FACULTAD DE INGENIERIA

UNIVERSIDAD NACIONAL AUTOMOMA DE MEXICO

ALEJANDRO JIMENEZ HERNANDEZ

APUNTES DE COMPILADORES

DIVISION DE INGENIERIA MECANICA Y ELECTRICA
DEPARTAMENTO DE INGENIERIA EN COMPUTACION

INTRODUCCION

La estructura de éstos apuntes de compiladores se basan en lo siguiente:

- se presentan los principios básicos junto con ejemplos de ese principio.
- los ejemplos no necesitan de programación auxiliar para trabajar, permitiendo al estudiante el experimentar desde el inicio con ellos.
- los ejemplos fueron escritos en lenguaje C que permite una mejor comprensión de las herramientas profesionales para generar los compiladores comerciales que existen en el mercado.
- proporcionar los elementos básicos para entender la construcción de compiladores.
- proporcionar un complemento o manual de laboratorio para trabajos, proyectos y tareas en la materia de compiladores.

Estos apuntes tratan de motivar al estudiante para que entienda, modifique y mejore los ejemplos presentados.

Alejandro Jiménez Hernández

INDICE

1. ESTRUCTURA DE UN COMPILADOR

INTRODUCCION

Lenguaje en su concepto más amplio es un conjunto de símbolos, sonidos y actitudes que se utilizan para expresar ideas y sentimientos entre seres humanos.

Asi un lenguaje natural como el español es tan amplio y versátil, que podemos dar ordenes, suplicar, expresar admiración, temor, odio, alegría, amor o simplemente comunicar lo que pensamos. Una clase especial de lenguaje más restringido en expresión es el que se utiliza para comunicarse con máquinas como las computadoras.

Una gran parte de lenguajes de computadora, solo tienen un modo de expresión que es el imperativo y que solo transmite órdenes de un ser humano a una computadora. Un programa en este tipo de lenguajes es una serie de instrucciones que debe de ejecutar la computadora. La función principal de éstos lenguajes es ordenar acciones de control a la computadora para obtener algún resultado y no expresar ideas. Un objetivo que se debe de cumplir al escribir un programa de éste tipo es el dar la secuencia correcta para realizar éstas acciones.

La necesidad de utilizar las computadoras en un mayor número de aplicaciones, el avance de la tecnología de circuitos digitales, y el aumento del costo en la producción de programas con respecto a los circuitos obligaron a desarrollar una nueva generación de lenguajes llamados declarativos.

En este tipo de lenguajes a partir de sentencias o expresiones del tipo 'si existe esta condición, se tiene una determinada conclusión' se pueden deducir o inferir soluciones a problemas. En éstos lenguajes se expresa conocimiento y se obtienen respuestas a partir de este sin tener que declarar como

lo debe de ejecutar la computadora. Por ejemplo si tenemos expresiones del tipo:

"todos los humanos son mortales" y " soy humano"

tendremos como conclusión:

"soy mortal"

TRADUCTOR

Un traductor es un programa que convierte un programa que se llama 'fuente' en otro 'objeto'. El programa fuente esta escrito en un lenguaje llamado fuente y el programa objeto en forma de lenguaje objeto.

Figura 1. Traductor

COMPILADOR

Es un traductor que convierte un programa escrito en un lenguaje de alto o medio nivel como C, PASCAL, MODULA, FORTRAN o COBOL en lenguaje de máquina o ensamblador. Posteriormente este programa objeto es procesado para su ejecución. El programa fuente y los datos del programa se procesan separadamente o en dos fases diferentes que son la compilación y la ejecución.

INTERPRETE

Es un programa que traduce y ejecuta al mismo tiempo un programa escrito en un lenguaje de alto nivel como BASIC, LISP, LOGO, APL, SMALLTALK. Su función es la traducción de un programa fuente y el procesamiento de los datos que utiliza ese programa.

JERARQUIA DE LENGUAJES

La jerarquía de lenguajes indica el grado de dependencia que tienen estos respecto de una máquina o procesador en especial y se pueden clasificar como:

- lenguajes de máquina
- lenguajes ensambladores
- lenguajes de nivel medio
- lenguajes de alto nivel de uso general
- lenguajes de propósito especial

- lenguajes de inteligencia artificial
- lenguajes naturales

El lenguaje de máquina es el 'nativo' de cada computadora y describe la estructura particular de esta. Cada instrucción y dirección dentro de la máquina son un código numérico. Para comunicarse con una computadora por medio de este lenguaje no se necesitan traductores.

El lenguaje ensamblador es una versión simbólica del lenguaje de máquina que tiene una relación uno a uno. Los códigos de las instrucciones se traducen a nemónicos tales como MOV, ADD, MUL y SUB y las direcciones toman nombres simbólicos. El lenguaje ensamblador permite el uso más eficiente de los recursos de la computadora.

Los lenguajes de nivel medio como C, permiten el manejo eficiente de las instrucciones de ensamblador y las estructuras de control de flujo, repetición y secuencia de un lenguaje de alto nivel.

Los lenguajes de alto nivel como PASCAL, ADA, MODULA permiten el tener independencia de la estructura interna de la computadora al manejar estructuras de control de flujo, repetición y secuencia de un lenguaje de alto nivel.

Los lenguajes de propósito especial son de alto nivel y permiten la solución de problemas específicos en determinadas áreas como SQL para manejo de bases de datos.

Los lenguajes de inteligencia artificial permiten el declarar el conocimiento explícitamente, aquí las estructuras de control de flujo, repetición y secuencia no se expresan en el programa, por lo que existe una separación entre lógica y control de la computadora.

FASES DE UN COMPILADOR

Los compiladores pueden ser clasificados de acuerdo a su construcción como de una pasada, dos pasadas, pasadas multiples, o dependiendo de la manera en que fue construido u otra característica especial como optimizado o depurador.

El compilador en general como se muestra en la figura 2 se divide en dos fases: análisis y síntesis.

La parte de análisis se compone de lo siguiente:

- 'scanner' o analizador de léxico. Este separa el programa fuente en 'tokens', que son fragmentos de texto o unidades básicas como palabras reservadas, variables, símbolos aritméticos, de relación, etc. de acuerdo a su significado. Además quita comentarios, procesa directivas de control y almacena los tokens en la tabla de símbolos.

- 'parser' o analizador de sintaxis. Verifica el correcto agrupamiento de los 'tokens' en frases gramaticales, que se representan para su análisis en un árbol de parsing.

- analizador semántico. Verifica la consistencia de las expresiones o el significado del programa localizando errores. ejem. verificar que cada operador tenga operandos del mismo tipo, valor o con una relación lógica adecuada.

La parte de síntesis está formada por:

- generador de código intermedio. Produce código para una máquina abstracta que sea fácil de simplificar y traducir.

- optimizador de código. El código intermedio se analiza y transforma en código equivalente que es más eficiente.

- generador de código. Produce código en ensamblador o código ejecutable.

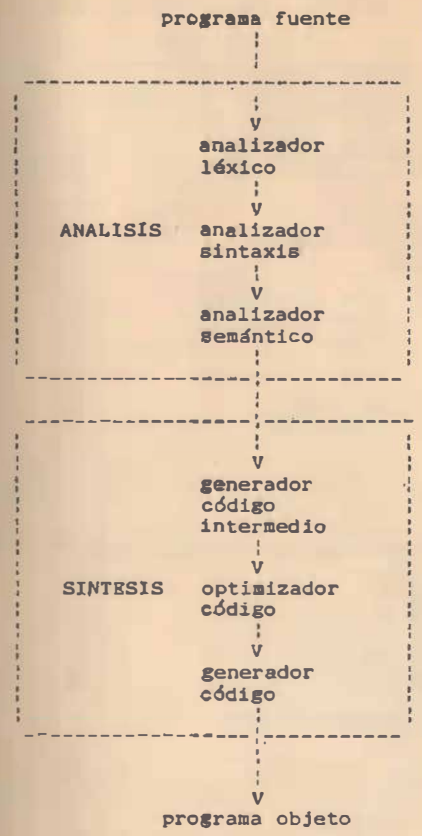


Figura 2 Fases de un Compilador

EJEMPLO: TRADUCTOR DE EXPRESIONES

El siguiente ejemplo traduce código fuente en forma de expresiones aritméticas de suma, resta, multiplicación, división y división entera, con operandos enteros e identificadores, a código objeto de una máquina de 'stack' o pila en notación polaca sufija.

El programa tiene la estructura de un compilador con analizador de léxico, de sintaxis, generador de código, tratamiento de errores y manejo de tabla de símbolos. Esta escrito en lenguaje 'C' y se muestra en la figura 3.

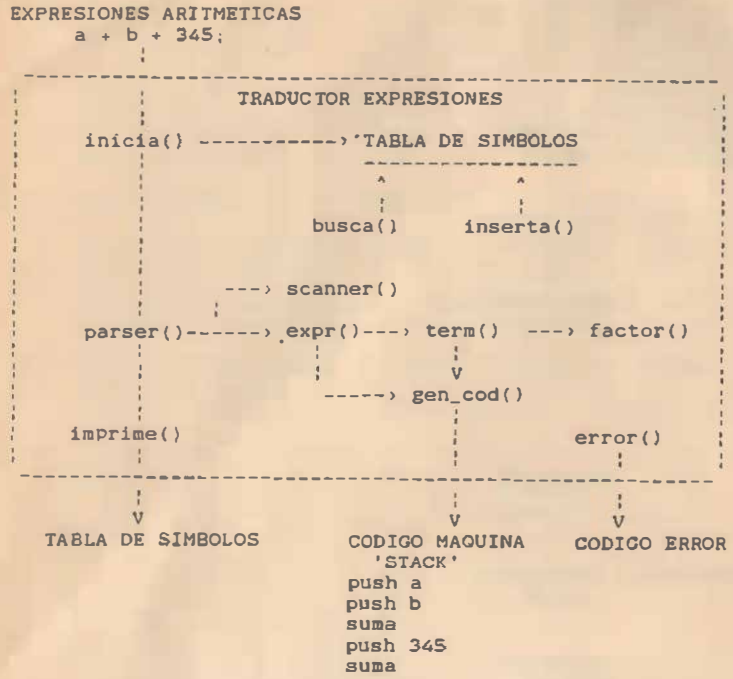


Figura 3 Ejemplo traductor expresiones

Las funciones realizan lo siguiente:

inicia(): inicializa la tabla de símbolos.

scanner(): separa el programa en 'tokens' y los almacena en la tabla de simbolos.

parser(): verifica que las expresiones esten sintácticamente correctas de acuerdo a la gramática.

error(): imprime los mensajes de error.

gen_cod(): genera el código de máquina de 'stack'.

ANALIZADOR DE LEXICO. scanner()

Esta función lee caracteres, los agrupa en 'tokens' y les asocia un número entero. Elimina los espacios, tabulaciones y

regresos de linea. Los 'tokens' que reconoce son:

identificadores números
operadores aritméticos: *,/,+,-,mod y div
operadores para agrupar: (,)
separar: ;
y fin de archivo: EOF

Se muestra en la figura 4.

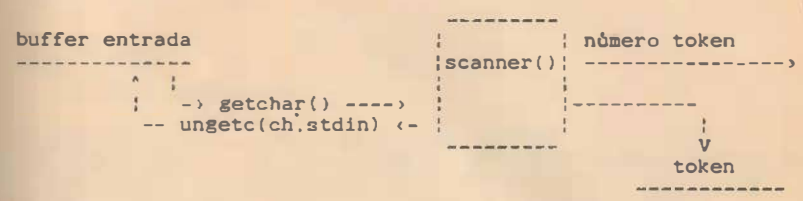


Figura 4 Funcionamiento del scanner()

ANALIZADOR DE SINTAXIS. parser()

El analizador de sintaxis es recursivo descendente y verifica la siguiente gramática de contexto libre:

```
inicio -> lista EOF
lista  -> expr ; lista | EPSILON
expr   -> term términos
términos -> + term términos | - term términos | EPSILON
term   -> factor factores
factores -> * factor factores | / factor factores |
            EPSILON
factor -> (expr) | ID | NUM
```

El analizador de lèxico o 'scanner' verifica el lenguaje que generan las siguientes expresiones regulares:

```
ID  = letra (dígito|letra)*
NUM = dígito dígito*
dígito = 0 | 1 | ... | 9
letra  = a | ... | z | A | ... | Z
```

TABLA DE SIMBOLOS

La tabla de símbolos se compone de un registro formado por dos campos: un apuntador al 'token' almacenado y la identificación o número de 'token', como se muestra en la figura 5 . Este arreglo se utiliza para ahorrar espacio al almacenar 'tokens' de diferente longitud.

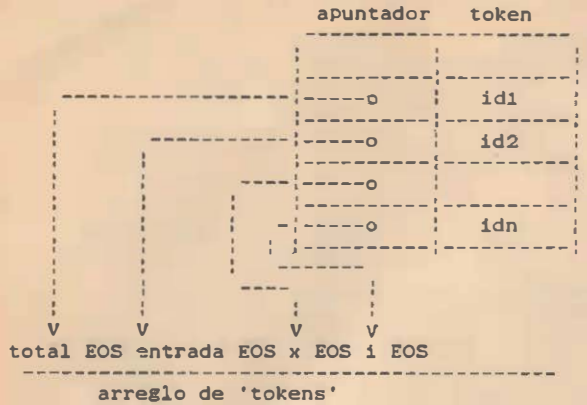


Figura 5 Estructura de la tabla de símbolos para los 'tokens' total, entrada, x e i.

Para manejar la tabla de símbolos se utilizan dos funciones:

busca(): dado un identificador, lo busca en la tabla de símbolos.

inserta(): dado un identificador lo inserta en la tabla se símbolos.

ASOCIATIVIDAD DE OPERADORES

Cuando un operando tiene operadores a su izquierda y derecha este se asocia a la izquierda. Así para las cuatro operaciones aritméticas existe esta convención de asociarse a la izquierda. Una excepción es la exponenciación que se asocia a la derecha.

PRECEDENCIA DE OPERADORES

En los operadores el orden de precedencia es mayor para * / que tienen precedencia sobre + -

Esto se expresa en las producciones de la siguiente manera:

+ -	asociación a la izquierda
* /	asociación a la izquierda

LISTADO DEL PROGRAMA

El programa trabaja en compiladores como TurboC y Microsoft. Se incluye a continuación el listado del programa.

Para rastrear su operación se debe de dar un valor de inicio de uno a la bandera de rastrea.

```
/*              FACULTAD DE INGENIERIA
    APUNTES DE COMPILADORES
    programa que traduce expresiones infijas
    a código máquina stack

                                                */

#include "stdio.h"

/*          definiciones globales                */

#define BUFFER 128      /* tama&o buffer entrada para token */
#define MAXIMO 100      /* número máximo de símbolos en tabla */
#define OTRO -1         /* otro token diferente a NUM,DIV,MOD,IF */
#define EOS '\0'        /* fin de cadena          */
#define MAX_IDS 999     /* tama&o máximo de arreglo de identificadors *

#define NUM 256     /* códigos asignados a tokens */
#define DIV 257
#define MOD 258
#define ID  259
#define FIN 260

int codigo_token;     /* valor del token */
int num_linea;        /* número línea entrada */
int rastrea = 0;      /* realiza rastreo programa */

struct tabla  {           /* estructura de tabla de símbolos */
       char *ptr_toks;    /* apuntador a arreglo de identificadors */
       int token;         /* código token */
       };

struct tabla tabla_simbolos[MAXIMO], *p;  /* declara tabla símbolos */

void inicia(), letrero(), parser(), imprime();

main()

{
  inicia();
  letrero();
  parser();
  imprime();
  exit(0);
 }



/*                  inicia.c                    */
/*  inicializa tabla de símbolos                */
```

```
void inicia()  /* carga palabras reservadas en tabla de símbolos */
{
struct tabla reservadas[] = {
            "div",DIV,
            "mod",MOD,
            0,0,        };

     for (p = reservadas; p->token; p++)
           inserta(p->ptr_toks,p->token);
     }


/*      función: scanner
    analizador de léxico o 'scanner'
    reconoce NUM, ID, FIN,
                                        */

#include "ctype.h"

char buffer[BUFFER];

int scanner()

{
 int t;
   num_linea = 1;
   codigo_token = OTRO;

 while(1) {
       t = getchar();      /* lee lookahead        */
       if (t == ' ' || t == '\t');  /* quita blancos  */
       else if (t == '\n')       /*  quita líneas  */
             num_linea = num_linea + 1;  /* pero cuentalas */
       else if (isdigit(t))  {  /* lee enteros  */
             ungetc(t,stdin);    /*  regresa lookahead  */
             scanf("%d", &codigo_token);  /* lee token  */
             return NUM;
             }
        else if (isalpha(t))  {  /* lee identificadores */
             int p, b = 0;
             while(isalnum(t))  {
                   buffer[b] = t; /* guarda
                                     caracteres identificador */
                   t = getchar();
                   b = b + 1;
                   if (b >= BUFFER)
                         error("error: longitud ID ");
             }
             buffer[b] = EOS; /* fin identificador */
             if (t != EOF)
                   ungetc(t,stdin);
             p = busca(buffer);   /*  búscalo en tabla símbolos */
             if (p == 0)          /* p es índice en tabla */
             p = inserta(buffer,ID);  /* no existe insertalo  */
```



```
                   codigo_token = p;
                   return tabla_simbolos[p].token;   /* apuntador
                                                        identificador */
                    }
             else if(t == EOF)
                   return FIN;    /* fin file termino  */
                   else {
                        codigo_token = OTRO;  /* era +, - */
                        return t;
                   }
             }
}


/*   función: parser
     analizador de sintaxis
     recursivo descendente                          */

int lookahead;
void parser()

{
       printf(">");   /* imprime cursor */
       lookahead = scanner();        /*  lee token   */
       while (lookahead != FIN ) {
             expr(); compara(';');   /* realiza parsing recursivo */
             }
}


expr()
{
       int t;

       term();      /*  regla expr  */
        if (rastrea==1)
           printf("estoy en expr;\nlei token = %c\n",lookahead);
       while(1)
           switch(lookahead)  {
           case '+' : case '-' :  /* regla términos */
                t = lookahead;
                compara(lookahead); term(); gen_cod(t,OTRO);
                continue;
           default:
             printf(">");   /* imprime cursor */
                return;
           }
}


/*      regla producción term   */
term()

{
       int t;
```

```
        factor();
        if (rastrea==1)
           printf("estoy en term;\nlei token = %c\n",lookahead);
        while(1)
            switch(lookahead) {    /*  regla factores  */
            case '*': case '/' : case DIV : case MOD:
                      t = lookahead;
                      compara(lookahead);
                      factor(); gen_cod(t,0TRO);
                      continue;
            default:
                     return;
            }
}


factor()
{
        if (rastrea==1)
           printf("estoy en factor;\nlei token = %d\n",lookahead);
        switch(lookahead)  {      /* regla factor */
            case '(' :
                 compara('('); expr(); compara(')'); break;
            case NUM :
                 gen_cod(NUM,codigo_token); compara(NUM); break;
            case ID :
                 gen_cod(ID,codigo_token); compara(ID); break;
            default:
                 error("error de sintaxis");
        }
}


compara(t)
int t;

{
        if (lookahead == t)
           lookahead = scanner();
        else error("error de sintaxis");
}


/*      funcion: gen_cod                          */

gen_cod(t,tval)       /* genera código o salida final */
int t,tval;

{
     switch(t)  {
            case '+': case '-': case '*': case '/':
                 printf("%c\n",t);
                 break;
            case DIV:
                 printf("DIV\n");
```



```
                 break;
            case MOD:
                 printf("MOD\n");
                 break;
            case NUM:
                 printf("push %d\n", tval);
                 break;
            case ID:
                 printf("push %s\n",tabla_simbolos[tval].ptr_toks);
                 break;
            default:
                 printf("token %d, codigo token %d\n",t,tval);
            }
}


/*      función letrero                           */

void letrero()

{
        printf("\t\tTRADUCTOR EXPRESIONES\n");
       printf("expresiones con operadores(+,-,*,/,div,mod)\n");
       printf("terminados con ;\n");
       printf("para terminar programa teclee CTRL-Z\n\n");
}


/*    función error                               */

error(m)        /* imprime mensajes de error */
char *m;

{
      fprintf(stderr, "linea %d: %s\n", num_linea, m);
      exit(1);
}



/*   función: tabla                                */
/*   manejo de tabla de símbolos                   */

char ids[MAX_IDS];  /* arreglo para almacenar identificadores */
int fin_ids = -1;   /* apuntador a última posición identificadores */
int fin_tabla = 0;   /* apuntador a última posición tabla símbolos */

int busca(s)  /* busca identificador en tabla símbolos  */
char s[];

{
   int p;

   for (p = fin_tabla; p>0; p = p-1)
       if (strcmp(tabla_simbolos[p].ptr_toks, s) == 0)
```

```
            return p;       /* regresa posición en tabla */
    return 0;               /* no esta en tabla */
}

/* inserta identificador y regresa su posición */

int inserta(s,cod_tok)
char s[];       /* en tabla de símbolos */
int cod_tok;

{
    int longg;      /* logitud identificador  */

    longg = strlen(s);
    if (fin_tabla + 1 >= MAXIMO)
            error("tabla de simbolos llena...");
    if (fin_ids + longg + 1 >= MAX_IDS)
            error("arreglo de identificadores lleno...");
    fin_tabla = fin_tabla + 1;   /* aumenta tama&o tabla */
    tabla_simbolos[fin_tabla].token = cod_tok;
    tabla_simbolos[fin_tabla].ptr_toks = &ids[fin_ids + 1];
    strcpy(tabla_simbolos[fin_tabla].ptr_toks,s);
    fin_ids = fin_ids + longg + 1;
    return fin_tabla;
}


void imprime()

{
    int p;

    for (p = 1; p <= fin_tabla ; p++)
        printf("%s\t%d\t\n",tabla_simbolos[p].ptr_toks,
                tabla_simbolos[p].token);
}
```


2. LENGUAJES FORMALES

El desarrollo de teorías formales en lenguajes de programación se ha limitado a los lenguajes de alto nivel, tales como Pascal, Modula, Ada y otros. Los lenguajes de bajo nivel como el de máquina y ensamblador son usualmente considerados como parte de la descripción de una computadora y como tales están dentro del conjunto de la teoría de máquinas secuenciales.

TEORIA DE LENGUAJES

La teoría de los lenguajes naturales ha sido desarrollada durante años por filósofos, pensadores, lógicos y lingüistas. De acuerdo a este modelo desarrollado, un lenguaje puede ser descrito por tres características: SINTAXIS, SEMANTICA Y PRAGMATICA

La teoría de lenguajes ha sido aplicada a los lenguajes de programación. Dado que esta teoría es preferentemente cualitativa, le da este enfoque al campo de los lenguajes de programación, enfatizando la distinción entre varios aspectos del lenguaje.

SINTAXIS Y GRAMATICA

Sintaxis es 'la especificación de la estructura de un lenguaje y la descripción de esta'. La gramática contiene las reglas para generar o reconocer estructuras correctas. La sintaxis asigna cada símbolo básico a una clase sintáctica elemental y esta, a su vez, sera combinada por sustitución dentro de otras clases sintácticas. Existen dos tipos de elementos: terminales y no terminales, El símbolo terminal denota un símbolo que forma parte del lenguaje y el no-terminal un símbolo que se utiliza para estructurar ese lenguaje sin formar parte de el.

La siguiente tabla ilustra algunas comparaciones entre los tipos sintácticos de lenguajes naturales y de programación.

LENGUAJES NATURALES	LENGUAJES DE PROGRAMACION
VOCABULARIO	VOCABULARIO
Símbolos Alfabéticos	Símbolos básicos
Marcas y Signos	Símbolos Elementales
	Alfabéticos
	Numéricos
	Marcas y Signos
	Símbolos Compuestos
Palabras	Identificadores
CLASES DE PALABRAS	CLASES DE PALABRAS
Sujetos	Dígito
Verbos	Letras
Adjetivos	Enteros
Adverbios, etc.	Variables, etc
CLASES SINTACTICAS DE SUB-ORACIONES	CLASES SINTACTICAS DE SUB-DECLARACIONES
Clausulas	Expresiones
Predicado	Aritméticas
Sujeto, etc.	Lógicas
	Condicionales
	Asignación de Variables, etc.
TIPOS DE ORACIONES	TIPOS DE DECLARACIONES
Declarativas	Declaraciones
Imperativas	Asignaciones
Interrogativas	Secuenciales
Exclamatorias	Procedimientos
PARRAFOS	BLOQUES O PROCEDIMIENTOS

La sintaxis de los lenguajes de programación es especificada de varias maneras. Los manuales de programación usualmente describen esta sintaxis en definiciones estrictas como los diagramas de sintaxis que eliminan ambigüedades Otra descripción es la forma BNF o Backus-Naur Form, la que describe la sintaxis mediante lo siguiente:

- los símbolos terminales que se representan a sí mismos.
- los símbolos no-terminales o clases sintácticas que son indicadas colocándolas a la izquierda de la expresión.
- el símbolo '->' o '::=' que define el símbolo sintáctico

a la izquierda en términos de otros símbolos a su derecha.
- el símbolo '¦' que separa definiciones alternativas.

EJEMPLO: Un identificador se define como una letra del alfabeto en el inicio, seguida por cualquier número de letras o dígitos.

```
identificador -> letra  ¦ identificador letra
                        ¦ identificador digito
letra -> A ¦ B ¦ ... ¦ Y ¦ Z
digito -> 0 ¦1 ¦ ... ¦ 9
```

Esta definición no supone un límite en el número máximo de caracteres que constituyen un identificador.

El BNF y formas equivalentes se usan para describir una parte de los lenguajes de programación. Se requieren proposiciones adicionales para dar una descripción sintáctica completa .

SEMANTICA

La semántica es definida como la relación entre un objeto y un grupo de significados. La palabra 'significado' tiene un sentido específico, considere por ejemplo una proposición tal como A = B + 3.5 / C . La semántica de esta proposición puede ser interpretada como: tome el número 3.5; divídalo por el valor de la variable C y asigne a la variable temporal T el resultado de esta operación; tome el valor de la variable B y súmelo al valor de la variable T; asigne a la variable A el resultado de la última operación.

Para que la descripción semántica anterior sea aceptable y se asigne un significado a la proposición original, es necesario que los términos como "divídalo", "súmelo", "asigne a", etc. sean :

- rigurosamente definidos, o
- considerados como 'primitivos'.

Las definiciones rigurosas para los términos anteriormente mencionados pueden ser dadas, pero para asegurar que no queden ambigüedades, es necesario que las definiciones expongan el efecto de cualquier operación en cualquier posible contexto. En este caso la semántica puede considerarse como un grupo de reglas que especifican el resultado de cualquier posible proposición en cualquier caso posible.

La asignación de ciertas operaciones como 'primitivas' es apropiada cuando las reglas de semántica son definidas por una máquina como una computadora, en cuyo caso la semántica puede

considerarse como la descripción de las proposiciones originales en términos de series de instrucciones en lenguaje de máquina.

PRAGMATICA

Una regla pragmática es aquella que "selecciona entre el grupo de significados atribuidos a un símbolo, aquel significado particular que es significativo para un usuario particular en un tiempo particular ". En el caso de un lenguaje de programación el usuario puede ser un ser humano o una máquina. En el último caso debemos considerar las propiedades de esa máquina como un reconocedor. la pragmática incluye procesos como la optimización.

GRAMATICAS FORMALES

DEFINICIONES.

Varias gramáticas han sido consideradas pero la aproximación mas adecuada hacia la creación de un modelo para lenguajes fue tomado por Chomsky en su estudio de gramáticas. Antes de describir esta teoria, se dan algunas definiciones formales para varios términos que se utilizan.

ALFABETO O VOCABULARIO

Un alfabeto es un conjunto finito no vacio de símbolos, donde estos símbolos para un lenguaje de computación son 26 letras, 10 digitos y símbolos especiales tales como (.)...+.-.*,/,=,:,;.{,}.

SENTENCIA

Una sentencia, cadena de caracteres o tokens sobre un vocabulario V o alfabeto es una secuencia de símbolos pertenecientes a V y que tienen un significado determinado. Asi para V = {a,b,c} todas los cadenas a, b, c, aaa, abc,aaaabbbaccc estan sobre V. Para un lenguaje de programación las sentencias serian: el identificador, palabras reservadas (if, then, while,...), operador :=, delimitadores ; u cualquier otro.

Si A es un alfabeto entonces A^* es el conjunto de todos los cadenas finitos que se pueden formar sobre A.

GRAMATICA

Una gramática de estructuras de frases es un cuadruple, G(V,T,S,P), formado por los siguientes elementos:

N que es un vocabulario finito de símbolos noterminales
T que es un vocabulario finito de símbolos terminales
P que es un conjunto de reglas de producción, rescritura o sustitución
Estas reglas aparecen en la forma 'Fi'-› 'PSI', donde 'FI' es llamado el lado izquierdo de la producción y 'PSI' llamado el lado derecho de la producción, ambos son símbolos sobre V y una flecha '-›' que significa "puede ser rescrito como".
S que es un símbolo no terminal y es llamado símbolo de inicio, cabeza del lenguaje o símbolo distintivo.

Además de que N /\ T = Ø. En otras palabras ningún símbolo es terminal y no terminal a la vez.

En lo sucesivo podemos utilizar la siguiente notación:

- las letras griegas como 'FI', 'PSI', etc., denotan palabras de terminales o no terminales.
- Las letras mayúsculas como A, B, etc denotan símbolos no terminales.
- Las letras minúsculas como a, b, etc. denotan símbolos terminales.

LENGUAJE

Dada una gramática G(V,T,S,P), el lenguaje L(G) es el conjunto de todas las cadenas de símbolos terminales que pueden ser generadas mediante las producciones P. comenzando con S.

$$L(G) = \{ w : S \rightarrow^* w, w \in T^* \}$$

Los lenguajes y gramáticas no están en correspondencia de uno a uno, de esta forma algunas gramáticas diferentes pueden referirse al mismo lenguaje, siendo uno de los problemas importantes para un diseñador de lenguajes la elección adecuada de una gramática para su lenguaje. Para escoger una gramática, dos importantes criterios deben de considerarse:

- Posibilidad de análisis sintácticos o 'parsers' eficientes.
- Simplicidad de especificación de la semántica.

Un lenguaje L sobre un vocabulario V es un subconjunto de V^*, que esta determinado por una gramática. Una gramática impone una estructura sobre el vocabulario de tal manera que solo ciertos cadenas sean sentencias válidas.

Un cadena, 'string' o proposición o palabra de un alfabeto es una secuencia finita de símbolos de ese alfabeto.

Si A es un alfabeto, A^* denota el grupo de todas las palabras finitas que pueden ser formadas mediante este alfabeto (incluyendo la cadena vacia). cualquier subgrupo de A^* es llamado

un lenguaje sobre A.

CLASIFICACION DE CHOMSKY DE GRAMATICAS

Esta clasificación divide en cuatro clases a las gramáticas y está basada en la colocación sucesiva de restricciones cada vez mas fuertes en las producciones 'FI'-›'PSI'. así cada gramática del tiene un número del 0 al 3.

GRAMATICAS TIPO '0' O SIN RESTRICCIONES

Todas las producciones tienen el formato de A -› B donde A e (N U T)+ y B e (N U T)*. Ningún tipo de restricciones es colocado en las reglas de producción. Así por ejemplo,

ABC -› gdB

o bién

AB -› BA

o bién

gB -› g

Pueden ser reglas permitidas para gramáticas del tipo 0. Note que las reglas como gB -› g, comprenden símbolos para desaparacer lo cual se hace para algún otro lenguaje en particular. estos lenguajes son demasiado generales como para ser útiles y la cuestión de cuando una palabra dada es legal es generalmente imposible de decidir. Puede demostrarse que una gramática asi es reconocida por una máquina de Turing.

GRAMATICAS DE TIPO 1 O CONTEXTO SENSITIVO

Todas las producciones son de la forma A -› w donde A e N+ , B e (N U T)* y |A| ‹ |B|. Es decir todas las reglas son de longitud no decreciente. Las reglas de producción pueden ser de la forma:

ALFA BETA GAMMA -› ALFA BETA GAMMA

donde ALFA Y GAMMA son cadenas que pueden ser vacías y BETA es una palabra no vacía.

Dado que una gramática de tipo 1 puede contener dos reglas tales como:

ALFA A GAMMA -› ALFA BETA GAMMA
LAMBDA A ETHA -› LAMBDA DELTA ETHA

parece que el símbolo A puede ser rescrito como BETA o DELTA dependiendo del contexto.

Basandose en lo anterior, las gramáticas de este tipo son llamadas de contexto sensitivo o dependiente.

Aunque estas gramáticas ofrecen una dependencia del contexto con fuertes reminiscencias de las características de los lenguajes naturales y con alguna tendencia hacia los lenguajes de computadoras,. se desprende de ello que dichas gramáticas no son útiles dado que no son capaces de proveer información acerca de los lenguajes que generan (tal como la estructura de una proposición) sin una búsqueda exhaustiva de todas las proposiciones permisibles de la longitud requerida.

GRAMATICAS DEL TIPO '2' O CONTEXTO LIBRE

Todas las producciones de esta gramática tienen el formato de A -› w donde w e (N U T)* y A e N. Esta clase describe a la mayoría de lenguajes de programación. Aqui la palabra 'FI' debe consistir de un solo símbolo no terminal por ejemplo todas las reglas deben ser de la forma: A -› 'PSI' es una palabra no vacía. Estas gramáticas y lenguajes son llamados libres de contexto. También han sido llamados lenguajes de estructura de fase, lenguajes de estructura constitutiva, grupos definibles, lenguajes BNF, lenguajes estilo Pascal, lenguajes de Chomsky de tipo 2 , lenguajes autómatas push-down, etc.

GRAMATICA DEL TIPO 3 O REGULAR

Todas las producciones tienen el formato de A -› Bc o A -› c donde A,B e N y c e T. Las producciones deben tener la forma:

A -› a o bién
A -› bB

Considerando que existen varios tipos de gramáticas podemos preguntarnos cuales generan lenguajes de programación, concluyendo que ninguna de ellas lo hace. Sin embargo, las gramáticas del tipo 2 generan una clase de lenguaje que puede ser modificado de tal modo que sea exactamente del tipo requerido para los lenguajes de programación.

Sin embargo esas gramáticas y lenguajes han sido extensamente estudiados por los formalistas de lenguajes de computación. la razón primaria por la que los lenguajes de programación (incluso aquellos como Pascal) no son lenguajes del tipo 2 es que contienen proposiciones declarativas, y aquellas proposiciones que son dependientes de contexto no pueden ser representadas dentro de la gramática de contexto libre.

Una gran parte de los lenguajes de programación puede ser definida mediante gramáticas de contexto libre de modo que los métodos sintácticos de análisis pueden aplicarse a una gran porción del texto fuente con todas las conveniencias que esos métodos implican. A pesar de que los teoremas actuales derivados para lenguajes libres de contexto no se aplican a los lenguajes

de programación actuales, cuestiones de ambiguedad, unicidad, finitividad, etc. son sin embargo imposibles de contestar para estos lenguajes.

Ha habido algunos intentos para definir nuevos tipos de gramáticas que correspondan a los lenguajes de programación actuales, pero no han tenido el èxito esperado.

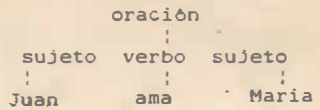
CARTAS SINTACTICAS O ARBOLES DE 'PARSING'

Las reglas de producción descritas por Chomsky se prestan por si mismas a una representacion diagramática, mediante árboles dirigidos.

Considere por ejemplo el lenguaje elemental representado por las siguientes producciones:

ORACION -› SUJETO VERBO SUJETO
VERBO -› ama
SUJETO -› Juan ¦ Maria

Una oración en este lenguaje puede ser formado como sigue.



Debido a la naturaleza simple de este lenguaje, cuatro diferentes oraciones pueden formarse. Sintácticamente estas cuatro oraciones son idénticas y el árbol sintáctico no cambia. Un lenguaje más variado podría permitir la construcción de un número de diferentes árboles sintácticos. Sin embargo si diferentes árboles sintácticos generan la misma oración. se dice que el lenguaje es ambiguo. Estas ambigüedades son siempre encontradas en los lenguajes naturales pero deben ser evitadas en un lenguaje de programación.

Un árbol de derivación es una representacion gráfica de la sintaxis de una sentencia determinada que está siendo construida o analizada.

Ejemplo. dibuje el árbol de derivación para el cadena 0101011 utilizando la siguiente gramática.

S -› 0S
S -› 1S
S -› 0
S -› 1

S -› 0S -› 01S -› 010S -› 0101S -› 01010S -› 010101S
-› 0101011

En la definición de Chomsky de ciertos tipos de gramáticas, es necesario especificar que no debe haber más de una regla con la misma parte derecha para dos no terminales diferentes, por ejemplo no debe haber dos reglas tales como:

A -› a
B -› a

Lenguajes generados por gramáticas que contienen reglas como estas son automáticamente ambiguos, dado que un símbolo sencillo como 'a' puede provenir de dos no terminales y de esa forma dar lugar a dos diferentes árboles sintácticos.

3. ANALISIS DE LEXICO

Para definir los analizadores de léxico se utilizan gramáticas y expresiones regulares. Las expresiones regulares son una representación compacta de estas gramáticas. Estas expresiones se utilizan para especificar la estructura de las unidades de léxico o 'tokens'.

Los autómatas deterministicos y no deterministicos, permiten construir analizadores de léxico o 'scanners'. Los generadores de analizadores de léxico como LEX permiten a partir de expresiones regulares construir 'scanners'.

EXPRESIONES REGULARES

Una expresión regular es una notación que describe de manera compacta lenguajes regulares. Las expresiones regulares se representan utilizando tres operadores: concatenación, alternación y cerradura.

Formalmente una expresión regular es aquella expresión que se construyen a partir de las siguientes reglas:

1. Ø es una expresión regular que denota un conjunto vacio

2. EPSILON es una expresión regular que denota un lenguaje que consiste de una cadena vacia

3. a es una expresión regular donde a E Terminales y que denota el lenguaje formado por un solo simbolo a

4. Si e1 y e2 son expresiones regulares que denotan a los lenguajes L1 y L2 respectivamente entonces:

   (e1) | (e2) es una expresión regular que denota L1 U L2

   (e1)(e2) es una expresión regular que denota L1L2

   e1* es una expresión regular que denota L1*

AUTOMATAS

Un autómata es un modelo idealizado de máquina que permite reconocer cadenas de caracteres que se pueden especificar mediante expresiones regulares. Una propiedad de los autómatas es que no utiliza memoria auxiliar para trabajar. El comportamiento del autómata se determina completamente por el estado presente y la entrada. Ejemplos de autómatas son el modelo cuántico del átomo y el ribosoma.

AUTOMATA FINITO NO DETERMINISTICO (NFA)

Un autómata no finito determinístico (NFA) es un conjunto de cinco elementos:

{S,A,M,I,F}

donde:

- S - un conjunto de estados S
- A - un conjunto de simbolos de entrada o alfabeto
- M - una función de transición M que mapea pares de estados-simbolos en estados
- I - un estado llamado de inicio
- F - un conjunto de estados finales o aceptores

Un NFA se representa mediante una gráfica dirigida, en donde los nodos representan estados, las flechas transiciones y los simbolos al lado de las flechas entradas.

CERRADURA EPSILON DE UN ESTADO S

La cerradura EPSILON de un estado NFA S es el conjunto de estados NFA a los que se puede llegar desde el estado S con solo transiciones EPSILON, este conjunto incluye a S.

SIGUIENTE ESTADO DE UN ESTADO S

El siguiente estado de un estado S sig_estado(S,a) es el conjunto de estados a los que se puede llegar desde un estado S al leer el caracter de entrada a.

CONSTRUCCION DE LA CERRADURA EPSILON DE UN ESTADO S

```
Sea
     S, S1, S2   estados NFA;
     cerraduraEPSILON   cerraduraEPSILON de un estado;

Begin
   S se agrega al conjunto cerraduraEPSILON;
   Repeat
     Si el estado S1 esta en cerraduraEPSILON y existe una
     transición EPSILON de S1 a S2, entonces S2 se agrega a
     cerraduraEPSILON(S) si no se entraba en el conjunto.
   Until(no existan mas estados)
End.
```

CERRADURA EPSILON DE UN CONJUNTO DE ESTADOS T

La cerradura EPSILON de un conjunto de estados NFA T es el conjunto de estados NFA a los que se puede llegar desde los estados T con solo transiciones EPSILON, este conjunto incluye a los elementos de T.

SIMULADOR

Al programa que realiza las funciones de un autómata determinístico se le llama simulador. Si este programa se realiza mediante tablas se le llama simulador universal pues sirve para cualquier autómata, al cambiar únicamente estas. Gran parte del trabajo de computación se puede realizar mediante autómatas combinados con otras estructuras.

SIMULADOR NFA

El algoritmo para simulación de un autómata NFA es:

```
Algoritmo NFA:
sea:
     X     cadena de entrada;
```



```
     a     caracter de entrada;
     S0    estado inicial automata;
     F     conjunto de estados finales o aceptores;
     sig_estado[]   matriz de siguiente estado del autómata;
     cerraduraEPSILON  función que calcula la cerradura EPSILON.
     lee_sig_caracter()   función que lee un caracter de la
                          entrada

Begin
     S := cerraduraEPSILON(S0);
     a := lee_sig_caracter();
     while a <> EOF do
          Begin
            S := cerraduraEPSILON(sig_estado[S,a]);
            c := lee_sig_caracter()
          End
     if S ^ F then
          return "token aceptado"
          else
          return "token invalido"
End.
```

AUTOMATA FINITO DETERMINISTICO (DFA)

Un autómata finito determinístico es un caso especial de un NFA donde no existen transiciones epsilon y por cada par de estados S y entrada a existe solamente una transición a otro estado.

SIMULADOR DFA

El algoritmo para simulación de un autómata DFA es:

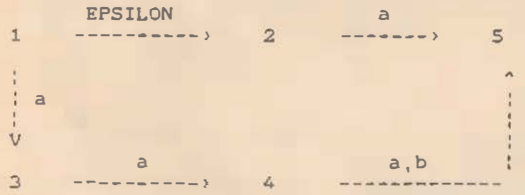
```
Algoritmo DFA:
sea:
     X     cadena de entrada;
     c     caracter de entrada;
     S0    estado inicial autómata;
     F     conjunto de estados finales o aceptores;
     sig_estado[]   matriz de siguiente estado del autómata;
     lee_sig_caracter()   función que lee un caracter de la
                          entrada


Begin
     S := S0;
     c := lee_sig_caracter(); /* caracter lookahead */
     while c <> EOF do
          Begin
            S := sig_estado[S,c];
            c := lee_sig_caracter()
          End
     if S IN F then
```

```
        return "token aceptado"
        else
        return "token invalido"
End.
```

ALGORITMO PARA CONVERSION DE UN AUTOMATA NFA A DFA POR MEDIO DE CONSTRUCCION DE SUBCONJUNTOS

```
Sea:
     N conjunto de estados NFA
     D conjunto de estados DFA
     a 'token' de entrada
     T,U estados DFA
     S0 estado incial NFA
     S0..Sn estados NFA
     cerraduraE()  función de cálculo de cerradura EPSILON
     DTRAN[T,a]  función de transición de automata DFA


Begin
     Se agrega cerraduraE(S0) como un estado DFA sin marcar;
     While (exista un estado T sin marcar) Do
       Begin
          Marca T;
          for (cada 'token' de entrada a en T) do
              Begin
                U := cerraduraE(sig_estado(T,a));
                If U := Not in D then
                    Begin
                        agrega U a D como estado sin marcar;
                        DTRAN[T,a] := U;
                    end;
                If (S in T es estado final) then
                        T es estado final;
              End;
     End
End.
```

ejemplo. Dado el siguiente autómata NFA conviertalo a DFA.



estado inicial = cerraduraEPSILON(1) = {1,2} = A

donde A es un nuevo estado DFA de inicio que se marca

cerraduraEPSILON(sig_estado(A,a)) = cerraduraEPSILON(3,4,5) = {3,4,5} = B

Puesto que los estados {3,4,5} no existen como estado DFA se crea un nuevo estado DFA llamado B que se marca

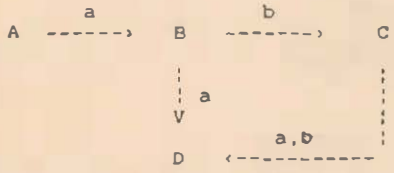
cerraduraEPSILON(sig_estado(B,a)) = cerraduraEPSILON(5) = {5} = C

cerraduraEPSILON(sig_estado(B,b)) = cerraduraEPSILON(4,5) = {4,5} = D

Se marca C, pero no existen caracteres para leer.

cerraduraEPSILON(sig_estado(D,a)) = cerraduraEPSILON(5) = C

El autómata equivalente DFA es:



Esto se puede verificar si vemos las posibles combinaciones de caracteres leidos hasta llegar al estado final en el autómata NFA que son:

aba, abb, aa, ab, a

y en el autómata DFA son:

ab, aa, a, aba, abb

RECONOCEDOR O 'SCANNER

Un scanner es un autómata que puede reconocer cadenas de caracteres que pertenecen a un lenguaje.

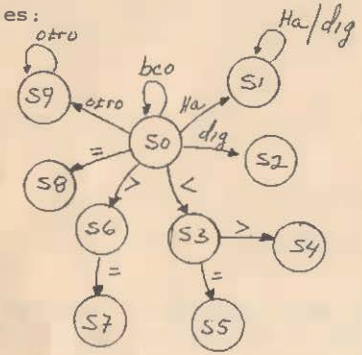
El scanner esta formado por el autómata y un filtro.

El filtro traduce letras, números y caracteres especiales a un código numérico que utiliza el autómata.

Adicionalmente a reconocer cadenas de caracteres proporciona la longitud del texto, el tipo de cadena y la cadena en un registro.

ejemplo: Construya un automata DFA y escriba el programa que lo simule y reconozca: identificadores, enteros y simbolos de comparación.

El autómata es:



El programa que simula al autómata esta escrito en lenguaje 'C.y es:

```
/* Facultad de Ingeniería                                   */
/* Apuntes de Compiladores                                  */
/* prof: A. Jiménez H.                                      */
/* ejemplo de 'scanner'                                     */
/*                                                          */

#include "stdio.h"
int ch;
main()
      {
         int token;
         printf("\tANALIZADOR DE LEXICO\n");
         printf(" para finalizar teclee CTL-C \n\n");
```

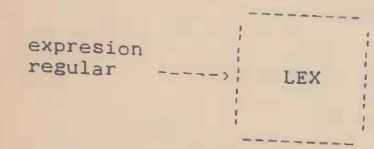


```
         printf("deme token = ");
         while ((ch=getchar())!= EOF )    {
            ungetc(ch,stdin);
            token=get_token();
            switch(token)    {
               case 1: printf("\t identificador \n");break;
               case 2: printf("\t entero \n");break;
               case 3: printf("\t menor \n");break;
               case 4: printf("\t diferente \n"); break;
               case 5: printf("\t menor o igual \n");break;
               case 6: printf("\t mayor \n");break;
               case 7: printf("\t mayor o igual \n");break;
               case 8: printf("\t igual \n");break;
               default: printf("\t error token invalido\n");
            }
         ungetc(ch,stdin);
         printf("\n*tro = ");
         }
         printf("terminamos....\n");
      }


#define EDO 10          /*  num de estados de autómata  */
#define COD 7           /*  num de códigos de entrada   */

get_token()

{
   unsigned cod;
   unsigned sal=0, edo=0;

   /* tabla siguiente estado  */
   static unsigned m_edo[EDO][COD]= {
 /*     let,dig,<,>,=,bco,otr  */
         1,2,3,6,8,0,9, /* estado inicial */
         1,1,0,0,0,0,0, /* S1  */
         0,2,0,0,0,0,0, /* S2  */
         0,0,0,4,5,0,0, /* S3  */
         0,0,0,0,0,0,0, /* S4  */
         0,0,0,0,0,0,0, /* S5  */
         0,0,0,0,7,0,0, /* S6  */
         0,0,0,0,0,0,0, /* S7  */
         0,0,0,0,0,0,0, /* S8  */
         0,0,0,0,0,0,9, /* S9  */  };

   /* tabla salidas   */
   static unsigned m_sal[EDO][COD]=
      {
    /*  let,dig,<,>,=,bco,otr  */
         0,0,0,0,0,0,0, /* estado inicial */
         0,0,1,1,1,1,1, /* id  */
         2,0,2,2,2,2,2, /* entero */
         3,3,3,0,0,3,3, /* <  */
         4,4,4,4,4,4,4, /* <>  */
         5,5,5,5,5,5,5, /* <=  */
```

```
            6,6,6,6,0,6,6, /* > */
            7,7,7,7,7,7,7, /* >= */
            8,8,8,8,8,8,8, /* = */
            9,9,9,9,9,9,0, /* error */  };

    while (sal==0)   {
       cod = filtro();
       sal = m_sal[edo][cod];
       edo = m_edo[edo][cod];
    }
    return(sal);
}

#include "ctype.h"

filtro()      /* lee y ordena símbolos de entrada */
{  int cod;

   if (isalpha(ch=getchar()))
      return(0);
   else if (isdigit(ch))
      return(1);
   else if (ch == '<')
      return(2);
   else if (ch == '>')
      return(3);
   else if (ch == '=')
      return(4);
   else if (ch == ' ' ||ch == '\t' || ch == '\n' )
      return(5);
   else return(6);
}
```

GENERADORES DE ANALIZADORES DE LEXICO
LEX

LEX es un generador de analizadores de léxico o 'scanners' y puede ser utilizado por analizadores de sintaxis o 'parsers' generados por YACC. LEX traduce expresiones regulares a un autómata DFA escrito en lenguaje 'C', que reconoce un lenguaje. Lex transforma la expresión regular en un autómata NFA, después en uno DFA, compactando finalmente las tablas correspondientes.



Estos tokens son organizados de acuerdo a una gramática

regular; cuando un 'token' ha sido reconocido, el código de usuario realiza una acción normalmente en la tabla de símbolos almacenando valores para hacer uso de ellos en otras acciones.

La ventaja de utilizar LEX es el ahorro de trabajo al programador al realizar el analizador de léxico o 'scanner' automáticamente, obteniendo un programa compacto y rápido. LEX se encuentra en los sistemas operativos UNIX.

Un programa o archivo en LEX está escrito en un lenguaje de propósito especial y consiste de tres secciones separadas por el símbolo %%

```
definiciones
%%
reglas de reconocimiento
%%
subprogramas de usuario
```

La primera parte de definiciones es opcional y puede contener:

- dimensiones de tablas internas de LEX
- código en lenguaje 'C' que es global, escrito dentro de los símbolos %{ y %}
- definiciones para reemplazo de textos, que consiste en declaraciones de la forma: nombre = expresión;

El nombre es un identificador y se utiliza como una expresión regular.

La parte de %% siempre se incluye.

La segunda parte de reglas de reconocimiento consiste en una tabla de patrones o expresiones regulares y acciones. A cada patrón corresponde una acción que puede ser una o varias instrucciones en lenguaje 'C' o una barra que indica que ese patrón utiliza la misma acción que el siguiente.

El formato es:

expresión { acción }

La tercera parte de subprogramas de usuario es opcional y contiene código en lenguaje 'C', que normalmente son funciones que se utilizan en la parte de acción de la segunda parte.

Adicionalmente al autómata DFA, LEX proporciona varias rutinas para copiar el 'token' reconocido, obtener su longitud, manejar los errores, incializar parámetros del autómata, etc.

Cuando el autómata se genera como una función a utilizarse por otro programa este toma el nombre de 'yylex()'.

PATRONES

Los patrones se utilizan para formar secuencias de caracteres o expresiones regulares y su significado se puede resumir como sigue:

- letras y dígitos se representan a sí mismos
- el punto '.' representa cualquier caracter menos LF
- paréntesis de la forma [] cuando encierran secuencias de caracteres definen conjuntos de caracteres que representan a cualquiera de sus componentes o si la secuencia empieza con el simbolo '^' a cualquier caracter que no este en la secuencia.
- - entre dos caracteres indica un rango o gama.
- * cuando sigue a un patrón representa cero o más concatenaciones de ese patrón.
- + cuando sigue a un patrón representa una o más concatenaciones de ese patrón.
- ? cuando sigue a un patrón representa cero o una concatenación de ese patrón.
- ^ al inicio de un patrón representa inicio de línea
- $ al fin de un patrón representa fin de linea
- | representa alternativas de reglas de producción
- los paréntesis ( ) se utilizan para manejar la prioridad de las expresiones.

La concatenación se expresa escribiendo las expresiones o caracteres adyacentes. ejemplo:

```
blanco = [ \t\n];
identificador = [A-Za-z] [A-Za-z0-9];
comentarioC = "/*".*"*/";
comentario = "{"[^}]*"}";
opmul = "*";
```

Ejemplo. Escriba un programa en LEX para que reconozca enteros, reales, identificadores y simbolos de comparación.

El programa en LEX es:

```
%{
  /* scanner en LEX para reconocer:
     identificadores, enteros, reales y simbolos
     de comparación
     */

%}

/* primera parte de un programa en LEX
   */
   /*
   definiciones de reemplazo de textos
   */

letra      =   [A-Za-z];
letrae     =   [E|e]
digito     =   [0-9];
id         =   letra(letra|digito)*;
entero     =   digito+;
real       =   entero\.letrae(digito)*;

%{

/* función principal que reemplaza al parser      */
/* el scanner generado por LEX se llama yylex()   */

main()
{
        register int    i;
        char            *buffer[80];
        extern char     *token();

        while (i = yylex()) {
             gettoken(buffer, sizeof buffer);
             printf("yylex tiene token %d, token = \"%s\"\n",
                        i, buffer);
             if (i == LEXERR) {
                   error("LEXERR -- aborta programa");
                   break;
                }
        }
}
%}

%%

id              {printf("yylex: identificador\n");
                 lexval = instala();
                 return(1);}
entero          {printf("yylex: entero\n");
                 lexval = instala();
                 return(2);}
real            {printf("yylex: real\n");
                 lexval = instala();
                 return(3);}
"<"             {printf("yylex: LT\n"); return(4);}
"<="            {printf("yylex: LE\n"); return(5);}
"="             {printf("yylex: EQ\n"); return(6);}
"<>"            {printf("yylex: NE\n"); return(7);}
">"             {printf("yylex: GT\n"); return(8);}
">="            {printf("yylex: GE\n"); return(9);}

%%

char *
instala()
/*
   función de usuario para grabar el token
   encontrado en la tabla de simbolos
 */
```

```
{
        register char   *buffer;    /* buffer almacena token */
        register char   *inicio;    /* -> byte inicio token */
        char            *fin;
        extern char     *token();

        inicio = token(&fin);    /* inicio/fin token */
        if ((buffer = alloc((fin - inicio) + 1)) == NULL)
                {
                error("no hay espacio en tabla simbolos");
                exit(1);
                }
        inicio = copy(buffer, inicio, (fin - inicio));
        *inicio = '\0';
        return(buffer);
}
```

Ejemplo. Escriba un programa en LEX para que reconozca los 'tokens' de un compilador de 'C'.

El programa en LEX es:

```
/*
 Apuntes de Compiladores

 reconocimiento de 'tokens' para
 el lenguaje C

*/

%{
extern  char    *instala();
%}

digito  = [0-9];
letra   = [a-zA-Z_$];
nombre  = letra (letra|digito)*;
entero  = digito digito*;

%{
main()
{
        register int    i;
        char            buffer[80];
        extern char     *token();

        while (i = yylex()) {
                gettoken(buffer, sizeof buffer);
                printf("yylex = %d, token = \"%s\"\n",
                        i, buffer);
                if (i == LEXERR) {
                        error("LEXERR -- abort");
                        break;
```



```
                }
        }
}
%}

%%

        /* patrones de reconocimiento  */
%{
        register int    c;
%}

#DEFINE         { return(1); }
#ELSE           { return(2); }
#ENDIF          { return(3); }
#IFDEF          { return(4); }
#INCLUDE        { return(5); }
STRUCT          { return(-1); }
AUTO            { return(6); }
EXTERN          { return(7); }
STATIC          { return(8); }
REGISTER        { return(9); }
GOTO            { return(10); }
RETURN          { return(11); }
IF              { return(12); }
WHILE           { return(13); }
ELSE            { return(14); }
SWITCH          { return(15); }
CASE            { return(16); }
BREAK           { return(17); }
CONTINUE        { return(18); }
DO              { return(19); }
DEFAULT         { return(20); }
FOR             { return(21); }
SIZEOF          { return(22); }
TYPEDEF         { return(23); }
UNION           { return(24); }
nombre          {
                        lexval = instala();
                        return(25);
                }
entero          {
                        lexval = instala();
                        return(26);
                }
"<"             { return(27); }
"<="            { return(28); }
"="             { return(29); }
"!="            { return(30); }
">="            { return(31); }
">"             { return(32); }
"<<"            { return(33); }
">>"            { return(34); }
"=+"            { return(35); }
```

```
"=-"            { return(36); }
"=/"            { return(37); }
"=%"            { return(38); }
"%"             { return(39); }
"/"             { return(40); }
"*"             { return(41); }
"=*"            { return(42); }
"=<<"           { return(43); }
"=>>"           { return(44); }
"&"             { return(45); }
"|"             { return(46); }
"=|"            { return(47); }
"=&"            { return(48); }
"+"             { return(49); }
"-"             { return(50); }
"++"            { return(51); }
"--"            { return(52); }
";"             { return(53); }
"?"             { return(54); }
"->"            { return(55); }
"."             { return(56); }
"/*"            {
                        comment("*/");
                        return(LEXSKIP);
                }
"'"             {
                    if ((c = mapch('\'', '\\')) != -1)
                        while (mapch('\'', '\\') != -1)
                        lexerror("cadena demasiado larga");
                     printf("%c", c);
                     return(57);
                }
"\""            { return(58); }
"\n"            { return(59); }
" "             { return(60); }
"\t"            { return(61); }
"||"            { return(62); }
"&&"            { return(63); }
"("             { return(64); }
")"             { return(65); }
"["             { return(66); }
"]"             { return(67); }
"{"             { return(68); }
"}"             { return(69); }

%%

char *
instala()
/*
   función de usuario para grabar el token
   encontrado en la tabla de símbolos
 */

{
```



```
register char   *buffer;    /* buffer almacena token */
register char   *inicio;    /* -> byte inicio token */
char            *fin;
extern char     *token();

inicio = token(&fin);   /* inicio/fin token */
if ((buffer = alloc((fin - inicio) + 1)) == NULL)
        {
        error("no hay espacio en tabla simbolos");
        exit(1);
        }
inicio = copy(buffer, inicio, (fin - inicio));
*inicio = '\0';
return(buffer);
```

4. ANALISIS DE SINTAXIS

Para definir los analizadores de sintaxis se utilizan gramáticas de contexto libre, estas gramáticas deben de no tener ciclos, símbolos inútiles y en algunos casos no ser recursivas por la izquierda.

Los analizadores de sintaxis se clasifican en dos por el método que utilizan para reconocer cadenas de terminales siendo: arriba-abajo y abajo-arriba.

NOTACION

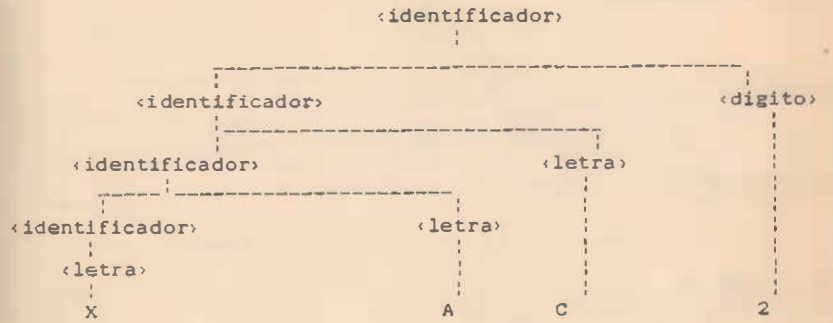
Para representar cadenas de símbolos se utiliza la siguiente notación:

- para símbolos terminales se utilizan las primeras letras minúsculas del alfabeto como a,b,c,d, etc.
- para símbolos no terminales las primeras letras mayúsculas del alfabeto a excepción de S para el símbolo de inicio, como A,B,C,D, etc.
- para los símbolos gramaticales que son símbolos terminales y no-terminales se asignan las últimas letras mayúsculas del alfabeto, U,V,W,X,Y,Z, etc.
- para las cadenas de símbolos terminales las últimas letras minúsculas del alfabeto, como u,v,w,x,y,z, etc.
- para las cadenas de símbolos gramaticales las letras griegas, como ALFA, BETA, etc.
- para indicar una derivación el símbolo $\Rightarrow$
- para indicar cero o varias derivaciones $\overset{*}{\Rightarrow}$
- para indicar una o varias derivaciones $\overset{+}{\Rightarrow}$

CARTAS SINTACTICAS O ARBOLES DE 'PARSING'

Las cartas sintácticas son una representación gráfica del proceso de derivación de sentencias de un lenguaje y permiten visualizar claramente el lenguaje.

Por ejemplo la derivación del identificador XAC2 es:



FORMA SENTENCIAL

Una forma sentencial es cualquier derivación de un único noterminal de inicio S que contiene símbolos no terminales. Así si:

$$S \Rightarrow ALFA$$

entonces ALFA es una forma sentencial de G.

SENTENCIA

Una sentencia es una forma sentencial con solo terminales.

FRASE

Para una forma sentencial LAMBDA = ALFA1 BETA ALFA2 de una gramática G, se dice que BETA es una frase de LAMBDA para un no terminal A si:

$$S \overset{*}{\Rightarrow} ALFA1\ \ BETA\ \ ALFA\ 2$$

y $$A \overset{+}{\Rightarrow} BETA$$

FRASE SIMPLE

BETA es una frase simple si $S \overset{*}{\Rightarrow} ALFA1\ A\ ALFA2$ y $A \Rightarrow BETA$
Donde esta frase es un subcadena de la forma sentencial.

GRAMATICA AMBIGUA

Una gramática es ambigua si permite derivaciones diferentes de una misma cadena de simbolos terminales, en otras palabras una gramática es ambigua si existe más de un árbol de sintaxis para un grupo de tokens. Para verificarlo simplemente se encuentra ese grupo de tokens. ejem. para la siguiente gramática el cadena 01 tiene dos derivaciones.

```
S -> TS | 0 | 1
T -> 0 | 1
```

o la gramática

```
S -> SS | 0 | 1
```

para el cadena 011
o la gramática

```
S -> S + S | S - S | 0 | 2 | 3 | 4 | ..| 9
```

para 5 + 3 - 2

SIMBOLOS INUTILES

Son los simbolos no terminales que son inalcansables o que no derivan simbolos terminales

Una gramática 'no reducida' es aquella que contiene no terminales inútiles. una vez quitados estos simbolos la gramática es 'reducida'.

RECURSION POR LA IZQUIERDA

Una gramática es recursiva por la izquierda si para un no terminal A existe una producción de la forma A +-> A ALFA para una cadena ALFA.

GRAMATICA LIBRE DE CICLOS

Una gramática esta libre de ciclos si no tiene derivaciones de la forma A -> A para cualquier no terminal A.

GRAMATICA LIBRE DE PRODUCCIONES EPSILON

Una gramática esta libre de produciones EPSILON si no tiene producciones que incluyan EPSILON.

ANALIZADOR SINTACTICO O 'PARSER'

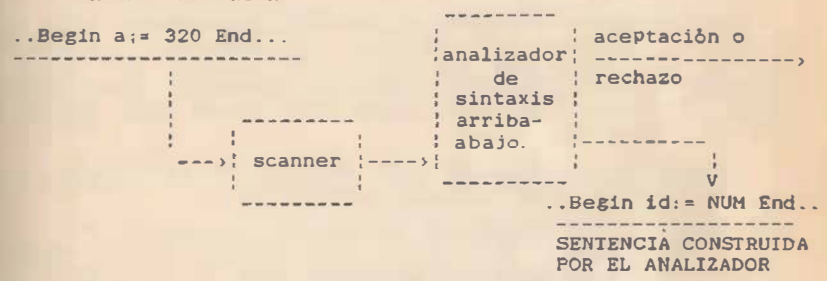
El analizador sintáctico o 'parser' verifica que una cadena de entrada pertenezca a un lenguaje.

Los analizadores de sintaxis se pueden clasificar en general en:

Arriba-abajo
Abajo-arriba

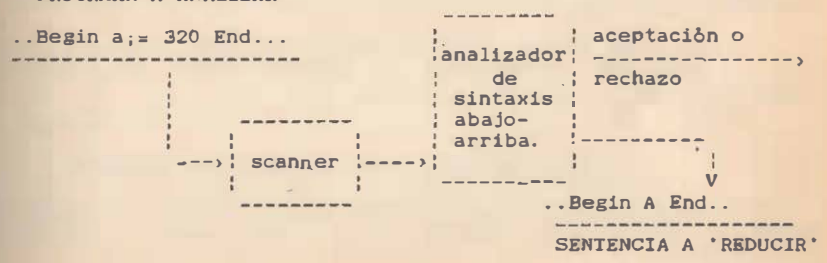
La anterior clasificación se relaciona con dos estrategias que se utilizan para reconocer sentencias o cadenas de entrada. La primera es tratar de construir a partir de la gramática a validar una sentencia que sea igual a la entrada que se esta reconociendo, si se logra construir esta, la sentencia de entrada es válida. Este procedimiento es utilizado por los 'parsers' arriba-abajo.

PROGRAMA A ANALIZAR



En el segundo caso se lee la sentencia de entrada y se trata comparar e identificar con cadenas iguales en el lado derecho de las producciones de la gramática, si se logra esta identificación se reemplaza el lado derecho con el lado izquierdo y se vuelve a leer y comparar hasta que se transforma toda la sentencia o cadena de entrada en un solo simbolo que es el de inicio. Este procedimiento lo utiliza el 'parser' abajo-arriba.

PROGRAMA A ANALIZAR



5. ANALIZADORES ARRIBA-ABAJO

El analizador sintáctico se caracteriza porque a partir del simbolo de inicio intenta construir una sentencia o cadena de simbolos terminales que sea igual a la entrada que se está analizando. Este proceso se realiza al sustituir sucesivamente producciones hasta que se genera una cadena de terminales idéntica a la entrada.

Se analiza el concepto de función de predicción, las funciones FIRST y FOLLOW y las gramáticas LL.

Existen tres métodos Arriba-abajo de importancia:

Analizador con retroceso
Analizador recursivo descendente
Analizador determinístico

ANALIZADOR SINTACTICO CON RETROCESO

Este método a partir del simbolo de inicio genera una cadena de simbolos terminales al aplicar sucesivamente las producciones de la gramática hasta que se genera la cadena de terminales deseada.

En el peor de los casos en este método cuando una cadena no se encuentra en el lenguaje, se generan todas las posibles combinaciones.

ANALIZADOR SINTACTICO RECURSIVO DESCENDENTE

En este método de análisis para cada producción o simbolo No Terminal se escribe una función o procedimiento . Asi cada

función retorna un valor de verdadero o falso, dependiendo si reconoce o no un cadena de entrada que es una expansión de ese no-terminal. La utilización de llamadas a funciones recursivas reemplaza la utilización explicita de un stack.

En este método la gramática que se utiliza no debe de tener recursión por la izquierda y las producciones se deben ordenar de tal manera que se verifique la alternativa más larga primero.

ANALIZADOR NO RECURSIVO DESCENDENTE DETERMINISTICO

Este analizador se basa en la función de predicción, que nos dice para un determinado simbolo terminal la producción que se debe de expander.

CALCULO DE FUNCION DE PREDICCION

La función de predicción se calcula a partir de las funciones FIRST y FOLLOW.

Si no existen producciones anulables en la gramática, el cálculo de la función de predicción se realiza solo con la función FIRST.

Si existen producciones EPSILON el cálculo de la función de predicción se realiza mediante las funciones FIRST y FOLLOW.

El concepto de derivaciones EPSILON de un no terminal, producción y cadena es importante en el cálculo de la función de predicción. Un cadena se anula o desaparece si se deriva de el EPSILON . Una producción se anula si su lado derecho deriva en EPSILON , y un simbolo no terminal se anula si es el lado izquierdo de una producción anulable.

FUNCION FIRST

Para una producción de la forma A -› ALFA o una cadena de simbolos gramaticales FIRST(ALFA) es el conjunto de simbolos terminales iniciales que se derivan de ALFA incluyendo a EPSILON.

CALCULO DE LA FUNCION FIRST

Para un símbolo gramatical X, FIRST(X) se cálcula de la siguiente manera:

- si X es terminal entonces FIRST(X) = { X }
- si X->EPSILON entonces FIRST(X) = { EPSILON }
- si X es un simbolo no terminal y existe una producción de la forma X->Y1Y2...Yk, entonces se agrega el símbolo terminal o 'token' 'a' si existe en la producción

FIRST(Yi). Si existe EPSILON en la producción este se agrega a FIRST(Yi).

FUNCION FOLLOW

Para un simbolo no terminal A la función FOLLOW(A), es el conjunto de terminales 'a' que aparecen inmediatamente a la derecha de A en alguna forma sentencial en el lado derecho de una producción.

CALCULO DE FOLLOW

Para un no terminal A, FOLLOW(A) se calcula:

- para FOLLOW(S) se incluye en el conjunto el simbolo de terminación de cadena de entrada { $ } donde S es el simbolo de inicio.
- si existe una producción de la forma A--› ALFA B BETA , el contenido de FIRST(BETA) excepto EPSILON se agrega a FOLLOW(B).
- si existe una producción A-› ALFA B o una producción A -› ALFA B BETA donde FIRST(BETA) contiene a EPSILON o lo deriva en la forma: $BETA \Rightarrow^{*} EPSILON$ el contenido de FOLLOW(A) se agrega a FOLLOW(B).

Ejemplo: Dada la siguiente gramática construya las funciones FIRST y FOLLOW.

```
E-›    T   E'
E'-›   O   T   E'  ;  EPSILON
T-›    F   T'
T'-›   M   F   T'  ;  EPSILON
F-›    P     ;  -  P
O-›    +     ;  -
M-›    *     ;  /
P-›  ( E )   ;  ENTERO
```

La función FIRST es:

```
FIRST(E) = FIRST(T) = FIRST(F) = FIRST(P) ={ -, (, ENTERO }
FIRST(E') = { +, -, EPSILON }
FIRST(O) = { +, - }
FIRST(T') = { *, /, EPSILON }
FIRST(M) = { *, / }
```

La función FOLLOW es:

```
FOLLOW(E') = FOLLOW(E) = { ), $ }
```



```
FOLLOW(T') = FOLLOW(T) = { FIRST(E') } = { +, - }
```

FUNCION DE PREDICCION

El conjunto de elementos que predicen la producción que se aplicará o expanderá están formados por los simbolos terminales encontrados en la función FIRST(), si la producción no contiene derivaciones EPSILON y de FIRST() FOLLOW() si esta los contiene.

```
PREDICCION(A-› X1....Xm) = IF EPSILON E FIRST(X1...Xm) THEN
                           (FIRST(X1...Xm) - EPSILON) U FOLLOW(A)
                           ELSE FIRST(X1...Xm) END
```

GRAMATICA LL(1)

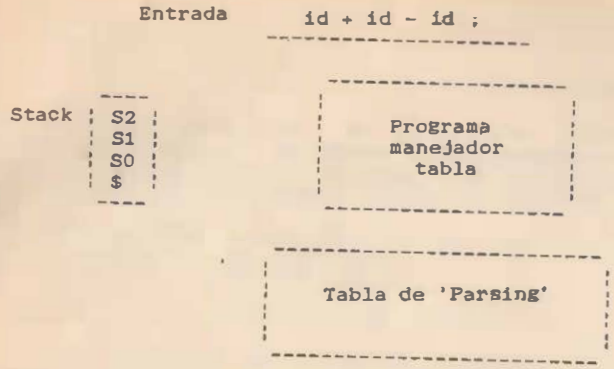
Una gramatica es LL(1) si los conjuntos de terminales formados a partir de la función de predicción para producciones alternativas de un no terminal son mutuamente exclusivos.

En otras palabras una gramática no es LL(1) si para un token t y dos producciones A-› X1...Xm y A-› Y1...Ym tenemos que:

```
t  E  PREDICCION(A-› X1...Xm) y t E PREDICCION(A-› Y1...Ym)
```

PARSER LL(1)

El 'parser' o reconocedor LL(1) consiste de una entrada, una pila o 'stack', una tabla de 'parsing' y una salida. El parser LL(1) revisa la cadena de entrada rechazándola cuando el prefijo que esta formando es no viable.




```
ALGORITMO RECONOCEDOR LL(1) O MANEJADOR DE AUTOMATA

BEGIN
     push(S);  { se carga el símbolo de inicio en el stack }
     WHILE NOT stack_vacio DO
     { sea X el elemento en el tope del stack y 'a' el símbolo
       de entrada }
     BEGIN
     IF   X es NO_TERMINAL THEN
          IF M(X,a) = X -> Y1...Ym THEN
                    BEGIN
                         pop(X); { realiza expansión }
                         push(Y1...Ym);
                         { Y1 en el tope del stack }
                    END
          ELSE error()
     ELSE { X es terminal }
          IF X = a THEN
               BEGIN
                    pop(X); { se reconoce parte de la entrada }
                    scanner(a);
               END
          ELSE error();
     END WHILE
END
```

Ejemplo: Dada una tabla de parsing LL(1), realize el proceso de aceptación de la sentencia:

(355 + 222 * 34) + 222 $

Tabla de 'parsing'

	num	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			EPSILON	EPSILON
T	FT'			FT'		
T'		EPSILON	*FT'		EPSILON	EPSILON
F	num			(E)		

La secuencia para aceptar la sentencia es:

'stack'	entrada	
E	(355 + 222 * 34) + 222 $	
TE'	(355 + 222 * 34) + 222 $	
FT'E'	(355 + 222 * 34) + 222 $	
(E)T'E'	(355 + 222 * 34) + 222 $	reconoce
E)T'E'	355 + 222 * 34) + 222 $	
TE')T'E'	355 + 222 * 34) + 222 $	
FT'E')T'E'	355 + 222 * 34) + 222 $	
numT'E')T'E'	355 + 222 * 34) + 222 $	reconoce
T'E')T'E'	+ 222 * 34) + 222 $	
E')T'E'	+ 222 * 34) + 222 $	
+TE')T'E'	+ 222 * 34) + 222 $	reconoce
TE')T'E'	222 * 34) + 222 $	
FT'E')T'E'	222 * 34) + 222 $	
numT'E')T'E'	222 * 34) + 222 $	reconoce
T'E')T'E'	* 34) + 222 $	
*FT'E')T'E'	* 34) + 222 $	reconoce
FT'E')T'E'	34) + 222 $	
numT'E')T'E'	34) + 222 $	reconoce
T'E')T'E'	) + 222 $	
E')T'E'	) + 222 $	
)T'E'	) + 222 $	reconoce
T'E'	+ 222 $	
E'	+ 222 $	
+TE'	+ 222 $	reconoce
TE'	222 $	
FT'E'	222 $	
numT'E'	222 $	reconoce
T'E'	$	
E'	$	
	$	aceptación

TABLA DE PARSING

La tabla de parsing M se forma a partir de la función de predicción siendo de la forma

$M(A,a) = A \rightarrow X_1 X_2 \ldots X_m$ si $A \in PREDICCION(A \rightarrow X_1 \ldots X_m)$;

M(A,a) = a -> error

Ejemplo: Dada una gramática LL(1) construya la tabla de parsing

La gramática es:

```
E -> TE'
E' -> +TE' | EPSILON
T -> FT'
T' -> *FT' | EPSILON
F -> (E) | id
```

Donde el 'token' NUMERO es:

NUMERO = (digito)+

La función de predicción es:

```
PREDICCION(E) = FIRST(E) = { (, id }
PREDICCION(E') = FIRST(E') U FOLLOW(E') = { + } U FOLLOW(E)
               = { + } U { $ , ) } = { +, $, ) }
PREDICCION(T) = FIRST(T) = { (, id }
PREDICCION(T') = FIRST(T') U FOLLOW(T') = { * } U FOLLOW(T)
               = { * } U FIRST(E') U FOLLOW(E)
               = { * } U { + } U { $, ) }
               = { *, +, $, ) }
```

La tabla de 'parsing' es:

	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			EPSILON	EPSILON
T	FT'			FT'		
T'		EPSILON	*FT'		EPSILON	EPSILON
F	id			(E)		

VENTAJAS DE PARSERS LL(1)

- no realizan retroceso o 'backtracking'
- si la gramática es LL(1) existe una derivación o parsing de izquierda a derecha correcta.
- todos los 'parsers' LL(1) operan en tiempos y espacio con respuesta lineal.
- todas las gramáticas LL(1) son no ambiguas

6. ANALIZADORES ABAJO-ARRIBA

Este capitulo trata los analizadores abajo-arriba, se estudian los 'parsers' LR(0), SLR(1), LR(1) y LALR(1). Se utilizan como herramientas para la construcción de 'parsers' las funciones FIRST() y FOLLOW(), la producción marcada o 'configuración' y la cerradura sobre 'configuraciones'.

Se presentan herramientas automáticas para generar 'parsers' como YACC.

ANTECEDENTES

Así como el trabajo de un 'parser' arriba-abajo es el tratar de encontrar una producción correcta para expander un no-terminal y construir una sentencia igual a la de entrada.

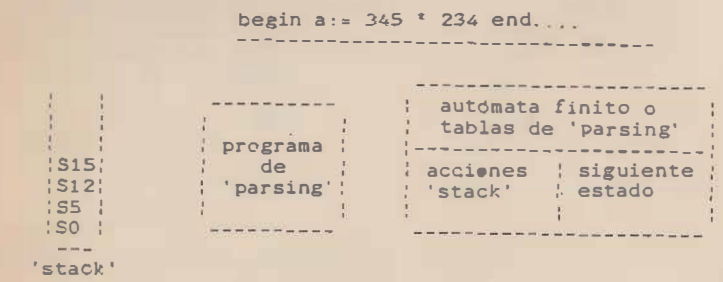
Un 'parser' abajo-arriba debe leer la sentencia de entrada, almacenarla y tratarla de identificar con lados derechos de producciones, si existe igualdad los reeemplaza por lados izquierdos, hasta que mediante este procedimiento se logra llegar al simbolo de inicio.

En otras palabras se construye un arbol sintáctico o de 'parsing' para un cadena de entrada empezando desde los nodos terminales y finalizando en el simbolo de inicio.

PARSING SHIFT-REDUCCION

Este proceso consiste en reducir y reemplazar subcadenas del cadena de entrada por los símbolos no terminales del lado izquierdo de la producción correspondiente, hasta que se llega al simbolo de inicio con lo que se reproduce una derivación más a la derecha al revés o en reversa.

Estos analizadores o parsers consisten de una máquina de estado finito o autómata representado por una tabla de parsing, con un 'stack' o pila, un buffer de entrada, una salida y un programa que los maneja.



ALGORITMO GENERAL DEL 'PARSER' SHIFT-REDUCCION O LR(K)

Las letras LR son la abreviación de como opera el parser: analizando la cadena de entrada de izquierda (Left) a derecha y generando la derivación mas a la derecha (Right) de manera inversa.

Este algoritmo lee símbolos terminales o 'tokens' de 'lookahead' de la entrada y los almacena en el stack, formando un cadena de la forma:

```
S S S .........S
1 2 3          m
```

Donde S representa un estado asociado a un símbolo gramatical, el par estado S y entrada sirve para accesar la tabla de parsing formada por dos partes una que decide que acción realizar en el 'stack 'y otra que define el estado siguiente. Las acciones en el 'stack' pueden ser las siguientes:

- 'shift' en el stack del símbolo terminal de entrada y estado siguiente.
- 'reducción' de la expresión o forma sentencial en el stack, aquí se hace un 'pop' de los símbolos que corresponden al lado derecho de la producción y un push del lado izquierdo de ésta.
- aceptación de la cadena de entrada como válida
- error en el cadena de entrada.

Los elementos del 'parser' son: siguiente token de entrada (lookahead); el estado actual es el que se encuentra en el tope del 'stack'. Los estados del autómata de estado finito están en la tabla de 'parsing'.

En el inicio de operación, la máquina esta en estado 0, la pila contiene solamente el estado 0 y no se ha leido el token "lookahead".

La máquina realiza las 4 funciones de 'shift', reducción, aceptación y error. La acción de 'shift' toma un 'token' de la entrada y lo introduce en el 'stack'.

La acción de 'reducción' se efectua cuando el parser ha visto el lado derecho de una regla de la gramática, anunciando que ha reconocido una regla de producción y reemplazando el lado derecho por el lado izquierdo. Puede ser necesario consultar el token de "lookahead" para decidir que lado izquierdo hay que reducir o sustituir, pero normalmente no lo es. Estas acciones están asociadas con reglas gramaticales individuales. La acción de reducción saca del 'stack' los estados asociados a los símbolos del lado derecho e introduce el estado del lado izquierdo.

La acción de aceptación indica que el programa o cadena de entrada ha sido reconocida como correcta y está de acuerdo a la especificación. Esta acción sólo aparece hasta leer la marca de fin de archivo e indica que el 'parser' ha terminado.

La acción error, representa un lugar donde el parser no puede continuar su trabajo de acuerdo con la especificación. Los 'tokens' de entrada han sido examinados junto con el 'token' de lookahead y no se puede continuar en esta condición. El 'parser' reporta un error e intenta recuperarse de esta situación y continuar.

GRAMATICA LR(k)

Una gramática es LR(k) si para un cadena de entrada ØBt en cada paso de la derivación más a la derecha realizado de manera inversa el 'handle' B puede identificarse al examinar la cadena ØB y cuando más los primeros 'k' símbolos de la cadena de entrada.

Dicho de otra manera:

$$S \xrightarrow[rm]{} aAw \xrightarrow[rm]{} aBw \quad y$$

$$S \xrightarrow[rm]{} yBx \xrightarrow[rm]{} aBy \quad y$$

FIRST(w) = FIRST(y)

implican que aAw = yBx

REDUCCIONES

Cuando se realiza el parsing de arriba a abajo los simbolos no terminales se expanden por el lado derecho de una de las alternativas de la regla de producción. En el parsing de 'abajo a arriba' una forma sentencial correspondiente al lado derecho de la producción es reemplazada por el no terminal de la izquierda, esta operación se llama 'reducción', el subcadena que se 'reduce' se llama 'handle'.

'HANDLE' O MANIJA

El 'handle' de una forma sentencial es la frase simple más a la izquierda de la producción. En otras palabras el 'handle' de una cadena de simbolos gramaticales es una subcadena de éste, que es igual a una de las alternativas de alguna producción y cuya reducción a un no terminal es un paso en reversa de la derivación más a la derecha de esa producción.

GRAMATICA AUMENTADA

Las gramáticas que reconocen los 'parsers' LR son aumentadas

CONSTRUCCION DE 'PARSERS' LR(0)

Para construir un analizador LR(0) se utilizan como herramientas los conceptos de 'configuración' y cerradura sobre 'configuraciones' para construir el autómata LR(0) que nos define este analizador.

```
                CONJUNTOS DE            AUTOMATA            TABLA
GRAMATICA  --> CONFIGURACIONES  --> CARACTERISTICO --> PARSING
```

CONJUNTO DE ELEMENTOS LR(0)

Los 'parsers' LR se basan en el concepto de 'configuración' o 'elemento'(item), que se representa por una regla de la gramática a analizar con un punto o marca en algún lugar del lado derecho de las producciones, de la forma:

$$A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j$$

El simbolo de punto o 'marca' en una 'configuración' puede aparecer en cualquier parte del lado derecho de la producción e indica cuantos simbolos de la producción han sido leidos o analizados.

La posición de la marca en una configuración nos dice si se

debe hacer un 'shift' de un simbolo o una reducción en el 'stack', en otras palabras una 'configuración' con el punto antes del final de la producción nos indica una operación de 'shift' y otra con el punto al final de la producción o en la extrema derecha indica que se ha reconocido un 'handle' y se debe de realizar una 'reducción'.

Las 'configuraciones' se agrupan en conjuntos que representan estados de un autómata finito que acepta prefijos viables. Al procedimiento de generar estos conjuntos se le llama 'cerradura'. Esta operación formaliza la búsqueda del parser de todos los 'handles' basados en derivaciones de un no terminal.

La operación de 'cerradura' genera los estados del autómata y garantiza que todas las 'configuraciones' necesarias para generar expresiones legales se incluyan.

FUNCION DE CERRADURA(I) LR(0)

Sean I y CERRADURA(I) conjuntos de 'configuraciones'.

1. Cada configuración 'I' se incluye en CERRADURA(I)
2. Si la configuración [B-> w . Cz] esta en CERRADURA(I), C E N y C->v es una regla de producción, entonces se agrega C-> . v a CERRADURA(I).
3. se aplica 2 hasta que no existan producciones

Dado un conjunto de 'configuraciones' I se puede calcular el conjunto sucesor I' cuando se avanza leyendo el siguiente simbolo gramatical X, mediante la función GOTO(I,X).

FUNCION GOTO(I,X) LR(0)

Con la función GOTO calculamos el conjunto de las configuraciones sucesoras a un estado si se leen los simbolos gramaticales que suceden al punto o marca.

Sean I, GOTO(I,X) y C -> V.XZ conjuntos de 'configuraciones' y X un simbolo gramatical.

GOTO(I,X) = CERRADURA(C-> VX . Z)
donde C-> V . XZ E I

en otras palabras se avanza la marca en todas las 'configuraciones' que antecedan a X y se calcula la CERRADURA.

Si la función GOTO(I,X) = Ø , entonces no existe ninguna configuración que tenga sucesor en X, y durante el 'parsing' se indica como error de sintaxis.

Teniendo las funciones de CERRADURA(I) y GOTO(I,X) se puede

construir un autómata finito que asocie los conjuntos de 'configuraciones' y operaciones de lectura de símbolos con estados del autómata y sus transiciones.

CONSTRUCCION DEL AUTOMATA LR(0)

Sea

1. AUTOMATA = CERRADURA(S -> .a). Se aplica la cerradura a la configuración asociada al símbolo de inicio.
2. Por cada conjunto de 'configuraciones' I y cada símbolo X tal que GOTO(I,X) es no vacío y no esta en CSI, se agrega GOTO(I,X) al AUTOMATA
3. Se repite 2 hasta que no se puede agregar nada al AUTOMATA

ejemplo. Dada la siguiente gramática construya el conjunto de configuraciones y autómata LR(0)

La gramática está formada por seis producciones que se numeran para identificarlas

1. S -> E$
2. E -> T
3. E -> E + T
4. E -> E - T
5. T -> id
6. T -> (E)

El conjunto de configuraciones se forma:

Conjunto de configuraciones S0

Se aplica la función CERRADURA(S -> E$)

```
S -> .E$
E -> .T
E -> .E + T
E -> .E - T
T -> .id
T -> .(E)
```

Para formar los conjuntos sucesores se deben de leer los simbolos gramaticales en este caso E, T, id y (, utilizando la función GOTO.

configuración S1

Se aplica la función GOTO(S0,E) y se obtiene:

```
S -> E.$
E -> E. + T
E -> E. - T
```


configuración S2

Se aplica la función GOTO(S0,T) y se obtiene la reducción de la producción 2.

```
E -> T.
```

configuración S3

Se aplica la función GOTO(S0,id) y se obtiene la reducción de la producción 5.

```
T -> id.
```

configuración S4

Se aplica la función GOTO(S0,() y se obtiene:

```
T -> ( . E )
E -> .T
E -> .E + T
E -> .E - T
T -> .id
T -> .(E)
```

configuración S5

Se aplica la función GOTO(S1,$) y se obtiene la aceptación.

```
S -> E$.
```

configuración S6

Se aplica la función GOTO(S1,+) y se obtiene:

```
E -> E + .T
T -> .id
T -> .( E )
```

configuración S7

Se aplica la función GOTO(S1,-) y se obtiene:

```
E -> E - .T
T -> .id
T -> .( E )
```

configuración S8

Se aplica la función GOTO(S4,E) y se obtiene:

T -> (E .)
E -> E .+ T
E -> E .- T

configuración S9

Se aplica la función GOTO(S6,T) y se obtiene la reducción de la producción 3.

E -> E + T.

configuración S10

Se aplica la función GOTO(S7,T) y se obtiene la reducción de la producción 4.

E -> E - T.

configuración S11

Se aplica la función GOTO(S8,)) y se obtiene la reducción de la producción 6.

T -> (E).

El autómata resultante es:

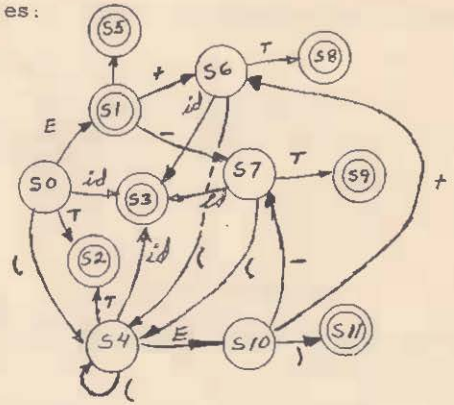


TABLA DEL PARSER LR(0)

El 'parser' LR(0) no requiere leer símbolos de la entrada o 'lookahead' para realizar las reducciones, esta se determina únicamente por el contenido del 'stack'

La tabla de 'parsing' para el ejemplo anterior es:

	ACCIONES 'STACK'						SIGUIENTE ESTADO							
EDO	id	+	-	(	)	$	id	+	-	(	)	$	S	E	T
0	S			S			3			4				1	2
1		S	S			S		6	7			5			
2	R1	R1	R1	R1	R1	R1									
3	R4	R4	R4	R4	R4	R4									
4	S			S			3			4				8	
5	A	A	A	A	A	A									
6	S			S			3			4					9
7	S			S			3			4					10
8		S	S		S			6	7		11				
9	R2	R2	R2	R2	R2	R2									
10	R3	R3	R3	R3	R3	R3									
11	R5	R5	R5	R5	R5	R5									

CONFLICTOS

Existen dos tipos de conflictos que pueden ocurrir al generar el autómata:

- 'shift'/reducción
- reducción/reducción

En el primero un shift y una reducción son posibles en una 'configuración'. En el segundo dos reducciones son posibles en la misma 'configuración'

PARSER SLR(1)

El nombre del parser SLR(1) significa Simple LR(1) y fué creado por DeRemer en 1971. Este método se basa en construir las configuraciones LR(0) y después agregarle el lookahead.

GRAMATICA SLR(1)

Una gramática es SLR(1) si para poder solucionar un conflicto de 'shift'/reducción se tiene que examinar un símbolo de más en la entrada utilizando la función FOLLOW para poder predecirlo, y en el conjunto de símbolos terminales resultantes no existen símbolos comunes.

ejemplo. Dada la siguiente gramática construya el conjunto de configuraciones, autómata SLR(1).

La gramática está formada por siete producciones que se numeran:

A. S -> E$
1. E -> E - T
2. E -> T

3. T -> F ^ T
4. T -> F
5. F -> (E)
6. F -> id

Se forma el conjunto de configuraciones:

Conjunto de configuraciones S0

Se aplica la función CERRADURA(S -> E$)

S -> .E$
E -> .E - T
E -> .T
T -> .F ^ T
T -> .F
F -> .(E)
F -> .id

configuración S1

Se aplica la función GOTO(S0,E) y se obtiene:

S -> E.$
E -> E. - T

configuración S2

Se aplica la función GOTO(S0,T) y se obtiene la reducción de la producción 2.

E -> T.

configuración S3

Se aplica la función GOTO(S0,F) y se obtiene:

T -> F .^ T
T -> F.

donde se tiene la reducción 4 y un 'shift' para otra.

configuración S4

Se aplica la función GOTO(S0,id) y se obtiene la reducción de la producción 6.

F -> id.

configuración S5

Se aplica la función GOTO(S0,() y se obtiene:

F -> (. E)
E -> .E - T
E -> .T
T -> .F ^ T
T -> .F
F -> .(E)
F -> .id

configuración S6

Se aplica la función GOTO(S1,$) y se obtiene la aceptación.

S -> E$.

configuración S7

Se aplica la función GOTO(S1,-) y se obtiene:

E -> E - .T
T -> .F ^ T
T -> .F
F -> .(E)
F -> .id

configuración S8

Se aplica la función GOTO(S7,T) y se obtiene la reducción de la producción 1.

E -> E - T.

configuración S9

Se aplica la función GOTO(S5,E) y se obtiene:

F -> (E .)
E -> E .- T

configuración S10

Se aplica la función GOTO(S9,)) y se obtiene la reducción de la producción 5.

F -> (E).

configuración S11

Se aplica la función GOTO(S3,^) y se obtiene:

T -> F ^ .T
T -> .F ^ T
T -> .F
F -> .(E)
F -> .id

configuración S12

Se aplica la función GOTO(S11,T) y se obtiene la reducción de la producción 3.

T -> F ^ T.

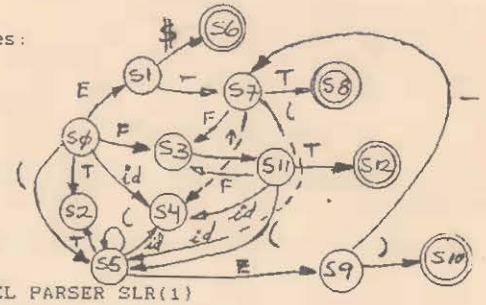
Resolución de estados de conflicto del ejemplo:

Para construir el autómata tenemos que resolver los estados de 'shift'/reducción en la producción 4, utilizando la función FOLLOW tenemos:

FOLLOW(F) = FOLLOW(T) = FOLLOW(E) = { $,), ^ }

Por lo que la reducción 4 se realiza en estos simbolos.

El autómata resultante es:



CONSTRUCCION DE TABLA DEL PARSER SLR(1)

ejemplo. Dado el autómata SLR(1) del ejemplo anterior construya la tabla del 'parser' SLR(1).

La tabla de 'parsing' para el ejemplo anterior es:

	ACCIONES 'STACK'						SIGUIENTE ESTADO								
EDO	id	-	^	(	)	$	EPS	id	-	^	(	)	$	S	E	T	F
0	S			S				4			5				1	2	3
1		S				S			7				6				
2		R2			R2	R2											
3		R4	S		R4	R4				11							
4		R6	R6		R6	R6											
5	S			S				4			5				9	2	3
6							A										
7	S			S				4			5					8	3
8		R1			R1	R1											
9		S			S				7				10				
10		R5	R5		R5	R5											
11	S			S				4			5					12	3
12		R3			R3	R3											

ALGORITMO PARSER SLR(1)

```
Sea
     a:   un caracter de entrada o 'token',
     i, T:  estados del parser;
     TOPE_STACK: el estado en el tope del 'stack';
     Rk: la reducción k de una regla de producción de la
         gramática del parser;
     SIG_EDO[ i,a ]: la parte de siguiente estado de la tabla del
                     parser SLR(1);
     ACCION[ i,a ]: la parte de acción en el 'stack' del parser;
     NO_ELEM[ Rk ]:  arreglo con el número de símbolos en cada
                     producción;
     LHS[Rk]: lado izquierdo de la producción;
     S:  indicación de 'shift' en el 'stack';

Begin
     Push del estado i inicial en el 'stack';
     lee caracter a;
     Repeat
         If  ACCION[ TOPE_STACK,a ] = 'S' then
                  Begin
                      T <- SIG_EDO[ TOPE_STACK,a ];
                      Push T;
                      lee siguiente caracter a;
                  end
         else
             If ACCION[ TOPE_STACK,a ] = 'Rk' then
                  Begin
                      For J = 1 to NO_ELEM[ Rk ];
                          pop TOPE_STACK;
                      T <- SIG_EDO[ TOPE_STACK, LHS[Rk] ];
                      Push T;
                  End;
         else If ACCION[ TOPE_STACK, a ] = 'A' then
                  Begin
                      write('cadena aceptada');
                      exit;
                  End;
         else Begin
                  write('error');
                  exit;
              End
   Until( a = EPSILON )
End.
```

ejemplo. Dada la tabla de 'parsing' del ejemplo anterior construya el 'parser' SLR(1).

Se escribe un programa en lenguaje 'C' que realiza el análisis sintáctico de este lenguaje en base al algoritmo del parser SLR(1).

Las tablas de 'parsing' se toman del ejemplo anterior y se crea un vector llamado 'no_elem' que contiene el número de simbolos gramaticales del lado derecho de las producciones, además del vector 'lhs' que contiene el número de columna que da la posición del simbolo gramatical del lado izquierdo de la producción. este simbolo es el que se introduce en el 'stack' en el momento de una reducción.

El programa imprime en cada 'shift' y reducción el contenido del 'stack'.

```
/* Facultad de Ingeniería                                  */
/* Apuntes de compiladores                                 */
/* prof: A. Jiménez H. .                                   */
/* ejemplo de parser realizado manualmente                 */
/*                  parser LR                              */
/*                                                         */
/*                                                         */
/*                                                         */

#include <stdio.h>
#define EDO 13      /*  número de estados                  */
#define SIM 8       /*  número de simbolos de entrada      */
#define SGR 11      /*  simbolos gramaticales              */
#define RED 7       /*  número de reducciones              */

char *malloc();
int *p1, *tos;

/* acciones en el stack    */
/* tokens de entrada ... i - ^ ( ) ; e    */

int accion[EDO][SIM] = {7,0,0,7,0,0,0,0, /* edo. inicial */
                        0,7,0,0,0,7,0,0,
                        0,2,0,0,2,2,0,0,
                        0,4,7,0,4,4,0,0,
                        0,6,6,0,6,6,0,0,
                        7,0,0,7,0,0,0,0,
                        0,0,0,0,0,0,8,0,
                        7,0,0,7,0,0,0,0,
                        0,1,0,0,1,1,0,0,
                        0,7,0,0,7,0,0,0,
                        0,5,5,0,5,5,0,0,
                        7,0,0,7,0,0,0,0,
                        0,3,0,0,3,3,0,0};

/*  función siguiente estado */
/* simbolos gramaticales. i - ^ ( ) ; S E T F  */

int sig_edo[EDO][SGR] = {4,0,0,5,0,0,0,1,2,3,0, /* edo inicial */
                         0,7,0,0,0,6,0,0,0,0,0,
                         0,0,0,0,0,0,0,0,0,0,0,
                         0,0,11,0,0,0,0,0,0,0,0,
                         0,0,0,0,0,0,0,0,0,0,0,
                         4,0,0,5,0,0,0,9,2,3,0,
```



```
                         0,0,0,0,0,0,0,0,0,0,0,
                         4,0,0,5,0,0,0,0,8,3,0,
                         0,0,0,0,0,0,0,0,0,0,0,
                         0,7,0,0,10,0,0,0,0,0,0,
                         0,0,0,0,0,0,0,0,0,0,0,
                         4,0,0,5,0,0,0,0,12,3,0,
                         0,0,0,0,0,0,0,0,0,0,0};

/*  No terminales:  E,  T,  F   */

int lhs[RED] =  {6,7,7,8,8,9,9};

/*  número de elementos en producciones          */

int no_elem[RED] = {2,3,1,3,1,3,1};


main()
{
  int valor1=0;
  int token,t;
  int edo = 0;
  int red, simbolo, i;

  printf("\t parser LR\n");
  printf("expresiones validas terminan con ; \n\n");
  p1 = (int *) malloc(20 * sizeof(int));
  if(!p1) {
     error(0);
     return;
     }
  tos = p1;    /* tos es tope del stack */

      push(valor1);
      printf("dame estring de entrada = ");
      simbolo = lee();
      while (1) {
          edo = topv();
          if (accion[edo][simbolo] == 7) {        /* shift */
                t = sig_edo[edo][simbolo];
                printf("..shift edo: %d\n",t);
                push(t);
                imprime();
                simbolo = lee();
              }
          else if (accion[edo][simbolo] == 0) {  /* error */
                imprime();
                error(2);
                exit(1);
              } /* end error */
          else if ((red = accion[edo][simbolo]) < 7){  /* reducción */
                  printf(" reduccion = %d \n",red);
                  for(i=1;i <= no_elem[red]; i++ )
                        printf(" pop estado %d \n",pop());
                  t = sig_edo[topv()][lhs[red]];
```

```
                if (t == 0) {
                        error(2);
                        exit(1);}
                push(t);
                printf("push estado %d\n",t);
                imprime();
                }
        else if (accion[edo][simbolo] == 8) {  /* aceptación */
                printf("string de entrada aceptado\n");
                exit(0);
                }
        } /*  endwhile  */
}

/* rutina para leer cadena de entrada y reconocer 'tokens' */
lee()

{
int ch;

while((ch = getchar()) ==  ' ' || ch == '\t' ) ;
   if (ch == '\n')
          return(6);
   else  if (ch == 'i')
          { printf(".....lei token = i\n") ; return(0);}
   else  if (ch == '-')
          { printf(".....lei token = -\n"); return(1); }
   else  if (ch == '^')
          { printf(".....lei token = ^\n"); return(2); }
   else  if (ch == '(')
          { printf(".....lei token = (\n"); return(3); }
   else  if (ch == ')')
          { printf(".....lei token = )\n"); return(4); }
   else  if (ch == ';')
          { printf(".....lei token = ;\n"); return(5); }
   else  { printf(" caracter = %c",ch);
          error(1);
          return(7);
        }
}


push(i)
int i;
{
 p1++;        /*    push estado       */
 if ((p1) == (tos + 20)) {
          printf("stack overflow\n");
          exit();
          }
 *p1 = i;
 }

pop()
{
```



```
if(p1 == tos) {
          printf("stack underflow\n");
          exit();
          }
 p1--;       /*  remuevelo     */
 return *(p1+1);
}

topv()
{
 if(p1 == tos) {
          printf("stack underflow\n");
          exit();
          }
 return *(p1);
}

imprime()
{
 int cuenta, diff;
 diff = (int)(p1-tos);
 printf("\t stack = ");
 for( cuenta = 1 ; cuenta <= diff ; cuenta++)
          printf(" %d ",*(tos + cuenta));
 printf("\n");
 }

error(nerror)
int nerror;
{
 static char *err[] = {
          "*** falla inicialización ***",
          "*** token no permitido ***",
          "*** error de sintaxis ***"};
printf("\n%s\n",err[nerror]);
}
```

PARSERS LR(1)

Este es el método más general, y tiene la particularidad de calcular el 'lookahead' al generar el autómata para el 'parser'.

ELEMENTO LR(1)

Un elemento LR(1) esta formado por dos componentes:

[A -> ALFA . BETA, u]

donde el primer componente es una producción marcada a -> ALFA BETA, llamada 'core' del elemento y u es un caracter de 'lookahead' que es un 'token'. Como en el elemento LR(0) la producción con el punto representa cuanto del lado derecho de la producción ha sido leido y el término 'lookahead' representa un

posible simbolo de 'lookahead' después de que la producción ha sido reconocida.

El agregar el 'lookahead' de esta forma le da al 'parser' un mayor poder a los métodos de 'parsing', pero aumenta significativamente el número de configuraciones posibles. Es frecuente encontrar configuraciones que solo son diferentes en el simbolo de 'lookahead'.

Un elemento LR(1) de la forma [A -> ALFA . BETA, u] es válido para un prefijo viable LAMBDA, si existe una derivación más a la derecha de la forma:

S => FI A t => FI ALFA BETA t

donde LAMBDA = FI ALFA y u es el primer simbolo terminal de t o EPSILON si t = EPSILON.

La definición del 'parser' SLR(1) es:

Si existe un elemento

[A -> ALFA . X BETA, u] y x -> DELTA

y existe una derivación de la forma:

S => FI A w => FI ALFA X BETA u w

entonces [X -> . DELTA, v]

es válido para el prefijo viable FI ALFA donde

v = { FIRST(BETA u w) }

y v = FIRST(BETA)
o v = u si BETA => EPSILON

FUNCION DE CERRADURA(I) LR(1)

Sean I y CERRADURA(I) conjuntos de 'configuraciones'.

1. cada configuración en 'I' se incluye en CERRADURA(I)
2. si la configuración [B-> w . Cz, b] esta en CERRADURA(I), C E N y C->v es una regla de producción donde el 'lookahead' es first(zb) entonces se agrega [C-> .v, c] a CERRADURA(I), si no se encuentra ya en ésta. Si el núcleo de la configuración ya existe en dos configuraciones se unen éstas con sus 'lookaheads'.
3. se aplica 2 hasta que no existan producciones

Dado un conjunto de 'configuraciones' I se puede calcular el conjunto sucesor I' cuando se avanza leyendo el siguiente símbolo gramatical X, mediante la función GOTO(I,X).

FUNCION GOTO(I,X) LR(1)

Sean I, GOTO(I,X) y [C -> V.XZ, b] conjuntos de 'configuraciones' y X un simbolo gramatical.

GOTO(I,X) = CERRADURA([C-> VX . Z, b])
donde [C->V . XZ, b] E I

se avanza la marca en todas las 'configuraciones' que antecedan a X y se calcula la CERRADURA.

Si la función GOTO(I,X) = Ø , entonces no existe ninguna configuración que tenga sucesor en X, y durante el 'parsing' se indica como error de sintaxis.

CONSTRUCCION DEL AUTOMATA LR(1)

Sean S0, S1, S2 ..... Sn estados del Autómata LR(1)

1. S0 = CERRADURA([S -> .A, epsilón]). Se aplica la cerradura a la configuración asociada al simbolo de inicio.
2. Por cada conjunto de 'configuraciones' I de la forma [A -> ALFA . X BETA, c] pertenecientes a un estado y cada símbolo X tal que GOTO(I,X) es no vacío y no esta en el AUTOMATA, se crea un nuevo estado Sn asociado a GOTO(I,X).
3. Se repite 2 hasta que no se pueden crear nuevos estados del AUTOMATA.

ejemplo. Dada la siguiente gramática calcule el autómata LR(0).

0. S -> E$
1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> id
6. F -> (E)

Se forma el conjunto de configuraciones:

Conjunto de configuraciones S0

Se aplica la función CERRADURA(S -> E$)

[S -> .E$, EPSILON]

Se deriva el no terminal E con lookahead de '$'

[E -> .E + T, $] , [E -> .T, $]

De la configuración [E -> .E + T, $] se deriva

[E -> .E + T, +] , [E -> .T, +]

de la configuración [E -> .T, $] se deriva:

[T -> .T * P, $] , [T -> .P, $]

de la configuración [T -> .P, $] se deriva:

[T -> .P, $] , [T -> .T * P, $]

de la configuración [T -> .T * F, $] se deriva:

[T -> .P, *] , [T -> .T * P, *]

[P -> .(E), +] , [P -> .id, +]

Se unifican 'configuraciones' con la misma producción pero diferente simbolo de 'lookahead' quedando lo siguiente:

```
[ S -> .E$, EPSILON ]
[ E -> .E + T, $+ ]
[ E -> .T, $+ ]
[ T -> .T * F, $+* ]
[ T -> .F, $+* ]
[ F -> .(E), $+* ]
[ F -> .id, $+* ]
```

Realizando el mismo procedimiento obtenemos las demás configuraciones.

configuración S1

Se aplica la función GOTO(S0,E) y se obtiene:

```
[ S -> E.$, EPSILON ]
[ E -> E. + T, $+ ]
```

configuración S2

Se aplica la función GOTO(S0,T) y se obtiene la reducción de la producción 2.

```
[ E -> T., $+ ]
[ T -> T. * F, $+* ]
```

configuración S3

Se aplica la función GOTO(S0,F) y se obtiene la reducción 4:

[T -> F., $+*]

configuración S4

Se aplica la función GOTO(S0,id) y se obtiene la reducción de la producción 6.

[F -> id., $+*]

configuración S5

Se aplica la función GOTO(S0,() y se obtiene:

```
[ F -> ( . E ), $+* ]
[ E -> .E + T, )+]
[ E -> .T, )+ ]
[ T -> .T * F, )+* ]
[ T -> .F, )+* ]
[ F -> .(E), )+* ]
[ F -> .id, )+* ]
```

configuración S6

Se aplica la función GOTO(S1,$) y se obtiene la aceptación.

[S -> E$., EPSILON]

configuración S7

Se aplica la función GOTO(S1,+) y se obtiene:

```
[ E -> E + .T, $+ ]
[ T -> .T * F, $+* ]
[ T -> .F, $+* ]
[ F -> .(E), $+* ]
[ F -> .id, $+* ]
```

configuración S8

Se aplica la función GOTO(S2,*) y se obtiene:

```
[ T -> T * .F , $+* ]
[ F -> . id, $+* ]
[ F -> .( E ), $+* ]
```

configuración S9

Se aplica la función GOTO(S5,E) y se obtiene:

```
[ F -> ( E . ), $+* ]
[ E -> E . + T, )+]
```

configuración S10

Se aplica la función GOTO(S5,E) y se obtiene:

[E -> T .,)+]
[T -> T . * F,)+*]

configuración S11

Se aplica la función GOTO(S5,F) y se obtiene:

[T -> F. ,)+*]

configuración S12

Se aplica la función GOTO(S5,id) y se obtiene:

[F -> id . ,)+*]

configuración S13

Se aplica la función GOTO($5,() y se obtiene:

[F -> (. E),)+*]
[E -> .E + T,)+]
[E -> .T,)+]
[T -> .T * F,)+*]
[T -> .F,)+*]
[F -> .(E),)+*]
[F -> .id,)+*]

configuración S14

Se aplica la función GOTO(S7,T) y se obtiene:

[E -> E + T. , $+]
[T -> T. * F, $+*]

Las demás transiciones en S7 son:

GOTO(S7,F) = S3
GOTO(S7,id)= S4
GOTO(S7,()= S5

configuración S15

Se aplica la función GOTO(S8,F) y se obtiene:

[T -> T. * F, $+*]

Las demás transiciones en S8 son:

GOTO(S8,id)= S4
GOTO(S8,()= S5

configuración S16

Se aplica la función GOTO(S9,)) y se obtiene:

[F -> (E). , $+*]

configuración S17

Se aplica la función GOTO(S9,+) y se obtiene:

[E -> E + .T,)+]
[T -> .T * F,)+*]
[T -> .F,)+*]
[F -> .(E),)+*]
[F -> .id,)+*]

configuración S18

Se aplica la función GOTO(S10,*) y se obtiene:

[T -> T * .F,)+*]
[F -> .(E),)+*]
[F -> .id,)+*]

Para las transiciones de S13 tenemos:

GOTO(S7,T) = S10
GOTO(S7,F) = S11
GOTO(S7,id)= S12
GOTO(S7,()= S19

configuración S19

Se aplica la función GOTO(S13,E) y se obtiene:

[F -> (E.),)+*]
[E -> E. + T,)+]

GOTO(S14,*)= S8

configuración S20

Se aplica la función GOTO(S17,T) y se obtiene:

[E -> E + T.,)+]
[T -> T. * F,)+*]

GOTO(S17,F) = S11
GOTO(S17,id)= S12
GOTO(S17,()= S13

configuración S21

Se aplica la función GOTO(S18,F) y se obtiene:

[T -> T * F. ,)+*]

configuración S22

Se aplica la función GOTO(S19,)) y se obtiene:

[F -> (E). ,)+*]

PARSERS LALR(1)

Los 'parsers' LALR(1) se pueden construir a partir de un parser LR(1) en el que se unen estados que solo difieren entre sí por el componente de 'lookahead'. LALR es el acrónimo de 'lookahead' LR y significa que a un autómata se le agrega el 'lookahead'.

GENERADORES DE ANALIZADORES DE SINTAXIS YACC

YACC es un generador de analizadores de sintaxis o 'parsers' LALR(1) y es el acrónimo de la frase en inglés de 'Yet Another Compiler-Compiler'. YACC puede utilizar 'scanners' generados por LEX o hechos por el usuario.

YACC recibe como entrada la especificación de una gramática de contexto libre que describe un lenguaje de programación. Esta especificación se compone de: las reglas de producción de esa gramática, el código que es invocado o acción a realizar cuando una regla es reconocida y el programa o función para hacer la lectura de los 'tokens' de entrada. Como salida se obtiene un programa en C o Ratfor que analiza o verifica del lenguaje.

ESTRUCTURA DE YACC

Cada programa en YACC consiste de tres secciones: declaraciones, reglas de la gramática y código de acción, las cuales están separadas entre si por medio de los simbolos %%

declaraciones
%%
reglas de la gramática

%%
código de acción

DECLARACIONES

La parte de declaraciones es opcional, los comentarios aparecen encerrados entre los simbolos /* */, como los comentarios en lenguaje 'C'. Los nombres que representan 'tokens' deben ser declarados en esta parte, anteponiendoles %token.

Los nombres que se asignan en YACC, sirven para denominar tanto a simbolos terminales o 'tokens' como a simbolos no-terminales.

Cada nombre no definido como 'token' o terminal se asume como un simbolo no-terminal. Cada simbolo no terminal debe aparecer en el lado izquierdo de al menos una regla.

El 'parser' reconoce como símbolo de inicio el no-terminal del lado izquierdo de la primera regla de la gramática en la sección de reglas. Este simbolo se puede declarar explicitamente anteponiendo %start.

REGLAS DE PRODUCCION

La parte de reglas de producción está formada por producciones gramaticales. Una regla gramatical tiene la siguiente forma:

A : cuerpo;

donde A representa un nombre de un símbolo noterminal, y el cuerpo representa una secuencia de cero o más nombres y literales. Los símbolos : y ; son símbolos de separación de Yacc.

Los nombres pueden tener una longitud arbitraria, pueden estar formados por letras, puntos, subguión y digitos no iniciales, las letras mayúsculas y minúsculas son distintas, los nombres usados en el cuerpo de la regla de la gramática pueden representar tokens o simbolos no-terminales.

Una literal consta de un caracter encerrado en comillas sencillas, el "\" es un caracter de escape dentro de las literales, todos los símbolos después del escape son reconocidos como tales. Si existen demasiadas reglas gramaticales en el lado derecho, se utiliza la barra vertical para evitar reescribir otra vez el lado izquierdo.

No es necesario que todas las reglas gramaticales con el mismo lado izquierdo aparezcan juntas en la sección de reglas, esto hace a la entrada más legible y fácil de modificar en la sección de declaraciones.

El final de la entrada al parser es señalado por el token de fin de archivo (end-of-file) o fin de registro (end-of-record).

ACCIONES

Cuando se invoca a YACC, el usuario especifica un lenguaje de programación. Una acción es un programa que puede ejecutar lectura de entradas, salidas, llamadas a subprogramas, y alterar tablas de simbolos y variables, y se especifica por medio de una o más estructuras, encerradas entre los caracteres { y }.

Las acciones que no terminan una regla, son manejadas por YACC por medio de la fabricación de un nuevo nombre de un simbolo no-terminal, una nueva regla marca este nombre en una cadena vacia.

En muchas aplicaciones, la salida no se hace directamente por medio de acciones, en lugar de ello se construye en memoria, una estructura de datos, como un árbol de 'parsing', y las transformaciones se aplican antes de generar la salida. Para construir este árbol el usuario debe de proporcionar las rutinas.

El usuario pude definir otras variables para ser usadas por las acciones. Las declaraciones y definiciones pueden aparecer en dos lugares de las especificaciones de Yacc: en la sección de declaraciones y en el encabezado de la sección de reglas, antes de la primera regla de la gramática. En cada caso, las declaraciones y definiciones están encerradas entre las marcas %{ y %}

MANEJO DE CONFLICTOS

Un conjunto de reglas gramaticales es ambiguo si existe una cadena de entrada que pueda tener dos o más árboles de 'parsing'. Existen dos tipos de conflictos:

- shift-reducción
- reducción-reducción

Para solucionar el conflicto shift-reducción se selecciona el 'shift', y en el de reducción-reducción' se selecciona la primera regla gramatical.

PRECEDENCIA

Las reglas de precedencia y asociatividad son usadas para resolver problemas de prioridad, se asignan a los 'tokens' en la sección de declaraciones, esto se realiza mediante la declaración de: %left, %right, o %nonassoc, seguida por una lista de tokens.

%left asocia los operadores a la izquierda, %rigth a la derecha y %nonassoc la anula.

Se considera que todos los tokens en la misma línea tienen el mismo nivel de precedencia y asociatividad; las líneas se enlistan en orden de precedencia ascendente.

MANEJO DE ERRORES

Cuando un error es encontrado, puede ser necesario mejorar el árbol de 'parsing', borrar o alterar la tabla de simbolos, y, en algunos casos, colocar interrupciones para evitar generar salidas adicionales. Es aceptable parar el procesamiento cuando se encuentra un error, sin embargo, es más usado el continuar rastreando la salida para encontrar mas errores.

GENERACION DE MAQUINAS LALR(1) MEDIANTE YACC

YACC proporciona la máquina LALR(1) de la gramática si se da la opción -v al procesar el archivo de entrada.

YACC -i EJEMPLO

Si se procesa en YACC la siguiente gramática:

```
/*  Apuntes de compiladores                 */
/*  gramática de suma y multiplicación      */
/*  EJEMPLO.Y                               */

%term ID END
%left '+'
%left '*'
%left '('

%%

/*  Definición de gramática     */

E:      E '+' T         = { 1; }
        | T             = { 2; }

T:      T '*' F         = { 3; }
        | F             = { 4; }

F:      '(' E ')'       = { 5; }
        | ID            = { 6; }

%%
```

Se obtendrá el listado de la máquina LALR(1) correspondiente a la anterior gramática:

```
state 0
```

```
$accept : _E $end

ID  shift 5
(  shift 4

E  goto 1
T  goto 2
F  goto 3

state 1
$accept :  E_$end
E :  E_+ T

$end  accept
+  shift 6


state 2
E :  T_     (2)
T :  T_* F

*  shift 7


state 3
T :  F_     (4)


state 4
F :  (_E )

ID  shift 5
(  shift 4

E  goto 8
T  goto 2
F  goto 3

state 5
F :  ID_     (6)


state 6
E :  E +_T

ID  shift 5
(  shift 4

T  goto 9
F  goto 3

state 7
T :  T *_F
```



```
ID  shift 5
(  shift 4

F  goto 10

state 8
E :  E_+ T
F :  ( E_)

+  shift 6
)  shift 11


state 9
E :  E + T_     (1)
T :  T_* F

*  shift 7


state 10
T :  T * F_     (3)


state 11
F :  ( E )_     (5)


8/127 terminals, 3/100 nonterminals
7/200 grammar rules, 12/450 states
0 shift/reduce, 0 reduce/reduce conflicts reported
7/125 working sets used
memory: states,etc. 64/1500, parser 8/1000
8/200 distinct lookahead sets
4 extra closures
13 shift entries, 1 exceptions
6 goto entries
3 entries saved by goto default
Optimizer space used: input 37/1500, output 218/1000
218 table entries, 206 zero
maximum spread: 257, maximum offset: 43
```

YACC agrega la configuración correspondiente a la producción de aceptación o inicio:

```
$accept : _E $end
```

El guión _ hace las veces del punto en la configuración e indica el punto donde se está realizando la cerradura LR o el análisis de la gramática.

YACC lista las configuraciones que forman los estados del autómata indicando las acciones en el 'stack', los estados siguientes del autómata correspondientes a la función GOTO y en caso de reducciones el número de la reducción. Así por ejemplo en el estado 0 del tenemos:

```
state 0                 ........ número de estado
$accept : _E $end       ........ descripción del nucleo de la
                                 configuración

ID  shift 5
(  shift 4              ....... acciones en el 'stack' y siguiente

E  goto 1
T  goto 2               ....... siguiente estado para reducciones
F  goto 3                        de no terminales
```

En el estado de reducción normalmente el caracter de 'lookahead' para reducir es cualquiera y se indica mediante un punto '.' que significa cualquier caracter.

```
state 10                        ......... número de estado
T :  T * F_     (3)             ......... configuración para
                                           reducción
```

En los casos especiales de shift-reducción donde existe conflicto, la máquina LALR(1) los resuelve al construirse. Así en el anterior ejemplo al especificar el shift ya se resolvió el conflicto que existia con los elementos de reducción.

Finalmente se presentan las estadisticas de la máquina LALR(1).

Ejemplo: para la siguiente gramática genere el programa en YACC para el analizador de sintaxis o 'parser'.

La gramática del lenguaje es:

```
PROGRAMA        -> program id; DECLARACIONES BLOQUE.

DECLARACIONES   -> var  DECLR_VAR | EPSILON
DECLR_VAR       -> VARIABLES | VARIABLES ; DECLR_VAR
VARIABLES       -> id : TIPO
TIPO            -> integer

BLOQUE          -> PROCEDURES INSTRUCCIONES
PROCEDURE       -> procedure id(VARIABLES); INSTRUCCIONES
                 | EPSILON

INSTRUCCIONES   -> begin LISTA_DECLR end
LISTA_DECLR     -> DECLR | DECLR ; LISTA_DECLR
DECLR           -> WHILE | IF | ASIGNA | INSTRUCCIONES | LLAMA_PROC
```



```
WHILE           -> WILE DECLR
WILE            -> while EXPR do
IF              -> FEXPR  THEN
                   | FEXPR  THENELSE
FEXPR           -> if EXPR
THEN            -> then DECLR  ;
THENELSE        -> INICIO  DECLR
INICIO          -> then DECLR else
ASIGNA          -> id := EXPR
LLAMA_PROC      -> id(id)
EXPR            -> EXPR_ENT | EXPR_ENT RELACION EXPR_ENT
EXPR_ENT        -> TERM    |  TERM SUMRES EXPR_ENT
TERM            -> FACTOR  |  FACTOR MULDIV TERM
SUMRES          -> +  |  -
MULDIV          -> *  |  /
FACTOR          -> id | num | ( EXPR )
RELACION        -> > | < | = | <> | >= | <=
```

EXPRESIONES REGULARES PARA LOS TOKENS

```
id      -> LETRA (LETRA'| DIGITO)*
LETRA   -> [A-Za-z]
DIGITO  -> [0-9]
NUM     -> DIGITO+
```

donde EPSILON = caracter sin elementos

Los comentarios no deben de ser tomados en cuenta y se encierran entre paréntesis { }

```
/* programa en YACC para la gramática  */

%term id end
%term num end
%token 'do'
%token 'if'
%token 'then'
%token 'else'
%token 'program'
%token 'while'
%token 'var'
%token 'begin'
%token 'integer'
%left '('
%left ')'
%left '>'
%left '<'
%left '='
%left '<='
%left '<>'
%left '='
%left ':='
%left '+'
%left '-'
%left '*'
```

```
%left '/'
%%
programa      :'program' id ';' declaraciones bloque '.'   = { 1; }
declaraciones : 'var' declr_var                            = { 2; }
                |                                          = { 3; }
declr_var     : variables                                  = { 4; }
                |variables ';' declr_var                   = { 5; }
variables     : id ':' tipo                                = { 6; }
tipo          : 'integer'                                  = { 7; }
bloque        : procedure instrucciones                    = { 8; }
procedure  : 'procedure' id '('variables')'';'instrucciones = { 9; }
                |                                          = { 10; }
instrucciones : 'begin' lista_declr end                    = { 11; }
lista_declr   : declr                                      = { 12; }
                |declr';'lista_declr                       = { 13; }
declr         : while                                      = { 14; }
                | if                                       = { 15; }
                |asigna                                    = { 16; }
                |instrucciones                             = { 17; }
                |llama_proc                                = { 18; }
while         : wile expr                                  = { 19; }
wile          : 'while' expr 'do'                          = { 20; }
if            : fexpr then                                 = { 21; }
                | fexpr thenelse                           = { 22; }
fexpr         : 'if' expr                                  = { 23; }
then          : 'then' declr ';'                           = { 24; }
thenelse      : inicio declr                               = { 25; }
inicio        : 'then' declr 'else'                        = { 26; }
asigna        : id ':=' expr                               = { 27; }
llama_proc    : id '(' id  ')'                             = { 28; }
expr          : expr_ent                                   = { 29; }
                | expr_ent relacion  expr_ent              = { 30; }
expr_ent      : term                                       = { 31; }
                |term sumres expr_ent                      = { 32; }
term          : factor                                     = { 33; }
                |factor muldiv term                        = { 34; }
sumres        : '+'                                        = { 35; }
                |'-'                                       = { 36; }
muldiv        : '*'                                        = { 37; }
                |'/'                                       = { 38; }
factor        : id                                         = { 39; }
                | num                                      = { 40; }
                | '('expr')'                               = { 41; }
relacion      : '>'                                        = { 42; }
                |'<'                                       = { 43; }
                |'='                                       = { 44; }
                |'<>'                                      = { 45; }
                |'>='                                      = { 46; }
                |'<='                                      = { 47; }
%%
```


7. TRADUCCION DIRIGIDA POR SINTAXIS.

Prácticamente todos los compiladores modernos utilizan la traducción dirigida por sintaxis. en este capítulo se presenta en 'parsers' abajo-arriba.

ANTECEDENTES

La traducción dirigida por sintaxis es la acción de generar código en el momento de realizar el análisis de sintaxis o 'parsing'. Esto se hace en la expansión de producciones en un 'parser' arriba-abajo o en la reducción del 'handle' en el parser abajo-arriba.

YACC es un generador de 'parsers' que se estudió en el capítulo de análisis sintáctico, genera programas que reconocen si una cadena pertenece a un lenguaje particular y traduce la cadena cuando la analiza. un programa asi es un "traductor dirigido por sintaxis" (TDS), porque la traducción se realiza según se va efectuando el análisis de la cadena de entrada.

Otros nombres que se les dan a este tipo de programas son: compilador-compilador y sistemas traductor-descriptor. Los métodos de parsing LL y LR pueden ampliarse para convertirse en traductor dirigido por sintaxis. Este capitulo muestra como modificar el 'parser' LR para convertirlo en traductor dirigido por sintaxis.

El acercamiento fundamental a traductor dirigido por sintaxis es el mismo que el de LEX en donde código de acción en C era asociado con cada regla de reconocimiento de patrónes. Ese código se ejecuta cada vez que un patrón o 'token' es reconocido.

El mismo acercamiento se aplica en la especificación del 'parser' de tal manera que el código de acción se ejecute en el momento de la selección y expansión de una regla de producción en un 'parser' arriba-abajo y con la reducción de un 'handle' en un 'parser' abajo-arriba.

En un traductor dirigido por sintaxis LR al efectuarse una reducción, se realizan las siguientes acciones:

- llenar la tabla de simbolos,
- realizar análisis semántico,
- crear las estructuras de datos en tiempo de corrida.
- generar código intermedio u objeto.

Cuando la parte del 'parser' de un traductor dirigido por sintaxis termina el análisis de la cadena de entrada, se termina también la traducción o generación de código.

TRADUCCION ABAJO-ARRIBA DIRIGIDA POR SINTAXIS

Para crear un traductor dirigido por sintaxis abajo-arriba se añade código de acción a la gramática.

El código de acción se coloca o asocia a una producción, en vez de permitir a los segmentos ser colocados en cualquier lugar en la parte derecha.

Ejemplo, Se desea producir un traductor que evalúe expresiones aritméticas donde la traducción es su valor; así para la expresión:

```
(4+2) * 5 / 2
```

que se traducirá a 15 al ejecutar el código de acción durante el análisis. la gramática que reconoce esta cadena es:

```
S-> E $
E-> T
E-> E O T
T-> F
T-> T M F
F-> ( E )
F-> ENTERO
O-> +
O-> -
M-> *
M-> /
```

Esta gramática no contiene código de acción pero al agregarsele se puede realizar la traducción dirigida por sintaxis con el auxilio de dos 'stacks' llamados stack_operando y stack_operador. donde se van almacenando los operadores y los operandos y al momento de realizar la reducción se realiza la operación aritmética. Una gramática que tiene código de acción se denomina atribuida o con atributos

Para el ejemplo se produce código intermedio de una máquina de 'stack', el término 'lhs' corresponde al valor del lado izquierdo de la expresión y 'rhs' al lado derecho. Así una expresión se representa como:


```
                lhs  operador  rhs
```

La gramática con código de acción es:

```
S-> { inicio() } E $  { fin() }

E->  T

     | E O T   {rhs=pop_operando();
                lhs=pop_operando();
                switch (pop_operador())  {
                   case _MAS: push_operando(lhs+rhs);
                              break;
                   case _MENOS:push_operando(lhs-rhs);
                              break;
                   default: error ("error en stack",FIN);
                     }
                  }

T-> F

     | T M F   {rhs=pop_operando();
                lhs=pop_operando();
                switch (pop_operador())  {
                    case _MULTP: push_operando(lhs*rhs);
                                 break;
                    case _DIV:   push_operando(lhs/rhs);
                                 break;
                    default:
                        error ("error en stack", FIN);
                  }
                 }

F-> ( E ) | ENTERO    {push_operando($1);}

O-> + {push_operador(_MAS);}
    | - {push_operador(_MENOS);}

M-> * {push_operador(_MULTP);}
    | / {push_operador(_DIV);}
```

código de acción es:

```
#define MAX_OPERANDO 100
#define MAX_OPERADOR 100  /* número máximo elementos */

int stack_operando[MAX_OPERANDO];
int stack_operador[MAX_OPERADOR];
int ptr_operando, ptr_operador;
int _atributo;

push_operando(t)
 int t;
  {
```

```
    if (++ptr_operando == MAX_OPERANDO)
          error ("stack overflow.", FIN);
    stack_operando[ptr_operando] = t;
  }

pop_operando()
  {
    if (ptr_operando < 0)
          error ("stack underflow.", FIN);
    return (stack_operando[ptr_operando--]);
  }

push_operador(t)
  int t;
  {
    if (++ptr_operador == MAX_OPERADOR)
          error ("stack overflow.", FIN);
    stack_operador[ptr_operador] = t;
  }

pop_operador()
  {
    if (ptr_operador < 0)
         error ("stack underflow.", FIN);
    return (stack_operador[ptr_operador--]);
  }

inicio()
  {
    ptr_operando = ptr_operador = -1;
  }

fin()
  {
    printf ("%d\n", pop_operando());
  }
```

Dos funciones especiales: inicio() y fin() se invocan al principio y final de la traducción. Para permitir la inicialización e impresión del resultado. En el ejemplo las dos variables que indican el tamaño de los 'stacks' son inicializadas a 1 cuando la función inicio es llamada, y el valor de la expresión final es impreso por la función fin.

Cada símbolo en el árbol de 'parsing' ya sea terminal o no terminal, tiene asociado una variable llamada _atributo cuyo tipo variará en cada traductor. La variable _atributo contiene información acerca de ese símbolo, la cual puede ser referenciada en el código de acción.

El analizador lexicográfico o yylex además de regresar el tipo de 'token' tal como _MULTP o _ENTERO, debe asignar un valor a atributo cuando le regrese el control al 'parser'. El 'parser' asociará ese atributo con el símbolo sentencial que representa al 'token'.

Esta variable _atributo puede ser inicializada y referenciada usando la notacion "$" en el código de acción . $1 es el nombre del _atributo del primer símbolo en la parte derecha de una producción.

$2 es el nombre del atributo del segundo símbolo y asi en adelante. $$ se usa para el atributo del no terminal en la parte izquierda de la regla. El _atributo de los terminales se asigna por el 'scanner' y se referencia en el código de acción, mientras que el _atributo de los no terminales debe ser asignado y referenciado en el código de acción.

En la regla F -> ENTERO, $1 se refiere al atributo de ENTERO, el primer símbolo en la parte derecha de la regla. La declaración de _atributo en el código de acción indica para este traductor un atributo de tipo int. Pero este puede ser de cualquier tipo incluyendo a struct, lo que permite que se asocien atributos individuales a cada nodo haciendo a cada nodo un campo de atributo.

En el ejemplo a _atributo se le asigna el valor de un token ENTERO, esto lo hace el scanner (lo que no se muestra explicitamente aquí). Este valor se almacena en la lista de operandos para referencia posterior después que el resto de la expresión ha sido analizado. El 'parser', almacena el valor entero, porque el 'parser', y no el 'scanner', tiene el contexto en el cual aparece el entero. En un ejemplo mas complejo, los enteros pueden aparecer en varios lugares requiriendo acciones para cada uno.

Usando el _atributo de cada nodo para guardar el valor parcial que produce la expresión:

```
E'-> E     { printf ("%d\n", $1);}

E-> T      {$$ = $1;}

  | E  O  T
       {switch ($2)  {
        case _MAS:  $$ = $1+$3;
                    break;
        case _MENOS: $$ = $1-$3;
                     break;
        default: error ("error en stack", FIN);
                 }
       }

T-> F    {$$ = $1;}

  | T  M  F
       {switch ($2)  {
          case _MULTP: $$ = 1*$3;
                       break;
           case _DIV:  $$ = $1/$3;
```

```
                                    break;
                            default: error ("error en stack", FIN);
                            }
                        }

F-> ( E )       {$$ = $2;}
        | ENTERO    {$$ = $1;}

O-> +       {$$ = _MAS;}
        | -     {$$ = _MENOS;}

M-> *       {$$ = _MULTP;}
        | /     {$$ = _DIV;}
```

_atributo es un entero. El código de acción también es mucho más simple, mostrando el hecho de que la parte derecha de una producción sencilla genera a otra expresión, esto es, un operando seguido de un operador, y un operando

Así como se cumplía para el TDS arriba-abajo, el permitir expresiones arbitrarias de atributos en el código de acción de un TDS abajo-arriba puede, en el peor de los casos, forzar al traductor a almacenar todo el árbol de sintaxis.

El no permitir atributos en la parte derecha de una producción a los cuales les haya sido asignado un valor en el código de acción obliga a la información a que pase solamente hacia arriba del árbol. Esto garantiza que la parte baja del árbol pueda ser eliminada conforme el árbol se construye y solo la forma sentencial necesita ser almacenada.

Los traductores arriba-abajo pueden parecer más poderosos porque permiten a segmentos de acción aparecer en cualquier lugar en la parte derecha mientras que los traductores abajo-arriba solo permiten un segmento de acción por producción.

De hecho, es trivial extender una gramática atribuida abajo-arriba para permitir a los segmentos de acción ser ejecutados en lugares arbitrarios en el reconocimiento del 'handle'. Para obligar al código {..} a ser ejecutado entre c y d en la expresión.

b-> w c d z

se reescribe esta producción por:

b-> w c E d z

E-> {..}

Con el nuevo no terminal E. Cuando se reduzca la producción nula E:, el código de acción {..} se ejecutará.

Finalmente si utilizamos el programa para realizar el

análisis sintáctico del capítulo 6 y le agregamos código de acción para evaluar las expresiones de esta gramática obtendremos lo siguiente:

```
/* Facultad de Ingeniería. UNAM                    */
/* Apuntes de compiladores                         */
/* prof: A. Jimenez H.                             */
/* ejemplo de TDS LR realizado manualmente         */
/*                con código de acción             */
/*                                                 */
/* la gramática es:

      0. S-> E $
      1. E-> E O T
      2. E-> T
      3. T-> T M F
      4. T-> F
      5. F-> ( E )
      6. F-> ENTERO
      7. O-> +
      8. M-> *
   */

#include <stdio.h>
#define EDO 15      /* número de estados                */
#define SIM 8       /* número de símbolos de entrada */
#define SGR 12      /* símbolos gramaticales  */
#define RED 9       /* número de reducciones  */
#define MAX_OPERANDO 100
#define MAX_OPERADOR 100  /* número máximo elementos */

int ch, j=0, lhs, rhs;
char numero[5];
int stack_operando[MAX_OPERANDO];
int stack_operador[MAX_OPERADOR];
int ptr_operando, ptr_operador;

char *malloc();
int *p1, *tos;

/* acciones en el stack   */
/* tokens de entrada ... i - ^ ( ) ; e   */

int mat_efe[EDO][SIM] = {9,0,0,9,0,0,0,0, /* edo. inicial */
                         0,9,0,0,0,9,0,0,
                         0,2,0,0,2,2,0,0,
                         0,4,9,0,4,4,0,0,
                         0,6,6,0,6,6,0,0,
                         9,0,0,9,0,0,0,0,
                         0,0,0,0,0,0,10,0,
                         9,0,0,9,0,0,0,0,
                         0,1,0,0,1,1,0,0,
                         0,9,0,0,9,0,0,0,
                         0,5,5,0,5,5,0,0,
```

```
                        9,0,0,9,0,0,0,0,
                        0,3,0,0,3,3,0,0,
                        7,0,0,7,0,0,0,0,
                        8,0,0,8,0,0,0,0};

/*  función GOTO */
/* símbolos gramaticales.i - ^ ( ) ; S E T F O P*/

int mat_gee[EDO][SGR] = {4,0,0,5,0,0,0,1,2,3,0,0,/* edo inicial */
                        0,13,0,0,0,6,0,0,0,0,7,0,
                        0,0,0,0,0,0,0,0,0,0,0,0,
                        0,0,14,0,0,0,0,0,0,0,0,11,
                        0,0,0,0,0,0,0,0,0,0,0,0,
                        4,0,0,5,0,0,0,9,2,3,0,0,
                        0,0,0,0,0,0,0,0,0,0,0,0,
                        4,0,0,5,0,0,0,0,8,3,0,0,
                        0,0,0,0,0,0,0,0,0,0,0,0,
                        0,13,0,0,10,0,0,0,0,0,7,0,
                        0,0,0,0,0,0,0,0,0,0,0,0,
                        4,0,0,5,0,0,0,0,12,3,0,0,
                        0,0,0,0,0,0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,0,0,0,0,0};

/*  No terminales:  E,  T,  F   */

int llhs[RED] =  {11,7,7,8,8,9,9,10,11};

/*  número de elementos en producciones        */

int size[RED] = {1,3,1,3,1,3,1,1,1};


main()
{
  int valor1=0;
  int token,t;
  int edo = 0;
  int red, simbolo, i;
  int buffer;

  printf("\t calculadora\n");
  printf("expresiones validas terminan con ; \n\n");
  inicio();
  p1 = (int *) malloc(20 * sizeof(int));
  if(!p1) {
     error(0);
     return;
     }
  tos = p1;   /* tos es tope del stack */

      push(valor1);
      printf("dame cadena de entrada = ");
      simbolo = lee();
      while (1) {
```



```
edo = topv();
if (mat_efe[edo][simbolo] == 9) {       /* shift */
       t = mat_gee[edo][simbolo];
       printf("..shift edo: %d\n",t);
       push(t);
       imprime();
       if(simbolo==0)
              buffer = atoi(numero);
       simbolo = lee();
     }
else if (mat_efe[edo][simbolo] == 0) {  /* error */
      imprime();
      error(2);
      exit(1);
     }  /* end error */
else if ((red = mat_efe[edo][simbolo]) < 9)
     {  /* reducción */
          printf(" reduccion = %d \n",red);
        /* código acción */
          switch(red) {
              case 1: rhs=pop_operando();
                      lhs=pop_operando();
                      pop_operador();
                      push_operando(lhs+rhs);
                      break;
              case 3: rhs=pop_operando();
                      lhs=pop_operando();
                      pop_operador();
                      push_operando(lhs*rhs);
                      break;
              case 6: push_operando(buffer);
                      printf("\t\tpush_opando %d\n",
                             buffer);
                      break;
              case 7: push_operador('+');
                      printf("\t\tpush_opador %c\n",'+');
                      break;
              case 8: push_operador('*');
                      printf("\t\tpush_opador %c\n",'*');
                      break;
              default: break;
              }
          for(i=1;i <= size[red]; i++ )
                printf(" pop estado %d \n",pop());
          t = mat_gee[topv()][llhs[red]];
          if (t == 0) {
                       error(2);
                       exit(1);}
          push(t);
          printf("push estado %d\n",t);
          imprime();
          }
else if (mat_efe[edo][simbolo] == 10) {  /* aceptación */
          printf("cadena de entrada aceptada\n");
/* imprime resultado */
```

```
                    fin();
                    exit(0);
                    }
          }  /*  endwhile  */
}


#include "ctype.h"
/* rutina para leer cadena de entrada  */
lee()

{
while((ch = getchar()) ==  ' ' || ch == '\t' ) ;
   if (ch == '\n')
            return(6);
   else  if (isdigit(ch))
            { limpia();
              while(isdigit(ch)){
                        numero[j++] = ch;
                        ch = getchar(); }
              numero[j] = '\0';
              ungetc(ch, stdin);
              printf(".....lookahead = digito\n");
              return(0);}
   else  if (ch == '+')
            { printf(".....lookahead = +\n"); return(1); }
   else  if (ch == '*')
            { printf(".....lookahead = *\n"); return(2); }
   else  if (ch == '(')
            { printf(".....lookahead = (\n"); return(3); }
   else  if (ch == ')')
            { printf(".....lookahead = )\n"); return(4); }
   else  if (ch == ';')
            { printf(".....lookahead = ;\n"); return(5); }
   else  { printf(" caracter = %c",ch);
           error(1);
           return(7);
         }
}


push(i)
int i;
{
 p1++;        /*   Push estado       */
 if ((p1) == (tos + 20)) {
           printf("stack overflow\n");
           exit();
           }
 *p1 = i;
 }

pop()
{
 if(p1 == tos) {
```



```
           printf("stack underflow\n");
           exit();
           }
 p1--;      /*  remuevelo    */
 return *(p1+1);
}

topv()
{
 if(p1 == tos) {
           printf("stack underflow\n");
           exit();
           }
 return *(p1);
}

imprime()
{
 int cuenta, diff;
 diff = (int)(p1-tos);
 printf("\t stack = ");
 for( cuenta = 1 ; cuenta <= diff ; cuenta++)
           printf(" %d ",*(tos + cuenta));
 printf("\n");
 }

error(nerror)
int nerror;
{
 static char *err[] = {
           "*** falla inicializacion ***",
           "*** token no permitido ***",
           "*** error de sintaxis ***",
           "*** stack operando overflow ***",
           "*** stack operando underflow ***",
           "*** stack operador overflow ***",
           "*** stack operador underflow ***"};
printf("\n%s\n",err[nerror]);
}

limpia()
{
   numero[0] = '\0';
   j = 0;
 }

/* stack para operandos */

push_operando(t)
int t;
    {
     if (++ptr_operando == MAX_OPERANDO)
           { error(3);
             exit(1); }
     stack_operando[ptr_operando] = t;
```

```
    }

pop_operando()
    {
     if (ptr_operando < 0)
            {  error (4);
               exit(1); }
     return (stack_operando[ptr_operando--]);
      }


/* stack para operadores */

push_operador(t)
     int t;
      {
        if (++ptr_operador == MAX_OPERADOR)
               { error (5);
                 exit(1); }
        stack_operador[ptr_operador] = t;
      }

pop_operador()
       {
        if (ptr_operador < 0)
             { error (6);
               exit(1); }
        return (stack_operador[ptr_operador--]);
       }

inicio()
      {
       ptr_operando = ptr_operador = -1;
      }

fin()
      {
       printf ("resultado es %d\n", pop_operando());
      }
```


8. TABLA DE SIMBOLOS Y ORGANIZACION DE MEMORIA EN TIEMPO DE CORRIDA

Las tablas de símbolos permiten el análisis semántico y la generación de código, las dos funciones más importantes que realiza son buscar e insertar símbolos.

La organización de memoria define la manera de implementar las variables de trabajo, la cantidad de bytes que se asignan a cada tipo de datos, la manera de accesarlos, donde almacenarlos y su tiempo de vida.

ORGANIZACION DE TABLA DE SIMBOLOS

Un lenguaje estructurado es aquel en el que un bloque de instrucciones puede contener otros bloques en los que aparecen variables locales.

Los atributos o valores de la tabla de símbolos son nombre de las variables, tipo, localización en bloque, clase y dimensión son encontrados explícitamente en la parte de las declaraciones de un programa e implícitamente al leer y analizar el programa.

ejemplo. Describa la tabla de símbolos para el siguiente programa:

```
        x,y    : real;
        nombre: string[80];

    Procedure Uno(i:integer)
        Begin
        x : integer;
        ....
        ....
```

```
    End;

Procedure Dos(j:integer)
    Begin
    ....
    ....
    End;

Begin
....
....
End.
```

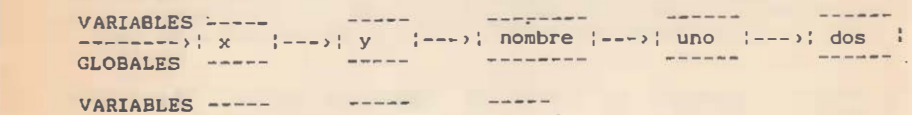
Una opción para la tabla de símbolos es:

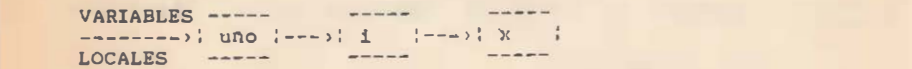
NO IND	NOMBRE	BLOQUE	CLASE	TIPO	LONGITUD BYTES
1	x	_principal	variable	real	4
2	y	_principal	variable	real	4
3	nombre	_principal	variable	cadena	80
4	Uno	_uno		procedure	
5	i	_uno	parametro	entero	2
6	x	_uno	variable	entero	2
7	Dos	_dos		procedure	
8	j	_dos	parametro	entero	2

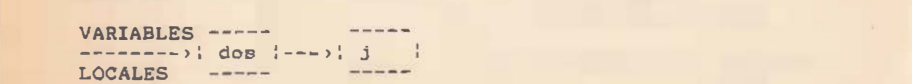
TABLAS DE SIMBOLOS TIPO 'STACK'

En este tipo de tabla los registros que contienen los atributos de las caracteristicas de las variables se acumulan en un 'stack' según se van encontrando en el programa.

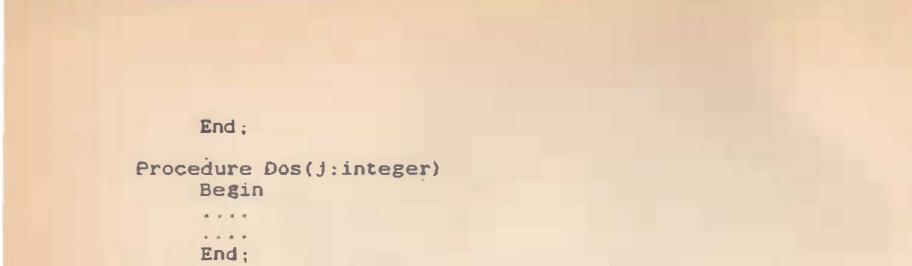
ejemplo. Escriba una estructura para la tabla de símbolos del ejemplo anterior.







ejemplo. Se escribe una tabla de símbolos para un lenguaje estructurado además de funciones para su manejo. la estructura de la tabla de símbolos es la siguiente:




```
/*  APUNTES DE COMPILADORES                                  */
/*                  rutinas de                               */
/*          manejo de tabla de simbolos                      */

/* definición de la estructura de tabla de símbolos */

struct tabla {
       char *t_nombre;     /*  nombre del id  */
       int t_tipo;         /* tipo del id     */
       int t_nivel;        /* nivel del bloque */
       struct tabla *t_link;
       int t_offset;       /* asignación en memoria */
       struct tabla *s_otro;  /* siguiente símbolo */
       };

#define t_plist t_link

/*   t_tipo    */

#define UDEF  0   /* no definido  */
#define PROC  1   /* procedure    */
#define UPROC 2   /* procedure no definido */
#define VAR   3   /* variable               */
#define PARM  4   /* parámetro              */


/*  funciones para manejo de tabla de símbolos  */

struct tabla *link_parm();    /* encadenar parámetros */
struct tabla *t_busca();      /* busca símbolo        */
struct tabla *crea_parm();    /* declara parámetros   */
struct tabla *crea_var();     /* define variables     */
struct tabla *crea_proc();    /* define procedures    */

/*  funciones de biblioteca de C  */

char *strsave();     /* salva cadenas  */
char *calloc();      /* obtiene memoria  */


/* definición de tabla de simbolos        */

static struct tabla
              tabla,       /* elemento referencia */
              *t_gbl;      /* parte baja stack    */
```

```
#define  t_lcl (&tabla)     /* tope stack          */


                            /* variable global,diferente */
                            /* a la del struct           */
static int nivel =0;        /* nivel de anidamiento de
                               de bloque                 */

/* agrega variables al tope del stack */

static struct tabla *t_crea(nombre)
       register char *nombre;
{
  register struct tabla *nuevo_elem = ( struct tabla *)
                  calloc(1, sizeof(struct tabla));

    if (nuevo_elem)
    {
       nuevo_elem->s_otro = t_lcl->s_otro;
       t_lcl-> s_otro = nuevo_elem;
       nuevo_elem ->t_nombre = strsave(nombre);
       nuevo_elem->t_tipo=UDEF;
       nuevo_elem->t_nivel = 0;
       return nuevo_elem;
    }
    error("no hay mas espacio");
}

/* si se encuentra un nombre de procedure o     */
/* variable global muevela al grupo de globales */

static t_mueve(simbolo)
       register struct tabla *simbolo;
{
     register struct tabla *ptr;
     /*   encuentra simbolo en tabla    */

    for(ptr = t_lcl; ptr->s_otro != simbolo; ptr = ptr->s_otro)
            if (!ptr->s_otro)
                    error("t_mueve);

    /* desligalo de la tabla */

    ptr->s_otro = simbolo->s_otro;

    /* ligalo al final de las globales */

    t_gbl->s_otro = simbolo;
    t_gbl = simbolo;
    t_gbl->s_otro = (struct tabla *) 0;
}

/* inicializa tabla de simbolos, el primer nombre  */
/* de un programa es el id después de program      */
```



```
init()
{
   ++nivel;
   t_gbl = t_crea("principal");
   t_gbl->t_tipo = UDEF;
}

/* localiza un id en la tabla de simbolos por nombre  */

struct tabla *t_busca(nombre)
       char *nombre;
{      register struct tabla *ptr;

       for(ptr = t_lcl->s_otro; ptr; ptr = ptr->s_otro)
             if(!ptr->t_nombre)
                  error("t_busca");

             else
                  if(strcmp(ptr->t_nombre, nombre) == 0)
                                return ptr;
       return (struct tabla *) 0;
}


/* liga parámetros en la tabla de símbolos    */

struct tabla *link_parm(simbolo, otro)
       register struct tabla *simbolo, *s_otro;
{
       switch (simbolo->t_tipo) {
       case PARM:
                error(parametro duplicado %s, simbolo->t_nombre)
                return otro;
       case PROC:
       case UPROC:
       case VAR:
                simbolo = t_crea(simbolo->t_nombre);
       case UDEF:
                break;
       default:
                error("link_parm");
       }
       simbolo->t_tipo = PARM;
       simbolo->t_nivel = nivel;
       simbolo->t_plist = otro;
       return simbolo;
}

/* define un parámetro en la tabla de símbolos */

struct tabla *crea_parm(simbolo)
       register struct tabla *simbolo;
{
       switch (simbolo->t_tipo) {
       case VAR:
```

```
                if(simbolo->t_nivel == 2)
                { error(parametro %s duplicado.
                   simbolo->t_nombre);
                   return simbolo;
                }
        case UDEF:
        case PROC:
        case UPROC:
                error("%s no es parametro", simbolo->t_nombre);
                simbolo = t_crea(simbolo->t_nombre);
        case PARM:
                break;
        default:
                error("crea_parm");
        }
        simbolo->t_tipo = VAR;
        simbolo->t_nivel = nivel;
        return simbolo;
}

/* define una variable en la tabla de simbolos */

struct tabla *crea_var(simbolo)
        register struct tabla *simbolo;
{
        switch (simbolo->t_tipo) {
        case VAR:
        case PROC:
        case UPROC:
                if(simbolo->t_nivel == nivel
                  || simbolo->t_nivel == 2 && nivel == 3)
                     error(nombre %s duplicado. simbolo->t_nombre);
                simbolo = t_crea(simbolo->t_nombre);
        case UDEF:
                break;
        case PARM:
                error("parametro no esperado %s", simbolo->t_nombre);
                break;
        default:
                error("crea_var");
        }
        simbolo->t_tipo = VAR;
        simbolo->t_nivel = nivel;
        return simbolo;
}

/* define un procedure en la tabla de simbolos */

struct tabla *crea_proc(simbolo)
        register struct tabla *simbolo;
{
        switch (simbolo->t_tipo) {
        case UPROC:
        case UDEF:
                break;
```



```
        case VAR:
                error("nombre funcion %s igual al blobal.
                                simbolo->t_nombre);
                return simbolo;
        case PROC:
                error("funcion duplicada %s", simbolo->t_nombre);
                return simbolo;
        default:
                error("crea_proc");
        }
        simbolo->t_tipo = PROC;
        simbolo->t_nivel = 1;
        return simbolo;
}

/* define los parámetros que no tienen tipo */

int parm_default(simbolo)
        register struct tabla *simbolo;
{
     register int cuenta = 0;

     while(simbolo)
     {      ++cuenta;
            if(simbolo->t_tipo == PARM)
                    simbolo->t_tipo = VAR;
            simbolo = simbolo->t_plist;
     }
     return cuenta;
}

/*  borra las variables locales que ya no se necesitan */

borra_loc()
{       register struct tabla *ptr;

        for(ptr = t_lcl->s_otro;
            ptr &&
            (ptr->t_nivel >= nivel || ptr->t_nivel == 0);
            ptr = t_lcl->s_otro)
        {
            if(!ptr->t_nombre)
                    error("borra_loc nombre nulo");

            if(ptr->t_tipo == UPROC)
                    error(" funcion no definida %s",
                            ptr->t_nombre);
            cfree(ptr->t_nombre);
            t_lcl->s_otro = ptr->s_otro;
            cfree(ptr);
        }
        --nivel;
```

```
/* verifica si el identificador que paso como parámetro  */
/* puede aparecer como variable dentro del procedure     */

verifica_var(simbolo)
      register struct tabla *simbolo;

{
   switch (simbolo->t_tipo){
      case UDEF:
               error("variable no declarada %s",
                              simbolo->t_nombre)
               break;
      case PARM:
               error("parametro no esperado %s",
                              simbolo->t_nombre);
               break;
      case PROC:
      case UPROC:
               error("procedure %s es utilizado como
                              variable", simbolo->t_nombre);
      case VAR:
               return;
      default:
               error("verifica_var");
      }
      simbolo->t_tipo = VAR;
      simbolo->t_nivel = nivel;
}
```

ORGANIZACION DE MEMORIA EN TIEMPO DE CORRIDA

En lenguaje ensamblador o de máquina una variable es simplemente una localidad de memoria, en lenguaje de alto nivel este nombre o identificador tiene asociado un tipo, alcance o gama de visibilidad y tiempo de vida.

IDENTIFICADORES ASOCIADOS A VARIABLES Y PROCEDIMIENTOS

Si se considera a un programa formado por rutinas o procedimientos, a la ejecución de una rutina o procedimiento se le llama activación del procedimiento. A los procedimientos que regresan un valor se les llama funciones.

La definición de un procedimiento es una declaración que asocia un identificador con un grupo de instrucciones. El identificador es el nombre del procedimiento y el grupo de instrucciones el cuerpo del procedimiento.

Cuando el nombre de un procedimiento aparece en un programa se dice que ese procedimiento esta siendo invocado o llamado.

IDENTIFICADORES ASOCIADOS A PARAMETROS

Los identificadores asociados a los procedimientos con el nombre de parámetros formales se utilizan para pasar o recibir información de este procedimiento, y normalmente aparecen entre paréntesis en la definición del procedimiento. En el momento de la invocación de ese procedimiento existen otros parámetros a los que se llama actuales y son los identificadores con los que se llama al procedimiento.

TIEMPO DE VIDA DE UN PROCEDIMIENTO

La vida de un procedimiento es el tiempo que dura la ejecución de éste desde su inicio hasta su terminación. Si X y Y son procedimientos y X se activa y a su vez activa a Y entonces el control del programa debe de terminar primero por Y y después por X.

Para llevar el control de los procedimientos que están activos, se utiliza un 'stack'. Cuando un procedimiento se activa se realiza un movimiento de 'push' en el 'stack' y cuando este termina se realiza un 'pop'.

ALCANCE DE VARIABLES. GLOBAL Y LOCAL

Una declaración en un lenguaje es una regla sintáctica que asocia información con un nombre. El alcance de una variable es la parte de un programa o procedimiento donde esta es válida. Así la ocurrencia de un nombre dentro de un procedimiento se dice que es local a este procedimiento si se declaró dentro de éste. De otra manera es no local o global.

TIPOS DE VARIABLES

Asociados al nombre en una declaración existen dos conceptos: medio ambiente y estado. El medio ambiente se refiera a una función que asocia un nombre en un tipo específico de localidad de memoria, como el tipo entero, real, booleano, etc. El término estado se refiere a la acción de colocar un valor en una determinada localidad de memoria. Así una instrucción como la asignación cambia el estado pero no el medio ambiente del nombre.

Una variable local en un procedimiento se asocia a diferentes localidades de memoria en cada activación.

MODELOS DE ASIGNACION DE MEMORIA

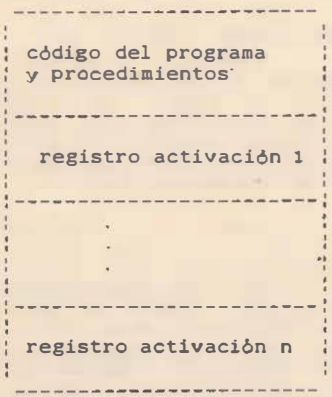
Existen tres tipos de almacenamiento en memoria:

- estàtico,
- con 'stack'
- con 'Heap'.

Para los tres tipos de áreas de almacenamiento de datos el manejo de memoria es diferente: en el área de datos estáticos las direcciones se calculan para todos los datos al compilar y después quedan fijas, para el 'stack' se manejan los registros de activación al compilar creándose o destruyendose al correr el programa, y para el 'heap' los registros de activación se crean y manejan al correr el programa.

ASIGNACION DE MEMORIA ESTATICA

En este tipo de asignación de memoria esta se asigna definitivamente en el momento de compilar, asi cada vez que un procedimiento o subrutina se activa las direcciones son las mismas, por lo que el valor de las variables no cambia aún cuando el procedimiento no tenga control. La posicion de los registros de activación son fijos. En este modelo no se permite la recursión en procedimientos y los datos no se pueden crear dinàmicamente. Un ejemplo es FORTRAN.



ASIGNACION DE MEMORIA POR MEDIO DE 'STACK'

Aqui los registros de activación se asignan al 'stack' cuando los procedimientos empiezan o se activan. Las direcciones de las variables locales se asignan en cada activación al hacer el 'push' del registro de activación. Estas se borran o desaparecen cada vez que el procedimiento termina o se hace un 'pop'.

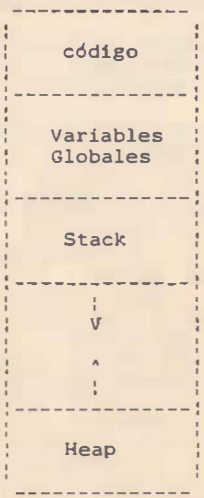
Una secuencia de activación o llamada se inicia al evaluar

los parámetros actuales del procedimiento, guardar la dirección de regreso, registros y estado del procesador, inicializar variables, activar 'stack' e iniciar ejecución del procedimiento.

Una secuencia de regreso coloca el valor de regreso restaura el valor del stack antiguo, los registros y status del procesador y regresa el control al procedimiento anterior.

El almacenamiento se divide en:

1. Código.
2. Datos estáticos.
3. 'Stack' de control.



REGISTROS DE ACTIVACION

Un registro de activación o 'frame' es un bloque contiguo de datos que contiene toda la información que se necesita para ejecutar un procedimiento. Este registro se almacena (push) en el 'stack' cuando se activa el procedimiento correspondiente y se saca (pop) cuando se retorna de este.

En general se maneja la siguiente información:

1.- Valores temporales. Son valores que normalmente aparecen en la evaluación de expresiones.

2.- Datos locales. Contienen la información de variables locales.

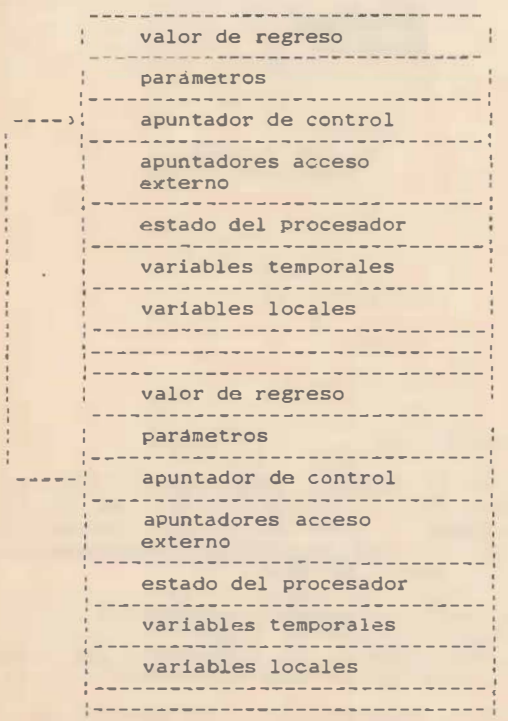
3.- Estado del procesador anterior a la llamada. Contiene información del contador de programa (PC) y los registros del procesador antes de haber llamado al procedimiento.

4.- Apuntadores de acceso externo. Contiene información acerca de datos que se encuentran en otros archivos de activación.

5.- Apuntador control. Apunta al registro de activación del programa que llamó.

6.- Parámetros actuales (opcional). Contiene los parámetros o variables de paso utilizados por el procedimiento que llama, aunque normalmente se pasan en registros.

7.- Valor de regreso de la función (opcional). Contiene el valor de retorno a el procedimiento que llama, aunque normalmente se regresa en registro.



Ejemplo. Rutina para manejar la asignación del offset de una variable en el registro de activación

```
int offset_gbl = 1,
    offset_lcl = 0,
    long_max;

/* asigna la localización en la memoria de stack  */
/* o registro de activacion a la variable simbolo */
/* asignando su offset correspondiente            */

asgmem_var(simbolo)
      register struct tabla * simbolo;

{     extern struct tabla *crea_var();

      simbolo = crea_var(simbolo);

      /* si no es parámetro entonces es variable local y
         se le asigna un offset  */

      switch(simbolo->t_nivel) {  /* variable local  */
      default: simbolo->t_offset = offset_lcl++;

      case 2:      /* parámetro */
             break;
      case 1:      /* variable global  */
             simbolo->t_offset = offset_gbl++;
             break;
      case 0:
             error("asgmem_var");
```

ejemplo. La asignación de memoria para los procesadores MC68000 y NS3200 puede ser:

ASIGNACION DE MEMORIA 'HEAP'

Aquí la memoria se divide y asigna a los archivos de activación cuando se necesita en el tiempo de corrida. Los archivos se pueden desactivar en cualquier orden por lo que el heap puede quedar dividido en áreas actuales y libres. Aquí el valor de las variables locales se mantiene aún cuando una activación termine.

valor l valor r
x = y ;

CODIGO PARA REGISTROS DE ACTIVACION

Las variables en un registro de activación se accesan mediante un apuntador al tope del 'stack' y un offset. Al compilar los procedimientos se calcula el número de parámetros y si se regresa algún valor.

ALGORITMO PARA LLAMAR PROCEDIMIENTOS

El código en el programa que llama:

1. Se evaluan los argumentos
2. Asigna memoria para el registro de activación que consta de parámetros, valor de regreso y apuntadores para el ligador.
3. Almacena el antiguo valor de tope del 'stack' en el registro de activación
4. Se coloca el tope del 'stack' apuntando al tope del nuevo registro de activación.
5. Se transfiere control al procedimiento llamado.

El código al entrar el procedimiento llamado:

1. Se guardan los registros de trabajo y de estado en el área de estado del procesador anterior a la llamada.
2. Se asigna memoria para lo que falta del registro de activación
3. Se inicializan las variables locales
4. Se inicia ejecución

El código al regresar del procedimiento llamado:

1. Guardar el valor del regreso si existe en el registro de activación.
2. Utilizando el apuntador de control se restablece el valor del 'stack' anterior.
3. Restablece los valores anteriores de registros y estado.
4. Salta a la dirección de regreso.

Ejemplo. Programa en C que llama a una rutina en ensamblador para limpiar la pantalla.

```
;
; rutina en ensamblador 8088 para limpiar la pantalla
; y que puede ser ligado a programas escritos en
;
; lenguaje de alto nivel.
; se demuestra el manejo del registro de activacion
;
```

```
_TEXT   SEGMENT  BYTE PUBLIC 'CODE'
        ASSUME CS:_TEXT
        PUBLIC _LIMPIA
_LIMPIA PROC NEAR
                        ; salva apuntador a registro de
INICIO: PUSH  BP        ; activación del procedure que llamo
        MOV   BP, SP    ; guarda apuntador frame en BP
        SUB   SP,8      ; espacio en frame para valor regreso
        PUSH  DI
        PUSH  SI
        PUSH  AX
        PUSH  BX
        PUSH  CX
        PUSH  DX

        MOV   AH, 6     ; función para limpiar pantalla
        MOV   AL, 0
        MOV   CX, 0
        MOV   DH, 23
        MOV   DL, 79
        MOV   BH, 7
        INT   10H

        POP   DX
        POP   CX
        POP   BX
        POP   AX
        POP   SI
        POP   DI
        MOV   SP, BP
        POP   BP

        RET

_LIMPIA ENDP
_TEXT   ENDS
        END
```

```
/* el programa en C que llama a la rutina anterior es: */

main()
      {
      LIMPIA();
      }
```

ASIGNACION DE MEMORIA 'HEAP'

Aqui la memoria se divide y asigna a los archivos de activación cuando se necesita en el tiempo de corrida. Los archivos se pueden desactivar en cualquier orden por lo que el heap puede quedar dividido en áreas actuales y libres. Aqui el valor de las variables locales se mantiene aún cuando una activación termine.


```
valor l     valor r
   x = y ;
```

PASO DE PARAMETROS

La manera de comunicar dos procedimientos es a través de variables globales y paso de parámetros. Para el paso de parámetros existen varios métodos:

- llamada por valor
- llamada por referencia
- valor-resultado
- llamada por nombre

LLAMADA POR VALOR

En este tipo de llamada el valor de una variable o valor 'r' se pasa al otro procedimiento, aqui las operaciones en los parámetros no afectan los valores en el registro de activación del procedimiento que llama.

LLAMADA POR REFERENCIA

En este tipo de llamada la dirección de la variable o valor 'l' se pasa a la función que se llama, si el parámetro es un valor o expresión que no tiene valor 'l' entonces se le asigna.

VALOR RESULTADO

Es una mezcla de las dos anteriores, un ejemplo es FORTRAN

LLAMADA POR NOMBRE

Aqui el procedimiento se comporta como una macro y los parámetros actuales se sustituyen por los formales.

9. CODIGO INTERMEDIO Y OPTIMIZACION

Esta forma de representación ayuda a optimizar el código a causa de que esta forma de representación se genera especificamente para facilitar la optimización. Este modelo facilita la producción de compiladores transportables, al separar la generación de código de las otras fases de compilación.

Las ventajas de utilizar código intermedio son:

- la máquina objeto se abstrae a una máquina virtual, esto ayuda a separar operaciones de alto nivel de otras de bajo nivel.

- optimizaciones pueden realizarse en un nivel de representación medio. ganándose transportabilidad de las rutinas de optimización. Al ser más abstracta e uniforma las rutinas de optimización son más simples.

FORMAS DE CODIGO INTERMEDIO

Existen dos formas de representar código intermedio: lineal o de tres direcciones y en forma de árbol.

CODIGO DE TRES DIRECCIONES

Un programa traducido de esta forma consiste de declaraciones o seudoinstrucciones de máquina con tres operandos o direcciones, en este tipo de declaración solo una operación es permitida por declaración. así en el segmento de programa se obtendria:

```
if (x <= y+1)
     x = x-z;
z= f/z;
```


se obtendria el siguiente código.

```
     T1 := y + 1;
     IF x > T1 GOTO E1
     T2 := x - z;
     x := T2;
E1:  T3 := (float) z
     T4 := f/ F T3
     T5 := (int)T4
     z := T5
```

CUADRUPLE

Es una forma de código intermedio de tres direcciones donde el resultado de una operación se escribe dentro del código. Las variables temporales se tienen que introducir dentro de la tabla de símbolos.

ejemplo. el código para el segmento de programa del ejemplo anterior quedaria.

	operador	operando1	operando2	resultado
1	+	y	1	T1
2	if > goto	x	T1	(5)
3	-	x	z	T1
4	:=	T1		x
5	(float)	z		T2
6	/F	f	T2	T2
7	(int)	T2		T1
8	:=	T1		z

TRIPLES

Aquí el resultado de una operación está implícito y se representa por medio de una referencia a una declaración.
ejemplo. para el ejemplo anterior se generaria.

	operador	operando1	operando2
1	+	y	1
2	if > goto	x	(1)
3		(6)	
4	-	x	z
5	:=	(4)	
6	(float)	z	
7	/F	f	(6)
8	(int)	(7)	
9	:=	(8)	

CODIGO EN FORMA DE ARBOL

Esta forma de representación de código intermedio utiliza una estructura de árbol y utiliza comunmente una seudomáquina de 'stack'.

OPTIMIZACION

La función del optimizador de código es transformar un programa en otro equivalente, que sea más eficiente tanto en el tiempo que toma en ejecutarse y la cantidad de memoria espacio que utiliza.

Las optimizaciones màs comunes son:

- asignación de registros
- 'folding'
- identidades algebraicas
- reducción de 'costo' de operadores
- eliminación de expresiones redundantes
- movimiento de código

ASIGNACION DE REGISTROS

El manejo adecuado de registros identifica los operadores mas utilizados y los coloca en registros para acelerar su ejecución. Existen las siguientes limitaciones al asignar:

- número limitado de registros
- los operandos màs utilizados cambian durante la ejecución del programa.
- Para cada arquitectura de máquina existen limitaciones en el uso de registros
- no es posible determinar que operandos son los más utilizados dentro del programa.

Para manejar los registros se mantienen listas de los que están libres y de los que se están utilizando. Los registros se manejan por medio de rutinas que determinan los registros libres y los asignan a operadores y que los liberan si están ocupados.

FOLDING

El 'folding' es el evaluar durante el tiempo de compilación las operaciones conocidas. Asi si expresiones tales como '20 + 30' se encuentran, el compilador calcula el resultado y genera código con el resultado de la operación. Realizar esta operación al compilar detecta en algunos casos errores antes de la corrida, tales como 'overflow' y división entre cero.

IDENTIDADES ALGEBRAICAS

Se utilizan las expresiones algebraicas de identidad para reducir el código, tales como:

```
x + 0 = x
0 + x = x
x * 1 = x
1 * x = x
0 / x = 0
x - 0 = x
```

REDUCCION DE COSTO DE OPERADORES

Se reeemplaza un operador por otro más simple o menos complejo. Asi se toma ventaja de que operadores como la suma es más rápida que la multiplicación y esta más que la división.

ejemplo.

si se encuentra		se sustituye por
i * 2	=	i + i
i / 2	=	(int) (i * F 0.5)
x * F 2.0	=	F x + F x
x / F 2.0	=	x * F 0.5

donde F indica un número real

ELIMINACION DE EXPRESIONES COMUNES

Dos operaciones son comunes o redundantes si producen el mismo resultado, al optimizar se traduce o calcula una vez y la siguientes únicamente se referencia la traducción.

Las condiciones para poder reducir el código son:

- que existan expresiones redundantes
- que los valores que se utilizan en esas expresiones no cambien al referenciarlos otra vez.

ejemplo.

```
y = x * x + ( x / y )
z = x * x + ( x / y )
```

ALGORITMO PARA ELIMINAR EXPRESIONES COMUNES

Entrada: Bloque de instrucciones B donde B[i] es la iésima instrucción.

```
{
    IF ( O esta marcada en tabla como reemplazable por R )
    reemplaza referencia a O por R en B[i];
    IF ( operación en B[i] esta en tabla )
```

```
        { marca en tabla que referencias futuras a esta
          tupla son reemplazadas por la referencia anterior
         }
          borra B[i];
        }
    ELSE dame tupla en tabla;
}
```

BLOQUES BASICOS

Casi todas las técnicas de optimización dependen de dividir un programa en bloques básicos y producir una gráfica que muestre el flujo de control. Un bloque básico es un segmento de programa que solo tiene una entrada y una salida, por lo que un bloque básico contiene sólo código lineal.

10. GENERACION DE CODIGO

Esta fase genera el lenguaje ensamblador o de máquina de la computadora objeto. La generación de código se puede separar en las siguientes tareas:

- asignación de memoria,
- generación de código,
- asignación de registros
- optimización final 'peephole'.

El objetivo principal al generar código es la eficiencia y esta es muy dependiente de la máquina que se está utilizando. Una tendencia reciente en el desarrollo de arquitecturas de computadoras es la reducción del tamaño del conjunto de instrucciones y del número de posibles direccionamientos, aumentando el número de registros de procesador, estos cambios mejoran la calidad de código.

GENERACION DE CODIGO

La traducción de código intermedio a código en lenguaje ensamblador se puede realizar de una manera simple al generar código en el orden en que el lenguaje intermedio aparece. Normalmente existe una relación uno a uno entre el lenguaje intermedio y ensamblador.

Ejemplo: Generación de código para la instrucción aritmética:

```
w = x + y - z ;
```

que se traduciría en el siguiente código intermedio de tres direcciones:

```
t1 := x + y
w  := t1 - z
```

la generación de código ensamblador se realiza en una máquina de dos operandos y utilizando un registro:

```
mov  ax, x
add  ax, y
sub  ax, z
mov  w, ax
```

ASIGNACION DE MEMORIA

La sustitución de código se realiza cuidando la localización de las variables que pueden encontrarse en memoria o en registros.. Se debe tomar en cuenta también la politica de administración de memoria para determinada variable.

Cuando un programa se compila se divide normalmente en segmentos de texto, datos y bss. El código se localiza en el segmento de texto, los datos globales en el segmento de datos y los datos globales no inicializados en el segmento bss. Las variables locales van en el 'stack'.

ASIGNACION DE REGISTROS

Si la generación de código y asignación de registros están separadas, el generador de código produce lenguaje ensamblador virtual, por utilizar registros virtuales. Esto permite que el manejador de registros optimize la utilización de un número limitado de registros de una máquina real.

Un programa asignador de registros debe identificar los operandos más frecuentemente utilizados y colocarlos en los registros del procesador para minimizar el tiempo de acceso. Existen tres limitantes para esta asignación:

- número reducido de registros de procesador
- los operandos referenciados varían durante el transcurso de la operación del programa
- no es posible determinar que operandos serán los más utilizados.

ALGORITMO PARA ASIGNACION DE REGISTROS

Algoritmo para asignación de registros:

```
Funcion asigna_registro(op: operando;
                        registro : integer): integer;

  Begin
  If (no existe un registro libre r)
```



```
      r = libera(registro);
  marca tabla de simbolos indicando que op esta en r;
  asigna op a r;
  genera código para cargar op en r;
  return r;
  end;


Funcion libera(registro : integer): integer;

  Begin
  If (no existe el valor del registro r en memoria)
       genera codigo para salvar en memoria el valor actual
       del registro r;
  libera todos los simbolos asociados a r;
  actualiza tabla de simbolos para liberar a r;
  return r;
  end;
```

GENERACION DE CODIGO

La generación de código se realiza dependiendo de la operación que se realiza y las caracteristicas de la máquina. Así un algoritmo para generar código para una operación aritmética entre dos operandos (x,y) con resultado en z seria:

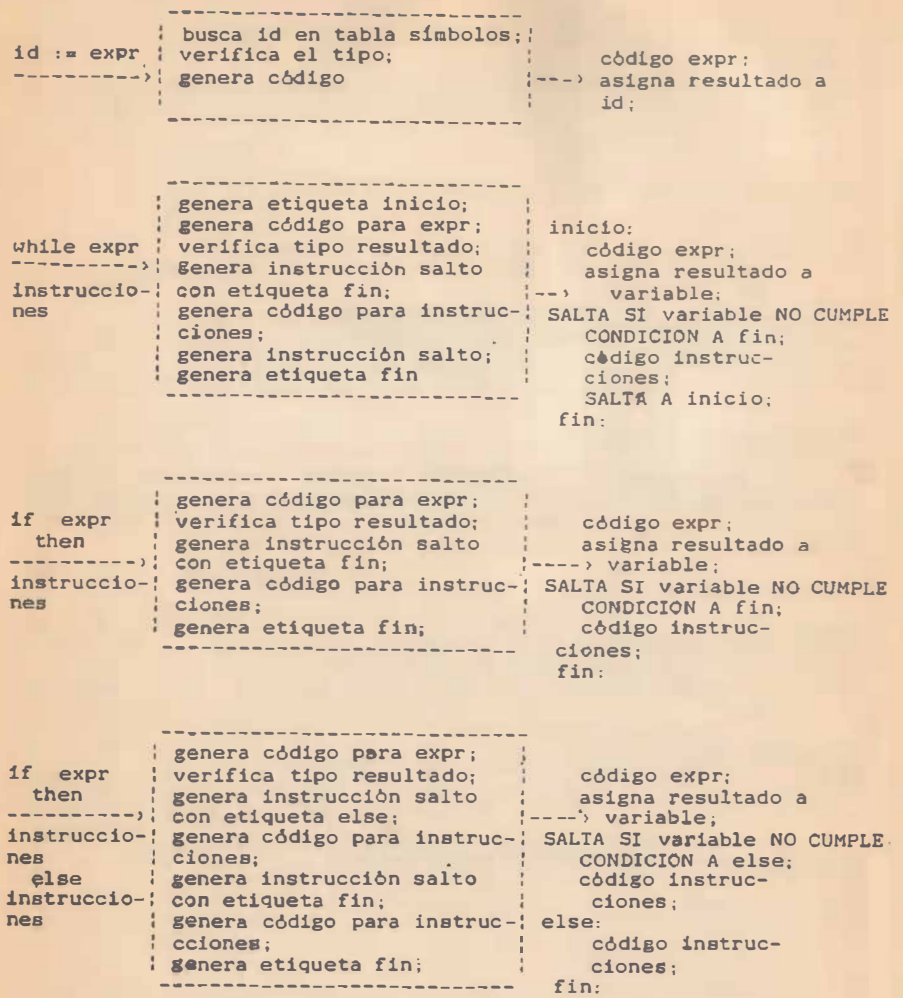
```
Procedure genera_codigo;

Begin
  if ( x no esta asignado a un registro r)
      if ( y se encuentra en el registro u)
          asigna_registro(x, tabla simbolos);
       else
          asigna_registro(x, NULO);
   genera código para z;
End;
```

El generador de código debe de tomar en cuenta los modos posibles de los operandos, que pueden ser:

- literal
- indexado
- indirecto
- registro 'libre'
- registro 'ocupado'

CODIGO PARA INSTRUCCIONES

Ejemplo: Se genera código para las estructuras de asignación, repetición (while) y deçisión (if-then-else).



Para generar código para estas estructuras utilizando traducción dirigida por sintaxis, éstas se deben de dividir para poder acomodar el código en el momento de realizar la reducción, así por ejemplo para la estructura IF-THEN e IF-THEN-ELSE podemos tener la siguiente gramática:


```
INSTR     -> IF | ASIGNA
IF        -> FEXPR  THEN | FEXPR  THENELSE
FEXPR     -> 'if' expr
THEN      -> 'then' INSTR ';'
THENELSE  -> INICIO  INSTR
INICIO    -> 'then' INSTR 'else'
ASIGNA    -> expr ':=' expr
```

La distribución de esta gramática permite incluir la generación código en las reducciones colocando las instrucciones de salto condicional y no condicional y etiquetas de destino en el lugar adecuado.

```
/*  formato en un traductor dirigido por sintaxis
    de la generación de código para la estructura
    if-then / if-then-else
    se utiliza un 'stack' para guardar nombres de
    etiquetas y después imprimirlas                  */

%term id end
%token 'if'
%token 'then'
%token 'else'
%left ':='
%%
instr     : if                              = {1; }
          | asigna                          = {2; }

if        : fexpr  then                     = {3; }
            /* pop etiqueta;
               imprime etiqueta;
               */

          | fexpr  thenelse                 = {4; }

fexpr     : 'if' expr                       = {5; }
            /* genera instrucción de
                 "SALTA_SI_ES_FALSO etiqueta";
               push etiqueta;
             */

then      : 'then' instr ';'                = {6; }

thenelse  : inicio  instr                   = {7; }
            /* pop etiqueta;
               imprime etiqueta;
               */
```

```
inicio      : 'then' instr 'else'          = {8; }

              /* genera instrucción de
                    "SALTO etiqueta";
                 push etiqueta;
                 pop etiqueta-1;
                 imprime etiqueta;
              */

asigna      : expr ':=' expr               = {9; }

              /* genera código asignacion */

expr        : 'id'                         = {10; }

              /* genera código  id */

%%
```

La generación de código se realiza imprimiendo directamente las instrucciones en lenguaje intermedio o ensamblador. Estas funciones normalmente constan de printf() o variantes de este.

Así por ejemplo para generar código para una instrucción que utiliza dos operandos y el segundo esta en un registro de activación podemos tener":

```
gen(op, mod1, mod2, val, comentario)
char *op;     /* código operación */
char *mod1;   /* operando1 */
char *mod2;   /* operando2 */
int val;      /* campo para offset de la segunda variable */
char *comentario;

{
  printf("\t%s\t%s,\t%s%d\t\t; %s\n", op, mod ,val, comentario);
}

/* para generar etiquetas        */

#define  ETIQ    "$%d"

static char  *formato_etiqueta(etiqueta)
int etiqueta;

{   static char     buffer[sizeof ETIQ + 2];

    sprintf(buffer, ETIQ, etiqueta);
    return buffer;
}


/* define localización del operando en memoria  */
```



```
/* es global, local o parametro?                                  */

char *gen_mod(simbolo)
struct tablasim *simbolo;

{
      switch (simbolo->s_modo)  {
            case 1:
                  return MOD_GLOBAL;
            case 2:
                  return MOD_PARAM;
            }
            return MOD_LOCAL;
}
/*  genera instrucciones de saltos                                */

int gen_jump(op, etiqueta, comentario)
char *op;       /* nemonico */
int etiqueta;   /* dirección de salto */
char *comentario;

{
     printf("\t%s\t%s\t\t; %s\n", op,
                formato_etiqueta(etiqueta),comentario);
     return etiqueta;
}
/* cuenta etiquetas              */

int otra_etiqueta()
{
       static int sig_etiqueta = 0;
       return ++sig_etiqueta;
}
/* genera llamadas a funciones                                    */

gen_jsr(simbolo,argumento)
        struct tablasim *simbolo; /* nombre función */
        char *argumento;

{
        verifica_parm(simbolo,cuenta);
        printf("\t%s\t%d,%s\n", JSR,
                          , simbolo->s_nombre);
              printf("PUSH \t%s",*argumento);
}
```

GENERACION DE CODIGO PARA EL PROCESADOR MC68000

Se analiza el procesador MC68000 este tiene un conjunto de instrucciones complejo, donde estas tienen longitudes y tiempos de ejecución desiguales.

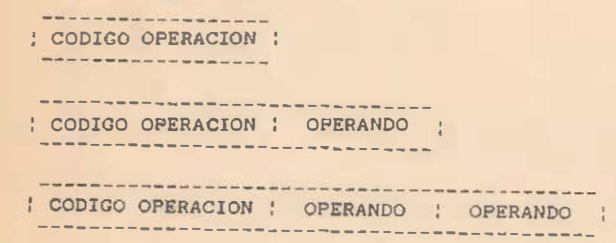
La reciente popularidad de procesadores llamados RISC con conjuntos de instrucciones simples y pequeños con tiempo de ejecución igual para todas las instrucciones permite una simplificación de los generadores de código.

Las características del procesador 68000 son:

FORMATO DE INSTRUCCIONES

	MC68000
longitudes instruccion	16,32,48,64,80
longitudes direcciones	16,32
no. dirs. x instruccion	1,2
direccionamiento ext.	Si
registros datos	D0-D7
registros direcciones	A0-A6, SP

FORMATO DE INSTRUCCIONES



MODOS DE DIRECCIONAMIENTO

	MC68000
registro	D[n]
registro direcciones	A[n]
registro indirecto	M[A[n]]
autoincremento	M[A[n]], INC[A[n]]
autodecremento	DEC[A[n]], M[A[n]]
directo	M[A[n]+ d]
indexado	M[A[n] + d + D[x]]
	M[A[n] + d + A[x]]
inmediato	M[PC]

MANEJO DE MEMORIA

El 68000 tiene una organización de manejo de memoria para tiempo de corrida orientada a apoyar lenguajes de alto nivel incluyendo 'stack' para manejo de registros de activación. En el 68000 se pueden utilizar tres de los siete registros de direcciones de propósito general para apuntar al campo de variables globales, el procedure más recientemente activado y el tope del 'stack'.

Para evaluación de expresiones y almacenamiento de resultados temporales se pueden utilizar los registros de datos, como se ve en la figura.

DESCRIPTORES DE MODO

MODO	REPRESENTACION
constante	valor
indirecto	dir, A
indexado	dir, D
registro dir	A
registro datos	D
salto con cond.	cc, salto T, salto J
tipo	type
llamada procedure	proc

ASIGNACIONES

El código que se genera para las asignaciones son las estructuras más simples

ejemplo. Genere código para las siguientes asignaciónes:

```
        x := y + z

MOVW    y, R0
ADDW    z, R0

        x:= 3 + 5

MOVW     8,  x

        x := r^.f

MOVW     f(A0), x

        a[i] := b[j]

MOVW     i, R0
LEA      a, A4
MOVW     j, R1
```

```
LEA     b, A3
MOVW    (A3, R1), (A4, R0)
```

```
x := y < z
```

```
MOVW    z, D0
CMPW    y, D0
SCS     D0
NEG     D0
MOVW    D0, x
```

DECLARACIONES Y LLAMADAS A PROCEDURES

Los parámetros para procedure se pasan por medio del 'stack' de registros de activación. El procesador debe depositar los parámetros de valores o direcciones en el tope del 'stack' antes de pasarle control al procedure. Los parámetros se direccionan relativos a la base local.

Ejemplo. Se genera código para la siguiente declaración de procedure.

```
PROCEDURE A(x,y: Integer; Var z:integer)
Var i,j: Integer;
Begin ......
```

Para el 68000 tenemos:

```
LINK    A6, 4
UNLK    A6
MOVE    (A7)+,  A4
ADDQ    #8,  A7
JMP     (A4)
```

Ejemplo. se genera código para una llamada al procedure anterior con los siguientes parámetros:

```
P(17, K+5, K)
```

```
MOVW    17, -(A7)
MOVW    k(A5), D0
ADDW    5, D0
MOVW    D0, -(A7)
PEA     k, (A5)
BSR     A
```

VARIABLES INDEXADAS

El procesador tiene instrucciones especiales para el rápido acceso de direcciones indexadas, incluyendo la validación de los límites de los arreglos, esto contribuye a la eficiencia en la

generación de código.

Ejemplo. para las siguientes declaraciones y asignaciones se genera código.

```
VAR  a:  ARRAY[0..99] OF INTEGER;
     b:  ARRAY[-10..+10] OF INTEGER;
     c:  ARRAY[0..99], [0..15] OF CHAR;
```

El código para la asignación:

```
u := a[9]
```

```
MOVW    a -18(A5), u(A5)
```

```
u := a[i]
```

```
MOVW    i(A5), D0
CHK     99, D0
ASLW    1, D0
LEA     a(A5),  A4
MOVW    0(A4, D0.W),  u(A5)
```

```
u := b[i];
```

```
MOVW    i(A5), D0
ADDW    10,  D0
CHK     20, D0
ASLW    1, D0
LEA     b(A5),  A4
MOVW    0(A4, D0.W),  u(A5)
```

```
u := c[9.9]
```

```
MOVB    c-450(A5), ch(A5)
```

EXPRESIONES ARITMETICAS

En el cálculo de expresiones aritméticas se pueden utilizar los registros de datos de manera similar a un 'stack'.

ejemplo: Genere código para la siguiente expresión:

```
( a + 10) - ((i + b * 5 + j * 2) / 4)
```

Utilizando los registros que se mencionaron en la parte de manejo de memoria y suponiendo que a y b son variables globales e i y j son locales, para el 68000 tenemos:

```
MOVW    a(A5), D0
ADDIW   10, D0          ; a + 10
MOVW    b(A5), D2
MULS    5, D2           ; 5 * b
ADDW    i(A6), D2       ; (5 * b) + i
```

```
MOVW    J(A6), D4
ASLW    1, D4
ADDW    D4, D2
EXTL    D2
DIVS    4, D2
SUBW    D2, D0          ; ((i + b * 5 + j * 2) / 4)
                        ; ( a + 10 ) -
                        ; ((i + b * 5 + j * 2) / 4)
```

EXPRESIONES BOOLEANAS

La definición de la semántica de las expresiones booleanas es inconsistente con su sintaxis. Esto se debe a que la sintaxis de las expresiones se define como asociativa por la izquierda, mientras que los operadores lógicos son asociativos por la derecha. Así por ejemplo la expresión x + y + z se entiende que es (x + y) + z y la expresión lógica p & q & r es equivalente a p & (q & r).

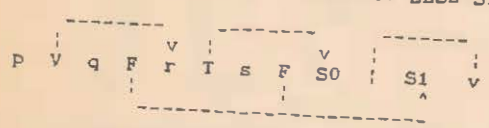
Los operadores lógicos se implementan mediante saltos condicionales. Puesto que las expresiones booleanas aparecen dentro de otras estructuras como while e if-then-else se deben de unificar los saltos condicionales dentro de las expresiones con los de las estructuras.

Las expresiones booleanas se pueden manejar mediante la condición del registro de estado o condición.

Ejemplo. Para la siguiente expresión se genera código:

IF ((x<y) OR (z<=x)) & ((u<v) OR (w<=u)) THEN x:=y ELSE u:=v

IF (p OR q) & (r OR s) THEN S0 ELSE S1



El código generado es:

```
        MOVW    X(A5), D0
        CMPW    Y(A5), D0

        BLT     L1

        MOVW    Z(A5), D0
        CMPW    X(A5), D0

        BGT     L3

L1:     MOVW    U(A5), D0
        CMPW    V(A5), D0

        BLT     L2

        MOVWB   W(A5), D0
        CMPW    U(A5), D0

        BGT     L3

L2:     MOVW    Y(A5), X(A5)

        BRA     L4

L3:     MOVW    V(A5), U(A5)
L4:     ..........
```

GENERADORES DE CODIGO AUTOMATICOS

Un generador de código produce código para una máquina en particular al darsele una serie de reglas que definen a esa máquina. Cuando estas reglas se cambian nuevas arquitecturas de máquinas son seleccionadas.

Existen problemas que dificultan esta generación:

- las arquitecturas de las máquinas no estan estandarizadas. Un generador de código debe de adaptarse a las condiciones particulares de una máquina.
- el generador de código debe producir código eficiente, por lo que las caracteristicas particulares o únicas de la máquina deben de tomarse en cuenta.
- las maneras de realizar una operación determinada por una máquina son diferentes.
- el generador de código debe de ser rápido.

Los generadores de código se basan en técnicas de descripción. Estas técnicas toman una descripción formal de exactamente que hace cada instrucción de máquina y entonces las busca en las operaciones que desea ejecutar el programa. Las diferentes máquinas tienen diferentes tipos de instrucciones, por lo que teniendo los efectos de instrucciones particulares de una máquina y comparandolas contra los resultados que se desean en el programa se puede generar código.

B I B L I O G R A F I A

Compiler DEsign and Construction tools an techniques.
with C an d Pascal. Arthur B. Pyster
Van Nostrand Relnhoc. 1988.

Introduction to Compiler Construction with Unix
Axel T. Schreiner, H. George Friedman Jr.
Prentice Hall. 1988.

Theory and Practice of Compiler Writing.
Jean Paul Tremblay, Paul G. Sorenson.
Mc.Graw Hill. 1985.