ChatGPT

Unidad 2: Gramáticas y Lenguajes Formales

Símbolos, Alfabetos y Palabras

Alfabeto: Un alfabeto es cualquier conjunto finito no vacío de símbolos. Se denota típicamente con letras griegas mayúsculas, por ejemplo Σ, Γ, Ω. Ejemplos de alfabetos:
- $\Sigma_1 = \{a\}$ es un alfabeto unitario con el símbolo "a".
- $\Sigma_2 = \{0, 1\}$ es el alfabeto binario (símbolos "0" y "1").
- $\Sigma_3 = \{a, b, c, \ldots, z\}$ alfabeto de letras minúsculas en español.

Cada elemento de un alfabeto también se llama **letra** o **carácter**. Es importante no confundir esto con letras de idiomas naturales: por ejemplo, "torre" es una secuencia de cinco letras del alfabeto {a,…,z}, pero en el contexto del ajedrez podría considerarse un único símbolo si definimos Γ = {torre, caballo, …} como alfabeto de piezas de ajedrez.

Dos alfabetos son iguales si contienen exactamente los mismos símbolos. Si todos los símbolos de un alfabeto Ω_1 están en otro Ω_2, se dice que Ω_1 está **incluido** en Ω_2 ($\Omega_1 \subseteq \Omega_2$). Si además Ω_2 tiene símbolos que Ω_1 no tiene, Ω_1 es un subconjunto **propio** de Ω_2 ($\Omega_1 \subset \Omega_2$). El **cardinal** de un alfabeto $|\Omega|$ es la cantidad de símbolos distintos que posee (siempre un número entero positivo).

Palabra o cadena: Es cualquier secuencia finita de símbolos de un alfabeto, escritos uno tras otro. Por ejemplo, tomando $\Sigma_3 = \{a,\ldots,z\}$, "gato" es una palabra de 4 símbolos perteneciente a Σ_3. Se suelen usar letras griegas minúsculas (α, β, γ, ω, etc.) para denotar palabras genéricas. La **concatenación** de palabras es la operación de unir dos palabras en secuencia. Si α = "hola" y β = "mundo", entonces α·β = "holamundo". La concatenación se puede indicar con un punto (α·β) o simplemente yuxtaponiendo (αβ). Esta operación **no es conmutativa** (por ejemplo "holamundo" ≠ "mundohola") pero sí es **asociativa** (el resultado no depende de cómo agrupemos concatenaciones sucesivas).

Ejemplo: Sean α = "abra" y β = "cadabra". Entonces:
- α·β = "abracadabra"
- β·α = "cadabraabra"

Vemos que α·β ≠ β·α (no conmutativa), pero (α·β)·α = α·(β·α) (asociativa), ambas expresiones dan "abracadabraabra".

La concatenación de una palabra consigo misma se puede abreviar con **potenciación**. Por ejemplo α^2 = α·α, α^3 = α·α·α, etc. Si α = "ab", entonces α^2 = "abab", α^3 = "ababab", etc. Más formalmente, para cualquier palabra α sobre algún alfabeto y entero $n \geq 0$, definimos recursivamente: α^0 = λ (la palabra vacía) y α^n = α·α^(n-1) para n > 0.

Longitud: La longitud de una palabra α, denotada |α|, es la cantidad de símbolos que contiene. Ejemplos: si α = "casa", |α| = 4; si β = "blanca", |β| = 6; entonces la palabra concatenada αβ = "casablanca" tiene longitud |αβ| = |α| + |β| = 10. Observar que si una palabra se considera como un único símbolo en cierto contexto (ej. "torre" como pieza de ajedrez), entonces su longitud en ese contexto sería 1 (porque es un solo símbolo del alfabeto de piezas).

Cadena vacía: Denotada usualmente como λ (lambda), es la palabra de longitud cero, que no contiene símbolos. Por definición |λ| = 0. La cadena vacía actúa como **elemento neutro** de la concatenación:

para cualquier palabra α, se cumple α·λ = λ·α = α. Solo existe una cadena vacía (es única para cada alfabeto). Ejemplo: En pseudocódigo, un bucle `for (int k=0; k<10; k++);` tiene un cuerpo vacío equivalente a ejecutar λ en cada iteración.

Toda palabra α tiene una **palabra refleja** o inversa denotada α^{-1}, que es la misma secuencia de símbolos pero en orden inverso. Por ejemplo, si α = "roma", entonces α^{-1} = "amor". Si α = "radar", α^{-1} = "radar" (cuando la palabra coincide con su inversa, es un palíndromo). Un **palíndromo** es justamente una palabra que se lee igual al derecho y al revés (ejemplos: "neuquen", "rotor", "1234321"). En números, también se les llama **capicúas** si tienen esa propiedad.

Prefijos, sufijos y subcadenas: Dada una palabra ω, un **prefijo** es cualquier cadena inicial de ω (incluyendo λ y la propia ω), y un **sufijo** es cualquier cadena final de ω. Una **subcadena** (o **subpalabra**) es cualquier porción consecutiva de símbolos dentro de ω (puede ser central). Se dice que un prefijo o sufijo es **propio** si no es ni la palabra completa ni la vacía. Ejemplo: Sea ω = "prueba". Sus prefijos son: "", "p", "pr", "pru", "prue", "prueb", "prueba"; sus sufijos: "", "a", "ba", "eba", "ueba", "rueba", "prueba". Los prefijos propios serían todos menos "" y "prueba" (lo mismo para sufijos propios). Subcadenas de ω incluyen, por ejemplo: "ru", "rue", "ueb", "b", "ue" etc., cualquier trozo interno. En general, para una palabra de longitud n, hay n+1 prefijos y n+1 sufijos (contando la vacía y la palabra entera).

Operaciones con alfabetos: Dado que un alfabeto es un conjunto, podemos aplicar operaciones de conjuntos:
- **Unión:** $\Sigma_1 \cup \Sigma_2$ (contiene todos los símbolos que están en Σ_1 o en Σ_2).
- **Intersección:** $\Sigma_1 \cap \Sigma_2$ (símbolos comunes a ambos alfabetos).
- **Diferencia:** $\Sigma_1 - \Sigma_2$ (símbolos de Σ_1 que no están en Σ_2).
- **Complemento:** $\bar{\Sigma}$ (si definimos un alfabeto universal de referencia, complemento de Σ serían los símbolos no en Σ, respecto a ese universo).

Además existe una operación de **producto (concatenación) de alfabetos**: Dado Σ_1 y Σ_2, definimos $\Sigma_1 \cdot \Sigma_2 = \{ xy \mid x \in \Sigma_1, y \in \Sigma_2 \}$. Es decir, concatenamos cada símbolo de Σ_1 con cada símbolo de Σ_2, formando palabras de longitud 2. Por ejemplo, {0,1}·{a,b,c} = {"0a","0b","0c","1a","1b","1c"}. Nótese que $\Sigma_1 \cdot \Sigma_2$ no suele ser un alfabeto (sus elementos no son símbolos indivisibles, sino cadenas de longitud 2), pero sirve para construir conjuntos de palabras. Podemos extender esta idea: $\Sigma^1 = \Sigma$ (palabras de longitud 1, o sea el alfabeto mismo), $\Sigma^2 = \Sigma \cdot \Sigma$ (palabras de longitud 2), $\Sigma^3 = \Sigma \cdot \Sigma \cdot \Sigma$ (longitud 3), etc. Más generalmente, definimos la **potencia n-ésima de un alfabeto** recursivamente: Σ^0 = {λ} (por convenio, la única palabra de longitud 0) y Σ^n = Σ^(n-1) · Σ, para $n \geq 1$. Así, Σ *(estrella de Kleene del alfabeto) se define como el conjunto de* todas *las palabras formadas con símbolos de Σ, de cualquier longitud finita:*

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots = \bigcup_{n=0}^{\infty} \Sigma^n.$$

Este es el universo de discurso de Σ, también llamado lenguaje universal sobre Σ. Incluye λ y todas las posibles cadenas. Asimismo, definimos Σ^+ (cierre positivo) = Σ − {λ}, es decir, todas las palabras de longitud ≥ 1.

Ejemplos:
- Si Σ = {0,1}, entonces:
Σ^0 = {λ}
Σ^1 = { "0","1" }
Σ^2 = { "00","01","10","11" } (todas las combinaciones de 2 bits)
Σ = *{λ, "0","1","00","01","10","11","000", ...} (todas las cadenas binarias finitas).*
Σ^+ = *Σ* sin incluir λ.

- Si Σ_e = {a,...,z, A,...,Z} (letras españolas sin acentos), entonces Σ_e *contiene* todas *las cadenas de letras (palabras posibles, incluso sin significado). El conjunto de palabras válidas del idioma (según el diccionario) sería un subconjunto de Σ_e*.

Ejercicios sugeridos (Alfabetos y Palabras)

1. Dado el alfabeto Σ = {a, b, c, d}, considerar las palabras α = "aab", β = "cdee", γ = "eddcc" y δ = "aacddd". Calcular: **a)** $|\alpha|$, $|\beta|$, $|\gamma|$, $|\delta|$ (longitudes de cada palabra). **b)** Las siguientes operaciones: α^2, $\alpha \cdot \delta$, β^{-1}, $(\delta \cdot \alpha)^{-1}$, y $[(\beta \cdot \gamma)^{-1}]^2$.
2. Sea ω = "aacddd". Listar todos sus prefijos propios, todos sus sufijos propios, y dar ejemplos de subcadenas de ω que no sean ni prefijos ni sufijos.
3. Dados dos alfabetos Σ_1 = {a, b, c, d} y Σ_2 = {c, d, e}, determinar: **a)** $(\Sigma_1 \cap \Sigma_2)^+$, **b)** $(\Sigma_1 \cup \Sigma_2)^*$, es decir, describir (por extensión, mostrando las primeras diez cadenas en orden creciente de longitud) los lenguajes formados por la clausura positiva de la intersección y la clausura de la unión.

Lenguajes Formales y Operaciones

Lenguaje: En forma general, un **lenguaje** (formal) sobre un alfabeto Σ es cualquier conjunto *posiblemente infinito* de palabras formadas con símbolos de Σ. Es decir, $L \subseteq \Sigma^*$. Un lenguaje puede ser finito (contiene un número limitado de palabras) o infinito. Ejemplos:
- L = {"hola","mundo"} es un lenguaje finito sobre Σ = {a,...,z}.
- El conjunto de todos los números binarios con cantidad par de símbolos es un lenguaje infinito sobre Σ = {0,1}.
- Σ *en sí mismo (el conjunto de todas las palabras sobre Σ) es un lenguaje particular llamado lenguaje universal** (en ese alfabeto). En cambio, el conjunto vacío ∅ también es un lenguaje (no contiene ninguna palabra).
- {λ} (el conjunto que contiene solo la palabra vacía) es un lenguaje distinto de ∅; tiene cardinalidad 1.

Como los lenguajes son conjuntos, podemos aplicar la teoría de conjuntos: Se dice L_1 es **sublenguaje** (subconjunto) de L_2 si toda palabra de L_1 pertenece también a L_2. Si además L_2 tiene al menos una palabra que L_1 no tiene, L_1 es subconjunto propio de L_2. Dos lenguajes son iguales ($L_1 = L_2$) si contienen exactamente las mismas palabras.

Operaciones con lenguajes: Dado $L_1, L_2 \subseteq \Sigma$ *(lenguajes sobre el mismo alfabeto Σ):*
- *Unión:* $L_1 \cup L_2 = \{ \alpha \mid \alpha \in L_1 \text{ o } \alpha \in L_2 \}$. *(Todas las palabras que estén en al menos uno de los lenguajes).*
- *Intersección:* $L_1 \cap L_2 = \{ \alpha \mid \alpha \in L_1 \text{ y } \alpha \in L_2 \}$. *(Palabras comunes a ambos).*
- *Diferencia:* $L_1 - L_2 = \{ \alpha \mid \alpha \in L_1 \text{ y } \alpha \notin L_2 \}$. *(Palabras de L_1 que no están en L_2).*
- *Complemento:* $\neg L_1$ *(o* $\Sigma - L_1$*)* $= \{ \alpha \in \Sigma \mid \alpha \notin L_1 \}$. *(Todas las palabras posibles que* no *pertenecen a L_1, tomando Σ* como universo).
- **Concatenación de lenguajes:** $L_1 \cdot L_2 = \{ \alpha\beta \mid \alpha \in L_1, \beta \in L_2 \}$. Es el conjunto de todas las palabras que se obtienen concatenando una palabra de L_1 con otra de L_2. Si L_1 = {"a","b"} y L_2 = {"00","1"}, entonces $L_1 \cdot L_2$ = {"a00","a1","b00","b1"}. En particular, podemos definir $L^2 = L \cdot L$, $L^3 = L \cdot L \cdot L$, etc., y $L^0 = \{\lambda\}$.
- **Clausura Kleene:** L = *conjunto de palabras que se pueden formar concatenando cero, una o más palabras de L (todas las posibles concatenaciones de elementos de L, incluyendo la opción de tomar cero elementos que produce λ). Formalmente,* $L^* = \bigcup_{n=0}^{\infty} L^n$.
- *Clausura positiva:* L^+ = *igual que L* pero excluyendo λ (es decir $L^+ = \bigcup_{n=1}^{\infty} L^n$). Obviamente, si λ ∈ L, entonces $L^+ = L^*$.

Ejemplo (Operaciones): Sea Σ_dec = {0,1,2,3,4,5,6,7,8,9} el alfabeto de dígitos decimales. Definamos estos lenguajes sobre Σ_dec:

- L_1 = {"10","11","12",...,"98","99"} (todos los números de dos dígitos del 10 al 99).
- L_2 = {"numerales de los números naturales de dos dígitos"} – que en realidad describe el mismo conjunto {"10","11",...,"99"}, así que $L_2 = L_1$.
- L_3 = {"10","11","12",...,"19"} (los números del 10 al 19).
- L_4 = {"0","1","2",...,"9","10"} (los numerales del 0 al 10).

Con estos lenguajes:

a) $L_1 = L_2$ (aunque definidos en palabras, resultan ser exactamente las mismas cadenas).

b) $L_3 \subset L_1$ porque cada elemento de L_3 está en L_1, y L_1 tiene más elementos (ej: "20" está en L_1 pero no en L_3).

c) $L_3 \cup L_4$ = {"0","1","2","3","4","5","6","7","8","9","10","11","12","13","14","15","16","17","18","19"} (se combinan sin repetir).

d) $L_3 \cap L_4$ = {"10"} (solo "10" está en ambos).

e) $L_4 - L_3$ = {"0","1","2","3","4","5","6","7","8","9"} (quitamos de L_4 el elemento común "10").

f) $L_3 - L_4$ = {"11","12","13","14","15","16","17","18","19"}.

g) $L_4 \cdot L_3$: todas las concatenaciones posibles de una cadena de L_4 seguida de una de L_3. Por ejemplo "0"·"10" = "010", "0"·"11" = "011", ..., "10"·"19" = "1019". Es un conjunto grande (L_4 tiene 11 elementos y L_3 tiene 10, resultando 110 combinaciones posibles).

h) El complemento de L_1 (en Σ_dec*) serían todas las cadenas de dígitos* no *de longitud 2 (es decir, todos los números que no tienen exactamente dos dígitos).*

i) L_1 (cierre Kleene de L_1) incluiría números formados por la concatenación de cualquier cantidad de números de dos dígitos (ej: "121099" = "12" + "10" + "99" pertenece a L_1*). L_1^+ sería lo mismo pero excluyendo λ (no podemos tomar cero cadenas).

Descripción por extensión vs comprensión: Un lenguaje se puede definir **por extensión** (enumerando todas sus palabras, útil solo si es finito y pequeño) o **por comprensión** (indicando una propiedad que caracterice a sus palabras, o dando un patrón algebraico). Por ejemplo, el lenguaje de "todos los programas correctos en C" es mejor especificarlo por comprensión que intentar listar todos los programas posibles. Algunas maneras de descripción por comprensión:

- En lenguaje natural: *T = {usuarios de números de teléfono fijo válidos en Córdoba al 31/12/2012}*.
- Por notación de conjuntos: *L = { α ∈ {0,1}^ | |α| = 2 } (lenguaje de todas las cadenas de 0s y 1s de longitud 2).*
- *Por fórmula o patrón algebraico:* B = { α = a^n b^m / n ≥ 1, 0 < m < 4 } *(descripción de un patrón de cadenas con ciertas cantidades de* a *seguidas de* b *... aunque este ejemplo parece incompleto, el objetivo es ilustrar la notación).*
- *Combinando notación:* C = { α = Begin^n x^m End^n / n ≥ 1, m ≥ 0 }*, donde "Begin" y "End" se consideran símbolos indivisibles (este describe cadenas que empiezan con n veces "Begin", luego m veces "x", y cierran con n veces "End").

Es importante notar que algunas descripciones coloquiales pueden ser concisas pero imprecisas. Por ejemplo, definir P = {programas fuente correctos en C} nos obliga a aclarar qué significa "correcto" y cómo verificarlo. En la práctica, para determinar si un programa pertenece a P, debemos ver si cumple todas las reglas sintácticas de C, lo cual típicamente hace un compilador. Esto motiva la necesidad de una descripción formal (gramática) que establezca exactamente qué cadenas de caracteres (programas) son sintácticamente válidas.

Ejercicios sugeridos (Lenguajes y Operaciones)

1. Definir por **extensión** (enumerando suficientes elementos para notar el patrón) los siguientes lenguajes dados por comprensión:
   a. L_1 = { d^n e^m / n ≥ 0 ∧ m ≥ 2 } sobre Σ_1 = {a,b,c,d}∪{e} (palabras con algunos d seguidos de al menos dos e).

b. L_2 = { a^r c d^r / r ≥ 1 } sobre Σ_1 = {a,c,d} (palabras que comienzan con r veces 'a', luego 'c', y terminan con r veces 'd').
c. L_3 = { 0^a 1^2^a / a ≥ 0 } sobre Σ_3 = {0,1,2} (palabras formadas por cierta cantidad de '0', luego dos '1', luego la misma cantidad de '2' que de '0').

2. Sea L = {w ∈ {0,1}^ | *w tiene longitud par}. Expresar L por comprensión y por extensión (mostrando las primeras 10 palabras). Luego determinar: a) un lenguaje L' tal que L = (L')* (pista: encontrar conjunto de "bloques" cuyo cierre genere exactamente las cadenas de longitud par). **b)** Si E = {0, 11}, ¿es cierto que L = E*? justificar.
3. Dados los lenguajes L_1 = {"ab", "aba"} y L_2 = {"ba", "aba"}, calcular:
a) $L_1 \cup L_2$,
b) $L_1 \cap L_2$,
c) $L_1 \cdot L_2$,
d) $L_2 \cdot L_1$,
e) L_1* (describir algunas palabras del infinito conjunto resultante).

Gramáticas Formales

Para describir con precisión qué cadenas pertenecen a un lenguaje, introducimos las **gramáticas formales**. Una gramática formal proporciona reglas para **generar** (o derivar) las palabras de un lenguaje.

Reglas de reescritura (producciones): Una **producción** es una regla que indica cómo reemplazar una cadena por otra. En su forma general es un par (α, β) de palabras, escrito como α := β. Se lee "α produce β" o "α puede reescribirse como β". Al símbolo ":=" lo llamamos **símbolo de producción**. Aquí α (lado izquierdo) y β (lado derecho) son secuencias de símbolos sobre algún alfabeto relevante. Durante la aplicación de reglas, se identifica α como una subcadena de alguna palabra y se sustituye por β.

Generalmente, restringiremos que α contenga al menos un símbolo especial llamado **no terminal** (ver definición de gramática más abajo), para que la producción tenga el sentido de reemplazar un símbolo abstracto por una secuencia concreta. Así evitamos reglas triviales que no hacen avanzar la derivación (excepto el caso especial de derivar la cadena vacía).

Derivación directa: Aplicar una producción a una palabra para obtener otra palabra se llama **derivación directa** (un solo paso). Si tenemos δ = x α y y aplicamos la producción α := β (si α aparece como subcadena en δ), obtenemos δ ⇒ x β y. Notación: δ → φ indica que φ se deriva directamente de δ mediante alguna producción.

Ejemplo: Sea la palabra δ = "casablanca". Si tenemos la producción "blanca := negra", podemos aplicar esta regla donde "blanca" aparece en δ (al final) y obtener "casanegra". Escribimos:
casablanca → casanegra.
Si además tuviéramos una producción "a := e", podríamos seguir derivando: casanegra → casanegre (reemplazando una 'a' por 'e'). En sucesivos pasos podríamos aplicar "a := e" en diferentes posiciones, por ejemplo:
casablanca → casanegra → cesanegra → cesenegra → cesenegre.
Cada flecha es una derivación directa aplicando una regla. Vemos que en varios pasos "casablanca" se transformó en "cesenegre" aplicando reglas en secuencia.

Gramática formal (definición): Una gramática G se define como una cuádrupla G = (Σ_T, Σ_N, S, P) donde:
- Σ_T es el **alfabeto de símbolos terminales**: los símbolos "finales" que componen las cadenas del

lenguaje generado. Son los caracteres concretos del lenguaje (por ejemplo letras, dígitos, signos de puntuación, etc., según el caso).
- Σ_N es el **alfabeto de símbolos no terminales** (o **variables**): símbolos auxiliares usados para definir estructuras sintácticas en las producciones. Nunca aparecerán en las cadenas finales del lenguaje; solo sirven durante las derivaciones. Por convenio, Σ_T y Σ_N son disjuntos (Σ_T ∩ Σ_N = ∅) y habitualmente se usan letras mayúsculas para no terminales, minúsculas (y otros símbolos) para terminales.
- S ∈ Σ_N es un símbolo distinguido llamado **axioma** o **símbolo inicial**. Es el punto de partida desde el cual se generan las cadenas del lenguaje.
- P es el conjunto de **producciones** (reglas de reescritura). Cada producción tiene la forma α := β con α, β ∈ (Σ_T ∪ Σ_N) *(cadenas de cualquier símbolo) pero α debe contener al menos un símbolo no terminal*. Esto asegura que podemos sustituir al menos un no terminal en cada paso, avanzando en generar terminales.

La idea es: comenzando desde S, aplicando secuencialmente las producciones de P, deberemos poder obtener todas las cadenas de terminales deseadas. Es decir, el **lenguaje generado** por la gramática G se define como:

$$L(G) = \{\, \omega \in \Sigma_T^* \mid S \to^* \omega \,\},$$

donde S →* *ω indica que existe una secuencia finita de derivaciones que lleva de S a ω (*→* representa la derivación en cero o más pasos). Cualquier cadena compuesta **solo** de terminales obtenida a partir de S es una **palabra generada** (pertenece al lenguaje). En cambio, las cadenas mixtas (con terminales y no terminales) que aparecen transitoriamente durante el proceso se llaman **formas sentenciales** o **metapalabras**. La derivación continúa hasta que ya no quedan no terminales por reemplazar, obteniendo una sentencia formada únicamente por terminales.

Dos gramáticas distintas pueden generar el mismo lenguaje (por ejemplo, se pueden usar diferentes no terminales o reglas y describir las mismas cadenas). Mientras las terminales coincidan y las producciones permitan formar exactamente las mismas palabras terminales, las gramáticas se consideran equivalentes en poder descriptivo.

Notación BNF: Es común escribir las producciones en un formato compacto llamado **Backus-Naur Form**. En BNF:
- Se escriben los **no terminales** entre ángulos `<...>`. (A veces podemos omitir los ángulos si queda claro).
- Una producción se escribe `<A> := α`, donde `<A>` es un no terminal y α una secuencia de símbolos (terminales o no terminales).
- Si hay múltiples producciones con el mismo lado izquierdo, se combinan separando las alternativas con `|`. Por ejemplo, en lugar de escribir `<A> := α₁` y `<A> := α₂` por separado, podemos escribir `<A> := α₁ | α₂`. Esto significa "A produce α₁ o α₂". Esta es solo una abreviatura: conceptualmente siguen siendo dos reglas distintas.

En nuestros ejemplos, solemos usar letras minúsculas, dígitos y símbolos puntuales para los terminales, y letras mayúsculas (o nombres descriptivos) para los no terminales. Por ejemplo:

```
<Frase> := <Sujeto> <Predicado>
<Sujeto> := <Artículo> <Sustantivo>
<Predicado> := <Verbo> <ObjetoDirecto>
<ObjetoDirecto> := <Preposición> <Sustantivo> | <Sustantivo>
```

```
<Artículo> := el | la
<Sustantivo> := gato | rata
<Verbo> := corre | come
<Preposición> := a | con
```

Aquí `<Frase>` es el axioma. Los símbolos entre `< >` son no terminales, y las palabras como "el", "gato" son terminales. Con estas reglas podemos derivar frases como "el gato corre a la rata", "la rata come con el gato", etc. Este ejemplo ilustra cómo una gramática formal genera oraciones de estructura válida combinando categorías (no terminales) que finalmente son reemplazadas por palabras específicas (terminales).

Derivaciones sucesivas: Cuando aplicamos una secuencia de producciones desde S hasta obtener una palabra, eso es una **derivación** completa. Podemos anotar la secuencia indicando en cada flecha qué producción usamos. Ejemplo: Supongamos las producciones: 1) `S := aA`, 2) `A := bA`, 3) `A := c`. Una posible derivación:
S –(1)→ aA –(2)→ abA –(2)→ abbA –(3)→ abbc.
En cada paso indicamos el número de regla aplicada. Al final, S → *"abbc". Si varias producciones son aplicables en algún paso, la gramática permite diferentes** derivaciones para la misma cadena (distintos caminos de generación). Puede haber caminos que no logran terminar en solo terminales (derivaciones sin salida), por ejemplo si uno reemplaza símbolos en bucle. Pero mientras exista al menos una derivación exitosa desde S para una cadena ω, entonces ω ∈ L(G).

Ejercicios sugeridos (Gramáticas y Derivaciones)

1. Dada la gramática G_1 = (Σ = {0,1}, N = {S, A, B}, S, P) con producciones:
   S := 0B | 0A1
   A := 0B | 0
   B := 1
   a) Enumerar todas las cadenas terminales que se pueden derivar desde S (lenguaje generado). **b)** Indicar paso a paso al menos una derivación completa para cada cadena obtenida. **c)** Clasificar el lenguaje generado (ver siguiente sección de tipos de lenguaje).
2. Considere la gramática G_2 = (Σ = {a,b,c}, N = {P,Q,R}, P, P_2) con producciones:
   P := aQ
   Q := ab | caR
   R := c | ac
   a) Encontrar dos derivaciones distintas (secuencia de reemplazos) que generen la cadena "abc" desde P (si existen). **b)** ¿La gramática puede generar la cadena "aac"? Justificar mostrando una derivación o argumentando por qué no es posible.
3. Dada la gramática G_3 = (Σ={a,b}, N={S,A,B}, S, P) con:
   S := aAb | bBa
   aAb := aBb
   aBb := abb | acb
   bBa := bca
   ¿Cuáles son las palabras terminales derivables desde S? (Sugerencia: primero ver qué puede generar A y B). Muestre las derivaciones correspondientes.
4. Sea G = ({a,b}, {S,A}, S, P) con producciones:
   S := aS | aA
   A := ab
   a) Listar algunas cadenas cortas en L(G). **b)** Si cambiamos P por:

P': S := aS | aab,
¿sigue generando el mismo lenguaje? (Comparar L(G) con L(G') con algunos ejemplos).

Clasificación de Gramáticas y Lenguajes (Jerarquía de Chomsky)

Noam Chomsky en 1956 propuso una jerarquía para clasificar las gramáticas formales (y sus lenguajes generados) en **cuatro tipos**: tipo 0, 1, 2 y 3, según restricciones en la forma de las producciones. A medida que agregamos restricciones al formato de las reglas, las gramáticas se vuelven menos potentes (generan menos lenguajes), obteniendo una clasificación **inclusiva**: los lenguajes de tipo 3 ⊂ tipo 2 ⊂ tipo 1 ⊂ tipo 0.

Tipo 0: Gramáticas irrestrictas o de frase estructurada. Son las más generales (lenguajes *recursivamente enumerables*). Sus producciones no tienen restricción aparte de la condición básica de toda gramática: el lado izquierdo debe contener al menos un símbolo no terminal. Es decir, reglas de la forma:
α := γ, donde α ∈ (Σ_T ∪ Σ_N) *Σ_N (Σ_T ∪ Σ_N)* y γ ∈ (Σ_T ∪ Σ_N).
*En otras palabras, α puede ser cualquier cadena con al menos un no terminal, y y cualquier cadena de terminales y/o no terminales (incluso y podría ser λ en teoría, si α no es el axioma; la única restricción original de gramática formal ya contemplaba no terminal en el lado izquierdo). Los lenguajes tipo 0 son exactamente aquellos que pueden ser reconocidos por una Máquina de Turing***** (esto excede el alcance de la materia, pero establece su potencia computacional máxima).

Tipo 1: Gramáticas dependientes del contexto (sensibles al contexto). Permiten reemplazos donde el no terminal a reemplazar puede depender de sus símbolos vecinos (contexto). Las producciones tienen la forma:
α A β := α γ β, con A ∈ Σ_N, α, β ∈ (Σ_T ∪ Σ_N) *y γ ∈ (Σ_T ∪ Σ_N)^+ (γ no vacío).*
*Esto significa: el no terminal A puede ser reemplazado por γ solo cuando está precedido por α y seguido por β en la cadena actual (ese es el contexto). Además, se requiere que $|\gamma| \geq 1$ y que la longitud del lado izquierdo no exceda la del lado derecho (para garantizar que no haya contracción de longitud, excepto en una regla especial para generar λ). En gramáticas de tipo 1, no se permiten reglas que reduzcan la longitud****** de la cadena derivada, a excepción de la posibilidad de generar λ desde el axioma si el lenguaje incluye la palabra vacía. Es decir, se permite S := λ como excepción (si se necesita λ en el lenguaje), pero ninguna otra producción puede ser "compresora".

Ejemplo de producción contexto-dependiente: `X Y := X Z Y`. Aquí A = Y, α = X, β = (vacío); indica que Y puede reemplazarse por ZY solo cuando está precedida por X en la cadena.

Los lenguajes tipo 1 son reconocidos por **automátas lineales acotados** (un tipo de máquina de Turing restringida en memoria).

Tipo 2: Gramáticas independientes del contexto (o de contexto libre). Son quizás las más estudiadas por su aplicación en la sintaxis de lenguajes de programación. En estas gramáticas, las producciones tienen la forma más sencilla:
A := γ, con A ∈ Σ_N y γ ∈ (Σ_T ∪ Σ_N).
*Es decir, el lado izquierdo es un único no terminal, y el lado derecho puede ser cualquier cadena de símbolos (excepto que, por convenio de no contracción, exigimos que si A no es el axioma entonces γ no sea vacío, salvo el caso permitido de S := λ similar al tipo 1). Esto implica que un no terminal A se puede reescribir como γ independientemente del contexto****** en el que A aparezca; de ahí "libre de contexto". Cada producción simplemente define cómo un símbolo no terminal se expande por una secuencia de símbolos. (Estas reglas también son no compresoras salvo S->λ si necesario).

Todas las gramáticas tipo 2 (GIC) generan **lenguajes independientes del contexto (LIC)**. Son exactamente los lenguajes reconocidos por un **autómata de pila** (pushdown automaton). La mayoría de los lenguajes de programación tienen su sintaxis definida por gramáticas libres de contexto.

Tipo 3: Gramáticas regulares (o lineales). Son las más restringidas. En las producciones de una gramática regular, el lado izquierdo es un único no terminal (igual que en tipo 2), pero el lado derecho está limitado a como mucho **un símbolo terminal seguido de un no terminal, o un solo terminal**. En otras palabras, las reglas válidas (además de S->λ opcional) deben verse así:
- (Forma **regular por la derecha**): A := aB **o** A := a, donde A, B ∈ Σ_N y a ∈ Σ_T.
- (Forma **regular por la izquierda**): A := B a **o** A := a.

Esto define dos subtipos equivalentes en potencia: *gramáticas regular derechas* vs *izquierdas*. No se permite mezclar ambas formas en la misma gramática (si mezclas reglas de una y otra forma, en realidad resulta ser tipo 2). Pero por cada gramática regular derecha existe una gramática regular izquierda equivalente en lenguaje generado (de hecho el lenguaje regular es el mismo; en capítulo de autómatas se verá correspondencia con autómatas finitos deterministas y no deterministas, etc.).

Los **lenguajes regulares** generados son aquellos reconocidos por **autómatas finitos** (deterministas o no deterministas). Ejemplos de lenguajes regulares: el conjunto de identificadores válidos de un lenguaje (que suelen seguir un patrón simple), todos los números enteros literales, todos los comentarios de un lenguaje de programación, etc. Muchas construcciones léxicas se describen con lenguajes regulares.

Resumen de jerarquía:
- Tipo 3 (regulares) ⊂ Tipo 2 (libres de contexto) ⊂ Tipo 1 (dependientes de contexto) ⊂ Tipo 0 (irrestrictos).
Esta inclusión se puede visualizar en un diagrama de Venn: cada categoría contiene a la siguiente más restringida. Por ejemplo, todo lenguaje regular es también libre de contexto, todo libre de contexto es sensible al contexto, etc., pero no a la inversa.

Determinando el tipo de una gramática: Se puede identificar el tipo de una gramática examinando sus producciones:
- Si **alguna** producción es de la forma α → β donde |α| > |β| (exceptuando S->λ), entonces la gramática es al menos de **tipo 0** (irrestricta), porque ha violado la restricción de no contracción de tipo 1.
- Si **todas** las producciones cumplen que |α| ≤ |β| y ninguna regla (salvo S->λ) reduce longitud, y además algunas requieren contexto (α ≠ A solo), entonces es **tipo 1**. Ej: regla "ABj := AZBj" es context-sensitive (B reemplazado por ZB con A...j como contexto).
- Si todas las producciones son de la forma A → γ (A único no terminal a la izquierda) entonces es al menos **tipo 2** (puede ser tipo 2 o 3).
- Si además cada γ es como mucho un terminal seguido de un no terminal, o un terminal solo (o λ solo permitido para S), entonces es **tipo 3**. Por ejemplo, "A := 0B | 1" es regular (derecha) porque cada alternativa es un terminal opcionalmente seguido de NT. En cambio, "A := 0B0" no es regular (tiene dos terminales con un NT en medio, eso es fuera del formato).

Ejemplo (Clasificación): Dada la producción `TREN := locomotora | TREN vagón`: aquí "locomotora" y "vagón" se consideran símbolos terminales individuales (palabras completas pero indivisibles). `TREN` es un no terminal. Esta gramática es regular derecha, porque tiene `TREN -> "locomotora"` o `"TREN" "vagón"`. Formato: A->a | A a (que se puede transformar a cumplir formato estricto). Genera cadenas como "locomotora", "locomotoravagón", "locomotoravagónvagón", etc. Representa la idea de "un tren es una locomotora seguida de cero o más vagones".

Ejercicios sugeridos (Jerarquía de Chomsky)

1. Identificar el tipo (0,1,2 o 3) de cada una de las siguientes gramáticas, justificando brevemente:
   a. P = { S := aA b, A := aA b | c }.
   b. P = { D := cE | d, E := cd }.
   c. P = { S := aB | aA c, A := aB c | aB b c, B := b }.
   d. P = { S := xX, X := xY, Y := yY | y }.
   (Sugerencia: verificar formato de reglas y presencia de contexto o longitud variable).
2. Para cada caso siguiente, indique si P1 y P2 definen **gramáticas equivalentes** (mismo lenguaje) y qué tipo serían:
   a. Σ={a,b}, N={S,A}. P1: { S := aS | aA, A := a b }. P2: { S := aS | aab }.
   b. Σ={a,b}, N={S}. P1: { S := aS | b }. P2: { S := a S b | b }.
   (En cada caso razonar si ambas generan las mismas cadenas y cuál es el nivel mínimo de la jerarquía que las puede describir.)

Lenguajes Regulares y Expresiones Regulares

Los **lenguajes regulares** son aquellos generados por gramáticas tipo 3. Debido a su amplia utilidad, existen varias formas equivalentes de representarlos: gramáticas regulares, **expresiones regulares**, autómatas finitos, etc. Todas estas representaciones tienen igual poder descriptivo para lenguajes regulares.

Definición constructiva de lenguaje regular: Podemos definir recursivamente la clase de lenguajes regulares así:
a) **Caso base:** Cualquier lenguaje **finito** (con número natural de cadenas) es regular. En particular, ∅ (el lenguaje vacío, sin cadenas) es regular, {λ} es regular, y cualquier conjunto finito de palabras específicas es regular. Incluso cada palabra individual puede verse como un lenguaje finito unitario.
b) **Cierre por unión:** Si L_1 y L_2 son lenguajes regulares, entonces $L_1 \cup L_2$ también es regular.
c) **Cierre por concatenación:** Si L_1 y L_2 son regulares, entonces $L_1 \cdot L_2$ es regular.
d) **Cierre por Kleene:** Si L_1 es regular, entonces L_1 *(estrella de Kleene) es regular.*
*e) Minimalidad:** No hay otros lenguajes regulares aparte de los obtenidos aplicando finitas veces las operaciones a), b), c) y d). Esta definición recursiva garantiza que partiendo de casos básicos simples (lenguajes finitos pequeños), la clase de lenguajes regulares está cerrada bajo unión, concatenación y estrella.

En particular, los lenguajes regulares son cerrados bajo muchas operaciones: además de las mencionadas, también bajo intersección, complemento, diferencia, etc. Esto los hace muy manejables teóricamente.

Expresiones regulares (ER): Son una notación algebraica concisa para describir lenguajes regulares. Se definen recursivamente:
- Base: 1. ∅ es una ER que denota al lenguaje ∅.
2. λ es una ER que denota al lenguaje {λ}.
3. Si a ∈ Σ es un símbolo terminal, entonces "a" (literal) es una ER que denota {a}.
- Recursión: Si E y F son ER que denotan lenguajes L(E) y L(F) respectivamente, entonces:
4. E + F es una ER que denota L(E) ∪ L(F) (unión). En muchas notaciones se usa `|` en lugar de `+` para la unión.
5. E F (o simplemente EF) es una ER que denota L(E) · L(F) (concatenación).
6. E *es una ER que denota [L(E)]* (estrella de Kleene del lenguaje de E).
7. (E) es una ER que denota L(E) (los paréntesis son para agrupar y no cambian el lenguaje, solo la

precedencia de operadores).
- Nada más es una ER excepto lo construido con las reglas anteriores.

Por ejemplo, sobre Σ = {0,1}, una ER `0 + 1` denota { "0","1" } (cualquier dígito binario). Una ER `0(0+1)*1` denota el lenguaje de las cadenas que comienzan con 0, terminan con 1, y pueden tener cualquier secuencia (incluyendo vacía) de bits en el medio (cadenas binarias que empiezan en 0 y terminan en 1).

Ejemplo: Sea Σ = {0,1}. Consideremos la ER `(01 + 1)*`. Analicemos su lenguaje:
- Subexpresión `01 + 1` denota { "01", "1" }.
- Colocando *: `(01 + 1)*` denota la estrella de ese conjunto, o sea todas las cadenas formadas por concatenar cero o más ocurrencias de "01" o "1". Esto incluye: λ (tomando cero ocurrencias), "1" (una ocurrencia de "1"), "01" (una ocurrencia de "01"), "11" ("1" concatenado con "1"), "011" ("01" + "1"), "101" ("1" + "01"), "0101" ("01"+"01"), "111" ("1"+"1"+"1"), "01101" ("01"+"1"+"01"), etc. En general, describe el lenguaje de cadenas binarias donde no aparecen dos '0' consecutivos (porque cualquier '0' en la cadena va seguido inmediatamente de '1' debido al patrón "01").

Las ER son ampliamente utilizadas en herramientas de búsqueda (grep, sed, etc.), en definiciones de tokens léxicos en compiladores, y en muchas aplicaciones de procesamiento de texto.

Ejercicios sugeridos (Lenguajes Regulares y ER)

1. Construir expresiones regulares para los siguientes lenguajes sobre Σ = {0,1}:
   a) El lenguaje de todas las cadenas que comienzan con "01" y terminan con "10".
   b) El lenguaje de todas las cadenas de longitud par.
   c) El lenguaje {λ, "0", "00", "000", ...} (cadenas de solo '0' de cualquier longitud, incluyendo vacía).
   d) El lenguaje de todas las cadenas que contienen al menos dos '1'.
   (Verificar cada ER proponiendo algunas derivaciones de cadenas y comprobando si cumplen la descripción.)
2. Dada la expresión regular E = `(a + bb)*(λ + b)`, describir en palabras el lenguaje L(E) que denota. Luego enumerar las palabras de L(E) de longitud ≤ 3.
3. Proponer una ER que denota el conjunto de identificadores válidos de un lenguaje imaginario donde las reglas son: comienza con letra, seguido por cualquier cantidad de letras o dígitos. (Usar un alfabeto Σ = {a,...,z, 0,...,9} apropiado).
4. Sean los lenguajes regulares L_1 = {a, ab} y L_2 = {b, λ}. Calcular utilizando cerraduras: **a)** $L_1 \cup L_2$, **b)** $L_1 \cdot L_2$, **c)** $L_2 \cdot L_1$, **d)** L_1, *e)* L_2. Dar expresiones regulares equivalentes a cada resultado.

Simplificación de Gramáticas: Gramática "limpia"

Al diseñar una gramática es deseable eliminar cualquier símbolo o producción que no contribuya realmente a generar el lenguaje (por claridad y eficiencia). Llamamos **gramática limpia** a una gramática libre de "impurezas", es decir, sin reglas inútiles ni símbolos innecesarios. Una gramática independiente del contexto G = (Σ_T, Σ_N, S, P) está limpia si cumple:
- No tiene **reglas innecesarias** del tipo A := A (donde un no terminal produce a sí mismo). Es claro que A := A no cambia ninguna cadena durante una derivación; aplicarla no hace avanzar, solo desperdicia pasos. Podemos eliminar cualquier producción trivial A->A sin alterar el lenguaje generado.
- No tiene **símbolos inaccesibles**: un símbolo X (terminal o no terminal) es inaccesible si no aparece en *ninguna* forma sentencial derivada desde S. En términos prácticos, X es inaccesible si no hay forma de alcanzarlo partiendo del axioma usando las producciones. Si un símbolo nunca aparece en ninguna derivación posible, se puede eliminar de la gramática (quitarlo del conjunto correspondiente y eliminar

producciones que lo mencionen) sin afectar L(G).
- No tiene **símbolos superfluos**: un símbolo no terminal X es superfluo si *no puede derivar* ninguna cadena de solo terminales (ni siquiera λ). Es decir, aunque X sea accesible (tal vez aparezca en derivaciones), una vez que aparece no hay forma de terminar derivando completamente a terminales desde X. Esto significa que cualquier derivación que incluya a X se atascará, generando un camino sin salida. Tales símbolos y sus reglas también deben eliminarse. Formalmente, X es superfluo si **no** existe alguna derivación X → *α con α ∈ Σ_T^* (α terminal puro). Si X no genera terminales, quitarlo (y reglas asociadas) no cambia el lenguaje final posible.

El proceso típico de limpieza: primero eliminar reglas A:=A triviales; luego detectar símbolos accesibles (a partir de S marcar todos los que se pueden alcanzar) y quitar los no alcanzables; luego detectar símbolos generativos (marcar no terminales que pueden producir algún terminal, trabajando inversamente desde producciones obvias hacia atrás) y quitar los que no producen terminales. El resultado es una gramática equivalente (mismo lenguaje) pero más simple.

Ejemplo (Gramática limpia): Considere la gramática:
G_1 = ({a,b,c,d}, {S,A,X}, S, { S := aAb, A := aAb | c }).
Observamos que los símbolos X (no terminal) y 'd' (terminal) nunca aparecen en las producciones del conjunto P; por lo tanto son inaccesibles desde S. Podemos quitarlos: obtenemos G_2 = ({a,b,c}, {S,A}, S, { S := aAb, A := aAb | c }), que genera el mismo lenguaje. También si hubiera habido una regla A := A, la habríamos eliminado por inútil. Al final, G_2 es más limpia. (En G_1, 'X' y 'd' eran símbolos totalmente irrelevantes).

Otro ejemplo: G_3 = ({a,b}, {S,A,B}, S, { S := aAb, A := aAb | ab | aB, B := aBb }). Aquí la producción B := aBb indica que B es recursivo y nunca produce solo terminales (siempre deja al menos un B); B tampoco tiene otra alternativa que no contenga B. Esto sugiere que B es superfluo (no genera terminales). Efectivamente, desde B solo se derivan formas con B de nuevo (aBb -> aaBbb -> aaaBbbb ... nunca sale de B). Entonces B no puede derivar una palabra terminal completa. Lo eliminamos junto con sus reglas y cualquier mención en otras reglas. La gramática resultante sería: S := aAb, A := aAb | ab. Esa gramática genera el mismo lenguaje de cadenas de la forma a^n b^n c?... (bueno, habría que analizar qué lenguaje es, pero lo importante es que B no aportaba cadenas nuevas).

Después de limpiar: decimos que la gramática está **limpia** si no quedan reglas innecesarias ni símbolos inaccesibles ni superfluos.

Ejercicios sugeridos (Simplificación – Gramática limpia)

1. Dada la gramática G = ({a,b,c,d}, {A,B,C,D,E}, A, P) con producciones:
   A := D a | E b a | λ
   B := b C d | d
   C := C (C := C es una regla innecesaria)
   D := b A
   E := a E | c E
   a) Identificar y eliminar las reglas innecesarias. **b)** Identificar símbolos inaccesibles y superfluos, eliminándolos junto con sus producciones. **c)** Escribir la gramática equivalente resultante, limpia.
2. Simplificar la gramática:
   G = ({0,1}, {S,A,B}, S, { S := 0A1 | 0, A := 0A1 | 0B | 0, B := 0B | 0 }).
   a) ¿Hay símbolos inaccesibles? **b)** ¿Algún símbolo superfluo (que no genere terminales)? **c)** Dar la gramática limpia equivalente.
3. Sea G = ({x,y,z}, {S,M,N}, S, P) con producciones:
   S := z M N z

M := λ | a M a
N := λ | b N b | z
a) ¿La gramática dada está limpia? Justificar revisando reglas innecesarias, inaccesibles, superfluas. **b)** En caso de encontrar "impurezas", eliminarlas y escribir la gramática limpia resultante.

Gramática bien formada (Eliminación de λ-producciones y unit-productions)

Una vez la gramática está limpia, podemos hacer transformaciones adicionales. Decimos que una gramática independiente del contexto está **bien formada** si, además de estar limpia, **no tiene reglas no generativas ni de renombramiento**. Aquí:
- **Regla no generativa** se refiere a una producción A := λ donde A **no** es el axioma. Esta regla "genera" la cadena vacía a partir de A, pero si A no es S, eso implica que podría usarse para eliminar A en contextos internos, acortando la longitud (es una regla compresora). Para gramáticas bien formadas, solo se permite la regla λ para el axioma si el lenguaje incluye la palabra vacía; ninguna otra variable puede anularse directamente.
- **Regla de renombramiento (redenominación)** es una producción del tipo A := B donde A y B son no terminales. Esto hace que A actué prácticamente como sinónimo de B, pudiendo derivarse lo mismo. Se busca eliminar esas cadenas de reemplazos inútiles directos entre no terminales.

Eliminación de reglas λ (no generativas): Si el lenguaje generado L(G) no necesita a λ (no la incluye), entonces podemos simplemente quitar cualquier producción A := λ; pero usualmente si existe tal producción es porque λ sí pertenece al lenguaje (por ejemplo, a través de A). En contextos libres, para eliminar A := λ (A no axioma) sin cambiar L(G), debemos *propagar* sus efectos: por cada producción X := α A β en P (donde A aparece en el lado derecho), agregamos una nueva producción X := α β (reemplazamos esa aparición de A por cadena vacía) para contemplar la posibilidad de que A derive λ en esa posición. Hacemos esto para todas las ocurrencias de A en todas las producciones. Tras agregar las reglas derivadas, podemos quitar A := λ. Este proceso elimina la necesidad explícita de la regla λ de A, ya que las derivaciones donde A desaparecía quedan cubiertas por las nuevas producciones. Esto se hace para cada variable con regla λ que no sea el axioma.

Ejemplo: Gramática G = (Σ={a,b}, N={S,A}, S, P) con P: { S := a A b, A := a A b | λ }. Aquí A tiene una producción λ. El lenguaje incluye palabras de la forma a^n b^n (n>0). Para eliminar A := λ:
- Miramos dónde aparece A en otras reglas: S tiene "a A b". Al poder A -> λ, S podría derivar "a" + "" + "b" = "ab". Entonces añadimos una producción S := a b directamente.
- También A aparece en sí misma: "A := a A b". Si A puede ser vacío, esta regla sugiere que A podría generar "ab" (un a y un b con A central anulándose). Añadimos A := a b.
Luego quitamos A := λ. La gramática resultante: S := a A b | a b, A := a A b | a b. Esta gramática ya no tiene λ-producciones (salvo que S podría derivar "ab", pero λ ya no está en P).

Eliminación de reglas de renombramiento (unitarias): Una regla A := B (ambos no terminales) se puede quitar *transfiriendo* todas las producciones de B a A. Es decir, por cada producción B := γ, agregamos A := γ. Finalmente removemos A := B. Si hay ciclos de renombramiento (A->B, B->A), también se eliminan y, en la práctica, ambas variables terminan teniendo las mismas expansiones posibles (se fusiona su comportamiento). Este proceso convierte esos sinónimos en un único conjunto de reglas directamente.

Las reglas unitarias son problemáticas porque encadenar muchos no terminales sin producir terminales puede llevar a complicaciones (incluso bucles A->B, B->A). Eliminarlas simplifica la gramática sin cambiar el lenguaje.

Ejemplo: Supongamos P tiene S := A (S renombra a A), y además A tenía producciones A := 0A S | 1. Antes de quitar S := A, agregamos a P las reglas S := 0A S y S := 1 (todo lo que produce A). Luego eliminamos S := A. Ahora S puede generar lo que antes generaba A, sin paso intermedio. Repetimos para cualquier otro caso de redenominación.

Tras eliminar reglas λ no autorizadas y redenominaciones, obtenemos la gramática **bien formada**. Esto es a menudo un paso previo para llevar la gramática a formas normales.

Ejercicios sugeridos (Gramática bien formada)

1. Dada la gramática:
   G = ({0,1}, {S,A,B}, S, { S := 0A0 | 0B, A := 0A0 | 1B | 1, B := 1B | 1 | A B | λ }).
   a) Identificar las reglas no generativas (λ-producciones) y de renombramiento en P. **b)** Eliminar primero todas las λ-producciones excepto quizá S->λ si existiera (¿el lenguaje genera λ? analizar). **c)** Luego eliminar las reglas unitarias. **d)** Presentar la gramática bien formada equivalente resultante.
2. Para la gramática del ejercicio anterior, demostrar con una derivación un ejemplo de cadena que antes requería la regla λ o la regla de renombramiento y ahora se deriva con las nuevas producciones agregadas.
3. Considere la gramática G = ({a,b}, {S,B}, S, { S := aS | aB, B := b }).
   a) ¿Está limpia? (Símbolos innecesarios o inaccesibles?). **b)** ¿Tiene reglas no generativas o de renombramiento? (No directamente). **c)** Si añadimos una producción B := S, ¿qué cambia? Eliminar esa redenominación adicional siguiendo el procedimiento.

Derivaciones por la Izquierda, por la Derecha y Árboles de Análisis

En una gramática libre de contexto, en cada paso de derivación puede haber varios no terminales en la cadena que podrían expandirse. **Derivación por la izquierda** significa que en cada paso siempre se reemplaza el no terminal más a la izquierda primero. **Derivación por la derecha** significa reemplazar primero el no terminal más a la derecha. Estas son dos estrategias de derivación posibles que, si la gramática no es ambigua (ver más adelante), conducen al mismo resultado final aunque quizás por caminos diferentes.

Ejemplo: Dada una gramática G(29):

```
S := P Q | 0 S 1
P := 0 Q | 1
Q := 1 P | 0
```

Generemos la cadena "0101":
- *Derivación por la derecha:* S → 0 S 1 → 0 P Q 1 → 0 P 0 1 → 0 1 0 1.
- *Derivación por la izquierda:* S → 0 S 1 → 0 P Q 1 → 0 1 Q 1 → 0 1 0 1.
En ambas se llegó a "0101", pero el orden de aplicación de reglas difiere. Sin embargo, si dibujamos el **árbol de derivación** (o **árbol sintáctico**) para "0101", veremos que es el mismo independientemente

de la estrategia izquierda o derecha: la estructura jerárquica de cómo S se descompone en terminales es única en este caso.

Árbol de derivación: Es una representación gráfica de una derivación completa. El árbol siempre: - Tiene la raíz etiquetada con el axioma S. - Los nodos internos están etiquetados con no terminales. - Las hojas están etiquetadas con símbolos terminales (o λ si se generó la vacía). Para construir el árbol, partimos de S en la raíz. Si S se expande con una producción $S := X_1\ X_2\ \ldots\ X_n$, entonces creamos n hijos para la raíz, de izquierda a derecha, etiquetados con X_1, X_2, ..., X_n. Luego, tomamos cada hijo que sea no terminal y lo expandimos a su vez usando la producción correspondiente, agregando sus hijos, y así sucesivamente. Cuando un símbolo es terminal, es hoja y no se expande más.

La propiedad fundamental: si lees de izquierda a derecha las hojas del árbol (recorriendo las hojas en orden), obtienes exactamente la cadena terminal derivada. Por ejemplo, en el árbol para "el gato corre la rata" derivado antes, las hojas en orden serían "el", "gato", "corre", "la", "rata".

El árbol captura la **estructura** de la derivación sin importar el orden en que se aplicaron las producciones. Distintas derivaciones (izquierda vs derecha) de la **misma cadena** conducen al **mismo árbol**, siempre que la gramática no permita ambigüedad. Por eso, para propósitos de análisis sintáctico, es conveniente pensar en la construcción del árbol más que en la secuencia exacta de reemplazos.

Subárboles y subcadenas: Un subárbol del árbol de derivación corresponde a la derivación de una subcadena particular de la cadena final. En el ejemplo anterior, la subcadena "10" de "0101" fue derivada por un subárbol específico. Esto no siempre es tan simple, pero si la subcadena corresponde a un componente sintáctico completo (un no terminal de alto nivel), aparecerá como subárbol.

Importancia de derivación izquierda/derecha: En la práctica de construcción de analizadores sintácticos, la derivación por la izquierda está asociada con los **parsers descendentes** (top-down) y la derivación por la derecha con **parsers ascendentes** (bottom-up). En ambos casos, el árbol sintáctico final debe ser el mismo; difiere el orden en que se lo construye.

Ejercicios sugeridos (Tipos de derivación y Árboles)

1. Para la gramática G(29) del ejemplo anterior, reconstruir el árbol de derivación de la cadena "0101" tanto para la derivación por la izquierda como por la derecha y verificar que son iguales. Dibujar el árbol mostrando los nodos S, P, Q y terminales 0,1.
2. Considerar la gramática:

   ```
   <decl> := <tipo>_<ident>;
   <tipo> := entero | real;
   <ident> := a | b | c | a<dig> | b<dig> | c<dig>;
   <dig>  := 0|1|...|9;
   ```

   a) Dar dos derivaciones distintas (una por la izquierda, otra por la derecha) para la cadena `entero_a0;`. **b)** Construir el árbol de análisis sintáctico de esa declaración e indicar qué parte del árbol corresponde al `<tipo>` y cuál al `<ident>`.
3. Sea la gramática G:
   S := S + S | S * S | a
   (Aquí '+' y '*' *son símbolos terminales literales, 'a' también terminal; S es tanto axioma como no terminal usado recursivamente).

a) Dar una derivación por la izquierda de la cadena `a+a*a`*. b) Dar una derivación por la derecha de la misma cadena. c)** Construir el árbol sintáctico resultante e identificar en el árbol las operaciones de suma y multiplicación. (Nota: esta gramática es ambigua, como se verá después, pero igualmente se pueden hacer derivaciones distintas y observar el árbol).

Ambigüedad en Gramáticas

Una gramática libre de contexto se dice **ambigua** si existe al menos una cadena terminal que puede ser derivada de dos o más **formas diferentes** que den lugar a **árboles de derivación distintos**. En otras palabras, hay al menos una sentencia que tiene dos (o más) árboles sintácticos válidos. Si una cadena simplemente tiene dos derivaciones diferentes pero que convergen en el mismo árbol (por ejemplo una derivación izquierda vs derecha), eso **no** cuenta como ambigüedad, pues el árbol único indica una única estructura. La ambigüedad requiere estructuras sintácticas diferentes.

Consecuencias: En lenguajes de programación, la ambigüedad es indeseable porque implicaría que una misma secuencia de tokens podría interpretarse de dos maneras distintas (p. ej., sintaxis de if-else anidados sin llaves puede ser ambigua). Siempre se busca una gramática no ambigua para definir un lenguaje de programación. Hay lenguajes (generalmente artificiales) que son **inherentemente ambiguos**, es decir, *cualquier* gramática que los genere será ambigua. Esos lenguajes no son usados para propósitos donde la interpretación inequívoca es necesaria.

Ejemplo clásico de ambigüedad: La gramática S := S + S | S * S | a (similar a la del ejercicio anterior) es ambigua con respecto a la precedencia de operadores. La cadena "a + a * a" puede parsearse de dos formas:
- Interpretando + con menor precedencia (intención usual): árbol donde la multiplicación se agrupa antes que la suma (S -> S + S, el segundo S se expande a S * S).
- Interpretando + con mayor precedencia: árbol donde la suma se agrupa primero (S -> S * S, donde el primer S se expande a S + S).
Ambos árboles son distintos, reflejan distintas interpretaciones (a+(a*a) vs (a+a)*a). Así la gramática es ambigua.

Detección de ambigüedad: No hay un algoritmo general eficiente para determinar si una gramática arbitraria es ambigua (el problema es indecidible en general). En la práctica, se identifica encontrando una cadena específica con dos derivaciones distintas. Para **probar** que una gramática es ambigua basta un contraejemplo de cadena con dos árboles. Para **probar** que es no ambigua, en cambio, se requiere justificar que ninguna cadena tiene dos árboles, lo cual suele hacerse dando propiedades de la gramática o convirtiéndola en un algoritmo determinista de parsing.

Si una gramática es ambigua, a veces es posible crear una **gramática equivalente no ambigua** para el mismo lenguaje. Por ejemplo, para eliminar la ambigüedad de la gramática de expresiones aritméticas, se introducen no terminales separados para términos y factores, imponiendo la jerarquía de operaciones (esto se ve en cursos de compiladores). Sin embargo, algunos lenguajes (como ciertas uniones de lenguajes contextuales) son inherentemente ambiguos y no admiten gramática libre de contexto no ambigua.

Nota: Ambigüedad es una propiedad de la gramática, no del lenguaje, aunque decimos "lenguaje inherentemente ambiguo" cuando *todas* las gramáticas posibles que lo describen son ambiguas.

Ejercicios sugeridos (Ambigüedad)

1. Considere la gramática:
   S := A B | A C
   A := a A | a
   B := b B | b
   C := b C | c
   a) Derivar la cadena "aa b" (con un espacio imaginario para separar lo derivado por B o C, es decir "aab" con una división) de dos maneras: una donde S -> A B y otra donde S -> A C, mostrando las producciones usadas. **b)** Dibujar los árboles de derivación correspondientes a cada derivación. **c)** ¿Es la cadena "aab" ambigua en esta gramática? ¿Por qué?
2. Sea la gramática G:
   S := S + S | S * S | a | b
   a) Encontrar al menos dos árboles de derivación distintos para la cadena "a + b * a". (Pista: diferente agrupación de + y *). b) En cada árbol, identificar qué operación se realiza primero. c)* Concluir por qué la gramática es ambigua e intentar proponer reglas adicionales o modificadas que establezcan una precedencia (por ejemplo, introducir Suma y Producto como no terminales distintos) para eliminar la ambigüedad.
3. Dadas las gramáticas a continuación, determinar si son ambiguas. Si lo son, proporcionar una cadena ejemplo con dos árboles; si no lo son, argumentar la razón:
   a. S := 0 S 0 | 0 0 (lenguaje de cadenas con dos 0 exteriores y cualquier cantidad de pares "00" dentro).
   b. S := S S | 0 | 1.
   c. S := A A, A := 0 | 0 A.
   (Analizar cada caso con algunas cadenas pequeñas.)

Normalización de Gramáticas (Formas Normales)

Para ciertas aplicaciones teóricas y prácticas, es útil restringir aún más la forma de las producciones de una gramática libre de contexto, sin perder generalidad. Existen transformaciones algoritmicas que llevan cualquier gramática independiente del contexto **bien formada** a una forma estándar. Las dos formas normales más conocidas son:

Forma Normal de Chomsky (FNC): Una gramática está en FNC si **todas** sus producciones son de uno de los siguientes tipos:
- A -> B C (un no terminal produce exactamente dos no terminales),
- A -> a (un no terminal produce exactamente un terminal),
- Opcionalmente, S -> λ está permitido *solo* si λ ∈ L(G) (y S no aparece en el lado derecho de ninguna producción).
No se permiten otras formas. En particular, no se permiten producciones con 3 o más símbolos en el lado derecho, ni mezclas de terminales y no terminales en el lado derecho (salvo el caso trivial de un solo terminal), ni cadenas vacías excepto desde el axioma.

En FNC, cada árbol de derivación es un **árbol binario** (cada producción A->BC genera un nodo con dos hijos). Esto simplifica demostraciones y algoritmos (por ejemplo, algoritmos de parsing como CYK operan sobre gramática en FNC).

Conversión a FNC: Dada una GIC bien formada, se puede convertir a FNC mediante:
1. **Eliminar símbolos terminales "mezclados":** Si hay alguna producción del tipo A -> u α v (cadena con longitud ≥2 que incluye uno o más terminales en medio), se introducen nuevos no terminales para cada terminal problemático. Por ejemplo, si tenemos A -> X a Y (donde a es terminal junto a otros

símbolos), creamos un no terminal say T_a con producción T_a -> a, y reemplazamos la 'a' en la regla original por T_a. Así todas las producciones tendrán ahora o solo terminal, o solo no terminales (salvo casos ya manejados). Hacemos esto para cada terminal que aparezca junto con otros símbolos en algún lado derecho.
2. **Eliminar producciones unitarias restantes:** (En la gramática bien formada ya no debería haber A->B, así que esto ya estaría resuelto).
3. **Descomponer producciones largas:** Si hay alguna producción con 3 o más símbolos en el lado derecho, por ejemplo A -> B C D, introducimos no terminales intermediarios para romperla en binarias. Una forma: reemplazar B C D por B X, con X un nuevo no terminal, y añadir X -> C D como producción. Si todavía D se expande en más de 2, repetimos creando más nodos intermedios, hasta que todas las reglas sean binarias o unitarias.
4. **Ajustar S -> λ si corresponde:** Si λ estaba en el lenguaje, nos aseguramos de conservar S -> λ (en FNC pura no se permite λ salvo desde S). Si S -> λ no estaba, no introducimos λ.

El resultado es una gramática equitativa en lenguaje, que cumple la forma normal de Chomsky. Es posible que se agreguen no terminales nuevos (variables auxiliares) pero eso no cambia el lenguaje generado.

Ejemplo: Dada G = ({a,b,c}, {A,B,C}, A, P) con P:
A := C B c | b B | λ
B := B C | b
C := c
Primero aseguramos que esté bien formada: vemos que A->λ es permitida solo si A es axioma (lo es en este caso, A = S digamos); B->B C es recursión izquierda (eso no impide FNC pero es algo a notar), no hay renombramiento directo. Vamos paso a paso:
- Terminales mezclados: A->C B c tiene un terminal 'c' junto a B. Creamos X_c := c y reemplazamos: A := C B X_c. También B->b (está bien, un terminal solo), B->B C (no hay terminal mezclado), C->c (terminal solo, ok).
- Ahora producción A->C B X_c (3 símbolos no terminales en lado derecho). Descomponemos: introducimos Y -> B X_c. Entonces A := C Y, y Y := B X_c. Ahora todas las reglas son de 2 símbolos o 1:
A -> C Y
Y -> B X_c
B -> B C (aquí hay un problema, B->B C sigue teniendo 2 símbolos pero es **recursiva**; recursividad está permitida, no rompe FNC, pero *¡cuidado!* B->B C es unitario? No, es 2 no terminales así que encaja en forma binaria, aunque B aparece en ambos lados pero eso es permitible en FNC)
B -> b
C -> c (original)
X_c -> c (nuevo)
- Vemos que B->B C no es una unidad (porque lado derecho tiene 2 símbolos) así que formalmente cumple FNC: (no dice que B y C no puedan ser iguales a la izquierda, solo restringe la forma, aunque esta producción es problemática por recursión inmediata izquierda, que la FNC no elimina, pero para parsing CYK igualmente formaría bucles). En fin, la gramática resultante está en FNC: cada regla es A->NT NT, Y->NT NT, B->NT NT (esa recursiva), B->t, C->t, X_c->t, S->λ (si necesitamos λ).

Forma Normal de Greibach (FNG): Una gramática está en FNG si todas sus producciones tienen la forma:
A -> a α, donde a ∈ Σ_T (un terminal) y α ∈ Σ_N *(cero o más no terminales) o** eventualmente A -> λ si A es el axioma (para permitir la vacía).
Es decir, cada producción comienza con un símbolo terminal seguido opcionalmente de algunos no terminales. Ninguna producción comienza con no terminal en el lado derecho (excepto la vacía del axioma si aplica). Esta forma normal es útil para ciertos algoritmos de análisis sintáctico (en teoría de

compiladores, por ejemplo, la FNG garantiza que un parser top-down puede avanzar al menos un símbolo terminal por producción aplicada).

Conversión a FNG: Cualquier gramática libre de contexto sin recursión izquierda se puede transformar en FNG. El algoritmo (Simplificación de Greibach) es más complejo:
1. Primero, eliminar recursividad izquierda (directa e indirecta) de la gramática. Esto se hace reordenando no terminales y sustituyendo producciones.
2. Luego, ordenar los no terminales (A_1, A_2, ..., A_n).
3. Iterativamente, para i de 1 a n: para cada producción A_i -> A_j y (que empieza con un no terminal A_j que tiene índice menor j < i), se reemplaza expandiendo A_j (sustituyendo las alternativas de A_j en esa producción). Esto eventualmente elimina cualquier producción cuyo lado derecho empiece con no terminal de índice menor.
4. Después de eso, es posible que queden producciones como A -> A α (recursividad izquierda indirecta a través de orden), pero esas ya se manejaron en paso 1. Lo que queda son producciones que comienzan con terminales. Si alguna producción empezara con más de un terminal (imposible en GIC, sería terminal-terminal consecutivos, pero eso es simplemente dos terminales seguidos, que sigue cumpliendo FNG pues arranca con un terminal).
5. Asegurarse de incluir S->λ si necesario.

Después de esos pasos se logra FNG: cada regla arranca con un terminal.

Nota: Convertir a FNG puede introducir varios no terminales nuevos y requiere eliminar recursividad izquierda primero, ya que FNG no puede representarla (porque recursión izquierda implicaría A -> A... y en FNG A debe producir un terminal primero, no A).

Tanto FNC como FNG mantienen la potencia expresiva: cualquier lenguaje independiente del contexto (excepto quizá el que es solo λ) tiene una gramática en FNC, y cualquier lenguaje independiente del contexto que no comience con λ en todas sus cadenas tiene una gramática en FNG.

Ejercicios sugeridos (Normalización – FNC y FNG)

1. Transformar la siguiente gramática a Forma Normal de Chomsky:
   G = ({a,b}, {S, A}, S, { S := A b | b, A := a A | a }).
   (Sugerencia: primero limpiar si es necesario, luego eliminar terminales mezclados, luego binarizar producciones.)
2. Dada la gramática G = ({0,1}, {S, A, B}, S) con:
   S := A0 | B1
   A := 0A | 1
   B := 1B | 0
   a) Convertir G en una gramática bien formada (sin renombramientos ni λ excepto S->λ si aplica).
   b) Convertir la gramática resultante a Forma Normal de Greibach (todas las reglas deben comenzar con terminal).
3. Sea G = ({a, b, c}, {S, X, Y}, S) con producciones:
   S := X | Y
   X := a X b | c
   Y := X Y | λ
   a) Eliminar cualquier recursividad izquierda (nota: Y -> X Y tiene recursión indirecta a través de X?). **b)** Ordenar los no terminales (por ejemplo S= A_1, X=A_2, Y=A_3) e iniciar la conversión a FNG eliminando producciones que comiencen con no terminal de menor índice paso a paso. **c)** Proporcionar la gramática final en FNG.
