



Unidad IV Análisis Sintáctico

M.C. Juan Carlos Olivares Rojas

Agenda

- 4.1 Introducción a las Gramáticas libres de contexto y árboles de derivación.
- 4.2 Diagramas de sintaxis.
- 4.3 Precedencia de operadores.
- 4.4 Analizador sintáctico.
 - 4.4.1 Analizador descendente (LL).
 - 4.4.2 Analizador ascendente(LR, LALR).



Agenda

- 4.5 Administración de tablas de símbolos
- 4.6 Manejo de errores sintácticos y su recuperación
- 4.7 Generadores de código para analizadores sintácticos: Yacc, Bison



4.1 Introducción a las Gramáticas libres de contexto y árboles de derivación

- Todo lenguaje posee una serie de reglas para describir los programas fuentes (sintaxis).
- Un analizador sintáctico implementa estas reglas haciendo uso de GICs



Gramáticas

- Son un formalismo matemático que permite decidir si una cadena pertenece a un lenguaje dado.
- Se define como la quarteta $G = (N, \Sigma, S, P)$, en donde N es el conjunto de símbolos terminales, Σ es conjunto de símbolos terminales, S es el símbolo inicial (S pertenece a N) y P es un conjunto de reglas de producción.



Gramáticas

- Los símbolos no terminales (N) son aquellos que pueden seguir derivando en otros; mientras que los terminales el proceso finaliza allí.
- Las reglas de producción siguen el formato: $\alpha \rightarrow \beta$ donde α y β pertenecen a N y Σ en cualquier forma.



Reglas de producción

- Son las reglas que permiten decidir si la cadena pertenece a un lenguaje y la estructura que lleva:
- $S \rightarrow A|aB$ $B \rightarrow \varepsilon$
- $A \rightarrow aA|bC$ $C \rightarrow \varepsilon$
- $S \rightarrow$ Genera cadenas del lenguaje a^*b u a



Tipos de gramáticas

- Las gramáticas más sencillas son las gramáticas regulares, debido a que no presentan anomalías de ningún tipo. Desafortunadamente este tipo de gramáticas no permiten expresar todos los lenguajes posibles y en especial los humanos por lo que se necesitan otros tipos de gramáticas. Las más utilizadas en informática son las libres del contexto.



Gramáticas Regulares

- Son las que se forman a través de Autómatas Finitos Deterministas y Expresiones regulares. No presentan ambigüedades.
- Sus reglas de producción son del tipo: $\alpha \rightarrow \beta$ donde α pertenece a N y β pertenece a $(\Sigma)^* N$?



Gramáticas Independientes del Contexto

- Son aquellas G cuya reglas de producción son de la forma: $\alpha \rightarrow \beta$, en donde α pertenece a N y β pertenece $(N \cup \Sigma)^*$
- Las ventajas de uso de GICs son:
- Proporcionan una estructura sintáctica precisa y fácil de comprender



Gramáticas Independientes del Contexto

- Proporciona al lenguaje fuente una estructura adecuada para la generación del código.
- Por medio de las GICs es fácil construir analizadores sintácticos
- Es sencillo añadir funcionalidades a un analizador sintáctico



GICs

- Hay que revisar que la gramática no sea inherentemente ambigua para poder eliminar esa ambigüedad o rediseñar la gramática sin anomalías.
- Algunas formas de eliminar esa ambigüedad es utilizando técnicas como algoritmos CYK y las formas normales de Chomsky (FNCh) y Greibach (FNG).



Ejemplos de GICs

- Expresiones válidas en lenguajes C:

Expr \rightarrow (expr) | - expr | expr op expr | VAR |
NUM

- Error sintáctico: cuando la secuencia de componentes léxicos no puede ser generada por la gramática del lenguaje fuente.



Ejemplos de GICs

- Declaración de variables en C:

Decl → TIPO listavar PYC

Listavar → var | var COMA listavar

Var → ID | ASTER var

Dimension → CI ENTERO CD | CI ENTERO
CD dimension



Jerarquía de Chomsky

- Las otras dos gramáticas en las cuales clasificó Chomsky (GR tipo 3, GIC tipo 2) son las gramáticas sensible al contexto (tipo 1, donde $|\alpha| < |\beta|$, donde α y β pertenecen a $(\Sigma \cup N)^*$ salvo ϵ) y las gramáticas del tipo 0 o sin restricciones, las cuales sus reglas de producción pueden ser de cualquier tipo.



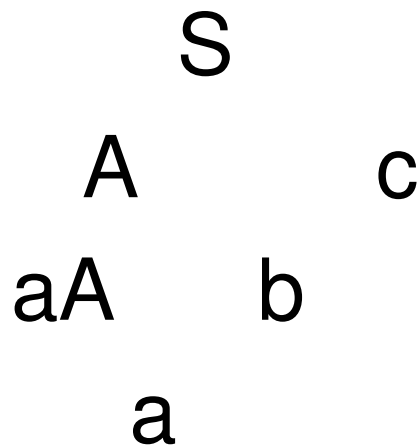
Árboles de Derivación

- Es la representación gráfica de la derivación de una cadena.
- Se crea utilizando el símbolo inicial como la raíz, los símbolos N representan nodos del árbol y los símbolos Σ las hojas del árbol.
- A través de los árboles de derivación se puede verificar la sintaxis de un lenguaje así como comprobar el significado de las palabras.



Árboles de derivación

- Si para la misma cadena existen dos o más árboles de derivación la gramática es ambigua.



BNF

- La Forma Backus-Naur es una meta-sintaxis; es decir, una sintaxis para representar sintaxis. Es un estándar para representar lenguajes.
- Los paréntesis triangulares \langle y \rangle sirven para indicar los símbolos no terminales.
- La barra vertical $|$ para representar \cup



BNF

- La doble flecha \Rightarrow indica las derivaciones
- $::=$ indica \rightarrow las producciones
- $[]$ indican elementos opcionales
- $\{\}$ indican términos repetitivos

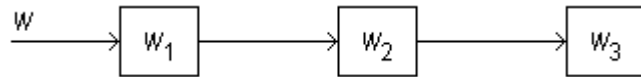


4.2 Diagramas de sintaxis

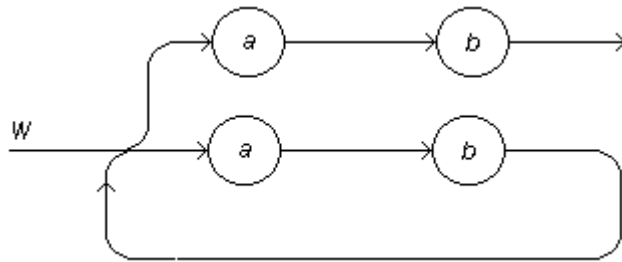
- Es otra forma (al igual que los árboles de derivación) de especificar gramáticas del tipo 2.
- La característica de este esquema es que permite ver las derivaciones al instante de que ocurren.



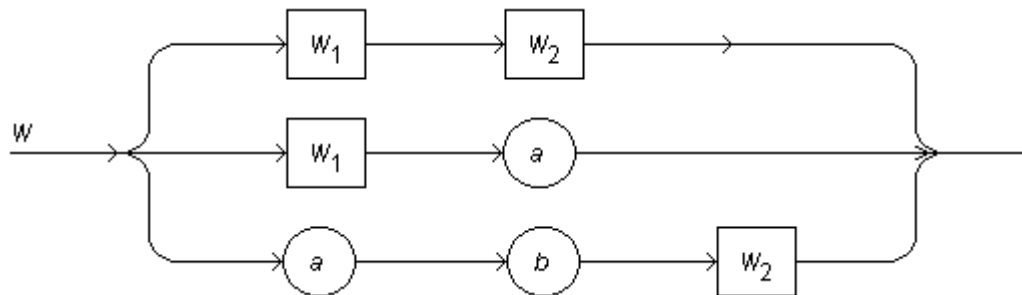
Diagramas de sintaxis



$\langle W \rangle ::= \langle W_1 \rangle \langle W_2 \rangle \langle W_3 \rangle$



$\langle W \rangle ::= ab \langle W \rangle$



$\langle W \rangle ::= \langle W_1 \rangle \langle W_2 \rangle \mid$
 $\langle W_1 \rangle a \mid ab \langle W_2 \rangle$



4.3 Precedencia de operadores

- La precedencia de operadores es de vital importancia en el proceso de análisis sintáctico ya que nos representará la forma en que debe construirse el árbol de derivación.
- En aritmética existen prioridades, por ejemplo: $*$ y $/$ tienen preferencia sobre $+$ y $-$. $()$ indican la máxima prioridad.



Prioridad de operadores

- La instrucción $a = b + c / 2$ en la mayoría de los lenguajes no se evalúa de la forma $a = (b + c) / 2$, sino de la forma $a = b + (c/2)$
- La forma de evaluación depende de cómo se construyan los operadores, ya sea en infijo, postfijo o prefijo.
- Las operaciones se realizan de abajo hacia arriba.



4.4 Analizador sintáctico

- Un analizador sintáctico (Parser) es un programa que reconoce si una o varias cadenas de caracteres forman parte de un determinado lenguaje. Los lenguajes habitualmente reconocidos por los analizadores sintácticos son los lenguajes libres de contexto.



Analizador sintáctico

- Los analizadores sintácticos fueron extensivamente estudiados durante los años 70 del siglo XX, detectándose numerosos patrones de funcionamiento en ellos, cosa que permitió la creación de programas generadores de analizadores sintácticos a partir de una especificación de la sintaxis del lenguaje, tales y como YACC, GNU bison y javacc.

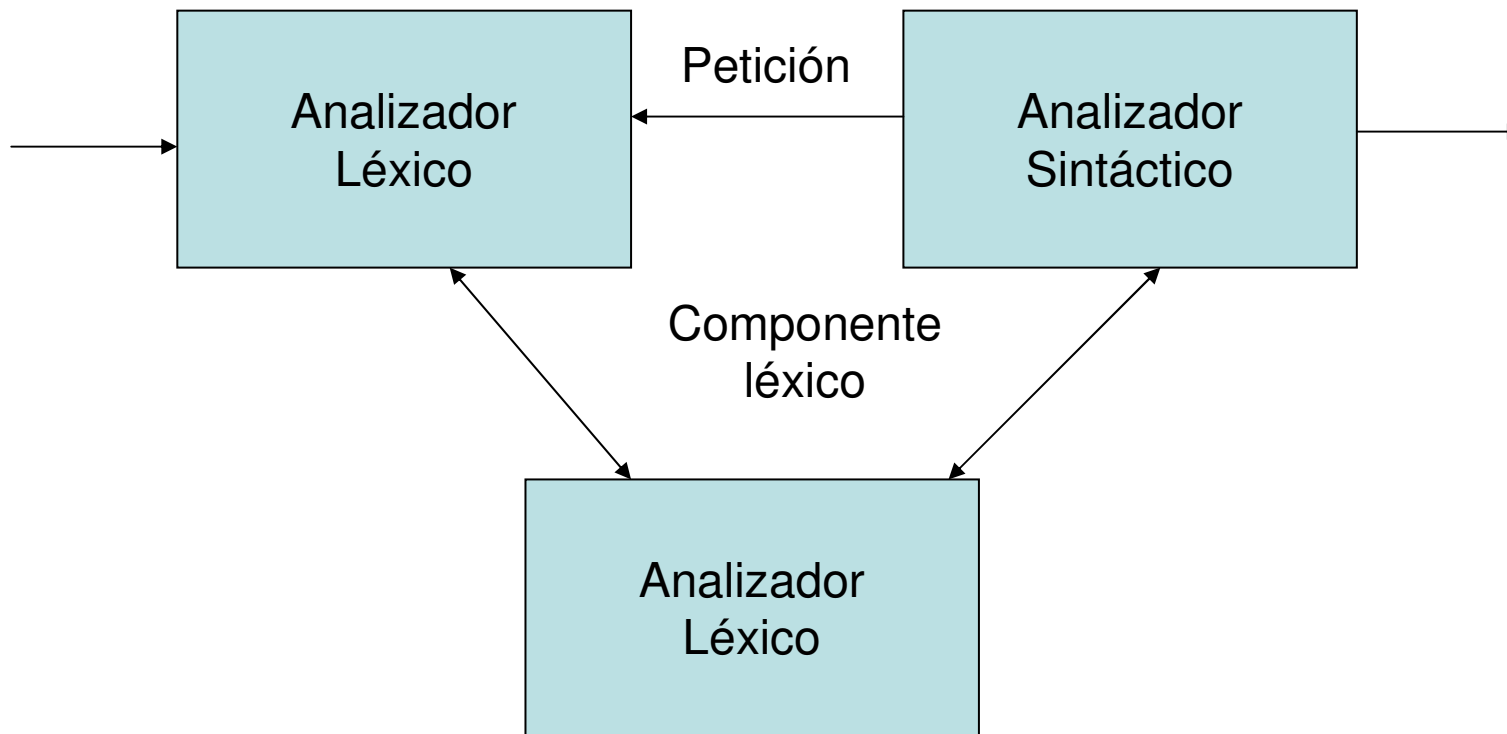


Análisis sintáctico

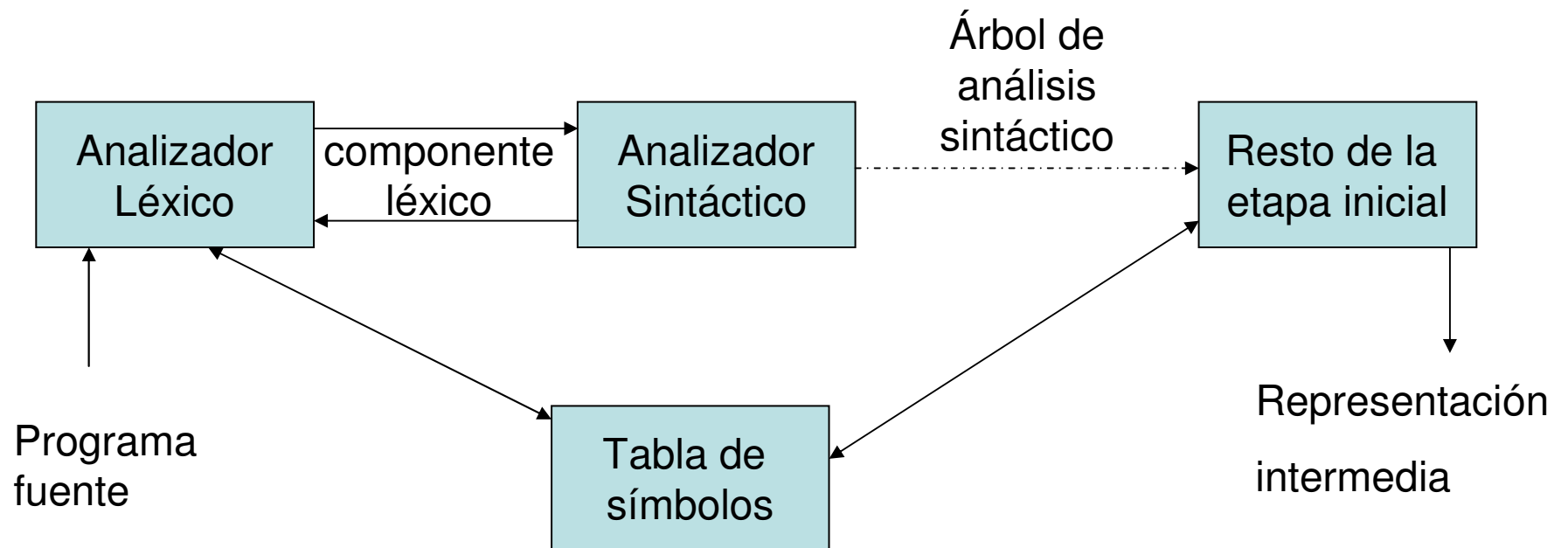
- Es el proceso de determinar si una cadena dada puede ser generada por una gramática.
- Los analizadores sintácticos de lenguajes de programación suele hacerse de izquierda a derecha, viendo un componente léxico a la vez
- Los analizadores pueden clasificarse dependiendo de la forma en como se construyen los nodos del árbol de derivación sintáctico: ascendentes y descendentes



Análisis sintáctico



Análisis sintáctico



Análisis sintáctico

- Las gramáticas se pueden expresar en forma BNF (Backus-Naur Form).
- El análisis sintáctico impone una estructura jerárquica.

id1 :=
id2 +
id3 *
60



Tipos de analizadores sintácticos

- LL (left to left) leen la cadena de izquierda a derecha y derivan por la izquierda
- LR (left to right)
- $S \rightarrow aA$
- $A \rightarrow aBbC$
- $B \rightarrow b$
- $C \rightarrow c$



4.4.1 Analizador descendente (LL)

- *Existen diferentes métodos de análisis sintáctico. La mayoría caen en una de dos categorías: ascendentes y descendentes. Los ascendentes construyen el árbol desde las hojas hacia la raíz. Los descendentes lo hacen en modo inverso.*
- *Un analizador ampliamente utilizado se denomina método de análisis predictivo descendente recursivo que es muy sencillo.*



Analizador descendente

- Derivación izquierda:
- $S \rightarrow aA \rightarrow aaBbC \rightarrow aabbC \rightarrow aabbc$ (1234)
- $S \rightarrow aA \rightarrow aaBbC \rightarrow aaBbc \rightarrow aabbc$ (3421)
- LL(k) traductores “top-down”
- Un análisis anticipado de k caracteres



Analizador descendente

- $S \rightarrow aS | cA$
- $A \rightarrow bA | cB | \epsilon$
- $B \rightarrow cB | a | \epsilon$
- Construir cadena acbb
- $S \rightarrow aS$ o $S \rightarrow cA$; al anticipar el primer símbolo



Analizador descendente

Prefijo	Anticipación	Regla Aplicar	a	Derivación
ϵ	a	$S \rightarrow aS$		$S \rightarrow aS$
a	c	$S \rightarrow cA$		$\rightarrow acA$
ac	b	$A \rightarrow bA$		$\rightarrow acbA$
acb	b	$A \rightarrow bA$		$\rightarrow acbbA$
Acbb	ϵ	$A \rightarrow \epsilon$		$\rightarrow accbb$



Analizador descendente

- $L = \{a^n abc^n \mid n > 0\}$
- $S \rightarrow aSc \mid aabc$

- Se requiere una anticipación de por los menos tres símbolos LL(3)

- $S \rightarrow aaAc$
- $A \rightarrow Sc \mid abc$ LL(1)



4.4.2 Analizador ascendente (LR, LALR)

- Algunos problemas no se pueden resolver de forma descendente ya que no están fácil quitar la ambigüedad. En algunos casos es más fácil demostrar algo ya existente.
- Generalmente los analizadores sintácticos LR(k) son del tipo “bottom-up”



Analizador ascendente

- El analizador trata de reducir la cadena de entrada w al símbolo inicial S . En un proceso que recorre el árbol de derivación en sentido inverso que se llama reducción.
- No sólo es necesario una gramática que no presente ambigüedades sino que también tenga el valor de k más pequeño.



Analizador ascendente

Reducción	Regla a aplicar
$(b)+b$	
$(T)+b$	$T \rightarrow b$
$(A)+b$	$A \rightarrow T$
$T+b$	$T \rightarrow (A)$
$A+b$	$T \rightarrow b$
A	$A \rightarrow A+T$
S	$S \rightarrow A$



4.5 Administración de tablas de símbolos

- La tabla de símbolos se crea durante la fase de análisis léxico a través de los componentes léxicos, pero en el proceso de análisis sintáctico sufren algunas modificaciones.
- Generalmente se agregan valores de tipo y significado para el análisis sintáctico.



4.6 Manejo de errores sintácticos y su recuperación.

- Si los traductores tuvieran que procesar programas correctas el proceso de implantación se simplificaría mucho.
- ¿Cómo debe de responder un compilador de pascal a un código Fortran?
- Ningún método de recuperación de errores resuelve todos los problemas



Tipos de errores

- Léxicos: como escribir mal un identificador, palabra clave u operador.
- Sintácticos: como una expresión aritmética con paréntesis no equilibrados.
- Semánticos: como un operador aplicado a un operadorando incompatible.
- Lógicos: como una llamada infinitamente recursiva



Tipos de errores

- La mayoría de los errores se centra en la fase de análisis sintáctico.
- El manejador de errores debe:
- Informar la presencia de errores con claridad y exactitud.



Administrador de errores

- Recuperar de cada error con la suficiente rapidez como para detectar errores posibles.
- No debe retrasar de manera significativa el procesamiento de programas correctos.
- Debe indicar la línea del error y algún mensaje informativo



Estrategias de recuperación de errores

- Modo Pánico
- Nivel de Frase
- Producciones de error
- Corrección global



Recuperación en modo pánico

- Es el más sencillo de implantar.
- El analizador sintáctico desecha componentes léxicos hasta encontrar un carácter de sincronización. Estos caracteres son el punto y como (;) entre otros.



Recuperación en modo pánico

```
int a,b,c;
```

```
struct c {
```

```
....
```

```
}
```

```
main()
```

```
{
```

```
  int a;
```

```
}
```



Recuperación a nivel de frase

- Esta técnica utiliza una corrección de caracteres adyacentes, ya sea por inserción, eliminación o intercambio.
- Esta técnica permite sustituir , por ;, etc. Son traductores que corrigen errores. Desafortunadamente para muchos casos no aplican por lo que no se utilizan demasiados.



Producciones de error

- Se pueden generar gramáticas para generar producciones de error y así de esta forma seguir con el proceso.
- La dificultad radica en el sentido de encontrar esas reglas gramaticales para generar error. En algunos casos sería incluso más extensa que la gramática del propio lenguaje.
- `for(i<3, a<10; i++)`



Corrección global

- Idealmente, sería recomendable que un traductor hiciera el mínimo de cambios para procesar una entrada inválida. Este algoritmo genera menores costos globales para realizar cambios.
- El problema radica en que el implementar estas estrategias son muy costosas en tiempo y espacio.



4.7 Generadores de código para analizadores sintácticos: Yacc, Bison

- YACC (YET ANOTHER COMPILER-COMPILER): provee una herramienta general para describir la entrada de un programa de computación. El usuario de YACC especifica las estructuras de su entrada, junto con el código que será invocado en la medida en que cada una de esas estructuras es reconocida.



YACC/BISON

- YACC convierte esa especificación en una subrutina que maneja el proceso de entrada.
- La subrutina de entrada producida por YACC llama a la rutina provista por el usuario para devolver el próximo ítem básico de la entrada.



YACC/BISON

- GNU Bison es un generador de parsers de propósito general que convierte una descripción gramatical desde una gramática libre de contexto LALR en un programa en C para hacer el parser.
- Es utilizado para crear parsers para muchos lenguajes, desde simples calculadoras hasta lenguajes complejos.



YACC/BISON

- GNU Bison tiene compatibilidad con Yacc: todas las gramáticas bien escritas para Yacc, funcionan en Bison sin necesidad de ser modificadas.
- Cualquier persona que esté familiarizada con Yacc podría utilizar Bison sin problemas. Es necesaria experiencia con C para utilizar Bison.



YACC

- Yet Another Compiler-Compiler
- Analizador.y (#include "lex.yy.c") → bison → analizador.c (y.tab.c) → gcc → analizador
- \$gcc analizador.c -o analizador -lfl



Estructura de un programa en Bison

```
%{
```

Declaraciones globales C

```
%}
```

Declaraciones bison

```
%%
```

Gramáticas

Nombre:prod1 |prod2|... |prodn;

```
%%
```

Código auxiliar C



Tips

- Todo lexema debe ser un entero

```
#define VAR 200 (256)  
return (VAR);
```

Gramática vacía

```
Gramática:      prod1 |  
                prod2 |  
                ;
```



Tips

- Reduce/Reduce \rightarrow ambigüedad infinita
- Shift/Reduce
- $S \rightarrow A$ $S \rightarrow b$
- $A \rightarrow b$
- $A \rightarrow a|b|c$ demasiada profundidad
- $A \rightarrow x|y|z$



Analizador.lex

```
%{  
  #include "ytab.h"  
}%  
sp  [\n\r\t]  
lf [i][f]  
%%  
{if}  {return (IF);}  
“(” {return (PI); }  
. return (ERROR);  
%%
```



Analizador.y

```
%{
```

```
#include "lex.yy.c"
```

```
%}
```

```
%token IF PI PD LLI LLD
```

```
%token ID NUM OPREL OPLOG
```

```
%%
```

```
programa: linea programa
```

```
        |
```

```
        ;
```



Analizador.y

```
Linea:  if linea
        |
        ;
if:  if PI condicion PD LLI campo LLD
    ;
.:  {printf("Error sintáctico");}
%%
main(int argc, char *argv[])
{
    FILE *f = fopen(argv[1], "r");
    yyin = f;
    while (yyparse());
    fclose(f);
}
```



Bison

```
$flex analizador.lex
```

```
$bison analizador.y
```

```
$gcc analizador.c -o analizador -lfl
```

- yytext componente léxico
- yyin flujo de entrada
- yylineno línea de error



```
%%
```

```
yyerror()
```

```
{  
    printf("Error sintáctico en %d linea", yylineno);  
}
```



¿Preguntas?

