Universidad de Guadalajara.

Centro Universitario de Ciencias Exactas e Ingenierías.

Ingeniería en Computación.

Seminario de solución de problemas en traductores de lenguaje II.

Diego Armando Sánchez Rubio.

217570609.

AVANCE.

Introducción.

En este documento se planea explicar de manera clara y concisa el cómo funciona el analizador léxico y sintáctico que se realizó, este teniendo un cierto número de tokens ya definidos, así como el funcionamiento de este, también realizarán pruebas para mostrar cómo funcionan en caso de éxito y en caso dónde falla, esto con el propósito de mostrar el cómo se manejan los errores y cómo funciona adecuadamente el programa.

Implementación.

En esta ocasión se decidió utilizar el Lenguaje C++ por la familiaridad que tengo con este, para eso tenemos varias clases, una para el analizador léxico, una para la estructura de datos que en este caso será la pila y otra para el ElementoPila, qué prácticamente consiste en decirnos si es un No Terminal, Terminal o Estado, el analizador sintáctico o la tarea que se hace para simular a este se encuentra en el main.

```
class Tipo
{
public:
    static const int ERROR = -1;
    static const int IDENTIFICADOR = 0;
    static const int ENTERO = 1;
    static const int REAL = 2;
    static const int CADENA = 3;
    static const int TIPO = 4;              // Palabra reservada: int, float
    static const int OPSUMA = 5;            // Operadores de suma: +, -
    static const int OPMUL = 6;             // Operadores de multiplicación: *, /
    static const int OPRELAC = 7;           // Operadores relacionales: <, >, <=, >=
    static const int OPOR = 8;              // Operador OR: ||
    static const int OPAND = 9;             // Operador AND: &&
    static const int OPNOT = 10;            // Operador NOT: !
    static const int OPIGUALDAD = 11;       // Operadores de igualdad: ==, !=
    static const int PUNTO_Y_COMA = 12;     // Punto y coma: ;
    static const int COMA = 13;             // Coma: ,
    static const int PAREN_IZQ = 14;        // Paréntesis izquierdo: (
    static const int PAREN_DER = 15;        // Paréntesis derecho: )
    static const int LLAVE_IZQ = 16;        // Llave izquierda: {
    static const int LLAVE_DER = 17;        // Llave derecha: }
    static const int ASIGNACION = 18;       // Operador de asignación: =
    static const int IF = 19;               // Palabra reservada: if
    static const int WHILE = 20;            // Palabra reservada: while
    static const int RETURN = 21;           // Palabra reservada: return
    static const int ELSE = 22;             // Palabra reservada: else
    static const int FIN = 23;              // Fin de cadena: $
};
```

En la imagen anterior se puede ver la clase Tipo qué define una serie de constantes estáticas que representan los diferentes tipos de tokens que el analizador léxico puede identificar. Cada constante está asociada a un valor entero único.

Detalles de los Tokens:

Tokens Básicos:

1. IDENTIFICADOR (0): Representa nombres de variables, funciones, etc.
2. ENTERO (1): Números enteros.
3. REAL (2): Números de punto flotante.
4. CADENA (3): Cadenas de texto.

Palabras Reservadas:

1. TIPO (4): Incluye palabras como int, float.
2. IF (19), WHILE (20), RETURN (21), ELSE (22): Palabras reservadas de control de flujo.

Operadores y Símbolos:

1. OPSUMA (5): Operadores de suma (+, -).
2. OPMUL (6): Operadores de multiplicación (*, /).
3. OPRELAC (7): Operadores relacionales (<, >, <=, >=).
4. OPOR (8), OPAND (9), OPNOT (10): Operadores lógicos (||, &&, !).
5. OPIGUALDAD (11): Operadores de igualdad (==, !=).
6. PUNTO_Y_COMA (12), COMA (13), PAREN_IZQ (14), PAREN_DER (15), LLAVE_IZQ (16), LLAVE_DER (17), ASIGNACION (18): Otros símbolos de puntuación y operadores.

FIN (23): Representa el fin de la cadena de entrada ($).

```
class Analizador
{
public:
    std::string simbolo;
    int tipo;

    Analizador(std::string palabra);
    Analizador();

    void entrada(std::string palabra)
    std::string tipoToString(int tipo

    int sigSimbolo();
    bool terminado();

private:
    std::string palabra;

    int indice, estado;
    bool continuar;
    char c;

    char sigCaracter();
    void sigEstado(int);
    void valido(int);
    bool esLetra(char);
    bool esDigito(char);
    bool esEspacio(char);
    void retroceso();
    bool esReservada(std::string);
```

La clase Analizador implementa el analizador léxico que procesa una cadena de entrada y extrae los tokens definidos en la clase Tipo. En esta clase tenemos atributos cómo std::string símbolo que almacena el lexema (cadena de caracteres) correspondiente al token actual identificado y int tipo que almacena el tipo del token actual, basado en las constantes definidas en la clase Tipo.

También tenemos los siguientes métodos públicos:

- void entrada(std::string palabra): Permite establecer o actualizar la cadena de entrada que será analizada.

- std::string tipoToString(int tipo): Convierte el tipo de token (entero) a una representación en cadena de texto para facilitar la depuración o presentación.
- int sigSimbolo(): Obtiene el siguiente token de la cadena de entrada.
- bool terminado(): Indica si el análisis de la cadena de entrada ha finalizado.

Así cómo también tenemos atributos privados que ayudan al programa cómo por ejemplo:

- std::string palabra: Almacena la cadena completa que se va a analizar.
- int indice: Índice actual en la cadena de entrada.
- int estado: Estado actual del analizador.
- bool continuar: Bandera que indica si el análisis debe continuar.
- char c: Caracter actual siendo procesado.

Y por ultimo también se tienen métodos privados:

- char sigCaracter(): Obtiene el siguiente carácter de la cadena de entrada.
- void retroceso(): Retrocede el índice una posición en la cadena de entrada.
- bool esLetra(char c): Verifica si un carácter es una letra alfabética.
- bool esDigito(char c): Verifica si un carácter es un dígito.
- bool esEspacio(char c): Verifica si un carácter es un espacio en blanco.
- bool esReservada(std::string palabra): Verifica si una cadena coincide con una palabra reservada y actualiza el tipo de token en consecuencia.

La parte más importante de este código es la función sigSimbolo ya que es la que hace la mayoría de clasificación de tokens. Funciona de la siguiente manera:
Ignorar Espacios en Blanco:

Si el carácter actual es un espacio en blanco, se omite y se continúa con el siguiente carácter.

Identificación de Identificadores y Palabras Reservadas:

- Si el carácter es una letra, se inicia la construcción de un identificador o una palabra reservada.
- Se concatena cada letra o dígito subsecuente para formar el lexema completo.
- Después de formar el lexema, se verifica si coincide con una palabra reservada utilizando el método esReservada.
- Si es una palabra reservada, se retorna su tipo correspondiente; de lo contrario, se clasifica como IDENTIFICADOR.

Identificación de Números (Enteros y Reales):

- Si el carácter es un dígito, se inicia la construcción de un número.
- Se concatenan todos los dígitos subsecuentes.
- Si después de los dígitos aparece un punto (.), se continúa concatenando dígitos para formar un número real.
- Dependiendo de la presencia del punto, se clasifica el número como ENTERO o REAL.

Identificación de Operadores y Símbolos Especiales:

- Operadores de Suma (+, -): Se clasifican como OPSUMA.
- Operadores de Multiplicación (*, /): Se clasifican como OPMUL.
- Operadores Relacionales (<, >, <=, >=): Se clasifican como OPRELAC. Si se detecta un = después de < o >, se incluye en el operador.
- Operadores de Negación y Comparación (!, !=): OPNOT para ! y OPIGUALDAD para !=.
- Operadores de Igualdad (==): Se clasifican como OPIGUALDAD.
- Operadores Lógicos (||, &&): OPOR para || y OPAND para &&. Si no se detecta el carácter esperado, se marca como ERROR.
- Otros Símbolos (;, ,, (,), {, }, =): Se clasifican según corresponda (por ejemplo, PUNTO_Y_COMA, COMA, etc.).
- Fin de Cadena ($): Marca el fin de la entrada y detiene el análisis.

- Caracteres No Reconocidos: Se clasifican como ERROR.

Retorno del Método:

El método retorna el tipo de token identificado, que puede ser utilizado por otras partes del compilador para el análisis sintáctico.

Analizador Sintáctico o Main.

Otra de las partes muy importantes para el programa es el analizador sintáctico, este se va a encargar de darle un sentido y de aprobar las sentencias que envíe el usuario, para esto el analizador tiene que guiarse por una gramática, en este caso es una proporcionada por el profesor en un archivo, entonces este tanto este analizador sintáctico cómo el léxico está adaptados para ese caso, el analizador sintáctico funciona de la siguiente forma.

```
struct ReglaProduccion {
    int id; // ID del no terminal
    int numSimbolos; // Cantidad de símbolos en el lado derecho
    string nombre; // Nombre del no terminal
};
```

Esta estructura representa a la regla de producción de la gramática que el analizador sintáctico utilizará durante el proceso de reducción, tiene los siguiente atributos:

- id: Identificador único del no terminal en la regla de producción.
- numSimbolos: Número de símbolos presentes en el lado derecho de la regla.
- nombre: Nombre del no terminal en el lado izquierdo de la regla.

```
struct TablaLR {
    vector<vector<int>> tabla; // Tabla LR(1)
    vector<string> simbolos; // Lista de símbolos
    vector<ReglaProduccion> producciones; // Lista de reglas de producción
    int numFilas;
    int numColumnas;
};
```

Después tenemos la estructura TablaLR que almacena la información necesaria para el análisis sintáctico, incluyendo la tabla LR(1), la lista de símbolos y las reglas de producción, esta tiene los siguientes atributos:

- tabla: Matriz que representa la tabla LR(1), donde las filas corresponden a estados y las columnas a símbolos (terminales y no terminales). Los valores en la tabla indican acciones como desplazamiento, reducción, aceptación o error.
- simbolos: Lista ordenada de símbolos que corresponden a las columnas de la tabla LR(1).
- producciones: Lista de reglas de producción que se utilizarán para realizar reducciones.
- numFilas: Número de filas en la tabla LR(1), correspondiente al número de estados.
- numColumnas: Número de columnas en la tabla LR(1), correspondiente al número de símbolos.

Ahora tenemos a dos de las funciones más importantes del programa que son las siguientes

```
bool cargarTablaLR(TablaLR& tabla, const string& nombreArchivo);
void analizarCadena(Analizador& analizador, TablaLR& tabla);
```

CargarTablaLR prácticamente consistes en cargar la tablaLR definida en el archivo, para poder hacer esto se tiene que hacer lo siguiente:

Apertura del Archivo:

- Intenta abrir el archivo especificado. Si falla, emite un mensaje de error y retorna false.
- Lectura de Reglas de Producción:
- Lee el número total de reglas (numReglas).
- Itera sobre cada regla, leyendo su id, numSimbolos y nombre, y las almacena en la lista producciones.

Lectura de la Tabla LR(1):

- Lee las dimensiones de la tabla (numFilas y numColumnas).
- Redimensiona la matriz tabla para acomodar las filas y columnas especificadas.
- Lee cada valor de la tabla, asegurando la correcta asignación de acciones en cada posición (i, j).

Definición de Símbolos Predefinidos:

Inicializa la lista simbolos con los símbolos terminales y no terminales en el orden esperado, priorizando primero los tokens (símbolos terminales) seguidos de los no terminales.

Verificación de No Terminales:

Asegura que todos los no terminales utilizados en las reglas de producción estén presentes en la lista de símbolos. Si alguno falta, lo agrega automáticamente y emite una advertencia.

Verificación del Número de Símbolos:

Compara el tamaño de la lista simbolos con el número de columnas de la tabla LR(1). Si hay discrepancias, emite una advertencia, lo cual puede indicar un mapeo incorrecto de símbolos a columnas.

Retorno:

Si todo se carga correctamente, retorna true; de lo contrario, false.

Ahora tenemos la siguiente función que se encarga de analizar toda la sentencia o cadena del usuario, está es la más importante dentro de este analizador, funciona de la siguiente forma:

Inicialización de la Pila:

- Crea una instancia de Pila (presumiblemente una estructura de datos que maneja estados y símbolos).
- Empuja el estado inicial (0) en la pila.

Obtención del Primer Token:

Llama a sigSimbolo() del analizador léxico para obtener el primer token de la cadena de entrada.

Proceso de Análisis:

Entra en un bucle continuo que se ejecuta hasta que se detecta un error o se completa el análisis.

Obtención del Estado Actual:

Obtiene el estado actual desde la cima de la pila.

Mapeo del Token a una Columna:

Busca el índice de la columna correspondiente al tipo de token actual en la lista de símbolos.

Verificación de Símbolo Reconocido:

Si el símbolo no se encuentra en la lista, emite un error y termina el análisis.

Determinación de la Acción a Realizar:

Obtiene la acción desde la tabla LR(1) usando el estado actual y la columna del token.

Acción de Desplazamiento (Shift):

- Si la acción es positiva, desplaza (shifts) el token y empuja el nuevo estado en la pila.
- Obtiene el siguiente token del analizador léxico.

Acción de Reducción (Reduce):

- Si la acción es negativa, determina la regla de producción correspondiente.
- Desapila la cantidad adecuada de símbolos y estados según el número de símbolos en la regla.

- Busca la columna correspondiente al no terminal de la regla y empuja el no terminal y el nuevo estado en la pila.

Acción Inválida:

- Si la acción es 0, emite un error y termina el análisis.
- Impresión del Estado de la Pila:
- Después de cada acción, imprime el estado actual de la pila para facilitar la depuración.

Finalización del Análisis:

El análisis finaliza cuando se encuentra una acción de aceptación, un error, o se completa el procesamiento de la cadena de entrada.

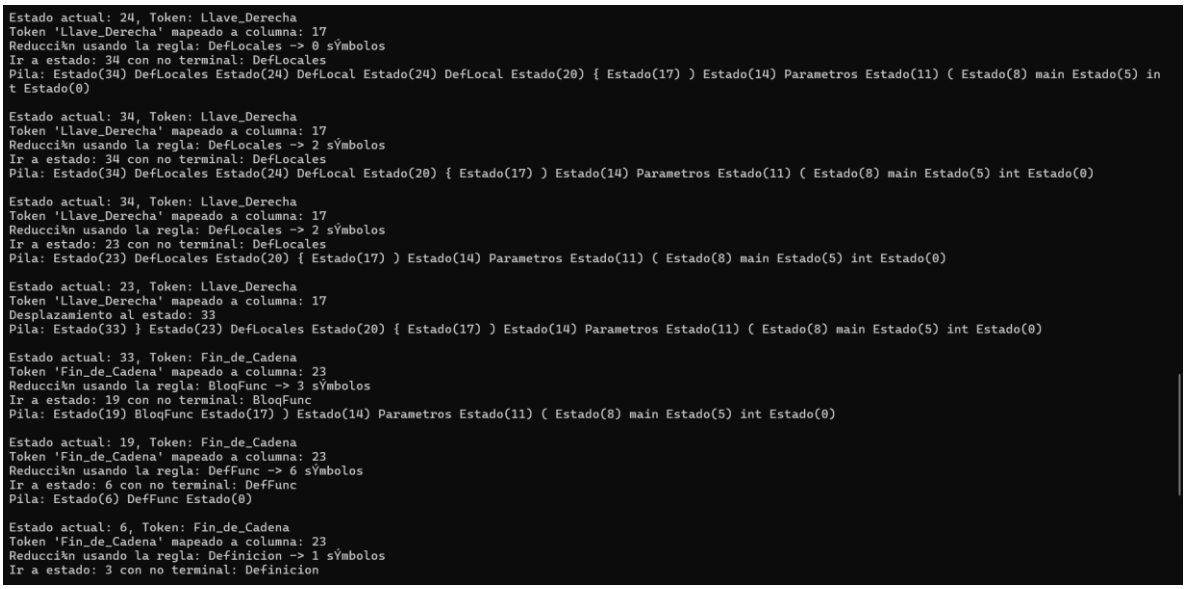
PROGRAMA EN FUNCIONAMIENTO.

Caso de éxito.

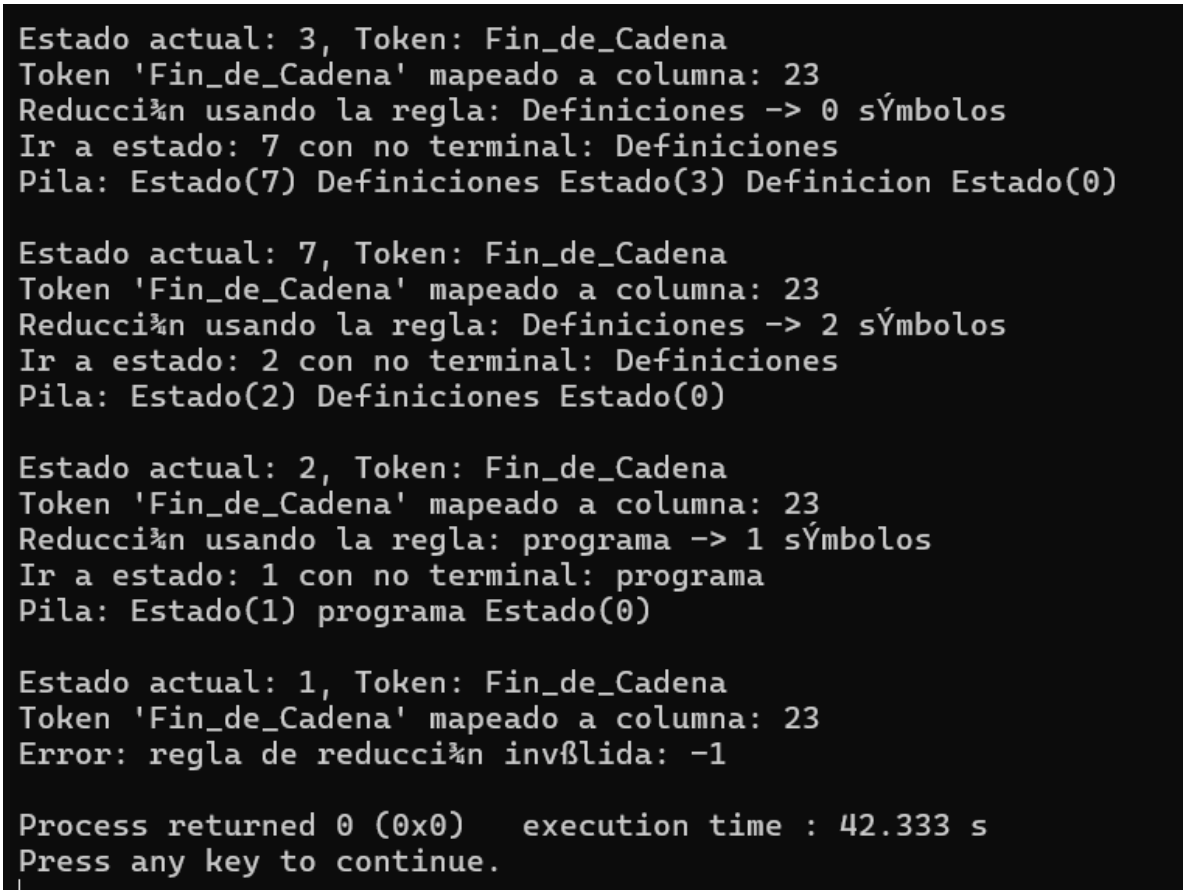
```
Ingrese la cadena a analizar: int main(){int a, b; float c;}

Estado actual: 0, Token: Tipo
Token 'Tipo' mapeado a columna: 4
Desplazamiento al estado: 5
Pila: Estado(5) int Estado(0)

Estado actual: 5, Token: Identificador
Token 'Identificador' mapeado a columna: 0
Desplazamiento al estado: 8
Pila: Estado(8) main Estado(5) int Estado(0)

Estado actual: 8, Token: Parentesis_Izquierdo
Token 'Parentesis_Izquierdo' mapeado a columna: 14
Desplazamiento al estado: 11
Pila: Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 11, Token: Parentesis_Derecho
Token 'Parentesis_Derecho' mapeado a columna: 15
Reducci¾n usando la regla: Parametros -> 0 sÝmbolos
Ir a estado: 14 con no terminal: Parametros
Pila: Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 14, Token: Parentesis_Derecho
Token 'Parentesis_Derecho' mapeado a columna: 15
Desplazamiento al estado: 17
Pila: Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 17, Token: Llave_Izquierda
Token 'Llave_Izquierda' mapeado a columna: 16
Desplazamiento al estado: 20
Pila: Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 20, Token: Tipo
Token 'Tipo' mapeado a columna: 4
Desplazamiento al estado: 5
Pila: Estado(5) int Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 5, Token: Identificador
Token 'Identificador' mapeado a columna: 0
Desplazamiento al estado: 8
```

```
Estado actual: 8, Token: Coma
Token 'Coma' mapeado a columna: 13
Desplazamiento al estado: 10
Pila: Estado(10) , Estado(8) a Estado(5) int Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 10, Token: Identificador
Token 'Identificador' mapeado a columna: 0
Desplazamiento al estado: 13
Pila: Estado(13) b Estado(10) , Estado(8) a Estado(5) int Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0
)

Estado actual: 13, Token: Punto_y_Coma
Token 'Punto_y_Coma' mapeado a columna: 12
Reducci%n usando la regla: ListaVar -> 0 sÝmbolos
Ir a estado: 16 con no terminal: ListaVar
Pila: Estado(16) ListaVar Estado(13) b Estado(10) , Estado(8) a Estado(5) int Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Es
tado(5) int Estado(0)

Estado actual: 16, Token: Punto_y_Coma
Token 'Punto_y_Coma' mapeado a columna: 12
Reducci%n usando la regla: ListaVar -> 3 sÝmbolos
Ir a estado: 9 con no terminal: ListaVar
Pila: Estado(9) ListaVar Estado(8) a Estado(5) int Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 9, Token: Punto_y_Coma
Token 'Punto_y_Coma' mapeado a columna: 12
Desplazamiento al estado: 12
Pila: Estado(12) ; Estado(9) ListaVar Estado(8) a Estado(5) int Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Es
tado(0)

Estado actual: 12, Token: Tipo
Token 'Tipo' mapeado a columna: 4
Reducci%n usando la regla: DefVar -> 4 sÝmbolos
Ir a estado: 25 con no terminal: DefVar
Pila: Estado(25) DefVar Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)
```

```
Estado actual: 25, Token: Tipo
Token 'Tipo' mapeado a columna: 4
Reducci%n usando la regla: DefLocal -> 1 sÝmbolos
Ir a estado: 24 con no terminal: DefLocal
Pila: Estado(24) DefLocal Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 24, Token: Tipo
Token 'Tipo' mapeado a columna: 4
Desplazamiento al estado: 5
Pila: Estado(5) float Estado(24) DefLocal Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 5, Token: Identificador
Token 'Identificador' mapeado a columna: 0
Desplazamiento al estado: 8
Pila: Estado(8) c Estado(5) float Estado(24) DefLocal Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 8, Token: Punto_y_Coma
Token 'Punto_y_Coma' mapeado a columna: 12
Reducci%n usando la regla: ListaVar -> 0 sÝmbolos
Ir a estado: 9 con no terminal: ListaVar
Pila: Estado(9) ListaVar Estado(8) c Estado(5) float Estado(24) DefLocal Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(
5) int Estado(0)

Estado actual: 9, Token: Punto_y_Coma
Token 'Punto_y_Coma' mapeado a columna: 12
Desplazamiento al estado: 12
Pila: Estado(12) ; Estado(9) ListaVar Estado(8) c Estado(5) float Estado(24) DefLocal Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8)
 main Estado(5) int Estado(0)

Estado actual: 12, Token: Llave_Derecha
Token 'Llave_Derecha' mapeado a columna: 17
Reducci%n usando la regla: DefVar -> 4 sÝmbolos
Ir a estado: 25 con no terminal: DefVar
Pila: Estado(25) DefVar Estado(24) DefLocal Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 25, Token: Llave_Derecha
Token 'Llave_Derecha' mapeado a columna: 17
Reducci%n usando la regla: DefLocal -> 1 sÝmbolos
Ir a estado: 24 con no terminal: DefLocal
Pila: Estado(24) DefLocal Estado(24) DefLocal Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)
```



```
Estado actual: 24, Token: Llave_Derecha
Token 'Llave_Derecha' mapeado a columna: 17
Reducci¾n usando la regla: DefLocales -> 0 sÝmbolos
Ir a estado: 34 con no terminal: DefLocales
Pila: Estado(34) DefLocales Estado(24) DefLocal Estado(24) DefLocal Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) in
t Estado(0)

Estado actual: 34, Token: Llave_Derecha
Token 'Llave_Derecha' mapeado a columna: 17
Reducci¾n usando la regla: DefLocales -> 2 sÝmbolos
Ir a estado: 34 con no terminal: DefLocales
Pila: Estado(34) DefLocales Estado(24) DefLocal Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 34, Token: Llave_Derecha
Token 'Llave_Derecha' mapeado a columna: 17
Reducci¾n usando la regla: DefLocales -> 2 sÝmbolos
Ir a estado: 23 con no terminal: DefLocales
Pila: Estado(23) DefLocales Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 23, Token: Llave_Derecha
Token 'Llave_Derecha' mapeado a columna: 17
Desplazamiento al estado: 33
Pila: Estado(33) } Estado(23) DefLocales Estado(20) { Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 33, Token: Fin_de_Cadena
Token 'Fin_de_Cadena' mapeado a columna: 23
Reducci¾n usando la regla: BloqFunc -> 3 sÝmbolos
Ir a estado: 19 con no terminal: BloqFunc
Pila: Estado(19) BloqFunc Estado(17) ) Estado(14) Parametros Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 19, Token: Fin_de_Cadena
Token 'Fin_de_Cadena' mapeado a columna: 23
Reducci¾n usando la regla: DefFunc -> 6 sÝmbolos
Ir a estado: 6 con no terminal: DefFunc
Pila: Estado(6) DefFunc Estado(0)

Estado actual: 6, Token: Fin_de_Cadena
Token 'Fin_de_Cadena' mapeado a columna: 23
Reducci¾n usando la regla: Definicion -> 1 sÝmbolos
Ir a estado: 3 con no terminal: Definicion
```



```
Estado actual: 3, Token: Fin_de_Cadena
Token 'Fin_de_Cadena' mapeado a columna: 23
Reducci¾n usando la regla: Definiciones -> 0 sÝmbolos
Ir a estado: 7 con no terminal: Definiciones
Pila: Estado(7) Definiciones Estado(3) Definicion Estado(0)

Estado actual: 7, Token: Fin_de_Cadena
Token 'Fin_de_Cadena' mapeado a columna: 23
Reducci¾n usando la regla: Definiciones -> 2 sÝmbolos
Ir a estado: 2 con no terminal: Definiciones
Pila: Estado(2) Definiciones Estado(0)

Estado actual: 2, Token: Fin_de_Cadena
Token 'Fin_de_Cadena' mapeado a columna: 23
Reducci¾n usando la regla: programa -> 1 sÝmbolos
Ir a estado: 1 con no terminal: programa
Pila: Estado(1) programa Estado(0)

Estado actual: 1, Token: Fin_de_Cadena
Token 'Fin_de_Cadena' mapeado a columna: 23
Error: regla de reducci¾n invßlida: -1

Process returned 0 (0x0)   execution time : 42.333 s
Press any key to continue.
```

Caso fallido.

```
Ingrese la cadena a analizar: int main({int a;}

Estado actual: 0, Token: Tipo
Token 'Tipo' mapeado a columna: 4
Desplazamiento al estado: 5
Pila: Estado(5) int Estado(0)

Estado actual: 5, Token: Identificador
Token 'Identificador' mapeado a columna: 0
Desplazamiento al estado: 8
Pila: Estado(8) main Estado(5) int Estado(0)

Estado actual: 8, Token: Parentesis_Izquierdo
Token 'Parentesis_Izquierdo' mapeado a columna: 14
Desplazamiento al estado: 11
Pila: Estado(11) ( Estado(8) main Estado(5) int Estado(0)

Estado actual: 11, Token: Llave_Izquierda
Token 'Llave_Izquierda' mapeado a columna: 16
Error: acci%n no vßlida (0).

Process returned 0 (0x0)   execution time : 15.891 s
Press any key to continue.
```