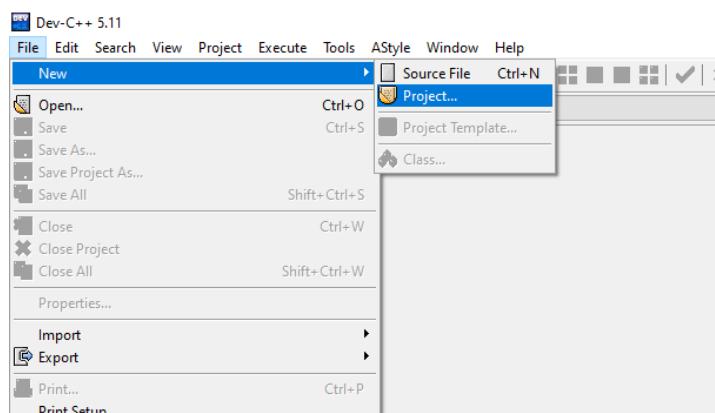
```

4. En la carpeta creada (COMPI) aparece generado el archivo léxico.c, parser.c y parser.h.

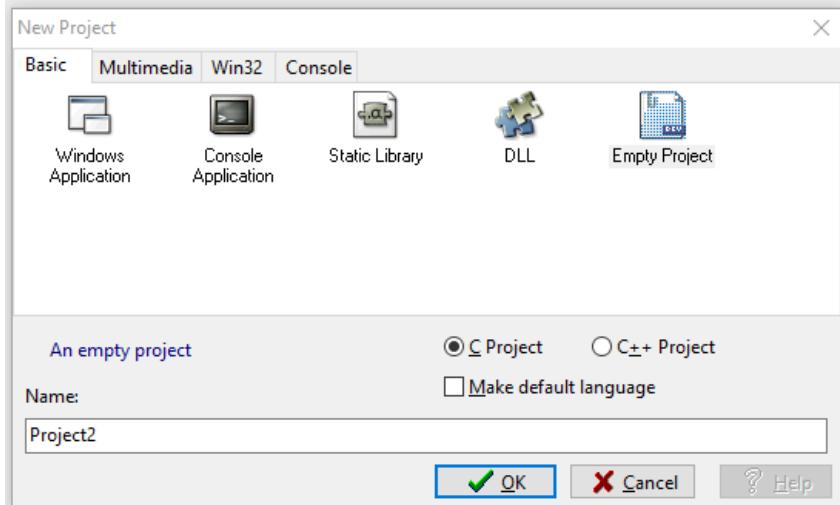


Paso 5: Generar el analizador léxico sintáctico.

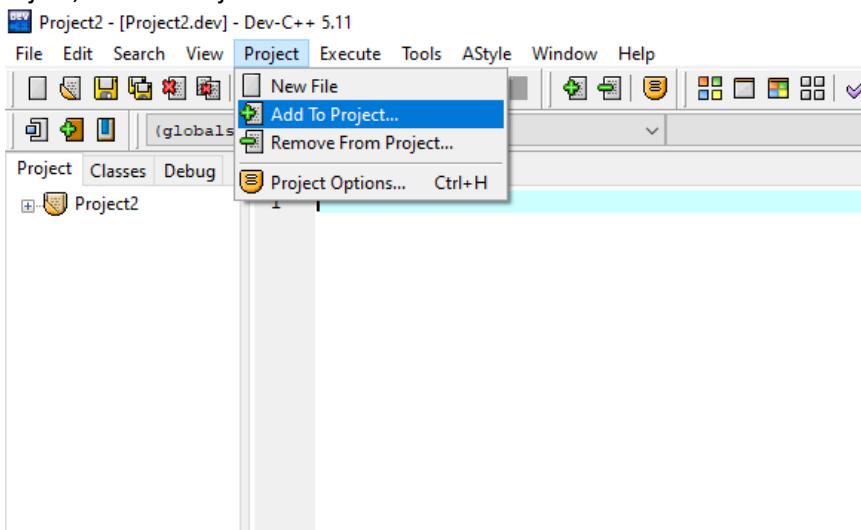
1. Cargar el programa DEV C++ y elegir New, Project



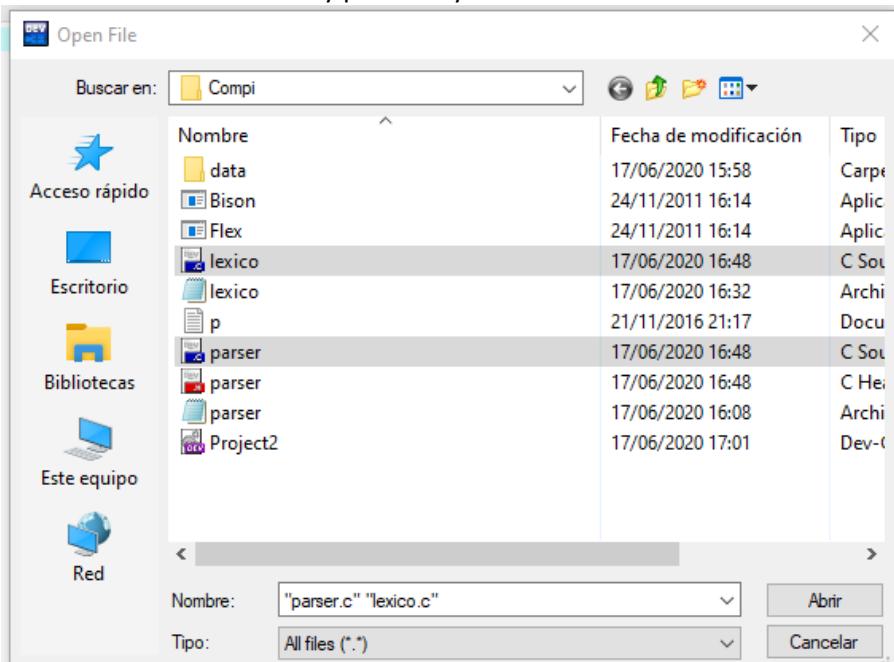
2. Elegir Empty Project, C Project, dar un nombre al proyecto y clic en OK



3. Elegir Project, Add To Project



4. Seleccionar los archivos lexico.c y parser. c y clic en Abrir



- Ejecutar el proyecto y listo.

Ejercicios propuestos:

Usando FLEX y BISON, generar un analizador léxico-sintáctico para los siguientes ejercicios:

- Modificar el ejemplo, con lo siguiente: (está resaltado de color verde lo agregado)

```
GLC
Asignacion   : IDENT '=' Expresion ';'
               | IDENT '=' Expresion ';' Asignacion
               ;
Expresion     : Expresion '+' Termino
               | Expresion '-' Termino
               |
               | Termino
               ;
Termino       : Termino '*' Factor
               | Termino '/' Factor
               | Factor
               ;
Factor         : ENTERO
               | IDENT
               | REAL
               | '('Expresion')'
               ;
```

Tokens compuestos:

Identificador: letra(letra|dígito)*

Entero: dígito+

Real: dígito+.dígito+

Tokens simples:

Asignador: =

Suma: +

Por: *

Paréntesis izquierdo: (

Paréntesis derecho:)

Punto y coma: ;

Menos: -

Entre: /

- Lo siguiente:

```
GLC
Programa      : Sentencia
               | Programa Sentencia
               ;
Sentencia     : Asignacion
               | Leer
               | Escribir
               ;
```

```

Asignacion   : IDENT '=' Expresion ';'
               | IDENT '=' Expresion ';' Asignacion
               ;
Expresion     : Expresion '+' Termino
               | Termino
               ;
Termino       : Termino '*' Factor
               | Factor
               ;
Factor         : ENTERO
               | IDENT
               | '('Expresion')'
               ;
Leer          : READ '(' IDENT ')';
               ;
Escribir : WRITE '(' Expresion ')'
               ;

```

Tokens compuestos:

Identificador: letra(letra|digito)*

Entero: digito+

Leer: READ

Escribir: WRITE

Tokens simples:

Asignador: =

Suma: +

Por: *

Paréntesis izquierdo: (

Paréntesis derecho:)

Punto y coma: ;

Programa de estudio	Experiencia curricular	Sesión
INGENIERÍA DE SISTEMAS COMPUTACIONALES	COMPILADORES Y LENGUAJES DE PROGRAMACIÓN	14

I. OBJETIVO

Rediseñar una GLC que presenta conflictos en la tabla de análisis sintáctico LR(1).

II. MATERIALES Y MÉTODOS

2.1. Materiales

Computador y el entorno de desarrollo integrado (IDE) Dev-C++ para programar en lenguaje C/C++ (<https://sourceforge.net/projects/orwelldevcpp/>). También BISON (serán proporcionado por el profesor)

2.2. Metodología

Los estudiantes identificarán los conflictos de una GLC, mediante el uso del programa BISON a fin de rediseñar la GLC, siguiendo la explicación del docente en base a ejemplos propuestos.

III. REFERENCIA BIBLIOGRAFÍA

Aho, A., Lam, M., Sethi, R. y Ullman, J. (2008). Compiladores. Principios, técnicas y herramientas. España: Editorial Addison Wesley Iberoamericana.

REDISEÑO DE UNA GLC QUE PRESENTA CONFLICTOS EN LA TABLA DE ANÁLISIS SINTÁCTICO LR

Cuando se genera el analizador sintáctico usando BISON, si la GLC presenta conflictos, estos son indicados por el BISON. Si ese es el caso, primero se identifican los conflictos y segundo se rediseña la GLC.

Ejemplo:

Rediseñar la siguiente GLC que presenta conflictos en la tabla de análisis sintáctico LR.

GLC

```
%token ENTERO IDENT GOTO
%start programa
```

```
%%
programa   : sentencia
           | programa sentencia
           ;
sentencia  : sentGoto
           | expresion
           ;
```

```

sentGoto    : GOTO etiqueta
;
etiqueta    : IDENT
;
expresion   : expresion '+' expresion
| ENTERO
| IDENT
;

```

SOLUCION:

Paso 1: Crear una carpeta en una unidad (Ejm. en la unidad E; crear la carpeta COMPI). En donde estarán todos los archivos.

Paso 2: Escribir el programa en BISON. Usando un editor de texto escribir la especificación BISON de la GLC. En la carpeta COMPI se debe guardar el archivo con extensión y. (Ejemplo, confli1.y)

```

%{
int yystopparser=0;
%}

%token ENTERO IDENT GOTO
%start programa

%%

/*GRAMATICA DE LIBRE CONTEXTO*/

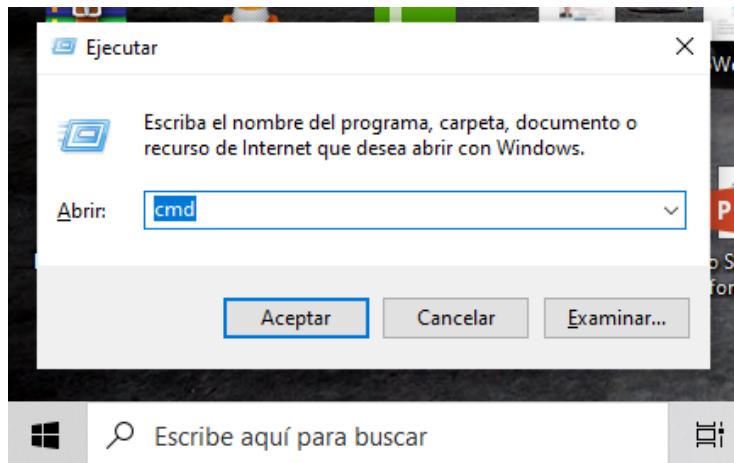
```

```

programa    : sentencia
| programa sentencia
;
sentencia   : sentGoto
| expresion
;
sentGoto    : GOTO etiqueta
;
etiqueta    : IDENT
;
expresion   : expresion '+' expresion
| ENTERO
| IDENT
;
```

Paso 3: Usando Bison, identificar las reglas que generan los conflictos

1. Ejecutar el símbolo del sistema.



2. Ubicarse en la carpeta creada (COMPI) y verificar que los archivos estén en la carpeta, usando el comando **dir**

```
E:\Compi>dir
El volumen de la unidad E es Prueba
El n mero de serie del volumen es: 32C1-ED97

Directorio de E:\Compi

08/07/2020 16:30    <DIR>      .
08/07/2020 16:30    <DIR>      ..
24/11/2011 16:14      509,440 Bison.exe
17/02/2003 16:52          625 Confl1.Y
08/07/2020 16:29    <DIR>      data
                  2 archivos      510,065 bytes
                  3 dirs   125,225,115,648 bytes libres

E:\Compi>
```

3. Escribir en el s mbolo del sistema:

Bison -v -oconfli1.c confli1.y para generar el archivo confli1.c que contiene el analizador sint ctico y el archivo confli1.output que contiene la tabla de an lisis sint ctico LR e indica las reglas que generan los conflictos.
(v es para generar confli1.output)

```
E:\Compi>dir
El volumen de la unidad E es Prueba
El n mero de serie del volumen es: 32C1-ED97

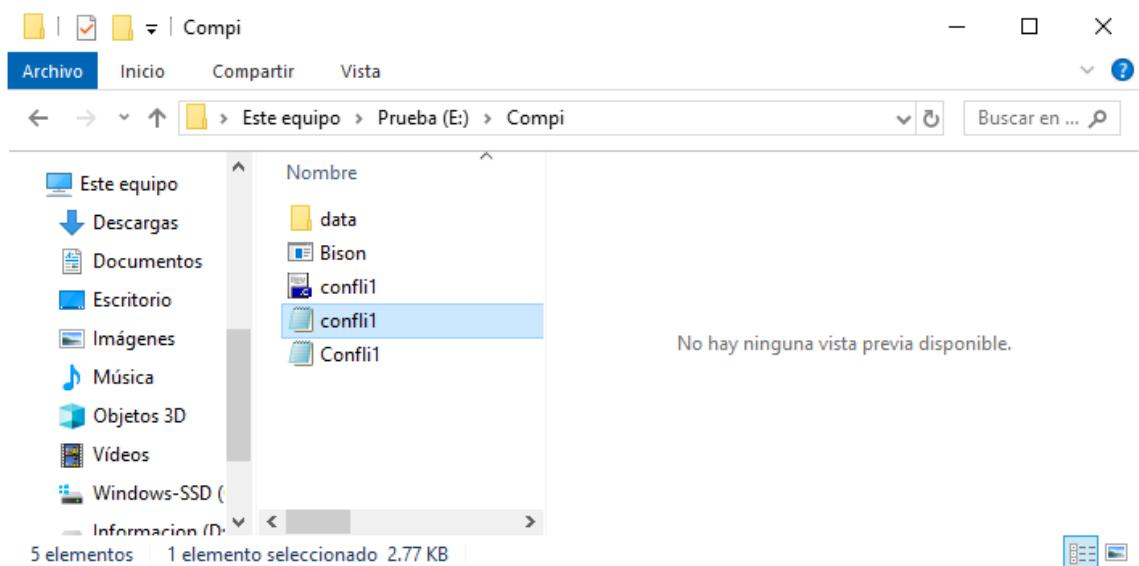
Directorio de E:\Compi

08/07/2020 16:30    <DIR>      .
08/07/2020 16:30    <DIR>      ..
24/11/2011 16:14      509,440 Bison.exe
17/02/2003 16:52      625 Confl1.Y
08/07/2020 16:29    <DIR>      data
                2 archivos      510,065 bytes
                3 dirs   125,225,115,648 bytes libres

E:\Compi>bison -v -oconfli1.c confl1.y
confli1.y: conflicts: 1 shift/reduce

E:\Compi>
```

4. En la carpeta creada (COMPI) aparece generado el archivo confl1.c y confl1.output



5. Cargar el archivo confl1.output que contiene la tabla de an lisis sint ctico LR e indica las reglas que generan conflictos.

confli1: Bloc de notas

Archivo Edición Formato Ver Ayuda

State 13 conflicts: 1 shift/reduce

Grammar

```
0 $accept: programa $end

1 programa: sentencia
2           | programa sentencia

3 sentencia: sentGoto
4           | expresion

5 sentGoto: GOTO etiqueta

6 etiqueta: IDENT

7 expresion: expresion '+' expresion
8           | ENTERO
9           | IDENT
```

Ln 1, Col 1 100% Windows (CRLF) UTF-8

confli1: Bloc de notas

Archivo Edición Formato Ver Ayuda

state 12

```
7 expresion: expresion '+' . expresion

ENTERO shift, and go to state 1
IDENT shift, and go to state 2

expresion go to state 13
```

state 13

```
7 expresion: expresion . '+' expresion
7           | expresion '+' expresion .

'+' shift, and go to state 12

'+' [reduce using rule 7 (expresion)]
$default reduce using rule 7 (expresion)
```

Ln 165, Col 43 100% Windows (CRLF) UTF-8

Paso 4: Rediseñar la gramática. Se debe rediseñar las reglas que generan el conflicto. En este caso la regla 7.

A continuación, se presenta la GLC rediseñada.

```
%{  
int yystopparser=0;  
%}  
  
%token ENTERO IDENT GOTO  
%start programa  
  
%%  
  
programa      : sentencia  
              | programa sentencia  
              ;  
sentencia     : sentGoto  
              | expresion  
              ;  
sentGoto      : GOTO etiqueta  
              ;  
etiqueta       : IDENT  
              ;  
expresion     : expresion '+' termino  
              | termino  
              ;  
termino        : ENTERO  
              | IDENT  
              ;
```

Ejercicios propuestos:

Rediseñar la siguiente GLC a fin de eliminar los conflictos.

```
%token ENTERO IDENT  
%start asignacion  
  
%%  
asignacion    : asignauno  
              | asignados  
              ;  
asignauno     : IDENT '=' expresion ';'  
              ;  
asignados     : asignauno otraasigacion  
              ;  
otraasigacion : /*vacio */  
              | asignacion  
              ;  
expresion     : ENTERO  
              | IDENT  
              ;
```

 UNIVERSIDAD PRIVADA DEL NORTE	Diseño de Gramáticas de Libre Contexto para un lenguaje de programación	Fecha: 26/11/2022
--	---	-------------------

Programa de estudio	Experiencia curricular	Sesión
INGENIERÍA DE SISTEMAS COMPUTACIONALES	COMPILEDORES Y LENGUAJES DE PROGRAMACIÓN	15

I. OBJETIVO

Diseñar gramáticas de libre contexto para un lenguaje de programación de alto nivel

II. MATERIALES Y MÉTODOS

2.1. Materiales

Computador y el entorno de desarrollo integrado (IDE) Dev-C++ para programar en lenguaje C/C++ (<https://sourceforge.net/projects/orwelldevcpp/>). También BISON (serán proporcionado por el profesor)

2.2. Metodología

Los estudiantes diseñan GLC para un lenguaje de programación, siguiendo la explicación del docente en base a ejemplos propuestos.

III. REFERENCIA BIBLIOGRAFÍA

Aho, A., Lam, M., Sethi, R. y Ullman, J. (2008). Compiladores. Principios, técnicas y herramientas. España: Editorial Addison Wesley Iberoamericana.

DISEÑO DE GLC PARA UN LP DE ALTO NIVEL

Ejemplo:

Paso 1. Identificar los componentes léxicos del lenguaje de programación

- | | |
|--------------------------|-------------------------|
| 1. Identificador: | letra(letra digito)* |
| 2. Entero: | digito+ |
| 3. Escribir | WRITE |
| 4. Leer | READ |
| 5. Cadena | "(letra digito : = ?)+" |
| 6. Asignador: | = |
| 7. Suma: | + |
| 8. Por: | * |
| 9. Paréntesis izquierdo: | (|
| 10. Paréntesis derecho: |) |
| 11. Punto y coma: | ; |

Paso 2. Escribir ejemplos de programas fuente típicos en el lenguaje de programación que se desea diseñar.

```
A=14+23;  
B= 66*45;  
Res=A+B;
```

```
Val1=14;  
Val2=34;  
Res1=Val1 * Val2;
```

```
WRITE ("Largo:");  
READ (Largo)  
WRITE ("Ancho:");  
READ (Ancho);  
Area = Largo * Ancho;  
WRITE ("El Area es:");  
WRITE (Area);
```

Paso 3. Diseñar la GLC

```
%{  
  
int yystopparser=0;  
%}  
  
%token IDENT ENTERO CADENA READ WRITE  
%start Sentencia  
  
%%
```

```
Sentencia : Asignacion Sentencia  
          | Leer Sentencia  
          | Escribir Sentencia  
          | Asignacion  
          | Leer  
          | Escribir  
          ;  
Leer      : READ('IDENT') ';' ;  
Escribir   : WRITE ('CADENA') ';' ;  
           | WRITE ('Expresion') ';' ;  
Asignacion : IDENT '=' Expresion ';' ;  
Expresion  : Expresion '+' Termino  
          | Termino  
          ;  
Termino    : Termino '*' Factor  
          | Factor  
          ;  
Factor     : ENTERO  
          | IDENT  
          | '('Expresion')'  
          ;
```

Paso 4. Verificar el diseño usando bison.

Ejercicio:

Tomando como base un lenguaje de programación que usted conoce, diseñar la GLC

Tomar como referencia C++

```
MiniC      : Librerias Cuerpo
;
Librerias   : INCLUDE CTYPE Librerias
| INCLUDE STDIO Librerias
| INCLUDE IOTREAM Librerias
| /*vacio*/
;
Cuerpo      | TipoCuerpo MAIN '(' VOID ')' '{' Detalles '}'
;
TipoCuerpo  : INT
| VOID
;
Detalles    : Variables Sentencias
;
Variables   : TipoVariable IDENT MasIdent ';' Variables
| /*vacio*/
;
TipoVariable: INT
| FLOAT
| CHAR
;
MasIdent    : ',' IDENT MasIdent
| /*vacio*/
;
Sentencias  : Asignacion Sentencias
| Leer Sentencias
| Escribir Sentencias
| Si Sentencias
| Mientras Sentencias
| /*vacio*/
;
Leer        : READ('IDENT') ';' 
;
Escribir    : WRITE "('CADENA') ';' "
| WRITE "('Expresion') ';' "
;
Asignacion  : IDENT '=' Expresion ';'
;
Expresion   : Expresion '+' Termino
| Termino
;
Termino     : Termino '*' Factor
| Factor
;
Factor      : ENTERO
| IDENT
| '('Expresion')'
```

