Sintaxis y Semántica de los Lenguajes

Resumen Segundo parcial

TIPOS DE ANÁLISIS SINTÁCTICOS

Al derivar una secuencia de tokens, si existe más de un noterminal en una cadena de derivación, debemos elegir cuál es el próximo noterminal que se va a expandir. Para ello se utilizan dos tipos de derivaciones que determinan con precisión cuál será el noterminal a tratar: derivación a izquierda y derivación a derecha.

Cada tipo de derivación determina un tipo de análisis sintáctico:

- **Análisis Sintáctico Descendente (LL1):** produce una derivación por izquierda que comienza en el axioma y finaliza en los terminales
- **Análisis Sintáctico Ascendente:** produce una derivación por derecha, pero de una manera especial, lo hace a la inversa, es decir, la última producción aplicada en la derivación a derecha es la primera en ser “descubierta” (se realiza el proceso que se hace con las tablas de evaluación)

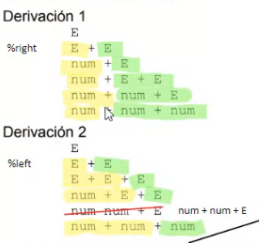
GRAMÁTICAS AMBIGUAS

Una **GIC** que permite **dos o más derivaciones a izquierda o derecha** para obtener la misma cadena de terminales se denomina **ambigua**. Las mismas deben ser evitadas para la implementación computacional, ya que deseamos construir analizadores sintácticos que funcionen de manera determinística.

Una forma de solucionar este problema es reformulando la GIC para encontrar una equivalente que no presente ambigüedad en la derivación

GIC con dos producciones: *E -> E+E*

E -> num



GRAMÁTICAS LL Y LR

- **Análisis Sintáctico LL:** consiste en analizar el flujo de tokens de **izquierda a derecha** por medio de una **derivación por izquierda**. Una **gramática LL** es una GIC que puede ser utilizada en un Análisis Sintáctico LL.

  Una **gramática LL** especial es la llamada **LL(1)**, que puede ser utilizada por un Parser LL, con **un solo símbolo de preanálisis**. Es decir, si un noterminal tiene varias producciones, el Parser puede decidir cuál lado derecho debe aplicar con solo conocer el próximo token del flujo de entrada.

- **Análisis Sintáctico LR:** consiste en analizar el flujo de tokens de **izquierda a derecha** pero por medio de una **derivación a derecha** utilizada a la inversa.

Obtención de gramáticas LL(1)

Para ser útiles y eficientes, los **Análisis Sintácticos Descendentes** deben basarse en **GICs LL(1)**. Sin embargo, estas gramáticas presentan **dos principales problemas**:

1. **No puede tener un noterminal con dos o más producciones cuyos lados derechos comiencen con el mismo terminal**. Los parsers deben ser capaces de elegir entre los distintos lados derechos de un determinado noterminal con solo conocer el próximo token, y a veces los formatos de las GICs presentan ambigüedad al respecto.
2. Muchos parsers **no pueden manejar las producciones recursivas a izquierda** porque los procedimientos que implementan esas producciones recursivas entrarían en un ciclo infinito.

Estos dos problemas se pueden solucionar aplicando algunas técnicas que producen GICs equivalentes y sin conflictos.

• Factorización a izquierda

```
<sentencia if> -> if ( <condición> ) <sentencia> else <sentencia> |
                  if ( <condición> ) <sentencia>
```

Este par de producciones no es apto para una GIC LL(1) ya que **ambos lados izquierdos comienzan con el mismo token** (prefijo) y por lo tanto, el Parser no sabrá cuál de las dos producciones seleccionar.

La solución es modificar las producciones de este noterminal para que el prefijo común quede aislado en una sola producción. En general:

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots\ldots\ldots \mid \alpha\beta_n$$

α: secuencia de símbolos terminales y no terminales (prefijo) que tienen en común todas las producciones de A.

β: secuencia de cero o más símbolos, diferente para cada producción.

Factorizando a izquierda obtenemos:

$$\begin{aligned} A &\to \alpha B \\ B &\to \beta_1 \mid \\ &\quad \beta_2 \mid \\ &\quad \ldots\ldots \\ &\quad \beta_n \mid \end{aligned}$$

En el ejemplo del if queda:

```
<sentencia if> -> if ( <condición> ) <sentencia> <opción else>
<opción else> -> else <sentencia> | ε
```

Observaciones:

> Si al realizar una factorización a izquierda, **se generan nuevos prefijos**, se debe volver a aplicar sobre las producciones que correspondan.

> Hay que tener cuidado con algunas producciones porque puede ocurrir que **a simple vista no sean ambiguas**, pero en realidad **cuando se reemplaza un noterminal por cierto terminal queda ambiguo** (porque empiezan igual). En ese caso lo que se hace es desagregar la producción en todos los casos que nos provee el noterminal y luego sacar factor común:

```
S -> YbaZ | aZ
Z -> XbZ | ε
X -> cR
R -> a | b | ε
Y -> a | ε
```

Fijémonos que si reemplazamos Y por a, la producción S quedaría ambigua

```
S -> abaZ | εbaZ | aZ
```

Entonces lo que hacemos es desagregar la producción, reemplazando por todos los caminos que nos proveé Y

```
S -> aU | baZ
U -> baZ | Z
```

Y factorizamos

- **Eliminación de la recursividad a izquierda**

Las gramáticas **LL(1) no pueden ser recursivas a izquierda**. Sin embargo, la misma es necesaria para lograr la asociatividad a izquierda de un operador. Es posible eliminar la recursividad a izquierda de una GIC conservando la propiedad de la recursividad a izquierda.

Sea el par de producciones:

```
X -> Xα | β
```

X: noterminal.

α y β: secuencias de terminales y noterminales.

Podemos eliminar la recursividad a izquierda de la siguiente manera:

```
X -> βZ          toda secuencia comienza con β
Z -> αZ | ε      seguida de cero o más α
```

Observaciones

> En caso de que tengamos más terminales como producciones del lado derecho lo que se debe hacer es **agregar el nuevo noterminal en todos los casos, ya que la recursividad se corta justamente con los terminales**. Por cada "corte" que hay se le agrega el nuevo noterminal:

$$X \rightarrow X\alpha \mid \beta$$

$$S \rightarrow SXb \mid Yba \mid a$$

$$S \rightarrow YbaZ \mid aZ$$

S -> aU | baZ

U -> Z | baZ

Z -> XbZ | ε

SÍMBOLOS DE PREANÁLISIS Y EL CONJUNTO PRIMERO

Para construir un Parser que realice un **Análisis Sintáctico Descendente LL(1)** es necesario que, para cada noterminal de la GIC, el Parser pueda **determinar la producción a aplicar con solo conocer cuál es el símbolo de preanálisis**, es decir, el próximo token del flujo de tokens a procesar.

Entonces, se necesita conocer el **conjunto de todos los posibles símbolos de preanálisis** que indican que tal producción será elegida. Como un símbolo de preanálisis siempre es un único terminal, debemos conocer el **primer símbolo que puede ser producido por cierta producción**.

$$A \rightarrow x_1 x_2 \dots x_n \quad (secuencia\ de\ símbolos)$$

$$Siendo\ x_i\ un\ símbolo\ término\ o\ no\ terminal$$

$$x_1 x_2 \dots x_n\ es\ una\ secuencia\ de\ terminales\ y\ no\ temrinales\ en\ cualquier\ orden\ y\ cantidad$$

Llamamos **Primero(A)** al **conjunto de "primer símbolo" que pueden ser producidos por A**.

Obtención del conjunto primero

Sea la GIC

$NT \rightarrow S_{11}\ S_{12}\ S_{13} \dots S_{1n}$ | *producción 1*

$S_{21}\ S_{22}\ S_{23} \dots S_{2m}$ | *producción 2*

.....

$S_{k1}\ S_{k2}\ S_{k3} \dots S_{kr}$ *producción **k***

Si $\mathbf{S_{11}}$ **es un TERMINAL** , entonces Primero ($\mathbf{S_{11}}\ S_{12}\ S_{13} \dots S_{1n}$) = { $\mathbf{S_{11}}$ }

Si $\mathbf{S_{11}}$ **es un NO TERMINAL** , entonces Primero ($\mathbf{S_{11}}\ S_{12}\ S_{13} \dots S_{1n}$) = **Primero ($\mathbf{S_{11}}$)**

Si **S11 -> ε**, entonces Primero(S11 S12 S13 … S1n) = **Primero(S11) u Primero(S12)**

Algo importante a tener en cuenta es que los conjuntos primero de cada una de las producciones de un terminal NT **deben ser disjuntos**. Esto es para evitar conflictos. Recordemos que si quedan conjuntos primero iguales, podemos desagregar la producción y eventualmente sacar factor común para eliminar la ambigüedad:

A -> 1234 |
B235 |
C112 |
237

B -> 5 | 41
C -> 21 | 25 | 3

Primero (A) = Primero (**1**234) U Primero (**B**235) U Primero (**C**112) U Primero (237)

Primero (**1**234) = {**1**}
Primero (**B**235) = Primero (B) = {4,5}
Primero (**C**112) = Primero (C) = {**2**,3}
Primero (**2**37) = {2}

Primer (A) = {1,2,3,4,5}

Veamos que existe conflicto ya que hay dos producciones que poseen elementos en común en el conjunto primero. Esto lo resolvemos desagregando la producción y factorizando:

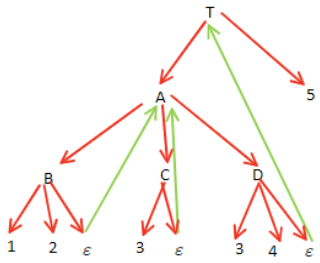
A -> 1234 |
B235 |
21112 |
25112 |
3112 |
237

A -> 1234 |
B235 |
3112
2Z

Z -> 1112 |
5112 |
37

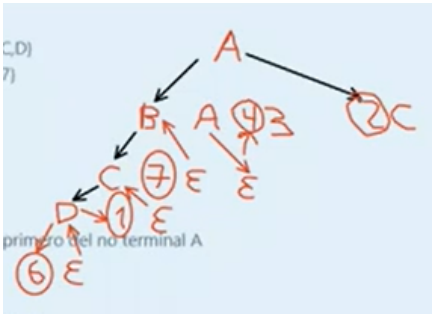
También existe otra forma de obtener el conjunto primero, **utilizando árboles** (notar que si algo produce epsilon, entonces se vuelve al padre y vamos al siguiente hermano del padre):

T -> A5
A -> BCD
B -> 1 | 2 | ε
C -> 3 | ε
D -> 3 | 4 | ε



{1,2} U {3} U {3,4} U {5}

A -> BA43 | 2C | ε
B -> C7 | ε
C -> D1 | ε
D -> 6 | ε



Primero (A) = {1,2,4,6,7}

Obtención del conjunto siguiente

Si tratamos de obtener los símbolos de preanálisis de la producción:

A -> X1….Xn

En caso de que cada Xn pueda producir ε, el símbolo de preanálisis para A está determinado por aquellos terminales que siguen inmediatamente al noterminal A. Este conjunto de terminales se denomina **Siguiente(A)**.

Para obtener el conjunto Siguiente(A) se deben tener las siguientes consideraciones:

1. El **noterminal A** puede estar seguido por el **terminal x** (...Ax…), en cuyo caso **x pertenece a Siguiente(A)**.
2. El **noterminal A** puede estar seguido por el **noterminal B** (...AB…), en cuyo caso **Siguiente(A) incluye a Primero(B)**.
3. El **noterminal A** puede ser el **último símbolo del lado derecho** en cierta producción de un **noterminal T** (T -> Y1…YmA), en cuyo caso **Siguiente(A) incluye a Siguiente(T)**.
4. Si eventualmente se llega al **axioma**, debemos tener en cuenta que el mismo **no tiene un conjunto siguiente**, ya que todo se deriva a partir de él.

T -> AE8
A -> BCD
B -> 1 | 2 | ε
C -> 3 | ε
D -> 3 | 4 | ε
E -> C | 7

Existe la posibilidad de que A genere (Derive) la palabra nula ε

Primero (T) = Primero (AE8) = Primero (BCD) = {1,2} U Primero (CD) = {1,2} U {3} U Primero(D) = {1,2} U {3} U {3,4} U ***Siguiente (A)***

Primero (T) = {1,2,3,4} U ***Siguiente (A)*** = {1,2,3,4} U ***Primero (E)*** = {1,2,3,4} U {7} U Primero (C) = {1,2,3,4} U {7} U {3} U ***Siguiente (E)***

Primero (T) = {1,2,3,4} U {7} U {3} U ***{8}*** = {1,2,3,4,7,8}

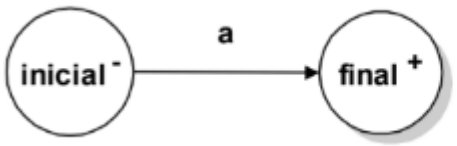
ALGORITMO DE THOMPSON

El Algoritmo de Thompson obtiene, a partir de una **expresión regular**, un **autómata finito no determinístico**.

A partir de una ER **construye un AFN por composición de bloques** de ER más sencillas realizando una lectura de izquierda a derecha.

AF asociados a ER básicas

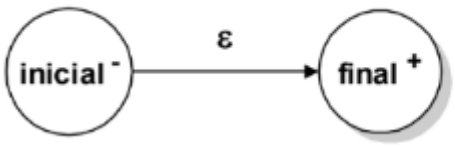
- **Autómata genérico que reconoce un símbolo del alfabeto:**



TT	a
q0	q1
q1	-

Este es un **caso particular** donde un autómata formado mediante este algoritmo queda como **AFD**, pero en general, al tener expresiones más grandes el algoritmo de Thompson siempre construye un AFN.

- **Autómata genérico que reconoce al símbolo epsilon (AFN):**

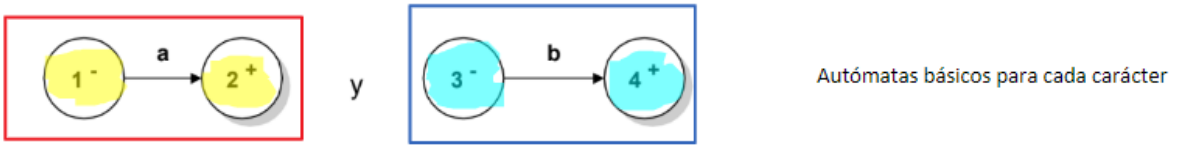


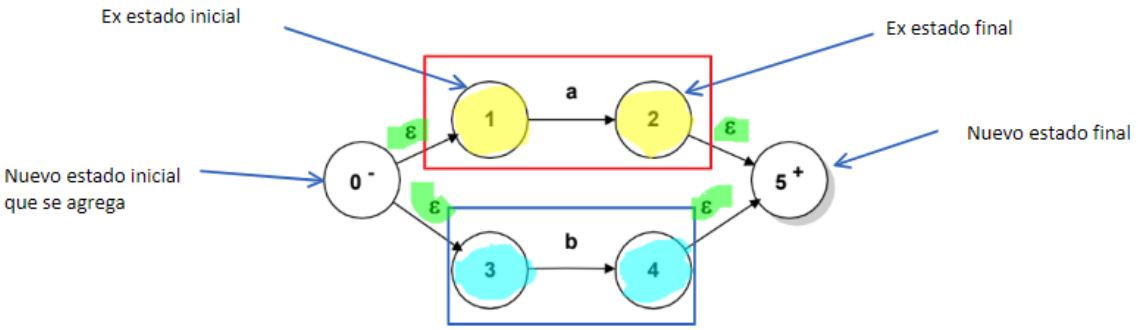
TT	ε
q0	q1
q1	-

Autómata para la unión

El **AF que reconoce la ER formada por unión de dos caracteres o símbolo ε** se construye de la siguiente forma:

1) Se construyen los dos autómatas básicos;
2) Los estados iniciales de los autómatas básicos dejan de ser iniciales y los estados finales de estos mismos autómatas dejan de ser finales;
3) Se agrega un nuevo estado inicial;
4) Se incorporan dos transiciones-ε que relacionarán al nuevo estado inicial con los dos ex estados iniciales;
5) Se añade un nuevo estado final;
6) Se trazan dos transiciones-ε para unir a los dos ex estados finales con este nuevo estado final, y el autómata está construido.



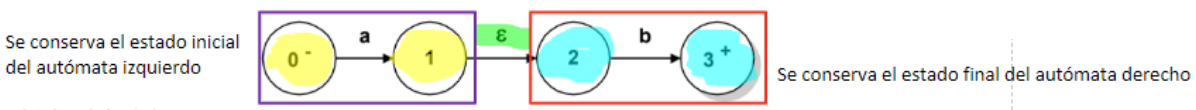


Autómata para la concatenación

1) Se construyen los autómatas básicos (denominemos *autómata izquierdo* al que reconoce el carácter "izquierdo" y *autómata derecho* al que acepta el carácter "derecho" de esta concatenación);
2) El estado final del *autómata izquierdo* deja de ser final y el estado inicial del *autómata derecho* deja de ser inicial (los otros estados no se alteran);
3) Se agrega una transición-ε que vincule al ex estado final del *autómata izquierdo* con el ex estado inicial del *autómata derecho*;
4) El AFN ya está terminado, siendo su estado inicial el estado inicial del *autómata izquierdo*, y siendo su estado final el estado final del *autómata derecho*.

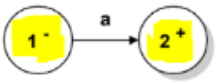
Sea la Expresión Regular ab. Para obtener el AFN "por Thompson" que reconoce a esta expresión, partimos de los dos autómatas básicos:

Siguiendo el algoritmo descripto, obtenemos el siguiente autómata:

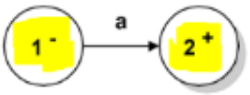


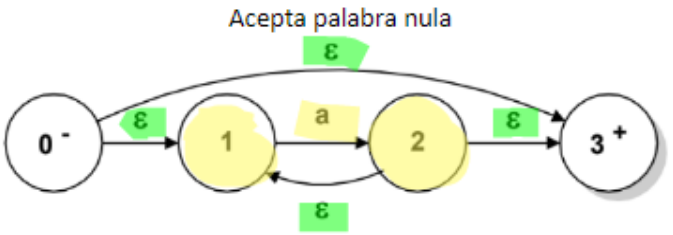
Autómata para la Clausura de Kleene

El autómata para la clausura de Kleene se obtiene así:



1) Se construye el autómata básico;
2) El estado inicial deja de ser inicial y el estado final deja de ser final;
3) Se incorporan un nuevo estado inicial y un nuevo estado final;
4) Se agrega una transición-ε desde el nuevo estado inicial hasta el ex estado inicial;
5) Se agrega una transición-ε desde el ex estado final al nuevo estado final;
6) Se incorpora una transición-ε desde el nuevo estado inicial al nuevo estado final (para que reconozca la palabra vacía);
7) Se agrega una transición-ε desde el ex estado final al ex estado inicial (para reconocer la repetición del carácter).





Restricciones del algoritmo de Thompson

> El algoritmo de Thompson **no permite que el estado inicial sea final.**

> El **estado inicial** es **único** (como en todo AF).

> El **estado final** es **único**.

> De un estado **pueden partir**, a lo sumo, **dos transiciones ε**. No pueden haber más de dos.

> En el algoritmo de Thompson se agregan estados y transiciones pero **nunca se suprimen ni estados ni transiciones**.

> En la composición por bloques de AFN **nunca se modifican los AFN ya generados**.

> Del **estado final no parten transiciones**, es decir, una vez que se transitó al estado final, no se puede transitar a ningún otro estado.

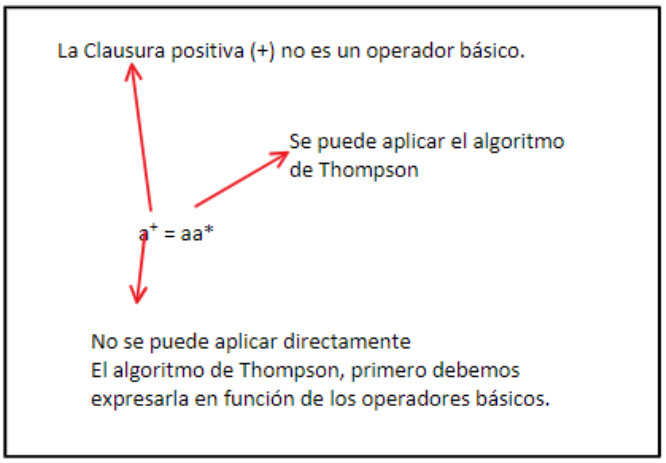
> Al **estado inicial no llegan transiciones**, es decir, una vez que se partió del estado inicial, no se puede volver a transitar al mismo desde ningún otro estado.

> El AFN Thompson **no presenta bucles**.

> **No** puede haber un estado que tenga **transición para un carácter y además una transición epsilon** (NO SE MEZCLAN).

> **No** podemos tener para un mismo estado **dos transiciones con caracteres**.

Observación



ALGORITMO CLAUSURA

El algoritmo Clausura **convierte un AFN a un AFD**. El mismo **se puede aplicar a cualquier AFN**, en particular, al AFN obtenido por el algoritmo de Thompson. Para poder definir los pasos a seguir para la construcción del autómata vamos a realizar algunas definiciones previas:

Conjunto clausura-ε de un estado

Sea q un estado de un AFN, la **clausura-ε(q)** es el conjunto de **estados** formados por **q** y todos aquellos a los cuales **se llega desde q**, utilizando solo **transiciones ε**.

En consecuencia, el **conjunto clausura-ε** de un estado **nunca puede ser vacío** porque contiene, como mínimo, a su propio estado.

Conjunto clausura-ε de un conjunto de estados

Sea R un conjunto de estados, entonces la **clausura-ε(R)** es la **unión de las clausuras-ε** de los **estados que componen al conjunto R**.

Conjunto hacia para un estado y un carácter (es como el anterior pero x, no incluye a q)

Sea q un estado y sea x un símbolo del alfabeto, entonces **hacia(q,x)** es el conjunto de estados al cual **se transita utilizando el símbolo x desde el estado q**.

Conjunto hacia para un conjunto de estados y un carácter

Sea R un conjunto de estados y sea x un símbolo del alfabeto, entonces **hacia(R,x)** es el conjunto de **estados a los cuales se transita por el símbolo x, desde los estados de R que tengan esa transición**.

Para construir el AFD desde el AFN se deben seguir los **pasos**:

1. Construir la tabla de transiciones del AFN.
2. Se obtiene el estado inicial del AFD, que es la **clausura-ε** del **estado inicial** del AFN.
3. Se agrega este estado a la primera columna de la tabla.
4. Para cada símbolo del alfabeto, **se calcula el conjunto hacia del estado que se acaba de agregar a la primera columna de la tabla**.
5. Se determinan **nuevos estados del AFD** por medio de la **clausura-ε de cada conjunto hacia recién obtenido**. Estos estados (conjuntos) se incorporan a la tabla.
6. Si un **nuevo estado obtenido** en el punto anterior **no existe todavía en la primera columna de la TT, se agrega**.
7. Se repiten los pasos 4 y 6 hasta que no surjan nuevos estados.

Si el conjunto hacia incluye un estado final, ya todo se vuelve un conjunto final.

ALGORITMO DE CLASES

El **Algoritmo de Clases** sirve para, dado un AFD, obtener el **AFD mínimo**, es decir, aquel con la menor cantidad de estados.

La importancia de obtener el AFD mínimo tiene tres aplicaciones fundamentales:

1. Determinar si dos o más AFD son equivalentes.
2. Probar la equivalencia de dos o más expresiones regulares.
3. Trabajar con el AFD mínimo beneficia la implementación computacional del AFD.

Observaciones

> Dos o más AFDs son equivalentes si el AFD mínimo que se obtiene a partir de ellos es el mismo.

> Dos o más ER son equivalentes si son reconocidas por el mismo AFD mínimo.

> No es necesario que el AFD esté completo, pero es recomendable hacerlo.

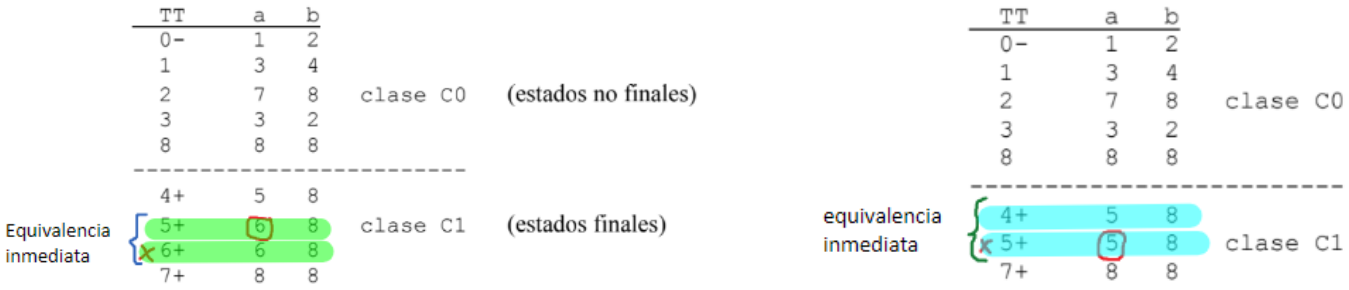
TT	a	b
0-	1	2
1	3	4
2	7	8
3	3	2
4+	5	8
5+	6	8
6+	6	8
7+	8	8
8	8	8

1. Armamos la **tabla de transición por clases**, particionando la TT en dos clases: la de **estados finales** (el estado inicial puede ser final) y la de **estados no finales** (el estado de rechazo siempre es no final). **En caso de que el estado inicial sea también final, este queda dentro de la clase de los estados finales**.

2. Analizamos **DENTRO DE CADA CLASE**, si existen estados con el mismo comportamiento. Dos estados dentro de la misma clase que además posean el mismo comportamiento se denominan estados de **equivalencia inmediata**.

   En caso de existir estos estados, debemos **tomar un representante** (en general, el de menor numeración) y eliminar el resto. Una vez eliminados estos estados, debemos **reemplazar en la tabla toda referencia a estos estados eliminados**, por el estado que tomamos como representante.

   Debemos repetir este proceso hasta eliminar todos los estados de equivalencia inmediata.

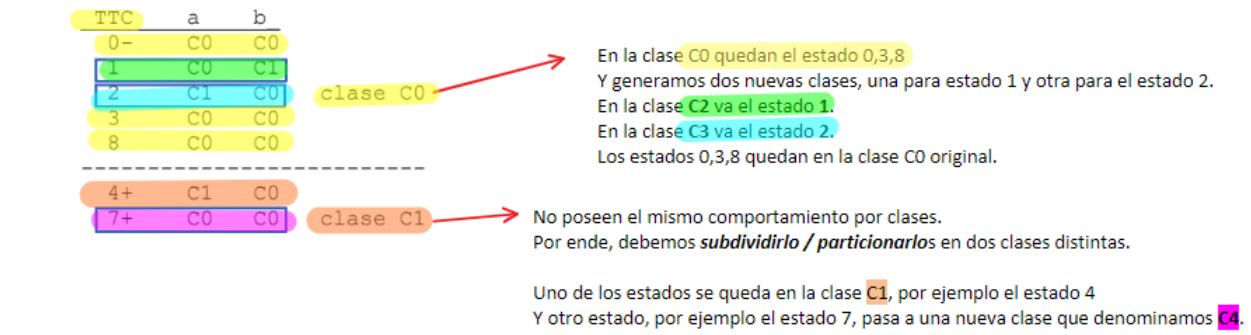


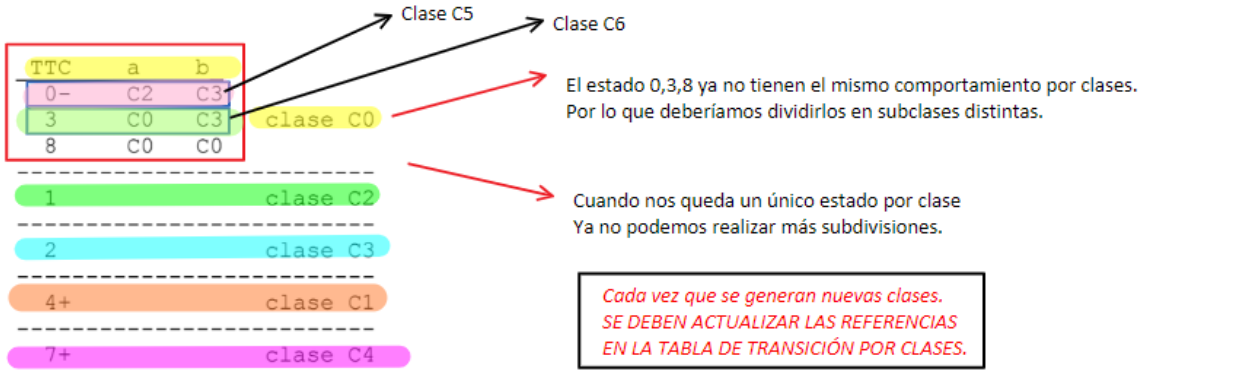
3. Existen **otros estados que pueden ser equivalentes** aunque no se lo determine de forma inmediata. Para poder determinarlos, primero armamos la **tabla de transición por clases,** tomando la última TT y reemplazando cada estado de llegada por la clase a la que pertenece ese estado.

   Luego debemos **buscar estados equivalentes por clase**, que son aquellos que están en la misma clase y tienen el mismo comportamiento. Cuando encontremos estos estados, lo que debemos hacer es **particionar la clase en subclases** que contengan a estos estados equivalentes por clase.

Recordemos que cada vez que se particiona, se deben actualizar las referencias en la tabla de transición por clases

Este proceso continúa hasta que el AFD no se pueda minimizar más, es decir, hasta que **cada clase quede formada por un único estado**, o bien cuando **algunas clases están constituidas por más de un estado equivalente por clase, pero ya no se puede particionar**, por lo cual **se toma un representante y se eliminan el resto**.





4. Una vez que tenemos todas clases formadas por un único estado, volvemos a la TT original y reemplazamos:

TT	a	b
0-	1	2
1	3	4
2	7	-
3	3	2
4+	4	-
7+	-	-

BISON Y TABLA DE SÍMBOLOS

Bison es una herramienta que nos permite construir un analizador sintáctico. La misma se vale de las funciones yyparse() y yylex() (esta última es propia del analizador léxico) para poder ejecutar el análisis.

Comunicación entre main, yylex e yyparse

El **main** analiza el archivo de entrada invocando a la función **yyparse()** que es propia del parser.

yyparse() solicita al scanner (léxico) el siguiente token. El scanner se encarga de enviar los tokens al analizador sintáctico a medida que se los solicita uno a uno.

A través de **yylex()**, el scanner se ocupa de realizar la lectura carácter a carácter del archivo fuente, y le devolverá el valor del token al parser a través de la **variable global yylval**, que es **compartida por el parser y el scanner**. Asimismo, el scanner retorna un valor entero que indica el número de token (cada token posee un número entero que lo indica), mientras que en la **variable yylval** queda almacenado, si corresponde, el valor semántico correspondiente a dicho token, es decir, toda la información correspondiente al token encontrado.

Tabla de Símbolos

La **tabla de símbolos** es un conjunto de estructuras de datos que se utiliza, como mínimo, para contener todos los identificadores del programa fuente que se está compilando, junto con los atributos que posee cada identificador.

Cabe destacar que la misma **interactúa con las tres fases de análisis** (léxico, sintáctico y semántico). Sin embargo, el principal uso que se le da es para realizar validaciones de tipo semánticas.

Las **validaciones semánticas** utilizan las rutinas semánticas que complementan a la GIC mediante el uso de la **tabla de símbolos** para realizar las validaciones. En la práctica, utilizar Gramáticas Sensibles al Contexto (GSC) para las validaciones semánticas no es algo factible para implementar computacionalmente.