**Lenguajes formales y autómatas -Resumen teoría-**

**Conceptos**

● Proposición: una sentencia que es o bien verdadera o bien falsa.

● Letras proposicionales para denotar proposiciones: P: está nublado; Q: llueve; R: hace frío

● Se pueden combinar proposiciones simples en más complejas utilizando conectivos (como en nuestro lenguaje)

Conectivos: sintaxis

¬ not/no (negación)

∧ and/y (conjunción)

∨ or/o (disyunción)

→ condicional (implicación)

**tablas de verdad**

| A | B | A∧B |
|---|---|---|
| true | true | **true** |
| true | false | **false** |
| false | true | **false** |
| false | false | **false** |

| A | B | A∨B |
|---|---|---|
| true | true | **true** |
| true | false | **true** |
| false | true | **true** |
| false | false | **false** |

| A | B | A→B |
|---|---|---|
| true | true | **true** |
| true | false | **false** |
| false | true | **true** |
| false | false | **true** |

| A | ¬A |
|---|---|
| true | **false** |
| false | **true** |

A ∧ B (a y b) solo es verdadero si ambas son verdaderas

A ∨ B (a o b) solo es falsa si ambas son falsas

A → B (de a deriva b) solo es falsa si A es verdadera y B es falsa

A, ¬A son opuestas

Puedo comprobar si dos fbf son equivalentes si tienen la misma tabla de verdad o con una prueba de equivalencia:

Probar que: $A \rightarrow (B \rightarrow C) \equiv B \rightarrow (A \rightarrow C)$

$$
\begin{aligned}
A \rightarrow (B \rightarrow C) &\equiv A \rightarrow (\neg B \vee C) && \text{conversión } A \rightarrow B \equiv \neg A \vee B \\
&\equiv \neg A \vee (\neg B \vee C) && \textit{idem} \text{ anterior} \\
&\equiv (\neg A \vee \neg B) \vee C && \text{asociatividad disyunción} \\
&\equiv (\neg B \vee \neg A) \vee C && \text{conmutatividad disyunción} \\
&\equiv \neg B \vee (\neg A \vee C) && \text{asociatividad disyunción} \\
&\equiv B \rightarrow (\neg A \vee C) && \text{conversión } A \rightarrow B \equiv \neg A \vee B \\
&\equiv B \rightarrow (A \rightarrow C) && \textit{idem} \text{ anterior}
\end{aligned}
$$

Cada fbf tiene una única tabla de verdad. Una fbf es:

- una tautología, si todos los valores de verdad en su tabla de verdad son true

- una contradicción, si todos los valores de verdad en su tabla de verdad son false

- una contingencia, si algunos valores de verdad de su tabla de verdad son true y otros son false

**Conversiones**

$$A \rightarrow B \equiv \neg A \vee B \qquad \neg(A \wedge B) \equiv \neg A \vee \neg B \qquad A \wedge (A \vee B) \equiv A$$

$$\neg(A \rightarrow B) \equiv A \wedge \neg B \qquad \neg(A \vee B) \equiv \neg A \wedge \neg B \qquad A \vee (A \wedge B) \equiv A$$

$$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C) \qquad A \wedge (\neg A \vee B) \equiv A \wedge B$$

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C) \qquad A \vee (\neg A \wedge B) \equiv A \vee B$$

Definición: Un literal es una variable proposicional, o la negación de una variable proposicional

Ejemplos: P, Q, ¬P, ¬Q

**Formas normales:**

- forma normal disyuntiva (FND) de una fbf: (¬Q ∧ ¬P ∧ R) ∨ (¬Q ∧ ¬P)
- forma normal conjuntiva (FNC) de una fbf: (¬Q ∨ ¬R) ∧ (¬Q ∨ ¬P ∨ R)

**Formas normales completas:**

Una formula en forma normal completa es una formula en forma normal DISYUNTIVA (FND) o Conjuntiva (FNC) que cumple con la condición de que cada conjunción fundamental tiene exactamente n literales uno porcada una de las n variables proposicionales que aparecen en W.

Por ejemplo, si W tiene las variables proposicionales p, q, y r, entonces una FNDC podría ser: (¬Q ∨ ¬R V P) ∧ (¬Q ∨ ¬P ∨ R).

Podemos obtener una forma normal completa con:

1. Método por equivalencias
2. Método de tablas de verdad:

   Escribe todas las combinaciones posibles de valores de verdad (verdadero V o falso F) para las variables de W y evalúa W para cada combinación.

**Para obtener la FNDC:**
Selecciona las filas donde W es verdadera (V) y escribe una conjunción fundamental para cada fila seleccionada:

- Si una variable es verdadera, se denota como un literal positivo (p).
- Si una variable es falsa (F), se denota como un literal negado (¬p).
- Combina los literales de cada fila con el operador ∧ ("y").
- Luego, une todas las conjunciones fundamentales con el operador ∨ ("o").

**Para obtener la FNCC:**
Selecciona las filas donde W es falsa (F) y escribe una disyunción fundamental para cada fila seleccionada:

- Si una variable es verdadera, se denota como un literal negado (¬p).

- Si una variable es falsa (FFF), se denota como un literal positivo (p). Combina los literales de cada fila con el operador ∨ ("o").
- Luego, une todas las disyunciones fundamentales con el operador ∧ ("y").

**Primer principio de inducción matemática:**

Si se verifica que: P(1) es verdad, y para todo k, si P(k) es verdad entonces P(k+1) es verdad.

Entonces P(n) es verdad para todo entero positivo n.

**Segundo principio de inducción matemática:**

Si se verifica que: P(1) es verdad, y para todo k, si P(r) es verdad para todo 1 <= r <= k entonces P(k+1) es verdad.

Entonces P(n) es verdad para todo entero positivo n.

| *Reglas de inferencia* | | |
|---|---|---|
| Modus Ponens (MP) $\frac{A, A \rightarrow B}{B}$ | Conjunción (Conj) $\frac{A, B}{A \wedge B}$ | Simplificación (Simp) $\frac{A \wedge B}{A}$ y $\frac{A \wedge B}{B}$ |
| Adición (Ad) $\frac{A}{A \vee B}$ y $\frac{A}{B \vee A}$ | Silogismo Disyuntivo (SD) $\frac{A \vee B, \neg A}{B}$ y $\frac{A \vee B, \neg B}{A}$ | Doble negación (DN) $\frac{\neg\neg A}{A}$ y $\frac{A}{\neg\neg A}$ |
| Contradicción (Contr) $\frac{A, \neg A}{false}$ | Prueba Condicional (PC) $\frac{de\ A, deriva\ B}{A \rightarrow B}$ | Prueba Indirecta (PI) $\frac{de\ \neg A, deriva\ false}{A}$ |

Realizar una prueba utilizando reglas de inferencia implica demostrar la validez de un argumento o derivar una conclusión a partir de un conjunto de premisas.

Ej: (A→ C) → (A→ B V C)

Con PI:

(A→ C) premisa

¬ (A→ B V C) premisa PI (subprueba)

Sin PI:

(A→ C) premisa

a. A premisa subprueba (para A→ B V C)

Luego aplico las reglas de inferencia para llegar a una conclusión.

Un sistema de prueba para la lógica proposicional se dice

Sensato (sano): si toda fbf que es teorema en el sistema es una tautología.

Completo: si toda fbf que es tautología es un teorema en el sistema.

**Prolog**

Variables: se representan con identificadores que empiezan con Mayúscula. *Ejemplos:* Ciudad, Mes, Nombre1.

Las constantes representan valores fijos. Hay dos tipos de constantes: números, y átomos. Los átomos se representan con identificadores que empiezan con minúscula. *Ejemplos:* bahia_blanca, octubre

Estructuras: se denota como un *átomo (es decir, tiene un nombre que empieza con minúscula)* y contiene una secuencia de argumentos dentro de paréntesis (*puede aplicarse recursivamente*).

*Ejemplos:* madre(X,juan), ciudad(bahia_blanca)

El orden importa: padre(juan,tito) ≠ padre(tito,juan) (juan es padre de tito ≠ tito es padre de juan)

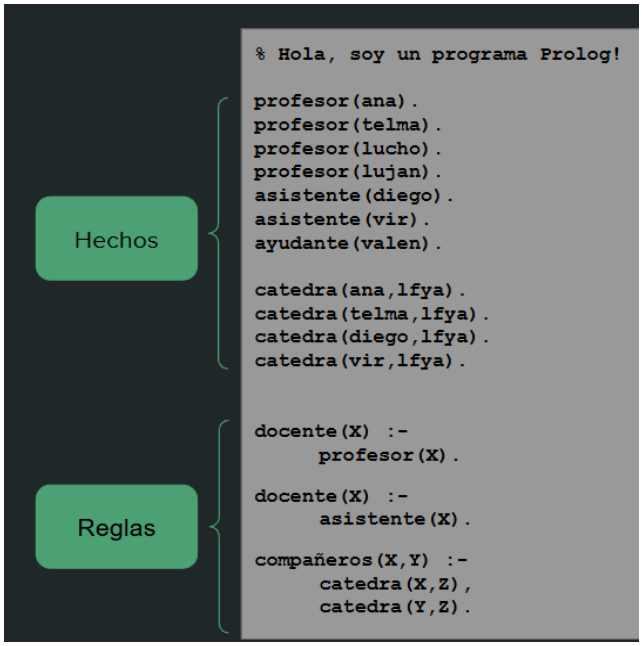
Un programa PROLOG está conformado por reglas y hechos.

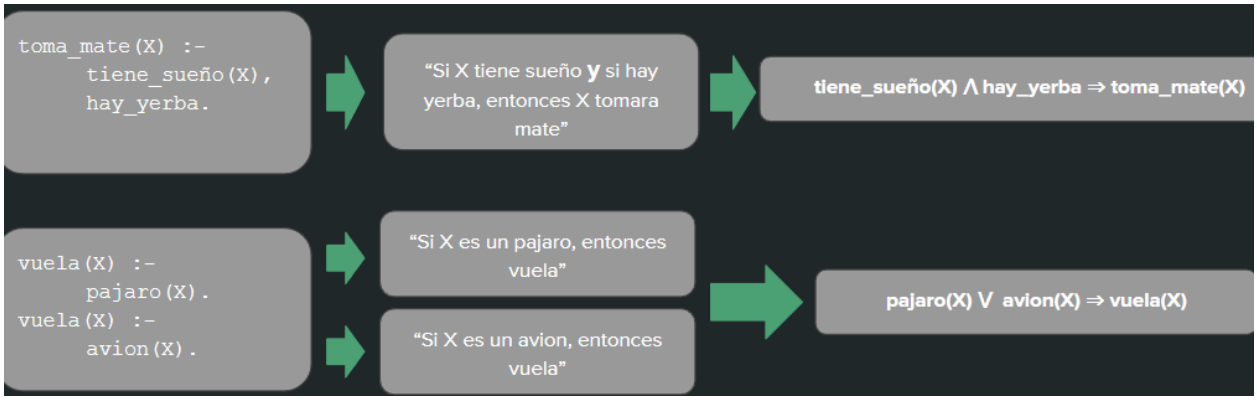
Las reglas permiten relacionar términos entre sí y tienen la forma:

<conclusion> :- <antecedente>.

Se interpretan como *"Si se cumple el antecedente (1 o más términos), entonces se cumple la conclusión (1 término)"*

Ej: *Si X es profesor, entonces X es docente.*




puedo realizar consultas a dicho programa para verificar qué información "vale"

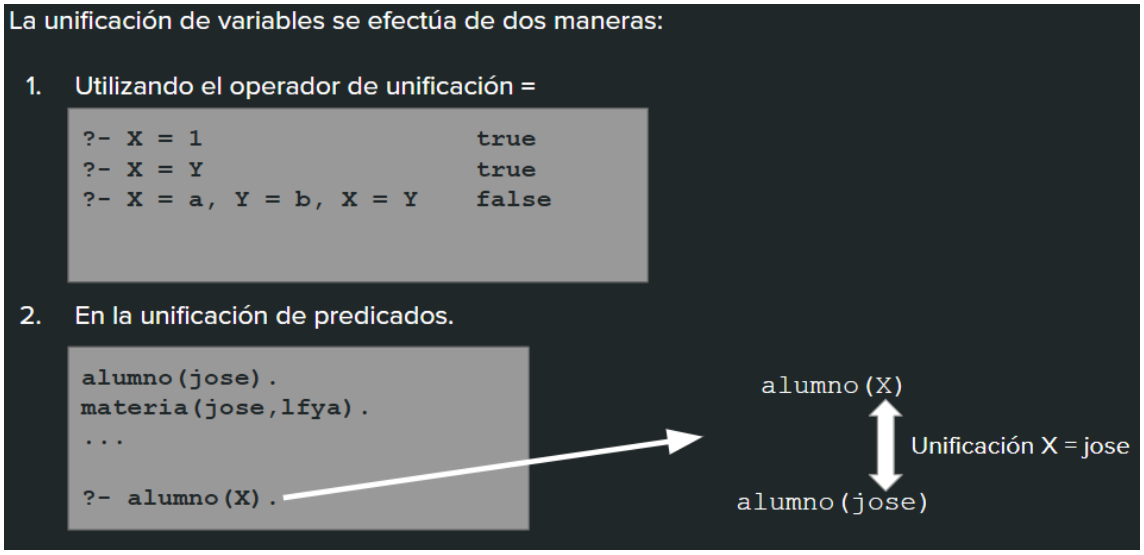
Ej: ?- compañeros(ana,telma): true ?- docente(valen): false

Unificación:

Una variable se puede unificar con cualquier cosa, siempre que la misma no tenga un valor asignado.

Dos constantes se pueden unificar si son idénticas.

Dos estructuras se unifican solo si tienen el mismo nombre, el mismo número de argumentos, y cada par de argumentos se puede unificar.




El operador \= nos permite verificar si 2 términos no son unificables.

Ej: X \= 4. False porque x si puede unificarse con 4

f(a,X) \= f(b,Y). true porque si bien ambas f tienen misma cantidad de parámetros, a no se puede unificar con b.

El operador IS nos permite unificar variables con expresiones numéricas:

“X is Y nos devuelve true si X se unifica con el valor resultante de evaluar la expresion Y”

Sea P un predicado, not(P) tendrá éxito cuando P no pueda probarse.

**Propiedades relaciones binarias:**

Reflexividad: para todo elemento a ∈ A, se cumple que aRa. Ejemplo**:** A = {1,2}, R={(1,1),(2,2)}

Simetría: para todo a,b ∈ A, si aRb, entonces bRa. Ejemplo**:** A = {1,2}, R = {(1,2),(2,1)}

Transitividad: para todo a,b,c ∈ A, si aRb, y bRc entonces aRc. Ej**:** A= {1,2,3}, R={(1,2),(2,3),(1,3)}

{Relación de equivalencia}

Antisimetría: para todo a,b ∈ A, si aRb, y bRa entonces a = b. Ej**:** A={1,2}, R={(1,1),(2,2),(1,2)}

Partición: Una partición de S es una colección no vacía de subconjuntos disjuntos de S cuya unión es igual a S

Ejemplo: S = {1,2,3,4,5}

- La colección {{1}, {2,3,4}, {5}} es una partición
- La colección {{1,3,5}, {2,4}} es una partición
- La colección {{1,2}, {4,5}} no es una partición
- La colección {{1,2,3}, {3,4,5}} no es una partición

**Clase de equivalencia [x]:**

- Si tienes una relación de equivalencia R sobre un conjunto S, para cualquier elemento x∈S, la clase de equivalencia [x] es el grupo de elementos de S que están relacionados con x.

- En otras palabras, [x] contiene todos los elementos que son "iguales" a x según R.

Ejemplo:
Si S= {1,2,3,4,5,6} la relación R dice que dos números son equivalentes si tienen el mismo residuo al dividirse entre 3, entonces:

- [1] = {1,4} (los números con residuo 1).
- [2] = {2,5} (los números con residuo 2).
- [0] = {3,6} (los números con residuo 0).

**Conjunto cociente S/R**

- Es el conjunto formado por todas las clases de equivalencia de S.
- En el ejemplo, S/R={{1,4},{2,5},{3,6}}

**Clausuras**

Una relación binaria ρ* sobre S es la clausura de una relación ρ sobre S con respecto a una propiedad P si

- ρ* cumple P
- $\rho \subseteq \rho^*$
- ρ* es subconjunto de cualquier otra relación sobre S que incluya ρ y tenga la propiedad P ($\rho*$ no tiene elementos extra innecesarios.)

Ejemplo:

Si S={a,b,c} } y ρ={(a,b),(b,c)}, queremos encontrar la clausura $\rho*$ para que sea transitiva:

- $\rho*$ cumple transitividad: Si (a,b)y (b,c) están en la relación, entonces (a,c) debe estar en $\rho*$.
- $P \subseteq \rho$: Los pares originales (a,b) y (b,c) siguen estando en $\rho*$.
- $\rho*$ es la más pequeña: No agregamos más pares que los necesarios para hacerla transitiva.

Por lo tanto, $\rho*$={(a,b),(b,c),(a,c)} es la clausura transitiva de ρ.

Dada una relación R definida sobre S, denotamos:

- Clausura reflexiva: r(R)
- Clausura simétrica: s(R)
- Clausura transitiva: t(R)

(estas tres clausuras siempre existen, pueden construirse).

Si R es una relación binaria sobre S, entonces t(s(r(R))) es la relación de equivalencia más pequeña que contiene a R.

Grafos: conjunto de nodos conectados entre sí a través de arcos (flechas).

Dos nodos son adyacentes si hay una flecha que los conecta. Es decir, si hay una línea entre A y B, entonces A y B son adyacentes.

Grafos dirigidos (o dígrafos): Si los arcos tienen una dirección. Por ejemplo, una flecha de A a B significa que la conexión va solo en esa dirección, no al revés.

**Definición formal de un grafo:**

- Un grafo G se describe como un par (V, E):
    - V es el conjunto de vértices (los nodos).
    - E es el conjunto de arcos (las conexiones entre nodos).
- Si el grafo es dirigido, los arcos se escriben como pares ordenados (a,b), donde la flecha va de a a b.

**Relación binaria y grafos:**

Una relación binaria definida sobre un conjunto S, puede ser representada como un grafo donde cada elemento de S es un nodo, y cada par ordenado en la relación es una flecha (o arco) entre esos nodos.

Ejemplo:

- Supón que S= {A, B, C} y la relación R= {(A, B), (B, C)}.
- El grafo correspondiente tiene:
    - Nodos: A, B, C
    - Aristas: Una flecha de A a B y otra de B a C.

**Tipos de grafos**

Etiquetados: los nodos tienen asociada información adicional, como un nombre, un número, o algún dato extra.

Ponderados: cada arco tiene asociado un valor numérico o peso. Este peso puede representar algo como distancia, costo, tiempo, etc.

Multigrafo: permite tener varias conexiones (arcos paralelos) entre los mismos nodos.

**Matrices**

Matriz de adyacencia del grafo

Qué representa:
Es una tabla (o matriz) que muestra cómo están conectados los vértices de un grafo.

Dimensión:
Si el grafo tiene n vértices (v1,v2,…,vn) la matriz tendrá tamaño n×n es decir, n filas y n columnas.

Contenido:

- Cada entrada [i,j] de la matriz nos dice cuántos arcos (o conexiones) hay desde el vértice vi al vértice vj.
- En un grafo no dirigido, [i,j]=[j,i] porque las conexiones no tienen dirección.
- En un dígrafo (grafo dirigido), el valor [i,j] puede ser diferente de [j,i], ya que las conexiones tienen dirección.

Ejemplo**:** Supongamos que tenemos un grafo con 3 vértices v1,v2,v3 y las siguientes conexiones:

- v1→v2
- v2→v3
- v3→v1

| | | |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 1 |
| 1 | 0 | 0 |

  - Aquí, el valor en [1,2] es 1 porque hay un arco de v1 a v2.
  - El valor en [2,3] es 1 porque hay un arco de v2 a v3
  - El valor en [3,1] es 1 porque hay un arco de v3 a v1
    Los demás valores son 0 porque no hay conexión directa entre esos vértices.

La matriz de adyacencia puede ser numérica o booleana, dependiendo del tipo de grafo:

- Si el grafo es no ponderado y no tiene arcos paralelos, la matriz de adyacencia será booleana (0 o 1).
- Si el grafo es ponderado, la matriz de adyacencia contiene los pesos de los arcos (0 si no hay conexión, o el valor del peso si hay conexión).

**Relaciones de orden parcial**

Decimos que una relación R ($\preccurlyeq$) es una relación de orden parcial si es transitiva, antisimétrica y reflexiva.

Una relación de orden es total (o lineal) si todos los pares de elementos son comparables.

- Es decir, para cualquier $x,y \in S$ si $xRy$ o $yRx$

Si $x \preccurlyeq y$ decimos que x es predecesor de y, o que y es sucesor de x. Si x es predecesor de y, y no existe un z tal que $x \preccurlyeq z \preccurlyeq y$, entonces decimos que x es predecesor inmediato de y (a su vez, y es sucesor inmediato de x).

**Diagrama de hasse**

Si se verifica que $x \preccurlyeq y$ y resulta que x es un predecesor inmediato de y, dibujamos un segmento (no flecha) entre x e y, dibujando x por debajo (en un nivel inferior) de y

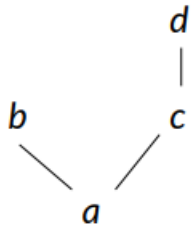


Diagrama de Hasse

Cadena: una relación es un orden total, o sea todos los elementos son comparables.

Anticadena: en una relación todos los elementos son incomparables.

**Elemento Maximal:**
En un conjunto ordenado, un elemento m es maximal si:

- No hay ningún otro elemento mayor que m, salvo él mismo.
- Es decir, si $m \preceq x$ (m está antes o igual que x), entonces $x = m$.

**Elemento Minimal:**
En un conjunto ordenado, un elemento m es minimal si:

- No hay ningún otro elemento menor que m, salvo él mismo.
- Es decir, si $x \preceq m$, entonces $x = m$.

Nota: Pueden existir varios elementos maximales o minimales en un conjunto y todo conjunto ordenado finito tiene, por lo menos, un elemento minimal o maximal.

**Primer elemento**

- Un primer elemento en un conjunto ordenado $(R,\preceq)$ es un elemento que esta antes o igual que todos los elementos de R.

**Ultimo elemento**

- Un último elemento en un conjunto ordenado $(R,\preceq)$ es un elemento que esta después o igual que todos los elementos de R.

Nota: El primer y último elemento son únicos si existen.

**Ínfimo ($\wedge$) de a y b:**

- El ínfimo de dos elementos a y b en un conjunto ordenado $(R,\preceq)$ es el mayor elemento común de a y b que está por debajo de ambos.

**Supremo (V) de a y b:**

- El supremo de dos elementos a y b en un conjunto ordenado $(R,\preceq)$ es el menor elemento común de a y b que está por encima de ambos.

Nota: si existe el ínfimo o supremo de dos elementos, es único.

**Reticulados**

Un conjunto ordenado $(R,\preccurlyeq)$ se dice:

- reticulado superior si todo par de elementos de R tiene supremo
- reticulado inferior si todo par de elementos de R tiene ínfimo
- reticulado si es reticulado inferior y reticulado superior.

**Orden topológico**

Un ordenamiento topológico es un procedimiento para encontrar el orden total a partir de un orden parcial, sobre un conjunto finito. Es útil para tareas donde hay una dependencia entre elementos.

Orden parcial: conjunto de elementos donde no todos están relacionados directamente. Por ejemplo: "A debe ir antes que B." y "C debe ir antes que D." Pero no hay información sobre la relación entre A y C.

Orden total: El orden topológico organiza todos los elementos de manera completa, respetando las relaciones iniciales. Siguiendo el ejemplo: Podríamos ordenar: A,B,C,D.

Reglas principales:

- Si en el orden parcial sabes que "x debe ir antes que y" (xPy), entonces en el orden total también "x debe ir antes que y" (xTy).
- El orden topológico solo funciona si no hay ciclos en las relaciones, es decir, no puedes tener algo como "A depende de B, y B depende de A".

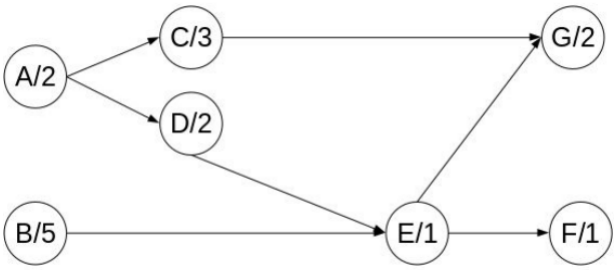
Algoritmo de Orden Topológico:

1. Inicializa una lista vacía Orden ([]).
2. Mientras el conjunto S no esté vacío:
    - Elige un elemento minimal m de S.
    - Añade m al final de la lista Orden.
    - Elimina m de S.
3. Retorna la lista Orden con el orden total.

**Diagrama Pert**

Los nodos son las tareas. Cada tarea tiene un tiempo de ejecución asociada.

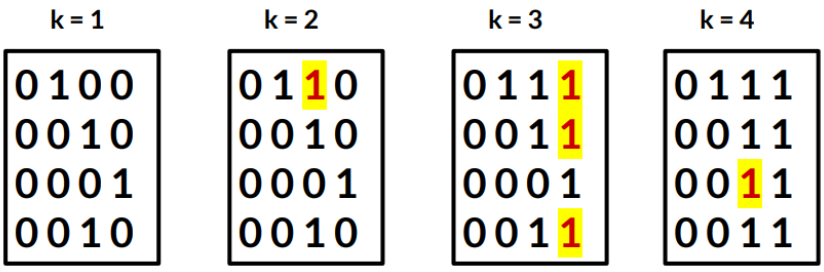
Para poder ejecutar una tarea, primero debe haber finalizado la ejecución de sus predecesoras inmediatas (y por lo tanto de todas sus predecesoras). Eso determina el tiempo mínimo necesario para completar todo el proyecto (todas las tareas)




El camino critico es la secuencia más larga de tareas dependientes, es decir, aquellas que no pueden ser retrasadas sin que el proyecto en su totalidad se retrase. Determina el tiempo mínimo necesario para completar el proyecto

**Algoritmo de Warshall**

Este algoritmo se utiliza para calcular la clausura transitiva de una relación binaria representada por una matriz de adyacencia booleana. Básicamente, busca todas las conexiones posibles entre los elementos de un conjunto a través de otros elementos. Si un nodo A está relacionado con un nodo B, y B está relacionado con C, el algoritmo asegura que A también estará relacionado con C (indirectamente).




En k= 1:

- Nodo 1 está conectado a Nodo 2.
- Nodo 2 está conectado a Nodo 3.
- Nodo 3 está conectado a Nodo 4.

- Nodo 4 está conectado a Nodo 3.

En k= 2:

- Nodo 1 está conectado a Nodo 3 (indirectamente a través de Nodo 2).

En k= 3:

- Nodo 1 está conectado a Nodo 4 (indirectamente a través de Nodo 2 y 3).
- Nodo 2 está conectado a Nodo 4 (indirectamente a través de Nodo 3).
- Nodo 4 está conectado a Nodo 4 (indirectamente a través de Nodo 3).

En k= 4:

- Nodo 3 está conectado a Nodo 3 (indirectamente a través de Nodo 4).

**El algoritmo de Floyd-Warshall**

Es un algoritmo para encontrar la distancia más corta entre todos los pares de nodos en un grafo ponderado (cada arco tiene un peso o valor). Este algoritmo trabaja con una matriz de adyacencia ponderada, donde las entradas representan las distancias o pesos de los arcos entre los nodos.

Matriz de adyacencia ponderada: Es una matriz M donde el valor M[i,j] representa el peso de la arista entre los nodos i y j. Si no existe una arista entre i yjj, se representa como ∞ (infinito).

| | | | | | |
|---|---|---|---|---|---|
| 0 | 10 | 10 | ∞ | 20 | 10 |
| ∞ | 0 | ∞ | 30 | ∞ | ∞ |
| ∞ | ∞ | 0 | 30 | ∞ | ∞ |
| ∞ | ∞ | ∞ | 0 | ∞ | ∞ |
| ∞ | ∞ | ∞ | 40 | 0 | ∞ |
| ∞ | ∞ | ∞ | ∞ | 5 | 0 |

nodo k=1: fila 1, columna 1

nodo k=2: fila 2, columna 2

…

Comienza con la matriz M tal como está en el grafo ponderado, donde:

- M[i,j] peso de la arista entre los nodos i y j
- Si no hay una arista entre i y j se representa como ∞ (infinito).
- M[i,i]=0 porque la distancia de un nodo consigo mismo es 0.

Proceso: El algoritmo de Floyd-Warshall itera sobre todos los nodos k, i, y j para actualizar la matriz de distancias. Para cada combinación de nodos i, j, y k, se verifica si la distancia de i a j a través de k es más corta que la distancia directa de i a j. Si es así, se actualiza la distancia. Se repite para cada nodo k.

**Conceptos:**

- Alfabeto: es un conjunto no vacío, finito de símbolos.
- Cadena: es una secuencia finita de símbolos de algún alfabeto.
- Cadena vacía (λ): es una secuencia con ningún símbolo.

- Longitud de una cadena: es la cantidad de símbolos que hay en la cadena (ej. 1001 es una cadena de longitud 4)
- Si Σ es un alfabeto definimos $\Sigma^k$ al conjunto de cadenas de longitud k, definidas sobre el alfabeto Σ.
- El conjunto de todas las cadenas definidas sobre un alfabeto Σ se denota como Σ* y se define como $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$
- Lenguaje: Un lenguaje definido un alfabeto Σ es un subconjunto de Σ* (son conjuntos de cadenas).

**Operaciones sobre lenguajes**

● Unión ($L_1 \cup L_2$): Todas las cadenas que están en L1 y las que están en L2.
Ej: $L_1 = \{a,b\}$ y $L_2 = \{b,d\}$, entonces: $L_1 \cup L_2 = \{a,b,d\}$.

● Intersección ($L_1 \cap L_2$): Todas las cadenas que están tanto en L1 como en L2.
Ej: $L_1 = \{a,b\}$ y $L_2 = \{b,d\}$, entonces: $L_1 \cup L_2 = \{b\}$.

● Diferencia ($L_1 / L_2$): Todas las cadenas que están en L1 pero no en L2.
Ej: $L_1 = \{a,b,c\}$ y $L_2 = \{b,d\}$, entonces: $L_1 \cup L_2 = \{a,c\}$

● Complemento ($L^c$): Todas las cadenas que no están en el lenguaje, pero que pertenecen al conjunto universal de cadenas definido sobre el alfabeto Σ.
Ej: Σ = {a,b}, L = {a,b}, entonces $L^c = \{\lambda,aa,bb,ab,ba,aaa,bba,\ldots\}$

Sean L, L1 y L2 lenguajes definidos sobre un alfabeto Σ, las operaciones propias de lenguajes son:

Concatenación de lenguajes $L_1 \cdot L_2 = \{w_1 \cdot w_2 : w_1 \in L_1 \text{ y } w_2 \in L_2\}$

La concatenación debe ser una cadena $w_1$ de $L_1$ seguida por una cadena $w_2$ de $L_2$.
No se permite mezclar elementos de $L_1$ y $L_2$ de manera intercalada.
$L_1 \cdot L_2 \neq L_2 \cdot L_1$

Ejemplo**:** Si: $L_1 = \{a,b\}$ y $L_2 = \{c,d\}$ entonces: $L_1 \cdot L_2 = \{ac,ad,bc,bd\}$

Clausura (de Kleene) o estrella $L^* = \{w_1 \cdot w_2 \cdot \ldots w_n : w_i \in L, 0 \leq i \leq n\}$

La clausura de Kleene de un lenguaje L incluye todas las cadenas que se pueden formar concatenando cero o más cadenas del lenguaje L.

Ejemplo**:** Si L={a,b}, entonces: $L*=\{\lambda,a,b,aa,ab,ba,bb,aaa,aab,\ldots\}$

Esto incluye todas las combinaciones posibles de a y b, incluso la cadena vacía.

**Autómatas**

Es un modelo para representar cómo un sistema cambia su estado en base a entradas o interacciones.

**Autómata finito determinista (AFD)**

1. Tienen un número finito de estados.
2. Cambian de estado dependiendo de lo que lean o procesen del entorno.
3. Son deterministas, es decir, para cada entrada hay un y solo un estado al cual el autómata puede transicionar desde su estado actual.

AFD (reconocedor): se denota con la quíntupla M = <S, Σ, δ, s0, F>

- S: Conjunto de estados.
- Σ: Alfabeto (símbolos de entrada).
- δ: Función de transición.
- s0: Estado inicial.
- F: Conjunto de estados aceptadores

Ejemplo: Un autómata que reconoce L = {w| w e {0,1}* tal que la cantidad de 1´s es impar} :

- Estados: S = {s0,s1}
- Alfabeto: Σ = {0,1}
- Estado inicial: s0 (->)
- Estados aceptadores: F = {s1} (*)
- Función de transición:

| | 0 | 1 |
|---|---|---|
| → s0 | s0 | s1 |
| *s1 | s1 | s0 |

**Autómata finito No Determinista (AFND)**

Se denota con la misma quintupla que los AFD, es decir M = <S, Σ, δ, s0, F>.

Se diferencia en que, en un AFND, desde cada estado y con cada símbolo de entrada, puede haber más de 1 estado destino. Ademas puede cambiar de estado sin consumir un símbolo de entrada (denotado como λ).

**Autómatas finitos traductores**

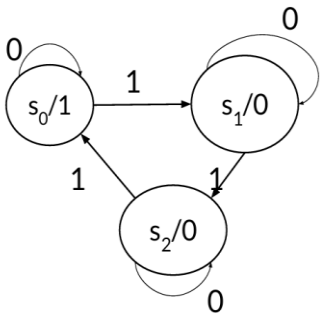
Toman una cadena de entrada y la traducen en una salida. Pueden ser automatas de Mealy o de Moore. Ambos son una séxtupla M = (S, Σ, Γ ,δ, s0 , f0 )

- S: Conjunto de estados. S≠Ø.
- Σ: Alfabeto de entrada.
- Γ: Alfabeto de salida.
- δ: Función de transición.
- s0: Estado inicial.
- f0: Función de salida.

Autómatas de Moore: las salidas van asociadas a los estados.

$M = <S, \Sigma, \Gamma, \delta, s_0, f_0>$
$S=\{s_0, s_1, s_2\}, \Sigma=\{0,1\}, \Gamma=\{0,1\}$

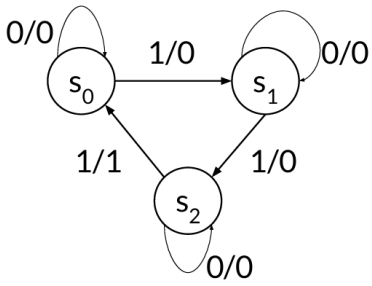
| δ | 0 | 1 | $f_0$ |
|---|---|---|---|
| $s_0$ | $s_0$ | $s_1$ | 1 |
| $s_1$ | $s_1$ | $s_2$ | 0 |
| $s_2$ | $s_2$ | $s_0$ | 0 |



Autómatas de Mealy: las salidas van asociadas a las transiciones.

$M = <S, \Sigma, \Gamma, \delta, s_0, f_0>$
$S=\{s_0, s_1, s_2\}, \Sigma= \{0,1\}, \Gamma=\{0,1\}$

| $\delta/f_0$ | 0 | 1 |
|---|---|---|
| $s_0$ | $s_0/0$ | $s_1/0$ |
| $s_1$ | $s_1/0$ | $s_2/0$ |
| $s_2$ | $s_2/0$ | $s_0/1$ |




**AFD (reconocedor)**

Un AFD procesa cadenas de entrada, símbolo a símbolo, transicionando por sus estados según su función de transición y al terminar de procesar/leer la cadena decide si la acepta o no. De esa forma, se puede asociar un lenguaje a un AFD: el lenguaje asociado al autómata es el conjunto de todas las cadenas que terminan en un estado aceptador (son aceptadas).

**Función de transición extendida $_{AFD}$**

La función δ* describe cómo el autómata transacciona desde un estado inicial a un estado final al procesar una cadena completa. Esto permite determinar si la cadena es aceptada por el autómata verificando si el estado final pertenece al conjunto de estados aceptadores (F).

$\delta^*(s,\lambda) = s$ (si se procesa una cadena vacía (λ) permanece en el mismo estado)

$\delta^*(s,w) = \delta(\delta^*(s,x),a)$ en el caso que w=xa

Obs: a es el último símbolo de la cadena w

**Lenguaje de un AFD**

El lenguaje de un AFD M = <S, Σ, δ, s0, F> es el conjunto de todas las cadenas que el autómata puede aceptar. Se denota L(M) y se define: $L(M) = \{w: \delta^*(s0, w) \in F\}$

Esto significa que w pertenece al lenguaje L(M) si, al procesar la cadena w desde el estado inicial s0, el autómata termina en algún estado aceptador (F).

**Función de transición extendida $_{AFND}$**

La función de transición extendida $\delta*$ de un AFND indica a qué conjunto de estados se puede llegar al procesar una cadena desde un estado inicial. A diferencia de los AFD, donde hay una única transición para cada símbolo y estado, en un AFND pueden existir múltiples estados posibles debido al no determinismo.

$\delta^*(s, \lambda) = s$ (si se procesa una cadena vacía (λ) permanece en el mismo estado).

Si la cadena w es xa (donde a es el último símbolo), primero se encuentran los estados alcanzables con x y luego, desde cada uno, se aplican las transiciones para a.

Ejemplo para la cadena “aab”:

$\delta^*(q0, \lambda) = \{q0\}$

Primer símbolo "a": $\delta^*(q0,"a") = \delta(q0,"a") = \{q0, q1\}$

Segundo símbolo "a":

Aplicamos δ(a) a cada estado alcanzado (q0, q1): $\delta(q0,"a") = \{q0, q1\}$, $\delta(q1,"a") = \{q2\}$

Unión de los resultados: δ∗(q0,"aa") = {q0, q1, q2}

Tercer símbolo "b":

Aplicamos δ(a) a cada estado alcanzado: δ(q0,"b") = ∅, δ(q1,"b") = ∅, δ(q2,"b") = {q3}

Unión de los resultados: δ∗(q0,"aab") = {q3} (estado final).

Nota: (un AFD o AFND puede tener más de un estado final (estados aceptadores)).

**Lenguaje de un AFND**

Se denota L(M) y se define como L(M) = {w: δ*(s0, w) ∩ F ≠ ∅} Es decir, si al aplicar la función de transición extendida δ*(s0, w) alguno de los estados alcanzados está en el conjunto de estados aceptadores F, entonces la cadena es aceptada.

(Si fuese "δ*(s0, w) ∈ F ≠ ∅", significaría que todo el conjunto de estados alcanzables debe estar en F, lo cual no es correcto, ya que, a diferencia de un AFD, en un AFND, pueden existir varios caminos y basta con que uno de esos estados pertenezca a F. es por ello que se escribe como "∩").

**Equivalencia entre AFD y AFND**

Todo Autómata Finito No Determinista (AFND) puede transformarse en un Autómata Finito Determinista (AFD) equivalente, es decir, que reconozca el mismo lenguaje.

Para construir un AFD a partir de un AFND, utilizamos el método de subconjuntos, donde cada estado del AFD representa un conjunto de estados del AFND.

Los estados aceptadores del AFD son aquellos subconjuntos que contienen al menos un estado aceptador del AFND.

Ejemplo

Supongamos un AFND con:

- Estados: S={A,B,C}
- Símbolos de entrada: Σ = {0,1}
- Estado inicial: s0 = A
- Estados Aceptadores: F={c}
- Transiciones:

| δ | 0 | 1 |
|---|---|---|
| ->A | {A,B} | {A} |
| B | ∅ | {C} |
| *C | ∅ | ∅ |

Construimos el AFD con estados representando subconjuntos de S:

- Estados: {∅,{A},{B},{C},{A,B},{A,C},{B,C},{A,B,C}}
- Estado inicial: {A}
- Estados aceptadores: Todo subconjunto que contenga C: {C},{A,C},{B,C},{A,B,C}.
- Transiciones:

| δ | 0 | 1 |
|---|---|---|
| ∅ | ∅ | ∅ |
| -> {A} | {A,B} | {A} |
| {B} | ∅ | {C} |
| *{C} | ∅ | ∅ |
| {A,B} | {A,B} U ∅ | {A,C} |
| *{A,C} | {A,B} | {A} |
| *{B,C} | ∅ | {C} |
| *{A,B,C} | {A,B} | {A,C} |

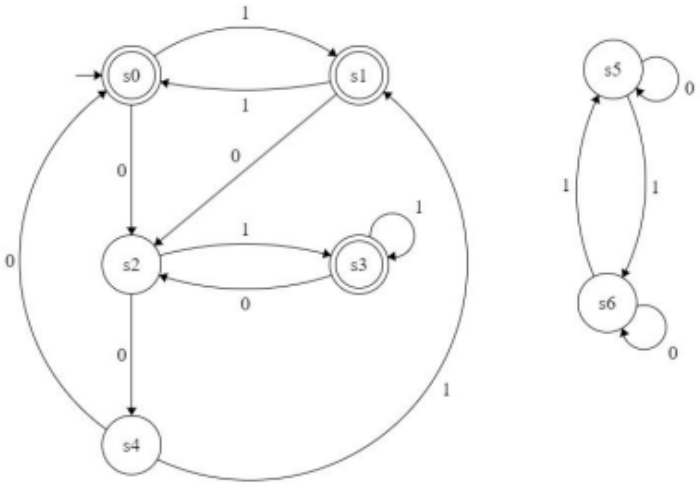
- Este AFD es equivalente al AFND original, y acepta el mismo lenguaje.

**Minimización de autómatas**

Dado un autómata finito M, queremos encontrar un M' que se comporte como M y que tenga menos estados (si es posible).

Pasos:

1. Eliminar estados inalcanzables.
2. Conseguir estados 0-equivalentes
3. Refinar clases k+1 equivalentes
4. Definir M'




1) Elimino estados inalcanzables

Miro el grafo, detectando los estados a los que no puedo llegar con ninguna cadena. Luego, en M' no incluyo dichos estados.

A s5 y s6 no puedo llegar con ninguna cadena desde el estado inicial, son estados inalcanzables.

2) Estados 0-equivalentes

Dos estados $s_i$, son k-equivalentes si para las cadenas de longitud k, llevan a un estado con la misma salida.

- En AFR → Agrupo los estados en "estados que aceptan" y los que no.
- En AFT → Agrupo los estados en "estados que imprimen la misma salida"

Ejemplo: Agrupamos a los estados alcanzables dependiendo si son aceptadores o no…

T = S \ {s5 ,s6 } = {s0 ,s1 ,s2 ,s3 ,s4 } → s0 , s1 , s3 son estados aceptadores → s2 , s4 no lo son
Luego, la clase de equivalencia |0| = {s0 ,s1 ,s3 }, {s2, s4} (s5 y s6 fueron eliminados por ser inalcanzables).

A B

3) clases k+1 equivalentes

s2 y s4 pertenecían a B pero con 0 llevan a clases distintas… Los tengo que separar.

|1| = {s0, s1, s3}, {s2}, {s4}

Como |1| ≠ |0| tengo que seguir..

Dado |1| = {$s_0$,$s_1$,$s_3$}, {$s_2$}, {$s_4$} busco |2|…

A B C

| δ | En qué conjunto esta? | 0 | 1 |
|---|---|---|---|
| → *$s_0$ | A | $s_2$ B | $s_1$ A |
| *$s_1$ | A | $s_2$ B | $s_0$ A |
| $s_2$ | B | $s_4$ C | $s_3$ A |
| *$s_3$ | A | $s_2$ B | $s_3$ A |
| $s_4$ | C | $s_0$ A | $s_1$ A |

Dado |0| = {$s_0$,$s_1$,$s_3$}, {$s_2$,$s_4$} busco |1|…

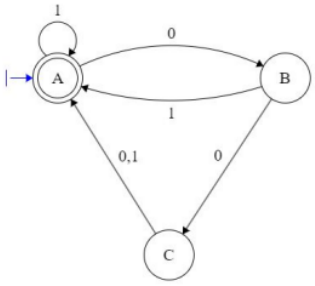
A B

| δ | En qué clase esta? | 0 | 1 |
|---|---|---|---|
| → *$s_0$ | A | $s_2$ B | $s_1$ A |
| *$s_1$ | A | $s_2$ B | $s_0$ A |
| $s_2$ | B | $s_4$ B | $s_3$ A |
| *$s_3$ | A | $s_2$ B | $s_3$ A |
| $s_4$ | B | $s_0$ A | $s_1$ A |

No tuve que hacer separaciones, por lo tanto: |2| = |1|

Ahora tengo que obtener M'.

| δ | 0 | 1 |
|---|---|---|
| → *A | B | A |
| B | C | A |
| C | A | A |




3) Definir M' (S',Σ,δ,A,F)

- S' tiene las clases que definimos anteriormente: S' = {A,B,C}
- Σ se mantiene igual: Σ = {0,1}
- El estado inicial será la clase donde está s0: A.
- F será el conjunto de clases que contenga estados finalizadores: F = {A}
- Definimos δ siguiendo la tabla obtenida anteriormente:

**Expresiones regulares**

Las expresiones regulares (ER) son una notación formal para describir lenguajes regulares, sin necesidad de construir un autómata.

Operadores:

- estrella de Kleene ( * )
- concatenación ( · )
- union (+)

Las expresiones regulares se definen inductivamente de la siguiente manera:

Casos básicos (ERs simples):

Ø representa el lenguaje vacío**:** L(Ø) =Ø

λ (cadena vacía) representa el lenguaje que solo contiene la cadena vacía**:** L(λ) = {λ}

Un símbolo a ∈ Σ representa el lenguaje unitario que contiene solo la cadena formada por ese símbolo: L(a) = {a}.

Operaciones sobre ERs:

Unión: E+F es una ER que denota el lenguaje L(E) ∪ L(F).

Concatenación: E·F es una ER que denota el lenguaje L(E)·L(F), es decir, todas las cadenas que resultan de concatenar una cadena de L(E) con una de L(F).

Cerradura de Kleene: E* una ER que denota el lenguaje L(E)*, es decir, cualquier cantidad (incluyendo cero) de repeticiones de cadenas en L(E).

Agrupación: (E) es una ER que denota el mismo lenguaje que E.

**Gramática**

Una gramática formal suele considerarse como un mecanismo generador de lenguajes. Es una 4-tupla $G = <V_N,V_T,S,P>$ donde:

- Vn es un conjunto de no terminales
- Vt un conjunto de símbolos terminales

- S un símbolo inicial (en Vn)
- P un conjunto finito de producciones de la forma α→β

**Clases de gramáticas**

Estructura de frase (tipo 0). Sin restricciones adicionales.

**Ejemplo de producción:**

- SAB → ASB
- ABC → DE
- S → aBCD

Sensible al contexto (tipo 1): Para cada producción α → β, |α| ≤ |β| (la longitud de α debe ser menor o igual que la longitud de β) (excepto S→λ ).

**Ejemplo de producción:**

- AB → BA
- AC → DEF
- aSb → AbC

Libre de contexto (tipo 2): Para cada producción α → β, α ∈ $V_N$ (es un único **no terminal**) y |α| ≤ |β| (excepto S→λ).

**Ejemplo de producción:**

- S → aSb
- S → λ
- A → aA|b

Regular (tipo 3): Para cada producción : α ∈ $V_N$ y β es de la forma t (un terminal solo) o tW (terminal seguido de un no terminal ), donde t ∈ $V_T$ y W ∈ $V_N$ (excepto S→λ).

**Ejemplo de producción:**

- S → aA
- A → bB
- B → c

Un lenguaje se dice de tipo N si puede ser generado por una gramática de tipo N.

Nota: Los lenguajes reconocidos por máquinas de Turing son los lenguajes de tipo 0.

Dada una gramática de tipo 3 (regular), puedo construir un autómata finito que reconozca el mismo lenguaje. Y a partir de un autómata finito, siempre se puede construir una gramática regular que genere el mismo lenguaje.

La misma relación de Equivalencia existe entre Expresiones Regulares y Autómatas Finitos.

1. De Expresión Regular a Autómata Finito

Podemos construir un Autómata Finito No Determinista (AFND) siguiendo estas reglas:

- Cada símbolo en la ER corresponde a una transición entre estados.
- La concatenación (•) representa un paso secuencial**.**
- La unión (+) crea una bifurcación.
- La cerradura de Kleene (*) indica auto-transiciones**.**

Ejemplo**:** Consideremos la expresión regular: (a+b)c* (empiezan con **a** o b y siguen con **c).**

El AFND equivalente sería:

- Estado inicial → q0
- Desde q0 transiciona con a o b a q1
- Desde q1 bucle con c

Estado final → q1

2. De AF a ER

1. Plantear las ecuaciones para cada estado del autómata basadas en sus transiciones. Al plantear la ecuación para el estado inicial s0, debemos agregar un término adicional para la cadena vacía (λ): ωs0 = λ + transiciones de s0**.**

Ej**:** Si s0, tiene transiciones desde s1 con a y desde s2 con b la ecuación sería: $\omega_{s0} = \lambda + \omega_{s1}\cdot a + \omega_{s2}\cdot b$

2. Despejar las ecuaciones usando el lema ω = β + ω·γ entonces ω = β·γ* (β es una parte de la ER que describe las transiciones hacia otros estados (sin bucles), mientras que γ representa un bucle).
3. La ER final es la unión (+) de las soluciones para los estados aceptadores. Ej**:** Si un autómata tiene dos estados aceptadores s2 y s3 las expresiones regulares correspondientes a esos estados serán $\omega_{s2}$ y $\omega_{s3}$ por lo que la ER final será: $\omega = \omega s_2 + \omega_{s3}$

**Propiedades de lenguajes regulares**

Lema: La clase de los lenguajes regulares es cerrada bajo la unión, concatenación, estrella de Kleene, complemento e intersección, es decir, si aplicamos estas operaciones a un lenguaje regular el resultado también será un lenguaje regular.

**Correctitud de algoritmos**

Un algoritmo es correcto si para cualquier entrada legal termina y produce la salida deseada.

Correctitud parcial: Si el algoritmo termina entonces produce la salida es la deseada. No garantiza que el algoritmo siempre termine.

Correctitud total: El algoritmo siempre termina y produce la salida deseada.

## Aserciones en la Correctitud de Algoritmos

Las aserciones son afirmaciones sobre el estado de ejecución de un algoritmo en puntos específicos de su ejecución. Se utilizan para probar la correctitud parcial de un algoritmo

**Tipos de Aserciones**

1. Precondiciones**:** Son condiciones que deben ser verdaderas antes de ejecutar un algoritmo, garantizan que la entrada sea válida.

2. Postcondiciones**:** Son condiciones que deben ser verdaderas después de que el algoritmo ha terminado, garantizan que la salida sea la esperada.

3. Invariante: Es una condición que permanece válida, antes, durante y al finalizar el bucle.

**Tripla de Hoare:** {Q} P {R}

Q: Precondición
P: Programa
R: Postcondición

Nota: Usamos la notación P(a/b) para decir que en la expresión P sustituimos cada aparición de a por b.

La sentencia (triple) {Q} if C then P1 else P2 {R} Se deriva de: {Q ∧ C} P1 {R} y {Q ∧ ¬C} P2 {R}

Cuando encontramos o inferimos una invariante para un bucle, debemos probar por inducción que la invariante es valida para cualquier cantidad de iteraciones mayor o igual a cero.

**Funciones: propiedades (repaso)**

Una función f: A → B es:

- Inyectiva si f(x) = f(y) implica que x = y (dos elementos de A no pueden relacionarse con el mismo elemento de B).
- Sobreyectiva si para todo b ∈ B, existe un x ∈ A tal que f(x) = b.
- Biyectiva si es inyectiva y sobreyectiva a la vez.

**Cardinalidad (cantidad de elementos)**

Sean A y B dos conjuntos. Si hay una biyección entre A y B, es decir, cada elemento de A se empareja con un único elemento de B y cada elemento de B tiene un único elemento en A que está relacionado con él. Denotamos eso como $|A| = |B|$. Entonces, decimos que A y B tienen el mismo tamaño / cardinalidad.

Ejemplo: Sean los conjuntos A = {1,2,3} y B = {a,b,c}.

La función f: A → B definida por f(1) = a, f(2) = b, f(3) = c es una biyección. Por lo tanto, decimos que $|A|=|B|= 3$.

**Conjuntos contables**

Un conjunto es contable si es finito o si existe una biyección entre el conjunto y los números naturales. En este caso decimos que es infinito contable.

Si un conjunto A es contable entonces decimos $|A|=|\mathbb{N}|$ Caso contrario, decimos que A es incontable.

**propiedades**

1. Todo subconjunto de $\mathbb{N}$ es contable. Ya sea finito o infinito.

2. un conjunto S es contable si y solo si $|S| \leq |\mathbb{N}|$.

3. Todo subconjunto de un conjunto contable es contable.

4. Toda imagen en un conjunto contable es contable.

**Unión de conjuntos contables**

Si $S_0$ , $S_1$ , …, $S_n$ , … una secuencia de conjuntos contables, la unión $S_0 \cup S_1 \cup \ldots \cup S_n \cup \ldots$ es un conjunto contable.

Nota: El conjunto A* de todas las cadenas definidas sobre el alfabeto finito A es un conjunto contable. $\mathbb{N}x\mathbb{N}$ es un conjunto contable.

**Diagonalización**

Es una técnica, que suele utilizarse para mostrar que un conjunto no es contable.

Supongo que tengo un **alfabeto A** con al menos dos símbolos e intento listar **todas** las posibles secuencias infinitas de elementos de A. Cada secuencia es una fila en una matriz infinita:

| | $0$ | $1$ | $2$ | $\cdots$ | $n$ |
|---|---|---|---|---|---|
| $S_0$ | $a_{00}$ | $a_{01}$ | $a_{02}$ | $\cdots$ | $a_{0n}$ |
| $S_1$ | $a_{10}$ | $a_{11}$ | $a_{12}$ | $\cdots$ | $a_{1n}$ |
| $S_2$ | $a_{20}$ | $a_{21}$ | $a_{22}$ | $\cdots$ | $a_{2n}$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| $S_n$ | $a_{n0}$ | $a_{n1}$ | $a_{n2}$ | $\cdots$ | $a_{nn}$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |

Ahora, construyo una nueva secuencia S a partir de la lista de elementos diagonales (a00 , a11 , a22 , …, ann) y los modifico de manera que el nuevo elemento an sea distinto al elemento ann para cada n.

De este modo, la nueva secuencia S no puede estar en la lista original. Pues difiere de al menos un elemento de cada secuencia Sn en la lista.

Por ejemplo, tomamos dos elementos x, y∈A y definimos: $a_n = \begin{cases} x & \text{if } a_{nn} = y \\ y & \text{if } a_{nn} \neq y. \end{cases}$

**El conjunto F de funciones de N→N es incontable**

Asumo por absurdo que F es contable. Entonces puedo listar todas las funciones de N→N en una secuencia enumerada como: f0,f1,f2,…,fn,..,

Cada función fn puede ser representada por la secuencia de sus valores: (fn(0), fn(1), fn(2),… )

Luego construyo una función g: N→N que no está en la lista usando la técnica de diagonalización, definiendo g(n) = fn(n) + 1 para cada n. Esto garantiza que g(n) ≠ fn(n), por lo que g es distinta de cada fn en al menos un valor.

Por lo tanto, como g no está en la lista de funciones se contradice la hipótesis inicial de que F era contable y se demuestra que el conjunto de todas las funciones de N→N es incontable.

**Álgebra**

Un álgebra es una estructura consistente de uno o más conjuntos (llamados carriers) con una o más operaciones definidas sobre esos conjuntos.

Ejemplo: <ℕ; +, 1, 0> (Carrier: ℕ, Operaciones: + , 1, 0 )

**Expresión algebraica**

Es una combinación válida de símbolos que representa un elemento dentro de un conjunto algebraico. Por ej: 3 + 2 es una expresión algebraica pero 3 + + no lo es.

**Álgebras: concretas y abstractas**

Un álgebra es concreta si los conjuntos (carriers) están bien definidos**,** es decir, sabemos exactamente qué elementos los componen y los operadores tienen reglas claras de cómo actúan sobre los elementos del conjunto.

Un álgebra abstracta solo define las propiedades que los conjuntos y operaciones deben cumplir, sin especificar exactamente qué son.

**operadores binarios**:
Dado un conjunto C y un operador binario •,

- Un elemento z en C se denomina cero si verifica que z • x = x • z = z
- Un elemento u en C se denomina identidad si verifica que u • x = x • u = x
- Un elemento y en C se denomina inverso si verifica que y • x = x • y = u

**Identidad en operación binaria**

Toda operación binaria tiene a lo sumo una identidad

Prueba: Sea * una operación binaria sobre un conjunto S. Para mostrar que * tiene a lo sumo una identidad, asumamos que u y e son identidades para *. Mostraremos que son iguales. Como u y e son identidades, u*x = x*u = x y e*x = x*e = x para todo x en S. Así que se verifica:
e = e*u (pues u es identidad para *)

= u (pues e es identidad para *). CQD

**Álgebras con una operación binaria**

Grupoide: tiene una operación binaria.

Semigrupo: la operación binaria es asociativa.

Monoide: la operación binaria es asociativa con identidad.

Grupo: la operación binaria es asociativa con identidad y todo elemento tiene inverso (unico).

**Inverso único en monoide**
En un monoide, si un elemento tiene inverso, es único.
Prueba: Sea un monoide, mostramos que, si un elemento x en M tiene inverso, el inverso es único.
Si y y z son ambos los inversos x, entonces son iguales.
y = y*u (u es identidad para *)
= y * (x*z) (z es inverso de x)
=(y*x)*z (* es asociativa)
=u*z (y es inverso de x)
=z (u es identidad para *) CQD

**Otras álgebras abstractas**
Un anillo (ring) es un álgebra [A;+, *] donde:
[A;+] es un grupo conmutativo,
[A;* ] es un monoide, y la operación * es distributiva (a izquierda y a derecha) sobre +.
Un cuerpo (field) es un anillo [A;+,* ] donde además se satisface que [A-{0}; *] es un grupo conmutativo, donde 0 es la identidad para [A,+].
**Homomorfismos - Isomorfismos**
Sean [S,*] y [T, +] estructuras algebraicas. Un mapeo f: S→T es un homomorfismo de [S,*] a [T,+] si para todo x, y S, f (x*y) = f (x) + f (y)
Sean [S,*] y [T, +] estructuras algebraicas. Un mapeo f: S→T es un isomorfismo de [S,*] a [T,+] si
1. para todo x, y S, f (x*y) = f (x) + f (y).
2. la función f es una biyección.

**Álgebra de Boole: <B, +, *, ‘ , 0, 1>**

Un conjunto B con dos operaciones binarias **+** (suma lógica o unión) y • (producto lógico o intersección), una operación unaria ’ (complemento), y dos elementos especiales 0 y 1, que cumple:

1. Propiedades de las Operaciones:

Conmutatividad: $x + y = y + x$

$x \bullet y = y \bullet x$

Asociatividad: $(x + y) + z = x + (y + z)$

$(x \bullet y) \bullet z = x \bullet (y \bullet z)$

Distributividad: $x + (y \bullet z) = (x + y) \bullet (x + z)$

$x \bullet (y + z) = (x \bullet y) + (x \bullet z)$

2. Identidades:

- Elemento neutro para + (suma lógica): $x + 0 = x$
- Elemento neutro para • (producto lógico): $x \bullet 1 = x$

3. Complemento:

Cada elemento $x \in B$ tiene un complemento $x’$ tal que:

$x + x’ = 1$

$x \bullet x’ = 0$