Gramática de la función Terminó:

<termino> -> <termino> \* <primaria>

<termino> -> <termino> / <primaria>

<termino> -> <primaria>

| Paso | Acción del Parser | Programa sin Procesar | Instruc. Generada para MV |
| --- | --- | --- | --- |
| 1 | Llamar Objetivo() | inicio a:= 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 2 | Llamar Programa() | inicio a:= 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 3 | Acción Semántica Comenzar() | inicio a:= 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 4 | Match(INICIO) | inicio a:= 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 5 | Llamar ListaSentencias() | a:= 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; ; fin FDT |  |
| 6 | Llamar Sentencia() | a:= 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 7 | Llamar Identificador() | a:= 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 8 | Match(ID) | a:= 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 9 | Acción Semántica ProcesarId() | := 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT | Declara a, Entera |
| 10 | Match(ASIGNACION) | := 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 11 | Llamar Expresion() | 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 12 | Llamar Termino() | 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 13 | Llamar Primaria() | 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 14 | Match(CONSTANTE) | 4; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 15 | Llamar procesarCte () | ; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 17 | Acción Semántica Asignar(a,4) | ; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT | Almacena 4,a |
| 18 | Match(PUNTOYCOMA) | ; b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 19 | Llamar sentencia() | b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 20 | Llamar identificador() | b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 21 | Match(ID) | b:= 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 22 | Acción Semántica ProcesarId() | := 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT | Declara b, Entera |
| 23 | match (ASIGNACION) | := 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 24 | Llamar Expresion() | 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 25 | Llamar Termino() | 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 26 | Llamar Primaria() | 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 27 | Match (CONSTANTE) | 5\*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 28 | Acción semántica procesarCte() | \*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 29 | Llamar Operador() | \*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 30 | Match (MULTIPLICACION) | \*(a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 31 | llamar procesarOp () | (a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 32 | Llamar Primaria() | (a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 33 | Match (PARENIZQUIERDO) | (a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 34 | Llamar Expresion () | a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 35 | Llamar Termino () | a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 36 | Llamar Primaria () | a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 37 | Llamar Identificador () | a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 38 | Match (ID) | a+4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 39 | Acción Semántica ProcesarId() | +4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT | (no imprime la declaración porque a ya está en la TS) |
| 40 | llamar Operador () | +4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 41 | Match (SUMA) | +4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 42 | Llamar Primaria () | 4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 43 | Match (CONSTANTE) | 4); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 44 | Acción Semántica ProcesarCte() | ); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 45 | GenInfijo (5, \*, (a+4)) | ); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT | Declara Temp&1,Entera,  Sumar a,4,Temp&1  Declara Temp&2,Entera,  Multiplicar 5,Temp&1,Temp&2 |
| 46 | Match (PARENDERECHO) | ); escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 47 | Accion semantica Asignar(b, “5\*(a+4)”) | ; escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT | Almacena Temp&2,b |
| 48 | Match (PUNTOYCOMA) | ; escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 49 | llamar Sentencia () | escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 50 | Match (ESCRIBIR) | escribir(a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 51 | match (PARENIZQUIERDO) | (a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 52 | llamar ListaExpresiones () | a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 53 | Llamar Expresion () | a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 54 | Llamar Termino () | a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 55 | llamar Primaria () | a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 56 | Llamar Identificador () | a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 57 | Match (ID) | a); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 58 | Acción Semántica procesarId () | ); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 59 | Llamar Escribir () | ); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT | Write a entera |
| 60 | Match (PARENDERECHO) | ); leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 61 | Match (PUNTOYCOMA) | ; leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 62 | Match (LEER) | leer(a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 63 | Match (PARENIZQUIERDO) | (a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 64 | Llamar ListaIdentificadores() | a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 65 | llamar Identificador () | a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 66 | Match (ID) | a,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 67 | Acción Semántica procesarId () | ,b); escribir(hola); leer(c); c:= a+b; fin FDT | (no se imprime la declaración de a porque ya fue declarada) |
| 68 | leer(a) | ,b); escribir(hola); leer(c); c:= a+b; fin FDT | Read a Entera |
| 69 | Match (COMA) | ,b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 70 | llamar Identificador () | b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 71 | Match (ID) | b); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 72 | Acción Semántica procesarId () | ); escribir(hola); leer(c); c:= a+b; fin FDT | (no se imprime la declaración de b porque ya fue declarada) |
| 73 | leer(b) | ); escribir(hola); leer(c); c:= a+b; fin FDT | Read b Entera |
| 74 | Match (PARENDERECHO) | ); escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 75 | Match (PUNTOYCOMA) | ; escribir(hola); leer(c); c:= a+b; fin FDT |  |
| 76 | SALTO AL FINAL DEL PROGRAMA |  |  |
| 77 | Match (FIN) | fin FDT |  |
| 78 | Match (FDT) | FDT |  |
| 79 | terminar () |  | Detiene |
| 80 |  |  |  |
| 80 |  |  |  |
|  |  |  |  |

| Accion del parser | Comportamiento | Contexto |
| --- | --- | --- |
| objetivo() | llama a programa(), match(FDT), terminar() |  |
| programa() | llama a comenzar() y hace un match de inicio. con ListaSentencias() irá procesando el contenido del código |  |
| math(INICIO) | se asegura que el próximo token a leer (el primero) sea inicio |  |
| listaSentencias() | hace un while infinito hasta que se encuentre que una linea no comienza con un ID, LEER o ESCRIBIR y se va procesando las sentencias |  |
| sentencia() | toma el próximo token y hace un switch con opciones ID, LEER, ESCRBIR. Dentro de cada caso verifica la estructura de los tipos de sentencias para cada caso |  |
| identificador(reg) | hace un match(ID) y se declara el ID con el struct REG\_EXPRESION |  |
| expresion(reg) | define un operandoIzq y operandoDer. verifica que el primer operando sea una primaria (que no sea un operador aditivo). Luego identifica el operador aditivo con una función y el próximo token lo procesa como una primaria llamando a Primaria(&operandoDer). Para definir el resultado utiliza la función GenInfijo y lo almacena el operandoIzq, que es la variable que va a devolver. |  |
| primaria(reg) | Verifica si el token es de tipo ID, CONSTANTE o un paréntesis izquierdo, en cuyo caso devuelve por parámetro la expresión contenida y verifica que el paréntesis cierre. |  |
| funciones de tipo “procesar” | asignan a una variable de tipo REG\_EXPRESION la clase (reg.clase = CONSTANTE, ID …etc), busca si la cadena esta en la tabla de símbolos y sino lo genera como nuevo con la función RevisaEnTS() (para el caso de las constantes y operadores este paso se saltea)  RevisaEnTS devuelve el buffer que pasará a representar el reg.nombre del registro semántico. Las funciones de procesar devuelven este registro con sus datos |  |
| OperadorAditivo(char\*) | establece el próximo token “t” y si es un token SUMA o RESTA hace un match(t) |  |
| proximoToken() | primero se fija que el flagToken este en 0. Si eso se cumple busca el token actual en el scanner. Si es un id, lo busca en la tabla de símbolos y si es un ERROR LEXICO llama a la función ErrorLexico(). por último se retorna el token actual |  |
| scanner() | Tiene una matriz de estados. Al igual que los autómatas, va buscando caracter por caracter su estado, utilizando fgetc(archivo). Una vez que llega al estado o un error léxico (estado 14) termina de almacenar en el **buffer** y el próximo elemento del buffer lo pone como \0.  Luego entra al switch y se fija en los estados de aceptación, si es un ID o una CONSTANTE, verifica y elimina espacios en blanco de más para asignar el \0. En los otros casos, directamente devuelve el tipo de token que es. |  |
| GenInfijo | Se utiliza para operaciones infijas de suma o resta. Primero identifica la cadena con el operador, a la cual le asigna un string que describe si es una suma o una resta. El sprintf convierte una variable que cuenta las temporales, numTemp, en una cadena de caracteres y la almacena en cadNum. Luego cadNum se concatena con cadTemp y se pone como atributo “nombre” de la reg que se devuelve. |  |
| GenerarInstruccion(4 de char\*) | hace un printf de la instrucción y las cadenas separadas por una coma |  |
| RegTS *struct* | cada elemento RegTS tiene una cadena llamada identifi y un token. para inicio, su identifi es “inicio” y su token es INICIO (los tokens son enums). Abajo de la declaración del struct se define una tabla con 5 declaraciones para inicio, fin, leer, escribir y $, asociado al 99. Este $ funciona como un sentinela que indica el fin de la tabla. |  |
| REG\_EXPRESION | tiene clase, nombre y valor |  |
| procesarCte() | con un REG\_EXPRESION le asigna la clase = constante, hace y strcpy del nombre con el contenido en el buffer y convierte el valor del buffer en entero para asignarlo al valor. |  |
| procesarId() | chequea que lo que se encuentra en el buffer exista en la tabla de símbolos (sino lo agrega) le asigna la clase ID y copia el nombre desde el buffer |  |
| procesarOp() | directamente devuelve el buffer |  |

**<sentencia> -> ID := <expresion> #asignar ;**

**cadena y lexema:**

Un **lexema** tiene un significado dentro del lenguaje de programación. Está asociado con un token y es relevante para el análisis léxico y sintáctico.

Una **cadena** es simplemente una secuencia de caracteres, sin necesidad de que tenga un significado léxico en un programa.

anotaciones:  
si se ingresa un leer(a); y a no está definida, eso sería un error semántico

sobre el codigo flex y bison:

**scaner.l: INCLUDES**

#include <stdio.h>: es un archivo de cabecera que incluye funciones y tipos que se utilizan para manejar entradas y salidas (como printf o scanf)

#include <stdlib.h>: Contiene funciones estándar para la gestión de memoria, control de procesos, conversiones de tipo y otras utilidades generales

#include <string.h>: proporciona funciones para manipular y trabajar con cadenas de caracteres (arreglos de tipo char) y operaciones relacionadas con bloques de memoria.

int yylex(): Flex genera automáticamente esta función, que es el escáner léxico. Su propósito es leer la entrada (como un archivo de código fuente) y reconocer tokens a medida que va escaneando el texto

**DEFINICIONES:**

se componen por un nombre y su definición

DIGITO [0 - 9]

la definición es una ER que pueden usar nombres definidos previamente.

**REGLAS:**

Se componen por un patron y una acción por ejemplo: \n caracteres++; lineas++;

Al encontrar un \n incrementa dos contadores

{IDENTIFICADOR} {yylval.cadena = strdup(yytext); return ID;}

Cuando Flex encuentra un identificador, copia el texto del identificador a yylval.cadena y devuelve el token ID.

{CONST} {yylval.num = atoi(yytext); return CONSTANTE; }

Cuando Flex encuentra una constante, convierte el texto a un número entero (con atoi) y lo guarda en yylval.num, luego devuelve el token CONSTANTE

El código del doctor no identifica cuando lees una variable no asignada y dice que la declaraste igualmente (cuando en realidad no).

PRIMARIA:  
El término "primaria" proviene del hecho de que estas expresiones suelen ser las unidades más pequeñas o simples que pueden aparecer en una expresión más compleja. Este concepto se usa en muchos lenguajes de programación para distinguir entre:

Expresiones simples (primarias), como variables, constantes o expresiones entre paréntesis.

Expresiones compuestas, que se forman combinando expresiones primarias con operadores (por ejemplo, sumas o restas).

número raro en resultado:  
Cuando hacemos escribir(a) y a no está definida, nos devuelve cualquier número. Eso es un valor basura, que se encuentra en la dirección asignada a esa variable.

Debería utilizar siempre recursión por la izquierda, porque puede analizar una secuencia de elementos sin ocupar espacio de pila (es decir, de forma mucho más eficiente en memoria)

**DESCENDENTE:**

<No terminal>

terminal : lo más básico (los que devuelve Flex)

inicio es un terminal pero se lo denomina axioma

**Gramática sintáctica**: Es el conjunto completo de reglas gramaticales que definen la estructura sintáctica válida de un lenguaje de programación o un lenguaje formal. Esta gramática incluye todas las reglas que describen cómo se puede generar una oración (o programa) válida en el lenguaje.

**stdio.h:** es un archivo de cabecera que incluye funciones y tipos que se utilizan para manejar entradas y salidas (como printf o scanf)

**string.h**: proporciona funciones para manipular y trabajar con cadenas de caracteres (arreglos de tipo char) y operaciones relacionadas con bloques de memoria. strcmp, strlen

**ctype.h :** Proporciona funciones para clasificar y transformar caracteres, por ejemplo isalpha, isdigit, entre otras.