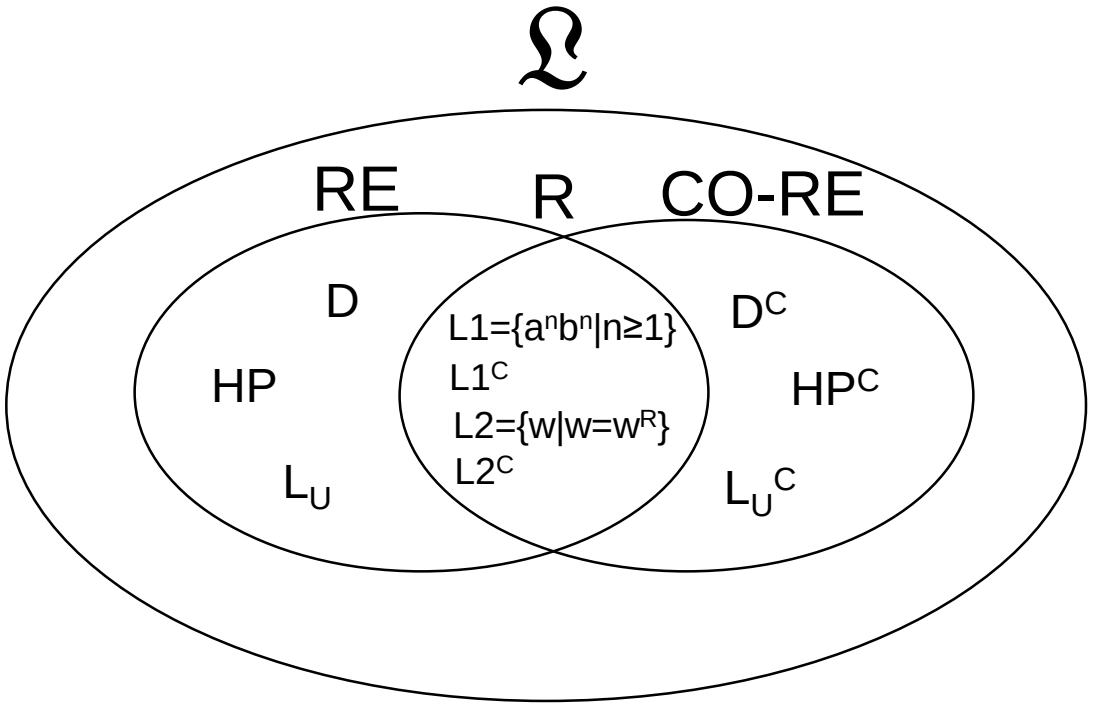
Clase 4. Reducciones de Problemas. MT restringidas.

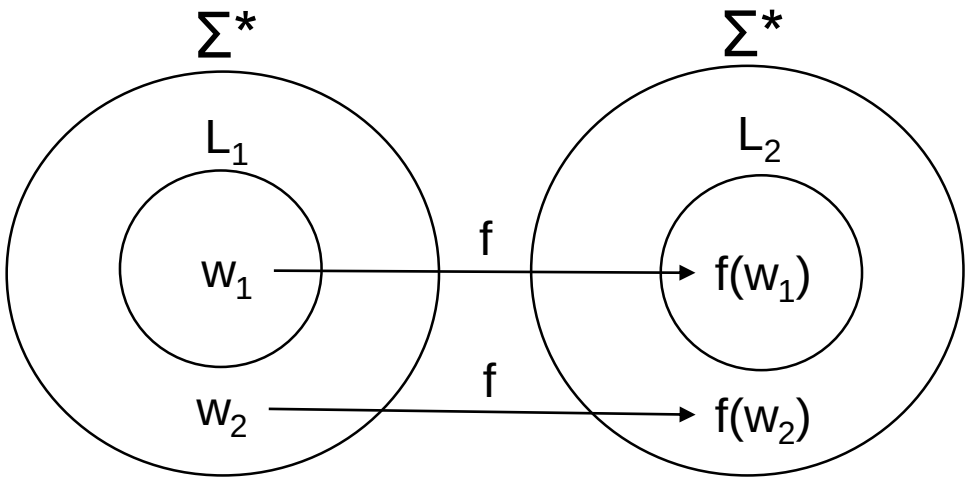
Reducciones de Problemas

- Para probar **pertenencia a las clases R o RE** hemos **construido MT.**
- Para encontrar primeros lenguajes **fuera de R o RE** hemos usado la técnica de **diagonalización.**
- Partiendo de algunos lenguajes ya ubicados fuera de R o RE (ver abajo), podremos seguir poblando dichas áreas del mapa de la computabilidad utilizando una técnica más sencilla que la diagonalización: la **reducción de problemas.**



Definición de reducción

La función f es una reducción del lenguaje (o problema) L_1 al lenguaje (o problema) L_2



*Existe una MT M_f que computa la función f para todo w de Σ**

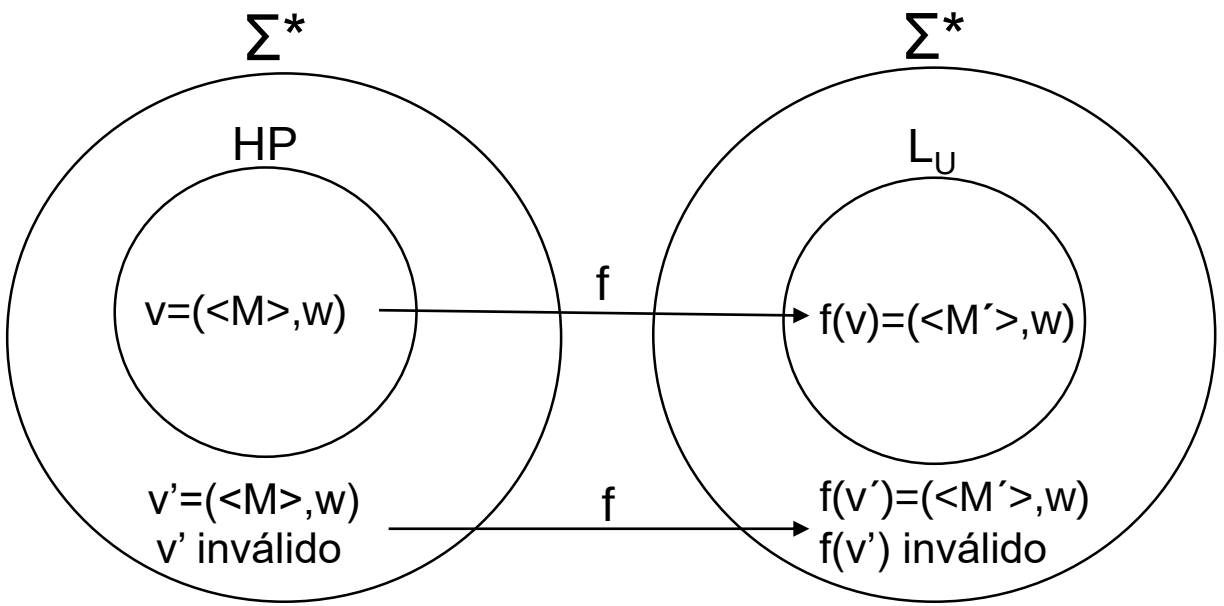
- Sean dos lenguajes L_1 y L_2 (como siempre, incluidos en Σ*).
- Y sea una función f tal que:
  a. f es **total computable**, es decir, existe una MT M_f que la computa para todo w de Σ*.
  b. $w \in L_1 \leftrightarrow f(w) \in L_2$, **es decir que f transforma todo elemento de L_1 en un elemento de L_2, y todo elemento fuera de L_1 en un elemento fuera de L_2.**

  En estas condiciones, **f es una reducción de L_1 en L_2.** Se puede usar la notación: **L_1 α L_2.**

- Anticipándonos: **si existe una MT M_2 que acepta L_2, también existe una MT M_1 que acepta L_1:**
  Dado un input w, la MT M_1 primero ejecuta M_f sobre w para obtener f(w),
  y luego ejecuta M_2 sobre f(w), aceptando o rechazando como M_2.

- **La idea es resolver un problema nuevo (L_1) utilizando la solución de uno conocido (L_2).**

<u>Ejemplo 1</u>. Sean HP = {(<M>, w) | M para sobre w} y L_U = {(<M>, w) | M acepta w}.
Vamos a probar: HP α L_U. La idea general, que se explica abajo, es:



En este caso hay que contemplar también cadenas que no tienen la forma (<M>,w), porque f debe ser total computable (debe estar definida para todo w de Σ).*

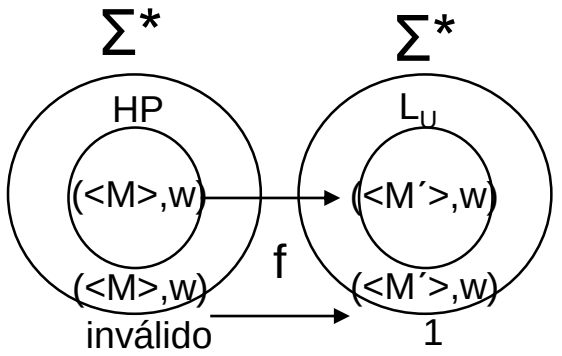
1. Definición de la función de reducción f.

- Para cadenas v de la forma (<M>,w), definimos f((<M>,w)) = (<M´>,w), siendo <M> y <M´> códigos idénticos salvo que los q_R de M se reemplazan en M´ por q_A.
  Así, si M para sobre w, tanto por q_A como por q_R, M´ para por q_A (se pasa de HP a L_U),
  y si M no para sobre w, M' tampoco para sobre w (se pasa de fuera de HP a fuera de L_U).
- Para cadenas v inválidas, o sea sin la forma (<M>,w), f devuelve también una cadena inválida, p.ej. 1, asegurando que f vaya de fuera de HP a fuera de L_U.

2. Prueba de que f es una función total computable.

Claramente existe una MT M_f que computa totalmente f. Para todo input v:
M_f primero analiza sintácticamente el input v. Luego:

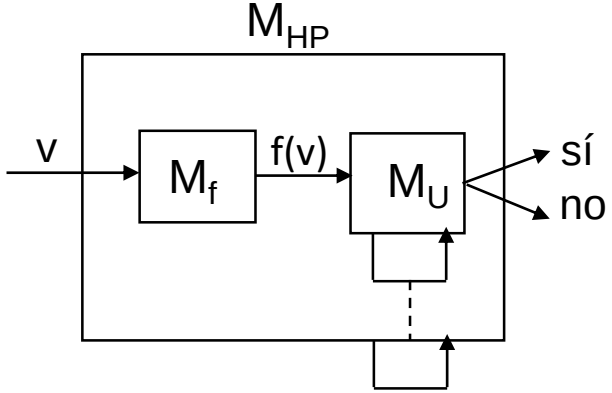
- Si v no es un par válido (<M>,w), directamente genera la cadena 1.
- Si v = (<M>,w), copia <M> pero cambiando los q_R por q_A, y luego copia w.



3. Prueba de que (<M>,w) ∈ HP ↔ f((<M>,w)) ∈ L_U.

(<M>,w) ∈ HP ↔ por def: M para sobre w ↔ por const: M' acepta w ↔ por def: (<M'>,w) ∈ L_U.
Otra manera de probar esto es: (a) partir de (<M>,w) ∈ HP y llegar a (<M'>,w) ∈ L_U, y (b) partir de (<M>,w) o inválido ∉ HP y llegar a (<M'>,w) o inválido ∉ L_U; la forma del ↔ es más sencilla.

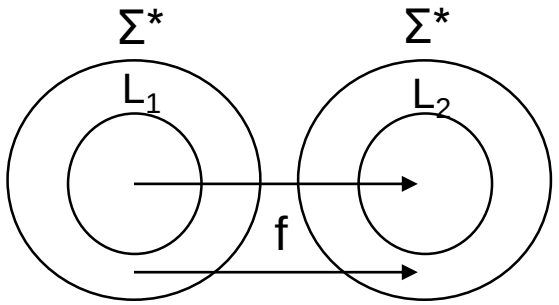
Resumiendo: hemos construido una MT M_f que transforma todo elemento de HP en un elemento de L_U,
y todo elemento fuera de HP en un elemento fuera de L_U.
De esta manera, a partir de una MT M_U que acepta L_U, se puede construir una MT M_{HP} que acepta HP, obtenida por la composición de las MT M_f y M_U.



v ∈ HP ⟶ f(v) ∈ L_U
v ∉ HP ⟶ f(v) ∉ L_U

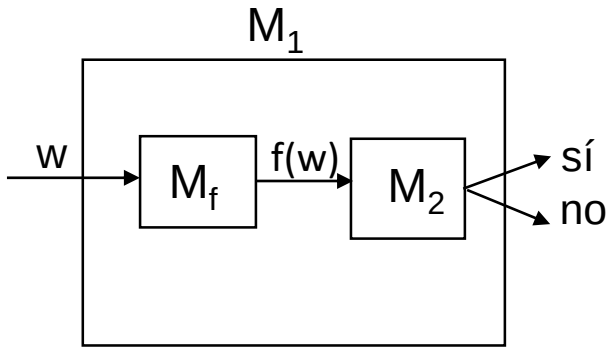
Veamos cómo utilizar las reducciones para poblar el mapa de la computabilidad

Teorema.
(a) Si L_1 α L_2, entonces $L_2 \in R \longrightarrow L_1 \in R$
(b) Si L_1 α L_2, entonces $L_2 \in RE \longrightarrow L_1 \in RE$



Prueba. La idea general se comentó en el ejemplo anterior. Si existe una MT M_f que reduce L_1 a L_2, y existe una MT M_2 que acepta L_2, entonces también existe una MT M_1 que acepta L_1: la que ejecuta primero M_f sobre el input w y luego M_2 sobre el output f(w) de M_f:

Parte (a): En este caso, la MT M_2 para siempre, y por lo tanto también lo hace la MT M_1:



La prueba formal es la siguiente (de la parte (a), la parte (b) es similar):

- Supongamos que L_1 α L_2 y $L_2 \in R$. Veamos que se cumple $L_1 \in R$.

- Por la hipótesis,
  - ✓ existe una MT M_f que computa f que cumple: $w \in L_1 \leftrightarrow f(w) \in L_2$,
  - ✓ y existe una MT M_2 que acepta L_2 y para siempre.

- Así, también existe una MT M_1 que acepta L_1 y para siempre:
  - ✓ dado un input w,
  - ✓ M_1 primero ejecuta M_f sobre w para obtener f(w),
  - ✓ luego ejecuta M_2 sobre f(w),
  - ✓ y acepta si y sólo si M_2 acepta.

- **M_1 para siempre** porque M_f y M_2 paran siempre.
- **$L_1 = L(M_1)$** :
  $w \in L_1 \leftrightarrow M_f$ a partir de w computa $f(w) \in L_2 \leftrightarrow M_2$ acepta $f(w) \leftrightarrow M_1$ acepta $w \leftrightarrow w \in L(M_1)$

La parte (b) queda como ejercicio.

- En el ejemplo anterior vimos que: $HP\ \alpha\ L_U$
  El teorema anterior indica que: $L_1\ \alpha\ L_2$ y $L_2 \in RE \longrightarrow L_1 \in RE$
  Entonces, como $L_U \in RE$, se cumple: **$HP \in RE$**

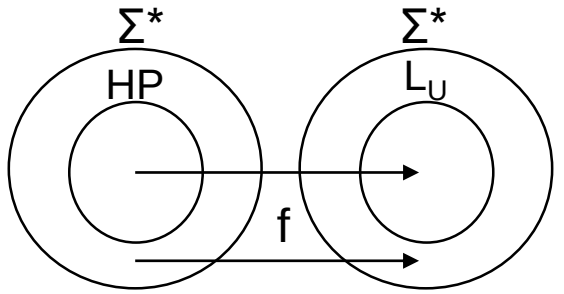
- Hemos encontrado **una manera alternativa** para probar que $HP \in RE$ (la manera que vimos antes era directamente construyendo una MT que acepte HP).
  Una variante del teorema nos permitirá probar **no pertenencia** a una clase (propiedad que no se puede probar constructivamente, **de aquí la importancia de las reducciones, que son mucho más sencillas que usar la técnica de diagonalización**).

<u>Corolario</u> (contrarrecíproco del teorema anterior).

a) Si $L_1\ \alpha\ L_2$ entonces $L_1 \notin R \longrightarrow L_2 \notin R$ (en lugar de $L_2 \in R \longrightarrow L_1 \in R$)

b) Si $L_1\ \alpha\ L_2$ entonces $L_1 \notin RE \longrightarrow L_2 \notin RE$ (en lugar de $L_2 \in RE \longrightarrow L_1 \in RE$)

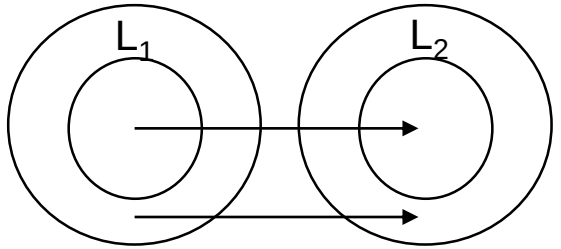
- Por el ejemplo anterior y el corolario, entonces, como $HP\ \alpha\ L_U$ y $HP \notin R$, entonces **$L_U \notin R$.**



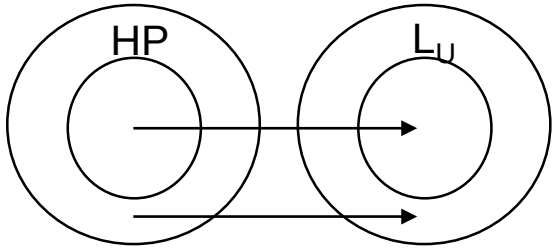
- Probamos **no pertenencia a una clase,** en este caso $L_U \notin R$, sin recurrir a la **diagonalización.**
  Una forma útil de leer lo anterior es la siguiente: **si L_U fuera recursivo también lo sería HP, pero como HP no es recursivo entonces tampoco lo es L_U.**

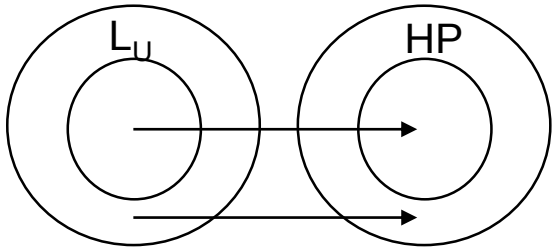
- Profundizando un poco más acerca de lo que significa que exista una reducción de L_1 a L_2 :

  - ✓ Hemos probado que si L_1 α L_2 y $L_1 \notin R$, entonces $L_2 \notin R$,
    y que si L_1 α L_2 y $L_1 \notin RE$, entonces $L_2 \notin RE$.
  - ✓ En otras palabras, si se puede reducir L_1 a L_2,
    entonces **L_2 es "tan o más difícil" que L_1** (en términos de computabilidad).
    Concretamente, si hay una reducción de L_1 a L_2,
    si $L_1 \notin R$ también $L_2 \notin R$, y si $L_1 \notin RE$ también $L_2 \notin RE$.
  - ✓ Otra forma de leer que L_2 es "tan o más difícil" que L_1 es:
    si se puede resolver L_2 entonces se puede resolver L_1.



- Volviendo al ejemplo anterior, entonces:
  - ✓ L_U es "tan o más difícil" que HP.
  - ✓ También se prueba que L_U α HP,
    y así también se cumple que HP es "tan o más difícil" que L_U.
  - ✓ Por lo tanto, **L_U y HP tienen dificultad equivalente en el marco de la computabilidad.**

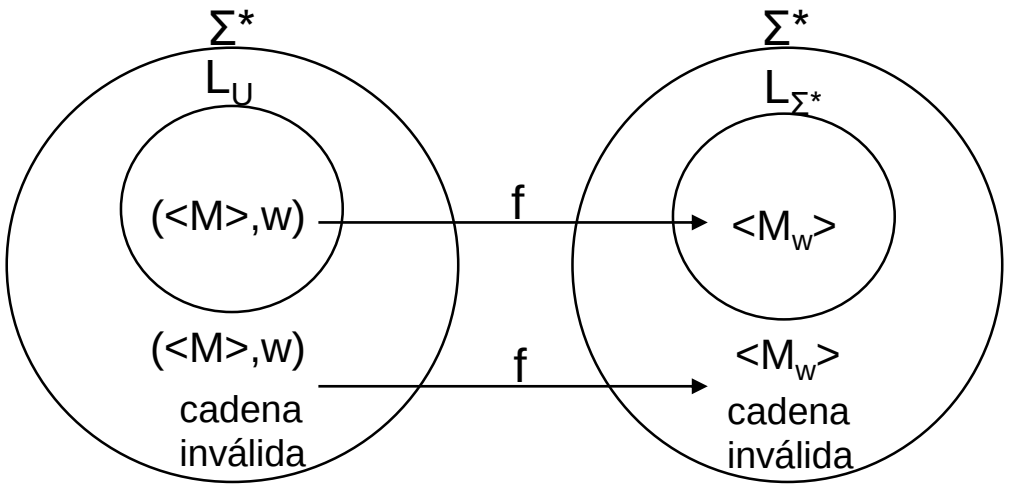




Ejemplo 2. Sea el problema: dada una MT M, ¿M acepta todas las cadenas de Σ*?

El lenguaje que representa el problema es: $L_{\Sigma^*} = \{<M> \mid L(M) = \Sigma^*\}$.
Vamos a probar por medio de una reducción de problemas que **$L_{\Sigma^*} \notin R$.**
Ejercicio: Intuitivamente, ¿puede ser $L_{\Sigma^*} \in R$? ¿Y $L_{\Sigma^*} \in RE$?

Debemos encontrar una reducción de la forma **L_1 α L_{Σ^*}**, de modo tal que **$L_1 \notin R$** (L_{Σ^*} debe ser "tan o más difícil" que L_1, y así probaremos que $L_{\Sigma^*} \notin R$). **Elegimos L_U α L_{Σ^*}.**



1. Definición de la función de reducción f.

- Para inputs válidos (<M>,w) se define: f((<M>,w)) = $<M_w>$.
  M_w es una MT que reemplaza su input por w, ejecuta M sobre w y acepta sii M acepta. Notar:
  - si **M acepta w**, como M_w siempre reemplaza su input por w entonces **acepta siempre**.
    Por lo tanto $L(M_w) = \Sigma^*$, y así $<M_w> \in L_{\Sigma^*}$, siendo $(<M>,w) \in L_U$.
  - si **M rechaza w**, como M_w siempre reemplaza su input por w entonces **rechaza siempre**.
    Por lo tanto $L(M_w) = \varnothing$, y así $<M_w> \notin L_{\Sigma^*}$, siendo $(<M>,w) \notin L_U$.
- Para inputs sin la forma (<M>,w), se define como antes el output inválido 1 (nuevamente la función de reducción f va de un elemento de fuera de L_U a un elemento de fuera de L_{Σ^*}).

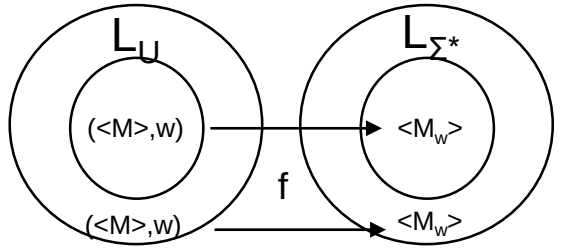
2. Prueba de que f es una función total computable.

Claramente existe una MT M_f que computa totalmente f. Para todo input v:

- M_f primero analiza sintácticamente el input v. Luego:
- Si v no es un par válido (<M>,w) genera la cadena 1.
- En caso contrario genera $<M_w>$, agregando al código <M> un fragmento inicial que borra su input y lo reemplaza por w.

3. Prueba de que (<M>, w) ∈ L_U ↔ f((<M>, w)) ∈ L_{Σ^*}.

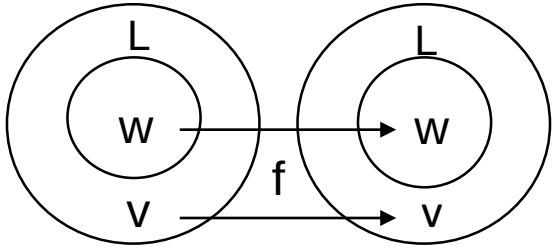
(<M>, w) ∈ L_U ↔ M acepta w ↔ $L(M_w) = \Sigma^*$ ↔ $<M_w> \in L_{\Sigma^*}$ ↔ f((<M>, w)) ∈ L_{Σ^*}



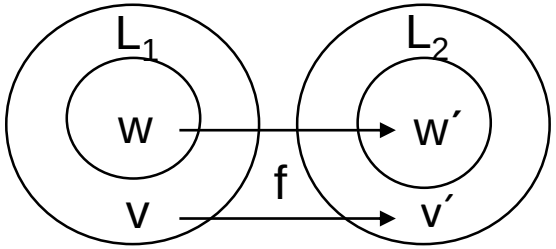
- Hemos probado entonces que $L_{\Sigma^*} = \{<M> \mid L(M) = \Sigma^*\} \notin R$.
- Intuitivamente pareciera incluso que $\mathbf{L_{\Sigma^*} \notin RE}$ (ejercicio anterior): si $L_{\Sigma^*} \in RE$, existe una MT M_{Σ^*} que acepta <M> sii M acepta todas las cadenas, lo que significa que M_{Σ^*} debe aceptar <M> después de comprobar que M acepta las **infinitas** cadenas de Σ*, lo que no parece razonable.
- Efectivamente se cumple que $\mathbf{L_{\Sigma^*} \notin RE}$. Para probarlo formalmente por medio de una reducción de problemas deberemos encontrar una reducción de la forma $\mathbf{L_1}$ **α** $\mathbf{L_{\Sigma^*}}$, pero ahora necesitaremos que $L_1 \notin RE$ (L_{Σ^*} debe ser "tan o más difícil" que L_1), y así probaremos que $L_{\Sigma^*} \notin RE$. No nos sirve la reducción anterior L_U α L_{Σ^*} porque $L_U \in RE$.
- En slides siguientes hacemos más referencias a L_{Σ^*} y su complemento.

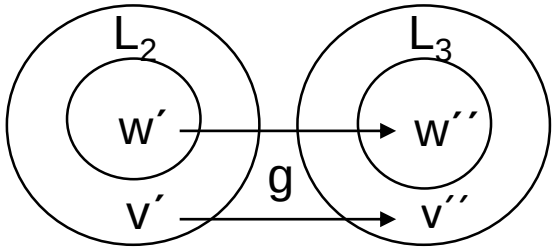
Algunas propiedades de las reducciones

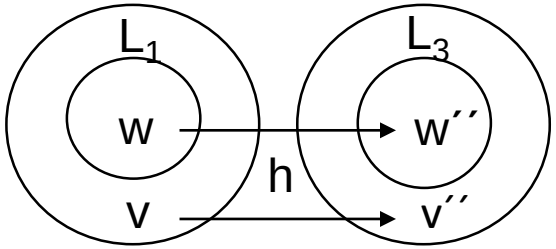
- **Reflexividad.** Para todo lenguaje L se cumple L α L.
  La función de reducción es la función identidad.



- **Transitividad.** Si L_1 α L_2 y L_2 α L_3, entonces L_1 α L_3.

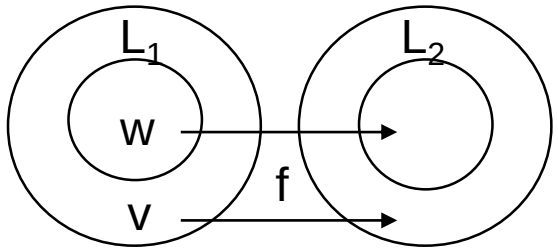


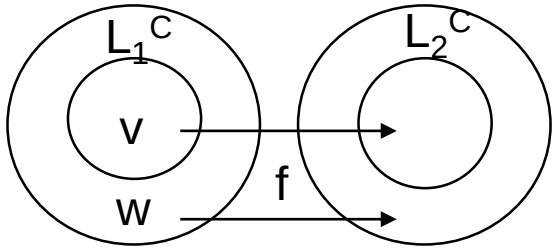




Para todo x, h(x) = g(f(x)), es decir que h es la composición de f con g. *Probarlo formalmente.*

- **Otra propiedad útil es:** L_1 α $L_2 \leftrightarrow$ L_1^C α L_2^C.
  Es la misma función de reducción en los 2 casos.

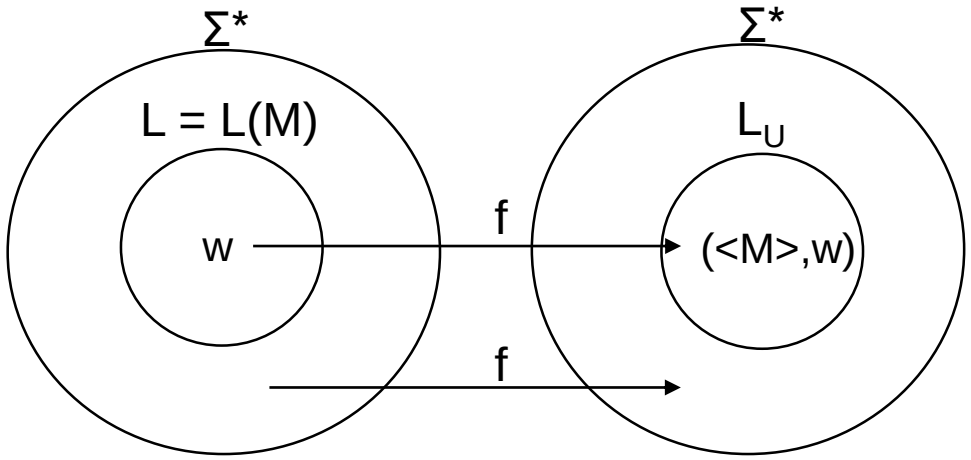




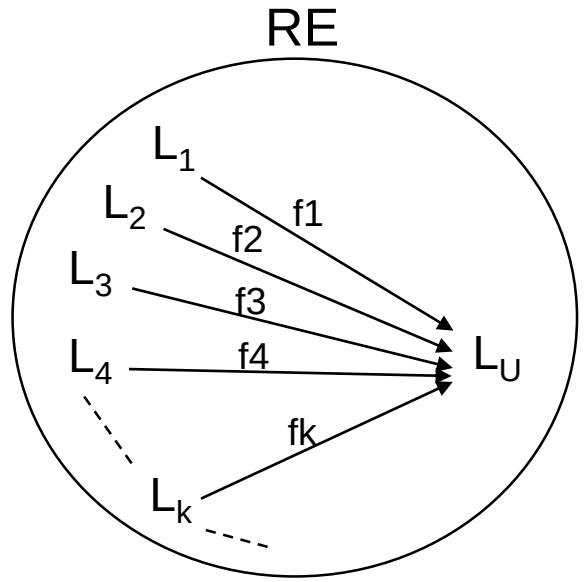
- **¿Qué sucede con la simetría?** ¿L_1 α L_2 implica L_2 α L_1? **No se cumple**:

  ✓ Sea cualquier L ∈ R. **No puede ser L_U α L** (L es "tan o más difícil" que L_U, y $L_U \notin$ R).

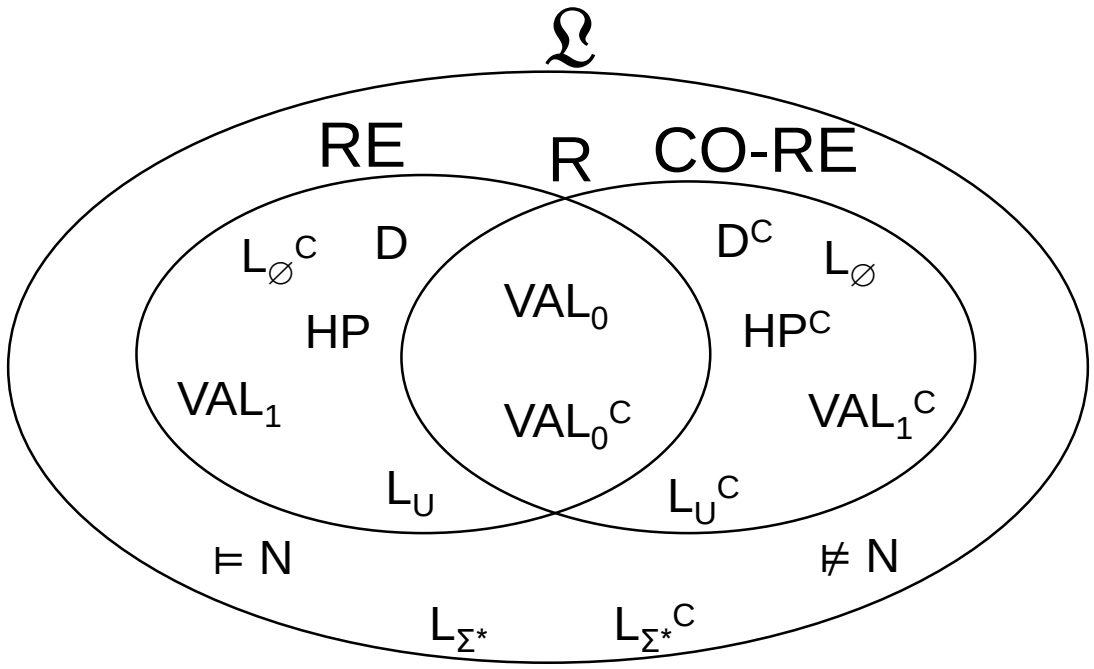
  ✓ Sin embargo **se cumple L α L_U**:
  Si M es una MT que acepta L, entonces f(w) = (<M>,w) es una reducción de L a L_U. Claramente f es total computable y se cumple: w ∈ L ↔ (<M>,w) ∈ L_U.



- Notar que lo anterior **vale para cualquier lenguaje L ∈ RE**.
- Por lo tanto, **todos los lenguajes de RE se reducen a L_U.**
- Así, **L_U es "tan o más difícil"** que cualquier lenguaje de RE.
- **Lo mismo se demuestra para HP** (algo dijimos en la clase 3).
- Ambos lenguajes se conocen como **RE-completos**, lo que significa que todo lenguaje de RE se reduce a ellos. En otras palabras, teniendo una solución para L_U o para HP, se tiene también una solución para todo lenguaje de RE.



Lo mismo vale para HP



- Mostramos en este **mapa final de la computabilidad** distintos lenguajes, algunos de los cuales han sido referidos a lo largo de las últimas clases. En particular introducimos ejemplares de la región más amplia de la jerarquía, la que está afuera de RE U CO-RE (en un ejemplo que se muestra más adelante se indica que efectivamente tanto L_{Σ^*} como $L_{\Sigma^*}{}^C$ están allí).

✓ VAL_0 es el lenguaje de las fórmulas válidas de la lógica proposicional (orden 0).
✓ VAL_1 es el lenguaje de las fórmulas válidas de la lógica de predicados (orden 1).
Turing y Church probaron por separado en 1936 que VAL_1 no es recusivo.
✓ ⊨ N es el lenguaje de los enunciados verdaderos de la aritmética.
Gödel probó en 1931 que ⊨ N no es siquiera recursivamente numerable (tampoco ⊭ N). En otras palabras, no se pueden enumerar los enunciados verdaderos de la aritmética, ni los falsos.

Máquinas de Turing restringidas

Autómatas finitos o AF (McCulloch-Pitts, 1943)

- Hay **una sola cinta.**
- La cinta es de **sólo lectura.**
- El cabezal se mueve **sólo a la derecha.**
- Hay **uno o más estados de aceptación** (se denominan **estados finales**).
- Cuando el cabezal llega al blanco de la derecha del input, el AF para (**acepta sii el estado es final**).
- Formalmente: **M = (Q, Σ, δ, q_0, F)**, donde F es el conjunto de estados finales.

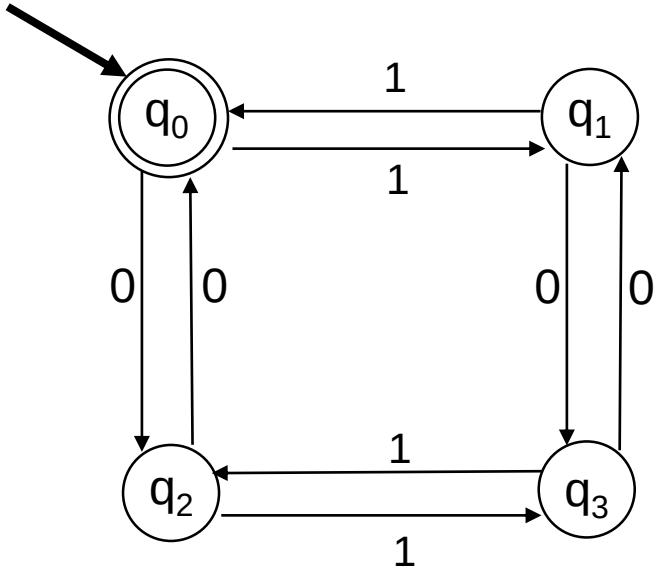
- El AF constituye un tipo de algoritmo ampliamente utilizado. Por ejemplo:
  - ✓ En los **compiladores**, para la detección de patrones (p.ej. la palabra IF, la variable x, etc).
  - ✓ Por la misma razón, en las **inspecciones de código fuente.**
  - ✓ En la **verificación automática de programas** (se utilizan AF para relacionar por *matching* las trazas de las computaciones de los programas con fórmulas de la lógica temporal que se supone que se cumplen en las computaciones, lo que se conoce como *model checking*).

Ejemplo. Sea el AF M = <Q, Σ, δ, q_0, F> siguiente:

Q = {q_0, q_1, q_2, q_3}, Σ = {0, 1}, q_0 = q_0, F = {q_0}, y la función δ es:

1. δ(q_0, 1) = q_1
2. δ(q_1, 1) = q_0
3. δ(q_1, 0) = q_3
4. δ(q_3, 0) = q_1
5. δ(q_3, 1) = q_2
6. δ(q_2, 1) = q_3
7. δ(q_2, 0) = q_0
8. δ(q_0, 0) = q_2

Este diagrama es una representación gráfica habitual de los AF.
La ejecución arranca desde donde indica la flecha.
Los estados con doble contorno son los estados finales.



Se comprueba que el AF descripto acepta todas las cadenas de 1 y 0 con una cantidad par de 1 y una cantidad par de 0.

- Notar que los AF **siempre paran.**

- Por no tener memoria (sólo una cinta de sólo lectura), salvo la que brindan los estados de algún modo, y otras limitaciones, los AF pueden aceptar un **tipo limitado de cadenas** (p.ej., no pueden aceptar cadenas con igual cantidad de 1 y 0, expresiones con paréntesis balanceados, etc).

- Los lenguajes que aceptan los AF se llaman **regulares** o de **tipo 3**.

- Se prueba que la clase de los lenguajes regulares es **cerrada con respecto a las operaciones, entre otras, de unión, intersección, complemento y concatenación** (diferencia con los lenguajes recursivamente numerables).

- A diferencia de las MT generales, con AF problemas como:

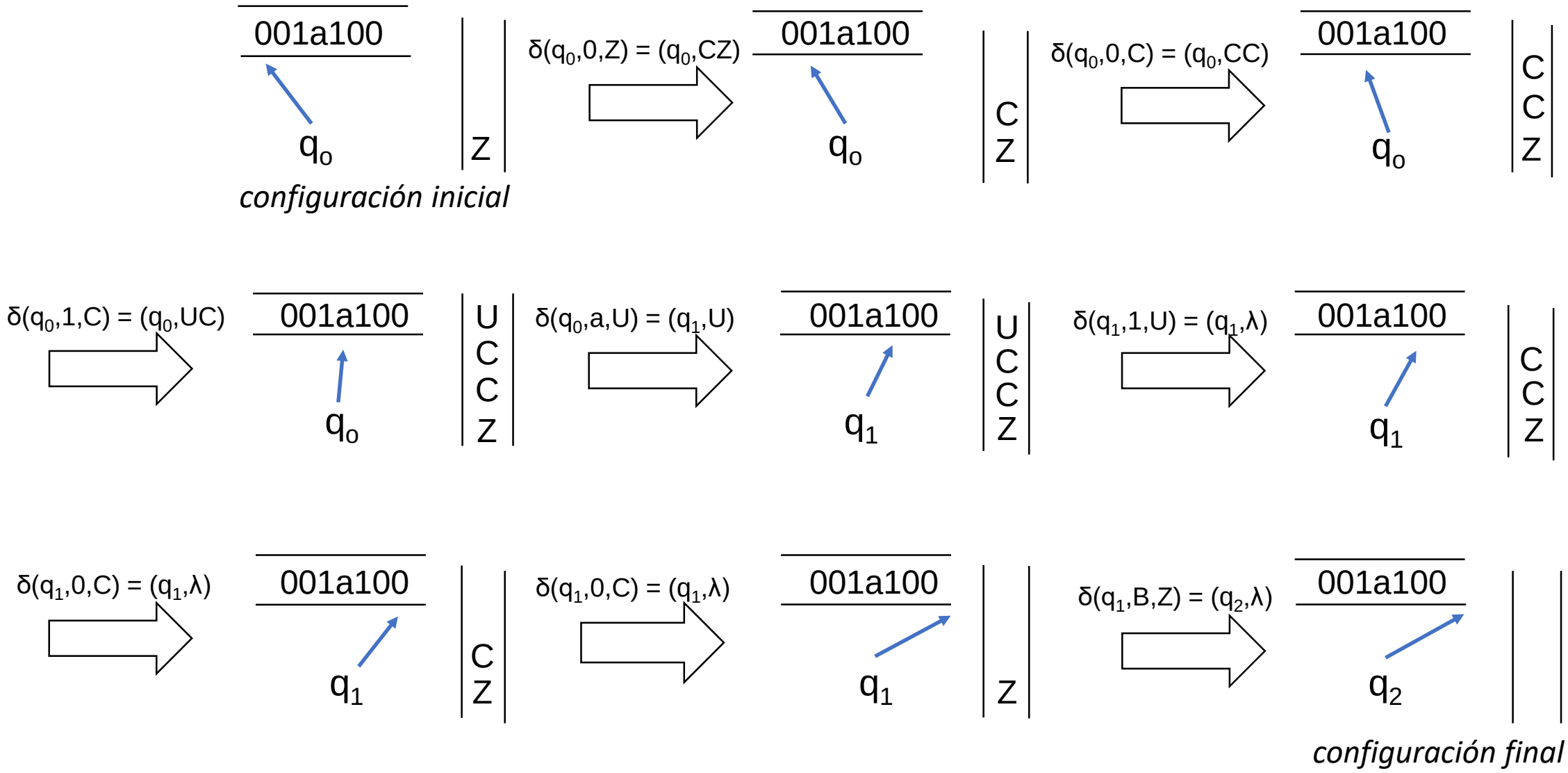
  - ✓ ¿$w \in L(M)$?
  - ✓ ¿$L(M) = \emptyset$?
  - ✓ ¿$L(M) = \Sigma^*$?
  - ✓ ¿$L(M) = L(M')$?

  son decidibles.

Autómatas con pila (AP)

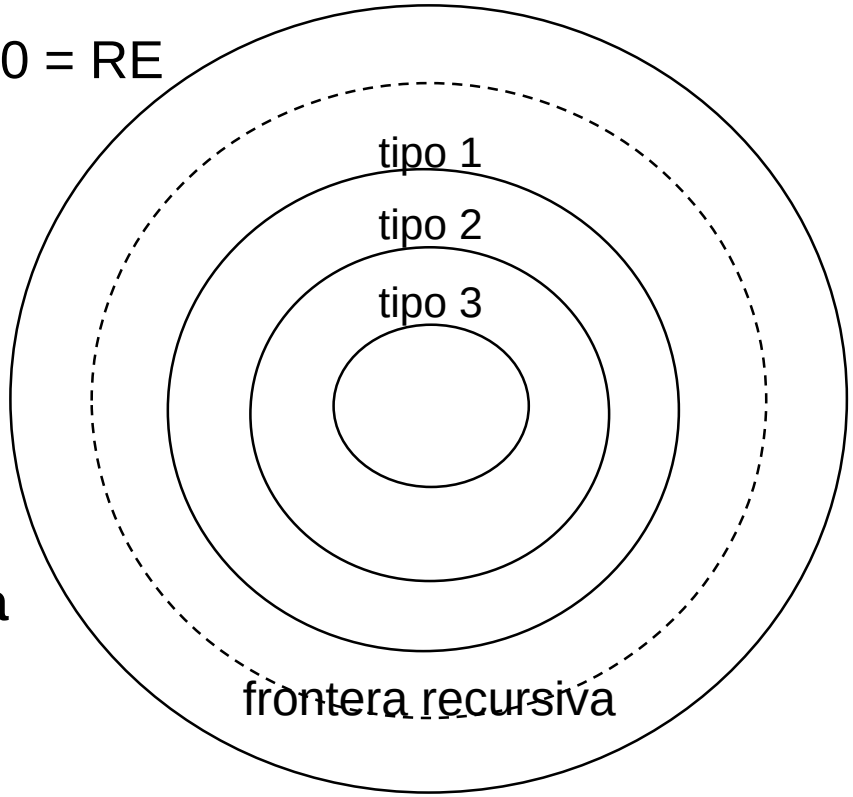
- Este tipo de autómata tiene **más potencia computacional** que el AF.
- Básicamente agrega a la cinta de input de sólo lectura **una segunda cinta en la que puede escribir, y que se comporta como una pila (*stack*)** - limitación que no tienen las cintas de las MT generales -.
- En todo momento se puede leer **de la cinta de input y de la pila.**
- En la cinta de input se avanza siempre **a la derecha** hasta el blanco a la derecha del input.
- En la pila se puede **apilar o desapilar.**
- Formalmente: **$M = (Q, \Sigma, \Gamma, \delta, q_o, Z)$,** tal que Z denota el primer símbolo de la pila.
- El autómata con pila acepta sii la pila al final queda **vacía.**
- Problemas típicos que resuelve un AP es el **análisis sintáctico** y la **evaluación de expresiones** (los compiladores de todos los lenguajes de programación de la industria lo incluyen).

Ejemplo. Un AP puede reconocer por ejemplo las cadenas waw^C, tales que w tiene 0 y 1 y w^C es la cadena inversa de w. Veamos a continuación una ejecución para el input **001a100**:



- Los AP, como los AF, **siempre paran.**
- Los lenguajes que aceptan se llaman **libres de contexto** o de **tipo 2.**
- Se prueba que la clase de los lenguajes libres de contexto es **cerrada con respecto a las operaciones de unión y concatenación, y no con respecto a la intersección ni complemento** (diferencia con los lenguajes regulares y los lenguajes de R y RE).
- A diferencia de las MT generales, con AP problemas como ¿$w \in L(M)$? y ¿$L(M) = \varnothing$? **son decidibles.**
- Al igual que con las MT generales, con los AP el problema ¿$L(M) = \Sigma^*$? **no es decidible** (lo que determina una diferencia con los lenguajes regulares, e identifica una mayor dificultad computacional de dicho lenguaje con respecto a los otros mencionados).

- La **Jerarquía de Chomsky (1956)** clasifica los lenguajes en lenguajes de tipo 3 (**regulares**), de tipo 2 (**libres de contexto**), de tipo 1 (**sensibles al contexto**) y de tipo 0 (son los lenguajes **recursivamente numerables**).
- La clase de lenguajes de tipo i incluye a la de tipo i + 1.
- Los lenguajes de tipo 1, 2 y 3 están incluidos en la clase R.
- **La idea subyacente es utilizar el autómata más adecuado para cada tipo de problema.**



Gramáticas

- Al igual que las MT, las gramáticas constituyen una **representación finita de los lenguajes.**
- P.ej., es muy popular la forma **BNF (Backus-Naur)** usada para los lenguajes de programación.

- Una gramática es una 4-tupla **$G = (V_N, V_T, P, S)$,** tal que:
  - ✓ V_N es un conjunto de símbolos *no terminales.*
  - ✓ V_T es un conjunto de símbolos *terminales.*
  - ✓ P es un conjunto de *reglas de reescritura* o *producciones*, con la forma $\alpha \rightarrow \beta$.
  - ✓ S es el *símbolo no terminal inicial* o *axioma* de G.

- P.ej., el lenguaje de las cadenas $a^n b^n$, con $n \geq 1$, se puede representar (y **generar**) con la gramática G = ({S}, {a, b}, {1, 2}, S), siendo las reglas 1 y 2 las siguientes:
  1. $S \rightarrow aSb$
  2. $S \rightarrow ab$

  Las cadenas se obtienen de aplicar cero o más veces la regla 1 y luego una vez la regla 2. Por ejemplo, la cadena aaabbb pertenece al lenguaje generado por G, o L(G), aplicando dos veces (1) y una vez (2).

- Por la forma de las reglas, las gramáticas también se clasifican según la **Jerarquía de Chomsky**: tipo 3, 2, 1 y 0. En efecto, un lenguaje de tipo 3 (o regular) puede ser **generado** por una gramática de tipo 3 y **aceptado** por un autómata finito; un lenguaje de tipo 2 (o libre de contexto) puede ser **generado** por una gramática de tipo 2 y **aceptado** por un autómata con pila; etc.

Ejemplo. Volvemos al problema: **¿La MT M acepta todas las cadenas de Σ*?**

- Vimos que el lenguaje que lo representa es $L_{\Sigma^*} = \{<M> \mid L(M) = \Sigma^*\}$.
- Probamos $\mathbf{L_{\Sigma^*} \notin R}$, e indicamos que en realidad también se cumple $\mathbf{L_{\Sigma^*} \notin RE}$.
- Dijimos que para probarlo tenemos que encontrar un lenguaje $L \notin RE$ tal que $L \alpha L_{\Sigma^*}$
- Una reducción posible es $\mathbf{L_U^C \alpha L_{\Sigma^*}}$. La prueba se puede encontrar en el libro de la materia.

- La prueba de que $L_{\Sigma^*} \notin R$ la hicimos recurriendo a la reducción $L_U \alpha L_{\Sigma^*}$
- Notar que, entonces, por una propiedad de las reducciones que analizamos en esta misma clase, también se cumple $\mathbf{L_U^C \alpha L_\Sigma^C}$.
- Por lo tanto, como $L_U^C \notin RE$, por el teorema estudiado llegamos a que también $\mathbf{L_{\Sigma^*}^C \notin RE.}$

- De esta manera, hemos identificado formalmente **un caso de lenguaje tal que él y su complemento no están en RE U CO-RE**, el lenguaje L_{Σ^*}.
- Por su ubicación en el mapa de la computabilidad, L_{Σ^*} (y lo mismo $L_{\Sigma^*}^C$) es más “difícil” que, p.ej., HP, HP^C, L_U y L_U^C. De hecho, recurriendo a un comentario anterior, la reducción $L_U \alpha L_{\Sigma^*}$ indica que L_{Σ^*} es “tan o más difícil” que L_U, y la reducción $L_U^C \alpha L_\Sigma^C$, que L_Σ^C es “tan o más difícil” que L_U^C**.**

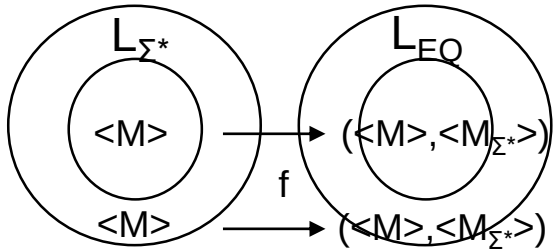
Ejemplo. Sea el problema: **Dadas dos MT M_1 y M_2, ¿M_1 es equivalente a M_2?**

El lenguaje que representa el problema es $L_{EQ} = \{(<M_1>,<M_2>) \mid L(M_1) = L(M_2)\}$.

Intuitivamente, no parece que $L_{EQ} \in RE$. Si lo fuera, existiría una MT M_{EQ} que acepta $(<M_1>,<M_2>)$ después de comprobar que M_1 y M_2 aceptan y rechazan exactamente las mismas cadenas del conjunto infinito Σ^*, lo que no parece razonable.

Efectivamente, se cumple **$L_{EQ} \notin RE$ (el problema de si 2 programas son equivalentes no es computable).** Se puede hacer **L_{Σ^*} α L_{EQ}** (como $L_{\Sigma^*} \notin RE$, entonces $L_{EQ} \notin RE$):

1. Función de reducción f.
Para inputs <M> se define $f(<M>) = (<M>,<M_{\Sigma^*}>)$, con $L(M_{\Sigma^*}) = \Sigma^*$.
Y para inputs sin la forma <M> se define el output 1.

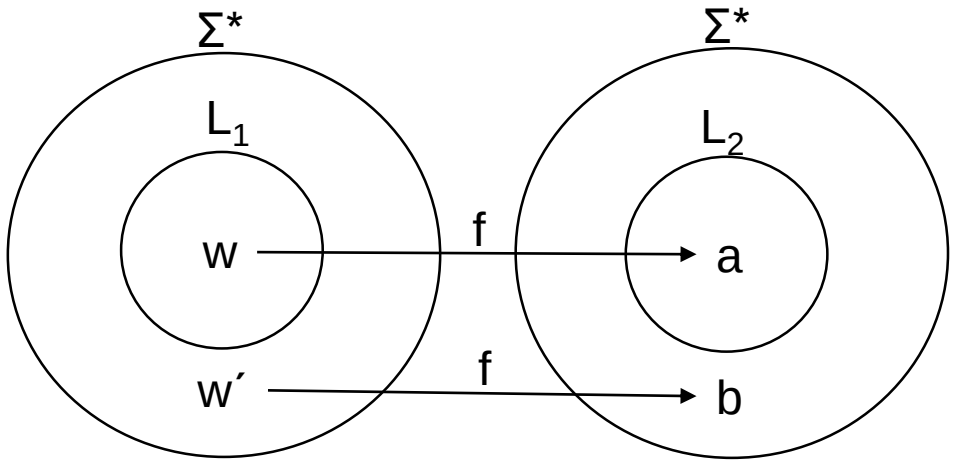


2. f es total computable.
Es totalmente computable chequear la sintaxis del input y generar las cadenas $(<M>,<M_{\Sigma^*}>)$ y 1.

3. $<M> \in L_{\Sigma^*} \leftrightarrow f(<M>) \in L_{EQ}$.
$<M> \in L_{\Sigma^*} \leftrightarrow L(M) = \Sigma^* \leftrightarrow L(M) = L(M_{\Sigma^*}) \leftrightarrow (<M>,<M_{\Sigma^*}>) \in L_{EQ} \leftrightarrow f(<M>) \in L_{EQ}$

<u>Ejemplo</u>. Veamos que a diferencia de lo que sucede en la clase RE, en la clase R se cumple, sin considerar los lenguajes especiales Σ^* y $\varnothing$, que cualquier lenguaje L_1 se puede reducir a cualquier lenguaje L_2.
En otras palabras, todos los lenguajes de R tienen "la misma dificultad". La prueba es sencilla:

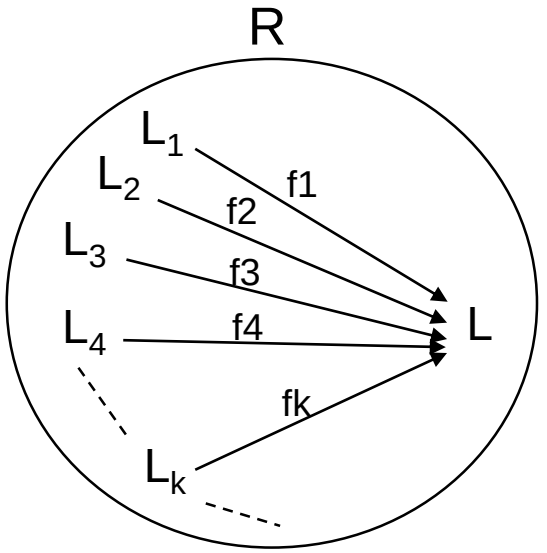
- Sean L_1 y L_2 distintos de Σ^* y $\varnothing$.
- Sean $a \in L_2$ y $b \notin L_2$.
- Sean M_1 y M_2 tales que deciden L_1 y L_2, respectivamente.



1. Función de reducción f.
$f(w) = a$ si $w \in L_1$.
$f(w) = b$ si $w \notin L_1$.

2. f es total computable.
Dado w, la MT M_f que computa f ejecuta M_1 sobre w, si acepta imprime a y si rechaza imprime b (la MT M_1 para siempre porque L_1 es recursivo).

3. $w \in L_1 \leftrightarrow f(w) \in L_2$.
$w \in L_1 \leftrightarrow f(w) = a \leftrightarrow f(w) \in L_2$



Todo L de R es R-completo