INTRODUCCIÓN AL PROCESO DE COMPILACIÓN

Un programa a traducir siempre se encuentra en un flujo de entrada que está implementado a través de un archivo de texto, como ocurre en Pascal con los archivos con extensión .pas o en ANSI C con los archivos con extensión .c; esto es: El programa que se debe compilar es una secuencia de caracteres que termina con un centinela.

CONCEPTOS BÁSICOS

Un compilador es un complejo programa que lee un programa escrito en un lenguaje fuente, habitualmente un lenguaje de alto nivel – como C y lo traduce a un programa equivalente en un lenguaje objeto, normalmente más cercano al lenguaje de máquina. Al programa original se lo llama programa fuente y al programa obtenido se lo denomina programa objeto.

El proceso de compilación está formado por dos partes:

1) el ANÁLISIS, que, a partir del programa fuente, crea una representación intermedia del mismo; y

2) la SÍNTESIS, que, a partir de esta representación intermedia, construye el programa objeto.

EL ANÁLISIS DEL PROGRAMA FUENTE

En la compilación, el análisis está formado por tres fases:

a) el Análisis Léxico,

b) el Análisis Sintáctico, y

c) el Análisis Semántico.

ANÁLISIS LÉXICO

Detecta los diferentes elementos básicos que constituyen un programa fuente, como: identificadores, palabras reservadas, constantes, operadores y caracteres de puntuación. Por lo tanto, el Análisis Léxico solo se ocupa de los Lenguajes Regulares que forman parte del Lenguaje de Programación.

Por ello, esta fase de análisis tiene como función detectar palabras de estos Lenguajes Regulares; en la jerga de la compilación, estas palabras se denominan **LEXEMAS** y los LRs a los que pertenecen estos lexemas se denominan **CATEGORÍAS LÉXICAS** o **TOKENS**.

La entrada para el Análisis Léxico son **caracteres** y la salida son **tokens.**

Los espacios que separan a los lexemas son ignorados durante el Análisis Léxico.

Ejercicio Sea, en ANSI C, el siguiente fragmento de programa:

... int WHILE; while (WHILE > (32)) –2.46; ...

Diseñe una tabla con dos columnas: lexema y token. Complete la tabla con todos los lexemas hallados en el Análisis Léxico y las categorías a las que pertenecen.

Sea, en ANSI C, la siguiente función: void XX (void) { int a; double a; if (a > a) return 12; } Diseñe una tabla con dos columnas: lexema y token. Complete la tabla con todos los lexemas hallados en el Análisis Léxico de esta función.

Describa todos los errores que encuentra en la función del Ejercicio anterior. Para ello, construya una tabla que indique: Nº de Línea – Descripción del Error – Tipo de Error (Léxico, Sintáctico, Otro)

ANÁLISIS SINTÁCTICO

El Análisis Sintáctico trabaja con los tokens detectados durante el Análisis Léxico, es decir: la entrada para el Análisis Sintáctico son esos tokens.

Esta etapa de análisis sí conoce la sintaxis del Lenguaje de Programación y, en consecuencia, tendrá la capacidad de determinar si las construcciones que componen el programa son sintácticamente correctas. Sin embargo, no podrá determinar si el programa, en su totalidad, es sintácticamente correcto.

Durante el Análisis Sintáctico, la GIC que describe la sintaxis del lenguaje – por su propia independencia del contexto – no permite detectar si una variable fue declarada antes de ser utilizada, si la misma variable fue declarada varias veces y con diferentes tipos, como en el Ejercicio 5, y muchas otras situaciones erróneas.

Debemos diferenciar, claramente, los términos “Análisis Léxico” vs “Analizador Léxico” y “Análisis Sintáctico” vs “Analizador Sintáctico”. Por ello, y para evitar confusiones, a partir de ahora siempre llamaremos Scanner (palabra inglesa) al “Analizador Léxico” y Parser (palabra inglesa) al “Analizador Sintáctico”.

Dada una definición sintáctica formal, como la que brinda una GIC, el Parser recibe tokens del Scanner y los agrupa en unidades, tal como está especificado por las producciones de la GIC utilizada. Mientras se realiza el Análisis Sintáctico, el Parser verifica la corrección de la sintaxis y, si encuentra un error sintáctico, el Parser emite el diagnóstico correspondiente.

En la medida que las estructuras semánticas son reconocidas, el Parser llama a las correspondientes rutinas semánticas que realizarán el Análisis Semántico.

ANÁLISIS SEMÁNTICO

El Análisis Semántico realiza diferentes tareas, completando lo que hizo el Análisis Sintáctico. Una de estas importantes tareas es la “verificación de tipos”, para que cada operador trabaje sobre operandos permitidos según la especificación del Lenguaje de Programación.

Ejercicio: void XX (void) { int a; double a; if (a > a) return 12; } Informe cuáles son errores que serán detectados durante el Análisis Semántico.

Las rutinas semánticas llevan a cabo dos funciones:

1) Chequean la semántica estática de cada construcción; es decir, verifican que la construcción analizada sea legal y que tenga un significado. Verifican que las variables involucradas estén definidas, que los tipos sean correctos, etc.

2) Si la construcción es semánticamente correcta, las rutinas semánticas también hacen la traducción; es decir, generan el código para una Máquina Virtual que, a través de sus instrucciones, implementa correctamente la construcción analizada.

Ejemplo Sea una producción E->E+T. Cuando esta producción es reconocida por el Parser, éste llama a las rutinas semánticas asociadas para chequear la semántica estática (compatibilidad de tipos) y luego realizar las operaciones de traducción, generando las instrucciones para la Máquina Virtual que permitan realizar la suma.

Una definición completa de un LP debe incluir las especificaciones de su Sintaxis y de su Semántica. La Sintaxis se divide, normalmente, en componentes Independientes del Contexto y componentes Sensibles al Contexto.

Las GICs sirven para especificar la Sintaxis Independiente del Contexto. Sin embargo, no todo el programa puede ser descripto por una GIC: por caso, la compatibilidad de tipos y las reglas de alcance de las variables son Sensibles al Contexto.

Ejemplo En ANSI C, la sentencia a = b + c; es ilegal si cualquiera de las variables no estuviera declarada; esta es una restricción Sensible al Contexto.

Debido a estas limitaciones de las GICs, las restricciones Sensibles al Contexto son manejadas como situaciones semánticas. En consecuencia, el componente semántico de un LP se divide, habitualmente, en dos clases**: Semántica Estática**, que es la que nos interesa en este momento, y **Semántica en Tiempo de Ejecución.**

La Semántica Estática de un LP define las restricciones Sensibles al Contexto que deben cumplirse para que el programa sea considerado correcto. Algunas reglas típicas de Semántica Estática son:

1) Todos los identificadores deben estar declarados;

2) Los operadores y los operandos deben tener tipos compatibles;

3) Las rutinas (procedimientos y funciones) deben ser llamados con el número apropiado de argumentos. Ninguna de estas restricciones puede ser expresada mediante una GIC.

UN COMPILADOR SIMPLE

Esta sección describirá un proceso formado por cuatro etapas:

1º la definición de un Lenguaje de Programación muy simple;

2º la escritura de un programa fuente en este lenguaje;

3º el diseño de un compilador para realizar la traducción;

4º la obtención de un programa objeto equivalente.

DESCRIPCIÓN INFORMAL DEL LENGUAJE DE PROGRAMACIÓN MICRO

Fischer presenta un LP que denomina Micro. Es un lenguaje muy simple que está diseñado, específicamente, para poseer un LP concreto sobre el que se pueda analizar la construcción de un compilador básico.

Informalmente, Fischer lo define de esta manera: -

1. El único tipo de dato es entero. –
2. Todos los identificadores son declarados implícitamente y con una longitud máxima de 32 caracteres. –
3. Los identificadores deben comenzar con una letra y están compuestos de letras y dígitos. –
4. Las constantes son secuencias de dígitos (números enteros). –
5. Hay dos tipos de sentencias:
    a. Asignación ID := Expresión; Expresión es infija y se construye con identificadores, constantes y los operadores + y –; los paréntesis están permitidos.
    b. Entrada/Salida
        i. leer (lista de IDs);
        ii. escribir (lista de Expresiones); -
6. Cada sentencia termina con un "punto y coma" (;).
7. El cuerpo de un programa está delimitado por inicio y fin. - inicio, fin, leer y escribir son palabras reservadas y deben escribirse en minúscula.

Ejemplo El siguiente es un programa fuente en Micro: inicio leer (a,b); cc := a + (b-2); escribir (cc, a+4); fin

Diseñe una tabla con dos columnas: lexema y token. Suponga que hay seis tipos de tokens: palabraReservada, identificador, constante, operador, asignación y carácterPuntuación. Realice el Análisis Léxico del programa del Ejemplo y complete la tabla.

SINTAXIS DE MICRO – DEFINICIÓN PRECISA CON UNA GIC (BNF)

En esta sección, agregamos una definición más precisa de Micro. Describimos las producciones de una Gramática Léxica que define los posibles lexemas que se pueden encontrar en un programa Micro, y las producciones de una Gramática Sintáctica que permite precisar las construcciones válidas en Micro.

Gramática Léxica

<token>→ uno de <identificador> <constante> <palabraReservada> <operadorAditivo> <asignación> <carácterPuntuación>

<identificador> → <letra> {<letra o dígito>}

<constante> → <dígito> {dígito>}

<letra o dígito> → uno de <letra> <dígito>

<letra> → una de a-z A-Z

<dígito> → uno de 0-9

<palabraReservada> → una de inicio fin leer escribir

<operadorAditivo> → uno de + -

<asignación> → :=

<carácterPuntuación> → uno de ( ) , ;

Gramática Sintáctica

<programa> → inicio <listaSentencias> fin

<listaSentencias> → <sentencia> {<sentencia>}

<sentencia> -> <identificador> := <expresión> ; | leer (<listaIdentificadores>) ; | escribir (<listaExpresiones>) ;

<listaIdentificadores> → <identificador> {, <identificador>}

<listaExpresiones> → <expresión> {, <expresión>}

<expresión> → <primaria> {<operadorAditivo> <primaria>}

<primaria> → <identificador> | <constante> | (<expresión>)

Para mayor claridad de lectura, en la Gramática Sintáctica encontramos algunos tokens que permanecen con sus caracteres originales – las palabras reservadas, la asignación, los paréntesis y otros – tal como aparecen en un programa fuente en Micro. Sin embargo, al construir el compilador, cada uno de estos tokens tendrá su propio nombre, como se ve en la siguiente tabla:

En el Programa Fuente	Nombre del Token
Inicio	INICIO
Fin	FIN
Leer	LEER
Escribir	ESCRIBIR
:=	ASIGNACIÓN
(	PARENIZQUIERDO
)	PARENDERECHO
,	COMA
;	PUNTOYCOMA
+	SUMA
-	RESTA

LA ESTRUCTURA DE UN COMPILADOR Y EL LENGUAJE MICRO

El ANÁLISIS LÉXICO

es realizado por un módulo llamado Scanner. Este analizador lee, uno a uno, los caracteres que forman un programa fuente, y produce una secuencia de representaciones de tokens. Es muy importante tener en cuenta que el Scanner es una rutina que produce y retorna la representación del correspondiente token, uno por vez, en la medida que es invocada por el Parser.

Existen dos formas principales de implementar un Scanner:

A. A través de la utilización de un programa auxiliar tipo lex, en el que los datos son tokens representados mediante Expresiones Regulares, como ya hemos visto;
B. Mediante la construcción de una rutina basada en el diseño de un apropiado AFD; este método es el que utilizaremos en Micro.

El ANÁLISIS SINTÁCTICO

es realizado por un módulo llamado Parser. Este analizador procesa los tokens que le entrega el Scanner hasta que reconoce una construcción sintáctica que requiere un procesamiento semántico. Entonces, invoca directamente a la rutina semántica que corresponde. Algunas de estas rutinas semánticas utilizan, en sus procesamientos, la información de la representación de un token, como veremos más adelante.

Nótese que no existe un módulo independiente llamado Analizador Semántico, sino que el ANÁLISIS SEMÁNTICO se va realizando, en la medida que el Parser lo requiere, a través de las rutinas semánticas. Las rutinas semánticas realizan el Análisis Semántico y también producen

una salida en un lenguaje de bajo nivel para una Máquina Virtual; esto último forma parte de la etapa de Síntesis del compilador.

Existen dos formas fundamentales de Análisis Sintáctico:

A. el **Análisis Sintáctico Descendente** (conocido como top-down), que permite ser construido por un programador, y
B. el **Análisis Sintáctico Ascendente** (conocido como bottom-up), que requiere el auxilio de un programa especializado tipo yacc. Para Micro, construiremos un Parser basado en el Análisis Sintáctico Descendente y en el próximo capítulo explicaremos más sobre ambos métodos y sus diferencias.

En cuanto al **ANÁLISIS SEMÁNTICO**, es la tercera fase de análisis que existe en un compilador. Por un lado, el Análisis Semántico está inmerso dentro del Análisis Sintáctico, y, por otro lado, es el comienzo de la etapa de Síntesis del compilador. Lo veremos, con más precisión, al desarrollar el compilador Micro.

La **TABLA DE SÍMBOLOS** (TS) es una estructura de datos compleja que es utilizada para el almacenamiento de todos los identificadores del programa a compilar. Dependiendo del diseño del compilador, a veces también es utilizada para que contenga las palabras reservadas del LP, los literales-constante y las constantes numéricas que existen en el programa, pero éstas últimas en forma de secuencia de caracteres. Cada elemento de la TS está formada por una cadena y sus atributos. En el caso de Micro, la TS contendrá las palabras reservadas y los identificadores; cada entrada tendrá, como único atributo, un código que indique si la cadena representa una "palabra reservada" o "un identificador". En general, la TS es muy utilizada durante toda la compilación y, específicamente, en la etapa de Análisis por las rutinas semánticas.

UN ANALIZADOR LÉXICO PARA MICRO

Los lexemas que constituyen un programa fuente en este LP Micro son: identificadores, constantes, las cuatro palabras reservadas ya mencionadas, paréntesis izquierdo y derecho, el "punto y coma" (para terminar cada sentencia), la "coma" (para formar una lista), el símbolo de asignación (:=), y los operadores de suma (+) y de resta (-).

Atención con la siguiente propiedad: cada lexema está formado por la mayor secuencia posible de caracteres para el token correspondiente, y esto requiere, a veces, que se lea el primer carácter que no pertenece a ese lexema, es decir: el carácter que actúa como centinela.

Este carácter extra que se lee debe ser retornado al flujo de entrada porque será el primer carácter (o único carácter) del siguiente lexema. Para ello, en ANSI C se utiliza la función ungetc.

Si, en cualquier momento de la compilación, el primer carácter del flujo de entrada que lee el Scanner para detectar el próximo lexema no es un carácter válido para ningún lexema en el LP Micro, entonces se detectó un error léxico. En este caso, se despliega un mensaje de error y se repara el error para continuar con el proceso. La reparación más sencilla consiste en ignorar este carácter espúreo y reiniciar el Análisis Léxico a partir del carácter siguiente.

Otra tarea del Scanner es ignorar los espacios – ya sea espacios en blanco, tabulados o marcas de fin de línea – que actúen como separadores de lexemas dentro del texto del programa fuente.

El Scanner para Micro estará basado en la construcción de un AFD que reconoce los LRs que forman los tokens de Micro.

En cuanto al reconocimiento de las palabras reservadas, el primer problema es que son identificadores. Existen varias soluciones; veamos dos de ellas.

Una solución es que cada palabra reservada sea reconocida, por el AFD diseñado, mediante un estado final propio para cada una de ellas, independiente del estado final para el reconocimiento de los identificadores generales.

Otra solución es que las palabras reservadas sean colocadas, previamente, en las primeras entradas de la Tabla de Símbolos y con el atributo "reservada"; es decir, la TS ya contiene las palabras reservadas antes de comenzar el Análisis Léxico.

En la segunda solución, que es la que usaremos, el AFD estará diseñado con un único estado final para reconocer a todos los identificadores, incluyendo las palabras reservadas. Entonces, cuando un identificador es reconocido por el Scanner, lo busca en la TS. Si lo encuentra y tiene el atributo "reservada", sabe que es una palabra reservada y cuál es; caso contrario, y si el identificador detectado todavía no se encuentra almacenado en la TS, lo agrega con el atributo "id".

Ejercicio * Sea el siguiente programa Micro: inicio a := 23; escribir (a, b+2); fin a) Describa el contenido que queda en la Tabla de Símbolos una vez que el Análisis Léxico haya finalizado. b) ¿Cómo hace el Scanner de Micro para reconocer que el identificador inicio es una Palabra Reservada?

Por último, el Parser debe saber cuándo el Scanner le pasó el último token detectado en el flujo de entrada. Para ello, se crea un token que llamaremos fdt (fin del texto). En consecuencia, cuando el Scanner llegue al final del flujo que contiene al programa fuente, retornará el token fdt.

El Scanner será implementado como una función sin parámetros que retorna valores de tipo TOKEN. Por ello, definimos los siguientes tipos de tokens en ANSI C: typedef enum { INICIO, FIN, LEER, ESCRIBIR, ID, CONSTANTE, PARENIZQUIERDO, PARENDERECHO, PUNTOYCOMA, COMA, ASIGNACION, SUMA, RESTA, FDT } TOKEN;

Suponemos que el prototipo de la función que implementa al Scanner es: TOKEN Scanner (void);

Nota El compilador Micro es muy simple. No obstante, la función Scanner no puede retornar solo un valor por enumeración y pretender que el compilador pueda seguir desarrollando el trabajo necesario con su Parser y con las rutinas de análisis semántico. Si la función Scanner retorna ID, los otros analizadores necesitan saber a qué identificador se refiere. Lo mismo ocurre cuando Scanner retorna CONSTANTE: los otros analizadores necesitan conocer su valor. Por ello, supondremos que existen variables globales que contendrán esta información y que podrán ser accedidas por los otros analizadores.

EL AFD PARA IMPLEMENTAR EL SCANNER

Dado que el Scanner debe reconocer palabras de diferentes Lenguajes Regulares, es lógico suponer que el diseño de un AFD adecuado sea una buena solución.

Tal como expresamos antes, construiremos un AFD con un único estado final para todos los identificadores, incluyendo las palabras reservadas; de esta manera, el AFD será más compacto. Además, este AFD deberá realizar ciertas acciones en varios estados, como veremos a continuación.

El AFD será representado mediante su Tabla de Transiciones. En la TT hay una fila por cada estado del AFD y una columna por cada carácter o conjunto de caracteres que tienen el mismo estado de llegada para cualquier transición.

¿Qué ocurre en el caso de Micro?

1º Todas las letras son equivalentes en cuanto a su funcionalidad, por lo que llamaremos L a la columna correspondiente. Estos caracteres se detectan en ANSI C por medio de la función isalpha;

2º Todos los dígitos, que se detectan en ANSI C con isdigit, cumplen la misma función. Llamaremos D a la columna correspondiente;

3º Los caracteres +, –, (,), "punto y coma", :, = y "coma" tienen, cada uno, una función particular; por lo tanto, habrá una columna para cada uno de ellos;

4º El centinela de "fin de texto", al que llamamos fdt, tiene una función especial y muy importante, ya que le indica al Parser que se terminaron los tokens. Por lo tanto, deberá tener su propia columna en la TT del AFD;

5º El AFD ignorará los espacios.

En consecuencia habrá una columna, a la que llamaremos sp, para los tres caracteres especiales que ya analizamos anteriormente. Estos caracteres son reconocidos en ANSI C con la función isspace.

6º Finalmente, habrá una columna llamada otro para cualquier otro carácter no mencionado en los puntos anteriores. Estos caracteres no pueden intervenir en ningún lexema válido en Micro, por lo que su detección producirá un error léxico.

CONCLUSIÓN

El Parser invoca al Scanner cada vez que necesita un token. El Scanner analiza el flujo de entrada a partir de la posición que corresponde, detecta el próximo lexema en el Programa Fuente y retorna el token correspondiente, o tal vez encuentra un error léxico y le informa al Parser de esta situación anómala.

En la construcción del Scanner juega un papel muy importante el AFD diseñado para reconocer los lexemas. Además de este AFD, existen almacenamientos y funciones auxiliares que completan la implementación del Scanner en ANSI C; estos son:

– La función ungetc de ANSI C.

– El vector de caracteres externo que llamamos buffer, utilizado para almacenar los caracteres de los identificadores y los dígitos de las constantes en la medida que son reconocidos por el AFD.

– La función con prototipo void AgregarCaracter(int); que añade un carácter al buffer. – La función con prototipo TOKEN EsReservada(void); que, dado el identificador reconocido por el AFD y almacenado en el buffer, retorna el token que le corresponde (ID o el código de alguna de las cuatro palabras reservadas).

– La función con prototipo void LimpiarBuffer(void); que "vacía" el buffer para el próximo uso. – La función feof de ANSI C, para detectar el centinela del texto, el fdt, y así saber que todos los caracteres del Programa Fuente fueron procesados. – La función fgetc de ANSI C, para leer cada carácter del flujo de entrada.

– Las funciones ANSI C isspace, isalpha, isalnum e isdigit.

ANÁLISIS LÉXICO, ANÁLISIS SINTÁCTICO Y ANÁLISIS SEMÁNTICO

Los tres elementos que forman la etapa de compilación. Cómo la llamada Tabla de Símbolos interactúa con todos ellos y, además.

```
/* Compilador del Lenguaje Micro (Fischer)       */
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define NUMESTADOS 15
#define NUMCOLS 13
#define TAMLEX 32+1
#define TAMNOM 20+1
/*****************Declaraciones Globales************************/
FILE * in;
typedef
enum {
 INICIO, FIN, LEER, ESCRIBIR, ID, CONSTANTE, PARENIZQUIERDO, PARENDERECHO,
PUNTOYCOMA, COMA, ASIGNACION, SUMA, RESTA, FDT, ERRORLEXICO
} TOKEN; typedef
struct {     char
identifi[TAMLEX];
    TOKEN t;       /* t=0, 1, 2, 3 Palabra Reservada, t=ID=4 Identificador (ver enum)
*/     } RegTS;
RegTS TS[1000] = { {"inicio", INICIO}, {"fin", FIN}, {"leer", LEER}, {"escribir", ESCRIBIR}, {"$", 99}
};

typedef struct{
TOKEN clase;     char
nombre[TAMLEX];
    int valor;
    } REG_EXPRESION;

char buffer[TAMLEX];
TOKEN tokenActual;
int flagToken = 0;
```

```
/*************************Scanner*********************************/
TOKEN scanner()
{
TOKEN scanner()

{
 int tabla[NUMESTADOS][NUMCOLS] =
            L  D  +  -  (  )  ,  ;  :  =  EOF ´ ´ OTRO
0         {{ 1,  3 ,  5 ,  6 ,  7 ,  8 ,  9 , 10, 11, 14, 13,  0, 14 },
 1        {  1,  1 ,  2 ,  2 ,  2 ,  2 ,  2 ,  2 ,  2 ,  2 ,  2 ,   2 },
 2 ID     { 14, 14, 14, 14, 14, 14, 14, 14, 14 , 14, 14, 14, 14 },
3          {  4,  3 ,  4 ,  4 ,  4 ,  4 ,  4 ,  4 ,  4,  4,  4,  4,  4 },
4 CTE    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
5 +       { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
6 -       { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
7 (       { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
8 )       { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
9 ,       { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
10 ;      { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
11        { 14, 14, 14, 14, 14, 14, 14, 14, 14, 12, 14, 14, 14 },
12 ASIG{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
13 fdt    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
14 Err    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 } };

int car;
int col;
int estado = 0;
int i = 0;  do {   car =
fgetc(in);
col = columna(car);
estado = abla[estado][col];
  if ( col != 11 )  { //si es espacio no lo agrega al buffer
     buffer[i] = car;    i++;
    }
 }
 while ( !estadoFinal(estado) && !(estado == 14) );
buffer[i] = '\0'; //complete la cadena
switch ( estado )
 {
  case 2 : if ( col != 11 ){              //si el carácter espureo no es blanco…
               ungetc(car, in);           // lo retorna al flujo
               buffer[i-1] = '\0';
              }
              return ID;
  case 4 : if ( col != 11 )  {
               ungetc(car, in);
               buffer[i-1] = '\0';
              }
         return CONSTANTE;
case 5 : return SUMA;   case 6
: return RESTA;   case 7 :
```

```
return PARENIZQUIERDO;
case 8 : return
PARENDERECHO;   case 9 :
return COMA;   case 10 :
return PUNTOYCOMA;   case
12 : return ASIGNACION;   case
13 : return FDT;
 case 14 : return ERRORLEXICO;
 }
 return 0;
}
```

```
int estadoFinal(int e){
 if ( e == 0 || e == 1 || e == 3 || e == 11 || e == 14 ) return 0;  return 1;
}
```

```
int columna(int c){  if ( isalpha(c) ) return 0;  if ( isdigit(c) ) return 1;  if ( c == '+' ) return 2;  if ( c == '-' ) return 3;  if ( c == '(' ) return 4;  if ( c == ')' ) return 5;  if ( c == ',' ) return 6;  if ( c == ';' ) return 7;  if ( c == ':' ) return 8;  if ( c == '=' ) return 9;  if ( c == EOF ) return 10;  if ( isspace(c) ) return 11;
 return 12;
}
```

```
/*************Fin Scanner*****************************
```