Parseo y Generación de Código

Lenguajes regulares
Normalización de gramáticas

Licenciatura en Informática con Orientación en Desarrollo de Software
Universidad Nacional de Quilmes

Lenguajes regulares

Repaso

- Un **lenguaje** en un alfabeto Σ es un conjunto de palabras $L \subseteq \Sigma^\star$.
- Una **gramática independiente del contexto** es una 4-upla (N, Σ, P, S):
  - N son los **símbolos no terminales**.
  - Σ son los **símbolos terminales**.
  - P es un conjunto de **producciones** de la forma $A \to \beta$.
  - S es el **símbolo inicial**.
- Notamos $L(G)$ al lenguaje generado por G.

Repaso

- Un **autómata finito determinístico** es una 5-upla $D = (Q, \Sigma, \delta, q_0, Q_F)$, donde:
  - Q es el **conjunto de estados**.
  - Σ es el **alfabeto**.
  - $\delta : Q \times \Sigma \to Q$ es la **función de transición**
  - $q_0 \in Q$ es el **estado inicial**.
  - $Q_F \subseteq Q$ son los **estados finales**.
- Notamos $L(D)$ al lenguaje aceptado por D.

Repaso

- Un **autómata finito no determinístico** es una 5-upla $N = (Q, \Sigma, \delta, q_0, Q_F)$, donde:
  - Q es el **conjunto de estados**.
  - Σ es el **alfabeto**.
  - $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ es la **función de transición no determinística**
  - $q_0 \in Q$ es el **estado inicial**.
  - $Q_F \subseteq Q$ son los **estados finales**.
- Notamos $L(N)$ al lenguaje aceptado por N.

Repaso

- Una **expresión regular** es un árbol R formado por los siguientes constructores:
  - $\emptyset$
  - ϵ
  - x para cada $x \in \Sigma$
  - $R \cdot S$
  - $R \mid S$
  - $R^\star$
- Notamos $L(R)$ al lenguaje denotado por R.

Repaso

En la clase pasada vimos:

$$\text{AFD} \leftrightarrow \text{AFN} \leftarrow \text{expresiones regulares}$$

- AFD $\rightarrow$ AFN: Todo AFD se puede interpretar como un AFN que acepta el mismo lenguaje.
  Demostración. Inmediato.
- AFN $\rightarrow$ AFD: A partir de un AFN se puede construir un AFD que acepta el mismo lenguaje.
  Demostración. Usando la construcción de subconjuntos.
- Expresiones regulares $\rightarrow$ AFN: A partir de una expresión regular R se puede construir un AFN $N(R)$ que acepta el lenguaje denotado por R.
  Demostración. Usando la construcción de Thompson.

¿Qué pasa con la siguiente dirección?

$$\text{autómatas finitos} \rightarrow \text{expresiones regulares}$$

De autómatas finitos a expresiones regulares

Teorema. Si N es un AFN sin transiciones ϵ, se puede construir una expresión regular que denota el lenguaje aceptado por N.

La construcción usa el **método de Brzozowski**:

- Si q es un estado de un AFN N, notamos $N|_q$ al AFN que resulta de N poniendo a q como estado inicial.
- Si $N = (\{q_0, \ldots, q_n\}, \Sigma, \delta, q_0, Q_F)$ notamos L_i a $L(N|_{q_i})$ es decir al lenguaje aceptado empezando desde el estado q_i.
- Considerar el siguiente sistema de ecuaciones entre lenguajes:

$$\left\{ \; L_i \;\; = \;\; \left(\bigcup_{x \in \Sigma,\ q_j \in \delta(q_i, x)} x L_j \right) \cup \textbf{if } q_i \in Q_F \textbf{ then } \{\epsilon\} \textbf{ else } \emptyset \right.$$

$$\text{para cada } 0 \leq i \leq n$$

- El sistema se puede resolver utilizando el **lema de Arden**: el lenguaje X más chico que es solución a la siguiente ecuación:

$$X = L \cdot X \cup L'$$

está dado por $X = L^\star \cdot L'$.

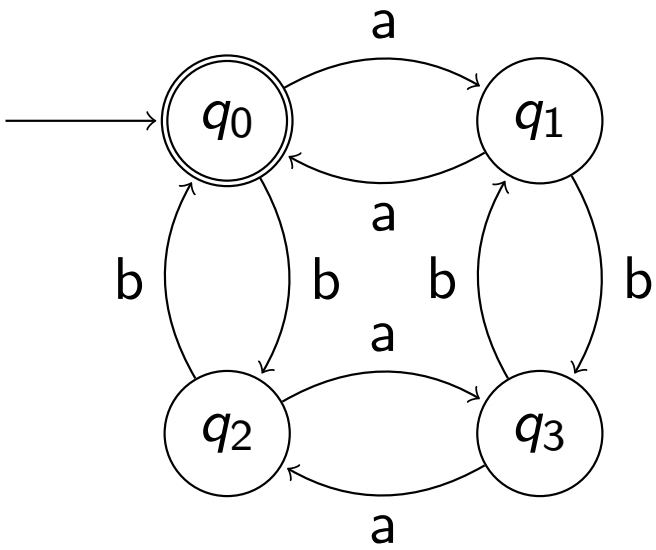
- Una vez resuelto el sistema se obtiene una expresión regular.

De autómatas finitos a expresiones regulares

Ejercicio. Dar una expresión regular que denote el lenguaje de las palabras que tienen simultáneamente un número par de *a*s y un número par de *b*s.

De autómatas finitos a expresiones regulares

El siguiente AFD en el alfabeto $\Sigma = \{a, b\}$ acepta el lenguaje que nos interesa: palabras que tienen simultáneamente un número par de as y un número par de bs.



El método de Brzozowski nos da el siguiente sistema de ecuaciones:

$$\begin{cases} L_0 &= aL_1 \cup bL_2 \cup \{\epsilon\} \\ L_1 &= aL_0 \cup bL_3 \\ L_2 &= aL_3 \cup bL_0 \\ L_3 &= aL_2 \cup bL_1 \end{cases}$$

De autómatas finitos a expresiones regulares

$$\begin{cases} L_0 &= aL_1 \cup bL_2 \cup \{\epsilon\} \\ L_1 &= aL_0 \cup bL_3 \\ L_2 &= aL_3 \cup bL_0 \\ L_3 &= aL_2 \cup bL_1 \end{cases}$$

- Reemplazar L_1 y L_2 en L_0 y L_3, obteniendo:

$$\begin{cases} L_0 &= aaL_0 \cup abL_3 \cup baL_3 \cup bbL_0 \cup \{\epsilon\} \\ L_3 &= aaL_3 \cup abL_0 \cup baL_0 \cup bbL_3 \end{cases}$$

De autómatas finitos a expresiones regulares

$$\begin{cases} L_0 &= aaL_0 \cup abL_3 \cup baL_3 \cup bbL_0 \cup \{\epsilon\} \\ L_3 &= aaL_3 \cup abL_0 \cup baL_0 \cup bbL_3 \\ &= (aa \cup bb)L_3 \cup (abL_0 \cup baL_0) \end{cases}$$

- Aplicar el lema de Arden en L_3 para deshacerse de la recursión en L_3:

$$\begin{cases} L_0 &= aaL_0 \cup abL_3 \cup baL_3 \cup bbL_0 \cup \{\epsilon\} \\ L_3 &= (aa \cup bb)^\star(abL_0 \cup baL_0) \end{cases}$$

- Reemplazar L_3 en L_0 obteniendo:

$$\begin{aligned} L_0 &= aaL_0 \cup abL_3 \cup baL_3 \cup bbL_0 \cup \{\epsilon\} \\ &= aaL_0 \cup (ab \cup ba)L_3 \cup bbL_0 \cup \{\epsilon\} \\ &= aaL_0 \cup (ab \cup ba)(aa \cup bb)^\star(abL_0 \cup baL_0) \cup bbL_0 \cup \{\epsilon\} \\ &= aaL_0 \cup ((ab \cup ba)(aa \cup bb)^\star(ab \cup ba))\, L_0 \cup bbL_0 \cup \{\epsilon\} \\ &= (aa \cup ((ab \cup ba)(aa \cup bb)^\star(ab \cup ba)) \cup bb)L_0 \cup \{\epsilon\} \end{aligned}$$

De autómatas finitos a expresiones regulares

$$L_0 = (aa \cup ((ab \cup ba)(aa \cup bb)^\star(ab \cup ba)) \cup bb)L_0 \cup \{\epsilon\}$$

- Por último, aplicando el lema de Arden nuevamente se obtiene:

$$L_0 = (aa \cup ((ab \cup ba)(aa \cup bb)^\star(ab \cup ba)) \cup bb)^\star$$

Esto resuelve el sistema, y nos asegura que L_0 se puede expresar como el lenguaje denotado por la siguiente expresión regular:

$$(aa \mid ((ab \mid ba)(aa \mid bb)^\star(ab \mid ba)) \mid bb)^\star$$

Una equivalencia más: gramáticas regulares

Una gramática independiente del contexto $G = (N, \Sigma, P, S)$ se dice **regular** si y sólo si todas las producciones son de alguna de las formas siguientes:

- $A \to xB$ para algún $x \in \Sigma$
- $A \to B$
- $A \to \epsilon$

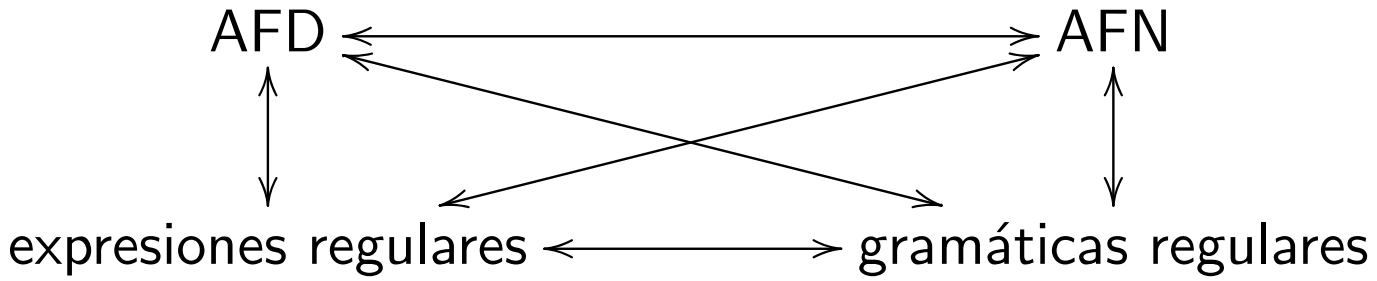
Una equivalencia más: gramáticas regulares

Es fácil ver que para cada AFN hay una gramática regular que genera el mismo lenguaje, y viceversa, según la siguiente correspondencia:

símbolo no terminal A	$\Leftrightarrow$	estado del AFN q_A
símbolo inicial S	$\Leftrightarrow$	estado inicial q_S
producción $A \to xB$	$\Leftrightarrow$	transición $q_B \in \delta(q_A, x)$
producción $A \to B$	$\Leftrightarrow$	transición $q_B \in \delta(q_A, \epsilon)$
producción $A \to \epsilon$	$\Leftrightarrow$	q_A es un estado final

Lenguajes regulares

Como conclusión obtenemos que todos los formalismos de abajo tienen el mismo poder expresivo:



Es decir, las siguientes condiciones son equivalentes:

- Existe un AFD que acepta el lenguaje L.
- Existe un AFN que acepta el lenguaje L.
- Existe una expresión regular que denota el lenguaje L.
- Existe una gramática regular que genera el lenguaje L.

Estos lenguajes se llaman **lenguajes regulares**.

Lenguajes regulares

Propiedades de los lenguajes regulares.
Los lenguajes regulares son cerrados por:

- intersección,
- unión,
- complemento,
- reverso,
- concatenación,
- clausura de Kleene.

(Ejercicios 12–14 de la práctica 2).

Lema de *pumping*

Lema de pumping. Si L es un lenguaje regular, existe un n tal que para toda cadena α en el lenguaje de longitud $|\alpha| \geq n$ se puede descomponer a α como concatenación de tres palabras:

$$\alpha = \beta\gamma\delta$$

de tal modo que:

- $|\beta\gamma| \leq n$,
- $\gamma \neq \epsilon$,
- $\beta\gamma^k\delta$ está en el lenguaje para todo $k \geq 0$.

Demostración. Más fácil de lo que parece, vemos la idea en el pizarrón.

Lema de *pumping*

Ejercicio. Usando el lema de *pumping*, probar que el lenguaje $L_1 = \{a^k b^k \mid k \geq 0\}$ no es regular.

Ejercicio. Usando el lema de *pumping*, probar que el lenguaje L_2 de las cadenas de corchetes (`[`, `]`) balanceados no es regular.

Normalización de gramáticas

Árboles de derivación vs. árboles sintácticos

En una gramática independiente del contexto $G = (N, \Sigma, P, S)$ toda derivación

$$S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$$

da lugar a un **árbol de derivación**.

El árbol de derivación se construye iterativamente de tal modo que una derivación $S \Rightarrow^\star X_1 \ldots X_n$ tiene asociado un árbol con raíz S y hojas $X_1, \ldots, X_n$, donde cada $X_i \in N \cup \Sigma$ es un símbolo terminal o no terminal.

- La derivación vacía $S \Rightarrow^\star S$ tiene asociado el árbol con un único nodo etiquetado con S.
- Para construir el árbol asociado a la derivación $S \Rightarrow^\star \alpha A \beta \Rightarrow \alpha X_1 \ldots X_n \beta$ se extiende el árbol asociado a $S \Rightarrow^\star \alpha A \beta$ de tal modo que el nodo A pasa a tener n hijos $X_1, \ldots, X_n$.

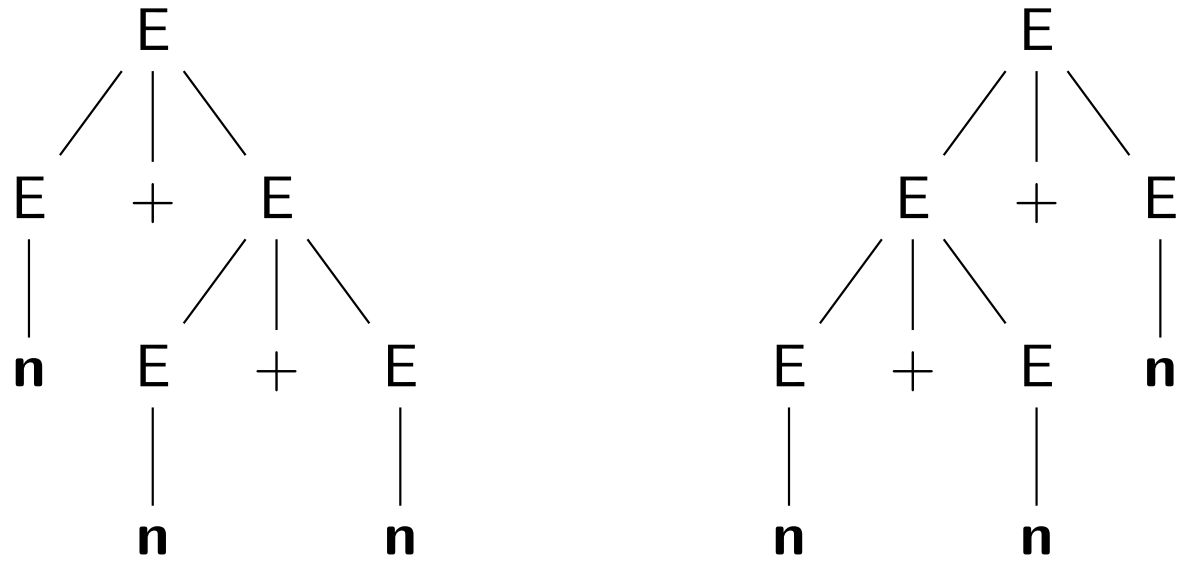
Árboles de derivación vs. árboles sintácticos

Por ejemplo, consideremos la siguiente gramática en el alfabeto $\Sigma = \{(,), \mathbf{n}, +\}$:

$$E \quad \to \quad E + E \mid \mathbf{n}$$

- La cadena $\mathbf{n} + \mathbf{n} + \mathbf{n}$ admite dos derivaciones más a la izquierda:
  - $E \Rightarrow E + E \Rightarrow \mathbf{n} + E \Rightarrow \mathbf{n} + E + E \Rightarrow \mathbf{n} + \mathbf{n} + E \Rightarrow \mathbf{n} + \mathbf{n} + \mathbf{n}$
  - $E \Rightarrow E + E \Rightarrow E + E + E \Rightarrow \mathbf{n} + E + E \Rightarrow \mathbf{n} + \mathbf{n} + E \Rightarrow \mathbf{n} + \mathbf{n} + \mathbf{n}$

  Los árboles de derivación son respectivamente:



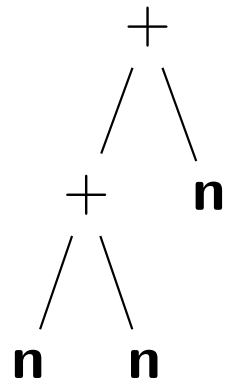
Árboles de derivación vs. árboles sintácticos

No se debe confundir el árbol de derivación con el **árbol sintáctico** o **árbol de sintaxis abstracta** (AST).

- El AST no se construye de acuerdo con una definición general que aplique universalmente a todas las gramáticas.
- La intención del AST es representar la estructura sintáctica de una expresión de acuerdo con lo que resulte relevante para el dominio al que se destina el lenguaje.

Árboles de derivación vs. árboles sintácticos

Por ejemplo, en el caso anterior, si el operador + es asociativo a izquierda, el AST sería en ambos casos:



Eliminación de la ambigüedad

Recordemos que una gramática independiente del contexto es **ambigua** si hay alguna cadena α que tiene dos derivaciones más a la izquierda distintas.

- Algunas gramáticas ambiguas se pueden reescribir para obtener otra gramática no ambigua que genera el mismo lenguaje.
- No hay un método para eliminar la ambigüedad en general. De hecho hay lenguajes *inherentemente ambiguos* que sólo se pueden escribir con gramáticas ambiguas.

Eliminación de la ambigüedad

Ejercicio. Dada la siguiente gramática:

$$E \rightarrow \mathbf{id} \mid EE \mid (E)$$

- Demostrar que es ambigua.
- Escribir una gramática no ambigua que genere el mismo lenguaje.

Eliminación de producciones ϵ

Objetivo. Dada una gramática $G = (N, \Sigma, P, S)$, queremos obtener otra gramática que genere el lenguaje $L(G) \setminus \{\epsilon\}$ y no tenga producciones de la forma $A \to \epsilon$.

Nota: si el lenguaje $L(G)$ incluye ϵ, se pueden eliminar todas las producciones ϵ y por último agregar una producción $S \to \epsilon$, obteniendo una gramática con una única producción ϵ.

Eliminación de producciones ϵ

Se pueden eliminar las producciones ϵ de G mediante el siguiente procedimiento:

Entrada: Una gramática $G = (N, \Sigma, P, S)$
Salida: Una gramática G' tal que $L(G') = L(G) \setminus \{\epsilon\}$ y tal que G' no tiene producciones ϵ.

while hay alguna producción $B \to \epsilon$
  Eliminar la producción $B \to \epsilon$.
  while hay alguna producción $A \to \alpha B \gamma$
    Reemplazar la producción $A \to \alpha B \gamma$
      por dos producciones $A \to \alpha\gamma$ y $A \to \alpha \underline{B} \gamma$
      donde $\underline{B}$ es un símbolo temporal asociado a B.
  end
end

Renombrar todos los símbolos temporales de $\underline{B}$ a B.

return G

Eliminación de producciones ϵ

Ejercicio. Eliminar las producciones ϵ de la gramática:

$$\begin{array}{lcl} A & \rightarrow & aBD \\ B & \rightarrow & b \mid D \\ D & \rightarrow & d \mid \epsilon \end{array}$$

Eliminación de ciclos

Un símbolo no terminal A es **cíclico** si hay una derivación $A \Rightarrow^+ A$.[1] Una gramática **tiene ciclos** si tiene algún símbolo cíclico.

Objetivo. Dada una gramática, obtener otra gramática sin ciclos que genere el mismo lenguaje.

- Dado un símbolo A, se puede determinar si es cíclico usando algoritmos usuales sobre grafos (p.ej. DFS o BFS).

[1] $\Rightarrow^+$ es la clausura transitiva de $\Rightarrow$, es decir, $\alpha \Rightarrow^+ \beta$ si $\alpha \Rightarrow^\star \beta$ usando al menos un paso de derivación.

Eliminación de ciclos

Una cadena $\alpha \in (N \cup \Sigma)^\star$ es una **ramificación**[2] de un símbolo cíclico A si $A = A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow \alpha$ con $n \geq 1$ donde todos los A_i son símbolos cíclicos y α no es un símbolo cíclico.

Entrada: $G = (N, \Sigma, P, S)$ y un símbolo cíclico $A \in N$.
Salida: El conjunto $\mathcal{R}$ de las ramificaciones de A.

$\mathcal{R} := \emptyset$
Marcar todos los símbolos como no visitados y poner *pila* $:= [A]$.
while *pila* no vacía
    $B :=$ *pila*.desapilar()
    foreach producción $B \to C$
            tal que C es un símbolo cíclico no visitado
        Marcar C como visitado.
        pila.apilar(C)
    end
    foreach producción $B \to \alpha$
            tal que α no es un símbolo cíclico
        $\mathcal{R} := \mathcal{R} \cup \{\alpha\}$
    end
end
return $\mathcal{R}$

[2] La terminología no es estándar.

Eliminación de ciclos

El siguiente algoritmo elimina símbolos cíclicos de una gramática **sin producciones** ϵ. Si la gramática tiene producciones ϵ se deben eliminar previamente.

Entrada: Una gramática $G = (N, \Sigma, P, S)$ sin producciones ϵ.
Salida: Una gramática G' que genera el mismo lenguaje y no tiene ciclos ni producciones ϵ.

foreach producción de la forma $A \to C$
tal que C es un símbolo cíclico
Computar el conjunto $\mathcal{R}$ de las ramificaciones de C.
Reemplazar la producción $A \to C$ por producciones
$A \to \alpha_1$
$\ldots$
$A \to \alpha_n$
donde $\{\alpha_1, \ldots, \alpha_n\} = \mathcal{R}$.
end

Eliminación de ciclos

Ejercicio. Eliminar los ciclos de la gramática:

$$\begin{array}{lcl} S & \to & B \mid aB \mid cB \\ B & \to & b \mid D \\ D & \to & d \mid B \end{array}$$

Eliminación de la recursión a izquierda

Una gramática es **recursiva a izquierda** si hay un símbolo no terminal A tal que $A \Rightarrow^+ A\alpha$.

Objetivo. Dada una gramática, obtener otra gramática sin recursión a izquierda que genere el mismo lenguaje.

Eliminación de la recursión a izquierda inmediata.
Si el símbolo A tiene las siguientes producciones:

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

donde las α_i no son vacías y los β_i no empiezan con A, se pueden reemplazar todas esas producciones por:

$$\begin{array}{lcl} A & \to & \beta_1 A' \mid \ldots \mid \beta_m A' \\ A' & \to & \alpha_1 A' \mid \alpha_n A' \mid \epsilon \end{array}$$

obteniendo una gramática que genera el mismo lenguaje.

Eliminación de la recursión a izquierda

Eliminación de la recursión a izquierda en general.

Entrada: Una gramática $G = (N, \Sigma, P, S)$
sin producciones ϵ ni ciclos.
Salida: Una gramática G' que genera el mismo lenguaje
y que no es recursiva a izquierda.

Poner los símbolos no terminales en algún orden $A_1, \ldots, A_n$.
foreach i = 1 **to** n
  foreach j = 1 to i-1
    Reemplazar cada producción $A_i \to A_j\gamma$
      por las producciones $A_i \to \delta_1\gamma \mid \ldots \mid \delta_k\gamma$
      donde $A_j \to \delta_1 \mid \ldots \mid \delta_k$
      son todas las producciones del símbolo A_j.
  end
  Eliminar la recursión a izquierda inmediata
    de las producciones del símbolo A_i.
end

Eliminación de la recursión a izquierda

Ejercicio. Eliminar la recursión a izquierda de la gramática

$$\begin{array}{lcl} S & \to & Ab \mid c \\ A & \to & d \mid Sf \mid Ag \end{array}$$

Factorización a izquierda

Una gramática con dos producciones $A \to \alpha\beta$ y $A \to \alpha\gamma$ donde $\alpha \neq \epsilon$ se puede reescribir de la siguiente manera:

$$\begin{array}{lcl} A & \to & \alpha A' \\ A' & \to & \beta \mid \gamma \end{array}$$

Esta transformación puede ser útil para preparar la gramática para hacer **análisis sintáctico predictivo**.

Generadores de parsers

Demo: generadores de parsers

Veremos cómo usar `flex` y `bison` para programar un parser para el lenguaje dado por la siguiente gramática en el alfabeto $\Sigma = \{+, *, (,), \textbf{num}\}$:

$$\begin{array}{lcl} E & \to & E \mid E + T \\ T & \to & T \mid T * F \\ F & \to & \textbf{num} \mid (E) \end{array}$$

Donde **num** está dado por la expresión regular `[0-9]+`.