

UNIVERSIDAD NACIONAL DE INGENIERÍA

Teoría de la Computación

Autómatas de pila y los límites de las gramáticas

PDA, equivalencia con las gramáticas, lema de bombeo y propiedades de cierre

Gerson Garrido

Semana 6

Hoy

Sesión A

1. Un autómata finito con una pila
2. Definición formal y configuraciones
3. Diseñar un PDA
4. De gramática a PDA
5. De PDA a gramática: las A_{pq}
6. El PDA determinista reconoce menos

Sesión B

7. Comparar tres cantidades
8. El lema de bombeo
9. Usar el lema: el juego y los casos
10. Propiedades de cierre
11. El mapa hasta ahora
12. Opcional: un parser real

> **Lectura**
>
> Apuntes de la semana 6. Apoyo: Sipser, sec. 2.2 (sesión A) y 2.3 (sesión B); sec. 2.4 solo como consulta para la parte opcional. Código en `code/week-06`.

SECCIÓN 01

Un autómata finito con una pila

01

De dónde venimos

- Semana 5: las **gramáticas libres de contexto** generan lenguajes que ningún DFA reconoce: $\{0^n 1^n\}$, paréntesis, expresiones.

De dónde venimos

- Semana 5: las **gramáticas libres de contexto** generan lenguajes que ningún DFA reconoce: $\{0^n 1^n\}$, paréntesis, expresiones.
- Pero una gramática *genera*. Para *decidir* si una cadena está hubo que usar CYK.

De dónde venimos

- Semana 5: las **gramáticas libres de contexto** generan lenguajes que ningún DFA reconoce: $\{0^n 1^n\}$, paréntesis, expresiones.
- Pero una gramática *genera*. Para *decidir* si una cadena está hubo que usar CYK.
- Los regulares tenían dos caras: la expresión regular describe, el autómata lee y responde.

De dónde venimos

- Semana 5: las **gramáticas libres de contexto** generan lenguajes que ningún DFA reconoce: $\{0^n 1^n\}$, paréntesis, expresiones.
- Pero una gramática *genera*. Para *decidir* si una cadena está hubo que usar CYK.
- Los regulares tenían dos caras: la expresión regular describe, el autómata lee y responde.

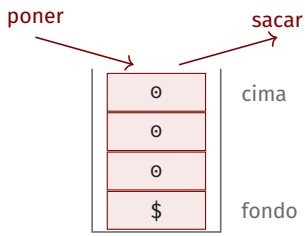
El plan de la semana

Sesión A: la máquina de los lenguajes libres de contexto, y la demostración de que es exactamente tan capaz como las gramáticas. Sesión B: su límite.

Lo que le faltaba al autómata finito

Un DFA no reconoce $\{0^n 1^n\}$: no tiene dónde anotar cuántos ceros leyó.

Le damos una **pila** *(stack)*:

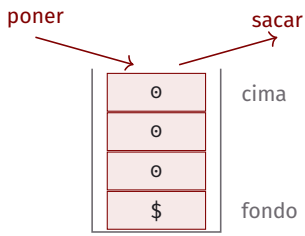


Lo que le faltaba al autómata finito

Un DFA no reconoce $\{0^n 1^n\}$: no tiene dónde anotar cuántos ceros leyó.

Le damos una **pila** *(stack)*:

- **No tiene límite de tamaño.** Es lo que faltaba.

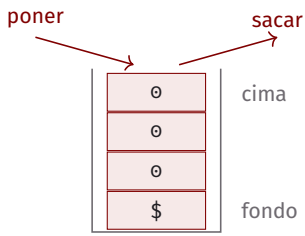


Lo que le faltaba al autómata finito

Un DFA no reconoce $\{0^n 1^n\}$: no tiene dónde anotar cuántos ceros leyó.

Le damos una **pila** *(stack)*:

- **No tiene límite de tamaño.** Es lo que faltaba.
- **Solo se ve la cima.** Para mirar más abajo hay que sacar, y lo que se saca se pierde.

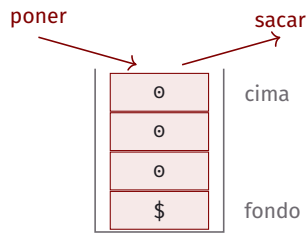


Lo que le faltaba al autómata finito

Un DFA no reconoce $\{0^n 1^n\}$: no tiene dónde anotar cuántos ceros leyó.

Le damos una **pila** *(stack)*:

- **No tiene límite de tamaño.** Es lo que faltaba.
- **Solo se ve la cima.** Para mirar más abajo hay que sacar, y lo que se saca se pierde.



Autómata de pila *(pushdown automaton)*, PDA

Un autómata finito con una pila. Decide con tres datos: el estado, el siguiente símbolo y la cima.

$\{0^n 1^n\}$ con una pila

1. Mientras lleguen ceros, poner cada uno en la pila.
2. Por cada 1 que se lee, sacar un 0.
3. Aceptar si la entrada termina justo cuando la pila queda vacía.

$\{0^n 1^n\}$ con una pila

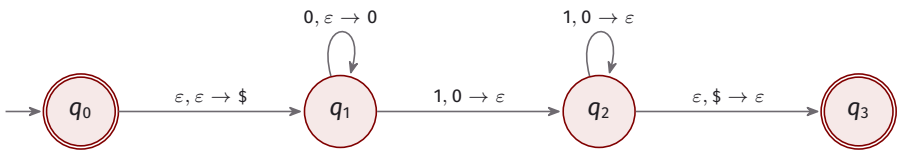
1. Mientras lleguen ceros, poner cada uno en la pila.
2. Por cada 1 que se lee, sacar un 0.
3. Aceptar si la entrada termina justo cuando la pila queda vacía.

Problema: la máquina no puede ver que la pila está vacía.

$\{0^n 1^n\}$ con una pila

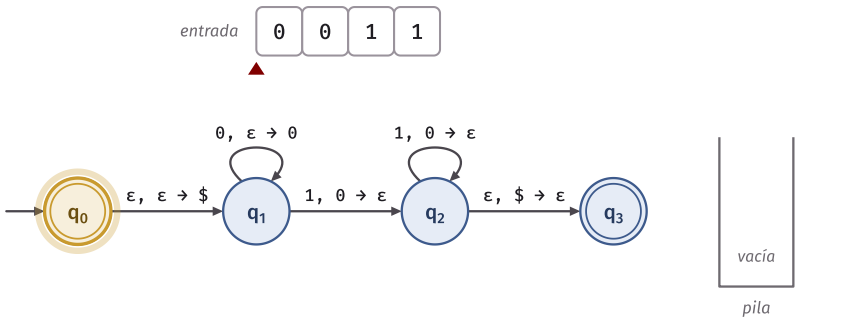
1. Mientras lleguen ceros, poner cada uno en la pila.
2. Por cada 1 que se lee, sacar un 0.
3. Aceptar si la entrada termina justo cuando la pila queda vacía.

Problema: la máquina no puede ver que la pila está vacía. **Solución:** marcar el fondo con \$.



Etiqueta de una flecha: lee, saca → pone.

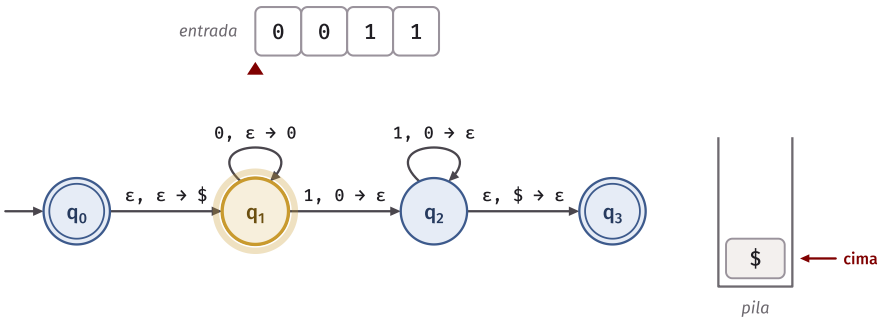
La máquina, paso a paso, con 0011



configuración $(q_0, 0011, \varepsilon)$ (estado, falta leer, pila)

Configuración inicial: estado q_0, toda la entrada por leer y la pila vacía.

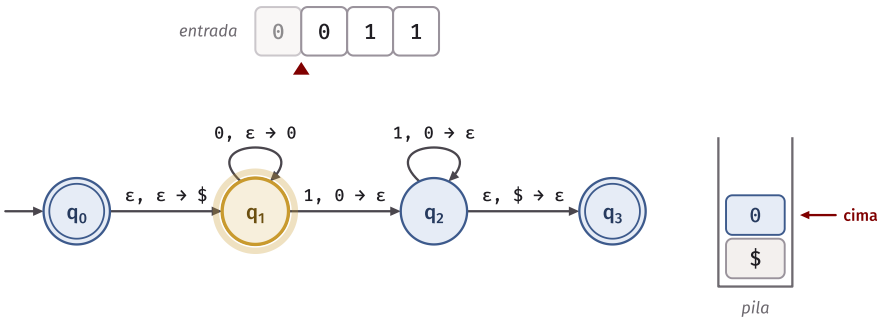
La máquina, paso a paso, con `0011`



configuración $(q_1, 0011, \$)$ (estado, falta leer, pila)

Paso 1: sin leer nada, pone $. Es la marca del fondo de la pila.

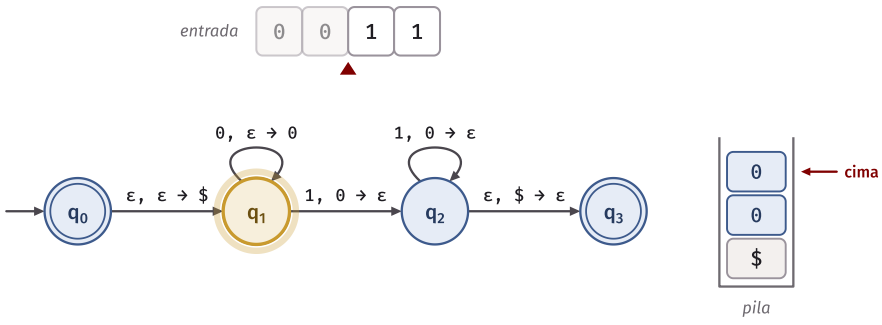
La máquina, paso a paso, con 0011



configuración $(q_1, 011, 0\$)$ (estado, falta leer, pila)

Paso 2: lee un `0` y lo pone en la pila.

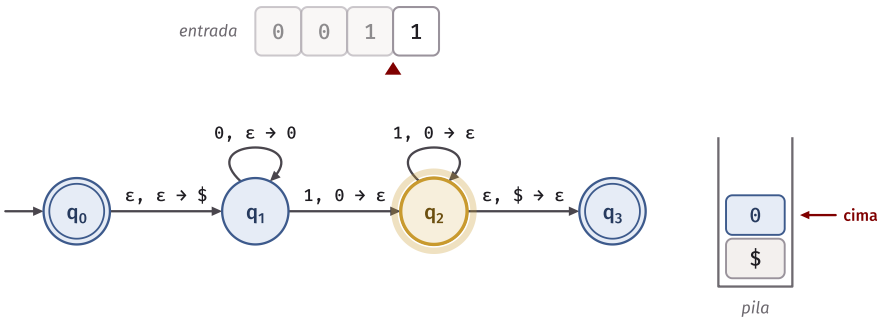
La máquina, paso a paso, con 0011



configuración $(q_1, 11, 00\$)$ (estado, falta leer, pila)

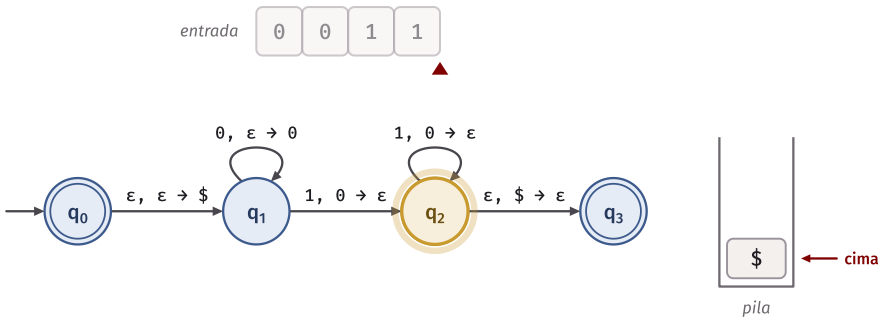
Paso 3: lee el otro `0` y lo pone encima. La pila es `00$`.

La máquina, paso a paso, con 0011



Paso 4: lee el primer 1 y saca un 0. Pasa a q_2: desde aquí solo saca.

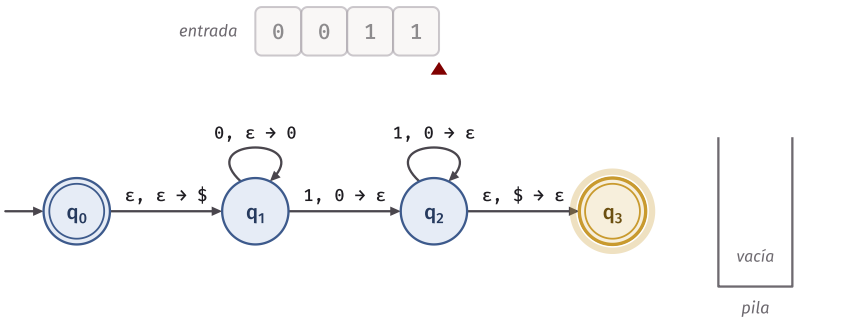
La máquina, paso a paso, con `0011`



configuración $(q_2, \varepsilon, \$)$ (estado, falta leer, pila)

Paso 5: lee el segundo `1` y saca el otro `0`. La cima vuelve a ser `$`.

La máquina, paso a paso, con `0011`



Paso 6: sin leer nada, saca $. No falta leer nada y q_3 es de aceptación: acepta.

La ejecución con `0011`

terminal

```
$ build/run_pda machines/zeros_ones.pda 0011 '' 001
w = 0011
  paso  transición  estado  falta leer  pila
     0              q0      0011        ε
     1  ε, ε → $    q1      0011        $
     2  0, ε → 0    q1      011         0$
     3  0, ε → 0    q1      11          00$
     4  1, 0 → ε    q2      1           0$
     5  1, 0 → ε    q2      ε           $
     6  ε, $ → ε    q3      ε           ε
  acepta en q3 (configuraciones exploradas: 7)
w = ε
  paso  transición  estado  falta leer  pila
     0              q0      ε           ε
  acepta en q0 (configuraciones exploradas: 1)
w = 001
  rechaza: 5 configuraciones alcanzables, ninguna acepta
```

- La pila se escribe con la **cima a la izquierda**.

La ejecución con `0011`

terminal

```
$ build/run_pda machines/zeros_ones.pda 0011 '' 001
w = 0011
  paso  transición  estado  falta leer  pila
     0              q0      0011        ε
     1  ε, ε → $    q1      0011        $
     2  0, ε → 0    q1      011         0$
     3  0, ε → 0    q1      11          00$
     4  1, 0 → ε    q2      1           0$
     5  1, 0 → ε    q2      ε           $
     6  ε, $ → ε    q3      ε           ε
  acepta en q3 (configuraciones exploradas: 7)
w = ε
  paso  transición  estado  falta leer  pila
     0              q0      ε           ε
  acepta en q0 (configuraciones exploradas: 1)
w = 001
  rechaza: 5 configuraciones alcanzables, ninguna acepta
```

- La pila se escribe con la **cima a la izquierda**.
- Fila 5: la cima es $. Recién ahí puede pasar a q_3.

La ejecución con `0011`

terminal

```
$ build/run_pda machines/zeros_ones.pda 0011 '' 001
w = 0011
  paso  transición  estado  falta leer  pila
     0              q0      0011        ε
     1  ε, ε → $    q1      0011        $
     2  0, ε → 0    q1      011         0$
     3  0, ε → 0    q1      11          00$
     4  1, 0 → ε    q2      1           0$
     5  1, 0 → ε    q2      ε           $
     6  ε, $ → ε    q3      ε           ε
  acepta en q3 (configuraciones exploradas: 7)
w = ε
  paso  transición  estado  falta leer  pila
     0              q0      ε           ε
  acepta en q0 (configuraciones exploradas: 1)
w = 001
  rechaza: 5 configuraciones alcanzables, ninguna acepta
```

- La pila se escribe con la **cima a la izquierda**.
- Fila 5: la cima es `$`. Recién ahí puede pasar a q_3.
- Con `001` termina en q_2 con `0$`: sobra un cero, y no hay transición.

SECCIÓN 02

Definición formal y configuraciones

02

La definición formal

Autómata de pila

Una séxtupla $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$:

1. Q, un conjunto finito de **estados**;
2. Σ, el **alfabeto de entrada**;
3. Γ, el **alfabeto de la pila**;
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$, la **función de transición**;
5. $q_0 \in Q$, el estado inicial; 6. $F \subseteq Q$, los estados de aceptación.

- $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$: un símbolo, o ninguno.

La definición formal

Autómata de pila

Una séxtupla $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$:

1. Q, un conjunto finito de **estados**;
2. Σ, el **alfabeto de entrada**;
3. Γ, el **alfabeto de la pila**;
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$, la **función de transición**;
5. $q_0 \in Q$, el estado inicial; 6. $F \subseteq Q$, los estados de aceptación.

- $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$: un símbolo, o ninguno.
- Es un NFA con dos agregados: Γ, y una coordenada más en la entrada y en la salida de δ.

La definición formal

Autómata de pila

Una séxtupla $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$:

1. Q, un conjunto finito de **estados**;
2. Σ, el **alfabeto de entrada**;
3. Γ, el **alfabeto de la pila**;
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$, la **función de transición**;
5. $q_0 \in Q$, el estado inicial; 6. $F \subseteq Q$, los estados de aceptación.

- $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$: un símbolo, o ninguno.
- Es un NFA con dos agregados: Γ, y una coordenada más en la entrada y en la salida de δ.
- El resultado de δ es un **conjunto**: el PDA es no determinista por definición.

Leer δ

$$(r, c) \in \delta(q, a, b)$$

En q, leyendo a y sacando b, **puede** pasar a r poniendo c.

$\varepsilon \to c$	pone c
$b \to \varepsilon$	saca b: exige que b esté en la cima
$b \to c$	reemplaza la cima b por c
$\varepsilon \to \varepsilon$	no toca la pila

zeros_ones.pda

```
# { 0^n 1^n : n >= 0 }
states: q0 q1 q2 q3
alphabet: 0 1
stack: 0 $
start: q0
accept: q0 q3
delta:
q0 eps eps -> q1 $
q1 0 eps -> q1 0
q1 1 0 -> q2 eps
q2 1 0 -> q2 eps
q2 eps $ -> q3 eps
```

```
estado lee saca -> destino pone
```

Leer δ

$$(r, c) \in \delta(q, a, b)$$

En q, leyendo a y sacando b, **puede** pasar a r poniendo c.

$\varepsilon \to c$	pone c
$b \to \varepsilon$	saca b: exige que b esté en la cima
$b \to c$	reemplaza la cima b por c
$\varepsilon \to \varepsilon$	no toca la pila

- Sacar ε se puede con cualquier cima, también con la pila vacía.

zeros_ones.pda

```
# { 0^n 1^n : n >= 0 }
states: q0 q1 q2 q3
alphabet: 0 1
stack: 0 $
start: q0
accept: q0 q3
delta:
q0 eps eps -> q1 $
q1 0 eps -> q1 0
q1 1 0 -> q2 eps
q2 1 0 -> q2 eps
q2 eps $ -> q3 eps
```

`estado lee saca -> destino pone`

Leer δ

$$(r, c) \in \delta(q, a, b)$$

En q, leyendo a y sacando b, **puede** pasar a r poniendo c.

$\varepsilon \to c$	pone c
$b \to \varepsilon$	saca b: exige que b esté en la cima
$b \to c$	reemplaza la cima b por c
$\varepsilon \to \varepsilon$	no toca la pila

- Sacar ε se puede con cualquier cima, también con la pila vacía.
- Una transición pone **a lo sumo un símbolo**. Para poner dos: un estado intermedio.

zeros_ones.pda

```
# { 0^n 1^n : n >= 0 }
states: q0 q1 q2 q3
alphabet: 0 1
stack: 0 $
start: q0
accept: q0 q3
delta:
q0 eps eps -> q1 $
q1 0 eps -> q1 0
q1 1 0 -> q2 eps
q2 1 0 -> q2 eps
q2 eps $ -> q3 eps
```

`estado lee saca -> destino pone`

Configuraciones

Configuración *(configuration)*

Una terna (q, x, γ): el estado, lo que **falta leer** y la pila, con la cima a la izquierda. La inicial con la entrada w es (q_0, w, ε).

Configuraciones

Configuración *(configuration)*

Una terna (q, x, γ): el estado, lo que **falta leer** y la pila, con la cima a la izquierda. La inicial con la entrada w es (q_0, w, ε).

$$(q, ax, b\gamma) \vdash (r, x, c\gamma) \quad \text{si } (r, c) \in \delta(q, a, b) \qquad a \in \Sigma_\varepsilon,\ b, c \in \Gamma_\varepsilon$$

Configuraciones

Configuración *(configuration)*

Una terna (q, x, γ): el estado, lo que **falta leer** y la pila, con la cima a la izquierda. La inicial con la entrada w es (q_0, w, ε).

$$(q, ax, b\gamma) \vdash (r, x, c\gamma) \quad \text{si } (r, c) \in \delta(q, a, b) \qquad a \in \Sigma_\varepsilon,\ b, c \in \Gamma_\varepsilon$$

Aceptar

P **acepta** w si $(q_0, w, \varepsilon) \vdash^* (q, \varepsilon, \gamma)$ para algún $q \in F$ y alguna pila γ.

Configuraciones

Configuración *(configuration)*

Una terna (q, x, γ): el estado, lo que **falta leer** y la pila, con la cima a la izquierda. La inicial con la entrada w es (q_0, w, ε).

$$(q, ax, b\gamma) \vdash (r, x, c\gamma) \quad \text{si } (r, c) \in \delta(q, a, b) \qquad a \in \Sigma_\varepsilon,\ b, c \in \Gamma_\varepsilon$$

Aceptar

P **acepta** w si $(q_0, w, \varepsilon) \vdash^* (q, \varepsilon, \gamma)$ para algún $q \in F$ y alguna pila γ.

$$(q_0, \texttt{0011}, \varepsilon) \vdash (q_1, \texttt{0011}, \$) \vdash (q_1, \texttt{011}, \texttt{0}\$) \vdash (q_1, \texttt{11}, \texttt{00}\$) \vdash (q_2, \texttt{1}, \texttt{0}\$) \vdash (q_2, \varepsilon, \$) \vdash (q_3, \varepsilon, \varepsilon)$$

Cuatro cosas que dice la definición

- **Basta una ejecución.** Como en un NFA: acepta si *alguna* secuencia de elecciones termina bien.

Cuatro cosas que dice la definición

- **Basta una ejecución.** Como en un NFA: acepta si *alguna* secuencia de elecciones termina bien.
- **Hay que leer toda la entrada.** Pasar por un estado de aceptación a mitad de camino no cuenta.

Cuatro cosas que dice la definición

- **Basta una ejecución.** Como en un NFA: acepta si *alguna* secuencia de elecciones termina bien.
- **Hay que leer toda la entrada.** Pasar por un estado de aceptación a mitad de camino no cuenta.
- **La pila puede quedar con cualquier cosa.** Se acepta por estado. Vaciarla es una costumbre.

Cuatro cosas que dice la definición

- **Basta una ejecución.** Como en un NFA: acepta si *alguna* secuencia de elecciones termina bien.
- **Hay que leer toda la entrada.** Pasar por un estado de aceptación a mitad de camino no cuenta.
- **La pila puede quedar con cualquier cosa.** Se acepta por estado. Vaciarla es una costumbre.
- **Un camino puede morir**: la transición pide sacar un símbolo y la cima es otro, o la pila está vacía.

Cuatro cosas que dice la definición

- **Basta una ejecución.** Como en un NFA: acepta si *alguna* secuencia de elecciones termina bien.
- **Hay que leer toda la entrada.** Pasar por un estado de aceptación a mitad de camino no cuenta.
- **La pila puede quedar con cualquier cosa.** Se acepta por estado. Vaciarla es una costumbre.
- **Un camino puede morir**: la transición pide sacar un símbolo y la cima es otro, o la pila está vacía.

Error común

"Acepta `00110` porque llega a q_3." Llega con un símbolo sin leer: $(q_3, \texttt{0}, \varepsilon)$. Desde ahí no hay transiciones, y no hay otro camino.

Lo que está debajo no participa

Un paso solo mira el primer símbolo que falta leer y la cima de la pila.

Lema del fondo

Si $(q, x, \alpha) \vdash^* (r, y, \beta)$, entonces para toda cadena z y toda pila γ:

$$(q, xz, \alpha\gamma) \vdash^* (r, yz, \beta\gamma)$$

Lo que está debajo no participa

Un paso solo mira el primer símbolo que falta leer y la cima de la pila.

Lema del fondo

Si $(q, x, \alpha) \vdash^* (r, y, \beta)$, entonces para toda cadena z y toda pila γ:

$$(q, xz, \alpha\gamma) \vdash^* (r, yz, \beta\gamma)$$

- Toda ejecución se puede repetir “más arriba”: encima de una pila que ya tenía cosas, y con más entrada esperando detrás.

Lo que está debajo no participa

Un paso solo mira el primer símbolo que falta leer y la cima de la pila.

Lema del fondo

Si $(q, x, \alpha) \vdash^* (r, y, \beta)$, entonces para toda cadena z y toda pila γ:

$$(q, xz, \alpha\gamma) \vdash^* (r, yz, \beta\gamma)$$

- Toda ejecución se puede repetir “más arriba”: encima de una pila que ya tenía cosas, y con más entrada esperando detrás.
- Al revés no siempre: una ejecución que saca un símbolo de γ no se puede repetir sin γ.
- Se usa en las dos demostraciones grandes de hoy.

SECCIÓN 03

Diseñar un PDA

03

Diseñar un PDA: repartir la memoria

En el estado: poco, y siempre a la vista.
En la pila: mucho, pero sale en orden inverso.

El plan de casi todos los diseños:

1. marcar el fondo;
2. una fase que guarda, otra que saca y compara;
3. el estado recuerda la fase;
4. al final, la pila vuelve a la marca.

Diseñar un PDA: repartir la memoria

En el estado: poco, y siempre a la vista.
En la pila: mucho, pero sale en orden inverso.

El plan de casi todos los diseños:

1. marcar el fondo;
2. una fase que guarda, otra que saca y compara;
3. el estado recuerda la fase;
4. al final, la pila vuelve a la marca.

Por qué la marca $

δ recibe un símbolo de Γ_ε. Con b, exige que b esté en la cima. Con ε, no exige nada.

No existe una coordenada que diga "la pila está vacía".

Diseñar un PDA: repartir la memoria

En el estado: poco, y siempre a la vista.
En la pila: mucho, pero sale en orden inverso.

El plan de casi todos los diseños:

1. marcar el fondo;
2. una fase que guarda, otra que saca y compara;
3. el estado recuerda la fase;
4. al final, la pila vuelve a la marca.

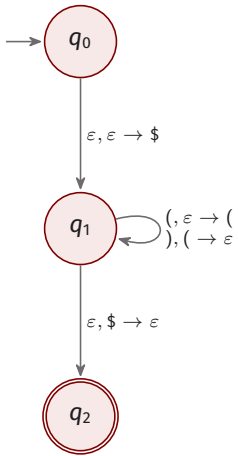
Por qué la marca $

δ recibe un símbolo de Γ_ε. Con b, exige que b esté en la cima. Con ε, no exige nada.

No existe una coordenada que diga "la pila está vacía".

La marca convierte "está vacía" en una pregunta que sí se puede hacer: "la cima es $".

Paréntesis balanceados



terminal

```
$ build/run_pda machines/balanced.pda '(())()'
w = (())()
  paso  transición  estado  falta leer  pila
     0              q0      (())()      ε
     1  ε, ε → $    q1      (())()      $
     2  (, ε → (    q1      ())()       ($
     3  (, ε → (    q1      ))()        (($
     4  ), ( → ε    q1      )()         ($
     5  ), ( → ε    q1      ()          $
     6  (, ε → (    q1      )           ($
     7  ), ( → ε    q1      ε           $
     8  ε, $ → ε    q2      ε           ε
  acepta en q2 (configuraciones exploradas: 11)
```

La altura de la pila es la cantidad de paréntesis abiertos. En la fila 5 *puede* saltar a q_2: ese camino muere.

Palíndromos de longitud par: $\{ ww^R \}$

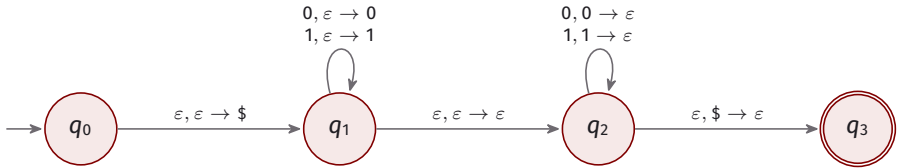
- Guardar la primera mitad; después sacar comparando. La pila devuelve al revés: justo lo que hace falta.

Palíndromos de longitud par: $\{ ww^R \}$

- Guardar la primera mitad; después sacar comparando. La pila devuelve al revés: justo lo que hace falta.
- ¿Cuándo termina la primera mitad? Nada en la entrada lo dice.

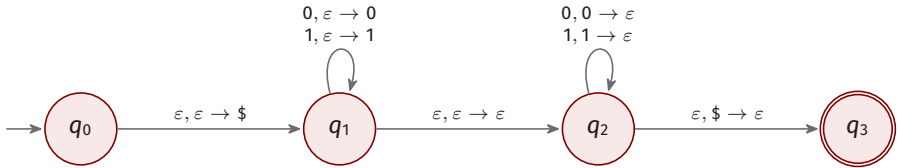
Palíndromos de longitud par: $\{ ww^R \}$

- Guardar la primera mitad; después sacar comparando. La pila devuelve al revés: justo lo que hace falta.
- ¿Cuándo termina la primera mitad? Nada en la entrada lo dice.
- La máquina lo **adivina**: en cualquier momento, sin leer nada, salta de q_1 a q_2.



Palíndromos de longitud par: $\{ ww^R \}$

- Guardar la primera mitad; después sacar comparando. La pila devuelve al revés: justo lo que hace falta.
- ¿Cuándo termina la primera mitad? Nada en la entrada lo dice.
- La máquina lo **adivina**: en cualquier momento, sin leer nada, salta de q_1 a q_2.



Adivinar no es suerte

Si la cadena es ww^R, la ejecución que salta en el medio *existe*. Si no lo es, *ningún* salto sirve.

La ejecución con `0110`

terminal

```
$ build/run_pda machines/even_palindromes.pda 0110 011
w = 0110
  paso  transición  estado  falta leer  pila
     0              q0      0110        ε
     1  ε, ε → $    q1      0110        $
     2  0, ε → 0    q1      110         0$
     3  1, ε → 1    q1      10          10$
     4  ε, ε → ε    q2      10          10$
     5  1, 1 → ε    q2      0           0$
     6  0, 0 → ε    q2      ε           $
     7  ε, $ → ε    q3      ε           ε
  acepta en q3 (configuraciones exploradas: 15)
w = 011
  rechaza: 11 configuraciones alcanzables, ninguna acepta
```

- Paso 4: adivina el medio, después de dos símbolos.

La ejecución con `0110`

terminal

```
$ build/run_pda machines/even_palindromes.pda 0110 011
w = 0110
  paso  transición  estado  falta leer  pila
     0              q0      0110        ε
     1  ε, ε → $    q1      0110        $
     2  0, ε → 0    q1      110         0$
     3  1, ε → 1    q1      10          10$
     4  ε, ε → ε    q2      10          10$
     5  1, 1 → ε    q2      0           0$
     6  0, 0 → ε    q2      ε           $
     7  ε, $ → ε    q3      ε           ε
  acepta en q3 (configuraciones exploradas: 15)
w = 011
  rechaza: 11 configuraciones alcanzables, ninguna acepta
```

- Paso 4: adivina el medio, después de dos símbolos.
- Exploró **15** configuraciones: también probó saltar en los otros momentos.

La ejecución con `0110`

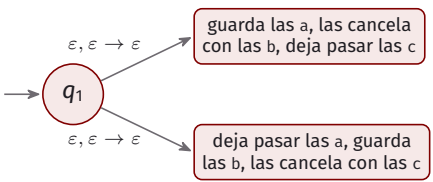
terminal

```
$ build/run_pda machines/even_palindromes.pda 0110 011
w = 0110
  paso  transición  estado  falta leer  pila
     0              q0      0110        ε
     1  ε, ε → $    q1      0110        $
     2  0, ε → 0    q1      110         0$
     3  1, ε → 1    q1      10          10$
     4  ε, ε → ε    q2      10          10$
     5  1, 1 → ε    q2      0           0$
     6  0, 0 → ε    q2      ε           $
     7  ε, $ → ε    q3      ε           ε
  acepta en q3 (configuraciones exploradas: 15)
w = 011
  rechaza: 11 configuraciones alcanzables, ninguna acepta
```

- Paso 4: adivina el medio, después de dos símbolos.
- Exploró **15** configuraciones: también probó saltar en los otros momentos.
- Si salta después de un símbolo: $(q_2, \texttt{110}, \texttt{0\$})$. Lee 1, la cima es 0: muere.

Elegir qué comparar

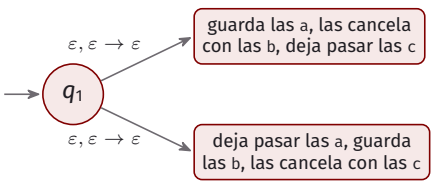
$$C = \{ a^i b^j c^k : i = j \text{ o } j = k \}$$



- Al empezar, sin leer nada, **elige** una rama.

Elegir qué comparar

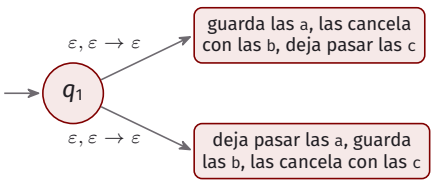
$$C = \{ a^i b^j c^k : i = j \text{ o } j = k \}$$



- Al empezar, sin leer nada, **elige** una rama.
- En general: así se reconoce la **unión** de dos lenguajes.

Elegir qué comparar

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



- Al empezar, sin leer nada, **elige** una rama.
- En general: así se reconoce la **unión** de dos lenguajes.

Error común

Usar la pila como contador de dos cosas a la vez. Para comparar hay que sacar, y lo que se saca se pierde: una pila compara **una sola pareja**.

Simular un PDA: buscar entre configuraciones

- NFA: bastaba el *conjunto de estados*.
- PDA: dos ejecuciones en el mismo estado pueden tener pilas distintas. Hace falta el conjunto de **configuraciones**.

Simular un PDA: buscar entre configuraciones

- NFA: bastaba el *conjunto de estados*.
- PDA: dos ejecuciones en el mismo estado pueden tener pilas distintas. Hace falta el conjunto de **configuraciones**.
- Es un grafo: se recorre por niveles o en profundidad, **recordando las vistas**.

Simular un PDA: buscar entre configuraciones

- NFA: bastaba el *conjunto de estados*.
- PDA: dos ejecuciones en el mismo estado pueden tener pilas distintas. Hace falta el conjunto de **configuraciones**.
- Es un grafo: se recorre por niveles o en profundidad, **recordando las vistas**.

Tres finales

- **Acepta:** encontró una configuración de aceptación.
- **Rechaza:** se acabaron las alcanzables.

Simular un PDA: buscar entre configuraciones

- NFA: bastaba el *conjunto de estados*.
- PDA: dos ejecuciones en el mismo estado pueden tener pilas distintas. Hace falta el conjunto de **configuraciones**.
- Es un grafo: se recorre por niveles o en profundidad, **recordando las vistas**.

Tres finales

- **Acepta:** encontró una configuración de aceptación.
- **Rechaza:** se acabaron las alcanzables.
- **Límite:** hay demasiadas. Un ciclo de transiciones ε que hace crecer la pila da *infinitas*.

Simular un PDA: buscar entre configuraciones

- NFA: bastaba el *conjunto de estados*.
- PDA: dos ejecuciones en el mismo estado pueden tener pilas distintas. Hace falta el conjunto de **configuraciones**.
- Es un grafo: se recorre por niveles o en profundidad, **recordando las vistas**.

Tres finales

- **Acepta:** encontró una configuración de aceptación.
- **Rechaza:** se acabaron las alcanzables.
- **Límite:** hay demasiadas. Un ciclo de transiciones ε que hace crecer la pila da *infinitas*.

Para qué sirve el conjunto de vistas

Para terminar cuando un ciclo de transiciones ε no cambia la pila, y para no repetir trabajo cuando dos caminos llegan a la misma configuración.

SECCIÓN 04

De gramática a PDA

04

De gramática a PDA: la idea

Teorema

Para toda gramática libre de contexto G existe un PDA P con $L(P) = L(G)$.

Una derivación por la izquierda, con $S \to (S)S \mid \varepsilon$:

$$S \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S)S \Rightarrow ()()S \Rightarrow ()()$$

De gramática a PDA: la idea

Teorema

Para toda gramática libre de contexto G existe un PDA P con $L(P) = L(G)$.

Una derivación por la izquierda, con $S \to (S)S \mid \varepsilon$:

$$S \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S)S \Rightarrow ()()S \Rightarrow ()()$$

- En cada paso: un **prefijo de terminales**, que ya no cambia, y **el resto**.

De gramática a PDA: la idea

Teorema

Para toda gramática libre de contexto G existe un PDA P con $L(P) = L(G)$.

Una derivación por la izquierda, con $S \to (S)S \mid \varepsilon$:

$$S \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S)S \Rightarrow ()()S \Rightarrow ()()$$

- En cada paso: un **prefijo de terminales**, que ya no cambia, y **el resto**.
- El prefijo se **compara con la entrada**. El resto va **en la pila**.

De gramática a PDA: la idea

Teorema

Para toda gramática libre de contexto G existe un PDA P con $L(P) = L(G)$.

Una derivación por la izquierda, con $S \to (S)S \mid \varepsilon$:

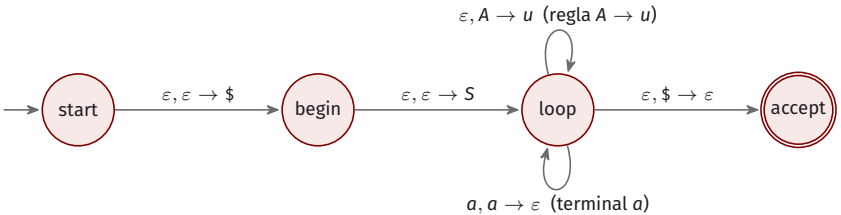
$$S \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S)S \Rightarrow ()()S \Rightarrow ()()$$

- En cada paso: un **prefijo de terminales**, que ya no cambia, y **el resto**.
- El prefijo se **compara con la entrada**. El resto va **en la pila**.

Idea clave

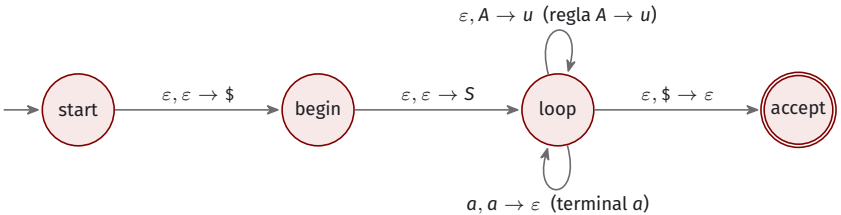
La pila guarda lo que **falta derivar**. El PDA adivina una derivación y comprueba que produce la entrada.

La construcción



- **Cima una variable** A**:** elegir una regla $A \to u$, sacar A y poner u. Aquí adivina.
- **Cima un terminal** a**:** compararlo con la entrada. Si no coincide, el camino muere.

La construcción



- **Cima una variable** A**:** elegir una regla $A \to u$, sacar A y poner u. Aquí adivina.
- **Cima un terminal** a**:** compararlo con la entrada. Si no coincide, el camino muere.
- "Poner u" se hace de un símbolo por vez, **de derecha a izquierda**, con $k - 1$ estados intermedios para una regla de $k \geq 2$ símbolos. Total: $4 + \sum(k - 1)$ estados.

Ejemplo: la gramática de los paréntesis

terminal

```
$ build/cfg_to_pda machines/balanced.cfg
# PDA de la gramática machines/balanced.cfg.
states: q_start q_begin q_loop q_accept r1_1 r1_2 r1_3
alphabet: ( )
stack: ( ) S $
start: q_start
accept: q_accept
delta:
q_start eps eps -> q_begin $
q_begin eps eps -> q_loop S
q_loop eps $ -> q_accept eps
q_loop eps S -> r1_1 S
q_loop eps S -> q_loop eps
q_loop ( ( -> q_loop eps
q_loop ) ) -> q_loop eps
r1_1 eps eps -> r1_2 )
r1_2 eps eps -> r1_3 S
r1_3 eps eps -> q_loop (
```

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

- Regla 1, con $k = 4$: tres estados intermedios.

Ejemplo: la gramática de los paréntesis

terminal

```
$ build/cfg_to_pda machines/balanced.cfg
# PDA de la gramática machines/balanced.cfg.
states: q_start q_begin q_loop q_accept r1_1 r1_2 r1_3
alphabet: ( )
stack: ( ) S $
start: q_start
accept: q_accept
delta:
q_start eps eps -> q_begin $
q_begin eps eps -> q_loop S
q_loop eps $ -> q_accept eps
q_loop eps S -> r1_1 S
q_loop eps S -> q_loop eps
q_loop ( ( -> q_loop eps
q_loop ) ) -> q_loop eps
r1_1 eps eps -> r1_2 )
r1_2 eps eps -> r1_3 S
r1_3 eps eps -> q_loop (
```

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

- Regla 1, con $k = 4$: tres estados intermedios.
- `q_loop eps S -> r1_1 S` saca S y pone el **último** símbolo. Después `)`, `S` y `(`: el paréntesis que abre queda en la cima.

Ejemplo: la gramática de los paréntesis

terminal

```
$ build/cfg_to_pda machines/balanced.cfg
# PDA de la gramática machines/balanced.cfg.
states: q_start q_begin q_loop q_accept r1_1 r1_2 r1_3
alphabet: ( )
stack: ( ) S $
start: q_start
accept: q_accept
delta:
q_start eps eps -> q_begin $
q_begin eps eps -> q_loop S
q_loop eps $ -> q_accept eps
q_loop eps S -> r1_1 S
q_loop eps S -> q_loop eps
q_loop ( ( -> q_loop eps
q_loop ) ) -> q_loop eps
r1_1 eps eps -> r1_2 )
r1_2 eps eps -> r1_3 S
r1_3 eps eps -> q_loop (
```

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

- Regla 1, con $k = 4$: tres estados intermedios.
- `q_loop eps S -> r1_1 S` saca S y pone el **último** símbolo. Después `)`, `S` y `(`: el paréntesis que abre queda en la cima.
- Regla 2: `q_loop eps S -> q_loop eps`.
- $4 + 3 = 7$ estados, 10 transiciones.

La ejecución con ( )

terminal

```
$ build/run_pda build/balanced_from_cfg.pda '()'
w = ()
  paso  transición  estado    falta leer  pila
     0              q_start   ()          ε
     1  ε, ε → $    q_begin   ()          $
     2  ε, ε → S    q_loop    ()          S$
     3  ε, S → S    r1_1      ()          S$
     4  ε, ε → )    r1_2      ()          )S$
     5  ε, ε → S    r1_3      ()          S)S$
     6  ε, ε → (    q_loop    ()          (S)S$
     7  (, ( → ε    q_loop    )           S)S$
     8  ε, S → ε    q_loop    )           )S$
     9  ), ) → ε    q_loop    ε           S$
    10  ε, S → ε    q_loop    ε           $
    11  ε, $ → ε    q_accept  ε           ε
  acepta en q_accept (configuraciones exploradas: 20)
```

En las filas de `q_loop`: leído + pila.

fila	leído	pila sin $
2	ε	S
6	ε	$(S)S$
7	(	$S)S$
8	(	$)S$
9	()	S
10	()	ε

La ejecución con ( )

terminal

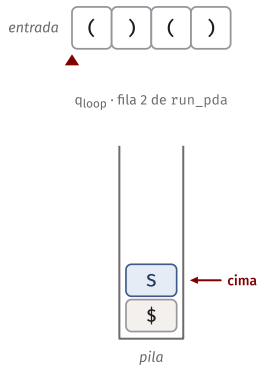
```
$ build/run_pda build/balanced_from_cfg.pda '()'
w = ()
  paso  transición  estado    falta leer  pila
     0              q_start   ()          ε
     1  ε, ε → $    q_begin   ()          $
     2  ε, ε → S    q_loop    ()          S$
     3  ε, S → S    r1_1      ()          S$
     4  ε, ε → )    r1_2      ()          )S$
     5  ε, ε → S    r1_3      ()          S)S$
     6  ε, ε → (    q_loop    ()          (S)S$
     7  (, ( → ε    q_loop    )           S)S$
     8  ε, S → ε    q_loop    )           )S$
     9  ), ) → ε    q_loop    ε           S$
    10  ε, S → ε    q_loop    ε           $
    11  ε, $ → ε    q_accept  ε           ε
  acepta en q_accept (configuraciones exploradas: 20)
```

En las filas de `q_loop`: leído + pila.

fila	leído	pila sin $
2	ε	S
6	ε	$(S)S$
7	(	$S)S$
8	(	$)S$
9	()	S
10	()	ε

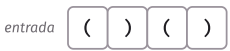
Juntos: $S \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()$, la derivación por la izquierda.

Leído + pila = la derivación, con ( ) ( )

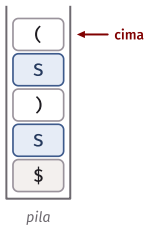


Fila 2, primera visita a q_{loop}: la pila tiene S sobre la marca. Falta derivar todo.

Leído + pila = la derivación, con ( ) ( )



q_{loop} · fila 6 de `run_pda`



LA DERIVACIÓN POR LA IZQUIERDA

S

⇒ (S)S S → (S)S

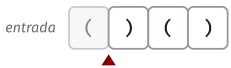
YA LEÍDO + PILA = LA FORMA ACTUAL



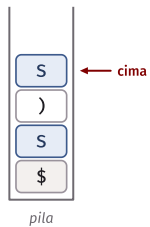
Reemplazar: la cima es S. Elige S → (S)S: saca S y pone (S)S, de derecha a izquierda: (queda en la cima.

La derivación avanza un paso. Pasa por `r1_1`, `r1_2` y `r1_3`, porque cada transición pone un solo símbolo.

Leído + pila = la derivación, con ( ) ( )



q_{loop} · fila 7 de `run_pda`



pila

LA DERIVACIÓN POR LA IZQUIERDA

S

⇒ (S)S S → (S)S

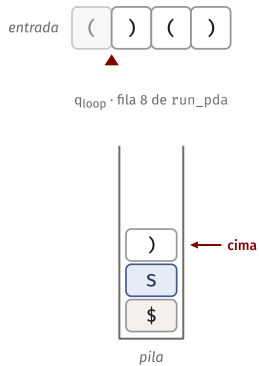
YA LEÍDO + PILA = LA FORMA ACTUAL



Comparar: la cima es (, igual al símbolo que sigue en la entrada. Lee uno y saca el otro.

Un terminal pasa de la pila a lo ya leído: la forma no cambia.

Leído + pila = la derivación, con ( ) ( )



LA DERIVACIÓN POR LA IZQUIERDA

S

⇒ (S)S S → (S)S

⇒ ( )S S → ε

YA LEÍDO + PILA = LA FORMA ACTUAL



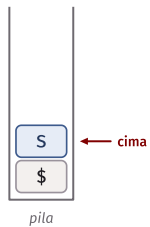
Reemplazar: la cima es S. Elige S → ε: saca S y no pone nada.

La derivación avanza un paso.

Leído + pila = la derivación, con ( ) ( )



q_{loop} · fila 9 de `run_pda`



LA DERIVACIÓN POR LA IZQUIERDA

S
⇒ (S)S S → (S)S
⇒ ( )S S → ε

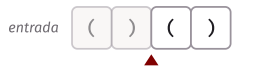
YA LEÍDO + PILA = LA FORMA ACTUAL



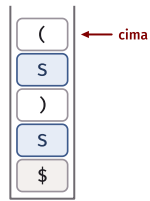
Comparar: la cima es), igual al símbolo que sigue en la entrada. Lee uno y saca el otro.

Un terminal pasa de la pila a lo ya leído: la forma no cambia.

Leído + pila = la derivación, con ( ) ( )



q_{loop} · fila 13 de `run_pda`



LA DERIVACIÓN POR LA IZQUIERDA

S	
⇒ (S)S	S → (S)S
⇒ ( )S	S → ε
⇒ **( )(S)S**	S → (S)S

YA LEÍDO + PILA = LA FORMA ACTUAL



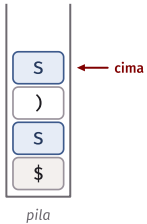
Reemplazar: la cima es S. Elige S → (S)S: saca S y pone (S)S, de derecha a izquierda: (queda en la cima.

La derivación avanza un paso. Pasa por `r1_1`, `r1_2` y `r1_3`, porque cada transición pone un solo símbolo.

Leído + pila = la derivación, con ( ) ( )



q_{loop} · fila 14 de `run_pda`



LA DERIVACIÓN POR LA IZQUIERDA

$$
\begin{aligned}
& S \\
\Rightarrow\ & (S)S && S \to (S)S \\
\Rightarrow\ & ()S && S \to \varepsilon \\
\Rightarrow\ & ()(S)S && S \to (S)S
\end{aligned}
$$

YA LEÍDO + PILA = LA FORMA ACTUAL



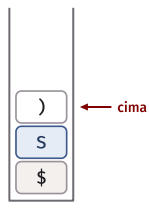
Comparar: la cima es (, igual al símbolo que sigue en la entrada. Lee uno y saca el otro.

Un terminal pasa de la pila a lo ya leído: la forma no cambia.

Leído + pila = la derivación, con ( ) ( )



q_{loop} · fila 15 de `run_pda`



LA DERIVACIÓN POR LA IZQUIERDA

S	
⇒ (S)S	S → (S)S
⇒ ( )S	S → ε
⇒ ( )(S)S	S → (S)S
⇒ **( )( )S**	S → ε

YA LEÍDO + PILA = LA FORMA ACTUAL



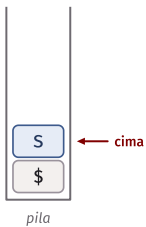
Reemplazar: la cima es S. Elige S → ε: saca S y no pone nada.

La derivación avanza un paso.

Leído + pila = la derivación, con ( ) ( )



q_{loop} · fila 16 de `run_pda`



LA DERIVACIÓN POR LA IZQUIERDA

S	
⇒ (S)S	S → (S)S
⇒ ( )S	S → ε
⇒ ( )(S)S	S → (S)S
⇒ ( )( )S	S → ε

YA LEÍDO + PILA = LA FORMA ACTUAL



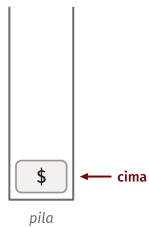
Comparar: la cima es), igual al símbolo que sigue en la entrada. Lee uno y saca el otro.

Un terminal pasa de la pila a lo ya leído: la forma no cambia.

Leído + pila = la derivación, con ( ) ( )



q_{loop} · fila 17 de `run_pda`



LA DERIVACIÓN POR LA IZQUIERDA

	S	
⇒	(S)S	S → (S)S
⇒	( )S	S → ε
⇒	( )(S)S	S → (S)S
⇒	( )( )S	S → ε
⇒	( )( )	S → ε

YA LEÍDO + PILA = LA FORMA ACTUAL



Reemplazar: la cima es S. Elige S → ε: saca S y no pone nada.

La derivación avanza un paso.

Leído + pila = la derivación, con ( ) ( )



q_{accept} · fila 18 de `run_pda`



LA DERIVACIÓN POR LA IZQUIERDA

	S	
⇒	(S)S	S → (S)S
⇒	( )S	S → ε
⇒	( )(S)S	S → (S)S
⇒	( )( )S	S → ε
⇒	( )( )	S → ε

YA LEÍDO + PILA = LA FORMA ACTUAL



Terminar: la cima es $ y no falta leer nada. Saca $ y pasa a q_{accept}: acepta.

Adivinó una derivación por la izquierda de ( )( ) y comprobó que produce la entrada.

Por qué funciona

1. Lo que la gramática deriva, el PDA lo consume

Si α deriva por la izquierda la cadena de terminales w, entonces para toda y y toda pila γ:

$$(q_{\text{loop}}, wy, \alpha\gamma) \vdash^* (q_{\text{loop}}, y, \gamma)$$

Por qué funciona

1. Lo que la gramática deriva, el PDA lo consume

Si α deriva por la izquierda la cadena de terminales w, entonces para toda y y toda pila γ:

$$(q_{\text{loop}}, wy, \alpha\gamma) \vdash^* (q_{\text{loop}}, y, \gamma)$$

2. Lo que el PDA consume, la gramática lo deriva

Si $(q_{\text{loop}}, xy, S\$) \vdash^* (q_{\text{loop}}, y, \beta\$)$, entonces $S \Rightarrow^* x\beta$.

Por qué funciona

1. Lo que la gramática deriva, el PDA lo consume

Si α deriva por la izquierda la cadena de terminales w, entonces para toda y y toda pila γ:

$$(q_{\text{loop}}, wy, \alpha\gamma) \vdash^* (q_{\text{loop}}, y, \gamma)$$

2. Lo que el PDA consume, la gramática lo deriva

Si $(q_{\text{loop}}, xy, S\$) \vdash^* (q_{\text{loop}}, y, \beta\$)$, entonces $S \Rightarrow^* x\beta$.

- La 1, por inducción sobre los pasos de la derivación. La 2, sobre los tramos entre dos visitas a q_{loop}.

Por qué funciona

1. Lo que la gramática deriva, el PDA lo consume

Si α deriva por la izquierda la cadena de terminales w, entonces para toda y y toda pila γ:

$$(q_{\text{loop}}, wy, \alpha\gamma) \vdash^* (q_{\text{loop}}, y, \gamma)$$

2. Lo que el PDA consume, la gramática lo deriva

Si $(q_{\text{loop}}, xy, S\$) \vdash^* (q_{\text{loop}}, y, \beta\$)$, entonces $S \Rightarrow^* x\beta$.

- La 1, por inducción sobre los pasos de la derivación. La 2, sobre los tramos entre dos visitas a q_{loop}.
- Con $x = w$ y $\beta = \varepsilon$: P acepta w si y solo si $S \Rightarrow^* w$. Completa, en los apuntes.

Un PDA que no se deja simular

La gramática de expresiones: $E \to E + T \mid T \qquad T \to T * F \mid F \qquad F \to (E) \mid \mathrm{a}$

terminal

```
$ build/cfg_to_pda machines/expr.cfg > build/expr.pda
$ grep -e '-> r1' -e '^r1' build/expr.pda
q_loop eps E -> r1_1 T
r1_1 eps eps -> r1_2 +
r1_2 eps eps -> q_loop E
$ build/run_pda --limite 5000 build/expr.pda 'a+a' 'a+' | tail -3
  acepta en q_accept (configuraciones exploradas: 107)
w = a+
  límite: más de 5000 configuraciones, sin aceptar
```

- $E \to E + T$ es **recursiva por la izquierda**: sin leer nada, la pila pasa de `E$` a `E+T$`, y otra vez a `E+T+T$`…

Un PDA que no se deja simular

La gramática de expresiones: $E \to E + T \mid T \qquad T \to T * F \mid F \qquad F \to (E) \mid \mathrm{a}$

terminal

```
$ build/cfg_to_pda machines/expr.cfg > build/expr.pda
$ grep -e '-> r1' -e '^r1' build/expr.pda
q_loop eps E -> r1_1 T
r1_1 eps eps -> r1_2 +
r1_2 eps eps -> q_loop E
$ build/run_pda --limite 5000 build/expr.pda 'a+a' 'a+' | tail -3
  acepta en q_accept (configuraciones exploradas: 107)
w = a+
  límite: más de 5000 configuraciones, sin aceptar
```

- $E \to E + T$ es **recursiva por la izquierda**: sin leer nada, la pila pasa de `E$` a `E+T$`, y otra vez a `E+T+T$`…
- Es un ciclo de transiciones ε que hace crecer la pila: infinitas configuraciones.

Un PDA que no se deja simular

La gramática de expresiones: $E \to E + T \mid T$ $\quad T \to T * F \mid F$ $\quad F \to (E) \mid \mathtt{a}$

terminal

```
$ build/cfg_to_pda machines/expr.cfg > build/expr.pda
$ grep -e '-> r1' -e '^r1' build/expr.pda
q_loop eps E -> r1_1 T
r1_1 eps eps -> r1_2 +
r1_2 eps eps -> q_loop E
$ build/run_pda --limite 5000 build/expr.pda 'a+a' 'a+' | tail -3
  acepta en q_accept (configuraciones exploradas: 107)
w = a+
  límite: más de 5000 configuraciones, sin aceptar
```

- $E \to E + T$ es **recursiva por la izquierda**: sin leer nada, la pila pasa de `E$` a `E+T$`, y otra vez a `E+T+T$`…
- Es un ciclo de transiciones ε que hace crecer la pila: infinitas configuraciones.
- El teorema está bien: para `a+` no *existe* una ejecución que acepte. Lo que falla es la búsqueda exhaustiva.

SECCIÓN 05

De PDA a gramática

05



De PDA a gramática: la idea

Teorema

Para todo PDA P existe una gramática libre de contexto G con $L(G) = L(P)$.

Una variable A_{pq} por cada **par de estados**: genera las cadenas que llevan a P de p con la pila vacía a q con la pila vacía.

De PDA a gramática: la idea

Teorema

Para todo PDA P existe una gramática libre de contexto G con $L(G) = L(P)$.

Una variable A_{pq} por cada **par de estados**: genera las cadenas que llevan a P de p con la pila vacía a q con la pila vacía.

- Miren la **altura de la pila** durante una ejecución, como un perfil de montaña.

De PDA a gramática: la idea

Teorema

Para todo PDA P existe una gramática libre de contexto G con $L(G) = L(P)$.

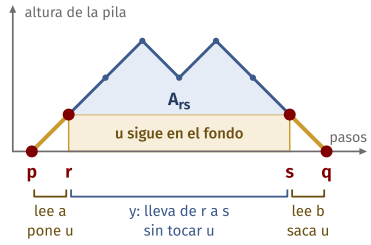
Una variable A_{pq} por cada **par de estados**: genera las cadenas que llevan a P de p con la pila vacía a q con la pila vacía.

- Miren la **altura de la pila** durante una ejecución, como un perfil de montaña.
- Si empieza y termina con la pila vacía, el perfil empieza y termina en el suelo. Solo puede tener **dos formas**.

Las dos formas del perfil

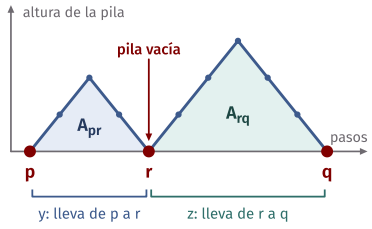
(A) NO TOCA EL SUELO EN EL MEDIO

$A_{pq} \to a\ A_{rs}\ b$



(B) TOCA EL SUELO EN UN ESTADO R

$A_{pq} \to A_{pr}\ A_{rq}$



En un PDA normalizado cada paso pone o saca un símbolo: el perfil sube o baja uno, y solo puede tener estas dos formas.

Caso base, sin moverse: $A_{pp} \to \varepsilon$.

Las tres reglas

- **(a) No se vacía en el medio.** El símbolo u que pone el primer paso se queda en el fondo, y es el que saca el último.

$$A_{pq} \to a\, A_{rs}\, b$$

si $(r, u) \in \delta(p, a, \varepsilon)$ y $(q, \varepsilon) \in \delta(s, b, u)$.

Las tres reglas

- **(a) No se vacía en el medio.** El símbolo u que pone el primer paso se queda en el fondo, y es el que saca el último.

$$A_{pq} \to a\, A_{rs}\, b$$

  si $(r, u) \in \delta(p, a, \varepsilon)$ y $(q, \varepsilon) \in \delta(s, b, u)$.
- **(b) Se vacía en r.** Se parte en dos.

$$A_{pq} \to A_{pr}\, A_{rq}$$

Las tres reglas

- **(a) No se vacía en el medio.** El símbolo u que pone el primer paso se queda en el fondo, y es el que saca el último.

$$A_{pq} \to a\, A_{rs}\, b$$

  si $(r, u) \in \delta(p, a, \varepsilon)$ y $(q, \varepsilon) \in \delta(s, b, u)$.
- **(b) Se vacía en r.** Se parte en dos.

$$A_{pq} \to A_{pr}\, A_{rq}$$

- **Sin moverse:** $A_{pp} \to \varepsilon$.

Las tres reglas

- **(a) No se vacía en el medio.** El símbolo u que pone el primer paso se queda en el fondo, y es el que saca el último.

$$A_{pq} \to a\, A_{rs}\, b$$

  si $(r, u) \in \delta(p, a, \varepsilon)$ y $(q, \varepsilon) \in \delta(s, b, u)$.
- **(b) Se vacía en r.** Se parte en dos.

$$A_{pq} \to A_{pr}\, A_{rq}$$

- **Sin moverse:** $A_{pp} \to \varepsilon$.

Idea clave

$A_{pq} \to aA_{rs}b$ empareja el paso que **pone** un símbolo con el paso que lo **saca**.

Es el mismo emparejamiento que hace $S \to 0S1$ entre un cero y su uno.

Las tres reglas

- **(a) No se vacía en el medio.** El símbolo u que pone el primer paso se queda en el fondo, y es el que saca el último.

$$A_{pq} \to a\, A_{rs}\, b$$

  si $(r, u) \in \delta(p, a, \varepsilon)$ y $(q, \varepsilon) \in \delta(s, b, u)$.
- **(b) Se vacía en r.** Se parte en dos.

$$A_{pq} \to A_{pr}\, A_{rq}$$

- **Sin moverse:** $A_{pp} \to \varepsilon$.

Idea clave

$A_{pq} \to aA_{rs}b$ empareja el paso que **pone** un símbolo con el paso que lo **saca**.

Es el mismo emparejamiento que hace $S \to \texttt{0}S\texttt{1}$ entre un cero y su uno.

Variable inicial: $A_{q_0 q_{\text{accept}}}$.

Primero, ordenar el PDA

El argumento del perfil necesita un PDA **normalizado**:

1. un **solo** estado de aceptación;
2. **vacía la pila** antes de aceptar;
3. cada transición **pone un símbolo o saca un símbolo**: ni las dos cosas, ni ninguna.

Primero, ordenar el PDA

El argumento del perfil necesita un PDA **normalizado**:

1. un **solo** estado de aceptación;
2. **vacía la pila** antes de aceptar;
3. cada transición **pone un símbolo o saca un símbolo**: ni las dos cosas, ni ninguna.

- Con 1 y 2: aceptar es ir de q_0 con la pila vacía a q_{accept} con la pila vacía.
- Con 3: cada paso sube o baja la altura en exactamente uno.

Primero, ordenar el PDA

El argumento del perfil necesita un PDA **normalizado**:

1. un **solo** estado de aceptación;
2. **vacía la pila** antes de aceptar;
3. cada transición **pone un símbolo o saca un símbolo**: ni las dos cosas, ni ninguna.

- Con 1 y 2: aceptar es ir de q_0 con la pila vacía a q_{accept} con la pila vacía.
- Con 3: cada paso sube o baja la altura en exactamente uno.
- **Siempre se puede**: una marca de fondo nueva, un estado que vacía la pila, y partir en dos las transiciones que reemplazan o que no tocan la pila.

Primero, ordenar el PDA

El argumento del perfil necesita un PDA **normalizado**:

1. un **solo** estado de aceptación;
2. **vacía la pila** antes de aceptar;
3. cada transición **pone un símbolo o saca un símbolo**: ni las dos cosas, ni ninguna.

- Con 1 y 2: aceptar es ir de q_0 con la pila vacía a q_{accept} con la pila vacía.
- Con 3: cada paso sube o baja la altura en exactamente uno.
- **Siempre se puede**: una marca de fondo nueva, un estado que vacía la pila, y partir en dos las transiciones que reemplazan o que no tocan la pila.

En el código

`build/normalize_pda machines/zeros_ones.pda`: el autómata de 4 estados queda con 9.

Un ejemplo pequeño

$\{0^n 1^n\}$ normalizado, con aceptación solo en q_3:

de	lee, saca $\to$ pone	a	
q_0	$\varepsilon, \varepsilon \to \$$	q_1	pone
q_1	$0, \varepsilon \to 0$	q_1	pone
q_1	$1, 0 \to \varepsilon$	q_2	saca
q_2	$1, 0 \to \varepsilon$	q_2	saca
q_1	$\varepsilon, \$ \to \varepsilon$	q_3	saca
q_2	$\varepsilon, \$ \to \varepsilon$	q_3	saca

Emparejar cada **pone** con cada **saca** del mismo símbolo:

Un ejemplo pequeño

$\{\mathtt{0}^n\mathtt{1}^n\}$ normalizado, con aceptación solo en q_3:

de	lee, saca $\to$ pone	a	
q_0	$\varepsilon, \varepsilon \to \$$	q_1	pone
q_1	$\mathtt{0}, \varepsilon \to \mathtt{0}$	q_1	pone
q_1	$\mathtt{1}, \mathtt{0} \to \varepsilon$	q_2	saca
q_2	$\mathtt{1}, \mathtt{0} \to \varepsilon$	q_2	saca
q_1	$\varepsilon, \$ \to \varepsilon$	q_3	saca
q_2	$\varepsilon, \$ \to \varepsilon$	q_3	saca

Emparejar cada **pone** con cada **saca** del mismo símbolo:

- \$: lo pone $q_0 \to q_1$; lo sacan $q_1 \to q_3$ y $q_2 \to q_3$.

$$A_{03} \to A_{11} \qquad A_{03} \to A_{12}$$

Un ejemplo pequeño

$\{0^n 1^n\}$ normalizado, con aceptación solo en q_3:

de	lee, saca → pone	a	
q_0	$\varepsilon, \varepsilon \to \$$	q_1	pone
q_1	$0, \varepsilon \to 0$	q_1	pone
q_1	$1, 0 \to \varepsilon$	q_2	saca
q_2	$1, 0 \to \varepsilon$	q_2	saca
q_1	$\varepsilon, \$ \to \varepsilon$	q_3	saca
q_2	$\varepsilon, \$ \to \varepsilon$	q_3	saca

Emparejar cada **pone** con cada **saca** del mismo símbolo:

- \$: lo pone $q_0 \to q_1$; lo sacan $q_1 \to q_3$ y $q_2 \to q_3$.

$$A_{03} \to A_{11} \qquad A_{03} \to A_{12}$$

- 0: lo pone $q_1 \to q_1$ leyendo 0; lo sacan $q_1 \to q_2$ y $q_2 \to q_2$ leyendo 1.

$$A_{12} \to 0A_{11}1 \qquad A_{12} \to 0A_{12}1$$

Un ejemplo pequeño

$\{0^n 1^n\}$ normalizado, con aceptación solo en q_3:

de	lee, saca → pone	a	
q_0	$\varepsilon, \varepsilon \to \$$	q_1	pone
q_1	$0, \varepsilon \to 0$	q_1	pone
q_1	$1, 0 \to \varepsilon$	q_2	saca
q_2	$1, 0 \to \varepsilon$	q_2	saca
q_1	$\varepsilon, \$ \to \varepsilon$	q_3	saca
q_2	$\varepsilon, \$ \to \varepsilon$	q_3	saca

Emparejar cada **pone** con cada **saca** del mismo símbolo:

- \$: lo pone $q_0 \to q_1$; lo sacan $q_1 \to q_3$ y $q_2 \to q_3$.

$$A_{03} \to A_{11} \qquad A_{03} \to A_{12}$$

- 0: lo pone $q_1 \to q_1$ leyendo 0; lo sacan $q_1 \to q_2$ y $q_2 \to q_2$ leyendo 1.

$$A_{12} \to 0A_{11}1 \qquad A_{12} \to 0A_{12}1$$

Más $4^3 = 64$ reglas $A_{pq} \to A_{pr}A_{rq}$ y 4 reglas $A_{pp} \to \varepsilon$: **72** reglas, 16 variables.

La gramática que sale

terminal

```
$ build/pda_to_cfg machines/zeros_ones_norm.pda
# Gramática del PDA machines/zeros_ones_norm.pda.
# La construcción da 16 variables y 72 reglas.
# Sirven 6 variables y 16 reglas.
variables: A(q0,q0) A(q0,q3) A(q1,q1) A(q1,q2) A(q2,q2) A(q3,q3)
terminals: 0 1
start: A(q0,q3)
rules:
A(q0,q0) -> A(q0,q0) A(q0,q0) | eps
A(q0,q3) -> A(q1,q1) | A(q1,q2) | A(q0,q0) A(q0,q3) | A(q0,q3) A(q3,q3)
A(q1,q1) -> A(q1,q1) A(q1,q1) | eps
A(q1,q2) -> 0 A(q1,q1) 1 | 0 A(q1,q2) 1 | A(q1,q1) A(q1,q2) | A(q1,q2) A(q2,q2)
A(q2,q2) -> A(q2,q2) A(q2,q2) | eps
A(q3,q3) -> A(q3,q3) A(q3,q3) | eps
```

- A_{00}, A_{11}, A_{22} y A_{33} solo generan ε. Queda: $A_{03} \to \varepsilon \mid A_{12}$, $\quad A_{12} \to 01 \mid 0A_{12}1$.

La gramática que sale

terminal

```
$ build/pda_to_cfg machines/zeros_ones_norm.pda
# Gramática del PDA machines/zeros_ones_norm.pda.
# La construcción da 16 variables y 72 reglas.
# Sirven 6 variables y 16 reglas.
variables: A(q0,q0) A(q0,q3) A(q1,q1) A(q1,q2) A(q2,q2) A(q3,q3)
terminals: 0 1
start: A(q0,q3)
rules:
A(q0,q0) -> A(q0,q0) A(q0,q0) | eps
A(q0,q3) -> A(q1,q1) | A(q1,q2) | A(q0,q0) A(q0,q3) | A(q0,q3) A(q3,q3)
A(q1,q1) -> A(q1,q1) A(q1,q1) | eps
A(q1,q2) -> 0 A(q1,q1) 1 | 0 A(q1,q2) 1 | A(q1,q1) A(q1,q2) | A(q1,q2) A(q2,q2)
A(q2,q2) -> A(q2,q2) A(q2,q2) | eps
A(q3,q3) -> A(q3,q3) A(q3,q3) | eps
```

- A_{00}, A_{11}, A_{22} y A_{33} solo generan ε. Queda: $A_{03} \to \varepsilon \mid A_{12}$, $\quad A_{12} \to 01 \mid 0A_{12}1$.
- La de siempre. La construcción es derrochadora; su valor es que **existe**.

Lo que dice la equivalencia

Teorema

Un lenguaje es libre de contexto si y solo si algún autómata de pila lo reconoce.

- El análogo del teorema de Kleene: dos descripciones de la misma clase.

Lo que dice la equivalencia

Teorema

Un lenguaje es libre de contexto si y solo si algún autómata de pila lo reconoce.

- El análogo del teorema de Kleene: dos descripciones de la misma clase.
- Sirve para elegir: unas propiedades salen más fácil con gramáticas, otras con autómatas.

Lo que dice la equivalencia

Teorema

Un lenguaje es libre de contexto si y solo si algún autómata de pila lo reconoce.

- El análogo del teorema de Kleene: dos descripciones de la misma clase.
- Sirve para elegir: unas propiedades salen más fácil con gramáticas, otras con autómatas.

Corolario

Todo lenguaje regular es libre de contexto.

Un DFA es un PDA que no usa la pila: $\delta'(q, a, \varepsilon) = \{(\delta(q, a), \varepsilon)\}$, con $\Gamma = \emptyset$.
La inclusión es estricta: $\{0^n 1^n\}$.

El PDA determinista reconoce menos

Determinista: en ninguna configuración hay más de una transición que se pueda usar.

El PDA determinista reconoce menos

Determinista: en ninguna configuración hay más de una transición que se pueda usar.

- El de 0^n1^n lo es. El de los palíndromos no: en q_1 puede guardar o saltar.

El PDA determinista reconoce menos

Determinista: en ninguna configuración hay más de una transición que se pueda usar.

- El de $0^n 1^n$ lo es. El de los palíndromos no: en q_1 puede guardar o saltar.

Teorema (sin demostración)

Ningún PDA determinista reconoce $\{ ww^R \}$.

El PDA determinista reconoce menos

Determinista: en ninguna configuración hay más de una transición que se pueda usar.

- El de $\texttt{0}^n\texttt{1}^n$ lo es. El de los palíndromos no: en q_1 puede guardar o saltar.

Teorema (sin demostración)

Ningún PDA determinista reconoce $\{ ww^R \}$.

La intuición

- Hay que empezar a sacar justo en el medio, y nada dice dónde está.
- Si después de `0110` vacía la pila para comprobar, perdió lo que necesita para `01100110`.

El PDA determinista reconoce menos

Determinista: en ninguna configuración hay más de una transición que se pueda usar.

- El de $0^n 1^n$ lo es. El de los palíndromos no: en q_1 puede guardar o saltar.

Teorema (sin demostración)

Ningún PDA determinista reconoce $\{ ww^R \}$.

La intuición

- Hay que empezar a sacar justo en el medio, y nada dice dónde está.
- Si después de `0110` vacía la pila para comprobar, perdió lo que necesita para `01100110`.

Idea clave

En los autómatas finitos el no determinismo no agrega poder: NFA $=$ DFA. En los de pila, **sí**.

SECCIÓN 06

Comparar tres cantidades

06

Un lenguaje que pide comparar tres cantidades

$$D = \{\, a^n b^n c^n : n \geq 0 \,\}$$

- Guardar las a, sacarlas con las b, y después…

terminal

```
$ build/run_pda machines/anbn_cm.pda aabbc
w = aabbc
  paso  transición  estado  falta leer  pila
     0              q0      aabbc       ε
     1  ε, ε → $    q1      aabbc       $
     2  a, ε → a    q1      abbc        a$
     3  a, ε → a    q1      bbc         aa$
     4  ε, ε → ε    q2      bbc         aa$
     5  b, a → ε    q2      bc          a$
     6  b, a → ε    q2      c           $
     7  ε, $ → ε    q3      c           ε
     8  c, ε → ε    q3      ε           ε
  acepta en q3 (configuraciones exploradas: 12)
```

Un lenguaje que pide comparar tres cantidades

$$D = \{\, a^n b^n c^n : n \geq 0 \,\}$$

- Guardar las a, sacarlas con las b, y después…
- …la pila está vacía. Las c no tienen con qué compararse.

terminal

```
$ build/run_pda machines/anbn_cm.pda aabbc
w = aabbc
  paso  transición  estado  falta leer  pila
     0              q0      aabbc       ε
     1  ε, ε → $    q1      aabbc       $
     2  a, ε → a    q1      abbc        a$
     3  a, ε → a    q1      bbc         aa$
     4  ε, ε → ε    q2      bbc         aa$
     5  b, a → ε    q2      bc          a$
     6  b, a → ε    q2      c           $
     7  ε, $ → ε    q3      c           ε
     8  c, ε → ε    q3      ε           ε
  acepta en q3 (configuraciones exploradas: 12)
```

Un lenguaje que pide comparar tres cantidades

$$D = \{\, a^n b^n c^n : n \geq 0 \,\}$$

- Guardar las a, sacarlas con las b, y después…
- …la pila está vacía. Las c no tienen con qué compararse.
- El intento acepta aabbc. Reconoce $\{a^n b^n c^m\}$.

terminal

```
$ build/run_pda machines/anbn_cm.pda aabbc
w = aabbc
  paso  transición  estado  falta leer  pila
     0              q0      aabbc       ε
     1  ε, ε → $    q1      aabbc       $
     2  a, ε → a    q1      abbc        a$
     3  a, ε → a    q1      bbc         aa$
     4  ε, ε → ε    q2      bbc         aa$
     5  b, a → ε    q2      bc          a$
     6  b, a → ε    q2      c           $
     7  ε, $ → ε    q3      c           ε
     8  c, ε → ε    q3      ε           ε
  acepta en q3 (configuraciones exploradas: 12)
```

Un lenguaje que pide comparar tres cantidades

$$D = \{\, a^n b^n c^n : n \geq 0 \,\}$$

- Guardar las a, sacarlas con las b, y después…
- …la pila está vacía. Las c no tienen con qué compararse.
- El intento acepta aabbc. Reconoce $\{a^n b^n c^m\}$.

terminal

```
$ build/run_pda machines/anbn_cm.pda aabbc
w = aabbc
  paso  transición  estado  falta leer  pila
     0              q0      aabbc       ε
     1  ε, ε → $    q1      aabbc       $
     2  a, ε → a    q1      abbc        a$
     3  a, ε → a    q1      bbc         aa$
     4  ε, ε → ε    q2      bbc         aa$
     5  b, a → ε    q2      bc          a$
     6  b, a → ε    q2      c           $
     7  ε, $ → ε    q3      c           ε
     8  c, ε → ε    q3      ε           ε
  acepta en q3 (configuraciones exploradas: 12)
```

Que un intento falle no demuestra nada. Hace falta una propiedad que *todos* los lenguajes libres de contexto tengan y D no.

SECCIÓN 07

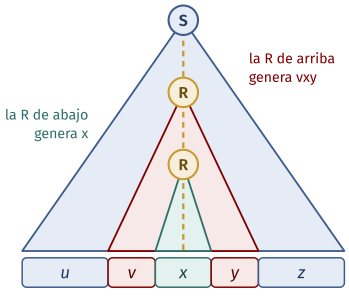
El lema de bombeo

07

La idea: un árbol muy alto repite una variable

El árbol de s = uvxyz

En un camino largo, una variable R aparece dos veces.

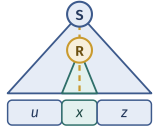


S ⇒* uRz **R ⇒* vRy** **R ⇒* x**

La del medio se puede usar cualquier cantidad de veces, o ninguna: S ⇒* uv^ixy^iz.

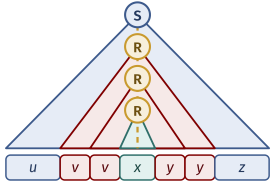
i = 0: uxz

El subárbol de la R de abajo, en el lugar del de arriba.



i = 2: uvvxyyz

Una copia del subárbol de arriba, en lugar del de abajo.



Lo que se lee del árbol

- Cada nodo tiene pocos hijos. Muchísimas hojas $\Rightarrow$ árbol **alto**.

Lo que se lee del árbol

- Cada nodo tiene pocos hijos. Muchísimas hojas $\Rightarrow$ árbol **alto**.
- En un camino más largo que la cantidad de variables, una variable R **se repite**.

Lo que se lee del árbol

- Cada nodo tiene pocos hijos. Muchísimas hojas $\Rightarrow$ árbol **alto**.
- En un camino más largo que la cantidad de variables, una variable R **se repite**.
- Del árbol se leen tres derivaciones:

$$S \Rightarrow^* uRz \qquad R \Rightarrow^* vRy \qquad R \Rightarrow^* x$$

Lo que se lee del árbol

- Cada nodo tiene pocos hijos. Muchísimas hojas $\Rightarrow$ árbol **alto**.
- En un camino más largo que la cantidad de variables, una variable R **se repite**.
- Del árbol se leen tres derivaciones:

$$S \Rightarrow^* uRz \qquad R \Rightarrow^* vRy \qquad R \Rightarrow^* x$$

- La del medio se usa las veces que se quiera:

$$S \Rightarrow^* uRz \Rightarrow^* uv^iRy^iz \Rightarrow^* uv^ixy^iz$$

Lo que se lee del árbol

- Cada nodo tiene pocos hijos. Muchísimas hojas $\Rightarrow$ árbol **alto**.
- En un camino más largo que la cantidad de variables, una variable R **se repite**.
- Del árbol se leen tres derivaciones:

$$S \Rightarrow^* uRz \qquad R \Rightarrow^* vRy \qquad R \Rightarrow^* x$$

- La del medio se usa las veces que se quiera:

$$S \Rightarrow^* uRz \Rightarrow^* uv^iRy^iz \Rightarrow^* uv^ixy^iz$$

Se repiten **dos** trozos, v e y, la misma cantidad de veces: como en $S \to 0S1$.

El enunciado

Lema de bombeo para lenguajes libres de contexto

Si A es libre de contexto, existe $p \geq 1$, la **longitud de bombeo**, tal que toda $s \in A$ con $|s| \geq p$ se puede dividir en $s = uvxyz$ con:

1. $uv^i xy^i z \in A$ para todo $i \geq 0$;
2. $|vy| > 0$;
3. $|vxy| \leq p$.

El enunciado

Lema de bombeo para lenguajes libres de contexto

Si A es libre de contexto, existe $p \geq 1$, la **longitud de bombeo**, tal que toda $s \in A$ con $|s| \geq p$ se puede dividir en $s = uvxyz$ con:

1. $uv^i xy^i z \in A$ para todo $i \geq 0$;
2. $|vy| > 0$;
3. $|vxy| \leq p$.

- Condición 2: v e y no son las dos vacías. Una sí puede serlo.

El enunciado

Lema de bombeo para lenguajes libres de contexto

Si A es libre de contexto, existe $p \geq 1$, la **longitud de bombeo**, tal que toda $s \in A$ con $|s| \geq p$ se puede dividir en $s = uvxyz$ con:

1. $uv^i xy^i z \in A$ para todo $i \geq 0$;
2. $|vy| > 0$;
3. $|vxy| \leq p$.

- Condición 2: v e y no son las dos vacías. Una sí puede serlo.
- Condición 3: los dos trozos caben, con lo que hay entre ellos, en una ventana de p símbolos.
- La ventana puede estar **en cualquier lugar** de s. No hay garantía de que esté al principio.

La demostración (1): cuánto vale p

Sea $G = (V, \Sigma, R, S)$ una gramática de A. Sea b el máximo entre 2 y la longitud del lado derecho más largo.

$$p = b^{|V|+1}$$

La demostración (1): cuánto vale p

Sea $G = (V, \Sigma, R, S)$ una gramática de A. Sea b el máximo entre 2 y la longitud del lado derecho más largo.

$$p = b^{|V|+1}$$

- **Un árbol bajo tiene pocas hojas.** Cada nodo tiene a lo sumo b hijos: un árbol de altura h genera a lo sumo b^h símbolos. (Inducción sobre h.)

La demostración (1): cuánto vale p

Sea $G = (V, \Sigma, R, S)$ una gramática de A. Sea b el máximo entre 2 y la longitud del lado derecho más largo.

$$p = b^{|V|+1}$$

- **Un árbol bajo tiene pocas hojas.** Cada nodo tiene a lo sumo b hijos: un árbol de altura h genera a lo sumo b^h símbolos. (Inducción sobre h.)
- Entonces, si $|s| \geq b^{|V|+1}$, todo árbol de s tiene altura **al menos** $|V| + 1$: con altura $|V|$ generaría a lo sumo $b^{|V|} < b^{|V|+1}$.

La demostración (1): cuánto vale p

Sea $G = (V, \Sigma, R, S)$ una gramática de A. Sea b el máximo entre 2 y la longitud del lado derecho más largo.

$$p = b^{|V|+1}$$

- **Un árbol bajo tiene pocas hojas.** Cada nodo tiene a lo sumo b hijos: un árbol de altura h genera a lo sumo b^h símbolos. (Inducción sobre h.)
- Entonces, si $|s| \geq b^{|V|+1}$, todo árbol de s tiene altura **al menos** $|V| + 1$: con altura $|V|$ generaría a lo sumo $b^{|V|} < b^{|V|+1}$.
- Sea τ un árbol de s con **la menor cantidad de nodos**, y su camino más largo: al menos $|V| + 1$ aristas, es decir, al menos $|V| + 1$ variables y una hoja.

La demostración (1): cuánto vale p

Sea $G = (V, \Sigma, R, S)$ una gramática de A. Sea b el máximo entre 2 y la longitud del lado derecho más largo.

$$p = b^{|V|+1}$$

- **Un árbol bajo tiene pocas hojas.** Cada nodo tiene a lo sumo b hijos: un árbol de altura h genera a lo sumo b^h símbolos. (Inducción sobre h.)
- Entonces, si $|s| \geq b^{|V|+1}$, todo árbol de s tiene altura **al menos** $|V| + 1$: con altura $|V|$ generaría a lo sumo $b^{|V|} < b^{|V|+1}$.
- Sea τ un árbol de s con **la menor cantidad de nodos**, y su camino más largo: al menos $|V| + 1$ aristas, es decir, al menos $|V| + 1$ variables y una hoja.
- Entre las $|V| + 1$ variables **más bajas** de ese camino, dos son la misma: R.

La demostración (2): las tres condiciones

- **Condición 1.** $S \Rightarrow^* uRz$, $R \Rightarrow^* vRy$ y $R \Rightarrow^* x$ dan uv^ixy^iz para todo $i \geq 0$.

La demostración (2): las tres condiciones

- **Condición 1.** $S \Rightarrow^* uRz$, $R \Rightarrow^* vRy$ y $R \Rightarrow^* x$ dan uv^ixy^iz para todo $i \geq 0$.
- **Condición 2.** Si $v = y = \varepsilon$, poner el subárbol de la R de abajo en el lugar del de la R de arriba da otro árbol de $s = uxz$ con **menos nodos**. Contradice la elección de τ.

La demostración (2): las tres condiciones

- **Condición 1.** $S \Rightarrow^* uRz$, $R \Rightarrow^* vRy$ y $R \Rightarrow^* x$ dan uv^ixy^iz para todo $i \geq 0$.
- **Condición 2.** Si $v = y = \varepsilon$, poner el subárbol de la R de abajo en el lugar del de la R de arriba da otro árbol de $s = uxz$ con **menos nodos**. Contradice la elección de τ.
- **Condición 3.** La R de arriba está entre las $|V| + 1$ variables más bajas del camino más largo. Su subárbol tiene altura a lo sumo $|V| + 1$, y genera a lo sumo $b^{|V|+1} = p$ símbolos: $|vxy| \leq p$.

La demostración (2): las tres condiciones

- **Condición 1.** $S \Rightarrow^* uRz$, $R \Rightarrow^* vRy$ y $R \Rightarrow^* x$ dan uv^ixy^iz para todo $i \geq 0$.
- **Condición 2.** Si $v = y = \varepsilon$, poner el subárbol de la R de abajo en el lugar del de la R de arriba da otro árbol de $s = uxz$ con **menos nodos**. Contradice la elección de τ.
- **Condición 3.** La R de arriba está entre las $|V| + 1$ variables más bajas del camino más largo. Su subárbol tiene altura a lo sumo $|V| + 1$, y genera a lo sumo $b^{|V|+1} = p$ símbolos: $|vxy| \leq p$.

El valor de p

Para $S \to \texttt{0}S\texttt{1} \mid \varepsilon$: $b = 3$, $|V| = 1$, $p = 9$. No es el más chico que sirve: alcanza $p = 2$.

Error común

Leer el lema al revés. Solo sirve para demostrar que un lenguaje **no** es libre de contexto.

SECCIÓN 08

Usar el lema

08

Usar el lema: el juego

Para demostrar que A **no** es libre de contexto, por contradicción:

1. **El adversario elige** p**.** No sabemos cuál: el argumento sirve para todos.

Usar el lema: el juego

Para demostrar que A **no** es libre de contexto, por contradicción:

1. **El adversario elige** p**.** No sabemos cuál: el argumento sirve para todos.
2. **Nosotros elegimos** s, con $s \in A$ y $|s| \geq p$. Casi siempre depende de p.

Usar el lema: el juego

Para demostrar que A **no** es libre de contexto, por contradicción:

1. **El adversario elige** p**.** No sabemos cuál: el argumento sirve para todos.
2. **Nosotros elegimos** s, con $s \in A$ y $|s| \geq p$. Casi siempre depende de p.
3. **El adversario elige la división** $s = uvxyz$, con $|vy| > 0$ y $|vxy| \leq p$.
   No sabemos cuál: aquí está el **análisis por casos**.

Usar el lema: el juego

Para demostrar que A **no** es libre de contexto, por contradicción:

1. **El adversario elige** p**.** No sabemos cuál: el argumento sirve para todos.
2. **Nosotros elegimos** s, con $s \in A$ y $|s| \geq p$. Casi siempre depende de p.
3. **El adversario elige la división** $s = uvxyz$, con $|vy| > 0$ y $|vxy| \leq p$.
   No sabemos cuál: aquí está el **análisis por casos**.
4. **Nosotros elegimos** i, y ganamos si $uv^ixy^iz \notin A$. Puede ser distinto en cada caso.

Usar el lema: el juego

Para demostrar que A **no** es libre de contexto, por contradicción:

1. **El adversario elige** p**.** No sabemos cuál: el argumento sirve para todos.
2. **Nosotros elegimos** s, con $s \in A$ y $|s| \geq p$. Casi siempre depende de p.
3. **El adversario elige la división** $s = uvxyz$, con $|vy| > 0$ y $|vxy| \leq p$.
   No sabemos cuál: aquí está el **análisis por casos**.
4. **Nosotros elegimos** i, y ganamos si $uv^i xy^i z \notin A$. Puede ser distinto en cada caso.

Dos maneras de caer

Las cadenas de los ejemplos son **bloques** de un mismo símbolo. Un trozo v o y puede estar **dentro de un bloque**, o **abarcar el borde** entre dos. Se comportan distinto al bombear, y el segundo caso es el que se olvida.

$\{a^n b^n c^n\}$ no es libre de contexto

Nuestra cadena: $s = a^p b^p c^p$. En los dos casos elegimos $i = 2$.

$\{a^n b^n c^n\}$ no es libre de contexto

Nuestra cadena: $s = a^p b^p c^p$. En los dos casos elegimos $i = 2$.

Caso 1: v e y de un solo tipo cada una

uv^2xy^2z sigue ordenada. Entre las dos tocan a lo sumo **dos** de los tres símbolos: una cantidad crece y otra se queda en p.

$\{a^n b^n c^n\}$ no es libre de contexto

Nuestra cadena: $s = a^p b^p c^p$. En los dos casos elegimos $i = 2$.

Caso 1: v e y de un solo tipo cada una

uv^2xy^2z sigue ordenada. Entre las dos tocan a lo sumo **dos** de los tres símbolos: una cantidad crece y otra se queda en p.

Caso 2: v o y tiene dos símbolos distintos

Si $v = a^j b^k$: $vv = a^j b^k a^j b^k$.
Hay una a después de una b: ya no es $a^* b^* c^*$.

$\{a^n b^n c^n\}$ no es libre de contexto

Nuestra cadena: $s = a^p b^p c^p$. En los dos casos elegimos $i = 2$.

Caso 1: v e y de un solo tipo cada una

uv^2xy^2z sigue ordenada. Entre las dos tocan a lo sumo **dos** de los tres símbolos: una cantidad crece y otra se queda en p.

Caso 2: v o y tiene dos símbolos distintos

Si $v = a^j b^k$: $vv = a^j b^k a^j b^k$.
Hay una a después de una b: ya no es $a^* b^* c^*$.

Error común

Olvidar el caso 2. Si $v = ab$, bombear no "agrega una a y una b" a sus bloques: mete una a en medio de las b. Tampoco está en el lenguaje, pero por **otra razón**.

$\{ ww \}$: un intento que falla

Parece hermano de $\{ww^R\}$, pero la pila devuelve al revés: lo contrario de lo que pide ww.

- Primera idea: $s = 0^p 1 0^p 1$.

$\{ ww \}$: un intento que falla

Parece hermano de $\{ww^R\}$, pero la pila devuelve al revés: lo contrario de lo que pide ww.

- Primera idea: $s = 0^p 1 0^p 1$.
- El adversario divide a los dos lados del 1 del medio:

$$u = 0^{p-1} \quad v = 0 \quad x = 1 \quad y = 0 \quad z = 0^{p-1} 1$$

$\{ ww \}$: un intento que falla

Parece hermano de $\{ww^R\}$, pero la pila devuelve al revés: lo contrario de lo que pide ww.

- Primera idea: $s = 0^p 1 0^p 1$.
- El adversario divide a los dos lados del 1 del medio:

$$u = 0^{p-1} \quad v = 0 \quad x = 1 \quad y = 0 \quad z = 0^{p-1} 1$$

- $|vy| = 2$, $|vxy| = 3 \leq p$, y para todo i: $uv^i xy^i z = 0^{p-1+i}\, 1\, 0^{p-1+i}\, 1 \in \{ww\}$.

$\{ ww \}$: un intento que falla

Parece hermano de $\{ww^R\}$, pero la pila devuelve al revés: lo contrario de lo que pide ww.

- Primera idea: $s = 0^p 1 0^p 1$.
- El adversario divide a los dos lados del 1 del medio:

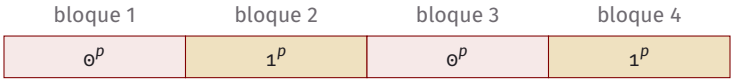
$$u = 0^{p-1} \quad v = 0 \quad x = 1 \quad y = 0 \quad z = 0^{p-1}1$$

- $|vy| = 2$, $|vxy| = 3 \leq p$, y para todo i: $uv^i xy^i z = 0^{p-1+i}\, 1\, 0^{p-1+i}\, 1 \in \{ww\}$.

Lo que enseña

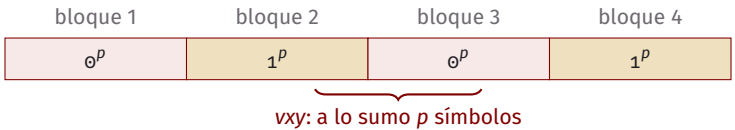
Esta s no sirve. Hay que alejar los ceros de las dos mitades **más de** p, para que la ventana no llegue de unos a otros.

$\{ ww \}$: la cadena buena



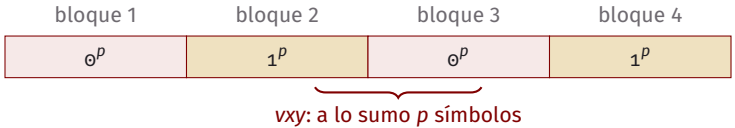
- $s = 0^p 1^p 0^p 1^p$. **Pareja** de un bloque: el otro del mismo símbolo.

$\{ ww \}$: la cadena buena



- $s = 0^p 1^p 0^p 1^p$. **Pareja** de un bloque: el otro del mismo símbolo.
- La ventana toca **un bloque o dos vecinos**. Si toca un bloque, **no toca a su pareja**.

$\{ ww \}$: la cadena buena



- $s = 0^p 1^p 0^p 1^p$. **Pareja** de un bloque: el otro del mismo símbolo.
- La ventana toca **un bloque o dos vecinos**. Si toca un bloque, **no toca a su pareja**.

Lema de los cuatro bloques

Si $0^a 1^b 0^c 1^d = ww$ con $a, b, c, d \geq 1$, entonces $a = c$ y $b = d$.

La cadena tiene 3 cambios de símbolo. Si w tiene m, entonces ww tiene $2m + 1$. Luego $m = 1$ y $w = 0^e 1^f$.

$\{ ww \}$: los dos casos

Caso 1: v e y dentro de un bloque cada una

Elegimos $i = 2$. Un bloque crece, y su pareja sigue midiendo p.

Cuatro bloques con $a \neq c$ o $b \neq d$: por el lema, no es ww.

$\{ ww \}$: los dos casos

Caso 1: v e y dentro de un bloque cada una

Elegimos $i = 2$. Un bloque crece, y su pareja sigue midiendo p.

Cuatro bloques con $a \neq c$ o $b \neq d$: por el lema, no es ww.

Caso 2: v o y abarca un borde

Elegimos $i = 0$: borrar no desordena. Dos bloques vecinos pierden símbolos, **sin vaciarse**, y sus parejas siguen en p.

Otra vez cuatro bloques, y el lema. (Con $i = 2$ saldrían seis.)

$\{ ww \}$: los dos casos

Caso 1: v e y dentro de un bloque cada una

Elegimos $i = 2$. Un bloque crece, y su pareja sigue midiendo p.

Cuatro bloques con $a \neq c$ o $b \neq d$: por el lema, no es ww.

Caso 2: v o y abarca un borde

Elegimos $i = 0$: borrar no desordena. Dos bloques vecinos pierden símbolos, **sin vaciarse**, y sus parejas siguen en p.

Otra vez cuatro bloques, y el lema. (Con $i = 2$ saldrían seis.)

- ¿Por qué no se vacían? Se borran a lo sumo p símbolos, al menos uno de cada bloque: a lo sumo $p - 1$ de cada uno.

$\{ ww \}$: los dos casos

Caso 1: v e y dentro de un bloque cada una

Elegimos $i = 2$. Un bloque crece, y su pareja sigue midiendo p.

Cuatro bloques con $a \neq c$ o $b \neq d$: por el lema, no es ww.

Caso 2: v o y abarca un borde

Elegimos $i = 0$: borrar no desordena. Dos bloques vecinos pierden símbolos, **sin vaciarse**, y sus parejas siguen en p.

Otra vez cuatro bloques, y el lema. (Con $i = 2$ saldrían seis.)

- ¿Por qué no se vacían? Se borran a lo sumo p símbolos, al menos uno de cada bloque: a lo sumo $p - 1$ de cada uno.
- La condición 3 fue esencial: sin ella, v en el bloque 1 e y en el bloque 3 crecerían juntos.

$\{a^i b^j c^k : i \leq j \leq k\}$: bombear hacia abajo

Con $s = a^p b^p c^p$ y siempre el exponente 2, el adversario gana: pone v e y en las c. Que crezca lo de la derecha no molesta.

$\{a^i b^j c^k : i \leq j \leq k\}$: bombear hacia abajo

Con $s = a^p b^p c^p$ y siempre el exponente 2, el adversario gana: pone v e y en las c. Que crezca lo de la derecha no molesta.

- **Caso 1: v o y abarca un borde.** Exponente 2: la cadena queda desordenada.

$\{a^i b^j c^k : i \leq j \leq k\}$: bombear hacia abajo

Con $s = a^p b^p c^p$ y siempre el exponente 2, el adversario gana: pone v e y en las c. Que crezca lo de la derecha no molesta.

- **Caso 1: v o y abarca un borde.** Exponente 2: la cadena queda desordenada.
- **Caso 2: en vy no hay ninguna** a**.** Exponente **0**: siguen las p letras a, y quedan menos b o menos c. Hay más a que b, o más b que c.

$\{a^i b^j c^k : i \leq j \leq k\}$: bombear hacia abajo

Con $s = a^p b^p c^p$ y siempre el exponente 2, el adversario gana: pone v e y en las c. Que crezca lo de la derecha no molesta.

- **Caso 1:** v **o** y **abarca un borde.** Exponente 2: la cadena queda desordenada.
- **Caso 2: en** vy **no hay ninguna** a**.** Exponente **0**: siguen las p letras a, y quedan menos b o menos c. Hay más a que b, o más b que c.
- **Caso 3: en** vy **hay alguna** a**.** Exponente 2: más de p letras a. Las b o las c no se tocaron y siguen en p: hay más a que b, o más a que c.

$\{a^i b^j c^k : i \leq j \leq k\}$: bombear hacia abajo

Con $s = a^p b^p c^p$ y siempre el exponente 2, el adversario gana: pone v e y en las c. Que crezca lo de la derecha no molesta.

- **Caso 1: v o y abarca un borde.** Exponente 2: la cadena queda desordenada.
- **Caso 2: en vy no hay ninguna a.** Exponente **0**: siguen las p letras a, y quedan menos b o menos c. Hay más a que b, o más b que c.
- **Caso 3: en vy hay alguna a.** Exponente 2: más de p letras a. Las b o las c no se tocaron y siguen en p: hay más a que b, o más a que c.

Lo que enseña

El exponente se elige **en cada caso**. A veces hay que bombear hacia abajo.

El juego, jugado por la computadora

build/pump_demo: dos de sus cinco tablas

```
{ ww },  s = 0^p 1 0^p 1  (una mala elección)
   p   |s|  divisiones  fuera con i=2  solo con i=0  sobreviven
   1     4           8              8             0           0
   2     6          37             37             0           0
   3     8         105            104             0           1
   4    10         235            232             0           3
   5    12         455            448             0           7
   6    14         798            785             0          13
  una división que sobrevive con p = 6: u = 00, v = 0, x = 0001, y = 0, z = 000001

{ a^i b^j c^k : i ≤ j ≤ k },  s = a^p b^p c^p
   p   |s|  divisiones  fuera con i=2  solo con i=0  sobreviven
   1     3           6              4             2           0
   2     6          37             26            11           0
   3     9         121             88            33           0
   4    12         295            219            76           0
   5    15         605            456           149           0
   6    18        1106            843           263           0
```

- Para una s y un p concretos, las divisiones son finitas: se prueban todas.

El juego, jugado por la computadora

build/pump_demo: dos de sus cinco tablas

```
{ ww },  s = 0^p 1 0^p 1  (una mala elección)
   p   |s|  divisiones  fuera con i=2  solo con i=0  sobreviven
   1     4           8              8             0           0
   2     6          37             37             0           0
   3     8         105            104             0           1
   4    10         235            232             0           3
   5    12         455            448             0           7
   6    14         798            785             0          13
  una división que sobrevive con p = 6: u = 00, v = 0, x = 0001, y = 0, z = 000001

{ a^i b^j c^k : i ≤ j ≤ k },  s = a^p b^p c^p
   p   |s|  divisiones  fuera con i=2  solo con i=0  sobreviven
   1     3           6              4             2           0
   2     6          37             26            11           0
   3     9         121             88            33           0
   4    12         295            219            76           0
   5    15         605            456           149           0
   6    18        1106            843           263           0
```

- Para una s y un p concretos, las divisiones son finitas: se prueban todas.
- Mala elección: desde $p = 3$ alguna sobrevive.

El juego, jugado por la computadora

build/pump_demo: dos de sus cinco tablas

```
{ ww },  s = 0^p 1 0^p 1  (una mala elección)
   p   |s|  divisiones  fuera con i=2  solo con i=0  sobreviven
   1     4           8              8             0           0
   2     6          37             37             0           0
   3     8         105            104             0           1
   4    10         235            232             0           3
   5    12         455            448             0           7
   6    14         798            785             0          13
  una división que sobrevive con p = 6: u = 00, v = 0, x = 0001, y = 0, z = 000001

{ a^i b^j c^k : i ≤ j ≤ k },  s = a^p b^p c^p
   p   |s|  divisiones  fuera con i=2  solo con i=0  sobreviven
   1     3           6              4             2           0
   2     6          37             26            11           0
   3     9         121             88            33           0
   4    12         295            219            76           0
   5    15         605            456           149           0
   6    18        1106            843           263           0
```

- Para una s y un p concretos, las divisiones son finitas: se prueban todas.
- Mala elección: desde $p = 3$ alguna sobrevive.
- Con $i \leq j \leq k$: 263 de 1106 solo caen hacia abajo.

El juego, jugado por la computadora

build/pump_demo: dos de sus cinco tablas

```
{ ww },  s = 0^p 1 0^p 1  (una mala elección)
   p   |s|  divisiones  fuera con i=2  solo con i=0  sobreviven
   1     4           8              8             0           0
   2     6          37             37             0           0
   3     8         105            104             0           1
   4    10         235            232             0           3
   5    12         455            448             0           7
   6    14         798            785             0          13
  una división que sobrevive con p = 6: u = 00, v = 0, x = 0001, y = 0, z = 000001

{ a^i b^j c^k : i ≤ j ≤ k },  s = a^p b^p c^p
   p   |s|  divisiones  fuera con i=2  solo con i=0  sobreviven
   1     3           6              4             2           0
   2     6          37             26            11           0
   3     9         121             88            33           0
   4    12         295            219            76           0
   5    15         605            456           149           0
   6    18        1106            843           263           0
```

- Para una s y un p concretos, las divisiones son finitas: se prueban todas.
- Mala elección: desde $p = 3$ alguna sobrevive.
- Con $i \leq j \leq k$: 263 de 1106 solo caen hacia abajo.
- **No demuestra nada**: prueba algunos p. Sirve para encontrar el caso olvidado.

SECCIÓN 09

Propiedades de cierre

09

Unión, concatenación y estrella: sí

Con gramáticas G_1 y G_2, variables iniciales S_1 y S_2, y una variable inicial nueva S:

Lenguaje	Reglas nuevas
$A_1 \cup A_2$	$S \to S_1 \mid S_2$
$A_1 A_2$	$S \to S_1 S_2$
A_1^*	$S \to S_1 S \mid \varepsilon$

Unión, concatenación y estrella: sí

Con gramáticas G_1 y G_2, variables iniciales S_1 y S_2, y una variable inicial nueva S:

Lenguaje	**Reglas nuevas**
$A_1 \cup A_2$	$S \to S_1 \mid S_2$
$A_1 A_2$	$S \to S_1 S_2$
A_1^*	$S \to S_1 S \mid \varepsilon$

- Funcionan porque las variables de G_1 solo aparecen en reglas de G_1.

Unión, concatenación y estrella: sí

Con gramáticas G_1 y G_2, variables iniciales S_1 y S_2, y una variable inicial nueva S:

Lenguaje	Reglas nuevas
$A_1 \cup A_2$	$S \to S_1 \mid S_2$
$A_1 A_2$	$S \to S_1 S_2$
A_1^*	$S \to S_1 S \mid \varepsilon$

- Funcionan porque las variables de G_1 solo aparecen en reglas de G_1.

Error común

Juntar las reglas sin cambiar los nombres. Con $S \to \mathtt{a}A$, $A \to \mathtt{b}$ y $S \to \mathtt{c}A$, $A \to \mathtt{d}$, la mezcla genera también $\mathtt{ad}$ y $\mathtt{cb}$.

Intersección y complemento: no

$$A = \{ a^n b^n c^m \} \qquad B = \{ a^m b^n c^n \}$$

- Los dos son libres de contexto: $\{a^n b^n\}$ seguido de c^*, y a^* seguido de $\{b^n c^n\}$.

Intersección y complemento: no

$$A = \{ a^n b^n c^m \} \qquad B = \{ a^m b^n c^n \}$$

- Los dos son libres de contexto: $\{a^n b^n\}$ seguido de c^*, y a^* seguido de $\{b^n c^n\}$.
- $A \cap B = \{ a^n b^n c^n \}$, que no es libre de contexto.

Intersección y complemento: no

$$A = \{ \mathtt{a}^n \mathtt{b}^n \mathtt{c}^m \} \qquad B = \{ \mathtt{a}^m \mathtt{b}^n \mathtt{c}^n \}$$

- Los dos son libres de contexto: $\{\mathtt{a}^n \mathtt{b}^n\}$ seguido de $\mathtt{c}^*$, y $\mathtt{a}^*$ seguido de $\{\mathtt{b}^n \mathtt{c}^n\}$.
- $A \cap B = \{ \mathtt{a}^n \mathtt{b}^n \mathtt{c}^n \}$, que no es libre de contexto.
- **Complemento:** si la clase fuera cerrada, con la unión y De Morgan lo sería bajo intersección: $A \cap B = \overline{\overline{A} \cup \overline{B}}$.

Intersección y complemento: no

$$A = \{ a^n b^n c^m \} \qquad B = \{ a^m b^n c^n \}$$

- Los dos son libres de contexto: $\{a^n b^n\}$ seguido de c^*, y a^* seguido de $\{b^n c^n\}$.
- $A \cap B = \{ a^n b^n c^n \}$, que no es libre de contexto.
- **Complemento:** si la clase fuera cerrada, con la unión y De Morgan lo sería bajo intersección: $A \cap B = \overline{\overline{A} \cup \overline{B}}$.

Por qué no hay producto

Con dos DFA se llevaba un par de estados. Con dos PDA harían falta **dos pilas**, y hay una sola.

Intersección con un lenguaje regular: sí

Teorema

Si A es libre de contexto y B es regular, entonces $A \cap B$ es libre de contexto.

- El producto de un PDA con un DFA: estados (p, m), **una sola pila**, la del PDA.

Intersección con un lenguaje regular: sí

Teorema

Si A es libre de contexto y B es regular, entonces $A \cap B$ es libre de contexto.

- El producto de un PDA con un DFA: estados (p, m), **una sola pila**, la del PDA.
- Si el PDA lee a: de (p, m) a $(p', \delta_M(m, a))$. Si no lee: de (p, m) a (p', m).

Intersección con un lenguaje regular: sí

Teorema

Si A es libre de contexto y B es regular, entonces $A \cap B$ es libre de contexto.

- El producto de un PDA con un DFA: estados (p, m), **una sola pila**, la del PDA.
- Si el PDA lee a: de (p, m) a $(p', \delta_M(m, a))$. Si no lee: de (p, m) a (p', m).

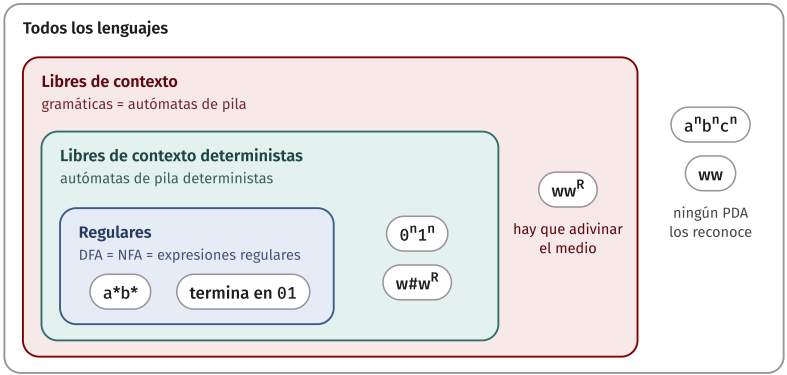
Para qué sirve: simplificar antes de bombear

K: las cadenas con la misma cantidad de a, de b y de c, en cualquier orden.

$$K \cap \mathrm{a}^*\mathrm{b}^*\mathrm{c}^* = \{\, \mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \,\}$$

Si K fuera libre de contexto, este también lo sería. No lo es: K tampoco.

El mapa hasta ahora



Cada inclusión es estricta: 0^n1^n no es regular (semana 4), ww^R no tiene PDA determinista, y $a^nb^nc^n$ y ww no son libres de contexto (lema de bombeo).

De los regulares a los libres de contexto

	Reg.	**L. de c.**
Unión, concatenación, estrella	sí	sí
Intersección	sí	**no**
Complemento	sí	**no**
Intersección con un regular	sí	sí
¿No determinismo: más poder?	no	**sí**
Decidir si w está	lineal	$O(n^3)$

De los regulares a los libres de contexto

	Reg.	**L. de c.**
Unión, concatenación, estrella	sí	sí
Intersección	sí	**no**
Complemento	sí	**no**
Intersección con un regular	sí	sí
¿No determinismo: más poder?	no	**sí**
Decidir si w está	lineal	$O(n^3)$

- Regulares: DFA $=$ NFA $=$ expresiones regulares. Libres de contexto: gramáticas $=$ PDA.

De los regulares a los libres de contexto

	Reg.	**L. de c.**
Unión, concatenación, estrella	sí	sí
Intersección	sí	**no**
Complemento	sí	**no**
Intersección con un regular	sí	sí
¿No determinismo: más poder?	no	**sí**
Decidir si w está	lineal	$O(n^3)$

- Regulares: DFA $=$ NFA $=$ expresiones regulares. Libres de contexto: gramáticas $=$ PDA.
- Lo que falta: **volver** sobre lo guardado sin destruirlo.

SECCIÓN 10

Opcional: un parser real

10



Opcional: lexer y parser

Un compilador no usa CYK ni una búsqueda.

- **Lexer:** caracteres → tokens. Cada token es una expresión regular, como `[0-9]+`: un autómata finito.

expr.h, resumido

```
// E -> E + T | T, como un ciclo.
Parsed expression() {
    Parsed left = term();
    while (peek().kind == TokenKind::Plus) {
        next++;
        Parsed right = term();
        left = {left.value + right.value,
                "(" + left.tree + " + "
                    + right.tree + ")"};
    }
    return left;
}
```

Opcional: lexer y parser

Un compilador no usa CYK ni una búsqueda.

- **Lexer:** caracteres → tokens. Cada token es una expresión regular, como `[0-9]+`: un autómata finito.
- **Parser:** una gramática sin ambigüedad, elegida para un PDA **determinista**.

$$E \to E + T \mid T \qquad T \to T * F \mid F$$

$$F \to (E) \mid \text{número}$$

expr.h, resumido

```
1  // E -> E + T | T, como un ciclo.
2  Parsed expression() {
3      Parsed left = term();
4      while (peek().kind == TokenKind::Plus) {
5          next++;
6          Parsed right = term();
7          left = {left.value + right.value,
8                  "(" + left.tree + " + "
9                      + right.tree + ")"};
10     }
11     return left;
12 }
```

Opcional: lexer y parser

Un compilador no usa CYK ni una búsqueda.

- **Lexer:** caracteres → tokens. Cada token es una expresión regular, como `[0-9]+`: un autómata finito.
- **Parser:** una gramática sin ambigüedad, elegida para un PDA **determinista**.

$$E \to E + T \mid T \qquad T \to T * F \mid F$$
$$F \to (E) \mid \text{número}$$

expr.h, resumido

```
1  // E -> E + T | T, como un ciclo.
2  Parsed expression() {
3      Parsed left = term();
4      while (peek().kind == TokenKind::Plus) {
5          next++;
6          Parsed right = term();
7          left = {left.value + right.value,
8                  "(" + left.tree + " + "
9                      + right.tree + ")"};
10     }
11     return left;
12 }
```

Una función por variable; el siguiente token decide la regla.
$E \to E + T$ tal cual se llamaría a sí misma sin consumir nada.

Opcional: tiempo lineal

terminal

```
$ build/parse_expr '2 + 3 * 4' '(2 + 3) * 4'
2 + 3 * 4
  tokens:  NUM(2) + NUM(3) * NUM(4)
  árbol:   (2 + (3 * 4))
  valor:   14
  5 tokens, 6 llamadas a las funciones del parser
(2 + 3) * 4
  tokens:  ( NUM(2) + NUM(3) ) * NUM(4)
  árbol:   ((2 + 3) * 4)
  valor:   20
  7 tokens, 9 llamadas a las funciones del parser
```

- La precedencia no está en ningún `if`: sale de la forma de la gramática.

Opcional: tiempo lineal

terminal

```
$ build/parse_expr '2 + 3 * 4' '(2 + 3) * 4'
2 + 3 * 4
  tokens:  NUM(2) + NUM(3) * NUM(4)
  árbol:   (2 + (3 * 4))
  valor:   14
  5 tokens, 6 llamadas a las funciones del parser
(2 + 3) * 4
  tokens:  ( NUM(2) + NUM(3) ) * NUM(4)
  árbol:   ((2 + 3) * 4)
  valor:   20
  7 tokens, 9 llamadas a las funciones del parser
```

- La precedencia no está en ningún `if`: sale de la forma de la gramática.
- La posición `next` **solo avanza**: cada token se consume una vez. Las llamadas nunca pasan de tres por token.

Opcional: tiempo lineal

terminal

```
$ build/parse_expr '2 + 3 * 4' '(2 + 3) * 4'
2 + 3 * 4
  tokens:  NUM(2) + NUM(3) * NUM(4)
  árbol:   (2 + (3 * 4))
  valor:   14
  5 tokens, 6 llamadas a las funciones del parser
(2 + 3) * 4
  tokens:  ( NUM(2) + NUM(3) ) * NUM(4)
  árbol:   ((2 + 3) * 4)
  valor:   20
  7 tokens, 9 llamadas a las funciones del parser
```

- La precedencia no está en ningún `if`: sale de la forma de la gramática.
- La posición `next` **solo avanza**: cada token se consume una vez. Las llamadas nunca pasan de tres por token.
- Es un PDA determinista: su pila es la **pila de llamadas**.

Opcional: tiempo lineal

terminal

```
$ build/parse_expr '2 + 3 * 4' '(2 + 3) * 4'
2 + 3 * 4
  tokens:  NUM(2) + NUM(3) * NUM(4)
  árbol:   (2 + (3 * 4))
  valor:   14
  5 tokens, 6 llamadas a las funciones del parser
(2 + 3) * 4
  tokens:  ( NUM(2) + NUM(3) ) * NUM(4)
  árbol:   ((2 + 3) * 4)
  valor:   20
  7 tokens, 9 llamadas a las funciones del parser
```

- La precedencia no está en ningún `if`: sale de la forma de la gramática.
- La posición `next` **solo avanza**: cada token se consume una vez. Las llamadas nunca pasan de tres por token.
- Es un PDA determinista: su pila es la **pila de llamadas**.
- Es la primera mitad del intérprete que escribieron en Introducción a la Programación.

SECCIÓN 11

Cierre

11

Resumen

- Un **PDA** es un NFA con una pila. Acepta si alguna ejecución lee toda la entrada y termina en F.
- Se marca el fondo; el no determinismo adivina lo que la entrada no dice. Una pila compara **una sola pareja**.
- **Gramática** $\to$ **PDA:** la pila guarda lo que falta derivar. **PDA** $\to$ **gramática:** las variables A_{pq}.
- Un lenguaje es libre de contexto si y solo si algún PDA lo reconoce. Los deterministas reconocen menos.
- **Lema de bombeo:** $s = uvxyz$, $uv^ixy^iz \in A$, $|vy| > 0$, $|vxy| \leq p$. Sale de una variable repetida en un árbol alto.
- Al usarlo: todas las divisiones, también con v o y sobre un borde. A veces, hacia abajo.
- No son libres de contexto: $\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n\}$, $\{ww\}$, $\{\mathrm{a}^i\mathrm{b}^j\mathrm{c}^k : i \leq j \leq k\}$.
- Cierre: unión, concatenación, estrella e intersección con un regular, sí. Intersección y complemento, no.

Para la próxima clase

- **Laboratorio:** la sesión es para la Práctica 2, sobre las semanas 4 y 5.
- **Opcional:** `code/week-06/lab/pda_sim`, un simulador de PDA con el juez. El enunciado fija el límite de configuraciones.
- **Leer:** en los apuntes de esta semana, la demostración completa de PDA a gramática. Apoyo: Sipser, sec. 2.2 y 2.3.
- **Semana 7:** máquinas de Turing. Una máquina que puede volver sobre lo que escribió.

Pregunta

$\{a^n b^n c^n\}$ no es libre de contexto, y un programa de diez líneas lo decide. ¿Qué hace ese programa que un PDA no puede hacer?