



EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNOLÓGICO
NACIONAL DE MÉXICO®

Instituto Tecnológico de Matamoros

Unidad 5. Análisis Sintáctico

Nombre de los estudiantes
Nicole Rodríguez González
Víctor Hugo Vázquez Gómez

Asignatura
Lenguajes y Autómatas I

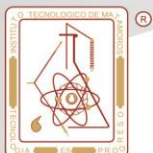
Carrera
INGENIERÍA EN SISTEMAS COMPUTACIONALES

Profesor
Hernández Compeán María Guadalupe

H. Matamoros Tamaulipas México

02 abril 2020

Excelencia en Educación Tecnológica®
Tecnología es progreso®



Índice

Contenido

Índice.....	1
Introducción.....	2
5.1 Definición y clasificación de gramáticas.....	3
5.2 Gramáticas libres de contexto.....	4
5.3 Árboles de derivación.....	5
5.4 Formas normales de Chomsky.....	6
Ejemplo 1:.....	6
5.5 Diagramas de sintaxis.....	7
Potencia de los diagramas de sintaxis.....	7
5.6 Eliminación de ambigüedad.....	8
5.7 Tipos de analizadores sintácticos.....	9
5.8 Generación de matriz predictiva (cálculo first and follow).....	9
5.9 Manejo de errores.....	11
5.10 Generadores de analizadores sintácticos.....	12
Ejemplo.....	13
Conclusión.....	15
Referencias Bibliográficas.....	15

Introducción

Todo lenguaje de programación obedece a unas reglas que describen la estructura sintáctica que debe seguir. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas de contexto libre o con la notación Backus-Naur Form (BNF).

Las gramáticas formales ofrecen ventajas significativas a los diseñadores de lenguajes y a los diseñadores de compiladores:

- Las gramáticas son especificaciones sintácticas y precisas de lenguajes de programación.
- A partir de una gramática se puede generar automáticamente un analizador sintáctico.
- El proceso de generación automática anterior puede llevar a descubrir ambigüedades.
- Una gramática proporciona una estructura a un lenguaje de programación, siendo más fácil generar código y detectar errores.
- Es más fácil ampliar o modificar el lenguaje si está descrito con una gramática.

A continuación comenzaremos con los conceptos básicos de un analizador sintáctico, empezando por la definición y clasificación de las gramáticas que son aceptadas en el analizador, seguido de una definición de qué son las gramáticas libres de contexto, pasando por los árboles de derivación, una explicación de qué son las formas normales de Chomsky y qué tienen que ver en la construcción de un compilador, los diagramas de sintaxis, tipos de analizadores sintácticos, manejo de errores y por último veremos cómo son los generadores de analizadores sintácticos.

5.1 Definición y clasificación de gramáticas

Una gramática libre de contexto tiene cuatro componentes:

- Un conjunto de **símbolos terminales**, a los que algunas veces se les conoce como tokens. Los terminales son símbolos elementales del lenguaje definido por la gramática.
- Un conjunto de **no terminales**, a los que se les conoce como “variables sintácticas”. Cada no terminal representa un conjunto de cadenas o terminales, de una forma que describiremos más adelante.
- Un conjunto de **producciones**, en donde cada producción consiste en un no terminal, llamada encabezado o lado izquierdo de la producción, una flecha y una secuencia de terminales y no terminales llamado cuerpo. La intención de una producción es especificar una de las formas escritas de una instrucción; si el no terminal del encabezado representa a una instrucción, entonces el cuerpo representa una forma escrita de la instrucción.
- Una **designación** de una de las no terminales como el símbolo inicial.

Para especificar las gramáticas presentamos sus producciones, en donde primero se listan las producciones para el símbolo inicial. Suponemos que los dígitos, los signos como $>$ $<$ y $<=$ $>=$ y las cadenas como **while** en negritas son terminales. Un nombre en cursiva es un no terminal, y se puede asumir que cualquier nombre o símbolo que no esté en cursiva es un terminal. Por conveniencia de notación, las producciones con el mismo no terminal que el encabezado puede agrupar sus cuerpos, con los cuerpos alternativos separados por el símbolo $|$ que leemos como “o”.

Ejemplo:

La cadena 9-5+2, 3-1 o 7

Debido a que debe aparecer un signo positivo o negativo entre dos dígitos, nos referimos a tales expresiones como “listas de dígitos separados por signos positivos o negativos”. La siguiente gramática describe la sintaxis de estas expresiones. Las producciones son:

lista \rightarrow *lista* + *dígito*

lista \rightarrow *lista* - *dígito*

lista \rightarrow *dígito*

dígito \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Los cuerpos de las tres producciones con la lista no terminal como encabezado pueden agruparse de la siguiente manera equivalente:

lista \rightarrow *lista* + *dígito* | *lista* - *dígito* | *dígito*

De acuerdo con nuestras convenciones, los terminales de la gramática son los siguientes símbolos:

+ - 0 1 2 3 4 5 6 7 8 9

Los no terminales son los nombres en cursiva *lista* y *dígito*, en donde *lista* es el símbolo inicial, ya que sus producciones se dan primero.

Decimos que una producción es *para* un no terminal, si el no terminal es el encabezado de la producción. Una cadena de terminales es una secuencia de cero o más terminales. La cadena de cero terminales escrita como ϵ , se llama cadena vacía.

5.2 Gramáticas libres de contexto

En el punto anterior vimos que una gramática libre de contexto consiste en **terminales**, **no terminales**, un **símbolo inicial** y **producciones**.

1. Los **terminales** son los símbolos básicos a partir de los cuales se forman cadenas. El término “nombre de token” es un sinónimo de “terminal”.
2. Los **no terminales** son variables sintácticas que denotan conjuntos de cadenas. Los conjuntos de cadenas denotados por los no terminales ayudan a definir el lenguaje generado por la gramática. Los no terminales imponen una estructura jerárquica sobre el lenguaje, que representa la clave para el análisis sintáctico y la traducción.
3. En una gramática, un **no terminal** se distingue como el **símbolo inicial**, y el conjunto de cadenas que denota es el lenguaje generado por la gramática. Por convención, las producciones para el símbolo inicial se listan primero.
4. Las producciones de una gramática especifican la forma en que pueden combinarse los terminales y los no terminales para formar cadenas. Cada producción consiste en:
 - a. Un **no terminal**, conocido como encabezado o lado izquierdo de la producción; esta producción define algunas de las cadenas denotadas por el encabezado.
 - b. El símbolo \rightarrow . Algunas veces se ha utilizado $::=$ en vez de la flecha.
 - c. Un **cuerpo** o **lado derecho**, que consiste en cero o más terminales y no terminales. Los componentes del cuerpo describen una forma en que pueden construirse las cadenas del no terminal en el encabezado.

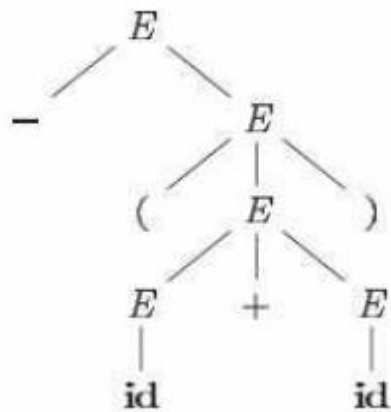
5.3 Árboles de derivación

Un árbol de análisis sintáctico es una representación gráfica de una derivación que filtra el orden en el que se aplican las producciones para sustituir los no terminales. Cada nodo interior de un árbol de análisis sintáctico representa la aplicación de una producción. El nodo interior se etiqueta con el no terminal A en el encabezado de la producción; los hijos del nodo se etiquetan, de izquierda a derecha, mediante los símbolos en el cuerpo de la producción por la que se sustituyó esta A durante la derivación.

Las hojas de un árbol de análisis sintáctico se etiquetan mediante no terminales o terminales y, leídas de izquierda a derecha, constituyen una forma de frase, a la cual se le llama producto o frontera del árbol.

Para ver la relación entre las derivaciones y los árboles de análisis sintáctico, considere cualquier derivación $a_1 \Rightarrow a_2 \Rightarrow \dots \Rightarrow a_n$, en donde a_1 es un sólo no terminal A . Para cada forma de frase a_i en la derivación, podemos construir un árbol de análisis sintáctico cuyo producto se a_i . El proceso es una inducción sobre i .

BASE: El árbol para $a_1 = A$ es un solo nodo, etiquetado como A .



5.4 Formas normales de Chomsky

Una Gramática Independiente del Contexto está en la Forma Normal de Chomsky si todas las producciones son de la forma $S \rightarrow \epsilon$ (donde S es el Símbolo Inicial) o de la forma $A \rightarrow a$, donde $a \in \Sigma$, o de la forma $A \rightarrow BC$; donde A , B y C son SNTs, es decir, del lado derecho de las producciones solo se permite que aparezca un solo Símbolo Terminal o dos SNTs, con la posible excepción de $S \rightarrow \epsilon$.

Cualquier GIC puede ser transformada a la Forma Normal de Chomsky, primero se tienen que aplicar el procedimiento anterior para depurar la gramática y eliminar las producciones ϵ , los símbolos inútiles y las producciones unitarias de G .

Ejemplo 1:

Sea la GIC, sobre el alfabeto $\Sigma = \{a, b\}$, que genera de manera no ambigua al lenguaje formado por las cadenas no vacías, que contienen la misma cantidad de a que b y que está libre de anomalías:

$$\begin{aligned} S &\rightarrow bA \mid aB \\ A &\rightarrow bAA \mid aS \mid a \\ B &\rightarrow aBB \mid bS \mid b \end{aligned}$$

Las únicas dos producciones que previamente están normalizadas, y a las cuales no se les debe hacer nada, son: $A \rightarrow a$ y $B \rightarrow b$

Ahora debemos definir dos producciones nuevas, para los Símbolos Terminales, que se sustituirán en las producciones que no están aún normalizadas:

$$\begin{aligned} C_a &\rightarrow a \\ C_b &\rightarrow b \end{aligned}$$

Entonces, podemos expresar las seis producciones restantes así:

$$\begin{aligned} S &\rightarrow C_bA \mid C_aB \\ A &\rightarrow C_bAA \mid C_aS \\ B &\rightarrow C_aBB \mid C_bS \end{aligned}$$

Cuatro producciones quedan normalizadas con este cambio, luego, para modificar las producciones con tres Símbolos No terminales, agregamos las producciones:

$$\begin{aligned} D_1 &\rightarrow AA \\ D_2 &\rightarrow BB \end{aligned}$$

Reemplazando estos nuevos Símbolos No Terminales en las dos producciones restantes, obtenemos finalmente la GIC equivalente en la FNCh:

$$\begin{aligned} S &\rightarrow C_bA \mid C_aB \\ A &\rightarrow C_bD_1 \mid C_aS \mid a \\ B &\rightarrow C_aD_2 \mid C_bS \mid b \\ C_a &\rightarrow a \\ C_b &\rightarrow b \\ D_1 &\rightarrow AA \\ D_2 &\rightarrow BB \end{aligned}$$

5.5 Diagramas de sintaxis

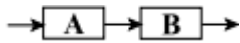
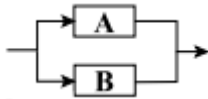
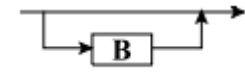
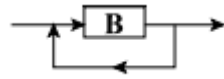
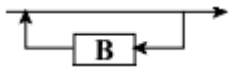
Un diagrama de sintaxis (también llamados diagramas de Conway) es un grafo dirigido donde los elementos no terminales aparecen como rectángulos, y los terminales como círculos o elipses.

Todo diagrama de sintaxis posee un origen y un destino, que no se suelen representar explícitamente, sino que se asume que el origen se encuentra a la izquierda del diagrama y el destino a la derecha.

Cada arco con origen en \hat{a} y destino en \hat{b} representa que el símbolo \hat{a} puede ir seguido del \hat{b} (pudiendo ser \hat{a} y \hat{b} tanto terminales como no terminales). De esta forma todos los posibles caminos desde el inicio del grafo hasta el final, representan formas sentenciales válidas.

Potencia de los diagramas de sintaxis

Demostraremos que los diagramas de sintaxis permiten representar las mismas gramáticas que la notación BNF, por inducción sobre las operaciones básicas de BNF:

Operación	BNF	Diagrama de sintaxis
Yuxtaposición	AB	
Opción	$A \mid B$	
	$g \mid B$	
Repetición	1 o más veces $\{B\}$	
	0 o más veces $[B]$	

Siguiendo la «piedra de Rosetta» que supone la tabla anterior, es posible convertir cualquier expresión BNF en su correspondiente diagrama de sintaxis, ya que A y B pueden ser, a su vez, expresiones o diagramas complejos.

Por otro lado, quien haya trabajado con profusión los diagramas de sintaxis habrá observado que no siempre resulta fácil convertir un diagrama de sintaxis cualquiera a su correspondiente en notación BNF.

5.6 Eliminación de ambigüedad

Algunas veces, una gramática ambigua puede reescribirse para eliminar la ambigüedad. Como ejemplo, vamos a eliminar la ambigüedad de la siguiente gramática del “else colgante”:

$$\begin{array}{lcl} instr & \rightarrow & \text{if } expr \text{ then } instr \\ & & \text{if } expr \text{ then } instr \text{ else } instr \\ & & \text{otra} \end{array} \quad (4.14)$$

Aquí, “otra” representa a cualquier otra instrucción. De acuerdo con esta gramática, la siguiente instrucción condicional compuesta:

if E_1 then S_1 else if E_2 then S_2 else S_3

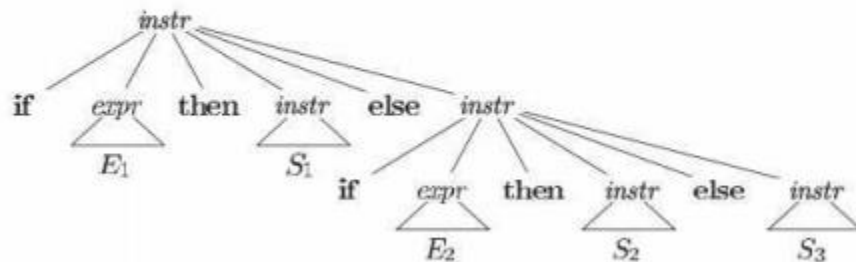


Figura 4.8: Árbol de análisis sintáctico para una instrucción condicional tiene el árbol de análisis sintáctico que se muestra en la figura 4.8. La gramática (4.14) es ambigua, y a que la cadena

$$\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2 \quad (4.15)$$

tiene los dos árboles de análisis sintáctico que se muestran en la figura 4.9.

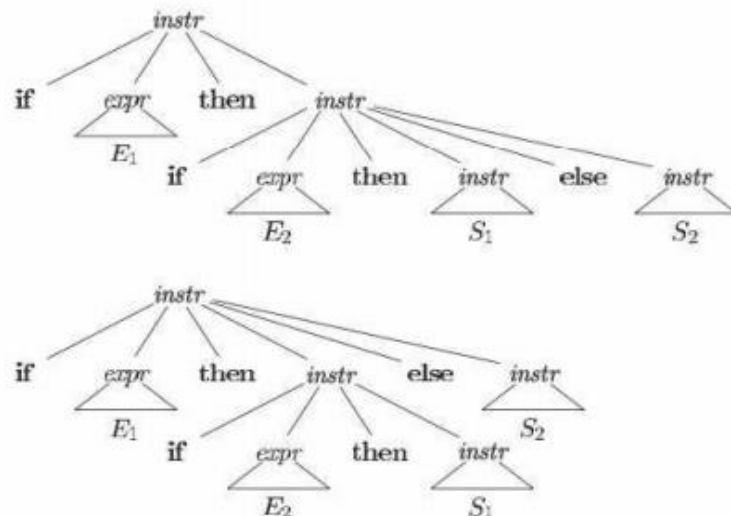


Figura 4.9: Dos árboles de análisis sintáctico para un enunciado ambiguo
En todos los lenguajes de programación con instrucciones condicionales de esta forma, se prefiere el primer árbol de análisis sintáctico. La regla general es,

“Relacionar cada **else** con el **then** más cercano que no esté relacionado”. Esta regla para eliminar ambigüedad puede, en teoría, incorporarse directamente en una gramática, pero en la práctica raras veces se integra a las producciones.

5.7 Tipos de analizadores sintácticos

Según la aproximación que se tome para construir el árbol sintáctico se desprenden dos tipos o clases de analizadores:

Descendentes: parten del axioma inicial, y van efectuando derivaciones a izquierda hasta obtener la secuencia de derivaciones que reconoce a la sentencia.

Pueden ser:

- " Con retroceso. "
- Con funciones recursivas
- De gramáticas LL(1)

Ascendentes: Parten de la sentencia de entrada, y van aplicando derivaciones inversas (desde el consecuente hasta el antecedente), hasta llegar al axioma inicial. Pueden ser:

- Con retroceso. "
- De gramáticas LR(1).

5.8 Generación de matriz predictiva (cálculo first and follow)

La construcción de los analizadores sintácticos descendentes y ascendentes es auxiliada por dos funciones, **primero** y **siguiente**, asociadas con la gramática G. Durante el análisis sintáctico descendente, primero y siguiente nos permiten elegir la producción que vamos a aplicar, con base a los siguientes símbolos de entrada.

Definimos a **primero(α)**, en donde α es cualquier cadena de símbolos gramaticales, como el conjunto de terminales que empiezan las cadenas derivadas a partir de $\alpha \Rightarrow \epsilon$, entonces ϵ también se encuentra en **primero(α)**

Para una vista previa de cómo usar PRIMERO durante el análisis sintáctico predictivo, considere dos producciones A, $A \rightarrow \alpha \mid \beta$, en donde PRIMERO(α) y PRIMERO(β) son conjuntos separados. Entonces, podemos elegir una de estas producciones A si analizamos el siguiente símbolo de entrada α , ya que α puede estar a lo más en PRIMERO(α) o en PRIMERO(β), pero no en ambos. Por ejemplo si α está en PRIMERO(β), elegimos la producción $A \rightarrow \beta$.

Definimos a SIGUIENTE(A), para el no terminal A, como el conjunto de terminales α que pueden aparecer de inmediato a la derecha de A en cierta forma de frase; es decir, el conjunto de terminales A de tal forma que exista una derivación de la forma $S \Rightarrow \alpha A \beta$, para algunas α y β .

Para calcular PRIMERO(X) para todos los símbolos gramaticales X, aplicamos las siguientes reglas hasta que no puedan agregarse más terminales o ϵ a ningún conjunto PRIMERO.

1. Si X es un terminal, entonces $\text{PRIMERO}(X) = \{X\}$
2. Si X es un no terminal y $X \rightarrow Y_1 Y_2 \dots Y_k$ es una producción para cierta $K \geq 1$, entonces se coloca α en $\text{PRIMERO}(X)$ si para cierta i , α está en $\text{PRIMERO}(Y_i)$ y ϵ está en todas las funciones $\text{PRIMERO}(Y_1), \dots, \text{PRIMERO}(Y_{i-1})$; es decir, $Y_1 \dots Y_{i-1} \Rightarrow \epsilon$. Si ϵ está en $\text{PRIMERO}(Y_j)$ para todas las $j = 1, 2, \dots, k$, entonces se agrega ϵ a $\text{PRIMERO}(X)$. Si Y_1 no deriva a ϵ , entonces no agregamos nada más a $\text{PRIMERO}(X)$, pero si $Y_1 \Rightarrow \epsilon$, entonces agregamos $\text{PRIMERO}(Y_2)$, y así sucesivamente.
3. Si $X \rightarrow \epsilon$ es una producción, entonces se agrega ϵ a $\text{PRIMERO}(X)$.

Ahora, podemos calcular PRIMERO para cualquier cadena $X_1 X_2 \dots X_n$ de la siguiente manera. Se agregan a $\text{PRIMERO}(X_1 X_2 \dots X_n)$ todos los símbolos que no sean ϵ de $\text{PRIMERO}(X_1)$. También se agregan los símbolos que no sean ϵ de $\text{PRIMERO}(X_3)$, si ϵ está en $\text{PRIMERO}(X_1)$ y $\text{PRIMERO}(X_2)$; y así sucesivamente. Por último, se agrega ϵ a $\text{PRIMERO}(X_1 X_2 \dots X_n)$ si, para todas las i , ϵ se encuentra en $\text{PRIMERO}(X_i)$.

Para calcular SIGUIENTE(A) para todas las no terminales A, se aplican las siguientes reglas hasta que no pueda agregarse nada a cualquier conjunto SIGUIENTE.

1. Colocar $\$$ en $\text{SIGUIENTE}(S)$, en donde S es el símbolo inicial y $\$$ es el delimitador derecho de la entrada.
2. Si hay una producción $A \rightarrow \alpha B \beta$, entonces todo lo que hay en $\text{PRIMERO}(\beta)$ excepto ϵ está en $\text{SIGUIENTE}(B)$.
3. Si hay una producción $A \rightarrow \alpha B$, o una producción $A \rightarrow \alpha B \beta$, en donde $\text{PRIMERO}(\beta)$ contiene a ϵ , entonces todo lo que hay en $\text{SIGUIENTE}(A)$ está en $\text{SIGUIENTE}(B)$.

5.9 Manejo de errores

Los errores en la programación pueden ser de los siguientes tipos:

- Léxicos, producidos al escribir mal un identificador, una palabra clave o un operador
- Sintácticos, por una expresión aritmética o paréntesis no equilibrados
- Semánticos, como un operador aplicado a un operando incompatible
- Lógicos, puede ser una llamada infinitamente recursiva
- De corrección, cuando el programa no hace lo que realmente el programador deseaba que hiciera.

El manejo de errores de sintaxis es el más complicado desde el punto de vista de la creación de compiladores. Nos interesa que cuando el compilador encuentre el error, no cancele definitivamente la compilación, sino que se recupere y siga buscando errores. Recuperar un error no quiere decir corregirlo, sino ser capaz de seguir construyendo el árbol sintáctico a pesar de los errores encontrados. En vista de esto, el manejador de errores de un analizador sintáctico debe tener como objetivos:

- Indica los errores de forma clara y precisa. Debe informar mediante los correspondientes mensajes del tipo de error y su localización.
- Recuperarse del error, para poder seguir examinando la entrada.
- Distinguir entre errores y advertencias.
- No ralentizar significativamente la compilación

Un buen compilador debe hacerse siempre teniendo también en mente los errores que se pueden producir, con lo que se consigue simplificar su estructura. Además si el propio compilador está preparado para admitir incluso los errores más frecuentes, entonces se puede mejorar la respuesta ante esos errores incluso corrigiéndolos.

Estrategias para gestionar los errores una vez detectados:

- **Ignorar el problema:** consiste en ignorar el resto de la entrada hasta llegar a una condición de seguridad. Una condición tal se produce cuando nos encontramos un token especial (como el ';' o END). A partir de ese punto se sigue analizando normalmente.
- **Recuperación a nivel de frase:** Intenta corregir el error una vez descubierto. Por ejemplo, insertar un ';' si hace falta al final de una línea que lo necesite. Este método es de especial cuidado porque puede dar lugar a recuperaciones infinitas, donde el intento no es el acertado e intenta corregirlo de manera equivocada.

- **Reglas de producción adicionales:** Este mecanismo añade a la gramática formal que describe el lenguaje de reglas de producción para reconocer los errores más comunes.
- **Corrección global:** Este método intenta por todos los medios obtener un árbol sintáctico para una secuencia de tokens. Si hay algún error y la sentencia no se puede reconocer, entonces este método infiere una secuencia de tokens sintácticamente correcta lo más parecida a la original y genera un árbol para dicha secuencia.

5.10 Generadores de analizadores sintácticos

Para esta investigación usaremos Yacc (Yet Another compiler-compiler) escrito por S. C. Johnson en los 70's. Este generador está disponible en UNIX y se ha utilizado para ayudar a implementar muchos compiladores de producción.

En primer lugar se prepara un archivo, por decir **traducir.y** el cual contiene una especificación de Yacc. El comando para ejecutar el fichero sería:

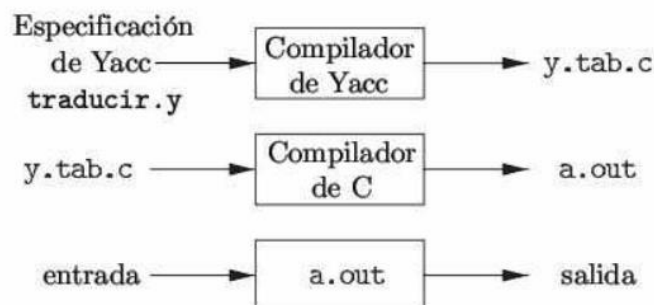
```
yacc traducir.y
```

se transforma a un archivo en c llamado y.tab.c. Este es una representación de un analizador sintáctico LALR escrito en C. La tabla de análisis sintáctico LR se compacta. Al compilar ese archivo en C junto con la biblioteca ly que contiene el programa de análisis sintáctico LR mediante el uso del comando:

```
cc y.tab.c -ly
```

obtenemos el programa objeto a.out, el cual realiza la traducción específica por el programa original en Yacc.

Un programa fuente en Yacc tiene tres partes:



```
declaraciones
%%
reglas de traducción
%%
soporte de las rutinas en C
```

Ejemplo

Programa de conversión binaria a decimal

Con este analizador en Yacc daremos un input en binario y el output será su conversión a decimal

Código para el analizador léxico

```
%{
#include<stdio.h>
#include<stdlib.h>
#include"y.tab.h"
extern int yylval;
}%

%%
0 {yylval=0;return ZERO;}
1 {yylval=1;return ONE;}
[ \t] {;}
\n return 0;
. return yytext[0];
%%

int yywrap()
{
return 1;
}
```

Código para el analizador sintáctico yacc/bison

```
%{
#include<stdio.h>
#include<stdlib.h>
void yyerror(char *s);
}%
%token ZERO ONE

%%
```

```

N: L {printf("\n%d", $$);}
L: L B {$$=$1*2+$2;}
  | B {$$=$1;}
B: ZERO {$$=$1;}
  | ONE {$$=$1;};
%%

int main()
{
while(yyparse());
}

yyerror(char *s)
{
fprintf(stdout, "\n%s", s);
}

```

Salida:



```

Terminal - victor@ada: ~/Documentos/escuela/sextosemestre/lenguajes-automatas-1/uni...
→ sintatic-analyzer git:(master) X yacc -d parser.y
→ sintatic-analyzer git:(master) X lex analizador.l
→ sintatic-analyzer git:(master) X gcc lex.yy.c y.tab.c -ll
y.tab.c: In function 'yyparse':
y.tab.c:1113:16: warning: implicit declaration of function 'yyerror' [-Wimplicit-function-declaration]
    yychar = yylex ();
               ^~~~~~
parser.y: At top level:
parser.y:21:1: warning: return type defaults to 'int' [-Wimplicit-int]
yyerror(char *s)
^~~~~~
parser.y:21:1: warning: conflicting types for 'yyerror'
parser.y:4:6: note: previous declaration of 'yyerror' was here
void yyerror(char *s);
      ^~~~~~
→ sintatic-analyzer git:(master) X ./a.out
10
2%
→ sintatic-analyzer git:(master) X ./a.out
1111
15%
→ sintatic-analyzer git:(master) X ./a.out
1010101
85%
→ sintatic-analyzer git:(master) X

```

Conclusión

El analizador sintáctico es una de las partes más complejas del compilador, construir uno requiere de mucho tiempo ya que de él dependen marcar los errores sintácticos. Estos que son tan comunes como por ejemplo la falta de un “;” al final de una línea.

Como vemos muchas de las tareas más largas del compilador las realiza este analizador sintáctico en base a una gramática dada por el compilador.

Para realizar un analizador sintáctico usamos Yacc donde recibe una gramática junto con la información de resolución de conflictos. Después el analizador realiza un análisis sintáctico ascendente y llama a una función asociada cada vez que realiza una reducción.

Referencias Bibliográficas

1. Traductores y compiladores con Lex/Yacc, JFLEX/CUP y JavaCC. Edición electrónica. Sergio Gálvez Rojas, Miguel Ángel Mora Mata. páginas 50, 51, 52, 56, 57, 60, 61
2. Compiladores, principios, técnicas y herramientas; segunda edición. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. páginas 192, 193, 194, 197, 201, 210, 220
3. Teoría de la computación, Jorge Eduardo Carrión Viramontes. ISBN: 968-5354-82-0. Páginas 123