UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

HERRAMIENTA DE DEPURACION PARA PROGRAMAS EN LENGUAJE "C"

T E S I S

QUE PARA OBTENER EL TITULO DE

LICENCIADO EN MATEMATICAS

P R E S E N T A

MARIA JULIA OROZCO MENDOZA

DIRECTOR DE TESIS
ING. RODRIGO SIGUENZA V.

MEXICO. D. F. 1987

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

INDICE GENERAL

INTRODUCCION

Uno de los problemas con los que comúnmente se enfrenta un programador al elaborar un programa es el tiempo que gasta en la búsqueda de errores. Un gran porcentaje de éste lo emplea en localizar errores de lógica; realmente existen pocos programas que lo auxilien en esta tediosa tarea.

A los programas de un sistema de cómputo que tienen como finalidad ayudar al usuario en la localización de errores de lógica, se les conoce con el nombre de depuradores.

Muchos sistemas operativos proveen de facilidades para desarrollar software; pero, en cambio, carecen de depuradores para lenguajes de alto nivel que sean fáciles de usar.

Casi todos los depuradores que existen funcionan por medio de "breakpoints", los cuales son localidades de memoria en un programa. Por medio de un "breakpoint" se puede regresar el control del programa al depurador, generalmente para imprimir el valor de algunas variables en el momento en que se llega al punto en que éste se definió; pero este medio de ayuda para la búsqueda de errores se realiza al nivel de código de máquina, razón por la cual resulta poco usado por programadores que utilizan lenguajes de alto nivel, además de que su uso requiere de un tiempo considerable de aprendizaje, ya que hay que interactuar con una gran variedad de comandos para poder insertar, borrar, habilitar e inhabilitar tanto "breakpoints" como otros comandos.

Como consecuencia, muchos programadores recurren al método de insertar rutinas de impresión en el programa fuente para visualizar como se va ejecutando, observar los valores de ciertas variables importantes en determinados puntos y localizar que sección del código del programa es la que está causando el problema.

Considerando que existen pocos depuradores que sean fáciles de usar, y que con un mínimo de comandos desplieguen

al usuario la información necesaria para localizar un error, se pensó en desarrollar una herramienta que facilite esta tediosa tarea.

Este trabajo describe un sistema que se llamo'"LOC_C"; su función es insertar rutinas de impresión en cada proposición de un programa fuente en lenguaje "C", evitando que el usuario, al hacerlo manualmente, caiga en errores no cometidos con anterioridad o que modifique la lógica del programa original. El sistema permite rastrear, durante la ejecución de un programa,los errores de lógica,sin tener que interactuar con una gran lista de comandos. Puede resultar de gran ayuda, sobre todo, para aquellos usuarios con poca experiencia en programación.

Se escogió el lenguaje "C" por ser uno de los lenguajes de propósito general más usados en la actualidad, además de ser un lenguaje que posee un alto grado de portabilidad; un programa en "C" genera un código eficiente en cuanto a taman~o y rapidez. Esta característica lo convierte en un lenguaje que puede ser usado en diversas aplicaciones ya sean científicas, comerciales, para programar sistemas operativos, para hacer juegos, etc.

"LOC_C" se implementó en una computadora ATT 7300 con sistema operativo Unix, SYSTEM V. Por estar escrito en "C", tiene la ventaja de ser independiente del compilador, lo que significa que es portátil de una máquina a otra. Se emplearon las utilerías de Unix, LEX y YACC, las cuales forman parte del software más nuevo que existe para la construcción de compiladores y son muy importantes en la creación y mantenimiento de algunas utilerías del sistema operativo UNIX.

En las siguientes secciones se hará una descripción de como utilizarlos y se dará una breve explicación de como funcionan . Se hablará también de la elaboración de LOC_C usando la interfase de LEX y YACC; por último se darán unos ejemplos del funcionamiento del depurador, así como las conclusiones de este trabajo.

CAPITULO I

CONTENIDO

I.1 - INTRODUCCION

Una de las dificultades más frecuentes que se presentan en la programación de computadoras, es el reconocimiento de las instrucciones de un lenguaje; aunque algunos comandos sean fáciles de reconocer otros, en cambio, requieren de técnicas bastante sofisticadas.

Generalmente el reconocimiento de estos comandos se ha dividido en dos fases : análisis lexicográfico y análisis sintáctico . Los objetos de bajo nivel en un lenguaje de programación, como son los números operadores y especialmente las palabras reservadas, se reconocen durante la fase del análisis lexicográfico; pero objetos de alto nivel como las proposiciones (statements), son reconocidos durante la fase del análisis sintáctico.

La entrada para un analizador lexicográfico consta de un flujo de caracteres, los cuales pueden provenir de una terminal, de un archivo, o bien ser salida de otro programa.

En cualquier caso el analizador lexicográfico examina la entrada de acuerdo a un conjunto de reglas y cuando reconoce un objeto, nos indica como salida qué tipo de objeto se ha encontrado; a esta indicación la conoceremos con el nombre de "token" (átomo). Frecuentemente se utiliza el término " scanner " para referirse a un programa que realiza el análisis léxico y el término " parser " para describir un programa que realiza el análisis sintáctico .

Lex es un programa que ayuda a generar un analizador lexicográfico; diremos que es un generador de programas que parte de una serie de reglas, las cuales tienen asociado código en diferentes lenguajes de programación, a los que llamaremos lenguajes anfitriones. "C" y "Ratfor" son los lenguajes que generalmente son usados como lenguajes anfitriones; en este trabajo se hace referencia unicamente a "C" como lenguaje anfitrión.

Lex tiene como entrada un archivo con las especificaciones del analizador léxico que se desea construir, y como salida una subrutina en "C" llamada "yylex" la cual puede ser compilada y ligada con otros programas y asi formar un analizador . Esta subrutina se encuentra en un archivo llamado lex.yy.c .

Los tipos de cuerdas que va a reconocer la subrutina generada por Lex, se van a especificar en un archivo, al que llamaremos " archivo fuente de Lex ". Las especificaciones se efectúan por medio de expresiones regulares a las que se les van a asociar acciones que conoceremos como " acciones semánticas " , y que deberán estar escritas en el lenguaje anfitrión. Cada acción se va a ejecutar al momento en que se reconozca la expresión regular a la que está asociada.

A continuación se dará una descripción del archivo fuente de Lex.

I.2 - ARCHIVO FUENTE DE LEX :

El archivo fuente de Lex está formado por una tabla que consta de expresiones regulares y acciones semánticas. Esta tabla va a ser transformada por Lex en un programa que, al reconocer una expresión regular, ejecute la acción semántica correspondiente a dicha expresión.

La forma general del archivo fuente es la siguiente :

```
definiciones
%%
reglas
%%
rutinas del usuario
```

En la primera sección el usuario tiene la opción de declarar variables, ya sea para ser usadas en su programa o por Lex. Dentro de las rutinas del usuario, se pueden definir rutinas para utilizarlas dentro de las acciones semánticas.

Tanto las declaraciones como las rutinas del usuario pueden omitirse. El segundo %% es opcional, pero el primero es necesario ya que va a marcar el inicio de las reglas.

Por lo tanto el programa mínimo que acepta LEX es :

```
%%
```

el cual no incluye declaraciones ni reglas, y copia su entrada a su salida.

1.3 - DEFINICIONES

Como se mencionó anteriormente, el usuario puede declarar variables para utilizarlas en las acciones semánticas. Una declaración puede ir en la sección de las definiciones, o bien, en la sección de las reglas; una declaración en la sección de las definiciones será global a cualquier función del código sumado por el usuario; una declaración en la sección de las reglas deberá aparecer inmediatamente después del primer delimitador "%%" y será local a las acciones.

La sección de las declaraciones va a estar formada por una combinación de las siguientes :

1) Definiciones en la forma :

```
nombre  espacio  traducción
```

Este tipo de definiciones pueden ser usadas para abreviar reglas gramaticales

Ejemplo :

Consideremos el siguiente programa, el cual reemplaza cada número entero por la palabra entero.

```
D     [Ø-9]
%%
(D)+ printf("entero");
```

Una definición de esta forma deberá comenzar en la primera columna. El nombre y la traducción deberán estar separados por lo menos por un blanco o tabulador, y el nombre deberá comenzar con una letra. Una traducción puede ser usada desde una regla gramatical por medio de la sintaxis:

{nombre}

2) Código en la forma :

espacio código

Cualquier línea que comience con un blanco o un tabulador, es copiada tal cual dentro del programa generado por Lex. Declaraciones de esta forma pueden ir en la sección de las reglas siempre y cuando se encuentren antes de la primera regla.

Ejemplo :

Consideremos un programa que cuente lineas

```
int line;
%%

\n    line++;
```

o bien, la declaración de la variable, en la segunda sección

```
%%
int line;
\n    line++;
```

En el primer caso la variable line es una variable global a todo el código sumado, tanto en la sección de las reglas como en la de rutinas del usuario; en el segundo caso actúa como local al código sumado en la sección de las reglas.

3) Código de la forma :

```
%{
   código
%}
```

En donde el código es cualquier tipo de declaración de "C", incluyendo comentarios.

Ejemplo :

```
%{
#include "mom.h"
int  line=1;
char *strsave();
%}
```

4) Condiciones iniciales de la forma :

```
% START nombre1 nombre2 ...
```

La palabra START puede abreviarse por "s" o "S". Una condición inicial puede ser referida desde una regla por medio de :

```
< nombre1 > expresión
```

siempre que Lex esté en una condición inicial nombre1. Para entrar a una condición inicial se ejecuta la acción :

```
BEGIN nombre1;
```

Ejemplo :

Consideramos el siguiente programa, el cual copia la entrada en la salida cambiando la palabra mágico por la palabra primero en cada línea que comience con la letra a; y cambiar mágico por segundo en cada linea que comience con la letra b, cualquier otra línea se copia tal cual.

```
int flag
```

```
%%
^a  {flag = 'a'; ECHO;}
^b  {flag = 'b'; ECHO;}
\n  {flag = 0 ;  ECHO;}
magico {
        switch (flag) {
        case 'a': printf("primero"); break;
        case 'b': printf("segundo"); break;
          default : ECHO; break; }
        }
```

NOTA : La expresión regular ^x , indica una x, al principio de una linea.
La proposición ECHO, imprime la cuerda actual.
Más adelante se explican con más detalle.

Por medio de condiciones iniciales podemos manejar el mismo problema de la siguiente manera:

```
%START AA BB
%%
^a    ( BEGIN AA;ECHO )
^b    ( BEGIN BB;ECHO )
\n    ( BEGIN 0;ECHO )
<AA>magico    { printf("primero");}
<BB>magico    { printf("segundo");}
```

5) Traducciones de caracteres por medio de una tabla en la sig. forma :

```
%T

 número  espacio  cuerda de caracteres
 ...
 ...

%T
```

en donde el número indica el valor asociado a la cuerda de caracteres. Ningún caracter deberá asociarse con el valor "0" (porque éste es reservado por Lex para reconocer un final de archivo).

Ejemplo :

En "C" la letra "a" está representada como el caracter constante 'a' Si se desea cambiar esta representación,

se hace por medio de una tabla de traducciones (siempre y cuando se cambien las rutinas de la biblioteca de entrada y salida input - output) :

```
%T
1     Aa
2     Bb
3     Cc
%T
```

6) Un lenguaje especifico (que debe preceder a cualquier còdigo incluido) en la forma %R que genera còdigo para"Ratfor", %C o nada para "C".

7) Cambio de taman~os en arreglos internos (los cuales sirven para ajustar los taman~os de arreglos generados por Lex para control interno) [Aho 77]

```
%x nnn
```

en donde nnn es un entero decimal que representa el taman~o del arreglo y "x" es cualquiera de los siguientes parámetros

```
p  posición
n  estados
e  nodos de árboles
a  transiciones
k  clases de caracteres empacados
o  taman~o del arreglo de salida
```

I.4 - REGLAS

La sección de las reglas está formada por una tabla en la cual la columna de la izquierda contiene expresiones regulares y la columna de la derecha contiene las acciones que deberán ejecutarse cuando las expresiones sean reconocidas.

Una línea en esta sección deberá tener la siguiente forma :

```
expresión regular     acción
```

En donde las acciones están representadas por fragmentos de programas escritos en "C", incluyendo comentarios.

1) Expresiones Regulares :

Entenderemos como una expresión regular a un conjunto de cadenas (sucesiones finitas de simbolos) sobre un alfabeto (conjunto finito de simbolos).

Una expresión regular sobre un alfabeto se define recursivamente como :

La cuerda vacia es una expresión regular que denota el conjunto que contiene a la cuerda vacia; cualquier elemento x, del alfabeto, es una expresión regular que denota al conjunto que contiene sólo la cadena x; si r y s son expresiones regulares que denotan a los conjuntos (de cadenas) R y S respectivamente entonces : i) r+s, es una expresión regular que denota a la unión de los conjuntos R y S, ii) rs, es una expresión regular que denota a la concatenación de los conjuntos R y S, iii) r*, es una expresión regular que denota la cerradura de Kleene del conjunto R.

Lex permite para la construccion de expresiones regulares, lo siguiente :

x	el caracter "x"
"x"	una "x", si x es un operador
\x	una "x", si x es un operador
[xy]	el caracter "x", o el caracter "y"
[x-z]	el caracter "x", "y", o "z"
[^x]	cualquier caracter exceptuando x
.	cualquier caracter exceptuando newline
^x	una x en el principio de la linea
<y>x	una x cuando Lex se encuentra en la condición inicial y.
x$	una x en el fin de una linea
x?	una x opcional
x*	cero o varias veces x
x+	una o varias veces x
x|y	una x o una y
(x)	una x
x/y	una x pero solo seguida de una y
{xx}	la translación de xx de la sección de las definiciones
x{m,n}	de m a n ocurrencias de x

El fin de una expresión está marcado por el primer blanco, o tabulador.

Ejemplos:

```
" "          regla que reconoce blancos

"\t"         regla que reconoce tabuladores

"\n"         regla que reconoce una nueva
             línea

[Ø-9]+            regla que reconoce enteros

{D}+"."{D}*({P})? |

{D}*"."{D}+({P})? |

{D}+{P}?     regla que reconoce reales,
             definiendo en la sección
             de las declaraciones
             traducciones de forma :

              D      [Ø-9]
              P      [eE][-+]?{D}+

(_|[A-Za-z])([A-Za-zØ-9]|_)*
             regla que reconoce
             identificadores en "C".
```

ii) - Acciones :

Una acción es el código asociado a cada expresión regular, el cual se ejecutará una vez que la expresión haya sido reconocida. A continuación se mencionan las facilidades que presenta Lex para realizar las acciones.

Si la acción es sólo una expresión en "C", deberá escribirse del lado derecho de la línea, pero si consta de un bloque de expresiones, deberá estar entre llaves. Hay una acción por omisión, que consiste en copiar la entrada en la salida.

Cuando la acción está formada por una proposición nula, el resultado es ignorar la entrada.

Ejemplo :

Consideremos el siguiente programa, el cual ignora los espacios en blanco, los tabuladores, y la indicación de una nueva linea.

```
[ \t\n]          ;
```

Otra forma de escribir esta acción es con el caracter "|", que indica que la acción de una regla, es la acción de la siguiente regla.

Ejemplo :

```
" "          |
"\t"         |
"\n"         ;
```

que tiene un estilo diferente, pero produce el mismo resultado.

En una acción semántica se puede utilizar la proposición " ECHO ", la cual nos permite saber el contenido de la cadena actual.

Ejemplo :

Consideremos una regla que reconozca cadenas de letras minúsculas y asociémosle una acción semántica que imprima dichas cadenas.

```
[a-z]+       ECHO;
```

Una cadena actual puede también obtenerse por medio de un arreglo de caracteres llamado "yytext". Por lo tanto, el ejemplo anterior podria realizarse de la siguiente manera :

```
[a-z]+       printf("%s",yytext);
```

Frecuentemente es conveniente saber cual es el último caracter de una cadena; para ésto, Lex provee de un contador (yyleng) del número de caracteres rastreados. De esta manera, el último elemento puede ser obtenido desde una acción semántica por medio de yytext[yyleng-1], ya que en "C", los arreglos comienzan en cero.

Una acciòn puede decidir si no se ha reconocido la expansiòn correcta de un conjunto de caracteres, por medio de las funciones yymore() y yyless(n).

Ejemplo:

Si consideramos la siguiente cuerda "abc\"def", y tenemos una regla que reconoce "cuerdas" al reconocer "abc\", llamamos yymore() para continuar el proceso. Yyless(n) nos sirve para indicar que no se guarden todos los caracteres en yytext; el argumento n indica el nùmero de caracteres que se desea guardar.

Lex tambièn permite el acceso a rutinas de I/O; èstas son:

1) input(), que regresa el siguiente caracter de la entrada .

2) output(c), que escribe el caracter "c" en la salida.

3) unput(c) , regresa el caracter "c" para poder ser leido nuevamente por la funciòn input().

Estas funciones estàn previstas como macrodefiniciones, pero el usuario las puede suplir como versiones privadas; como establecen la relaciòn entre los archivos externos y los caracteres internos, se debe tener cuidado de modificarlas consistentemente, teniendo cuidado de mantener la relaciòn entre input y unput para que el siguiente token de entrada (token de look-ahead) pueda funcionar.

Cada regla que termine con cualquiera de los siguientes simbolos lleva implìcito un token de "look-ahead" : '+','*','?' ; o bien que incluya un simbolo "/". El simbolo de "look-ahead" tambièn es necesario en las expresiones que son prefijas de otras expresiones.

En lex no es posible escribir una regla que reconozca una marca de fin de archivo; el ùnico acceso a esta condiciòn es por medio de la funciòn yywrap(). Esta rutina de la biblioteca de lex regresa un "1" cuando encuentra el fin de la entrada; el usuario la puede redefinir para continuar con el curso del anàlisis en nuevas fuentes de entrada.

EJEMPLOS

```
%{

#include "mom.h"          /* contiene la definición de los
                                  tokens */

int  line=1;
char *strsave();

%}


D          [0-9]
P          [eE][-+]?(D)+
%%

" "             |
"\t"            ;
#[^\n]*         ;   /* fuera preprocesador */

"\n"            { line++; }


"/*"            {
                     kill_comment ();
                }


\"              {
                     get_string ();
                     yylval.txt = strsave(yytext);
                     return STRING;
                }



'[^']*               {
                     if (yytext[yyleng-1] == '\\' &&
                          yytext[yyleng-2] !=  '\\'          )
                         yymore ();
                     else {
                          input ();
                          yytext[yyleng++]='\'';
                          yytext[yyleng]=0;
                          yylval.txt =strsave(yytext);
                          return CONS; }
                     }
```

```
(0x|0X)[0-9a-fA-F]+ {
                      yylval.txt = strsave(yytext);
                      return CONS;
                    }

[0-9]+                {
                      yylval.txt=strsave(yytext);
                      return CONS;
                      }

{D}+"."{D}*({P})?[lL]? |
{D}*"."{D}+({P})?[lL]? |
{D}+{P}[lL]?          { yylval.txt =strsave(yytext);
                        return CONS;
                      }
switch                { return SWITCH; }
case                  { return CASE; }
default               { return DEFAULT; }
if                    { return IF; }
else                  { return ELSE; }
for                   { return FOR; }
while                 { return WHILE; }
do                    { return DO; }
break                 { return BREAK; }
continue              { return CONTINUE; }
goto                  { return GOTO; }
return                { return RETURN; }
sizeof                { return SIZEOF; }
long                  { return LONG; }
static                { return STATIC; }
struct                { return STRUCT; }
union                 { return UNION; }
short                 { return SHORT; }
unsigned              { return UNSIGNED; }
char                  { return CHAR; }
float                 { return FLOAT; }
double                { return DOUBLE; }
int                   { return INT; }
typedef               { return TYPEDEF; }
extern                { return EXTERN; }
register              { return REGISTER; }
auto                  { return AUTO; }


"++"            {  return INCREMENT; }
"--"            {  return DECREMENT; }
"->"            {  return MEMBER; }
```

```
">>"            {  return SR; }
"<<"            {  return SL; }
"<="            {  return LE; }
">="            {  return GE; }
"=="            {  return EQ; }
"!="            {  return NE; }
"||"            {  return LOR; }
"&&"            {  return LAND; }

"+="            {  return A_P; }
"-="            {  return A_M; }
"*="            {  return A_S; }
"/="            {  return A_D; }
"%="            {  return A_MOD; }
">>="               {  return A_SR; }
"<<="               {  return A_SL; }
"&="            {  return A_AND; }
"^="            {  return A_X; }
"|="            {  return A_OR; }

(_|[A-Za-z])([A-Za-z0-9]|_)*
                {
                  yylval.txt=strsave(yytext);
                                return ID;
                }
                {
                  return yytext[0];
                }
%%

static kill_comment () {

     char chant, current;

     chant = 0;
     for (;;) {
          current=input ();
          if (current == '\n') line++;
          if (chant == '*' && current == '/') break;
          chant = current;
     }
}

static get_string () {
     char chant;
     char *pyytxt;

     pyytxt = yytext; chant=yyleng=0;
```

```
        for (;;) {
                *pyytxt=input ();
                if (*pyytxt == '"' && chant != '\\') break;
                if (*pyytxt == '\n') line++;
                chant = *pyytxt++;
                yyleng++; }
        *pyytxt=Ø;
}
```

I.5 - FUNCIONAMIENTO

Lex transforma la tabla de especificaciones en un programa que al reconocer una expresión regular, ejecute la acción semántica correspondiente a dicha expresión.

El reconocimiento de expresiones se efectúa por medio de un autómata finito determinístico (autómata con un número finito de estados). Entenderemos por un autómata finito a un reconocedor construido en base a ciertas reglas que, al ser alimentado por una cadena, nos responde si esa cadena pertenece o no a un conjunto.

Generalmente un autómata finito se define como un quìntuple [Aho 77], formado por :
un conjunto finito de estados; alfabeto finito de entradas; una función de transición, la cual asigna un nuevo estado; un estado inicial en el que el autómata arranca siempre; un subconjunto del conjunto finito de estados, designados como estados finales o estados que aceptan.

El autómata finito es determinìstico, si en cada estado y al leer un símbolo dado hay una transición única a otro estado.

El número de reglas especificadas en el archivo fuente de Lex o la complejidad de las mismas, no afectan la velocidad del reconocimiento pero incrementan el taman~o del autómata finito y por lo tanto el taman~o del programa generado por Lex.

Al construir Lex el autómata finito, requiere de arreglos internos para control interno. Por ejemplo para cada estado necesita un arreglo que contenga las transiciones correspondientes a ese estado. Existen archivos de especificaciones que requieren el mìnimo (o más) del taman~o permitido por Lex. El taman~o de estos arreglos se puede ajustar dependiendo de las necesidades del usuario

por medio de la proposición " %x " (sección de las definiciones) .

Por ejemplo, el número máximo de estados que define lex es de 500; si se desea generar un programa con un número mayor de estados se hace por medio de la declaración

%n 550

Se efectuaron algunas pruebas del tamaño que genera lex para cada parámetro en diferentes archivos conteniendo una o varias reglas,obteniéndose los siguientes resultados:

0 reglas:

0/1000 nodes(%e), 2/2500 positions(%p), 2/500 (%n), 0 transitions, 0/1000 packed char classes(%k), 0/2000 packed transitions(%a), 0/3000 output slots(%o)

1 regla para reconocer enteros :

3/1000 nodes(%e), 8/2500 positions(%p), 3/500 (%n), 20 transitions, 2/1000 packed char classes(%k), 11/2000 packed transitions(%a), 58/3000 output slots(%o)

reglas para reconocer el lenguaje "C" :

451/1000 nodes(%e), 1460/2500 positions(%p), 193/500 (%n), 8802 transitions, 78/1000 packed char classes(%k), 614/2000 packed transitions(%a), 520/3000 output slots(%o)

Se efectuaron algunas pruebas con distintos archivos de especificaciones conteniendo diferentes reglas para ver el tamaño y la velocidad del reconocimiento obteniendo los siguientes resultados:

	TAMAÑO (bytes)	TIEMPO (seg)
sin reglas	16330	.1
una regla	16604	.4
reglas que reconocen el lenguaje "C"	24870	.9

CAPITULO II

CAPITULO II

CONTENIDO

II.1 INTRODUCCION

La segunda fase en el reconocimiento de comandos de un lenguaje la realiza el analizador sintáctico. A los programas en un sistema de cómputo que realizan este análisis se les conoce como "parsers". Para realizar su función el analizador sintáctico requiere como entrada de las especificaciones del lenguaje por medio de una serie de reglas, llamadas reglas gramaticales. Si un comando está formado conforme a alguna de esas reglas , entonces una acción se realiza; de lo contrario, es un error.

Un papel importante dentro del proceso del análisis sintáctico lo realiza el analizador léxico , el cual debe de leer la entrada y reconocer las estructuras de bajo nivel para comunicar esos átomos (tokens) al analizador sintáctico. Si el analizador léxico ha reconocido una secuencia de caracteres, el analizador sintáctico es el encargado de verificar que esa secuencia represente una expresión válida y entonces ejecutar una serie de acciones válidas. A una estructura que sea reconocida por el

analizador sintáctico la conoceremos como símbolo no terminal.

Generalmente se utilizan letras mayúsculas para denotar a los símbolos terminales y letras minúsculas para denotar a símbolos no terminales.

Como mencionamos anteriormente el analizador sintáctico para realizar su función, requiere de las especificaciones del lenguaje dado. Existen varios métodos para la descripción de un lenguaje, pero el método generativo por excelencia es el de las "Gramáticas Formales".

Entenderemos como una gramática a un cuádruplo formado por :

un alfabeto de símbolos terminales; un alfabeto de símbolos no terminales; un conjunto de producciones; un símbolo inicial.

Una producción es una regla de sustitución de la forma :

U ::= u

en donde "U" representa un símbolo no terminal y "u" es una cadena que puede estar formada por estructuras sintácticas (símbolos no terminales) o bien por aquellos símbolos que forman parte del alfabeto del cual está formado el lenguaje (tokens). El símbolo inicial es un símbolo no terminal "S" que representa la estructura más general descrita por la gramática . A partir de S se empiezan a hacer las sustituciones, de tal manera que S debe aparecer al menos una vez del lado izquierdo de alguna de las producciones.

Por la forma particular que tengan las producciones, las gramáticas formales se clasifican en varios tipos a saber :

-Gramáticas regulares o Tipo 3.- Sus producciones son de la forma

A : : = aB A : : = a

lo que significa que del lado izquierdo de la producción sólo puede aparecer un símbolo no terminal y del lado derecho un símbolo terminal seguido de uno no terminal, o bien un solo símbolo terminal. Sirven entre otras cosas para

describir los átomos de un lenguaje de programación. Estas gramáticas son equivalentes a las expresiones regulares.

-Gramáticas libres de contexto o Tipo 2.- En estas gramáticas las producciones son de la forma

A : : = g

del lado izquierdo se tiene un sólo simbolo no terminal mientras que del lado derecho admitimos cualquier cadena formada por terminales y no terminales. Generalmente los lenguajes de programación se describen con gramáticas de este tipo.

-Gramáticas dependientes del contexto o Tipo 1.- La forma general de las producciones es la siguiente

G : : = h

en donde G debe tener al menos un símbolo no terminal y h es cualquier cadena de símbolos. La única restricción que se pone es que el número de símbolos en h sea mayor o igual que el número de símbolos en G.

-Gramáticas generales o Tipo Ø.- Se permite todo, incluso aquellas donde la longitud del lado derecho de alguna producción sea menor que el lado izquierdo.

.....

II.2 YACC

Yacc es un programa que nos ayudá a construir un analizador sintáctico. La entrada para Yacc consiste de un archivo el cual debe contener las especificaciones del lenguaje por medio de una gramática formal libre de contexto, en donde cada regla gramatical tiene asociado código al que conoceremos como acciones semánticas, que serán invocadas cada vez que la proposición sea reconocida.

La diferencia que existe entre las especificaciones para Lex y para Yacc, radica en la forma de las reglas. Las

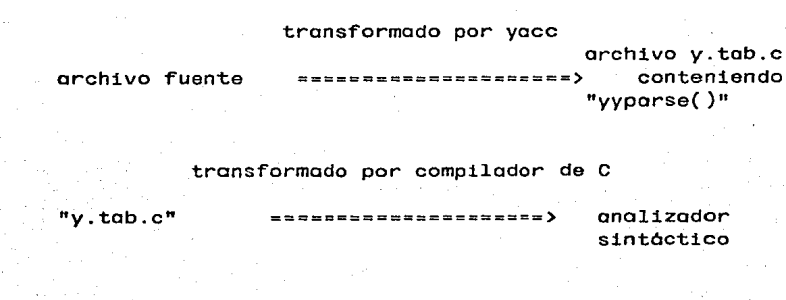
de Lex estan formadas por expresiones regulares mientras que las de yacc por reglas gramaticales.

Yacc va a transformar las especificaciones a una función en "C", llamada "yyparse", la cual va a controlar el proceso del análisis sintáctico; "yyparse" va a llamar a una función del usuario, para que desempeñe el papel del analizador léxico. Yacc genera el archivo "y.tab.c", el cual va a contener la función "yyparse".

Yacc está escrito en "C", y por lo tanto muchas de las convenciones son como en "C". La clase de reglas gramaticales aceptada por yacc son LALR(1) [Aho 77] , pertenecientes a las gramáticas libres de contexto

El código asociado a cada regla gramatical (acciones), deberá estar también en "C".

Las transformaciones que están involucradas para crear un analizador sintáctico por medio de Yacc, se muestran en el siguiente diagrama :



1) Archivo fuente de yacc

El archivo fuente va a estar dividido en tres secciones a saber : declaraciones, reglas y rutinas del usuario. Cada sección deberá estar separada por la marca " %% ", y con el siguiente formato :

```
declaraciones
%%
reglas
%%
rutinas del usuario
```

El usuario tiene la opción tanto de declarar variables (sección de las declaraciones), como de definir funciones (sección de rutinas), con el fin de utilizarlas en el código asociado a cada regla gramatical.

Aunque la sección de declaraciones puede omitirse, la primera marca de "%%" siempre deberá existir; la segunda marca deberá omitirse en el caso de que no haya sección de rutinas del usuario, por lo tanto el conjunto de especificaciones más pequen~o permitido por yacc es :

```
%%
reglas
```

el cual no contiene sección de declaraciones ni sección de rutinas.

ii) Declaraciones

La sección de las declaraciones está formada por dos partes. La primera consiste de declaraciones en "C", delimitadas por "%(" y "%)". Las variables declaradas en esta sección son globales a todas las acciones asociadas a las reglas. La segunda parte de esta sección consiste en la declaración de los tokens, los cuales deben estar declarados de la siguiente forma :

```
%token nombre1 nombre2 ...
```

con la restricción del nombre "error", el cual no deberá incluirse en las declaraciones de los tokens puesto que yacc lo reserva para el manejo de errores.

Cada nombre que no esté definido será asumido como un símbolo no terminal y deberá aparecer por lo menos una vez en el lado izquierdo de las producciones.

Yacc también produce un archivo llamado "y.tab.h", el cual contiene los tokens con sus respectivos valores. Este archivo es necesario para establecer la comunicación entre el analizador léxico y el analizador sintáctico. El valor entero asociado a cada literal (caracteres entrecomillados) es el valor en código ASCII; a los nombres declarados como tokens se les asigna un valor en orden creciente comenzando en el 257. Estos valores pueden ser cambiados por el usuario por medio del mecanismo " #define nombre valor ". El

valor deberá ser un entero positivo, ya que se reserva el valor de menor o igual que cero, para el fin de archivo.

Es muy conveniente incluir dentro de la definición de los tokens la declaración del simbolo inicial . De los simbolos no terminales este tiene particular importancia, puesto que el analizador sintáctico está disen~ado para reconocerlo, debido a que representa la estructura más general dentro de las reglas gramaticales. La declaración debe hacerse de la siguiente forma :

```
%start simbolo
```

Resumiendo, la forma general que tiene la sección de las declaraciones es la siguiente :

```
{%
     declaraciones en "C"

%}

%token nombre1 nombre2 nombre3  ....
%start x
```

iii) Reglas

La sección de reglas consta de una o varias reglas gramaticales, con la siguiente forma :

```
A : cuerpo;
```

en donde la producción A representa un simbolo no terminal, y el cuerpo es una secuencia de cero o más nombres y literales. Una literal consiste de caracteres entrecomillados (que de hecho son tokens), y un nombre está representado por simbolos no terminales y tokens. Cada nombre que represente un token, debe de estar declarado como se vió previamente en la sección de las declaraciones.

Si se ha omitido la declaración del simbolo inicial en la sección de las definiciones , Yacc va tomar como tal aquel simbolo que se encuentre del lado izquierdo de la primera regla gramatical.

Los signos de puntuación utilizados para la definición de las reglas son :

":" marca el inicio de una definición
"|" separa las diferentes alternativas en una definición
";" indica el final de una definición

iv) Acciones

Cada regla gramatical tiene asociada una acción semántica. Esas acciones pueden regresar valores, y pueden obtener valores regresados por acciones anteriores.

Una acción está formada por una o varias proposiciones de C, las cuales deberán estar delimitadas por "{" y "}". Para regresar el valor de una acción se utiliza la variable "$$"; y para obtener valores regresados por acciones anteriores se utilizan "$1", "$2", ... los cuales se refieren a los valores regresados por los componentes del lado derecho de cada regla, asignándoseles "$i" a la regla que ocupa el lugar i, de izquierda a derecha.

Ejemplo :

```
A : B C;
    { $$ = $2; }
```

La producción A regresa $2 que es el valor asociado a la producción C.

Una acción puede hacer referencia a valores regresados que estén a la izquierda de la regla actual por medio de la variable $, seguida de un entero que puede ser cero o negativo. Cuando no se especifican acciones, Yacc asume que el valor de una regla es el valor del primer elemento de ella, es decir el de $1.

Ejemplo :

```
A : B C;
```

La producción A tendrá asociado el valor de la producción B.

Yacc permite ejecutar acciones antes de que la regla sea reconocida, es decir, permite que se escriban acciones intermedias a los elementos de una regla .

Ejemplo :

```
A  :  B
   {  $$ = 1; }
      C
   {  x = $2; y = $3; }
   ;
```

El efecto que produce es asignar el valor de 1 a la variable "x" , y a la variable "y" el valor regresado por C.

Estos casos también pueden manejarse por medio de intercalar reglas vacias, es decir :

```
A  :  B aux C
   {  x = $2; y = $3; }
   ;

aux :  /* empty */
   {  $$ = 1;  }
    ;
```

El tipo de los valores regresados por las acciones y por el analizador léxico por omisión, son enteros. Para regresar otro tipo de valores (incluso estructuras), deberá definirse una unión en la segunda sección de las definiciones, es decir con la declaración de los tokens y del simbolo inicial. Los miembros de la unión deben indicar los tipos de valores que la acción deberá regresar :

Ejemplo :

```
%{
     declaraciones
%}

%union{
       cuerpo de la union
}
```

Alternativamente la unión puede declararse como un encabezado (header) dentro de la primera sección de las

declaraciones, es decir, entre las llaves %{ y %} y por medio de la proposición "typedef" definir la variable YYSTYPE para representar la unión.

Ejemplo :

```
%{
     declaraciones

typedef union{
tipo  valor1;
tipo  valor2;
}YYSTYPE;

%}
```

Si la definición de una unión consta de varios tipos, se tiene que indicar dentro de la sección de las declaraciones por medio de la llave " %type ", el tipo de valor asociado tanto a cada símbolo terminal como a cada símbolo no terminal .

Se utilizan los picoparèntesis "<" y ">" para indicar el miembro de una unión, por medio de la construcción <nombre> .

Ejemplo :

```
%union{
     tipo  valor1;
     tipo  valor2;
}

%token <valor1> nombre1

%type <valor1> A B
%type <valor2> C
```

nos indica que el token nombre1 regresa el tipo de valor1 mientras que las reglas A y B, regresan el tipo de valor1, y la regla C el tipo de valor2.

La declaración de una unión implica que los valores regresados por las acciones, y las variables yylval y yyval deberán tener el mismo tipo que la unión.

Si yacc se invoca con la opción -d, la declaración de la unión se copia dentro del archivo y.tab.h .

Un tipo se puede indicar en la referencia de una acción, insertando el miembro de una unión inmediatamente después del primer $.

Ejemplo :

```
regla     : A
          { $<valor1>$ = $1; }
          ;
```

o bien por medio de : $$.valor

Ejemplo :

```
regla     : A
          { $$.valor1 = $1; }
          ;
```

II.3 FUNCIONAMIENTO DEL ANALIZADOR SINTACTICO

El analizador sintáctico generado por Yacc es una máquina finita con stack, en donde el tope del stack va a contener siempre un símbolo que corresponde al estado actual. Los estados están representados por etiquetas formadas por números enteros.

El analizador sintáctico va a ser capaz de leer y recordar el siguiente token de entrada al que se le conoce con el nombre de "look-ahead". En una acción semántica se puede hacer referencia a este token por medio de la variable yychar.

Al iniciarse el proceso del análisis, la máquina se encuentra en el estado "Ø" y por lo tanto el stack va a contener sólo este estado, no existiendo en ese momento ningún token de look-ahead.

La máquina tiene sólo cuatro acciones posibles a realizar :

mover (shift), reducir (reduce), aceptar (accept) y error.

Una transición del analizador sintáctico es alguna de las siguientes:

1.- Basado en el estado actual, el analizador sintáctico decide cuando necesita un token de look-ahead para efectuar la acción que debe seguir. Si decide que lo necesita , y no lo tiene, llama al analizador léxico para obtener el siguiente token.

2.- Usando el estado actual y el token de look-ahead (en dado caso que lo requiera), el analizador decide la siguiente acción y la ejecuta . La acción puede ser meter un estado dentro del stack o sacarlo.

La acción de mover es la acción más común del analizador ; cada vez que esta acción se realiza siempre hay un token de "look-ahead". Por ejemplo el estado "x" puede tener la acción:

x IF shift y

lo que significa que en el estado "x" si el simbolo de look-ahead es IF el estado actual (en este caso) "x" se guarda en el stack y el estado "y" viene a ser el estado actual, es decir, se encuentra en el tope del stack y el simbolo de "look-ahead" se borra.

La acción reducir se efectúa cuando el analizador ha reconocido el lado derecho de la regla gramatical y debe reemplazarlo entonces por el lado izquierdo de la regla.Esta acción, depende del simbolo del lado izquierdo de la regla y del número de simbolos del lado derecho de la regla. Para reducir, el número de estados que se sacan del tope del stack, es igual al número de símbolos del lado derecho de la regla.

Como una acción de reducir se asocia con una regla gramatical en particular, las reglas gramaticales tienen asociada una etiqueta, por ejemplo la acción :

. reduce 18

se refiere a la regla gramatical 18 mientras la acción :

IF shift 18

se refiere al estado 18.

Se observa que el número 18 representa dos cosas: 1) en una acción de reducir se refiere al número de regla que se reduce; 2) en una acción de mover se refiere a lo que en los

reconocedores LALR es un "goto" y el número indica el estado al que hay que transferirse (colocar el número del estado en el tope del stack sin borrar el simbolo de "look-ahead").

Ejemplo:

A goto 2Ø

causa que el estado 2Ø se meta al stack y sea el estado actual sin afectar al token de "look-ahead".

La acción de reducir también es importante en el curso de las acciones que asoció el usuario a las reglas gramaticales, ya que cuando una regla es reducida el código asociado a la regla se realiza antes de que se reajuste el stack. Para esto se tiene otro stack que corre paralelo al stack que contiene los estados, el cual contiene los valores regresados por el analizador léxico y por las acciones. El stack de valores es accesado en una acción semántica por medio de las variables $1,$2, ...

Cuando una acción de mover se realiza, la variable externa yylval se guarda en el stack de valores al igual que cuando se efectúa un goto.

La acción de aceptar indica que la entrada cumple con alguna de las especificaciones y el analizador ha realizado entonces satisfactoriamente su trabajo. Esta acción se realiza sólo cuando el token de "look-ahead" es la marca de fin de entrada.

La acción error se realiza cuando el analizador no puede continuar de acuerdo a las especificaciones, es decir que encontró una entrada ilegal.

Si Yacc se invoca con la opción -v, produce también un archivo llamado "y.output" el cual contiene una descripción legible del funcionamiento del analizador.

EJEMPLO :

Si consideramos el siguiente archivo de especificaciones :

```
%token DING DONG DELL
%%
ritmo  : sonido nota
       ;
sonido : DING DONG
       ;
nota   : DELL
       ;
```

a continuación se muestra el archivo y.output producido, en el cual se especifica la acción para cada estado. El caracter "_", indica la lectura actual :

```
state 0
     $accept : _ritmo $end

     DING  shift 3
     .  error

     ritmo  goto 1
     sonido  goto 2

state 1
     $accept :  ritmo_$end

     $end  accept
     .  error


state 2
     ritmo :  sonido_nota

     DELL  shift 5
     .  error

     nota  goto 4

state 3
     sonido :  DING_DONG

     DONG  shift 6
     .  error
```

```
state 4
     ritmo :  sonido nota_     (1)

     .  reduce 1


state 5
     nota :  DELL_     (3)

     .  reduce 3


state 6
     sonido :  DING DONG_     (2)

     .  reduce 2
```

Supongamos que tenemos la siguiente entrada, para visualizar el funcionamiento del analizador

DING DONG DELL

Inicialmente el estado actual es el estado Ø. El analizador una vez que conoce el primer token DING y el token de "look-ahead", decide la acción a ejecutar que en este caso es realizar un movimiento del estado 3 (el estado 3 se mete al stack), y se borra el token de "look-ahead".El estado actual es entonces el estado 3. Se lee el siguiente token DONG y se requiere de un "look-ahead". La acción en el estado 3 con el token DONG es hacer un movimiento del estado 6 (el estado 6 se mete al stack) y el simbolo de "look-ahead" se borra. El stack contiene en ese momento los estados Ø, 3 y 6. En el estado 6 sin requerir un simbolo de "look-ahead" se reduce la regla 2;los estados 3 y 6 se sacan del stack, dejando en el tope el estado Ø, en el cual se requiere un "goto 2" o sea una transferencia a 2 que viene a ser el estado actual. Se lee el siguiente token de entrada DELL. La acción en el estado 2 con el token DELL es hacer un movimiento del estado 5 (se mete el estado 5 al stack). El stack contiene actualmente,los estados Ø,2 y 5 y se borra el token de "look-ahead". En el estado 5 la única acción posible es la de reducir con la regla 3, ésta tiene un solo simbolo del lado derecho, por lo tanto sólo se saca del stack el estado 5. En el tope queda el estado 2, en el cual con el lado izquierdo de la regla 3, se requiere hacer un "goto 4". Ahora el stack contiene los estados Ø, 2 y 4. En el estado 4 la única acción es reducir con la regla 1.

por lo tanto se sacan 2 estados (4 y 2), quedando en el tope el estado Ø; se hace un "goto 1" (se mete al stack el estado 1). Se lee el siguiente token , que es el fin de la entrada (indicado por $end en el archivo y.output). La acción en el estado 1, cuando se tiene el fin de entrada es terminar el análisis satisfactoriamente, es decir, aceptar la cuerda de entrada.

Las acciones de error y aceptación del analizador pueden ser simuladas en una acción semántica por medio del uso de los macros YYACCEPT y YYERROR. El primero causa que la función yyparse() regrese el valor Ø. Cuando se encuentra un error de sintaxis YYERROR llama a la función yyerror(), la cual imprime un mensaje de error. Esta función se encuentra en la biblioteca de yacc, o bien puede ser redefinida por el usuario. Como se mencionó anteriormente el token "error" se reserva para el manejo de errores, el cual puede ser usado en las reglas gramaticales. Cuando se encuentra un error, puede ser necesario borrar o alterar las entradas de la tabla de símbolos si se desea continuar el análisis, en lugares en donde los errores son esperados y se desea recuperarlos . Si no se especifica ninguna acción el proceso se detiene al encontrarse algún error. Si algún error se detecta cuando el analizador se encuentra en un estado de error, no se emite ningún mensaje y la entrada se borra silenciosamente. Por esta razón hay una proposición "yyerrok" , la cual en una acción causa que el analizador regrese a su estado normal. Este mecanismo hace creer al analizador que se ha completado la recuperación del error.

Ejemplo :

```
entrada   : error '\n'
          { printf("Reinstala la linea : " ); }
            input
          { $$ = $4; }
          ;
```

Si el analizador se encuentra en un estado de error no se va a emitir ningún mensaje por lo tanto este ejemplo puede ser tratado de la siguiente manera :

```
entrada   : error '\n'
          { yyerrok;
            printf("Reinstala la linea : " ); }
            input
          { $$ = $4; }
```

Una manera de rastrear el proceso del análisis al momento de que éste se está efectuando es por medio de la variable externa yydebug, la cual tiene normalmente el valor de Ø, pero si se redefine como 1 el analizador sintáctico da como salida además de las acciones que se están ejecutando, los símbolos de entrada leídos. Por ejemplo, si consideramos el archivo de especificaciones dado anteriormente, al invocar yacc con la opción -t , y habiendo definido yydebug como 1, el analizador sintáctico generado al tener como entrada la cuerda DING DONG DELL, da la siguiente salida :

```
State Ø, token -none-
Received token DING
State 3, token -none-
 Received token DONG
State 6, token -none-
Reduce by (2) "sonido : DING DONG"
State 2, token -none-
 Received token DELL
State 5, token -none-
Reduce by (3) "nota : DELL"
State 4, token -none-
Reduce by (1) "ritmo : sonido nota"
State 1, token -none-
Received token end-of-file
```

II.4 AMBIGUEDADES Y CONFLICTOS

Un conjunto de reglas gramaticales es ambiguo, si hay alguna cuerda de entrada la cual pueda estructurarse siguiendo dos o más caminos diferentes.

Por ejemplo para la siguiente regla :

```
exp : exp op exp
```

se puede tener la cuerda de entrada :

```
exp op exp op exp
```

La regla permite que esta cuerda pueda ser estructurada por medio de dos caminos diferentes :

```
(exp op exp) op exp
\_________/
     exp
```

a esta forma de estructurar la llamaremos asociatividad por la izquierda; o bien, se puede estructurar como

```
exp op (exp op exp)
       \_________/
         exp
```

y a esta forma la llamaremos asociatividad por la derecha.

Cuando el analizador sintáctico ha leìdo exp op exp, puede reducir aplicando la regla. El analizador continùa leyendo op exp y vuelve entonces a reducir. Este efecto es producido por una asociatividad por la izquierda, pero alternativamente, cuando el analizador ha leido exp op exp, puede suspender la aplicaciòn inmediata de la regla (ejecuta una acciòn de mover) y continuar la entrada completa hasta tener exp op exp op exp. Entonces puede aplicar la regla a los simbolos de la derecha, reduciéndolos a una exp, y formar exp op exp. Ahora la regla se aplica una vez màs reduciéndola a una exp. Este es el efecto de una asociatividad por la derecha.

En el ejemplo anterior el analizador se encuentra con que puede realizar dos acciones, un movimiento y una reducciòn, ambas legales. A un conflicto de este tipo lo llamaremos conflicto shift/reduce.Cuando se presenta el caso de que el analizador tenga dos o mas reducciones diferentes, diremos que se trata de un conflicto reduce/reduce. Yacc detecta estas ambiguedades, y aun asi puede construir el analizador sintáctico por medio de reglas desambiguas. Emite un mensaje con el número de conflictos que detectò y de que tipo de conflicto se trata; una regla desambigua es aquella que elige un camino a seguir cuando hay conflictos.

Yacc maneja dos reglas desambiguas a saber :

1) En un conflicto shift/reduce elige un movimiento (shift)

2) En un conflicto reduce/reduce, reduce la regla que reconozca primero durante la secuencia de la cuerda de entrada.

Los conflictos pueden examinarse en el archivo "y.output" que produce Yacc, al invocarlo con la opción : -v .

En dado caso de que Yacc no pueda quitar la ambiguedad de una regla, no se puede construir el analizador y manda entonces un mensaje de " error fatal ".

.....

1) Precedencia

A los tokens puede asignárseles una precedencia y asociatividad, con el fin de quitar conflictos y resolver ambiguedades en las reglas. Esto se hace en la sección de las declaraciones, por medio de una serie de lineas comenzando cada una con "%left", "%right", o "%nonassoc" seguidas de una lista de tokens. Se asume que todos los tokens que se encuentren en la misma línea tienen el mismo nivel de precedencia y asociatividad. Las lineas deberán listarse en orden creciente de precedencia.

Ejemplo :

```
%{                                          |
      declaraciones de variables            |
%}                                          |
                                            |
%right '='                                  |  sección de
%left '+' '-'                               |  declaraciones
%left '*' '/' '%'                           |
%left '(' ')'                               |
                                            |
/* declaraciones de otros tokens */         |
```

las cuales describen la precedéncia y asociatividad de los operadores aritmèticos, "+","-", se asocian por la izquierda y tienen menor precedencia que "*" y "/", que se asocian también por la izquierda.

Muchas veces un operador binario y un operador unario tienen la misma representación simbólica pero diferente

precedencia. La llave "%prec" cambia la precedencia y asociatividad en una regla gramatical particular. La llave deberá aparecer inmediatamente después del cuerpo de la regla y antes de la acción semántica y seguida de una literal cualquiera.

Ejemplo

```
exp : exp '-' exp
    | '-' exp  %prec !
      { accion }
    ;
```

Cuando se presenta un conflicto shift/reduce o reduce/reduce, y ninguno de los simbolos de entrada o la regla tienen marcada una asociatividad, se reporta el conflicto. Pero en cambio, cuando hay un conflicto shift/reduce y ambos, el simbolo de entrada y la regla gramatical,tienen asignados una precedencia y asociatividad, entonces Yacc resuelve el conflicto en favor de la acción (shift o reduce) asociada a la más alta precedencia, si tienen la misma precedencia entonces se utiliza la asociatividad.

CAPITULO III

CAPITULO III

CONTENIDO

L O C _ C :

III.1 ESPECIFICACIONES :

Como se mencionó en la introducción, "LOC-C" es un sistema que tiene la función de insertar código de depuración dentro de un programa fuente en "C", antes de que éste sea compilado, con el fin de proporcionar datos sobre como se comporta el programa al ejecutarse.

Despliega al usuario todos los detalles importantes, sin sobrecargarlo de información. Permite visualizar como se van modificando las variables y seguir el flujo del programa, pues nos indica el momento de entrar a un ciclo y su terminación. Sen~ala la entrada y salida de una función, asi como los valores tanto de los parámetros que le son pasados, como el valor que regresa dicha función; se toman en cuenta también las funciones de cualquier otro fuente que incluya el usuario, mediante la proposición #include " archivo ".

Para hacer uso de LOC_C, se utiliza el siguiente comando :

LOC_C nombre del archivo fuente [parámetros]

el archivo fuente es procesado produciendo otro programa en "C", donde cada proposición lleva una rutina de impresión, la cual se va a encargar de realizar el despliegue de los datos y de regresar el valor apropiado para que se efectúe dicha proposición. Esta rutina pude ser insertada sólo en algunas funciones indicando los nombres de éstas como párametros.

Ejemplo :

```
LOC_C fact calculo escritura
```

En este caso, LOC_C va a realizar el despliegue sólo de las proposiciones que se encuentren en las funciones "cálculo" y "escritura" del archivo fact.c.

Puesto que la impresión del desarrollo del programa debe ser legible, se lleva una indentación apropiada a cada proposición. Asi mismo se indica el número de linea de la proposición que se está ejecutando, permitiéndonos visualizar ciclos anidados.

Ejemplo:

```
main(){
     int k;

     k=4;
     n=1;
     while(--k)
     n = n*(k+1);
     printf("%d\n",n);
}
```

LOC_C genera el siguiente programa :

```
# 1 "factor.c"
int t_1,t_2,t_3,t_4,t_5,t_6;
main()
{
int n,k;
printf("main()\n");
prin_tf("{\n",6,1);
imp(" k = %d; \n",k=4,6,1);
imp(" n = %d; \n",n=1,7,1);
while(imp("while(%d)\n",--k,8,1))
imp(" n = %d; \n",n=n*(k+1),9,1);
printf("%d\n",n);
prin_tf("}\n",11,0);
```

```
}
```

En este archivo se emplean dos funciones de impresión . Una de ellas es la función "prin_tf", la cual tiene como parámetros una cuerda que representa el texto a imprimir, el número de linea y las variables necesarias para llevar a cabo la indentación. En cambio,la función "imp" además tiene como parámetro la operación a efectuar y regresa el resultado de dicha operación. Por ejemplo, en la linea número 8 del programa generador por LOC_C en el ejemplo anterior, tenemos la siguiente proposición:

```
while(imp("while(%d)\n",--k,8,1))
```

Como se puede observar la función "imp", tiene como segundo parámetro una expresión "--k" . La función "imp" se va a encargar de decrementar el valor de la variable "k" y de regresar dicho valor para que el "while" se pueda evaluar.

Una vez que LOC_C ha generado el archivo con el código de depuración necesario, lo compila para crear un archivo ejecutable con el nombre del programa fuente, que al correr nos produce la siguiente salida :

```
main()
6{
6     k = 4;
7     n = 1;
8    while(3)
9     n = 4;
8    while(2)
9     n = 12;
8    while(1)
9     n = 24;
8    while(0)
24
11}

.................
```

El formato de despliegue de cada instrucción depende del tipo de expresión que cada proposición tiene asociada . Por ejemplo,una proposición que tenga asociada una expresión booleana se va a desplegar de diferente modo que una con una expresión de asignación.

Como podemos observar en el programa fuente del ejemplo anterior en la proposición

```
while(--k)
```

se manda desplegar sólo el valor final de la expresión "--k"

```
8    while(3)
     ....
8    while(2)
     ....
8    while(1)
     ....
8    while(Ø)
```

Si por el contrario la proposición tuviera asociada una expresión booleana :

```
while(k++<3)
```

la forma de desplegar seria de la siguiente manera :

```
while(k++<=3  1 )
```

en donde el "1" nos indica que la expresión tiene un valor verdadero, o en su defecto un "Ø", si es falso, desplegándose en la terminal en modo de video inverso.

El despliegue del texto se va a realizar durante la ejecución de la proposición , es decir la rutina de impresión va a estar dentro de la proposición (salvo excepciones que más adelante se discutirán) . Si se efectuara antes podriamos no conocer el valor actual de algunas variables y se tendrian que declarar variables auxiliares para no modificar la lógica del programa fuente. Por ejemplo, si consideramos el siguiente caso :

```
if (factorial(a))
```

la impresión del texto correspondiente a la proposición se pudo haber realizado antes de que ésta se ejecute de la siguiente forma :

```
printf("if(factorial(%d))",a);
if(factorial(a))
```

También podria haberse mandado a imprimir una vez evaluada la proposición :

```
if(factorial(a)) {
  printf("if(factorial(%d))",a);
  desarrollo del if
}
```

pero no se tendria ninguna información en caso de que la expresión asociada al "if" no resultara verdadera. En cambio el realizar la impresión del texto durante su ejecución nos evita declarar variables auxiliares sin que esto modifique la lògica del programa fuente. Además, en la mayoría de los casos, nos evita abrir y cerrar llaves .

Ejemplo:

```
if(rutina_de_impresion("if(%d)",factorial(a))
```

En donde la rutina de impresión se encarga de imprimir el texto y de regresar el valor apropiado para que se evalùe la proposición . Más adelante se discutirá como se trataron las diferentes proposiciones del lenguaje.

III.2) ELABORACION E INSTRUMENTACION

La elaboración de LOC_C está basada en los mètodos ya existentes para la construcción de compiladores. El lenguaje se encuentra descrito por medio de una gramàtica formal libre de contexto; el reconocimiento de cada proposición involucrada en el programa fuente se lleva a cabo por medio de un analizador léxico y de un analizador sintáctico, en donde el analizador lèxico trabaja bajo el control del analizador sintáctico como una función, la cual se encarga de leer caracteres sucesivamente del programa fuente hasta encontrar un token para mandárselo al analizador sintáctico. Esta comunicación se realiza por medio de una pareja de valores. El primer valor es el valor del token y el segundo es un apuntador a la cadena leida.

Una vez que el analizador sintáctico ha reconocido el lado derecho de una producción (es decir, que se ha encontrado una proposición del lenguaje), se ejecuta la accion semántica asociada a la regla gramatical en cuestión.

Por ejemplo, si el archivo fuente contiene una proposición

"a = 8;"

debemos generar una función que imprima la cuerda "a=8" y que realice a su vez la operación de asignar el valor "8" a la variable "a". Para esto el analizador sintáctico primero tuvo que reducir las siguientes producciones y ejecutar cada acción semántica asociada.

```
PRODUCCION                  ACCION
----------                  ------
lvalue  : ID                guardar la cuerda "a"
        | lvalue '.' ID     por ser un ID
        | lvalue '->' ID
        ;

exp     : lvalue            regresar la expresión "a"
        | nlvalue           por ser un lvalue
        ;

nlvalue     : CONS          crear la cuerda "8"
            | STRING        por ser un CONS
            ;
exp     : lvalue
        | nlvalue           regresar la expresión "8"
        | ass_exp
        ;

ass_exp     : exp '=' exp       concatenar la cuerda "a"
                                con el operador "=", para
                                formar la cuerda "a=",
                                misma que se concatena
                                con la    cuerda "8"
        | exp '+=' exp
        | exp '-=' exp
        ;

exp     : lvalue
        | nlvalue
        | una_exp
        | bin_exp
        | ass_exp       indicar que la cuerda es una
                        expresión de asignación; guardar
                        la cuerda "a=8"
        ;
```

```
expresiòn : exp                 regresar la expresiòn
                                "a=8"
          | exp_coma
          ;
stat      : expresiòn ';'       generar el texto a
                                  imprimir :
                                  "a=%d;" y regresar la
                                  expresiòn a evaluar : a=8
          ;
```

En el anexo I se proporciona la gramatica usada en este trabajo.

i) Interfase lex y yacc

El analizador lèxico està representado por la funciòn entera yylex (), generada por medio de lex. El archivo de especificaciones para Lex se encuentra en el archivo fuente "c.l", en el cual se incluye el archivo que contiene la definiciòn de los tokens. Existe además otro valor asociado a los tokens; este valor se asigna a la variable externa yylval, la cual representa un apuntador a la tabla de identificadores.

Por medio de yacc se generò un analizador sintàctico LALR(1), el cual se encuentra representado por la funciòn yyparse(). El archivo de especificaciones requerido para generar esta funciòn se encuentra en el archivo "c.y".

Como la comunicaciòn entre yylex() y yyparse() se realiza por medio de la variable yyval, el stack de valores manejado por el analizador sintàctico se definiò del tipo :

```
%union
{
 char * txt ;
}
```

Al invocar yacc con la opciòn -d, la declaraciòn de la uniòn se copia en el archivo de declaraciones de los tokens. El valor entero asociado a cada literal es el valor en còdigo ASCII, los demàs tokens tienen un valor en orden creciente comenzando en el 257.

El miembro de la uniòn se asociò a los siguientes simbolos terminales :

ID

CONS
STRING

En cuanto a los simbolos no terminales, se les asoció el miembro de la unión a todos las producciones no vacias, es decir toda regla de este tipo va a regresar un apuntador a una cuerda, y el stack de variables va a contener apuntadores a cuerdas.

La precedencia y asociatividad de los operadores se encuentra definida, en orden creciente de la siguiente manera :

izquierda a derecha	,
derecha a izq.	= += -= *= /= %= >>= <<= &= ^= |=
derecha a izq	? :
izquierda a derecha	||
izquierda a derecha	&&
izquierda a derecha	|
izquierda a derecha	^
izquierda a derecha	&
izquierda a derecha	== !=
izquierda a derecha	< <= > >=
izquierda a derecha	>> <<
izquierda a derecha	+ -
izquierda a derecha	* / %
derecha a izquierda	! ~ ++ -- SIZEOF
izquierda a derecha	( ) [ ] -> .

ii) Acciones semánticas y estructuras de datos

El despliegue de cada proposición se va a realizar por medio de diferentes funciones de tipo entero . El tipo de función a insertar en cada proposición del programa fuente, va a depender del tipo de proposición o, en su defecto, del tipo de expresión que cada proposición tenga asignada.

Cada función va a tener como parámetros el texto a imprimir y el valor o valores asociados a cada texto, asi como los parámetros necesarios para efectuar la indentación apropiada a cada proposición, y va a regresar el valor requerido para que se ejecute dicha proposición.

El texto a imprimir se va a ir formando a la vez que la proposición se va analizando. Para esto las variables y proposiciones se van a tratar durante el análisis sintáctico como cuerdas, mismas que se van a unir (concatenar) para formar el texto requerido por las funciones antes mencionadas. Este texto va a representar la cadena de control requerida por la función "printf" (de la biblioteca de "C") de la cual van a hacer uso las funciones de impresión. En cuanto a las especificaciones de conversión se utilizan: "%d","%c","%f","%s".El tipo de conversión se va a determinar por medio de una tabla de símbolos manejada por el analizador sintáctico.

A continuación se presentan las proposiciones de la gramática de "C", cada una con el tipo de función que utiliza para el desdoblamiento del texto , las principales acciones semánticas asociadas a cada proposición y los ejemplos respectivos.

I) FUNCIONES

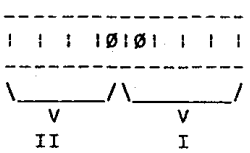
Las funciones en "C", tienen la forma siguiente:

```
tipo identificador(lista de parámetros)
declaración de parámetros
{

        Cuerpo de la función

}
```

Se maneja una tabla de simbolos en donde se almacenan todos los nombres de las variables que se encuentren en el programa fuente junto con su tipo. La tabla consiste de un arreglo de estructuras en donde cada estructura contiene dos miembros a saber : "sname[]", de tipo arreglo y "satt", definida como utiny (1 byte), el primero contiene el nombre de la variable y el segundo contiene el tipo junto con algún atributo de la siguiente forma :



La primera parte contiene el tipo básico definidos como :

CHAR	Ø1
INT	Ø2
FLOAT	Ø4
DOUBLE	Ø4

El segundo contiene la indicación de si tiene algún modificador :

PTR	Ø4Ø
NOP	ØØ

La tabla va a estar dividida en dos secciones. La primera contiene las variables globales al programa fuente; la segunda, que consiste de 5Ø localidades, almacena las variables locales. Para esto se manejan dos apuntadores: glob apunta el inicio de la sección de variables globales y loc el de las variables locales. Cada vez que se inicia una función, se activa la tabla de variables locales.

Una vez que el analizador sintáctico reconoce el fin de una función se inicializa la sección de variables locales. Esta tabla se utiliza para saber el tipo del valor que se va a imprimir.

En cuanto a la impresión, se manda desplegar el tipo, nombre de la función y los valores de los parámetros que se le pasan asi como las llaves que empiezan y terminan el cuerpo de la función.

Ej.:

ENTRADA	SALIDA AL MOMENTO DE EJECUCION

strcpy(s,t)
char s[],t[];
{
cuerpo de la funcion
}
}
	strcpy(hola,adios)
	{
	cuerpo de la funcion
	}

II) EXPRESIONES

Las expresiones se han dividido dentro de la gramática en los siguientes casos:

i) una lista de valores (lvalue)
ii) una expresión auxiliar (exp_aux)
iii)una expresión con operador coma (exp_coma)

i) En el caso de tratarse de una lista de valores (lvalue), se concatena la expresión y se mete al stack de valores. En cuanto al despliegue, se evalúa toda la expresión y se imprime sólo el valor final de ésta por medio de la función imp(). Se distingue el caso en que la lista va seguida de paréntesis cuadrados, pues entonces es necesario el token de "look-ahead".

Fue necesario manejar reglas vacias dentro de la gramática, para analizar el caso de arreglos de varias dimensiones y arreglos anidados

Ej.:

1) lvalor.identificador

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
fecha.mes;	junio;
ENTRADA	SALIDA AL MOMENTO DE EJECUCION
apun->mes;	junio;

ii) Dentro de las expresiones auxiliares encontramos :

a) una_exp.- En todos los casos se concatena el texto de la expresión y se guarda en el stack de valores. Se distinguen los siguientes casos, en los cuales un operador binario es igual a un operador unario : &lvalue,-exp, en los cuales se cambió el orden de precedencia de los operadores. En cuanto al despliegue se manda imprimir de igual forma que el de una lista de valores.

Ejemplos

1) ++lvalor

En este tipo de expresiones se manda imprimir el valor de lvalor, ya incrementado.

lvalor : n, en donde n es un entero con valor 9

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
++n	1Ø

2) --lvalor

En este tipo de expresiones, se manda imprimir el valor de lvalor, ya decrementado.

Ej.:

lvalor : a[i], en donde a[i] es un entero con valor 5

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
--a[i]	4

3) lvalor--

Ej.:

lvalor : a[i], en donde i es un indice con valor actual 1

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
a[i]--	1

4) &expresion

Ej.:

expresion : a[i], en donde a[ ] es un arreglo de caracteres.

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
&a[i]	314444

3) *expresion

Ej.:

expresion : ap, en donde ap -> arreglo de caracteres.

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
*(pa + i)	c

b) bin_exp.- Dentro de éstas, se van a distinguir dos clases de expresiones, las cuales se mandarán desplegar de diferente manera. El primer caso es el de las expresiones separadas por operadores aritméticos en cuyo caso se va a concatenar la expresión y se trata como un lvalue; el segundo por operadores de igualdad, relacionales, lógicos y condicionales. En este caso, se evalúa la expresión y se indica si resulta falsa o verdadera por medio de un "Ø" o un "1" respectivamente.

Ej.:

expresion1 : j > Ø

expresion1': j

expresion2': Ø

```
                op'   :  >

     expresion2 : v[j] < v[j+gap]

                expresion1'':  v[j]

                expresion2'':  v[j+gap]

                op''  :   <

     op         : &&
```

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
(j > f() && v[j] < v[j+gap])	(j > f() && v[j] < v[j+gap] ø)

Nos indica que la expresión toma el valor de falso

c)ass_exp

Tienen la siguiente forma :

exp operador de asignación exp

Se manda escribir cuando es un operador de asignación la exp seguida del operador de asignación (asgnop), y del valor final de la expresión. En el caso de tener en la expresión dos o más operadores de asignación, se manda desplegar el último.

Ej.

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
c=a[prin++]	c=d
x+=2	x=4
x*=2	x=4
a=b=c=8	a=8
fecha.mes = s;	fecha.mes = junio;

Se distingue el caso en que la primera expresión es un arreglo. Se manejó por medio de intercalar reglas vacias después del operador. Se utilizó un arreglo de estructuras para guardar el nombre del arreglo asi como los indices en el caso de tratarse de un arreglo de varias dimensiones. Es el único caso en el que es necesario la declaración de variables auxiliares, para el manejo de los indices, sin alterar la lógica del programa fuente.

En este caso se utilizan funciones de impresión "imp_a#", donde "#" es la dimensión del arreglo. Es decir, existen las funciones "imp_a1" para arreglos de una dimensión, "imp_a2" para arreglos de dos dimensiones, etc. Estas funciones requieren como parámetros el texto a imprimir, el índice (o los índices), la expresión a evaluar y las variables de indentación.

Ej.

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
arreglo[a+b] = exp	arreglo[10] = exp
arr_1[arr_2[a]] = exp	arr_1[5] = exp
arreglo[i][++i] = exp	arreglo[5][6] = exp
arreglo[arr[a]][b] = exp	arreglo[5][5] = exp

En el primer ejemplo la función de impresión va a tener la siguiente forma :

```
imp_a1("arre[%d]=exp",t_1=a+b,arreglo[t_1]=exp,line,tab);
```

d) expresión1 ? expresión2 : expresión3

Se concatena toda la expresión y se guarda en el stack de valores. Este tipo de expresiones se manda desplegar de acuerdo al tipo de la expresión1.

Ej.:

expresion1 : (a>b) en donde a y b toman los valores de 8 y 4

expresion2 : a

expresion3 : b

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
z=(a>b)? a : b	z=8

iii) expresión con operador coma .- Incluye todas las expresiones que contengan un operador coma. Sólo cabe mencionar que el tipo de despliegue va a ser de acuerdo al tipo de la expresión2.

Ej.

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
n++,k=n;	n++,k=2;

III) PROPOSICION CONDICIONAL

Las dos formas de una proposición condicional en "C" son las siguientes:

1) if(expresión) proposición
2) if(expresión) proposición else proposición2

En ambas se despliega el "if" seguido de la expresión

if(el valor de la expresión)

En caso de que la expresión resulte verdadera se despliegan a continuación las proposiciones correspondientes. En la proposición condicional del segundo tipo, si la expresión resulta falsa, se imprime el letrero del "else" seguida del desarrollo de la proposición2.

Ej.1: Caso en el que la expresión resulte verdadera.

Supongamos que un "if" tiene la siguiente expresión (n++/f)%2 asignada, la cual tiene un valor actual de 100; entonces el despliegue de la proposición al momento de ejecución será de la siguiente manera:

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
if((n++/f)%2)	if(100)

```
    proposicion1                desarrollo proposicion1
else proposicion2
```

como la evaluación de la expresión resultó verdadera a continuación del despliegue del "if" se imprime el desarrollo de la proposición1 .

Ej.2: Caso en el que la expresión resulte falsa. Supongamos que un "if" tiene la siguiente expresión "n > x && n != DIEZ" asignada, la cual al evaluarse resulta falsa; entonces el despliegue de la proposición al momento de ejecución será de la siguiente manera:

```
ENTRADA                          SALIDA AL MOMENTO DE
                                       EJECUCION

if(n > x && n != DIEZ)           if(n >x  && n!=DIEZ   Ø)
  proposicion1                   else
else proposicion2                desarrollo proposicion2
```

como la evaluación de la expresión no resultó verdadera a continuación del despliegue del "if" se imprime el letrero del "else" seguido del desarrollo de la proposición2.

En este caso la rutina de impresión es insertada después del "else", razón por la cual aunque la proposición2 consista de una sola proposición deberá ser convertida en un conjunto de proposiciones. Para esto es necesario adherir, durante el análisis sintáctico, una llave "{" en el archivo fuente después de cada token "else" que se reconozca con el fin de insertar la rutina de impresión correspondiente, sin alterar el curso del programa. Por lo consiguiente hay que sumar otra llave "}" al terminar la proposición2. Por lo tanto, cada vez que el analizador sintáctico reconozca el token "else", la acción semántica asociada a la regla correspondiente modificará el archivo fuente del usuario en la siguiente forma :

```
ARCHIVO FUENTE          ARCHIVO GENERADO POR LOC_C

else                      else{
      a=Ø;                        prin_tf("else");
                                  imp("a=Ø",a=Ø);
                            }
```

IV) ACCIONES

Una acción en "C", consiste de alguna de las siguientes proposiciones:

1).- Proposición break
2).- Proposición continue
3).- Proposición return
4).- Proposición goto ID
5).- Proposición ID :
6).- Proposición return(expresión)

En los tres primeros casos se trata de proposiciones en las cuales no se tiene ninguna expresión asignada, por lo que la acción correspondiente a estas proposiciones es la de insertar en el archivo fuente una rutina de impresión conteniendo como parámetro sólo un apuntador al tipo de proposición. Esta rutina aparece en el archivo fuente antes de la proposición para lo que es necesario abrir y cerrar las respectivas llaves . En el caso de una acción del tipo 4 , primero se concatenan el token "GOTO" con el valor del token ID y se maneja como una proposición del los tipos anteriores. En resumen, la acción semántica asociada a cualquiera de estas reglas será modificar el archivo fuente de la siguiente manera :

```
ARCHIVO FUENTE          ARCHIVO GENERADO POR LOC_C

accion                    {
                                prin_tf("accion");
                                accion
                          }
```

```
ARCHIVO FUENTE          ARCHIVO GENERADO POR LOC_C

id :                      id :
proposicion               {
                              prin_tf("id :");
                              desarrollo de la
                              proposicion
                          }
```

Ej.:

```
ENTRADA                 SALIDA AL MOMENTO DE
                              EJECUCION

   return;                    return;
```

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
goto id;	goto id;
ENTRADA	SALIDA AL MOMENTO DE EJECUCION
id :	id :

En el caso de tratarse de una acción del tipo 5, se genera la rutina de impresión de acuerdo a la expresión asociada.

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
return((c==EOF)?NULL:s);	return(+);

IV) PROPOSICION SWITCH

La forma de un switch en "C" es la sig.:
switch(expresión) proposición

En este caso, sólo se despliega, el caso que se evalúe.

Ej.:

ENTRADA	SALIDA AL MOMENTO DE EJECUCION
switch(c=getchar())	switch(c = +)
{	{
case '*' : proposicion;	case '+' : proposicion;
break;	break;
case '-' : proposicion;	}
break;	
case '/' : proposicion;	
break;	
case '+' : proposicion;	
break;	
default : proposicion;	
break;	
}	

V) CICLOS

Las tres formas de un ciclo, en "C", son las sig.:

1) while(expresión) proposición

2) do proposición while(expresión)

3) for(expresión-1;expresión-2;expresión-3)
   proposición

Una proposición de esta forma está representada dentro de la gramática por la siguiente regla:

```
loop : WHILE '(' exp ')'  W stat
```

la regla W representa una regla vacia, la cual se intercaló con el fin de generar la rutina de impresión correspondiente una vez que el analizador sintáctico ha reconocido :

```
while '(' exp ')'
```

El segundo tipo de un ciclo se maneja de la misma forma, solo que se acude a la regla vacia una vez que el desarrollo de la proposición se ha analizado:

```
"DO" stat 'WHILE '(' exp ')' W ';'
```

En cuanto al "FOR" que tiene la siguiente forma dentro de la gramática

```
loop : FOR '(' OExp;' PC FO OExp ';' F1 OExp)
       ( acciones ) stat
```

se maneja de diferente forma ya que se intercalan acciones antes de que la regla termine con el fin de guardar la primera expresión, para una vez que el analizador sintáctico termine el reconocimiento de la tercera expresión concatenarla y guardarla en el stack de valores.

En cuanto a la forma de despliegue en los tres casos va a depender de la expresión que la proposición tenga asignada.

Ej.:

ENTRADA	SALIDA AL MOMENTO DE EJECUCION

```
while((c=getchar()) != EOF)
        proposicion
```

```
while((c=getchar()) != EOF  Ø)
        proposicion
while((c=getchar()) != EOF  Ø)
   proposicion
while((c=getchar()) != EOF  1)
```

ENTRADA

```
for(ini=Ø;pal!=' ';ini++)
        proposicion
```

SALIDA AL MOMENTO DE EJECUCION

```
for(ini=Ø;pal!=' ' 1;ini++)
     proposicion
for(ini=Ø;pal!=' ' 1;ini++)
     proposicion
for(ini=Ø;pal!=' ' Ø;ini++)
```

ENTRADA

```
do
proposicion
while(a=b);
```

SALIDA AL MOMENTO DE EJECUCION

```
do
  proposicion
while(a=2);
do
  proposicion
while(a=4);
do
  proposicion
while(a=Ø);
```

iii) Archivos que configuran el sistema

En la creación de LOC_C se utilizaron los siguientes archivos :

1.- "ma.c", archivo que contiene la función principal del sistema (main()), la cual llama a la función yyparse(), que representa al analizador sintáctico y a la función yyerror, lo que nos indica que el programa fuente contiene un error de sintaxis.

2.- "c.y", el cual contiene las especificaciones por medio de una gramática libre de contexto para el lenguaje "C". Incluye el archivo "sym.h" .

3.- "sym.h", archivo que contiene los encabezados para el manejo de la tabla de simbolos utilizada para el manejo de las variables.

4.- "y.tab.c", archivo producido por yacc y que contiene la función yyparse(). Contiene además las redefiniciones de las funciones yywrap() y yyerror(), utilizadas por yyparse(). La función yyerror() imprime un mensaje cuando se detecta un error de sintaxis en el archivo fuente del usuario.

5.- "y.tab.h", archivo creado por yacc , contiene las definiciones de los nombres que representan a los tokens.

6.- "sym.c",que contiene la definición de las funciones utilizadas en el código asociado a cada regla gramatical en el archivo de especificaciones, encargadas de crear el formato de impresión requerido por las rutinas. Asi mismo contiene las funciones para concatenar cuerdas y checar tipos.

7.- "simb.c", archivo que contiene las funciones que se encargan de definir el tipo de valor que se tiene que imprimir, ya sea entero, caracter o una cuerda de caracteres.

8.- "c.l", archivo que contiene las especificaciones necesarias para crear el analizador lèxico por medio de lex. Incluye el archivo y.tab.h .

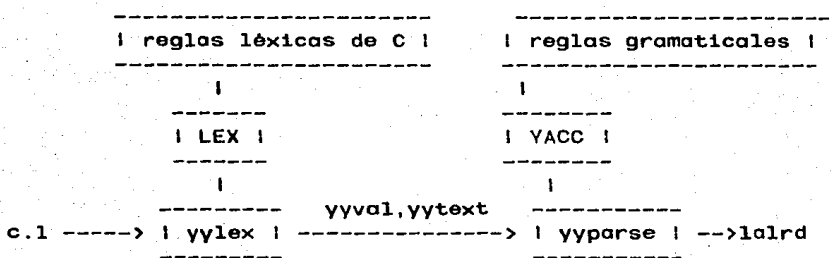
9.- "lex.yy.c" , archivo que es creado por lex y contiene la función yylex() la cual realiza el análisis lèxico.

10.-"decla.c" , archivo que contiene la función strsave(), utilizada en las reglas asociadas a las expresiones regulares. La función regresa un apuntador a la tabla de tokens.

11.-"makefile", el cual es un archivo requerido por el comando "make" (1) , por medio del cual se encuentran relacionados los 10 archivos anteriores para formar el programa lalrd (el cual realiza el análisis sintáctico).

12.-"dlib.c", archivo que contiene las diferentes rutinas de impresión y las funciones necesarias para manejar la pantalla.

13.-"lalrd", realiza el análisis tanto lèxico como sintáctico de un programa fuente y genera el archivo con el código de depuración. A continuación se muestra la interfase de lex y yacc utilizada en lalrd



14.-"LOC_C" se encuentra formado por un archivo de comandos para shell, cuya función es pasar el archivo fuente por el preprocesador del compilador de "C" para agregar en el análisis todos los fuentes incluidos por medio del #include "archivo". Una vez preprocesado el fuente, se le comunica a "lalrd" para generar el archivo con el código de depuración, para compilarlo y ligarlo con las rutinas de impresión incluidas en dlib.c. Borra todos los archivos intermedios.

III.3 LIMITES

LOC_C puede aceptar un archivo fuente que contenga como máximo cien variables globales y cincuenta variables locales.

Un arreglo puede tener como máximo 6 dimensiones.

La gramática no considera la inicialización de una variable en su declaración.

No se implementó la asignación de estructuras.

Puesto que el propósito de LOC_C es rastrear un programa con el fin de ayudar a la localización de errores de lógica, se da por hecho que el fuente del usuario, es un archivo sin errores de sintaxis.

CONCLUSIONES

1.- Seguir la ejecución de un programa con la ayuda de LOC_C, resulta sencillo debido a tres razones primordialmente : 1) se lleva una indentación, la cual corresponde a la que normalmente se utiliza en un programa escrito en "C", 2) se proporciona el número de línea en que se encuentra cada proposición en el archivo fuente del usuario, 3) el despliegue de algunas proposiciones importantes en modo de video inverso favorece la localización de ciclos infinitos y evita que sea monótono el rastreo.

2.- El ambiente proporcionado por el sistema operativo Unix, facilitó la elaboración de LOC_C gracias a las herramientas que proporciona como make, programa utilizado para configurar el sistema, ya que permitió controlar el proceso de compilación; yacc, programa que se utilizo'para generar una analizador sintáctico para efectuar el reconocimiento de las estructuras de alto nivel (como las proposiciones), a partir de una descripción gramatical del lenguaje; y lex, programa análogo a yacc , el cual se utilizó para construir un analizador léxico para efectuar el reconocimiento de estructuras de bajo nivel (como los identificadores), a partir de expresiones regulares.

3.- El taman~o de un programa objeto generado por medio de LOC_C , crece aproximadamente un 32.5% en comparación con el programa objeto del fuente del usuario, por lo que el programador debera determinar en funci'on del taman~o de su programa si le es útil LOC_C.

4.- Se comprobó la portabilidad de LOC_C , puesto que se implementó en una computadora ATT 7300, con sistema operativo UNIX SYSTEM V, y actualmente se encuentra funcionando en una computadora CODATA , con sistema operativo UNIX versión 7 . Portarlo de una máquina a otra implicó pasarlo y compilarlo sin efectuar modificación alguna , incluyendo los fuentes de Lex y de Yacc.

BIBLIOGRAFIA

[Aho 77] Aho Alfred, Ullman Jefrey,"Principles of Compiler Design " , Bell Laboratories, Murray Hill, New Jersey, 1977.

[Aho 86] Aho Alfred, Ullman Jefrey,"Compilers: Principles, Thectniques and Tools ", Addison - Wesley , 1986.

[AT&T 85] AT&T Unix Pc , Model 73ØØ , Unix System V , Programmer's guide .

[Fitz 81] Fitzhorn Patrick A. and Johnson Gearold R., " C : Toward a Concise Syntactic Description ", Colorado State University, Ft. Collins, ACM Sigplan Notices, Vol. 16, number 1Ø-12, December 1981.

[Harb 85] Harbison Samuel P. , Steele Guy L. Jr ., "C" a Reference Manual, Tartan Laboratories , Prentice-Hall, Inc, Englewood Cliffs, New Jersey 1985.

[John 78] Johnson Stephen C. , Yacc : Yet Another Compiler-Compiler , Bell Laboratories , Murray Hill, New Jersey, 1978.

[Kern 78] Kernighan B. W . and Ritchie D. M. , The "C" Programming Language, Prentice - Hall, Englewood

Cliffs, NJ 1978.

[Kenn --] Kenneth C.R. , C. Arnold , Screen Updating and Cursor Movement Optimization : A Library Package, Department of Electrical Engineering and Computer Science , University of California, Berkeley, California.

[Lesk 78] Lesk M.E. and Schmidt E. , Lex : A Analizer Generator , Bell Laboratories , Murray Hill, New Jersey, 1978.

[Mara 77] Maranzano J . F . , Bourne S . R . , "A Tutorial Introduction to ADB" , Bell Laboratories, Murray Hill, New Jersey 1977.

[Stef 84] Steffen Joseph L., " Experience with a Portable Debugging Tool ", Bell Laboratories, Naperville, Illinois , Software - Practice and Experience, Vol. 14(4) 323-334 , April 1984 .

ANEXO A

GRAMATICA DE "C"

* En lo sucesivo se abreviara :
*
* agg -> aggregate
* com -> complex
* cond -> conditional
* D -> data
* dec -> decalration
* decR -> declarator
* def -> definition
* exp -> expression
* ext -> external
* fun -> function
* hea -> header
* ID -> identifier
* mem -> member
* mod -> module
* oexp -> optional expression
* ocexp -> optional constant expression
* par -> parameter
* pro -> program
* qua -> qualifier
* sim -> simple
* spec -> specifier
* stat -> statement
* T -> type
*

* Convenciones :

una 'L' despues de una de la abreviaturas anteriores significa lista.

Lista de tokens :

ID
TIPOID
SWITCH
CASE

```
DEFAULT
IF
ELSE
FOR
WHILE
DO
BREAK
CONTINUE
GOTO
RETURN
SIZEOF
LONG
VOID
STATIC
STRUCT
UNION
SHORT
UNSIGNED
CHAR
FLOAT
DOUBLE
INT
ENUM
TYPEDEF
EXTERN
REGISTER
AUTO
CONS
STRING
INCREMENT       /* ++   increment */
DECREMENT       /* --   idem      */
MEMBER          /* ->   idem      */
SR              /* >>   shift right       */
SL              /* <<   shift left        */
LE              /* <=   less or  equal    */
GE              /* >=   greater or equal  */
EQ              /* ==   equal             */
NE              /* !=   not equal         */
LOR             /* ||   logical or        */
LAND            /* &&   logical  and      */
A_P             /* +=   assig plus        */
A_M             /* -=   assig minus       */
A_S             /* *=   assig star        */
A_D             /* /=   assig divition    */
A_MOD           /* %=   module            */
A_SR            /* >>=  shift right       */
A_SL            /* <<=  shift left        */
A_AND           /* &=   and               */
```

```
A_X             /* ^=   xor                */
A_OR            /* |=   or                 */


c_pro           :       /* empty */
                |       sc Ddec ';' c_pro
                |       fun_def c_pro
                ;


/*
 *      Function definitions :
 */

fun_def         :       sc Tspec FdecR aux_var_Ø par_dec
                        compound_stat
                |       sc FdecR  par_dec compound_stat
                ;

FdecR           :       SID '('  ')'
                |       SID '(' parL ')'
                |       '*' FdecR
                |       FdecR '[' ']'
                |       FdecR '(' ')'
                |       '(' FdecR ')'
                ;

SID             :       ID
                |       '(' ID ')'
                ;

parL            :       ID
                |       ID ',' parL
                ;

par_dec         :       /* empty */
                |       sc Tspec par_def ';'
                ;

par_def         :       DdecR
                |       DdecR ',' par_def
                ;

/*
 *      data declarations :
 */

internalDdec    :       /* empty */
```

```
                |       sc Ddec ';' internalDdec
                ;

Ddec            :       Tspec Dspec
                |       Tspec Fdec_aux
                ;

Fdec_aux        :       FdecR
                |       FdecR ',' Fdec_aux
        ;

Dspec           :       DdecR
                |       DdecR ',' Dspec
                ;

DdecR           :       SID
                |       CT
                ;

CT              :       BT
                |       '*' CT
                |       CT '[' ocexp ']'
                |       CT '(' ')'
                |       '(' CT ')'
                ;

BT              :       SID '[' ocexp ']'
                |       '*' SID
                ;

ocexp           :       /* empty */
                |       CONS
                ;

/*
 *      statmets :
 */

statL           :       /* empty */
                ;

stat            :       compound_stat
                |       switch_stat
                |       cond_stat
                |       loop
                |       action
                |       null
                |       expresion ';'
```

```
                ;

compound_stat   :       '{' internalDdec statL '}'
                ;

switch_stat     :       SWITCH '(' expresion ')' '{'
                        caseL default '}'
                ;

caseL           :       case_stat
                |       case_stat caseL
                ;

case_stat       :       CASE CONS ':' statL
                ;


default         :       /* empty */
                |       DEFAULT ':' statL
                ;

cond_stat       :       IF '(' expresion ')' stat
                |       IF '(' expresion ')'  stat ELSE
                        stat
                ;


loop            :       WHILE '(' expresion ')'  stat
                |       DO stat WHILE '(' expresion ')' ';'
                |       FOR '(' oexp ';' oexp ';' oexp ')'
                        stat
                ;

action          :       BREAK ';'
                |       CONTINUE ';'
                |       GOTO ID ';'
                |       ID ':'  stat
                |       RETURN ';'
                |       RETURN expresion ';'
                ;


null            :       ';'
                ;

/*
 *      expressions :
```

```
 */

expresion       :       exp
                |       exp_coma
                ;

exp_coma        :       expresion ',' expresion
                ;

exp             :       lvalue
                |       exp_aux
                |       '(' exp_aux ')'
                ;

exp_aux         :       una_exp
                |       '(' exp_coma ')'
                |       bin_exp
                |       ass_exp
                |       con_exp
                |       nlvalue
                ;

una_exp         :       '&' lvalue      %prec '!'
                |       '-' exp         %prec '!'
                |       '!' exp
                |       '~' exp
                |       INCREMENT lvalue
                |       DECREMENT lvalue
                |       lvalue INCREMENT
                |       lvalue DECREMENT
                |       '(' Tname ')' exp
                |       SIZEOF exp
                |       SIZEOF Tname
                ;

bin_exp         :       exp '*' exp
                |       exp '/' exp
                |       exp '%' exp
                |       exp '+' exp
                |       exp '-' exp
                |       exp SR exp
                |       exp SL exp
                |       exp '>' exp
                |       exp '<' exp
                |       exp LE exp
                |       exp GE exp
                |       exp EQ exp
                |       exp NE exp
```

```
                |       exp '|' exp
                |       exp LOR exp
                |       exp '&' exp
                |       exp LAND exp
                |       exp '^' exp
                ;

ass_exp         :       exp '=' exp
                |       exp A_P exp
                |       exp A_M exp
                |       exp A_S exp
                |       exp A_D exp
                |       exp A_MOD exp
                |       exp A_SR exp
                |       exp A_SL exp
                |       exp A_AND exp
                |       exp A_X exp
                |       exp A_OR exp
                ;

con_exp         :       exp '?' con_aux
                ;


con_aux         :       exp ':' exp
                ;

nlvalue         :       CONS
                |       STRING
                |       ID '(' expL ')'
                ;

lvalue          :       ID
                |       lvalue '.' ID
                |       lvalue MEMBER ID
                |       '*' lvalue
                |       '*' nlvalue
                |       '*' '(' exp ')'
                |       '(' lvalue ')'
                |       lvalue '['  exp  ']'
                ;

oexp            :       /* empty */
                |       expresion
                ;

expL            :       /* empty */
                |       exp
```

```
                |       exp ',' expL
                ;

/*
 *      type analysis
 */

Tspec           :       simT
                |       aggT
                ;

simT            :       LONG T
                |       SHORT T
                |       T
                ;

T_pe            :       /* empty */
                |       TYPEDEF
                ;

T_un            :       /* empty */
                |       UNSIGNED
                ;

otros           :       TIPOID
                ;

T               :       otros
                |       T_pe T-un INT
                |       T_pe T_un CHAR
                |       T_pe T_un FLOAT
                |       T_pe T_un DOUBLE
                ;


aggT            :       T_pe STRUCT agg_decR
                ;

agg_decR        :       ID '{' memL '}'
                |       '{' memL '}'
                |       ID
                ;

memL            :       mem
                |       mem memL
                ;

mem             :       Ddec ';'
```

```
                ;

Tname           :       INT
                |       CHAR
                ;

sc              :       /* empty */
                |       REGISTER
                |       STATIC
                |       AUTO
                ;
```