

UNIVERSIDAD NACIONAL DE INGENIERÍA

Teoría de la Computación

Gramáticas libres de contexto

Derivaciones, ambigüedad, forma normal de Chomsky y CYK

Gerson Garrido

Semana 5

Hoy

Sesión A

1. Lo que las expresiones regulares no alcanzan
2. Gramáticas, derivaciones y árboles
3. Diseñar una gramática
4. Demostrar que genera un lenguaje
5. Ambigüedad

Sesión B

6. El problema de la pertenencia
7. La forma normal de Chomsky
8. La conversión en cuatro pasos
9. El algoritmo CYK
10. Su costo

Lectura

Apuntes de la semana 5. Apoyo: Sipser, sec. 2.1 y, de la sec. 7.2, el teorema de que todo lenguaje libre de contexto está en P; Erickson, *Models of Computation*, nota 5. Código en `code/week-05`.

SECCIÓN 01

Lo que las expresiones regulares no alcanzan

01



De dónde venimos

- Semana 4: $\{\texttt{o}^n\texttt{1}^n : n \geq 0\}$ **no es regular**. Un autómata finito no cuenta sin tope.
- Ese lenguaje tiene la forma de casi todo lo que le escribimos a una computadora: algo que se abre, algo adentro, algo que se cierra.

Paréntesis

`(())()` sí
`)(` y `(()` no

Expresiones

`a+a*a`
`(a+a)*a`

JSON

`[1, [2, 3], []]`
listas dentro de listas

Estructura anidada

Las tres se describen igual: cómo se arma una cadena del lenguaje **a partir de cadenas más cortas del mismo lenguaje**.

Reconocer y generar

Un autómata reconoce

- Recibe una cadena.
- Responde sí o no.

Una gramática genera

- Da reglas para producir las cadenas.
- Las reglas pueden ser recursivas.

Un valor de JSON, simplificado:

$$V \to \mathrm{n} \mid [\,] \mid [\,L\,]$$
$$L \to V \mid V\,,\,L$$

Un valor es un número, una lista vacía, o una lista de valores entre corchetes.

En la práctica

La sintaxis de C++, de Python y de JSON se publica así. El programa que encuentra la estructura de un texto según una gramática es un **parser**.

SECCIÓN 02

Gramáticas, derivaciones y árboles

02

La definición formal

Gramática libre de contexto *(context-free grammar, CFG)*

Una cuádrupla $G = (V, \Sigma, R, S)$:

1. V: conjunto finito de **variables**;
2. Σ: conjunto finito de **terminales**, sin elementos en común con V;
3. R: conjunto finito de **reglas** $A \to u$, con $A \in V$ y $u \in (V \cup \Sigma)^*$;
4. $S \in V$: la **variable inicial**.

- Los terminales son el alfabeto del lenguaje. Las variables nombran lo que falta escribir.
- $A \to \varepsilon$ es una regla válida. $A \to u \mid v$ abrevia dos reglas.
- El primer ejemplo: $S \to \texttt{0}S\texttt{1} \mid \varepsilon$.

Las gramáticas como archivos

machines/zeros_ones.cfg

```
# { 0^n 1^n : n >= 0 }
variables: S
terminals: 0 1
start: S
rules:
S -> 0 S 1 | eps
```

terminal

```
$ build/show_cfg machines/zeros_ones.cfg
G = (V, Σ, R, S)
  V = {S}
  Σ = {0, 1}
  S = S
  R tiene 2 reglas:
    S → 0S1
    S → ε
Cadenas de L(G) con longitud a lo sumo 6 (4):
  ε 01 0011 000111
```

Los símbolos van **separados por espacios**, y la alternativa vacía es `eps`. Error común: escribir `S -> 0S1`. Sin espacios, `0S1` es un solo nombre, que no es variable ni terminal.

Derivaciones

Un paso

Si $A \to x$ es una regla, entonces $uAv \Rightarrow uxv$, para cualesquiera cadenas u y v.

$$S \Rightarrow \texttt{0}S\texttt{1} \Rightarrow \texttt{00}S\texttt{11} \Rightarrow \texttt{000}S\texttt{111} \Rightarrow \texttt{000111}$$

- El último paso usa $S \to \varepsilon$: la S se reemplaza por nada.
- $u \Rightarrow^* v$: de u se llega a v con **cero o más** pasos.
- $L(G) = \{\, w \in \Sigma^* : S \Rightarrow^* w \,\}$. Solo cuentan las cadenas **sin variables**.
- En cada paso hay elecciones. $L(G)$ junta lo que sale por **todos** los caminos.
- Un lenguaje es **libre de contexto** si alguna gramática libre de contexto lo genera.

Por qué “libres de contexto”

La regla $A \to x$ se aplica a una A **esté donde esté**: lo que la rodea no importa.

Lema del contexto

Si $x \Rightarrow^* y$, entonces $uxv \Rightarrow^* uyv$.

Lema de separación

Si $X_1 X_2 \cdots X_k \Rightarrow^* w$ con n pasos, entonces $w = w_1 w_2 \cdots w_k$, con $X_i \Rightarrow^* w_i$ en n_i pasos y $n_1 + \cdots + n_k = n$.

- Cada símbolo trabaja por su cuenta, y los trozos quedan en el mismo orden.
- Tiene sentido preguntar “¿qué cadenas genera A?” para cada variable por separado.

Derivaciones por la izquierda

Con $S \to AB$, $A \to \mathtt{a}$, $B \to \mathtt{b}$, la cadena $\mathtt{ab}$ tiene dos derivaciones:

$$S \Rightarrow AB \Rightarrow \mathtt{a}B \Rightarrow \mathtt{ab} \qquad\qquad S \Rightarrow AB \Rightarrow A\mathtt{b} \Rightarrow \mathtt{ab}$$

Derivación por la izquierda *(leftmost derivation)*

Cada paso reemplaza la variable que está **más a la izquierda**. Toda cadena de $L(G)$ tiene al menos una.

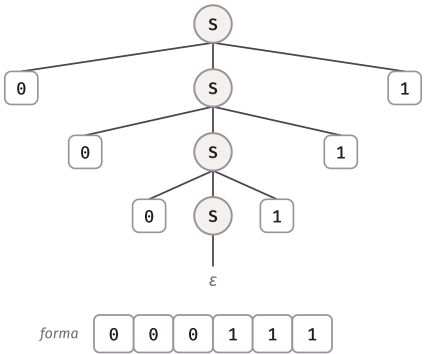
terminal

```
$ build/derive machines/two_derivations.cfg ab --sin-arboles
w = ab
derivaciones por la izquierda: 1

1. S ⇒ AB ⇒ aB ⇒ ab
   3 pasos
```

De una derivación sale un árbol

EL ÁRBOL



LA DERIVACIÓN

	S	
⇒	0S1	S → 0S1
⇒	00S11	S → 0S1
⇒	000S111	S → 0S1
⇒	000111	S → ε

Paso 4, regla S → ε: la S se reemplaza por nada, y la forma queda sin variables.

Las hojas, de izquierda a derecha y saltando ε, dan `000111`. Fueron 4 pasos: uno por cada nodo interno.

Árboles de derivación

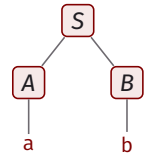
Árbol de derivación *(parse tree)*

- Cada nodo interno lleva una variable; cada hoja, un terminal o ε.
- Un nodo A con hijos $X_1, \ldots, X_k$ es una regla $A \to X_1 \cdots X_k$.

- Las hojas, de izquierda a derecha: la cadena. En el árbol anterior, `000111`.
- Guarda **qué regla** se aplicó a cada variable, sin el orden. Cada nodo interno es un paso.

Un árbol, una derivación por la izquierda

- De una derivación sale un árbol: cada paso le cuelga hijos a una variable.
- De un árbol salen **varias** derivaciones: una por cada orden de sus nodos internos.
- De un árbol sale **una sola** derivación por la izquierda, en preorden. Y árboles distintos dan derivaciones por la izquierda distintas.



Las dos derivaciones de `ab` tienen este árbol.

Idea clave

El árbol es la **estructura** de la cadena. Un compilador quiere el árbol: dice qué operación se aplica a qué.

SECCIÓN 03

Diseñar una gramática

03

El método: armar con cadenas más cortas

1. **¿Cuáles son las cadenas más cortas?** Los casos base: reglas sin variables a la derecha.
2. **¿Cómo sale una más larga de una o dos más cortas?** Las reglas recursivas.
3. **¿Toda cadena del lenguaje se arma así?** Si no, faltan reglas.

$\{0^n 1^n\}$

$$S \to 0S1 \mid \varepsilon$$

Un símbolo en cada extremo **a la vez**.

Palíndromos sobre $\{a, b\}$

$$S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$$

Mismo símbolo en los extremos, palíndromo en medio.

Lo que un DFA no puede imitar

La variable queda **en medio**: lo que se escribe a su izquierda queda emparejado con lo de su derecha.

Paréntesis balanceados

Balanceada: ningún prefijo cierra más de lo que abrió, y al final no queda nada abierto.

$$S \to (S)S \mid \varepsilon$$

El primer (se cierra en algún). Lo de adentro está balanceado, y lo que sigue también.

$$S \to SS \mid (S) \mid \varepsilon$$

Dos balanceadas seguidas, o una entre paréntesis, o nada.

terminal

```
$ build/show_cfg machines/parens.cfg 6 | tail -2
Cadenas de L(G) con longitud a lo sumo 6 (9):
  ε () (()) ()() ((())) (()()) (())() ()(()) ()()()
```

Las dos generan lo mismo y se comportan muy distinto. La segunda vuelve en la sesión B.

Igual número de a y de b

$$S \to aSb \mid bSa \mid SS \mid \varepsilon$$

terminal

```
$ build/show_cfg machines/equal_ab.cfg 4 | tail -2
Cadenas de L(G) con longitud a lo sumo 4 (9):
  ε ab ba aabb abab abba baab baba bbaa
```

- Cada regla agrega una `a` y una `b`, o ninguna: no genera de más.
- ¿Genera **todas**? `abba` sale de SS: `ab` seguida de `ba`. Que siempre se pueda partir hay que demostrarlo.
- Error común: olvidar el caso base. Con solo $S \to 0S1$ ninguna derivación termina: la gramática genera $\emptyset$.

Unir gramáticas

Si G_1 genera L_1 y G_2 genera L_2:

1. renombrar para que no compartan variables;
2. juntar todas las reglas;
3. agregar $S \to S_1 \mid S_2$.

- El primer paso elige con cuál sigue.
- Con $S \to S_1 S_2$: concatenación.
- Con $S \to S_1 S \mid \varepsilon$: estrella.

$\{a^i b^j c^k : i = j \text{ o } j = k\}$

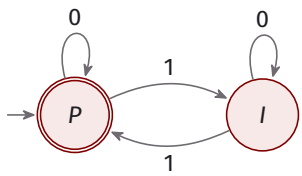
$$
\begin{aligned}
S &\to XC \mid AY \\
X &\to aXb \mid \varepsilon \\
C &\to cC \mid \varepsilon \\
A &\to aA \mid \varepsilon \\
Y &\to bYc \mid \varepsilon
\end{aligned}
$$

XC: las de $i = j$. AY: las de $j = k$.

De un DFA a una gramática

Una variable por estado.

- $\delta(q, a) = p$ da $R_q \to aR_p$
- $q \in F$ da $R_q \to \varepsilon$
- inicial: R_{q_0}



$$P \to 0P \mid 1I \mid \varepsilon$$
$$I \to 0I \mid 1P$$

terminal

```
$ build/derive machines/even_ones.cfg 1001 --sin-arboles
w = 1001
derivaciones por la izquierda: 1

1. P ⇒ 1I ⇒ 10I ⇒ 100I ⇒ 1001P ⇒ 1001
   5 pasos
```

- La derivación copia la ejecución: lo leído, y la variable del estado actual.
- **Todo lenguaje regular es libre de contexto.** La variable siempre queda al final: no empareja nada.

SECCIÓN 04

Demostrar que una gramática genera un lenguaje

04



Dos inclusiones

$L(G) = L$ es una igualdad de conjuntos.

$L \subseteq L(G)$

Toda cadena del lenguaje se deriva. La gramática no se queda corta.

Inducción sobre la **cadena**: se le construye una derivación.

$L(G) \subseteq L$

Toda cadena derivada está en el lenguaje. La gramática no genera de más.

Inducción sobre el **número de pasos**: se describen las formas.

Error común

Demostrar una sola. "Lo que genera tiene tantas $\mathtt{a}$ como $\mathtt{b}$" también vale para $S \to \mathtt{a}S\mathtt{b} \mid \varepsilon$.

$S \to 0S1 \mid \varepsilon$ genera exactamente $\{0^n 1^n\}$

Primera inclusión. Por inducción sobre n: $S \Rightarrow^* 0^n S 1^n$.

- Base: $S \Rightarrow^* S$ con cero pasos.
- Paso: $S \Rightarrow^* 0^n S 1^n \Rightarrow 0^{n+1} S 1^{n+1}$.
- Con $S \to \varepsilon$ al final: $S \Rightarrow^* 0^n 1^n$.

Segunda inclusión. Por inducción sobre k: si $S \Rightarrow^* u$ con k pasos, entonces

$$u = 0^k S 1^k \qquad \text{o} \qquad k \geq 1 \text{ y } u = 0^{k-1} 1^{k-1}.$$

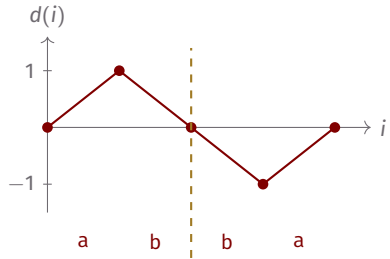
- Para dar un paso más, la forma tiene que tener una variable: es $0^k S 1^k$.
- Con $S \to 0S1$ queda $0^{k+1} S 1^{k+1}$. Con $S \to \varepsilon$ queda $0^k 1^k$.
- Una cadena sin variables es entonces $0^{k-1} 1^{k-1}$. □

Se describen **todas** las formas, como en un DFA se dice qué significa cada estado.

Igual número de a y de b: el caso difícil

Inducción fuerte sobre $|w|$, con $w \in L$ no vacía.

- Empieza con a y termina con b: $w = \mathtt{a}x\mathtt{b}$, y $S \Rightarrow \mathtt{a}S\mathtt{b} \Rightarrow^* w$.
- Al revés: con $S \to \mathtt{b}S\mathtt{a}$.
- **Mismo símbolo en los extremos.** $d(i)$: las a menos las b del prefijo de longitud i.
- $d(1) = 1$ y $d(|w| - 1) = -1$, con pasos de tamaño 1: pasa por 0 en medio.
- Ahí se parte: $w = xy$, las dos en L y más cortas. $S \Rightarrow SS \Rightarrow^* xy$.



Con abba: $0, 1, 0, -1, 0$.

SECCIÓN 05

Ambigüedad

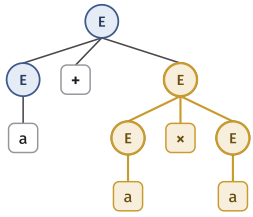
05

Dos árboles para a + a × a

$E \to E + E \mid E \times E \mid (E) \mid \mathrm{a}$ En los archivos el producto es $*$.

Árbol 1: la raíz es una suma

E ⇒ E+E ⇒ a+E ⇒ a+E×E ⇒ a+a×E ⇒ a+a×a



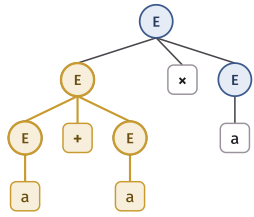
a + (a × a)

el operando derecho de la suma es el producto

con 2 + 3 × 4: 2 + 12 = 14

Árbol 2: la raíz es un producto

E ⇒ E×E ⇒ E+E×E ⇒ a+E×E ⇒ a+a×E ⇒ a+a×a



(a + a) × a

el operando izquierdo del producto es la suma

con 2 + 3 × 4: 5 × 4 = 20

Las hojas de los dos árboles dicen a+a×a. La cadena es la misma y la estructura no: la gramática es ambigua.

Gramática ambigua

Gramática ambigua *(ambiguous grammar)*

Genera alguna cadena con dos o más **árboles de derivación** distintos. Es lo mismo: con dos o más **derivaciones por la izquierda** distintas.

terminal

```
$ build/derive machines/expr_ambiguous.cfg 'a+a*a' --sin-arboles
w = a+a*a
derivaciones por la izquierda: 2

1. E ⇒ E+E ⇒ a+E ⇒ a+E*E ⇒ a+a*E ⇒ a+a*a
   5 pasos

2. E ⇒ E*E ⇒ E+E*E ⇒ a+E*E ⇒ a+a*E ⇒ a+a*a
   5 pasos
```

Dos derivaciones no es ser ambigua

Error común

Llamar ambigua a una gramática porque una cadena tiene dos derivaciones. Lo que cuenta son dos **árboles**, o dos derivaciones **por la izquierda**.

- Con $S \to AB$, $A \to \mathtt{a}$, $B \to \mathtt{b}$, la cadena $\mathtt{ab}$ tiene dos derivaciones y **un** árbol. No es ambigua.
- Para mostrar que es ambigua se exhibe una cadena y sus dos árboles.
- $\mathtt{a+a+a}$ también tiene dos: $E \to E + E$ no dice de qué lado se agrupa.

Idea clave

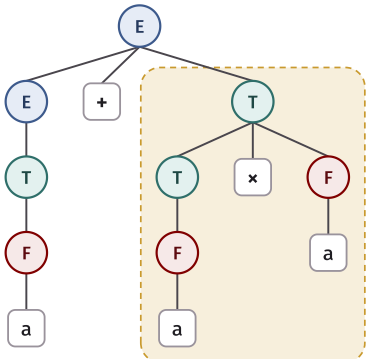
La ambigüedad es de la **gramática**, no del lenguaje. Y la gramática decide qué árbol tiene cada cadena: decide qué significa.

Niveles de precedencia

$$E \to E + T \mid T$$
$$T \to T \times F \mid F$$
$$F \to (E) \mid \mathrm{a}$$

- **Un nivel por operador.** Un producto solo tiene factores como operandos, y una suma no es un factor: eso da la precedencia.
- **Recursión de un solo lado.** $E \to E + T$: a la derecha de un $+$ no cabe otra suma. Eso da la asociatividad, por la izquierda.
- $E \to T$ y $T \to F$ son **reglas unitarias**. Vuelven en la sesión B.

El producto queda debajo de la suma



UN NIVEL POR OPERADOR

E → E + T | T
una expresión: una suma de términos

T → T × F | F
un término: un producto de factores

F → (E) | a
un factor: a, o una expresión entre ( )

El producto a × a es un solo término T: queda entero debajo de la suma.

E → T y T → F son reglas unitarias: bajan de nivel sin agregar símbolos, y el árbol es más alto.

E ⇒ E+T ⇒ T+T ⇒ F+T ⇒ a+T ⇒ a+T×F ⇒ a+F×F ⇒ a+a×F ⇒ a+a×a

La única derivación por la izquierda: 8 pasos, uno por cada nodo interno del árbol.

La misma cadena, un solo árbol

terminal

```
$ build/derive machines/expr_precedence.cfg 'a+a*a' --sin-arboles
w = a+a*a
derivaciones por la izquierda: 1

1. E ⇒ E+T ⇒ T+T ⇒ F+T ⇒ a+T ⇒ a+T*F ⇒ a+F*F ⇒ a+a*F ⇒ a+a*a
   8 pasos
```

- Las dos gramáticas son **equivalentes**: generan el mismo lenguaje.
- Cambia cuántos árboles tiene cada cadena.
- Las pruebas lo comprueban para todas las cadenas de hasta 7 símbolos. Eso no es una demostración.
- En la práctica: yacc y bison aceptan la gramática ambigua si se les declara aparte la precedencia y la asociatividad de cada operador.

El else colgante

$$S \to \texttt{i}S \mid \texttt{i}S\texttt{e}S \mid \texttt{a}$$

i: `if (c)`. e: `else`. a: otra instrucción.

iiaea

```
1  if (c1) if (c2) f(); else g();
```

terminal

```
$ build/derive machines/dangling_else.cfg iiaea --sin-arboles
w = iiaea
derivaciones por la izquierda: 2

1. S ⇒ iS ⇒ iiSeS ⇒ iiaeS ⇒ iiaea
   4 pasos

2. S ⇒ iSeS ⇒ iiSeS ⇒ iiaeS ⇒ iiaea
   4 pasos
```

¿De cuál `if` es el `else`?

- La 1: el `else` es del `if` de adentro. La 2: del de afuera. Son dos programas distintos.
- La convención: el `else` va con el `if` más cercano que todavía no tiene uno.

El else colgante, resuelto

$$S \to M \mid U$$
$$M \to \texttt{i}M\texttt{e}M \mid \texttt{a}$$
$$U \to \texttt{i}S \mid \texttt{i}M\texttt{e}U$$

- M: todo `if` tiene su `else`.
- U: algún `if` quedó sin `else`.
- Entre un `if` y su `else` va una M: nada suelto que pueda reclamar ese `else`.

$$S \Rightarrow U \Rightarrow \texttt{i}S \Rightarrow \texttt{i}M \Rightarrow \texttt{ii}M\texttt{e}M \Rightarrow \texttt{iiae}M \Rightarrow \texttt{iiaea}$$

La única derivación por la izquierda de `iiaea`.

En la práctica

Los lenguajes dejan la gramática ambigua y agregan la regla en palabras: el parser, si puede seguir leyendo un `else`, lo lee. Python lo evita con la indentación; Go y Rust, exigiendo llaves.

Lenguajes inherentemente ambiguos

Inherentemente ambiguo

Un lenguaje libre de contexto tal que **todas** las gramáticas que lo generan son ambiguas.

Existen. Un ejemplo, que se enuncia sin demostración: $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k : i = j \text{ o } j = k\}$.

- Con la gramática de la unión, $\mathtt{abc}$ tiene dos derivaciones por la izquierda, una por cada mitad:

$$S \Rightarrow XC \Rightarrow \mathtt{a}X\mathtt{b}C \Rightarrow \mathtt{ab}C \Rightarrow \mathtt{abc}C \Rightarrow \mathtt{abc}$$

$$S \Rightarrow AY \Rightarrow \mathtt{a}AY \Rightarrow \mathtt{a}Y \Rightarrow \mathtt{ab}Y\mathtt{c} \Rightarrow \mathtt{abc}$$

- El teorema dice que ninguna otra gramática lo arregla.
- Tampoco hay un algoritmo que decida si una gramática es ambigua. Llega en la semana 10.

SECCIÓN 06

El problema de la pertenencia

06

Sesión B: una sola pregunta

El problema de la pertenencia

Dadas una gramática G y una cadena w, ¿G genera w?

- Con un DFA: se sigue la única ejecución, un paso por símbolo.
- Con una gramática: $w \in L(G)$ si **existe** una derivación. La definición no dice cómo encontrarla.
- Primer intento: probar derivaciones, todas, hasta que alguna termine en w.

Es la pregunta de todo compilador

La gramática es la del lenguaje de programación, y w es el programa que acaban de escribir.

Buscar derivaciones puede no terminar

terminal

```
$ build/derive machines/parens_ss.cfg '()' --pasos 5 --sin-arboles | head -8
w = ()
derivaciones por la izquierda: 4 (la búsqueda se cortó: derivaciones de hasta 5 pasos, 20 como máximo)

1. S ⇒ SS ⇒ (S)S ⇒ ()S ⇒ ()
   4 pasos

2. S ⇒ SS ⇒ S ⇒ (S) ⇒ ()
   4 pasos
```

- $S \Rightarrow SS \Rightarrow S$: un rodeo que vuelve a donde empezó. Se puede repetir sin fin.
- Con una cadena que **no** se genera, la búsqueda nunca puede responder “no”.
- Los culpables: las **reglas** ε, que acortan, y las **reglas unitarias** $A \to B$, que no avanzan.

SECCIÓN 07

La forma normal de Chomsky

07

La forma normal de Chomsky

Forma normal de Chomsky *(Chomsky normal form)*

Cada regla tiene una de estas formas:

- $A \to BC$, con B y C variables, ninguna la variable inicial;
- $A \to a$, con a un terminal;
- $S \to \varepsilon$, solo para la variable inicial.

$\{\mathrm{a}^n\mathrm{b}^n : n \geq 0\}$:

$$S_0 \to AT \mid AB \mid \varepsilon$$
$$S \to AT \mid AB \qquad T \to SB$$
$$A \to \mathrm{a} \qquad B \to \mathrm{b}$$

- La inicial no aparece a la derecha: $S \to \varepsilon$ nunca se usa en medio de una derivación.
- Es para los **algoritmos** y las **demostraciones**: solo dos formas de regla.

Una derivación tiene exactamente $2n - 1$ pasos

Teorema

Si G está en forma normal de Chomsky y $w \in L(G)$ con $|w| = n \geq 1$, toda derivación de w tiene exactamente $2n - 1$ pasos.

Contando. Sean p los pasos con reglas $A \to BC$ y q los pasos con reglas $A \to a$.

- $A \to BC$ alarga la forma en un símbolo; $A \to a$ no cambia su longitud. De longitud 1 a longitud n: $p = n - 1$.
- $A \to a$ escribe un terminal, y un terminal no se borra. La cadena tiene n: $q = n$.
- Total: $(n - 1) + n = 2n - 1$. □

Consecuencia: alcanza con probar las derivaciones de $2n - 1$ pasos, que son finitas. La búsqueda **termina siempre**, y puede responder "no".

Termina, pero ¿cuánto tarda?

terminal

```
$ build/search_cost build/parens_ss_cnf.cfg '()((' '()()()((' '()()()()()((' '()()()()()()((' '()()()()()()()'
  n   respuesta    pasos probados   comprobaciones de CYK   cadena
  4   no genera                22                      80   ()((
  8   no genera               179                     672   ()()()((
 12   no genera              1535                    2288   ()()()()()((
 16   no genera             15496                    5440   ()()()()()()((
 16   genera                   62                    5440   ()()()()()()()
```

- La gramática: los paréntesis con $S \to SS$, convertida con `to_cnf`.
- Cada cuatro símbolos más, los pasos probados se multiplican por más de 8. Es **exponencial**.
- Con suerte, una cadena que sí se genera se encuentra pronto. El costo está en responder “no”.
- La columna de CYK crece mucho más despacio, y no depende de la respuesta.

SECCIÓN 08

La conversión en cuatro pasos

08

Cuatro pasos, en este orden

Teorema

Todo lenguaje libre de contexto es generado por una gramática en forma normal de Chomsky.

1. Agregar una **variable inicial nueva**.
2. Quitar las **reglas** ε.
3. Quitar las **reglas unitarias**.
4. **Partir las reglas largas** y sacar los terminales de las reglas de dos símbolos.

De cada paso: qué se construye, por qué el lenguaje no cambia, y qué deja garantizado.
El ejemplo: listas de bloques $\texttt{0}^n\texttt{1}^n$ con $n \geq 1$, como `0101` o `001101`.

$$S \to BS \mid \varepsilon \qquad\qquad B \to \texttt{0}B\texttt{1} \mid \texttt{01}$$

Paso 1: una variable inicial nueva

terminal

```
$ build/to_chomsky machines/blocks.cfg --pasos
La gramática original (4 reglas)
  S -> B S | eps
  B -> 0 B 1 | 0 1

Paso 1: variable inicial nueva (5 reglas)
  S0 -> S
  S -> B S | eps
  B -> 0 B 1 | 0 1
```

- **Qué:** una variable nueva S_0 con la única regla $S_0 \to S$.
- **Por qué no cambia el lenguaje:** toda derivación empieza con $S_0 \Rightarrow S$ y sigue como antes.
- **Qué garantiza:** la variable inicial no aparece a la derecha de ninguna regla.

Paso 2: quitar las reglas ε

Variable anulable

A es anulable si $A \Rightarrow^* \varepsilon$.

Se calculan marcando:

- $A \to \varepsilon$ marca a A;
- $A \to X_1 \cdots X_k$ con todas marcadas, también.

Aquí: S por $S \to \varepsilon$, y S_0 por $S_0 \to S$.

continúa

```
Paso 2: sin reglas ε (6 reglas)
  S0 -> S | eps
  S -> B S | B
  B -> 0 B 1 | 0 1
```

- **Qué:** por cada regla, todas las que salen de **borrar algunas apariciones de variables anulables**, salvo la que queda vacía. Si S_0 es anulable, se agrega $S_0 \to \varepsilon$.
- De $S \to BS$ salen $S \to BS$ y $S \to B$. La decisión de borrar la S se **adelanta**.
- Apareció una regla unitaria, $S \to B$, que antes no estaba.

Paso 2: por qué el lenguaje no cambia

Para toda variable A y toda $w \neq \varepsilon$: $A \Rightarrow^* w$ antes del paso si y solo si $A \Rightarrow^* w$ después, sin usar $S_0 \to \varepsilon$.

- **Lo nuevo no genera de más.** Una regla nueva es una vieja con variables anulables borradas. La gramática vieja hace lo mismo: usa la regla y después deriva cada una en ε.
- **Lo nuevo no se queda corto.** Inducción sobre el número de pasos. El primer paso usa $A \to X_1 \cdots X_k$, y $w = w_1 \cdots w_k$ por el lema de separación.
  - Si $w_i = \varepsilon$, X_i es anulable: se borra.
  - Si no, se deja, y $X_i \Rightarrow^* w_i$ en la nueva por la hipótesis.
  - Como $w \neq \varepsilon$, no se borra todo: lo que queda es una regla nueva.
- **La cadena vacía.** Sin $S_0 \to \varepsilon$ ninguna forma queda vacía. Y S_0 no está a la derecha: su regla ε solo sirve para $S_0 \Rightarrow \varepsilon$.

Paso 3: quitar las reglas unitarias

Par unitario

(A, B) tal que $A \Rightarrow^* B$ usando solo reglas unitarias. Todo (A, A) lo es.

continúa

```
Paso 3: sin reglas unitarias (9 reglas)
  S0 -> eps | B S | 0 B 1 | 0 1
  S -> B S | 0 B 1 | 0 1
  B -> 0 B 1 | 0 1
```

Aquí, además de los (A, A):
(S_0, S), (S, B) y (S_0, B).

- **Qué:** por cada par (A, B) y cada regla $B \to u$ **no unitaria**, la regla $A \to u$.
- **Por qué:** $A \Rightarrow^* B \Rightarrow u$ ya se podía. Y al revés, una cadena de reglas unitarias seguida de una regla no unitaria se reemplaza por una sola regla nueva.
- **Qué garantiza:** no hay unitarias, y **no aparecieron reglas** ε: solo se copian lados derechos que ya estaban.

Paso 4: partir las reglas largas

continúa

```
Paso 4: reglas de a lo sumo dos símbolos, y terminales solos (14 reglas)
  S0 -> eps | B S | U_0 S0_1 | U_0 U_1
  S -> B S | U_0 S_1 | U_0 U_1
  B -> U_0 B_1 | U_0 U_1
  S0_1 -> B U_1
  S_1 -> B U_1
  B_1 -> B U_1
  U_0 -> 0
  U_1 -> 1

Está en forma normal de Chomsky.
```

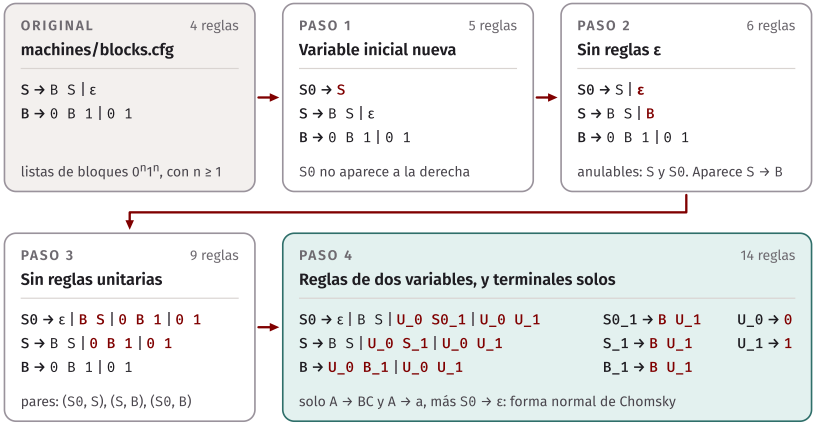
$A \to X_1 X_2 \cdots X_k$, $k \geq 3$:

$$A \to X_1 A_1$$
$$A_1 \to X_2 A_2$$
$$\cdots$$
$$A_{k-2} \to X_{k-1} X_k$$

A_i: lo que falta de la regla.

- Después, en las reglas de dos símbolos, cada terminal a se cambia por U_a, con $U_a \to a$.
- $B \to 0B1$ quedó como $B \to U_0 B_1$ y $B_1 \to BU_1$. De 2 variables y 4 reglas a 8 y 14.

Los cuatro pasos, juntos



Resaltado, lo que cada paso agrega. El lenguaje es el mismo en los cinco bloques.

El orden importa

Con $S \to AB \mid \mathtt{a}$, $A \to \mathtt{a} \mid \varepsilon$, $B \to \mathtt{b}$, haciendo el paso 3 **antes** que el paso 2:

terminal

```
$ build/to_chomsky machines/unit_order.cfg --unitarias-primero | tail -n 6
  S0 -> A B | B | a
  S -> A B | B | a
  A -> a
  B -> b

No está en forma normal de Chomsky. S0 -> B: es una regla unitaria.
```

Error común

Quitar las unitarias antes que las reglas ε. Quitar reglas ε **crea** unitarias: de $A \to BC$ con B anulable sale $A \to C$.

SECCIÓN 09

El algoritmo CYK

09



CYK: una tabla en lugar de una búsqueda

- Cocke, Younger y Kasami. Recibe G en forma normal de Chomsky y $w = w_0 w_1 \cdots w_{n-1}$.
- La búsqueda derivaba el mismo trozo de la cadena una y otra vez.
- Programación dinámica sobre intervalos, como en Algoritmos: se pregunta por **todas las subcadenas**, de las más cortas a las más largas.

La tabla

Para $0 \leq i \leq j < n$:

$$T[i][j] = \{ A \in V : A \Rightarrow^* w_i \cdots w_j \}$$

las variables que generan la subcadena de i a j.

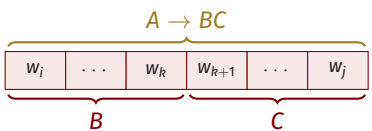
G genera w si y solo si la variable inicial está en $T[0][n-1]$.

La recurrencia

Teorema

Si G está en forma normal de Chomsky, $T[i][i] = \{ A : A \to w_i \text{ es una regla} \}$ y, para $i < j$,

$$T[i][j] = \bigcup_{k=i}^{j-1} \{ A : A \to BC \text{ es una regla}, \ B \in T[i][k], \ C \in T[k+1][j] \}.$$



- k es el **punto de corte**.
- Sin reglas ε: los dos trozos no son vacíos, y son **más cortos**.
- Dos variables a la derecha: alcanza **un** corte.

Por qué vale la recurrencia

Observación. Si B no es la variable inicial y $B \Rightarrow^* x$, entonces $x \neq \varepsilon$: la única regla que acorta es $S \to \varepsilon$, y S no aparece en ninguna forma derivada de B.

- **De derecha a izquierda.** Si $A \to BC$, $B \Rightarrow^* w_i \cdots w_k$ y $C \Rightarrow^* w_{k+1} \cdots w_j$, entonces $A \Rightarrow BC \Rightarrow^* w_i \cdots w_j$.
- **De izquierda a derecha.** Si $A \Rightarrow^* w_i \cdots w_j$ con $i < j$, el primer paso no es $S \to \varepsilon$ ni $A \to a$: es $A \to BC$.
- Por el lema de separación, la subcadena es xy con $B \Rightarrow^* x$ y $C \Rightarrow^* y$.
- Por la observación, x e y no son vacías: x termina en algún k con $i \leq k < j$. □

Las tres condiciones de la forma normal se usan

Sin ε: trozos más cortos. Sin unitarias: una celda no depende de sí misma. Dos variables: un solo corte.

El algoritmo

CYK, para n >= 1

```
algoritmo CYK(G, w)
    para i desde 0 hasta n - 1
        T[i][i] ← { A : A → w[i] es una regla de G }
    fin para
    para largo desde 2 hasta n
        para i desde 0 hasta n - largo
            j ← i + largo - 1
            T[i][j] ← ∅
            para k desde i hasta j - 1
                para cada regla A → B C de G
                    si B está en T[i][k] y C está en T[k+1][j] entonces
                        agregar A a T[i][j]
                    fin si
                fin para
            fin para
        fin para
    fin para
    devolver (S está en T[0][n-1])
fin algoritmo
```

- Si $n = 0$, antes de todo: devolver si $S \to \varepsilon$ es una regla.
- La tabla se llena **por longitudes crecientes**.
- Una celda consulta celdas de subcadenas más cortas, que ya están.
- Cuatro ciclos: largo, inicio, corte, regla.
- Error común: llenar por filas desde $i = 0$. $T[0][4]$ consulta $T[1][4]$, que está en una fila de abajo.

El ejemplo: la gramática

terminal

```
$ build/to_chomsky machines/expr_ambiguous.cfg
# machines/expr_ambiguous.cfg en forma normal de Chomsky.
variables: E0 E E0_1 E0_2 E0_3 E_1 E_2 E_3 U_+ U_* U_( U_)
terminals: a + * ( )
start: E0
rules:
E0 -> E E0_1 | E E0_2 | U_( E0_3 | a
E -> E E_1 | E E_2 | U_( E_3 | a
E0_1 -> U_+ E
E0_2 -> U_* E
E0_3 -> E U_)
E_1 -> U_+ E
E_2 -> U_* E
E_3 -> E U_)
U_+ -> +
U_* -> *
U_( -> (
U_) -> )
```

La gramática ambigua de expresiones, convertida.

- `E_1`: un $+$ y una expresión.
- `E_2`: un $\times$ y una expresión.
- `E_3`: una expresión y un `)`.
- `E0`: la inicial, con una copia de las reglas de `E`.

La cadena: `a+a*a`, con $n = 5$.

La tabla, llena a mano

		j = 0 a	j = 1 +	j = 2 a	j = 3 *	j = 4 a
i = 0	a	{E0,E}	∅	{E0,E}	∅	{E0,E}
i = 1	+		{U_+}	{E0_1,E_1}	∅	{E0_1,E_1}
i = 2	a			{E0,E}	∅	{E0,E}
i = 3	*				{U_*}	{E0_2,E_2}
i = 4	a					{E0,E}

Longitud 5

1 celda, 4 puntos de corte

Se llena por diagonales, hacia la esquina.

T[0][4], la subcadena a+a*a: se buscan reglas A → BC con B en la fila 0 y C en la columna 4.

k = 0	B en T[0][0], C en T[1][4]	**sirven E0 → E E0_1 y E → E E_1**
k = 1	B en T[0][1], C en T[2][4]	nada: T[0][1] es ∅
k = 2	B en T[0][2], C en T[3][4]	**sirven E0 → E E0_2 y E → E E_2**
k = 3	B en T[0][3], C en T[4][4]	nada: T[0][3] es ∅

T[0][4] = {E0, E}. La variable inicial E0 está en la esquina: la gramática genera a+a*a.

La misma tabla, con el programa

terminal

```
$ build/cyk machines/expr_ambiguous_chomsky.cfg 'a+a*a' | head -9
w = a+a*a, n = 5
        j=0 a       j=1 +       j=2 a       j=3 *       j=4 a
i=0  a {E0,E}       ∅           {E0,E}      ∅           {E0,E}
i=1  +              {U_+}       {E0_1,E_1}  ∅           {E0_1,E_1}
i=2  a                          {E0,E}      ∅           {E0,E}
i=3  *                                      {U_*}       {E0_2,E_2}
i=4  a                                                  {E0,E}
E0 está en T[0][4]: genera
comprobaciones de reglas A → BC: 240
```

- La diagonal: las subcadenas de un símbolo. Llenar por longitudes es llenar por diagonales.

Idea clave. CYK no busca una derivación. Calcula, para **cada** subcadena, **todas** las variables que la generan. Cada celda se calcula una vez.

Recuperar el árbol

- A entra a $T[i][j]$ por una regla $A \to BC$ y un corte k. Eso es un nodo: A, con un hijo B para $i..k$ y un hijo C para $k{+}1..j$.
- Se empieza en la inicial y $T[0][n-1]$, y se repite con cada hijo hasta la diagonal.
- `Eo` entra a $T[0][4]$ por **dos caminos**: $k = 0$ y $k = 2$. Son los dos árboles.
- La ambigüedad se ve en la tabla: una variable que entra a una celda de más de una manera.

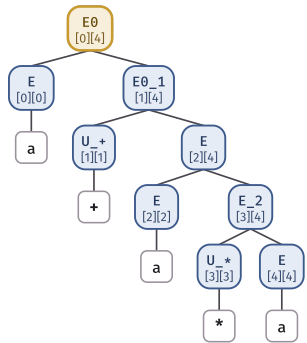
El programa informa `árboles de derivación: 2`: cuenta con la misma tabla.

El árbol del corte $k = 0$: a + (a × a)

LA TABLA, YA LLENA

	j = 0	j = 1	j = 2	j = 3	j = 4
	a	+	a	*	a
i = 0 a	E0 E	∅	E0 E	∅	**E0** E
i = 1 +		U_+	E0_1 E_1	∅	**E0_1** E_1
i = 2 a			E0 E	∅	E0 E
i = 3 *				U_*	E0_2 E_2
i = 4 a					E0 E

EL ÁRBOL QUE IMPRIME CYK



`E0` entra a `T[0][4]` por la regla `E0 → E E0_1` con el corte k = 0: su hijo `E` está en `T[0][0]` y su hijo `E0_1`, en `T[1][4]`.

Con el corte k = 2 habría salido el otro árbol.

Se empieza en la esquina y se repite con cada hijo en su celda, hasta la diagonal, donde el hijo es el símbolo w_i.

SECCIÓN 10

El costo de CYK

10



El costo

Sea $n = |w|$ y r el número de reglas $A \to BC$. La tabla tiene $n(n+1)/2$ celdas.

- **Cortes:** una celda de longitud ℓ tiene $\ell - 1$, y hay $n - \ell + 1$ celdas de esa longitud.

$$\sum_{\ell=1}^{n} (n - \ell + 1)(\ell - 1) = \binom{n+1}{3} = \frac{n^3 - n}{6}$$

- Elegir celda y corte es elegir $i \leq k < j$: tres números distintos $i < k{+}1 < j{+}1$ entre 0 y n.
- Por cada uno se revisan las r reglas. En el ejemplo: $20 \cdot 12 = 240$.

Costo de CYK

$O(n^3 \cdot |G|)$, si saber si una variable está en una celda cuesta tiempo constante. Convertir se hace una vez por gramática: **todo lenguaje libre de contexto está en P.**

Por qué hace falta la forma normal

- **Con** $A \to BCD$**:** hay que partir en tres trozos, con dos cortes. Del orden de n^2 pares por celda: el costo sube a n^4.
- **Con reglas** ε**:** un trozo podría ser vacío y el otro la subcadena entera. La celda dependería de sí misma.
- **Con una regla unitaria** $A \to B$**:** A entra a una celda porque B está en la misma celda. Otra vez.

En la práctica

El paso 2 puede multiplicar las reglas: una regla con k variables anulables tiene 2^k maneras de borrar. Un programa para gramáticas grandes parte primero las reglas largas, y entonces cada regla deja a lo sumo tres.

Por qué los compiladores no usan CYK

- $O(n^3)$ es polinomial, y sigue siendo demasiado: un archivo tiene decenas de miles de tokens. Con $n = 10\,000$, $n^3 = 10^{12}$.
- Un compilador no recibe una gramática cualquiera. La de su lenguaje se **diseña** para leer la entrada una vez, de izquierda a derecha, decidiendo con el siguiente token.
- Para esas gramáticas hay parsers **lineales**, $O(n)$.
- Los niveles de precedencia y la regla del `else` son parte de ese diseño.

Dónde sí

Cuando la gramática no se puede elegir: gramáticas ambiguas por naturaleza, como las de los idiomas, y herramientas que aceptan cualquier gramática.

SECCIÓN 11

Cierre

11

Laboratorio: un parser con CYK

1. Leer una gramática en forma normal de Chomsky y una lista de cadenas.
2. Llenar la tabla por longitudes crecientes y responder `genera` o `no genera`.
3. Para las que se generan, imprimir un árbol.
4. Pasar los casos de borde: la cadena vacía, una gramática que no genera nada, variables sin reglas.
5. Convertir a mano los paréntesis balanceados y probarlos.

Dónde: `code/week-05/lab/cyk_parser`. El árbol de `ab` se escribe `[S0 [A a] [B b]]`. Hay varios árboles correctos: el juez usa un **verificador**.

`casos/01.in`, sin el comentario

```
variables: S0 S T A B
terminals: a b
start: S0
rules:
S0 -> A T | A B | eps
S -> A T | A B
T -> S B
A -> a
B -> b
strings: 4
aabb

aab
ab
```

Resumen

- Una gramática $G = (V, \Sigma, R, S)$ **genera** un lenguaje: $L(G) = \{w : S \Rightarrow^* w\}$.
- El **árbol de derivación** es la estructura de la cadena. Un árbol, una derivación por la izquierda.
- $L(G) = L$ se demuestra con **dos inclusiones**, cada una por inducción.
- **Ambigua**: alguna cadena con dos árboles. Es una propiedad de la gramática.
- **Forma normal de Chomsky**: $A \to BC$, $A \to a$ y $S \to \varepsilon$. Una derivación tiene $2n - 1$ pasos.
- **Cuatro pasos, en orden**: inicial nueva, reglas ε, unitarias, reglas largas.
- **CYK**: $T[i][j]$, las variables que generan la subcadena de i a j. Costo $O(n^3 \cdot |G|)$.

Para la próxima clase

- Leer los apuntes de la semana 5 y terminar el laboratorio.
- Preguntas de repaso de la guía: una gramática para $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k : i = j \text{ o } j = k\}$; convertir $S \to \mathtt{a}S\mathtt{b} \mid SS \mid \varepsilon$ con los cuatro pasos; llenar una tabla de CYK de cuatro símbolos.
- Próxima semana: la máquina que hay detrás de las gramáticas, el autómata de pila, y lo que las gramáticas libres de contexto no pueden generar.

Pregunta

Un DFA no reconoce $\{\mathtt{0}^n\mathtt{1}^n\}$ porque no puede contar. ¿Qué es lo mínimo que habría que agregarle para que pueda? ¿Alcanzaría eso para $\{\mathtt{0}^n\mathtt{1}^n\mathtt{2}^n\}$?