**UN PARSER PARA MICRO**

Recordemos que el Parser trabaja con tokens y es guiado por la GIC que describe la sintaxis del LP, Micro en este caso. Estos tokens le son provistos uno a uno por el Scanner, cada vez que el Parser lo invoca.

Antes de desarrollar el Parser, recordemos la Gramática Sintáctica

<programa> -> **inicio** <listaSentencias> **fin**

<listaSentencias> -> <sentencia> {<sentencia>}

<sentencia> -> <identficador> **:=** <expresión>**;** |

**leer (** <listaIdentificadores> **);** |

**escribir (** <listaExpresiones> **);**

<listaIdentificadores> -> <identificador> {**,** <identificador>}

<listaExpresiones> -> <expresión> {**,** <expresión>}

<expresión> -> <primaria> {<operadorAditivo> <primaria>}

<primaria> -> <identificador> | <constante> |

**(** <expresión> **)**

Extenderemos y modificaremos esta GIC para que refleje exactamente cómo será utilizada por el Parser. En primer lugar, agregaremos una producción global que defina la totalidad del programa que es analizado, recordando que el programa fuente es un texto que termina con un centinela. En segundo lugar, utilizaremos los nombres a los tokens

typedef enum {

INICIO, FIN, LEER, ESCRIBIR, ID, CONSTANTE, PARENIZQUIERDO,

PARENDERECHO, PUNTOYCOMA, COMA, ASIGNACION, SUMA, RESTA, FDT

} TOKEN;

En consecuencia la GIC actualizada que utilizará el Parser será:

<objetivo> -> <programa> FDT (Esta es la producción global que se agrega)

<programa> -> INICIO <listaSentencias> FIN

<listaSentencias> -> <sentencia> {<sentencia>}

<sentencia> -> ID ASIGNACIÓN <expresión> PUNTOYCOMA |

LEER PARENIZQUIERDO <listaIdentificadores> PARENDERECHO PUNTOYCOMA |ESCRIBIR PARENIZQUIERDO <listaExpresiones> PARENDERECHO PUNTOYCOMA

<listaIdentificadores> -> ID {COMA ID}

<listaExpresiones> -> <expresión> {COMA <expresión>}

<expresión> -> <primaria> {<operadorAditivo> <primaria>}

<primaria> -> ID | CONSTANTE |

PARENIZQUIERDO <expresión> PARENDERECHO

<operadorAditivo> -> *uno de* SUMA RESTA

En general, no es común que se realice esta transformación. En esta

oportunidad la hacemos para afianzar este concepto: los terminales de la GIC utilizada en el Análisis Sintáctico son los tokens que retorna el Scanner y no los caracteres o las secuencias de caracteres – según corresponda – que se encuentran en el Programa Fuente.

Utilizaremos una técnica de Análisis Sintáctico muy conocida, llamada ANÁLISIS SINTÁCTICO DESCENDENTE RECURSIVO (ASDR). Su nombre se debe a que utiliza rutinas, que pueden ser recursivas, cuya ejecución va “construyendo” un árbol de análisis sintáctico (AAS) para la secuencia de entrada – formada por tokens – que debe reconocer.

Un Árbol de Análisis Sintáctico parte del axioma de una GIC y representa la derivación de una construcción (declaración, sentencia, expresión, bloque y hasta el programa completo). Un AAS tiene las siguientes propiedades:

1. La raíz del AAS está etiquetada con el axioma de la GIC.
2. Cada hoja está etiquetada con un token. Si se leen de izquierda a derecha, las hojas representan la construcción derivada.
3. Cada nodo interior está etiquetado con un noterminal.
4. Si A es un noterminal y X1, X2,...,Xm son sus hijos de izquierda a derecha, entonces existe la producción A->X1...Xm, donde cada Xi puede ser un noterminal o un terminal.

El ASDR es una de las técnicas más simples de las utilizadas en la construcción de un compilador, pero no se puede utilizar con cualquier GIC, como veremos en el próximo capítulo.

La idea básica del ASDR es que cada noterminal de la Gramática Sintáctica tiene asociado una rutina de Análisis Sintáctico que puede reconocer cualquier secuencia de tokens generada por ese noterminal; esta rutina se implementa como un procedimiento (lo llamaremos PAS, Procedimiento de Análisis Sintáctico). En ANSI C, los PAS se construyen como funciones void.

Como dijimos, cada PAS implementa un noterminal de la Gramática Sintáctica. La estructura de cada PAS sigue fielmente el desarrollo del lado derecho de la producción que implementa, es decir: dentro de un PAS, tanto los noterminales como los terminales del lado derecho de la producción deben ser procesados y en el orden en que aparecen. Esto se realiza de la siguiente manera:

1. Si se debe procesar un noterminal <A>, invocamos al PAS correspondiente, que, por convención, lo llamaremos con el mismo nombre, A. Esta llamada puede ser recursiva y de ahí el nombre de este tipo de Análisis Sintáctico.
2. Para procesar un terminal t, invocaremos al procedimiento llamado Match con argumento t.

Este procedimiento Match con argumento t, es decir, Match(t), invoca al Scanner para obtener el próximo token del flujo de tokens de entrada. Si el token obtenido por el Scanner es t, es decir, coincide con el argumento con el cual se invoca a Match, entonces todo es correcto porque hubo concordancia (eso significa *match*); si es así, el token es guardado en una variable global llamada tokenActual.

En cambio, si el token obtenido por el Scanner no coincide con el argumento t, entonces se ha producido un Error Sintáctico; se debe emitir un mensaje de error y tener en cuenta esta situación porque el proceso de compilación ya no puede ser correcto. Además, se debe realizar algún tipo de “reparación” del error para poder continuar con el Análisis Sintáctico.

Para ver cómo trabajan estos PAS (procedimientos de Análisis Sintáctico), desarrollaremos algunos de ellos. En todos los casos, veremos cómo estos procedimientos reflejan las definiciones dadas por las producciones de la Gramática Sintáctica actualizada y descripta en la página anterior.

El Parser comienza su actividad cuando se invoca al PAS que corresponde al noterminal

<objetivo>, cuya producción hemos agregado en la GIC actualizada:

void Objetivo (void) {

/\* <objetivo> -> <programa> FDT \*/

Programa();

Match(FDT);

}

Esto significa: para determinar que el Análisis Sintáctico de un programa en Micro ha finalizado, debemos hallar la correspondencia con la secuencia de tokens generada por <programa> y seguida de FDT, el token-centinela que agregamos para indicar la terminación del programa fuente.

Siguiendo el orden de las producciones de la GIC, para el noterminal <programa> tenemos un PAS que se puede construir fácilmente a partir de su producción:

<programa> -> INICIO <listaSentencias> FIN

Para el noterminal <listaSentencias>, el PAS es un poco más complicado porque hay una

secuencia opcional de sentencias; es decir, <listaSentencias> es una o más sentencias, como se ve en la definición de este noterminal:

<listaSentencias> -> <sentencia> {<sentencia>}

Además, existen tres tipos de sentencias, como se aprecia en la definición de <sentencia>:

<sentencia> -> ID ASIGNACIÓN <expresión> PUNTOYCOMA |

LEER PARENIZQUIERDO <listaIdentificadores> PARENDERECHO PUNTOYCOMA |

ESCRIBIR PARENIZQUIERDO <listaExpresiones> PARENDERECHO PUNTOYCOMA

Estos tres tipos de sentencias tienen una importante particularidad: cada uno comienza con un token

diferente (ID, LEER y ESCRIBIR). Por lo tanto, conociendo el primer token, ya sabemos cuál es el tipo de sentencia que corresponde. Este enfoque no funciona para todas las GICs pero sí lo hace para la gramática LL(1) como la que estamos utilizando y que analizaremos más extensamente en el próximo capítulo.

Sea ProximoToken una función que retorna el próximo token a ser correspondido. Como se ve en las producciones de <sentencia>, el próximo token debería ser ID, LEER o ESCRIBIR, cada uno de los cuales es el primer token en una de las tres posibles sentencias, como se indicó anteriormente.

Entonces, si el token retornado por ProximoToken es uno de los tres mencionados, el Parser tratará de reconocer una sentencia de la secuencia opcional definida en el lado derecho de <listaSentencias>. Caso contrario, concluiremos que la lista completa de sentencias fue procesada.

Esto se ve más claramente en el desarrollo del siguiente PAS para el noterminal <listaSentencias>:

void ListaSentencias (void) {

/\* <listaSentencias> -> <sentencia> {<sentencia>} \*/

Sentencia(); /\* la primera de la lista de sentencias \*/

while (1) { /\* un ciclo indefinido \*/

switch (ProximoToken()) {

case ID: case LEER: case ESCRIBIR: /\* detectó token correcto \*/

Sentencia(); /\* procesa la secuencia opcional \*/

break;

default:

return;

} /\* fin switch \*/

}

}

Al definir el PAS correspondiente al noterminal sentencia nos encontramos con el problema que hay varias producciones que tienen al noterminal sentencia en su lado izquierdo. En consecuencia, debemos decidir cuál de las producciones deberá ser procesada. En el caso de Micro, esta situación se simplifica porque cada producción para el noterminal sentencia comienza con un terminal diferente y, por lo tanto, con solo conocer cuál es el próximo terminal ya sabemos que producción aplicar. Esta es una propiedad muy ventajosa de las gramáticas LL(1).

El PAS para el noterminal sentencia puede ser construido de esta manera:

void Sentencia(void) {

TOKEN tok = ProximoToken();

switch (tok) {

case ID: /\* <sentencia> -> ID := <expresion>; \*/

Match(ID); Match(Asignación>); Expresión(); Match(PUNTOYCOMA);

break;

case LEER: /\* <sentencia> -> LEER ( <listaIdentificadores> ); \*/

Match(LEER); Match(PARENIZQUIERDO); ListaIdentificadores();

Match(PARENDERECHO); Match(PUNTOYCOMA);

break;

case ESCRIBIR: /\* <sentencia> -> ESCRIBIR (<listaExpresiones>); \*/

Match(ESCRIBIR); Match(PARENIZQUIERDO); ListaExpresiones();

Match(PARENDERECHO); Match(PUNTOYCOMA);

break;

default:

ErrorSintactico(tok); break;

}

}

Desarrollemos otro PAS, con una dificultad diferente a los anteriores, aunque siempre reflejando, en su construcción, la estructura de la correspondiente producción de la GIC:

void Expresion (void) {

/\* <expresion> -> <primaria> {<operadorAditivo> <primaria>} \*/

TOKEN t;

Primaria();

for (t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken())

{

OperadorAditivo(); Primaria();

}

}

Por último, veamos otro PAS en el que aparece un llamado al procedimiento ErrorSintactico:

void OperadorAditivo (void) {

/\* <operadorAditivo> -> *uno* de SUMA RESTA \*/

TOKEN t = ProximoToken();

if (t == SUMA || t == RESTA)

Match(t);

else

ErrorSintactico(t);

}

De esta manera, hemos comenzado con la construcción del Parser. Hasta el momento, se trata, básicamente, de un conjunto de procedimientos que denominamos PAS y que implementan las definiciones dadas por las producciones de la Gramática Sintáctica de Micro. Por supuesto, se ha visto que también hay un grupo de funciones auxiliares que contribuyen en la construcción del Parser.

**LA ETAPA DE TRADUCCIÓN, LAS RUTINAS SEMÁNTICAS Y LA AMPLIACIÓN DE LOS PAS**

La etapa de traducción de un programa en Micro, que pertenece a lo que se denomina Síntesis dentro de la compilación, consistirá en la generación de código para una máquina virtual (MV) Supongamos que esta MV tiene instrucciones con el siguiente formato:

OP A,B,C donde OP es el código de operación, A y B designan a los operandos, y C especifica la dirección donde se almacena el resultado de la operación. Los operandos A y B pueden ser nombres de variables o constantes enteras. Para algunos códigos de operación, A o B o C pueden no ser usadas.

El formato de la salida será una cadena de caracteres.

*Ejemplo 7*

He aquí una instrucción de esta máquina virtual:

Declara A,Entera

Esta instrucción, que declara a la variable A como entera, tiene Declara como código de operación y dos operandos; no utiliza el tercer operando.

Gran parte de la traducción es realizada por rutinas semánticas llamadas por el Parser. A la Gramática Sintáctica conocida se le pueden agregar símbolos de acción para especificar cuándo se debe realizar un procesamiento semántico.

Los símbolos de acción, indicados como #nombre, se pueden ubicar en cualquier lugar del lado derecho de una producción. A cada símbolo de acción le corresponde una rutina semántica implementada como una función o un procedimiento.

*Ejemplo*

A un símbolo de acción llamado #sumar le corresponderá la rutina semántica Sumar.

Cuando se crea un PAS, las invocaciones a las rutinas semánticas son insertadas en las posiciones designadas por los símbolos de acción.

Los símbolos de acción no forman parte de la sintaxis especificada por una GIC, sino que le agregan “comentarios” a esta GIC para indicar cuándo se necesitan ejecutar las acciones semánticas correspondientes.

*Ejemplo 9*

La producción para <operadorAditivo> con el agregado del símbolo de acción que le corrsponde, sería:

<operadorAditivo> -> SUMA #procesar\_op | RESTA #procesar\_op

**INFORMACIÓN SEMÁNTICA**

Al diseñar las rutinas semánticas, es muy importante la especificación de los datos sobre los cuales operan y la información que producen.

El enfoque utilizado en este caso es asociar un registro semántico a cada símbolo gramatical, tanto noterminal (por ejemplo, <expresión>) como terminal (por ejemplo, ID). Como caso especial, habrá ciertos símbolos con registros semánticos nulos, sin datos.

*Ejemplo 10*

El símbolo PUNTOYCOMA no requiere un registro semántico.

Como ejemplo representativo de la situación explicada en el párrafo anterior, consideremos la siguiente producción, ampliada con el agregado del correspondiente símbolo de acción:

<expresión> -> <primaria> + <primaria> #sumar

Por cada aparición del noterminal <primaria> en el lado derecho de esta producción, se generará un registro semántico. Cada registro semántico contendrá datos sobre cada operando, como, por ejemplo, dónde está almacenado y cuál es su valor.

Cuando la función Sumar es invocada como resultado de la aparición del símbolo de acción #sumar, se le deben pasar estos registros semánticos como argumentos. Como resultado, Sumar producirá un nuevo registro semántico correspondiente al noterminal <expresión> con la información necesaria.

*Ejemplo 11*

En el caso de Micro, tenemos solo estos dos registros semánticos:

1. REG\_OPERACION, que solo contendrá el valor del token SUMA o RESTA.

2. REG\_EXPRESION, que contendrá el tipo de expresión y el valor que le corresponde; este valor puede ser una cadena (para el caso de un identificador) o un número entero (para el caso de una constante). La cadena será almacenada en un vector de caracteres previamente definido y tendrá una longitud máxima de 32 caracteres, límite que se informa en la sección de los identificadores.

**LA GRAMÁTICA SINTÁCTICA DE MICRO CON LOS SÍMBOLOS DE ACCIÓN**

En primer lugar, agregamos una nueva producción:

<identificador> -> ID #procesar\_id

Esta producción es muy útil porque ID aparece en varios contextos diferentes de la GIC, y necesitamos llamar a la función Procesar\_ID inmediatamente después que el Parser haya encontrado el correspondiente ID, para acceder a los caracteres que se encuentran en el buffer y así construir el registro semántico apropiado.

Además, se utilizarán algunas funciones auxiliares en la construcción del compilador:

1. Generar: una función que recibe cuatro argumentos que son cadenas, que corresponden al código de operación y a los tres operandos de cada instrucción de la MV; esta función producirá la correspondiente instrucción en el flujo de salida.
2. Extraer: una función tal que dado un registro semántico, retorna la cadena que contiene.

Esta cadena puede ser un identificador, un código de operación, representar una constante antes de ser convertida a número entero, etc.

Dado que la Tabla de Símbolos de Micro es muy simple porque el LP es muy sencillo, las rutinas que se necesitan para operar con ella son solo dos:

1. Buscar: una función tal que dada una cadena que representa a un identificador, determina si ya se encuentra en la Tabla de Símbolos.
2. Colocar: almacena una cadena en la Tabla de Símbolos.

Una rutina auxiliar utilizada por varias rutinas semánticas es Chequear. He aquí su implementación

en ANSI C:

void Chequear (string s) {

if (! Buscar(s)) { /\* ¿la cadena está en la Tabla de Símbolos? No: \*/

Colocar(s); /\* almacenarla, es el nombre de una variable \*/

Generar("Declara", s, "Entera", ""); /\* genera la instrucción \*/

}

}

He aquí tres funciones auxiliares en ANSI C que son necesarias para definir las rutinas semánticas que corresponden a los símbolos de acción de Micro:

void Comenzar (void); /\* inicializaciones semánticas \*/

void Terminar (void) {

/\* genera la instrucción para terminar la ejecución del programa \*/

Generar("Detiene", "", "", "");

}

void Asignar (REG\_EXPRESION izquierda, REG\_EXPRESION derecha) {

/\* genera la instrucción para la asignación \*/

Generar("Almacena", Extraer(derecha), izquierda.nombre, "");

}

La Gramática Sintáctica para Micro, ampliada con los símbolos de acción correspondientes, será:

<objetivo> -> <programa> FDT #terminar

<programa> -> #comenzar **inicio** <listaSentencias> **fin**

<listaSentencias> -> <sentencia> {<sentencia>}

<sentencia> -> <identificador> **:=** <expresión> #asignar **;** |

**read (** <listaIdentificadores> **);** |

**write (** <listaExpresiones> **);**

<listaIdentificadores> -> <identificador> #leer\_id

{, <identificador> #leer\_id}

<listaExpresiones> -> <expresión> #escribir\_exp

{, <expresión> #escribir\_exp}

<expresión> -> <primaria> {<operadorAditivo> <primaria> #gen\_infijo}

<primaria> -> <identificador> |

CONSTANTE #procesar\_cte |

**(** <expresión> **)**

<operadorAditivo> -> SUMA #procesar\_op | RESTA #procesar\_op

<identificador> -> ID #procesar\_id

**ALGUNAS RUTINAS SEMÁNTICAS**

Veamos, ahora, algunas de las rutinas semánticas utilizadas en el compilador de Micro.

void Leer (REG\_EXPRESION in) {

/\* genera la instrucción para leer \*/

Generar("Read", in.nombre, "Entera", "");

}

REG\_EXPRESION ProcesarCte (void) {

/\* convierte cadena que representa número a número entero y construye un registro semántico \*/

REG\_EXPRESION t;

t.clase = CONSTANTE;

sscanf (buffer, "%d", &t.valor);

return t;

}

Otras rutinas semánticas interesantes son:

void Escribir (REG\_EXPRESION out) {

Generar("Write", Extraer(out), "Entera", "");

}

REG\_EXPRESION ProcesarId (void) {

/\* Declara ID y construye el correspondiente registro semántico \*/

REG\_EXPRESION t;

Chequear(buffer); t.clase = ID;

strcpy(t.nombre, buffer);

return t;

}

**PROCEDIMIENTO DE ANÁLISIS SINTÁCTICO (PAS) CON SEMÁNTICA INCORPORADA**

Recordemos uno de los PAS que hemos descripto oportunamente:

void Expresion (void) {

/\* <expresion> -> <primaria> {<operadorAditivo> <primaria>} \*/

token t;

Primaria();

for (t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken())

{

OperadorAditivo(); Primaria();

}

}

Si ahora le agregamos el procesamiento semántico, vemos cómo se modifica este procedimiento y cuál es su contenido definitivo:

void Expresion (REG\_EXPRESION \*resultado) {

/\* <expresión> -> <primaria> {<operadorAditivo> <primaria> #gen\_infijo}\*/

REG\_EXPRESION operandoIzq, operandoDer;

REG\_OPERACION op;

token t;

Primaria(&operandoIzq);

for (t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken())

{

OperadorAditivo(&op); Primaria(&operandoDer);

operandoIzq = GenInfijo(operandoIzq, op, operandoDer);

}

\*resultado = operandoIzq;

}

En el procedimiento Expresion se invoca a la función GenInfijo; esta función tiene la siguiente estructura general y realiza la siguiente actividad:

REG\_EXPRESION GenInfijo (REG\_EXPRESION e1, REG\_OPERACION op,

REG\_EXPRESION e2) {

/\* Genera la instrucción para una operación infija y construye un

registro semántico con el resultado \*/

. . .

}

**ANÁLISIS LÉXICO, ANÁLISIS SINTÁCTICO Y ANÁLISIS SEMÁNTICO**

Luego que se mostrara una introducción al diseño de un compilador y algunos detalles de cómo se construye uno para un LP muy simple, nos ocuparemos ahora de los tres ingredientes que forman la etapa de compilación conocida como ANÁLISIS: el Análisis Léxico, el Análisis Sintáctico y el Análisis Semántico. También veremos cómo la llamada Tabla de Símbolos interactúa con todos ellos y, además, ciertos conceptos importantes sobre la misma.

**INTRODUCCIÓN A LA TABLA DE SÍMBOLOS**

Dado que la Tabla de Símbolos interactúa con las tres fases de análisis, es conveniente que, primero, veamos algunos conceptos fundamentales de la misma.

La Tabla de Símbolos, también conocida como Diccionario, es un conjunto de estructuras de datos que se utiliza, como mínimo, para contener todos los identificadores del programa fuente que se está compilando, junto con los atributos que posee cada identificador.

En el caso que estos identificadores sean nombres de variables, sus atributos son: tipo y ámbito (la parte del programa donde tiene validez). En el caso que los identificadores sean nombres de rutinas (funciones, procedimientos), algunos atributos son: cantidad de parámetros y tipo de cada uno, métodos de transferencia, tipo del valor que retorna. Estos atributos se codifican mediante números enteros o valores por enumeración.

Muchos diseñadores de compiladores también usan la Tabla de Símbolos para almacenar en ella todas las palabras reservadas del Lenguaje de Programación; cada una de ellas tendrá el atributo que indica que es una palabra reservada.

*Ejemplo 1*

Sea la siguiente función en ANSI C:

int XX (double a) {

char s[12];

double b;

b = a + 1.2;

if (b > 4.61) return 1;

else return 0;

}

Esta función posee diferentes identificadores: palabras reservadas, nombre de una función y nombres de variables.

El identificador XX debe poseer atributos que indiquen que es una función, que retorna un valor de tipo int, que tiene un parámetro y que ese parámetro es de tipo double. Podríamos representarlo de esta manera: (XX, *funcion, int, 1, double*). Nótese que, en esta representación, XX es una secuencia de caracteres porque es el identificador, pero los restantes elementos son los atributos de XX y se representan mediante números enteros o valores por enumeración.

**EL ANÁLISIS LÉXICO**

El Análisis Léxico, realizado por el Scanner, es el proceso que consiste en recorrer el flujo de caracteres que forman el Programa Fuente, detectar los lexemas que componen este programa, y traducir la secuencia de estos lexemas en una secuencia de tokens cuya representación es más útil para el resto del compilador.

Flujo de Caracteres -> Secuencia de Lexemas -> Secuencia de Tokens

Debemos tener en cuenta que el Análisis Léxico es un proceso que se realiza sobre palabras (los lexemas) de Lenguajes Regulares (los tokens). Cuando estos LRs son infinitos, sus lexemas son obtenidos recién cuando se detecta un carácter espúreo llamado centinela. En cambio, si los LRs son finitos, pueden requerir o no un centinela para obtener el lexema correspondiente.

*Ejemplo 2*

Sea la siguiente función en ANSI C:

int Abcd (void) {

int a, b = 4;

a = b + 18;

return a;

}

1º caso: El nombre de la función, Abcd, es un identificador. Como pertenece a un LR infinito, requiere un centinela para saber cuándo termina; en este caso, el centinela es el carácter *espacio*.

2º caso: El carácter {, que pertenece a un LR (token) finito, no requiere un centinela porque, en ANSI C, no hay lexemas que comiencen con { y que luego puedan tener más caracteres.

3º caso: Pero, ¿qué ocurre con el operador +? Este lexema también pertenece a un LR finito, el lenguaje de los operadores. Este LR contiene otros dos operadores que comienzan con +: el operador incremento (++) y el operador de asignación combinado con la suma (+=). En consecuencia, cuando el Scanner encuentra el carácter + no puede afirmar que encontró un lexema; necesita conocer la existencia de un centinela para confirmarlo.

/\* 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;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Prototipos de Funciones\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

TOKEN scanner(); // el scanner

int columna(int c);

int estadoFinal(int e);

void Objetivo(void); // del PAS

void Programa(void);

void ListaSentencias(void);

void Sentencia(void);

void ListaIdentificadores(void);

void Identificador(REG\_EXPRESION \* presul);

void ListaExpresiones(void);

void Expresion(REG\_EXPRESION \* presul);

void Primaria(REG\_EXPRESION \* presul);

void OperadorAditivo(char \* presul);

REG\_EXPRESION ProcesarCte(void);

REG\_EXPRESION ProcesarId(void);

char \* ProcesarOp(void);

void Leer(REG\_EXPRESION in);

void Escribir(REG\_EXPRESION out);

REG\_EXPRESION GenInfijo(REG\_EXPRESION e1, char \* op, REG\_EXPRESION e2);

void Match(TOKEN t);

TOKEN ProximoToken();

void ErrorLexico();

void ErrorSintactico();

void Generar(char \* co, char \* a, char \* b, char \* c);

char \* Extraer(REG\_EXPRESION \* preg);

int Buscar(char \* id, RegTS \* TS, TOKEN \* t);

void Colocar(char \* id, RegTS \* TS);

void Chequear(char \* s);

void Comenzar(void);

void Terminar(void);

void Asignar(REG\_EXPRESION izq, REG\_EXPRESION der);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Programa Principal\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(int argc, char \* argv[])

{

TOKEN tok;

char nomArchi[TAMNOM];

int l;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Se abre el Archivo Fuente\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// verifica errores posibles

if ( argc == 1 ) {

printf("Debe ingresar el nombre del archivo fuente (en lenguaje Micro) en la linea de comandos\n"); return -1; }// no puso nombre de archivo fuente

if ( argc != 2 ) {

printf("Numero incorrecto de argumentos\n"); return -1; }//los argumentos deben ser 2

strcpy(nomArchi, argv[1]);

l = strlen(nomArchi);

if ( l > TAMNOM ) {

printf("Nombre incorrecto del Archivo Fuente\n"); return -1; }

// requiere para compilar un archivo de extensión.m archivo.m

if ( nomArchi[l-1] != 'm' || nomArchi[l-2] != '.' ) {

printf("Nombre incorrecto del Archivo Fuente\n"); return -1; }

if ( (in = fopen(nomArchi, "r") ) == NULL ) {

printf("No se pudo abrir archivo fuente\n"); return -1;//no pudo abrir archivo

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Inicio Compilacion\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Objetivo();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Se cierra el Archivo Fuente\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

fclose(in);

return 0;

}

/\*\*\*\*\*\*\*\*\*\*Procedimientos de Analisis Sintactico (PAS) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*Procedimientos de Analisis Sintactico (PAS) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Objetivo(void)

{

/\* <objetivo> -> <programa> FDT #terminar \*/

Programa();

Match(FDT);

Terminar();

}

void Programa(void)

{

/\* <programa> -> #comenzar INICIO <listaSentencias> FIN \*/

Comenzar();//de inicio semántico en caso de corresponder

Match(INICIO);

ListaSentencias();

Match(FIN);

}

void ListaSentencias(void)

{

/\* <listaSentencias> -> <sentencia> {<sentencia>} \*/

Sentencia();

while ( 1 ) {// se repite hasta que retorna al no encontrar sentencia {<sentencia>}

switch ( ProximoToken() ) {

case ID : case LEER : case ESCRIBIR :

Sentencia();

break;

default : return; //si no es sentencia termina la funcion

}// fin del switch

}// fin del while

}// fin funcion

void Sentencia(void) {

TOKEN tok = ProximoToken();

REG\_EXPRESION izq, der;

//typedef struct{ TOKEN clase; char nombre[TAMLEX]; int valor; } REG\_EXPRESION;

switch ( tok ) {

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

Identificador(&izq);

Match(ASIGNACION);

Expresion(&der);

Asignar(izq, der); //genera instrucción de asignacion

Match(PUNTOYCOMA);

break;

case LEER : /\* <sentencia> -> LEER ( <listaIdentificadores> ) \*/

Match(LEER);

Match(PARENIZQUIERDO);

ListaIdentificadores();

Match(PARENDERECHO);

Match(PUNTOYCOMA);

break;

case ESCRIBIR : /\* <sentencia> -> ESCRIBIR ( <listaExpresiones> ) \*/

Match(ESCRIBIR);

Match(PARENIZQUIERDO);

ListaExpresiones();

Match(PARENDERECHO);

Match(PUNTOYCOMA);

break;

default : return;

}

}

void ListaIdentificadores(void) {

/\* <listaIdentificadores> -> <identificador> #leer\_id {COMA <identificador> #leer\_id} \*/

TOKEN t;

REG\_EXPRESION reg;

Identificador(&reg);

Leer(reg);

for ( t = ProximoToken(); t == COMA; t = ProximoToken() ) {

Match(COMA);

Identificador(&reg);

Leer(reg);

}

}

void Identificador(REG\_EXPRESION \* presul) {

/\* <identificador> -> ID #procesar\_id \*/

Match(ID);

\*presul = ProcesarId();//rutina semantica

}

void ListaExpresiones(void) {

/\* <listaExpresiones> -> <expresion> #escribir\_exp {COMA <expresion> #escribir\_exp} \*/

TOKEN t;

REG\_EXPRESION reg;

Expresion(&reg);

Escribir(reg);

for ( t = ProximoToken(); t == COMA; t = ProximoToken() ) {

Match(COMA);

Expresion(&reg);

Escribir(reg);

}

}

void Expresion(REG\_EXPRESION \* presul){

/\* <expresion> -> <primaria> { <operadorAditivo> <primaria> #gen\_infijo } \*/

REG\_EXPRESION operandoIzq, operandoDer;

char op[TAMLEX];

TOKEN t;

Primaria(&operandoIzq);

for ( t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken() ) {

OperadorAditivo(op);

Primaria(&operandoDer);

operandoIzq = GenInfijo(operandoIzq, op, operandoDer);

}

\*presul = operandoIzq;

}

void Primaria(REG\_EXPRESION \* presul) {

TOKEN tok = ProximoToken();

switch ( tok ) {

case ID : /\* <primaria> -> <identificador> \*/

Identificador(presul); break;

case CONSTANTE : /\* <primaria> -> CONSTANTE #procesar\_cte \*/

Match(CONSTANTE); \*presul = ProcesarCte(); break;

case PARENIZQUIERDO : /\* <primaria> -> PARENIZQUIERDO <expresion> PARENDERECHO

Match(PARENIZQUIERDO); Expresion(presul);

Match(PARENDERECHO); break;

default : return;

}

}

void OperadorAditivo(char \* presul) {

/\* <operadorAditivo> -> SUMA #procesar\_op | RESTA #procesar\_op \*/

TOKEN t = ProximoToken();

if ( t == SUMA || t == RESTA ) {

Match(t);

strcpy(presul, ProcesarOp());

} else

ErrorSintactico(t);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Rutinas Semanticas\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

REG\_EXPRESION ProcesarCte(void)

{

/\* Convierte cadena que representa numero a entero y construye un registro semantico \*/

REG\_EXPRESION reg;

reg.clase = CONSTANTE;

strcpy(reg.nombre, buffer);

sscanf(buffer, "%d", &reg.valor);

return reg;

}

REG\_EXPRESION ProcesarId(void) {

/\* Declara ID y construye el correspondiente registro semantico \*/

REG\_EXPRESION reg;

Chequear(buffer); //function auxiliar

reg.clase = ID;

strcpy(reg.nombre, buffer);

return reg;

}

char \* ProcesarOp(void) {

/\* Declara OP y construye el correspondiente registro semantico \*/

return buffer;

}

void Leer(REG\_EXPRESION in) {

/\* Genera la instruccion para leer \*/

Generar("Read", in.nombre, "Entera", "");

}

void Escribir(REG\_EXPRESION out) {

/\* Genera la instruccion para escribir \*/

Generar("Write", Extraer(&out), "Entera", "");

}

REG\_EXPRESION GenInfijo(REG\_EXPRESION e1, char \* op, REG\_EXPRESION e2){

/\* Genera la instruccion para una operacion infija y construye un registro semantico con el resultado \*/

REG\_EXPRESION reg;

static unsigned int numTemp = 1;

char cadTemp[TAMLEX] ="Temp&";

char cadNum[TAMLEX];

char cadOp[TAMLEX];

if ( op[0] == '-' ) strcpy(cadOp, "Restar");

if ( op[0] == '+' ) strcpy(cadOp, "Sumar");

sprintf(cadNum, "%d", numTemp);

numTemp++;

strcat(cadTemp, cadNum);

if ( e1.clase == ID) Chequear(Extraer(&e1));

if ( e2.clase == ID) Chequear(Extraer(&e2));

Chequear(cadTemp);

Generar(cadOp, Extraer(&e1), Extraer(&e2), cadTemp);

strcpy(reg.nombre, cadTemp);

return reg;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Funciones Auxiliares\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Match(TOKEN t) {

if ( !(t == ProximoToken()) ) ErrorSintactico();

flagToken = 0;

}

TOKEN ProximoToken() {

if ( !flagToken ) {

tokenActual = scanner();

if ( tokenActual == ERRORLEXICO ) ErrorLexico();

flagToken = 1;

if ( tokenActual == ID ) {

Buscar(buffer, TS, &tokenActual);

}

}

return tokenActual;

}

void ErrorLexico() {

printf("Error Lexico\n");

}

void ErrorSintactico() {

printf("Error Sintactico\n");

}

void Generar(char \* co, char \* a, char \* b, char \* c) {

/\* Produce la salida de la instruccion para la MV por stdout \*/

printf("%s %s%c%s%c%s\n", co, a, ',', b, ',', c);

}

char \* Extraer(REG\_EXPRESION \* preg) {

/\* Retorna la cadena del registro semantico \*/

return preg->nombre;

}

int Buscar(char \* id, RegTS \* TS, TOKEN \* t) {

/\* Determina si un identificador esta en la TS \*/

int i = 0;

while ( strcmp("$", TS[i].identifi) ) {

if ( !strcmp(id, TS[i].identifi) ) {

\*t = TS[i].t;

return 1;

}

i++;

}

return 0;

}

void Colocar(char \* id, RegTS \* TS){

/\* Agrega un identificador a la TS \*/

int i = 4;

while ( strcmp("$", TS[i].identifi) ) i++;

if ( i < 999 ) {

strcpy(TS[i].identifi, id );

TS[i].t = ID;

strcpy(TS[++i].identifi, "$" );

}

}

void Chequear(char \* s){

/\* Si la cadena No esta en la Tabla de Simbolos la agrega,

y si es el nombre de una variable genera la instruccion \*/

TOKEN t;

if ( !Buscar(s, TS, &t) ) {

Colocar(s, TS);

Generar("Declara", s, "Entera", "");

}

}

void Comenzar(void) {

/\* Inicializaciones Semanticas \*/

}

void Terminar(void) {

/\* Genera la instruccion para terminar la ejecucion del programa \*/

Generar("Detiene", "", "", "");

}

void Asignar(REG\_EXPRESION izq, REG\_EXPRESION der){

/\* Genera la instruccion para la asignacion \*/

Generar("Almacena", Extraer(&der), izq.nombre, "");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*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 },

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 = tabla[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\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/