TEMA 4:

Gramáticas Libres del Contexto

Autómatas y Lenguajes Formales.
Grado en Informática (2° curso).

Dpto. de Ingeniería de la Información y las Comunicaciones.

UNIVERSIDAD DE MURCIA

Índice general

1. Gramáticas y lenguajes libres del contexto

Sabemos que existen lenguajes como $\{(^n)^n \mid n \geq 0\}$ que no son regulares y eso quiere decir que no pueden ser descritos mediante expresiones regulares ni aceptados por autómatas finitos. Para estos lenguajes con cadenas de formato más complejo se necesitan otros formalismos más potentes. Uno de esos formalismos son las *gramáticas*, cuyo estudio se debe al lingüista Noam Chomsky. Existen distintos tipos de gramáticas adecuadas para distintos tipos de lenguajes. En este tema introducimos las *gramáticas libres del contexto*, que son más potente que las expresiones regulares, en el sentido de que permiten describir ciertos lenguajes que no son regulares, como el lenguaje anterior.

Definición 1 *Una GRAMÁTICA LIBRE DEL CONTEXTO, abreviadamente GLC, (en inglés: context free grammar, CFG), se define formalmente mediante una cuádrupla* $G = (V_N, V_T, S, P)$ *donde:*

> V_T *es un alfabeto de* ***símbolos terminales***
>
> V_N *es el alfabeto de* ***variables*** *(o símbolos no terminales)*
>
> S *es el* ***símbolo inicial*** *y se cumple que* $S \in V_N$
>
> P *es un conjunto finito de* ***reglas de producción*** *de la forma:*
>
> $$\boxed{A \to \alpha} \text{ donde } A \in V_N,\ \alpha \in (V_N \cup V_T)^*$$
>
> *Como vemos, en la parte izquierda de las reglas debe aparecer una única variable y en la parte derecha una cadena de símbolos de la gramática (terminales/variables). Se permiten* λ*-reglas, que son las del tipo* $A \to \lambda$ *(caso* $\alpha = \lambda$*).*

Por **convenio**, en notación teórica usaremos letras mayúsculas para las variables; dígitos y las primeras letras minúsculas del alfabeto para los símbolos terminales; las últimas letras minúsculas del alfabeto para cadenas de símbolos terminales y letras griegas como α para cualquier cadena de terminales/variables (ej. $\alpha \in (V_N \cup V_T)^*$).
Usando este convenio, a veces se suele **describir una gramática de manera compacta** enumerando únicamente sus reglas de producción y cuando varias reglas tienen la misma parte izquierda, se suelen agrupar separándolas con | (indica unión o alternancia de reglas). La variable que aparece en la parte izquierda de la primera regla entenderemos que es el símbolo inicial de la gramática.

Ejemplo 1 *Sea la gramática* G *cuyas reglas de producción son:*

$$S \to aSa \mid bSb \mid a \mid b \mid \lambda$$

La única variable de la gramática es S *(esto es,* $V_N = \{S\}$*) y* S *también es el símbolo inicial.* $V_T = \{a, b\}$ *y* $P = \{S \to aSa, S \to bSb, S \to a, S \to b, S \to \lambda\}$.

Ahora necesitamos saber cómo **usar las reglas de producción para obtener una cadena de símbolos terminales** a partir del símbolo inicial de una gramática. Para ello necesitamos definir el concepto de derivación.

Definición 2 *Se dice que una regla del tipo $A \to \gamma$ es aplicable a la cadena α si y sólo si en α aparece al menos una ocurrencia de la variable A. La* ***aplicación de la regla*** *consiste en sustituir en α la variable A (parte izquierda de la regla) por γ (parte derecha de la regla).*

Definición 3 (derivación directa) *Decimos que hay una relación de* DERIVACIÓN DIRECTA *de α en β, y lo escribimos como $\boxed{\alpha \Rightarrow \beta}$ ("α deriva directamente en β"), si β se puede obtener a partir de α por aplicación de una regla de la gramática.*

Ejemplo 2 *Considerando la gramática $S \to aSa \mid bSb \mid a \mid b \mid \lambda$, cualquiera de las reglas para la variable S son aplicables a la cadena $\alpha = abSba$. Aplicando la regla $S \to aSa$ se tiene que $ab\boxed{S}ba \Rightarrow ab\boxed{aSa}ba$.*

El nombre "gramática libre del contexto" (o independiente del contexto) viene de que en el proceso de derivación se puede sustituir cualquier variable por la parte derecha de una regla para esa variable con independencia de lo que haya a la izquierda o derecha (contexto) de la variable. Por ejemplo, en $ab\boxed{S}ba$ se puede sustituir la S por aSa sin tener en cuenta el contexto a la izquierda (la subcadena ab) o a la derecha (la subcadena ba) de la variable S.

Para generar cadenas que contengan sólo símbolos terminales a partir del símbolo inicial lo normal es que necesitemos aplicar varias reglas. Por eso hay que extender el concepto de derivación directa.

Definición 4 (derivación) *Decimos que hay una relación de* DERIVACIÓN *de α en β, que denotamos como $\alpha \Rightarrow^* \beta$ ("α* ***deriva en*** *β") si y sólo si se verifica una de las condiciones siguientes:*

1. ***Las cadenas*** *α* ***y*** *β* ***son iguales.*** *En este caso se tiene que $\alpha \Rightarrow^* \alpha$ en* ***cero pasos****, porque no se aplica ninguna regla.*
2. ***Existe una secuencia de derivaciones directas*** *de la forma:*

$$\alpha \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n = \beta, \quad \text{con } n \geq 1$$

En este caso se tiene que $\alpha \Rightarrow^* \beta$ en ***n pasos****, porque en la secuencia se aplican n reglas.*

Si $n = 1$ entonces según el caso 2 tendríamos $\alpha \Rightarrow \gamma_1 = \beta$. Esto indica que siempre que $\alpha \Rightarrow \beta$ se tiene que $\alpha \Rightarrow^* \beta$. Así que la relación de derivación **extiende la relación de derivación directa.**

La relación de derivación entre cadenas de una gramática libre del contexto cumple las siguientes **propiedades**:

1. Es *reflexiva* porque $\alpha \Rightarrow^* \alpha$, para cualquier cadena α.
2. Es *transitiva* porque si tenemos que $\alpha \Rightarrow^* \beta$ y $\beta \Rightarrow^* \gamma$ entonces $\alpha \Rightarrow^* \gamma$.
3. *Regla de inferencia de derivaciones*: si tenemos la cadena $\alpha A \beta$ y sabemos que $A \Rightarrow^* \gamma$ entonces se deduce que $\alpha A \beta \Rightarrow^* \alpha\gamma\beta$.

Definición 5 (forma sentencial y sentencia) *Sea una GLC $G = (V_N, V_T, S, P)$. Se dice que una cadena $\alpha \in (V_N \cup V_T)^*$ es una FORMA SENTENCIAL de la gramática si se cumple la condición $S \Rightarrow^* \alpha$ (α es derivable del símbolo inicial). Si $S \Rightarrow^* w$ y además w es una cadena de símbolos terminales entonces se dice que w es una SENTENCIA.*

Ejemplo 3 *Con la gramática $S \to aSa \mid bSb \mid a \mid b \mid \lambda$ tenemos que $S \Rightarrow^* bbabb$ ($bbabb$ es una sentencia) y para probarlo mostramos la* ***secuencia de derivaciones directas*** *que comienza con S y acaba con $bbabb$ (indicamos en cada paso la regla que se aplica):*

$$S \overset{S\to bSb}{\Rightarrow} bSb \overset{S\to bSb}{\Rightarrow} bbSbb \overset{S\to a}{\Rightarrow} bbabb \qquad \text{luego } S \Rightarrow^* bbabb \text{ en 3 pasos}$$

La cadena anterior a $bbabb$ en la derivación, esto es $bbSbb$, no es una sentencia porque "no está terminada", ya que aparece la variable S. Sin embargo $bbSbb$ es una ***formal sentencial*** *porque $S \Rightarrow^* bbSbb$.*

• *Podemos deducir que $S \Rightarrow^* a^n\, bbabb\, a^n$ para todo $n > 0$ porque:*
1: $S \Rightarrow^* a^n S a^n$ por aplicación de la regla $S \to aSa$ $n > 0$ veces.
2: Se tiene por otra parte que $S \Rightarrow^* bbabb$ (ya se ha probado antes).
3: Usamos la ***regla de inferencia*** *aplicada a la cadena $a^n S a^n$ (derivable de S según 1) y a la derivación $S \Rightarrow^* bbabb$ (del paso 2) y se deduce que : $S \Rightarrow^* a^n \boxed{S} a^n \Rightarrow^* a^n \boxed{bbabb} a^n$, como queríamos probar.*

Definición 6 (lenguaje generado) *Sea una gramática $G = (V_N, V_T, S, P)$. Se llama LENGUAJE GENERADO por la gramática G, y se denota como $L(G)$, al lenguaje formado por todas las cadenas de símbolos terminales que son derivables del símbolo inicial de la gramática (todas las sentencias). Formalmente,*

$$\boxed{L(G) = \{w \in V_T^* \mid S \Rightarrow^* w\}}$$

Un lenguaje se dice que es un LENGUAJE LIBRE DEL CONTEXTO si existe una gramática libre del contexto que lo genera.

2. Descripción de lenguajes mediante gramáticas

Las gramáticas libres del contexto se usan sobre todo para **describir formalmente la sintaxis de los lenguajes de programación y para construir compiladores**. En la construcción de un compilador se manejan modelos, algoritmos y herramientas que combinan expresiones regulares, gramáticas y autómatas en las diferentes etapas de construcción de la aplicación. Sin estos métodos formales sería muy costoso el desarrollo de nuevos lenguajes de programación o de versiones actualizadas de lenguajes ya existentes.

En el ámbito de los lenguajes de programación o lenguajes de comandos de un sistema operativo, las gramáticas se suelen describir mediante la **notación BNF**. Una de las particularidades de esta notación es que se le da a las variables un nombre significativo, normalmente encerrado entre ángulos, que "recuerda" el tipo de cadenas derivables de cada variable. Los terminales se encierran entre comillas, para diferenciarlos de las variables y los símbolos propios de la gramática (flecha, símbolo | de alternancia de reglas).

Ejemplo 4 *Recordamos un ejemplo que se introdujo en el tema 1. Supongamos que tenemos un lenguaje de programación simple y describimos informalmente la sintaxis de los programas de este lenguaje como: "un* PROGRAMA *comienza por la palabra clave begin, va seguido de un* BLOQUE DE SENTENCIAS *(debe haber al menos una* SENTENCIA*) y termina por la palabra clave end. Una* SENTENCIA *puede ser una sentencia de* ASIGNACIÓN *o una sentencia condicional tipo* IF *o una sentencia iterativa tipo* WHILE*. Un sentencia de* ASIGNACIÓN *se forma . . . , etc." Esta descripción narrada de la sintaxis se especifica formalmente mediante la siguiente gramática (se incluyen sólo las primeras reglas):*

$\langle$*programa*$\rangle \rightarrow$ *"begin"*$\langle$*bloque-sentencias*$\rangle$*"end"*
$\langle$*bloque-sentencias*$\rangle \rightarrow \langle$*sentencia*$\rangle \mid \langle$*sentencia*$\rangle\langle$*bloque-sentencias*$\rangle$
$\langle$*sentencia*$\rangle \rightarrow \langle$*sent-if*$\rangle \mid \langle$*sent-while*$\rangle \mid \langle$*sent-asig*$\rangle$
$\langle$*sent-asig*$\rangle \rightarrow \ldots$
$\ldots$

Lo que aparece entre ángulos son variables de la gramática y representan porciones de las cadenas del lenguaje, en este caso de los programas. Por ejemplo, la variable $\langle$*programa*$\rangle$ *representa a un programa completo y la variable* $\langle$*sentencia*$\rangle$ *representa a una sentencia de un programa. Cada regla de la gramática tiene una variable en la parte izquierda de la flecha y una o varias expresiones separadas por* $\mid$ *en parte derecha, que describen en qué consiste la parte de un programa representada por la variable de la izquierda.*

2.1. Justificación de la corrección de una gramática

El proceso de especificación de un lenguaje mediante una GLC es un proceso creativo que tiene gran importancia práctica, sobre todo en la descripción de la sintaxis de lenguajes de programación, como hemos comentado antes. Por tanto es esencial asegurar que las reglas obtenidas sirven para lo que realmente se requiere, en definitiva, que la gramática propuesta es correcta para describir el lenguaje en cuestión. Algo parecido ocurre cuando se requiere diseñar un autómata finito u obtener una expresión regular para describir cierto tipo de cadenas.

Supongamos que partimos de la especificación de cierto lenguaje LEN (informal o descripción matemática) y obtenemos una gramática G para describir LEN. Deberíamos asegurar que la gramática es correcta y lo será si y sólo si $LEN = L(G)$. Para comprobar esta igualdad hay que justificar que se cumplen las dos condiciones siguientes:

- **La gramática no es demasiado estricta**, o lo que es lo mismo, cualquier cadena del lenguaje LEN es derivable de S (símbolo inicial de G). Formalmente supondría demostrar que se cumple $LEN \subseteq L(G)$.
- **La gramática no es demasiado general**, esto es, ninguna cadena fuera del lenguaje LEN es derivable de S. Formalmente supondría demostrar que se cumple $L(G) \subseteq LEN$.

En general, si una gramática tiene variables independientes (las cadenas derivables de cada variable no dependen de lo que deriva otra variable) entonces es relativamente sencillo deducir cuál es el lenguaje $L(G)$ que genera la gramática y comprobar si coincide con el lenguaje de

partida. Esto es debido a que podemos deducir cuál es el conjunto de cadenas derivables de cada variable A de la gramática, esto es, cadenas w tal que $A \Rightarrow^* w$ y usar esta información para deducir qué cadenas son derivables del símbolo inicial S, que serán las cadenas de $L(G)$.

Ejemplo 5 *Supongamos que partimos del lenguaje* $L_{aba} = \{a^{2i}b^{j+1}a^{2k} \mid i, j, k \geq 0\}$ *y obtenemos una gramática* G *dada por* $S \to ABA,\ A \to aAa \mid \lambda,\ B \to bB \mid b$. *Vamos a justificar que la gramática es correcta para ese lenguaje,* ***considerando todas las posibles derivaciones para cada variable*** *y mostrado derivaciones genéricas, no casos concretos.*

Deducimos qué cadenas se derivan de A. *Tenemos que:*

1. $A \Rightarrow \lambda$ *por la regla* $A \to \lambda$
2. $A \overbrace{\Rightarrow^*}^{\text{apl. } i>0 \text{ veces } A\to aAa} a^iAa^i \overbrace{\Rightarrow}^{\text{apl. } A\to\lambda} a^ia^i\ , (i > 0)$
3. *Por 1 y 2 se deduce que* $\boxed{A \Rightarrow^* a^{2i},\ (i \geq 0)}$

Deducimos qué cadenas se derivan de B. *Tenemos que:*

4. $B \Rightarrow b$ *por la regla* $B \to b$
5. $B \overbrace{\Rightarrow^*}^{\text{apl. } j>0 \text{ veces } B\to bB} b^jB \overbrace{\Rightarrow}^{\text{apl. } B\to b} b^jb,\ (j > 0)$
6. *Por 4 y 5 se deduce que* $\boxed{B \Rightarrow^* b^{j+1},\ (j \geq 0)}$

Finalmente deducimos qué cadenas se derivan de S *usando la regla de inferencia de derivaciones partiendo de lo que ya hemos comprobado. Tenemos que:*

$$S \Rightarrow ABA \overbrace{\Rightarrow^*}^{\text{por 3}} a^{2i}BA \overbrace{\Rightarrow^*}^{\text{por 6}} a^{2i}b^{j+1}A \overbrace{\Rightarrow^*}^{\text{por 3}} a^{2i}b^{j+1}a^{2k},\ (i, j, k \geq 0)$$

Es importante observar que la derivación de la primera variable A *en* ABA *es independiente de la derivación de la segunda* A. *Por tanto sería un error deducir que* $S \Rightarrow^* a^{2i}b^{j+1}a^{2i}$. *Resumiendo, se ha comprobado que:*

$$\boxed{L(G) = \{a^{2i}b^{j+1}a^{2k} \mid i, j, k \geq 0\}}$$

y por tanto $L(G) = L_{aba}$, *con lo que la gramática es correcta, como queríamos probar.*

2.2. Gramáticas a partir de definiciones recursivas

Aparte de la descripción matemática por comprensión, podemos especificar formalmente un lenguaje mediante una **definición recursiva**. Para ello se describen las cadenas más simples que pertenecen al lenguaje (caso base) y se usan reglas recursivas para describir cadenas más complejas en función de otras cadenas más simples del lenguaje (casos recursivos).

Es sencillo **trasladar la definición recursiva del lenguaje a una gramática** y la corrección de la misma prácticamente está asegurada si se interpreta bien la definición del lenguaje, porque la gramática es como una formalización computacional de una descripción recursiva.

Ejemplo 6 *Sea el lenguaje B_i que se define recursivamente de la siguiente forma:*

$$\begin{array}{l}\textit{CASO BASE:}\quad \lambda \in B_i\\ \textit{CASO RECURSIVO 1: si}\ \ w \in B_i\ \ \textit{entonces}\ \ 0w1 \in B_i\\ \textit{CASO RECURSIVO 2: si}\ \ w \in B_i\ \ \textit{entonces}\ \ 1w0 \in B_i\\ \textit{CASO RECURSIVO 3: si}\ \ x, y \in B_i\ \ \textit{entonces}\ \ xy \in B_i\end{array}$$

A partir de esta descripción recursiva primero generamos unas cuantas cadenas del lenguaje para intentar deducir de qué lenguaje se trata. Sabemos inicialmente que $\lambda \in B_i$ por el caso base. Tomando $w = \lambda$, se deduce que $0 \circ \lambda \circ 1 = 01 \in B_i$ (por el caso 1); por otra parte $1 \circ \lambda \circ 0 = 10 \in B_i$ (por el caso 2). Por ser $01, 10 \in B_i$ podemos hacer que x, y sean cualquiera de esas dos cadenas y aplicando la regla del caso 3 se deduce que: $01 \circ 01 = 0101 \in B_i, 01 \circ 10 = 0110 \in B_i, 10 \circ 01 = 1001 \in B_i, 10 \circ 10 = 1010 \in B_i$. Por otra parte, a partir de que $01, 10 \in B_i$ y aplicando las reglas de los casos 1 y 2, podemos deducir que $0011, 1100 \in B_i$.
Como vemos, las cadenas generadas: $\lambda, 01, 10, 0101, 0110, 1001, 1010, 0011, 1100$, cumplen la condición de tener igual número de ceros que de unos y son todas las cadenas posibles con igual número de ceros y unos y longitud menor que 6. Si volvemos a aplicar las reglas de los casos recursivos se generan todas las cadenas de longitud $6, 8$, etc. Siguiendo un razonamiento inductivo podemos probar que todas las cadenas que pertenece a B_i según la definición recursiva cumplen la propiedad de tener el mismo número de ceros que de unos y todas las cadenas binarias que cumplen esa propiedad pueden generarse a partir de los casos de la definición recursiva de B_i. Por tanto,

$$B_i = \{w \in \{0,1\}^* \mid ceros(w) = unos(w)\}$$

Para obtener una gramática G_i para el lenguaje B_i podemos hacerlo "a ojo", una vez que sabemos qué tipo de cadenas están en ese lenguaje, o bien podemos obtener la gramática directamente a partir de la definición recursiva. Lo haremos de esta segunda forma. Vamos a ***justificar la corrección de la gramática razonando por qué incluimos cada regla.*** *Tenemos que tener en cuenta que una condición del tipo $z \in B_i$ en los casos de la definición recursiva debe ser equivalente a decir que $z \in L(G_i)$, o lo que es lo mismo, a afirmar que $S \Rightarrow^* z$. La variable S representa a cualquier cadena de igual número de ceros que de unos. Ahora incluimos las reglas para S según cada caso de la definición recursiva, de manera que se mantenga la condición de que pertenencia a B_i equivalga a ser derivable de S.*

- *Del caso base: puesto que $\lambda \in B_i$, necesitamos incluir la regla $S \to \lambda$ para que también se cumpla que $S \Rightarrow^* \lambda$.*
- *Del caso recursivo 1: supuesto que $w \in B_i$, y se supone que equivale a $S \Rightarrow^* w$, entonces sabemos que $0w1 \in B_i$. Por eso tenemos que incluir la regla $S \to 0S1$. De esta forma tendríamos $S \Rightarrow 0S1 \Rightarrow^* 0w1$, lo que asegura que $0w1$ es derivable de S, como debe ser.*
- *Del caso recursivo 2: supuesto que $w \in B_i$, y se supone que equivale a $S \Rightarrow^* w$, entonces sabemos que $1w0 \in B_i$. Por incluimos la regla $S \to 1S0$. De esta forma tendríamos $S \Rightarrow 1S0 \Rightarrow^* 1w0$, lo que asegura que $1w0$ es derivable de S, como debe ser.*

- *Del caso recursivo 3:* *supuesto que $x, y \in B_i$, y se supone que equivale a $S \Rightarrow^* x$ y $S \Rightarrow^* y$, entonces sabemos que $xy \in B_i$. Por eso tenemos que incluir la regla $S \to SS$.*
  Así tendríamos que $S \Rightarrow SS \Rightarrow^* xS \Rightarrow^* xy$, lo que asegura que xy es derivable de S.

La gramática G_i sería pues:

$$S \to \lambda \mid 0S1 \mid 1S0 \mid SS$$

y esta gramática es correcta para describir al lenguaje B_i, es decir $B_i = L(G_i)$, porque sus reglas se han obtenido conforme a la definición recursiva de B_i. Con eso también hemos probamos que el lenguaje B_i ***es un lenguaje libre del contexto****, al ser generado por una* GLC.

3. Análisis sintáctico. Árbol de derivación

Dada una gramática libre del contexto G y una cadena de símbolos terminales w, el **análisis sintáctico** de w consiste en comprobar si dicha cadena es una sentencia de la gramática, esto es, comprobar si $S \Rightarrow^* w$, o equivalentemente deducir si $w \in L(G)$. Cuando G es una gramática para un lenguaje de programación y w es un supuesto programa en dicho lenguaje, el proceso de análisis sintáctico consiste en comprobar si el programa es sintácticamente correcto conforme a las reglas sintácticas del lenguaje. Este proceso es de vital importancia en la construcción de compiladores y otro tipo de traductores. El análisis sintáctico (*parsing*, en inglés) se lleva a cabo en un módulo software llamado **analizador sintáctico**. El estudio de algoritmos e implementaciones de analizadores sintácticos se verá en la asignatura de Compiladores, así que aquí sólo veremos cómo comprobar "manualmente" si una cadena es una sentencia de cierta GLC.

Ejemplo 7 *Sea la gramática G con reglas $S \to ABA,\ A \to aAa \mid \lambda,\ B \to bB \mid b$.*

- *Vamos a probar que $aab \in L(G)$ (la cadena aab es una sentencia de G). Debemos probar que $S \Rightarrow^* aab$ y para ello mostramos la secuencia de derivaciones directas y las reglas que se aplican en cada paso:*

$$S \overset{S\to ABA}{\Rightarrow} ABA \overset{A\to aAa}{\Rightarrow} aAaBA \overset{A\to\lambda}{\Rightarrow} aaBA \overset{B\to b}{\Rightarrow} aabA \overset{A\to\lambda}{\Rightarrow} aab$$

  luego $S \Rightarrow^* aab$ en 5 pasos, porque se aplican 5 reglas.

- *Vamos a demostrar que $aabaaaa$ es sintácticamente correcta según la gramática. Debemos probar que $S \Rightarrow^* aabaaaa$ y, de nuevo, mostramos la secuencia de derivaciones directas y las reglas que se aplican en cada paso:*

$$S \overset{S\to ABA}{\Rightarrow} ABA \overset{A\to aAa}{\Rightarrow} aAaBA \overset{A\to\lambda}{\Rightarrow} aaBA \overset{B\to b}{\Rightarrow} aabA \overset{A\to aAa}{\Rightarrow} aabaAa \overset{A\to aAa}{\Rightarrow} aabaaAaa \overset{A\to\lambda}{\Rightarrow} aabaaaa$$

  luego $S \Rightarrow^* aabaaaa$ en 7 pasos, porque se aplican 7 reglas.

- *La cadena de menor longitud derivable de S es b, porque $A \Rightarrow \lambda$ por la regla $A \to \lambda$, pero por las reglas de B tenemos que B no deriva en λ. Por tanto tenemos que*

$$S \Rightarrow ABA \Rightarrow BA \Rightarrow bA \Rightarrow b \quad (S \Rightarrow^* b)$$

- *La cadena λ no es una sentencia de la gramática, puesto que hemos comprobado antes que la cadena de menor longitud derivable de S es b.*

- *Vamos a probar que babb no es una sentencia. Lo hacemos por reducción al absurdo. Supongamos que $S \Rightarrow^* babb$. Habría que aplicar inicialmente la regla $S \to ABA$, no hay otra opción. Tendríamos entonces que $S \Rightarrow ABA$. Observamos que a partir de la forma sentencial ABA no es posible generar una cadena donde una a esté rodeada de $b's$, en todo caso sería al contrario. Por tanto, S no deriva en babb, en contra de lo supuesto. Luego babb no es una sentencia.*

3.1. Árbol de derivación

En una GLC una derivación de un sentencia w se suele representar gráficamente mediante un ÁRBOL DE DERIVACIÓN (un tipo de *árbol de análisis sintáctico*). El árbol de derivación ayuda a visualizar la estructura de la sentencia (reglas que se aplican, subcadenas que son derivables de ciertas variables, etc.) y se usa en los compiladores en el proceso automático de análisis sintáctico y la posterior traducción del código fuente a código objeto.

Un árbol de derivación para una cadena w **se construye** teniendo en cuenta lo siguiente:

• Se parte del **nodo raíz** etiquetado con el símbolo inicial S. Este tendrá tantos hijos como símbolos tiene la parte derecha de la regla de S que se debe aplicar inicialmente. Cada nodo hijo se etiqueta con el símbolo correspondiente.

• Un **nodo terminal** es aquel que está etiquetado con un símbolo terminal de la gramática y se corresponde con una hoja del árbol, no puede expandirse.

• Un **nodo variable** es aquel que está etiquetado con una variable de la gramática. Un nodo variable etiquetado con cierta variable A se expande de manera análoga a como se hace para el nodo raíz. Procediendo de manera recursiva, de cada nodo variable parte un **subárbol** cuyas hojas representan la porción de la sentencia que se puede derivar a partir de esa variable.

Si $S \Rightarrow^* w$ (w es una sentencia) entonces el árbol se puede terminar de construir y concatenando las hojas del árbol de izquierda a derecha se obtiene la cadena w. Si $S \not\Rightarrow^* w$ (w es sintácticamente incorrecta) entonces no es posible terminar de expandir todos los nodos del árbol para que coincidan las hojas con los símbolos de la cadena.

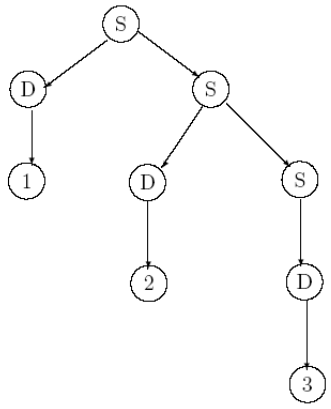
Ejemplo 8 *Sea la GLC dada por las reglas:*

$$S \to D \mid DS$$
$$D \to 0 \mid 1 \mid \ldots \mid 9$$

Con esta gramática podemos generar cualquier secuencia de dígitos de 0 a 9. Por ejemplo la sentencia 123 *se puede obtener mediante la derivación:*

$$S \overset{S\to DS}{\Rightarrow} DS \overset{D\to 1}{\Rightarrow} 1S \overset{S\to DS}{\Rightarrow} 1DS \overset{S\to D}{\Rightarrow} 1DD \overset{D\to 3}{\Rightarrow} 1D3 \overset{D\to 2}{\Rightarrow} 123$$

A esta derivación en 6 pasos le corresponde el siguiente árbol de derivación:



Nota: A un mismo árbol le pueden corresponder derivaciones distintas de la misma sentencia (diferente orden en la aplicación de las reglas), pero **a una derivación de una sentencia le corresponde un único árbol**. Por ejemplo, el árbol mostrado más arriba representa también la derivación:

$$S \overset{S\to DS}{\Rightarrow} DS \overset{S\to DS}{\Rightarrow} DDS \overset{S\to D}{\Rightarrow} DDD \overset{D\to 2}{\Rightarrow} D2D \overset{D\to 1}{\Rightarrow} 12D \overset{D\to 3}{\Rightarrow} 123$$

que usa las mismas reglas para derivar 123 que en la anterior derivación, pero en distinto orden.

3.2. Ambigüedad

Definición 7 *Una* SENTENCIA AMBIGUA *de una* GLC *es una sentencia que cumple alguna de las condiciones siguientes:*

1. *La sentencia tiene al menos* ***dos árboles de derivación distintos****.*
2. *La sentencia tiene al menos* ***dos derivación más a la izquierda distintas****, donde una derivación más a la izquierda es una secuencia de derivaciones donde en cada paso se aplica una regla para la variable que aparece más a la izquierda de la forma sentencial.*
3. *La sentencia tiene al menos* ***dos derivación más a la derecha distintas****, donde una derivación más a la derecha es una secuencia de derivaciones donde en cada paso se aplica una regla para la variable que aparece más a la derecha de la forma sentencial.*

Las tres condiciones anteriores son equivalentes, de manera que si se cumple una también se cumple cualquiera de las otras dos.

Definición 8 *Una* GRAMÁTICA AMBIGUA *es una gramática para la cual podemos encontrar al menos una sentencia ambigua.*

Ejemplo 9 *Sea la gramática* $E \to E + E \mid E * E \mid (E) \mid i$. *Con esta gramática* ***se generan expresiones aritméticas restringidas*** *(representadas por la variable* E*) con paréntesis y operadores de suma y producto aplicados a identificadores (representados aquí por el* ***token*** i*, que actúa como símbolo terminal). El conjunto de símbolos terminales de esta gramática es* $V_T = \{+, *, i, (,)\}$. *Las reglas de esta gramática podrían ser parte de otra gramática más amplia que describa la sintaxis de un lenguaje de programación cualquiera.*
Esta gramática ***es ambigua*** *porque la sentencia* $i + i * i$ *es ambigua y lo probamos encontrando dos árboles de derivación distintos para esta sentencia o equivalentemente dos derivaciones más a la izquierda o más la derecha distintas. Lo hacemos de las tres formas posibles.*

- *Derivaciones más a la izquierda (MI):*

$$\text{(MI-1): } E \overset{E\to E+E}{\Rightarrow} E + E \overset{E\to i}{\Rightarrow} i + E \overset{E\to E*E}{\Rightarrow} i + E * E \overset{E\to i}{\Rightarrow} i + i * E \overset{E\to i}{\Rightarrow} i + i * i$$

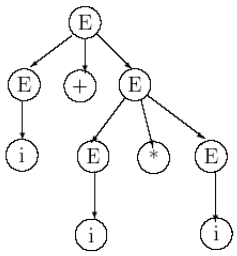
$$\text{(MI-2): } E \overset{E\to E*E}{\Rightarrow} E * E \overset{E\to E+E}{\Rightarrow} E + E * E \overset{E\to i}{\Rightarrow} i + E * E \overset{E\to i}{\Rightarrow} i + i * E \overset{E\to i}{\Rightarrow} i + i * i$$

- *Derivaciones más a la derecha (MD):*

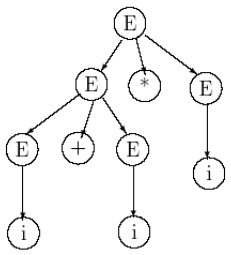
$$\text{(MD-1): } E \overset{E\to E+E}{\Rightarrow} E + E \overset{E\to E*E}{\Rightarrow} E + E * E \overset{E\to i}{\Rightarrow} E + E * i \overset{E\to i}{\Rightarrow} E + i * i \overset{E\to i}{\Rightarrow} i + i * i$$

$$\text{(MD-2): } E \overset{E\to E*E}{\Rightarrow} E * E \overset{E\to i}{\Rightarrow} E * i \overset{E\to E+E}{\Rightarrow} E + E * i \overset{E\to i}{\Rightarrow} E + i * i \overset{E\to i}{\Rightarrow} i + i * i$$

La ambigüedad queda más clara cuando obtenemos dos árboles de derivación distintos:



ARBOL (1)



ARBOL (2)

Se aprecia cómo la sentencia $i + i * i$ *(que viene representada por la variable* E *en la raíz del árbol) puede dividirse en expresiones más simples que se combinan de distinta forma mediante operadores. Eso significa que habría dos maneras distintas de interpretar o evaluar la expresión. Puesto que el operador* $*$ *tiene más prioridad que el de suma, primero se debería realizar el producto de los dos últimos operandos y sumar este resultado al primero, como si la expresión llevara los paréntesis* $i + (i * i)$. *Esto es lo que refleja el árbol (1): concatenando las etiquetas de los nodos en el primer nivel tenemos* $E + E$ *(representa una suma de términos) y observamos que del subárbol que parte de la primera* E *se obtiene* i*, lo que indica* $E \Rightarrow^* i$ *y del subárbol que parte de la segunda* E *se obtiene* $i * i$*, lo que indica que* $E \Rightarrow^* i * i$. *Según el árbol (2) las operaciones se harían* $(i + i) * i$*, cosa que no es correcta desde el punto de vista semántico, pero sintácticamente es correcto porque en las reglas de la gramática ambigua no se tiene en cuenta la precedencia ni la asociatividad de los operadores.*

La ambigüedad es una característica no deseable para una GLC ya que esto supone que una misma sentencia puede tener significados distintos, como la del ejemplo anterior. Si la gramática se usa para describir la sintaxis de un lenguaje de programación, esto puede implicar que al compilar un programa "ambiguo" el código resultante produzca resultados indeseados. Lo que se hace en la práctica para evitar la ambigüedad es **establecer reglas de precedencia**, aparte de las reglas gramaticales, que obliguen a generar sólo el árbol de significado correcto en el proceso de análisis sintáctico. Otro forma de proceder es **transformando la gramática en una equivalente que no sea ambigua**.

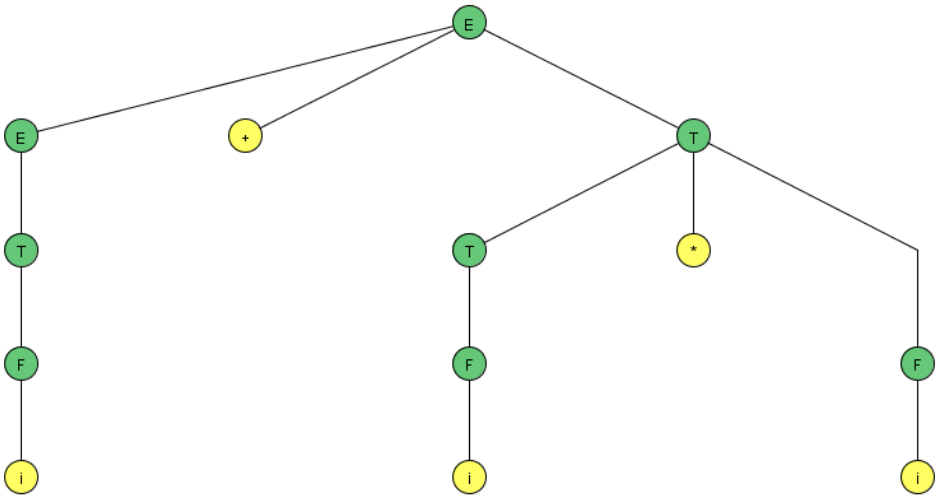
Ejemplo 10 *Existe una gramática no ambigua que genera las mismas expresiones aritméticas restringidas que la gramática anterior, por ejemplo la siguiente:*

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to i \mid (E)$$

La variable E representa a cualquier expresión aritmética restringida, que puede ser un único término o una suma de términos. La variable T representa a término, que puede ser un único terminal i (identificador) o un producto de factores. La variable F representa a un factor, que puede ser un identificador o una expresión aritmética entre paréntesis.
Teniendo en cuenta lo que representa cada variable, podemos describir la gramática anterior con nombres significativos para las variables, tipo notación BNF:

$$\langle ExprArit \rangle \to \langle ExprArit \rangle + \langle Termino \rangle \mid \langle Termino \rangle$$
$$\langle Termino \rangle \to \langle Termino \rangle * \langle Factor \rangle \mid \langle Factor \rangle$$
$$\langle Factor \rangle \to i \mid (\langle ExprArit \rangle)$$

Esta gramática tiene en cuenta en las propias reglas la precedencia de operadores y asociatividad por la izquierda por eso no es ambigua. Una sentencia como $i + i * i$ ya no es ambigua, porque tiene un único árbol de derivación, donde se aprecia que la suma se aplica al primer operado y al producto de los dos últimos, como si fuera $i + (i * i)$, y no de la forma $(i + i) * i$.



Es un resultado conocido en Teoría de la Computación que *el problema de la ambigüedad de GLC es indecidible* y eso quiere decir que **no existe ningún algoritmo que sirva para**

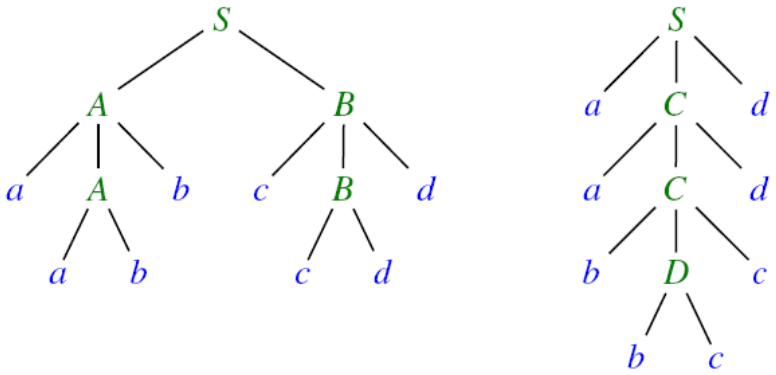
comprobar si una gramática es ambigua. Como consecuencia, tampoco existe un algoritmo que transforme una gramática en otra no ambigua equivalente. Así que el proceso de eliminar la ambigüedad de una GLC debe resolverse manualmente a la hora de obtener la gramática, como se ha hecho antes con la gramática de expresiones aritméticas. Sin embargo, **existen lenguajes libres del contexto que no pueden ser generados por gramáticas no ambiguas** (son *inherentemente ambiguos*) y eso complica el tratamiento computacional de estos lenguajes.

Ejemplo 11 *Dado el lenguaje* $L_{amb} = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^i b^j c^j d^i \mid i, j \geq 1\}$. *Vamos a obtener una GLC que genere* L_{amb} *y demostramos que es ambigua.*

$$
\begin{aligned}
S &\to AB \mid aCd \\
A &\to aAb \mid ab \\
B &\to cBd \mid cd \\
C &\to aCd \mid bDc \mid bc \\
D &\to bDc \mid bc
\end{aligned}
$$

Vemos que L_{amb} *es la unión de dos lenguajes* $L_1 \cup L_2$. *El primero* $L_1 = \{a^n b^n c^m d^m \mid n, m \geq 1\}$ *se genera a partir de* $S \Rightarrow AB$ *y usando las reglas para* A *y para* B. *Por otro lado,* $L_2 = \{a^i b^j c^j d^i \mid i, j \geq 1\}$ *se genera a partir de* $S \Rightarrow aCd$ *y usando las reglas para* C *y para* D.

Esta gramática es ambigua porque existe una sentencia ambigua, como por ejemplo: aabbccdd. Mostramos dos árboles de derivación distintos para esta sentencia:



La intersección $L_1 \cap L_2$ *es el lenguaje formado por las cadenas del tipo* $a^k b^k c^k d^k$ *con* $k \geq 1$. *Todas estas cadenas son sentencias ambiguas de* L_{amb}. *Además* ***no existe ninguna gramática no ambigua equivalente que genere*** L_{amb}, *por eso se dice que este lenguaje es* ***inherentemente ambiguo***.

4. Algoritmos de combinación de gramáticas. Propiedades de cierre

Igual que los autómatas finitos o las expresiones regulares se pueden combinar mediante operaciones de unión, concatenación o clausura, también ocurre que las gramáticas libres del contexto pueden combinarse de forma análoga para construir otra gramática que genere la unión, concatenación o clausura de los lenguajes generados por las gramáticas que se combinan.

Métodos algorítmicos para unión, concatenación y clausura de GLCs

1. **Unión de gramáticas**. Sean dos GLC $G_1 = (V_{N1}, V_T, S_1, P_1)$ y $G_2 = (V_{N2}, V_T, S_2, P_2)$ para las que se cumple que $V_{N1} \cap V_{N2} = \varnothing$ (si tuvieran variables en común se renombran en una de ellas). A partir de G_1 y G_2 se construye una GLC G_{union} tal que $L(G_{union}) = L(G_1) \cup L(G_2)$ de la siguiente forma:

$$\boxed{G_{union} = (V_{N1} \cup V_{N2} \cup \{S\}, V_T, S, P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\})}$$

   Justificación de que G_{union} genera la unión de los lenguajes: para que $S \Rightarrow^* w$ la derivación tiene que comenzar con $S \Rightarrow S_1$ y seguir derivando S_1 con las reglas de G_1 obteniendo $S_1 \Rightarrow^* z_1$, o bien comenzar por $S \Rightarrow S_2$ y seguir derivando S_2 con las reglas de G_2 obteniendo $S_2 \Rightarrow^* z_2$. De lo anterior se deduce que $w \in L(G_{union}) \Leftrightarrow (w = z_1 \wedge z_1 \in L(G_1)) \vee (w = z_2 \wedge z_2 \in L(G_2))$, luego $L(G_{union}) = L(G_1) \cup L(G_2)$.

2. **Concatenación de gramáticas**. Sean dos GLC G_1 y G_2 bajo el supuesto del método anterior. A partir de G_1 y G_2 se construye una GLC G_{concat} tal que $L(G_{concat}) = L(G_1) \circ L(G_2)$ de la siguiente forma:

$$\boxed{G_{concat} = (V_{N1} \cup V_{N2} \cup \{S\}, V_T, S, P_1 \cup P_2 \cup \{S \to S_1 S_2\})}$$

   Justificación de que G_{concat} genera la concatenación de los lenguajes: para que $S \Rightarrow^* w$ la derivación tiene que comenzar con $S \Rightarrow S_1 S_2$ y seguir derivando S_1 con las reglas de G_1 obteniendo $S_1 \Rightarrow^* z_1$ y derivando S_2 con las reglas de G_2 obteniendo $S_2 \Rightarrow^* z_2$. De lo anterior se deduce que $w \in L(G_{concat}) \Leftrightarrow w = z_1 z_2 \wedge z_1 \in L(G_1) \wedge z_2 \in L(G_2)$, luego $L(G_{concat}) = L(G_1) \circ L(G_2)$.

3. **Clausura de una gramática.** Sea $G = (V_N, V_T, S_1, P)$ una GLC. A partir de G se construye una GLC G_{star} tal que $L(G_{star}) = L(G)^*$ de la siguiente forma:

$$\boxed{G_{star} = (V_N \cup \{S\}, V_T, S, P \cup \{S \to S_1 S \mid \lambda\})}$$

   Justificación de G_{star} genera la clausura del lenguaje: para que $S \Rightarrow^* w$ la derivación debe comenzar con $S \Rightarrow \lambda$ (y acabar para el caso $w = \lambda$) o bien se debe aplicar $n > 0$ veces la regla $S \to S_1 S$. De esa manera se obtiene la forma sentencial $S \Rightarrow^* \overbrace{S_1 S_1 \ldots S_1}^{\text{n veces}} S$ y luego se debe derivar cada S_1 para generar una subcadena z_i que pertenece a $L(G)$ y acabar con la regla $S \to \lambda$. En definitiva, se cumple que S deriva en λ o en $z_1 z_2 \ldots z_n$, donde cada $z_i \in L(G)$, para todo $1 \leq i \leq n$. Y estas cadenas son precisamente las de $L(G)^*$, luego se cumple $L(G_{star}) = L(G)^*$.

Los métodos anteriores puede ser usados cuando tenemos un lenguaje que puede expresarse como unión, concatenación o clausura de lenguajes libres del contexto más simples. De esa forma podemos **abordar un problema de obtener una gramática para cierto lenguaje complejo obteniendo gramáticas para los lenguajes simples** y combinándolas convenientemente según el caso.

Ejemplo 12 *Para probar que* $L_c = \{a^i b^i c^j d^j \mid i, j \geq 1\}$ *es un lenguaje libre del contexto debemos encontrar una gramática libre del contexto que lo genere. Esta gramática es fácil de obtener a partir del hecho de que* $L_c = L_1 \circ L_2$*, donde:*

$$L_1 = \{a^i b^i \mid i \geq 1\} \quad y \quad L_2 = \{c^j d^j \mid j \geq 1\}$$

Una GLC *que genera* L_1 *es* G_1 *con reglas* $S_1 \to aS_1b \mid ab$ *y otra que genera* L_2 *es* G_2 *con reglas* $S_2 \to cS_2d \mid cd$*. Por tanto, una* GLC *que genera* L_c *se obtiene aplicando el método de concatenación a las gramáticas* G_1 *y* G_2*. Esto se hace introduciendo un nuevo símbolo inicial* S*, una nueva regla* $S \to S_1S_2$ *para "enlazar" las dos gramáticas y mantenemos las reglas para* G_1 *y* G_2*. La gramática resultante* G_c *que cumple* $L(G_c) = L_1 \circ L_2 = L_c$ *es:*

$$\begin{array}{l} S \to S_1S_2 \\ S_1 \to aS_1b \mid ab \\ S_2 \to cS_2d \mid cd \end{array}$$

Ejemplo 13 *Partimos ahora de* $L_u = \{w \in \{a,b,c,d\}^* \mid (w = a^n b^n \vee w = c^n d^n) \wedge n \geq 1\}$ *y queremos obtener una gramática libre del contexto que lo genere. Observamos que* $L_u = L_1 \cup L_2$*, donde* L_1 *y* L_2 *son como en el ejemplo anterior:*

$$L_1 = \{a^n b^n \mid n \geq 1\} \quad y \quad L_2 = \{c^n d^n \mid n \geq 1\}$$

A partir de G_1 *para* L_1 *con reglas* $S_1 \to aS_1b \mid ab$ *y* G_2 *para* L_2 *con reglas* $S_2 \to cS_2d \mid cd$ *se puede obtener* G *para* L_u *aplicando el método de unión de gramáticas. Se hace introduciendo un nuevo símbolo inicial* S*, dos nuevas reglas* $S \to S_1 \mid S_2$ *y manteniendo las reglas para* G_1 *y* G_2*. La gramática resultante* G_u *que cumple* $L(G) = L_1 \cup L_2 = L_u$ *es:*

$$\begin{array}{l} S \to S_1 \mid S_2 \\ S_1 \to aS_1b \mid ab \\ S_2 \to cS_2d \mid cd \end{array}$$

Ejemplo 14 *Sea de nuevo* $L_u = \{w \in \{a,b,c,d\}^* \mid (w = a^n b^n \vee w = c^n d^n) \wedge n \geq 1\}$*. Queremos obtener ahora una* GLC *que genere el lenguaje* L_u^**. Este lenguaje contiene cadenas formadas por concatenación repetida de cadenas de* L_u *y por tanto pueden llevar a's/b's/c's/d's, a diferencia de las cadenas de* L_u *que llevan símbolos a's/b's o c's/d's, pero no todos mezclados.*

Puesto que disponemos de la gramática G_u *que genera* L_u*, que es la dada en el ejemplo anterior:*

$$\begin{array}{l} S \to S_1 \mid S_2 \\ S_1 \to aS_1b \mid ab \\ S_2 \to cS_2d \mid cd \end{array}$$

podemos obtener otra gramática G_{star} *a partir* G_u *aplicando el método de clausura de gramáticas. Se hace introduciendo un nuevo símbolo inicial* S'*, nuevas reglas* $S' \to SS' \mid \lambda$ *y manteniendo las reglas para* G_u*. La gramática resultante* G_{star} *que cumple* $L(G_{star}) = L(G_u)^* = L_u^*$ *es:*

$$\begin{array}{l} S' \to SS' \mid \lambda \\ S \to S_1 \mid S_2 \\ S_1 \to aS_1b \mid ab \\ S_2 \to cS_2d \mid cd \end{array}$$

Los métodos algorítmicos de combinación de gramáticas también son de utilidad en resultados teóricos como el que se muestra en el siguiente teorema.

Teorema 1 (propiedades de cierre) *La clase $\mathcal{L}_{LC}$, que contiene a todos los lenguajes libres del contexto, es cerrada bajo las operaciones de unión, concatenación y clausura, o lo que es lo mismo, la unión, concatenación o clausura de lenguajes libres del contexto es un lenguaje libre del contexto.*

Dem.- Para demostrarlo habría que probar que si L_1 y L_2 son lenguajes libres del contexto cualesquiera entonces también lo es la unión $L_1 \cup L_2$ y la concatenación $L_1 \circ L_2$. Por otra parte tenemos que probar que si L es libre de contexto cualquiera también lo es su clausura L^*. Veamos que esto es así.
• Si L_1 y L_2 son libres del contexto entonces, por definición, existen gramáticas libres del contexto que los generan, pongamos por caso que $L_1 = L(G_1)$ y $L_2 = L(G_2)$. Entonces por el método de unión de gramáticas podemos asegurar que la gramática G_{union} cumple que $L(G_{union}) = L(G_1) \cup L(G_2) = L_1 \cup L_2$. Luego la unión $L_1 \cup L_2$ es un lenguaje libre del contexto al ser generado por la GLC G_{union}.
• De forma análoga se demuestra que $L(G_{concat}) = L(G_1) \circ L(G_2) = L_1 \circ L_2$, donde G_{concat} es la GLC obtenida por el método de concatenación de gramáticas. Luego la concatenación $L_1 \circ L_2$ es un lenguaje libre del contexto.
• Por último, si L es un lenguaje libre del contexto, y por tanto generado por cierta GLC G, entonces $L(G_{star}) = L(G)^* = L^*$, donde G_{star} es la GLC obtenida por el método de clausura de gramática. Luego L^* es un lenguaje libre del contexto. □

5. Gramáticas regulares y lenguajes regulares

Un tipo restringido de gramáticas libres del contexto son las llamadas **gramáticas regulares**. Se llaman así porque los lenguajes que se pueden generar con ellas son los mismos que los lenguajes que pueden ser descritos mediante expresiones regulares o aceptados por autómatas finitos, es decir, los **lenguajes regulares**.

Definición 9 *Una GRAMÁTICA REGULAR, abreviadamente GR, se define formalmente mediante una estructura de 4 componentes $G = (V_N, V_T, S, P)$ donde V_N, V_T, S se definen como en las gramáticas libres del contexto y lo que varía es el conjunto P de **reglas de producción**, donde las reglas son ahora de la forma:*

$$\boxed{\begin{array}{l} A \to bC \\ A \to b \\ A \to \lambda \end{array}}$$

donde $A, C \in V_N$ y $b \in V_T$.

Como vemos las reglas tienen una restricción adicional con respecto a las GLC generales y es que en la parte derecha siempre empiezan por un terminal y opcionalmente van seguidas de una única variable. Se permiten también λ-reglas del tipo $A \to \lambda$.

Ejemplo 15 *Sea lenguaje de los números pares N_{par}, descrito como:*

$$N_{par} = \{xp \mid x \in V_{dig}^* \wedge p \in \{0, 2, 4, 6, 8\}\}$$

Vamos a ***obtener una gramática regular*** *G_{npar} para generar el lenguaje N_{par}. Necesitamos reglas para derivar cualquier cadena que represente un número natural par y las reglas sólo deben derivar ese tipo de cadenas, no otras.* ***Justificamos la inclusión de las reglas.***
• *Incluimos las reglas: $S \to 0 \mid 2 \mid 4 \mid 6 \mid 8$ para derivar las cadenas más cortas de N_{par}, que son las que sólo tienen un dígito y es par.*
• *Sabemos que al añadir un dígito cualquiera a la izquierda de una cadena de N_{par} se obtiene también una cadena de N_{par}. Como el símbolo inicial S representa a todas las cadenas de N_{par} entonces necesitamos las reglas $S \to 0S \mid 1S \mid 2S \mid \cdots \mid 9S$ para permitir añadir dígitos a la izquierda. La gramática regular G_{npar} sería pues:*

$$S \to 0 \mid 2 \mid 4 \mid 6 \mid 8 \mid 0S \mid 1S \mid 2S \mid \cdots \mid 9S$$

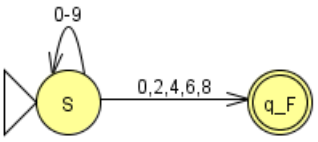
El lenguaje N_{par} es regular porque se puede describir alternativamente mediante la expresión regular $(0|1|2|3|4|5|6|7|8|9)^*(0|2|4|6|8)$. *También es sencillo obtener un autómata finito que acepte dicho lenguaje. Además este proceso de convertir una* GR *en un* AF *puede realizarse de forma algorítmica como indicamos a continuación.*

Métodos algorítmicos convertir una GR en un AF y viceversa

1. **Método GRtoAF**. Partimos de una gramática regular G y se obtiene M tal que $L(G) = L(M)$. Para ello se hace que el estado inicial de M se corresponda con el símbolo inicial de la gramática, se incluye un estado por cada variable y un estado adicional que hace de estado final. El diagrama de transición se obtiene:

   - Si tenemos la regla $A \to aB$ entonces añadimos el arco $A \xrightarrow{a} B$
   - Si tenemos la regla $A \to a \in P$ entonces añadimos el arco $A \xrightarrow{a} q_F$
   - Si tenemos la regla $A \to \lambda$ entonces añadimos el arco $A \xrightarrow{\lambda} q_F$

   Ejemplo. De la gramática $S \to 0 \mid 2 \mid 4 \mid 6 \mid 8 \mid 0S \mid 1S \mid 2S \mid \cdots \mid 9S$ que genera el lenguaje de cadenas de números pares se pasa al siguiente autómata que acepta el mismo lenguaje:

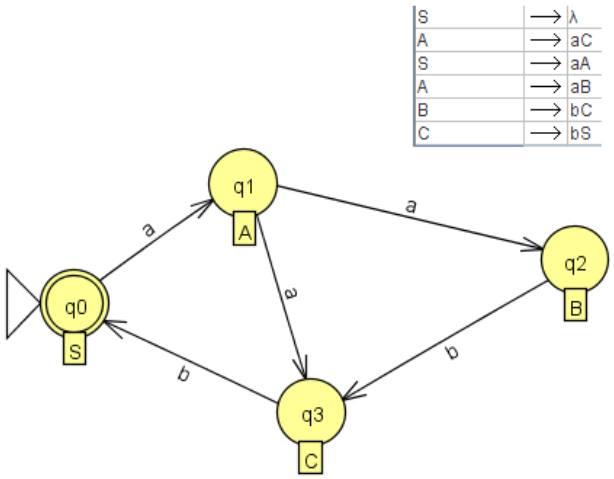
   

2. **Método AFtoGR**. Partimos de un autómata finito M sin λ-transiciones (si las tuviera se podría convertir en un AFD equivalente) y se obtiene G tal que $L(M) = L(G)$. Para ello se hace que el símbolo inicial de G se corresponda con el estado inicial de M y cada variable de G se corresponde un estado en el autómata. Las reglas de producción se obtienen a partir del diagrama de la siguiente forma:

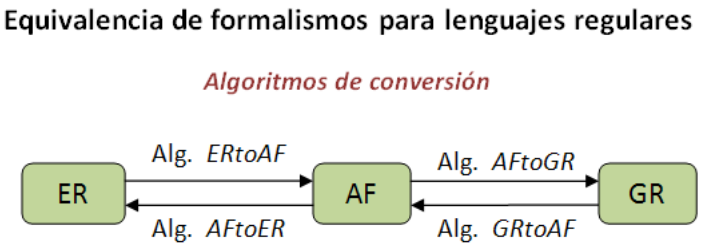
   - Por cada arco $q \xrightarrow{a} p$ del diagrama de transición de M se añade la regla $q \to ap$

- Si $q \in F$ añadimos la regla $q \to \lambda$

Ejemplo. En la siguiente figura se muestra un AF, donde se ha asociado una variable a cada estado y se muestran las reglas de la gramática que obtienen con JFLAP cuando tenemos editado un AF y seleccionamos en el menú *Convert to Grammar*.



Los algoritmos anteriores y los que se derivan del teorema de Kleene permiten asegurar que autómatas finitos, expresiones regulares y gramáticas regulares son formalismos igual de potentes, en el sentido de que todos tratan con lenguajes regulares, aunque con distinto objetivo.



Teorema 2 *Todo lenguaje regular también es un lenguaje libre del contexto, pero existen lenguajes libres del contexto que no son regulares.*

Justificación.- Habría que probar, por una parte, que **si un lenguaje L_r es regular entonces es libre del contexto**, o sea, generado por una gramática libre del contexto. Esto es cierto, porque al ser L_r regular, por definición puede ser descrito por una expresión regular y por el teorema de Kleene eso implica que existe un autómata finito M tal que $L_r = L(M)$. Por el método $AFtoGR$ sabemos que podemos obtener una gramática regular G_r a partir de M tal que $L(G_r) = L(M)$. Ahora bien, si G_r es regular entonces también es libre del contexto, porque las gramáticas regulares son un caso restringido de gramáticas libres del contexto. Luego se puede afirmar que L_r es también un lenguaje libre del contexto.
Por otra parte, hay que probar (sólo lo justificamos, no hacemos demostración formal) que **hay**

al menos un lenguaje libre del contexto que no es regular. El lenguaje de paréntesis anidados balanceados $L_{pbal} = \{(^k)^k \mid k \geq 0\}$ no es regular, como se indicó en el tema 3 al argumentar que no puede ser descrito por una expresión regular ni aceptado por autómata finito. Sin embargo este lenguaje es libre del contexto. Basta observar que $L_{pbal} = L(G)$, donde G es la GLC con reglas: $S \to (S) \mid \lambda$.

6. Algoritmos de transformación de gramáticas libres del contexto

Se dice que dos gramáticas G_1 y G_2 son equivalentes si generan el mismo lenguaje, esto es, si $L(G_1) = L(G_2)$. A veces interesa encontrar una gramática libre del contexto equivalente a una dada, por ejemplo para quitarle las variables o terminales innecesarios, para eliminar las λ-reglas, etc. En esta sección introducimos varios algoritmos que transforman una gramática eliminando o añadiendo reglas y esa transformación no afecta al lenguaje generado. Algunas transformaciones son necesarias para asegurar que ciertos algoritmos de análisis sintáctico funcionen correctamente y otras son convenientes para mejorar la eficiencia del proceso de análisis sintáctico. Estos algoritmos se aplicarán en la asignatura de Compiladores.

6.1. Gramática sin símbolos inútiles

Definición 10

- Una variable A de una GLC se dice que es **improductiva** si a partir de ella no se puede derivar una cadena de símbolos terminales. En otro caso, la variable es **productiva** (cumple que existe una cadena de terminales z tal que $A \Rightarrow^* z$).
- Un símbolo $X \in V_N \cup V_T$ (variable o terminal) se dice que es **inaccesible** si no aparece en ninguna forma sentencial de la gramática (no existe una derivación de la forma $S \Rightarrow^* \alpha X \beta$). En otro caso X es **accesible**.
- Un símbolo $X \in V_N \cup V_T$ es **inútil** si es una variable improductiva o es un símbolo inaccesible.

Los símbolos inútiles se puede suprimir de una GLC sin que ello afecte al lenguaje generado por la gramática. Si eliminamos símbolos inútiles, la gramática se simplifica y su tratamiento computacional es más eficiente. El siguiente algoritmo obtiene una gramática equivalente a una dada sin símbolos inútiles.

Algoritmo de eliminación de símbolos inútiles de una GLC

ENTRADA: una gramática libre del contexto $G_{in} = (V_N, V_T, S, P)$.

SALIDA: una gramática libre del contexto G_{out} equivalente a G_{in} sin símbolos inútiles.

PASO 1: **Eliminar variables improductivas:**

1.1 Inicializar un conjunto $Vpro$ de variables productivas añadiendo todas las variables A de V_N para las que existe una regla $A \to w$, tal que $w \in V_T^*$ (sólo deriva en terminales o λ);

1.2 REPETIR:

Examinar cada regla $B \to \alpha$ de P y si todas las variables que aparecen en α están ya en $Vpro$ entonces añadir B a $Vpro$;

HASTA que no se añadan variables nuevas a $Vpro$;

1.3 Hacer $V_N \leftarrow Vpro$ y de esa forma se eliminan las variables improductivas de V_N;

1.4 Eliminar de P aquellas reglas que tienen alguna variable improductiva (ya no pertenece a V_N) en la parte izda. o derecha;

PASO 2: **Eliminar símbolos inaccesibles**

2.1 Inicializar un conjunto $Vacc$ de variables accesibles añadiendo el símbolo inicial S;

2.2 REPETIR:

Para cada variable A en $Vacc$, examinar cada regla del tipo $A \to \alpha$ en P y añadir a $Vacc$ todas las variables que aparecen en α;

HASTA que no se añadan variables nuevas a $Vacc$;

2.3 Hacer $V_N \leftarrow Vacc$ y de esa forma se eliminan las variables inaccesibles de V_N (y quedan sólo las útiles);

2.4 Eliminar de P aquellas reglas que tienen alguna variable inaccesible (ya no pertenece a V_N) en la parte izda. o derecha;

2.5 Eliminar de V_T todos aquellos símbolos terminales que no aparecen en las reglas que quedan en P (son terminales inaccesibles);

Devolver (G_{out}), con los componentes V_N, V_T, P modificados por los pasos anteriores y S es el símbolo inicial.

Ejemplo 16 *Vamos a obtener una gramática equivalente sin símbolos inútiles aplicando el algoritmo anterior.*

$$S \to aAS \mid AA$$
$$A \to AbB \mid ACa \mid a$$
$$B \to ABa \mid Ab \mid \lambda$$
$$C \to Cab \mid CC$$
$$D \to CD \mid Cb \mid e$$
$$E \to dA$$

1. Eliminar variables improductivas. *Tenemos que seleccionar primero las variables productivas. En principio las variables A, B, D son productivas (por las reglas $A \to a$, $B \to \lambda$ y $D \to e$) y son las que inicialmente se añaden a $Vpro$ (paso 1.1). Después, por el bucle 1.2., podemos añadir a $Vpro$ la variable S (por la regla $S \to AA$) y la variable E (por la regla $D \to dA$). Y ya no se añade ninguna más al conjunto de variables improductivas, que queda $Vpro = \{S, A, B, D, E\}$. Así que el conjunto de variables queda modificado (paso 1.3) como: $V_N = \{S, A, B, D, E\}$, donde se ha eliminado la variable improductiva C.*
En el paso 1.4. modificamos el conjunto de reglas P eliminando las que tienen variables improductivas (C en este caso):

$$S \to aAS \mid AA$$
$$A \to AbB \mid a$$
$$B \to ABa \mid Ab \mid \lambda$$
$$D \to e$$
$$E \to dA$$

2. Eliminar símbolos inaccesibles. *Incializamos el conjunto de variables accesibles: $Vacc = \{S\}$ (paso 2.1). Por el bucle del paso 2.2 revisamos las reglas de P y vamos añadiendo de forma incremental a $Vacc$ todas las variables que aparecen en la parte derecha de las reglas para las variables que ya están en $Vacc$ (al principio S). Se añade A por la regla $S \to aAS$ y queda $Vacc = \{S, A\}$. Ahora se examinan las reglas para A y se añaden las variables que aparecen en la parte derecha. Se añade B a $Vacc$ por la regla $A \to AbB$ y ya no se añade ninguna más porque las variables que aparecen en las reglas para B ya se han añadido. Así que queda $Vacc = \{S, A, B\}$ y se hace que $V_N = \{S, A, B\}$ (paso 2.3). Por tanto, se han eliminado las variables inaccesibles D, E.*
En el paso 2.4. modificamos el conjunto de reglas, eliminando las que tienen variables inaccesibles D, E y finalmente el conjunto de reglas queda como:

$$S \to aAS \mid AA$$
$$A \to AbB \mid a$$
$$B \to ABa \mid Ab \mid \lambda$$

Por último, se obtiene $V_T = \{a, b\}$, que son los símbolos que aparecen en las reglas anteriores (el símbolo 'e' era inaccesible al serlo la variable D). Y la gramática resultante sin símbolos inútiles que se obtiene como salida el algoritmo tiene como componentes: $(V_N = \{S, A, B\}, V_T = \{a, b\}, S, P)$, donde P contiene las reglas indicadas en la caja anterior.

En el proceso de eliminación de símbolos inútiles es importante **respetar el orden de los pasos: primero eliminar variables improductivas y segundo eliminar símbolos inaccesibles**. Si se hace en el orden contrario el resultado puede ser incorrecto.

6.2. Gramática λ-libre

Definición 11 *Decimos que una GLC con símbolo inicial S es λ-LIBRE si no contiene reglas de la forma $A \to \lambda$ (λ-reglas), excepto $S \to \lambda$, siempre y cuando S no aparezca en la parte derecha de ninguna regla de producción.*

Teorema 3 *Para toda gramática libre del contexto existe otra GLC equivalente que es λ-libre.*

La parte constructiva de la demostración de este teorema proporciona un algoritmo para transformar una GLC en otra equivalente λ-LIBRE. El método elimina las λ-reglas e introduce a cambio reglas nuevas para obtener una gramática equivalente λ-libre. Un ejemplo que motiva el método sería el siguiente. Supongamos que tenemos en una gramática las reglas: $A \to \lambda, D \to aABc$ y la derivación:

$$D \Rightarrow aABc \Rightarrow aBc, \text{por lo que } D \Rightarrow^* aBc$$

Entonces si eliminamos $A \to \lambda$ deberíamos introducir la regla $D \to aBc$, para que $D \Rightarrow^* aBc$, sin utilizar la λ-regla de A. Si además tenemos $B \to \lambda$ entonces al eliminar esta regla se debe introducir la regla $D \to ac$.

Algoritmo de transformación a gramática λ-libre

ENTRADA: una gramática libre del contexto $G_{in} = (V_N, V_T, S, P)$.

SALIDA: una gramática G_{out} equivalente a G_{in} y λ-libre.

1. **Obtener el conjunto de variables anulables** $Vanu = \{A \in V_N | A \Rightarrow^* \lambda\}$ (derivan en λ):

   1.1 Inicializar $Vanu$ añadiendo todas las variables A tal que $A \to \lambda \in P$;

   1.2 REPETIR:

   Por cada regla del tipo $B \to C_1C_2 \ldots C_n$, donde todas las variables $C_i \in Vanu$ (son anulables) entonces se añade B a $Vanu$;

   HASTA que no se añadan variables nuevas a $Vanu$;

2. **Eliminar λ-reglas y añadir nuevas**

   2.1 Eliminar de P todas las reglas de la forma $A \to \lambda$;

   2.2 Por cada regla $A \to \alpha \in P$ se añaden a P todas las reglas que se generan al considerar que cada variable anulable que aparece en α se incluye en una regla y no se incluye en otra regla (si hay n variable anulables en α entonces se podrían generar 2^n reglas);

   2.3 Eliminar de P las λ-reglas que se hayan generado en el paso anterior (ocurre cuando toda la parte derecha de una regla contiene sólo anulables);

3. **Añadir nuevo símbolo inicial y reglas adicionales si es necesario**
   Si S está en $Vanu$ ($S \Rightarrow^* \lambda$ en la gramática de entrada) entonces:

   Si S no aparece en la parte derecha entonces añadir la regla $S \to \lambda$ a P;

   En otro caso

- Considerar un **nuevo símbolo inicial** S' y añadir S' a V_N;
- Añadir las reglas $S' \to S|\lambda$ a P;

Devolver (G_{out}), con los componentes de G_{in} modificados por los pasos anteriores.

Es de destacar que **la gramática podría quedar con símbolos inútiles** al finalizar el algoritmo de transformación, como se muestra en el siguiente ejemplo.

Ejemplo 17 *Aplicamos el algoritmo para obtener una GLC λ-libre equivalente a la siguiente:*

$$\begin{aligned} S &\to AB \mid 0S1 \\ A &\to 0ABC \mid \lambda \\ B &\to B1 \mid \lambda \\ C &\to \lambda \end{aligned}$$

Paso 1.- identificamos las ***variables anulables****. En principio se inicializa $Vanu = \{A, B, C\}$ por las reglas $A \to \lambda, B \to \lambda, C \to \lambda$. Y luego se añade también S en el paso 1.2, por la regla $S \to AB$. Por tanto queda $Vanu = \{S, A, B, C\}$ (todas las variables son anulables).*

Paso 2.- eliminamos las λ-reglas (paso 2.1) y queda:

$$\begin{aligned} S &\to AB \mid 0S1 \\ A &\to 0ABC \\ B &\to B1 \end{aligned}$$

Ahora se considera cada regla por turno. Por cada una de estas reglas se genera un grupo de reglas nuevas, por el paso 2.2. Las reglas generadas son las que resultan al considerar que cada variable anulable en la parte derecha de la regla se puede incluir en una regla y se puede excluir en otra. Por ejemplo, a partir de la regla $S \to AB$, y teniendo en cuenta que A y B son anulables, se generan las reglas $S \to AB \mid A \mid B \mid \lambda$. La regla $S \to \lambda$ se genera considerando el caso en que A y B se excluyen simultáneamente, pero habría que eliminarla luego en el paso 2.3. Procediendo de esta forma, el conjunto de reglas quedaría:

$$\begin{aligned} S &\to AB \mid A \mid B \mid 0S1 \mid 01 \\ A &\to 0ABC \mid 0AB \mid 0AC \mid 0BC \mid 0A \mid 0B \mid 0C \mid 0 \\ B &\to B1 \mid 1 \end{aligned}$$

Paso 3.- Puesto que $S \Rightarrow^* \lambda$ en la gramática de entrada, tenemos que permitir que el símbolo inicial de la gramática de salida derive en λ, pero no podemos incluir la regla $S \to \lambda$ porque tenemos la regla $S \to 0S1$ (S aparece en la parte derecha de una regla). Lo que se hace en este caso es ***añadir un nuevo símbolo inicial*** *S' e introducir las reglas $S' \to S \mid \lambda$. La gramática resultante es:*

Gramática λ-libre:

$$\begin{aligned} S' &\to S \mid \lambda \\ S &\to AB \mid 0S1 \mid A \mid B \mid 01 \\ A &\to 0ABC \mid 0AB \mid 0AC \mid 0BC \mid 0A \mid 0B \mid 0C \mid 0 \\ B &\to B1 \mid 1 \end{aligned}$$

Con esto hemos obtenido una gramática λ-libre equivalente. Pero como efecto colateral de haber suprimido la regla $C \to \lambda$ se tiene que la variable C queda improductiva. Se puede eliminar esta variable improductiva aplicando el algoritmo de eliminación de símbolos inútiles a la gramática anterior y queda:

Gramática λ-libre y sin símbolos inútiles:
$S' \to S \mid \lambda$
$S \to AB \mid 0S1 \mid A \mid B \mid 01$
$A \to 0AB \mid 0A \mid 0B \mid 0$
$B \to B1 \mid 1$

6.3. Gramática sin reglas unitarias

Definición 12 *Llamamos* REGLA UNITARIA *a una regla de la forma $A \to B$, con $A, B \in V_N$.*

Las reglas unitarias suponen alargar las derivaciones con pasos innecesarios (también pasa esto con las λ-reglas). Eliminando convenientemente estas reglas se hace más eficiente el proceso de análisis sintáctico. El siguiente algoritmo permite transformar cualquier gramática libre del contexto en otra equivalente sin reglas unitarias.

Algoritmo de eliminación de reglas unitarias

ENTRADA: una GLC $G_{in} = (V_N, V_T, S, P)$.

SALIDA: una GLC equivalente G_{out} sin reglas unitarias.

0. **Paso previo:** si G_{in} no es λ-libre entonces G_{in} se sustituye por la gramática que se obtiene al aplicar a G_{in} el algoritmo de transformación a gramática λ-libre.
1. Para cada variable $A \in V_N$ se calcula $V_{uni}(A) = \{B \in V_N | A \Rightarrow^* B, B \neq A\}$, que es el conjunto de **variables derivables de A por reglas unitarias**;
2. **Eliminar las reglas unitarias** de P;
3. **Añadir nuevas reglas:** para cada variable A tal que $V_{uni}(A) \neq \varnothing$ hacer
   Para cada variable $B \in V_{uni}(A)$
   Para cada regla de la forma $B \to \beta$
   Añadir la regla $A \to \beta$;
4. Eliminar de V_N todas aquellas variables que no aparecen en las reglas que quedan en P;
5. Devolver (G_{out}), con los componentes de G_{in} modificados por los pasos anteriores.

Al igual que pasaba con el algoritmo de transformación a gramática λ-libre, en este caso también pueden aparecer símbolos inútiles en la gramática resultante.

Ejemplo 18 *Vamos a obtener una gramática sin reglas unitarias equivalente a la siguiente:*

$$S \to aBc \mid A \mid aAb$$
$$A \to B \mid cd$$
$$B \to ccBS \mid dc$$

Paso 0.- *Como la gramática de entrada es λ-libre se pasa al siguiente paso.*

Paso 1.- *Calculamos conjunto de variables derivables de cada variable por reglas unitarias:*

$$V_{uni}(S) = \{A, B\}, V_{uni}(A) = \{B\}, V_{uni}(B) = \varnothing$$

Paso 2.- *En este paso eliminamos las reglas unitarias de P y queda:*

$$S \to aBc \mid aAb$$
$$A \to cd$$
$$B \to ccBS \mid dc$$

Paso 3.- *Ahora añadimos reglas a P a cambio de las unitarias que hemos excluido, con objeto de que el lenguaje generado no varíe. Para ello nos fijamos en los conjuntos V_{uni} obtenidos en el paso 1. Consideramos un orden arbitrario para las variables, por ejemplo: $S < A < B$ y vamos añadiendo reglas para cada variable, desde S hasta B.*
Puesto que $V_{uni}(S) = \{A, B\}$, se añaden reglas para S con la parte derecha de las reglas de las variables A y B. Con lo cual habría que añadir $S \to cd \mid ccBS \mid dc$. Después consideramos la variable A, con $V_{uni}(A) = \{B\}$, y habría que añadir reglas para A con la parte derecha de las reglas para B, o sea, añadimos $A \to ccBS \mid dc$. Como $V_{uni}(B) = \varnothing$ ya no se añade ninguna regla más y la gramática con variables no unitarias equivalente es:

$$S \to aBc \mid aAb \mid \underline{cd \mid ccBS \mid dc}$$
$$A \to cd \mid \underline{ccBS \mid dc}$$
$$B \to ccBS \mid dc$$

6.4. Gramática propia

Definición 13 *Una gramática libre del contexto es **libre de ciclos** si es imposible que se produzca una derivación de la forma $A \Rightarrow^+ A$.*

Algunos métodos de análisis sintáctico, al intentar comprobar si cierta cadena de entrada w es derivable del símbolo inicial, funcionan con backtracking probando todas las reglas posibles hasta conseguir generar una sentencia que coincide con la de partida o determinar que la cadena w es sintácticamente incorrecta por agotar todas las posibilidades de aplicación de reglas sin conseguir generar la sentencia. Si la gramática tiene ciclos eso puede suponer que el algoritmo entre en bucle infinito, por ejemplo al intentar derivar cierta variable A escogiendo las reglas que llevan una y otra vez a A (por ser $A \Rightarrow^+ A$). Por eso se considera que los ciclos deben eliminarse de una gramática.

Definición 14 *Una gramática libre del contexto es **propia** si no tiene símbolos inútiles, es λ-libre y libre de ciclos.*

Teorema 4 *Para toda gramática libre del contexto existe una* GLC *equivalente que es propia.*

La parte constructiva de la demostración proporciona un algoritmo para transformar una gramática en otra equivalente propia, que mostramos a continuación.

Algoritmo de transformación a gramática propia

ENTRADA: una GLC G.

SALIDA: una gramática equivalente propia.

1. Se aplica a G el algoritmo de **eliminación de símbolos inútiles** (para simplificar el resto de pasos) y se obtiene otra equivalente G_2.
2. Se aplica a G_2 el algoritmo de **transformación a λ-libre** y se obtiene G_3.
3. Se aplica a G_3 el algoritmo de **eliminación de las reglas unitarias** (y con ello los ciclos) y se obtiene G_4.
4. Se aplica a G_4 el algoritmo de **eliminación de símbolos inútiles** y se obtiene G_5.
5. Devolver (G_5)

Ejemplo 19 *El hecho de que se use en el paso 3 el algoritmo de eliminar reglas unitarias para conseguir que no haya ciclos en la gramática no implica que una gramática propia no pueda tener reglas unitarias. Veamos si esta gramática es propia:*

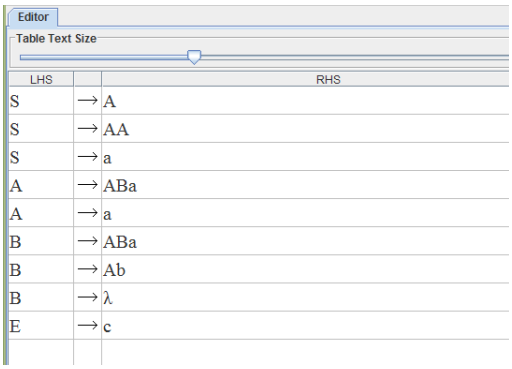
$$S \to aBc \mid A \mid aAb$$
$$A \to B \mid cd$$
$$B \to ccBS \mid dc$$

Es propia porque se puede comprobar que es λ-libre, sin símbolos inútiles y sin ciclos, aunque tenga reglas unitarias.

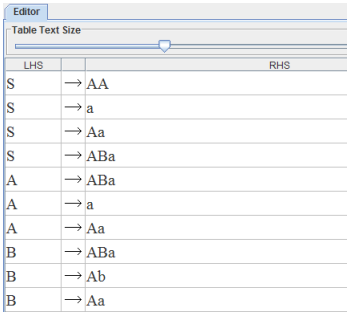
Algunos algoritmos de transformación de gramáticas exigen que la gramática de entrada sea propia. Al ser propia no tiene ciclos y eso garantiza que el algoritmo no entra en bucle infinito. Al no tener tampoco λ-reglas se mejora la eficiencia al acortar las derivaciones. Sin embargo, una gramática propia obtenida por el método de transformación puede tener bastantes más reglas que la gramática original, como consecuencia de la generación de reglas por eliminación de las λ-reglas y reglas unitarias. Hay más reglas, pero puede suponer menos pasos en las derivaciones.

Ejemplo 20 *El siguiente ejemplo se puede hacer paso a paso con* JFLAP *(ver video tutorial5-GLC-jflap, accesible desde Aula virtual → Recursos → carpeta* JFLAP*).*

- *Entra en la opción Grammar del menú principal y abre el fichero ejem-G-a-propia.jff de la carpeta tema4-jflap, disponible en recursos del aula virtual*



- *Selecciona en la ventana de editor de gramática Convert → Transform Grammar.*
- *Primero se aplica el algoritmo Lambda removal (para obtener gramática λ-libre) y se ve el resultado con Do all.*
- *Después con Proceed pasa a aplicar el algoritmo Unit removal (eliminación de reglas unitarias) y la gramática aparece tras Do all.*
- *Después de Proceed aplica el algoritmo Useless removal (elimina símbolos inútiles) y Do all.*
- *Finalmente seleccionamos Export y ya se tiene la gramática propia disponible en otra ventana:*



7. Preguntas de evaluación

7.1. Problemas resueltos

1. Sea G con reglas: $S \to aSa \mid bSb \mid a \mid b \mid \lambda$. Deduce cuál es el lenguaje generado por esta gramática, mostrando todas las posibles derivaciones genéricas y descríbelo con palabras de forma concisa y matemáticamente por comprensión.

   Solución. Algunas sentencias son $\lambda, a, b, aa, bb, aba, bab, abba, baab, aaaa, \ldots$. Las cadenas generadas parecen seguir un patrón y es que se leen igual de izquierda a derecha que de derecha a izquierda, o sea, son palíndromos. Veamos si esto es siempre así.

   Las derivaciones en un paso que generan sentencias son ($S \Rightarrow \lambda,\ S \Rightarrow a,\ S \Rightarrow b$) y las cadenas generadas son palíndromos, los de menor longitud sobre el alfabeto $V = \{a, b\}$. En un paso, aplicando las otras reglas se tienen las formas sentenciales aSa o bSb.
   Si se sigue derivando S para obtener formas sentenciales en dos o más pasos se cumple que cuando se sustituye S por aSa o por bSb se obtiene una forma sentencial de longitud mayor que se lee igual de izquierda a derecha que de derecha a izquierda.
   Entonces todas las formas sentenciales que no son sentencias son de la forma zSz^R, donde $|z| \geq 1$. Para terminar en una cadena de terminales se tienen que aplicar una de las tres reglas:

$S \to a \mid b \mid \lambda$. Por tanto, todas las derivaciones posibles son: $S \Rightarrow^* zSz^R \Rightarrow zaz^R$, o bien, $S \Rightarrow^* zSz^R \Rightarrow zbz^R$, o bien $S \Rightarrow^* zSz^R \Rightarrow zz^R$.
Luego, por todas las derivaciones en un paso y en dos o más pasos, se deduce que todas las sentencias w son de tipo palíndromo y también podemos comprobar que los palíndromos de cualquier longitud son derivables de S, según se ha visto en las derivaciones.
CONCLUSIÓN: Tenemos que $L(G) = L_{pal}$ donde L_{pal} se describe informalmente como el lenguaje formado por todos los palíndromos sobre el alfabeto $\{a, b\}$. Definimos formalmente L_{pal} como:

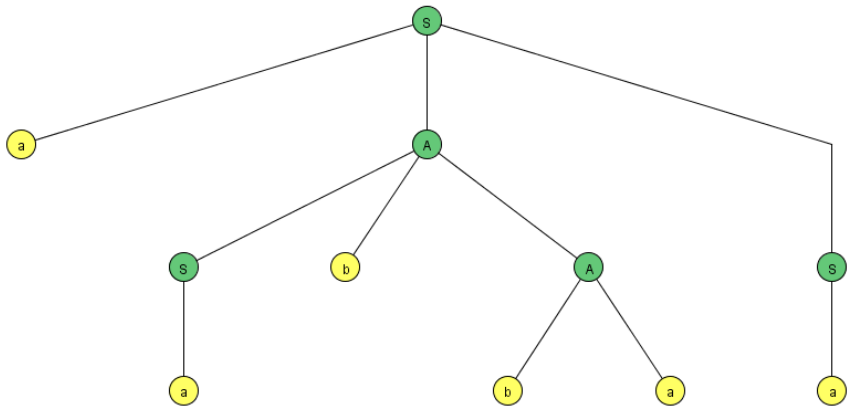
$$L_{pal} = \{w \in \{a, b\}^* \mid w = w^R\}$$

2. Dada la gramática libre del contexto con reglas: $S \to aAS \mid a, \quad A \to SbA \mid SS \mid ba$, obtener el árbol de derivación para la sentencia *aabbaa* e indicar de qué variable deriva la subcadena subrayada *aabbaa*.

   Solución: podemos obtener directamente el árbol o fijarnos en una derivación y a partir de ahí obtener el árbol. Por ejemplo:

$$S \Rightarrow aAS \overset{A \to SbA}{\Rightarrow} aSbAS \overset{S \to a}{\Rightarrow} aabAS \overset{A \to ba}{\Rightarrow} aabbaS \overset{S \to a}{\Rightarrow} aabbaa$$

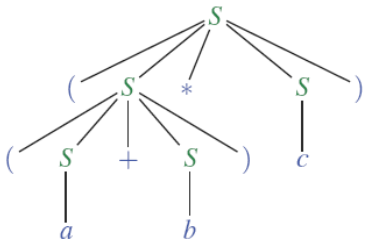
   El árbol de derivación para la sentencia *aabbaa* es:



   Podemos generar el árbol con JFLAP accediendo a *Grammar*, editando la gramática (una regla por línea) y seleccionando *Input→Brute force parse*. Se escribe la cadena de entrada y genera el árbol (*noninverted tree*).

   Para ver de qué variable deriva la subcadena subrayada *aabbaa* podemos observar en el árbol que es derivable de la variable A, es decir, $A \Rightarrow^* abba$ porque concatenando las hojas del subárbol que parte de A en el primer nivel se obtiene la subcadena *abba*.

3. Sea G una GLC y sea $((a + b) * c)$ una de sus sentencias. A continuación se muestra el árbol de derivación para $((a + b) * c)$:



   a) Obtén una derivación más a la izquierda de la sentencia $((a + b) * c)$ a partir del árbol de derivación.

Solución: la derivación más a la izquierda es la que aplica en cada paso la regla correspondiente a la variable más a la izquierda de la forma sentencial que se está derivando. El árbol refleja la aplicación de las reglas sin tener en cuenta el orden, pero considerando el orden más a la izquierda en la derivación tenemos que:

$$S \Rightarrow (S * S) \Rightarrow ((S + S) * S) \Rightarrow ((a + S) * S) \Rightarrow ((a + b) * S) \Rightarrow ((a + b) * c)$$

b) En el árbol aparecen aplicadas todas las reglas de la gramática. Describe los 4 componentes de la gramática.

Solución: teniendo en cuenta cómo se construye un árbol de derivación para una cadena y la hipótesis de partida del enunciado deducimos que la gramática es $G = (V_N, V_T, S, P)$, donde:

- S es el símbolo inicial de la gramática, pues aparece en la raíz del árbol.
- $V_T = \{a, b, c, (,), *, +\}$ pues en las hojas del árbol para una sentencia sólo hay símbolos terminales y esos son los únicos símbolos terminales que aparecen en el árbol.
- $V_N = \{S\}$, pues la única variable que aparece en un nodo no hoja de este árbol es el símbolo inicial de la gramática.
- Las reglas que se usan en este árbol (todas las posibles según el enunciado) son:

$$S \to (S * S) \mid (S + S) \mid a \mid b \mid c$$

Puede comprobarse que con estas reglas se generan cadenas que representan expresiones aritméticas con operador de suma y producto y con paréntesis obligatorios para subexpresiones con operadores. Los paréntesis obligatorios aseguran la no ambigüedad de las sentencias. Los símbolos terminales a, b, c representan tres tipos diferentes de tokens, como podrían ser variables, constantes reales y constantes enteras.

4. Consideramos el lenguaje PB de los paréntesis balanceados, que se define recursivamente de la siguiente forma:

   1. Base: () ∈PB
   2. Si w ∈PB entonces (w) ∈PB
   3. Si x, y ∈PB entonces xy ∈PB

   Obtener una gramática G_{PB} que genere el lenguaje de los paréntesis balanceados, justificando por qué se incluye cada regla.

   Solución: primero observamos que, por el caso base de la definición, la cadena más simple de paréntesis balanceados es (). La regla 2 establece que si encerramos entre paréntesis una cadena de paréntesis balanceados se obtiene una cadena de paréntesis balanceados. La regla 3 dice que si concatenamos dos cadenas de paréntesis balanceados el resultado es una cadena de paréntesis balanceados. Ahora obtenemos reglas gramaticales que se deducen de cada caso de la definición recursiva, teniendo en cuenta que usamos la variable S para representar a cualquier cadena de paréntesis balanceados.

   a) Puesto que $() \in PB$ necesitamos incluir la regla $S \to ()$ para que también se cumpla que $S \Rightarrow^* ()$, o sea $() \in L(G_{PB})$.

   b) Supuesto que $w \in PB$, y se supone que equivale a $S \Rightarrow^* w$, entonces sabemos que $(w) \in PB$. Por eso tenemos que incluir la regla $S \to (S)$, para que $S \Rightarrow (S) \Rightarrow^* (w)$.

   c) Supuesto que $x, y \in PB$, y se supone que equivale a $S \Rightarrow^* x$ y $S \Rightarrow^* y$, entonces sabemos que $xy \in PB$. Por eso tenemos que incluir la regla $S \to SS$, para que $S \Rightarrow SS \Rightarrow^* xS \Rightarrow^* xy$.

La gramática G_{PB} sería pues: $S \to () \mid (S) \mid SS$ y esta gramática es correcta para describir al lenguaje PB porque sus reglas se han obtenido conforme a la definición recursiva. Con eso también hemos probamos que el lenguaje de los paréntesis balanceados es un lenguaje libre del contexto, al ser generado por una GLC.

5. Sea la gramática G con reglas $S \to \lambda \mid A,\ A \to AA \mid c$. Obtener una gramática equivalente que sea regular.

   Solución: primero tenemos que analizar esta gramática para deducir cuál es el lenguaje $L(G)$. Para ello tenemos que comprobar qué cadenas son derivables de A y de S.

   Tenemos que $A \Rightarrow^* c^n$, con $n \geq 1$, por aplicación de la regla $A \to c$ (caso $n = 1$), o bien $n \geq 1$ veces la regla $A \to AA$ y luego sustituyendo cada A por c con la regla $A \to c$ (caso $n \geq 2$).
   Entonces, a partir de S tenemos las siguiente posiblidadades:
   (1) $S \Rightarrow \lambda = c^0$, o bien, (2) $S \Rightarrow A$ y puesto que ya hemos comprobado que $A \Rightarrow^* c^n$, con $n \geq 1$, se deduce que $S \Rightarrow^* c^n$, con $n \geq 1$. Por (1) y (2) se tiene finalmente que $S \Rightarrow^* c^n$, con $n \geq 0$, por lo que $L(G) = \{c^n \mid n \geq 0\}$.

   Ahora hay que encontrar una gramática regular, que llamamos G_r, que genere $L(G)$ y es sencillo pues no hay más que considerar las reglas $S \to \lambda \mid cS$, que cumplen las restricciones que requiere una gramática regular. Y podemos comprobar fácilmente que $L(G_r) = \{c^n \mid n \geq 0\} = L(G)$.

6. Dada la siguiente GLC con reglas: $\begin{cases} S \to aSbS \mid A \mid \lambda \\ A \to B \mid bC \mid \lambda \\ B \to AC \mid dd \\ C \to bA \end{cases}$

   obtener una gramática propia equivalente, aplicando todos los pasos del algoritmo de transformación a gramática propia.

   Solución: una gramática propia no debe tener λ-reglas, ni ciclos ni símbolos inútiles. Hay que aplicar los algoritmos de transformación a gramática λ-libre, eliminación de reglas unitarias y eliminación de símbolos inútiles (al principio y al final).

   La gramática G de partida no tiene símbolos inútiles, así que pasamos a obtener una **gramática λ-libre equivalente**. Obtenemos que $Vanu = \{S, A\}$, eliminamos λ-reglas y añadimos otras a cambio y se obtiene G_2:

   $$\begin{array}{l} S' \to S \mid \lambda \\ S \to aSbS \mid aSb \mid abS \mid ab \mid A \\ A \to B \mid bC \\ B \to AC \mid C \mid dd \\ C \to bA \mid b \end{array}$$

   donde S' es el nuevo símbolo inicial, necesario para permitir que $\lambda \in L(G_2)$ sin violar la condición de que el símbolo inicial no aparezca en la parte derecha de ninguna regla.

   Puede comprobarse que la gramática G_2 no tiene ciclos, pues no se da ninguno de estos cuatro casos: $S \Rightarrow^* S$, o $A \Rightarrow^* A$, $B \Rightarrow^* B$, $C \Rightarrow^* C$. También puede comprobarse que todos los símbolos son útiles. Así pues la gramática ya es propia. No obstante, como el enunciado pide la aplicación de todos los pasos del algoritmo tenemos que aplicar ahora el algoritmo que elimina las reglas unitarias a la gramática G_2.

   Eliminamos reglas unitarias y añadimos nuevas. Tenemos que:

   $$V_{uni}(S') = \{S, A, B, C\}, V_{uni}(S) = \{A, B, C\}, V_{uni}(A) = \{B, C\}, V_{uni}(B) = \{C\}$$

Una vez eliminadas las unitarias se añaden las necesarias teniendo en cuenta estos conjuntos y se obtiene la gramática G_3 sin unitarias:

$$\begin{array}{l} S' \to R_S \mid \lambda \\ S \to aSbS \mid aSb \mid abS \mid ab \mid bC \mid AC \mid dd \mid bA \mid b \\ A \to bC \mid AC \mid dd \mid bA \mid b \\ B \to AC \mid dd \mid bA \mid b \\ C \to bA \mid b \end{array}$$

donde R_S se refiere a todas las partes derechas de las reglas para S, tal y como aparece en la fila de las reglas de S.

Finalmente aplicamos a G_3 el algoritmo que **elimina símbolos inútiles** y se obtiene otra gramática propia equivalente:

$$\boxed{\begin{array}{l} S' \to R_S \mid \lambda \\ S \to aSbS \mid aSb \mid abS \mid ab \mid bC \mid AC \mid dd \mid bA \mid b \\ A \to bC \mid AC \mid dd \mid bA \mid b \\ C \to bA \mid b \end{array}}$$

donde se ha eliminado la variable B (y las reglas donde aparece) porque esta variable ha quedado inaccesible tras la eliminación de reglas unitarias.

7. Obtén una gramática λ-libre equivalente a la siguiente: $\begin{cases} S \to ABb \mid ABC \\ A \to aA \mid \lambda \\ B \to bB \mid \lambda \\ C \to AB \end{cases}$

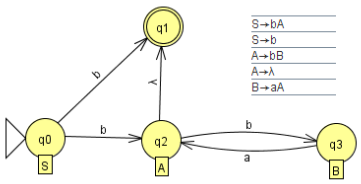
Solución: el conjunto de variables anulables es: $Vanu = \{S, A, B, C\}$ y en el proceso de eliminar y generar reglas se tiene (se omite el desarrollo):

$$\boxed{\begin{array}{l} S \to \lambda \\ S \to ABb \mid ABC \mid Ab \mid Bb \mid b \mid AB \mid AC \mid BC \mid A \mid B \mid C \\ A \to aA \mid a \\ B \to bB \mid b \\ C \to AB \mid A \mid B \end{array}}$$

Se ha añadido $S \to \lambda$ puesto que S es anulable y S no aparece en la parte derecha.

8. Dada la gramática G con reglas:
$$S \to bA \mid b, \quad A \to bB \mid \lambda, \quad B \to aA$$
Obtener un AF que acepte el lenguaje generado por esta gramática aplicando el algoritmo $GRtoAF$.

Solución: en la siguiente figura se muestra M tal que $L(G) = L(M)$ obtenido con Jflap:



7.2. Problemas propuestos

1. Obtén una GLC para cada uno de los siguientes lenguajes:

   a) $\{a^n b^{2n} \mid n \geq 0\}$

   b) $\{a^n b^m \mid m > n \geq 0\}$

   c) $\{0^k 1^m 2^n \mid n = k + m,\ k, n \geq 0\}$

2. Sea un lenguaje de declaraciones de funciones simple, que llamamos $DFUN$. Por ejemplo, la cadena ***F-aaa[b1;aa10101]*** se considera una declaración de función sintácticamente correcta en este lenguaje. Cada carácter que aparece en la cadena es un símbolo y no hay más símbolos aparte de esos. El lenguaje lo definimos mediante las siguientes reglas sintácticas informales:

   1: Una **declaración de función** comienza por el prefijo ***F-***, seguida de un identificador y de los parámetros. Todos los símbolos van seguidos, no se permiten espacios.

   2: Un **identificador** empieza por la letra **a** ó por la letra **b** seguida de una secuencia opcional de símbolos que pertenecen al conjunto $\{a, b, 0, 1\}$.

   3: Los **parámetros** se escriben entre corchetes y consisten en un único identificador, o bien, una serie de identificadores separados por un punto y coma. Se entiende que el ';' no puede aparecer al principio o al final de la cadena entre corchetes y tampoco es admisible que puedan aparecer varios símbolos ; seguidos.

   Obtén una gramática libre del contexto G que genere el lenguaje $DFUN$. Para ello indica las reglas gramaticales que deben introducirse según cada regla informal de la definición anterior. Usar un nombres significativos entre ángulos para las variables, en consonancia con el tipo de subcadenas derivables de esa variable. Se permite un máximo de 4 variables en esta gramática.

3. Una expresión regular limitada (abreviadamente ERL) se define recursivamente como:

   1: (Base) Las constantes **0** y **1** son ERL

   2: Si R_1 y R_2 son dos ERL entonces $R_1 R_2$ es una ERL

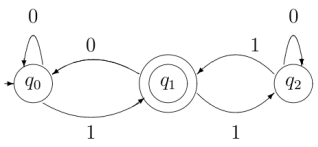
   3: Si R_1 y R_2 son dos ERL entonces $R_1|R_2$ es una ERL

   4: Si R es una ERL entonces R^* es una ERL

   5: Si R es una ERL entonces (R) es una ERL

   Por ejemplo, 0*101 es una ERL. Obtén una gramática libre del contexto que sirva para describir formalmente la sintaxis del lenguaje de expresiones regulares restringidas. Describe los 4 componentes de la gramática.

4. Obtener una gramática regular G que genere el lenguaje $L(M)$, donde M es:



5. Obtén una gramática regular para cada uno de los dos lenguajes siguientes:

$$L_1 = \{ab^n a \mid n \geq 0\} \quad \| \quad L_2 = \{0^n 1 \mid n \geq 0\}$$

6. Encuentra gramáticas para generar los lenguajes L_1L_2 y $L_1 \cup L_2$ y $(L_1 \circ L_2)^*$ del ejercicio anterior.

7. El fichero *propuesto-G-a-propia.jff* de *tema4-jflap* contiene la siguiente gramática:

$$\begin{array}{l} S \to aCS \mid AB \\ A \to SA \mid \lambda \mid c \\ B \to BAb \mid b \\ C \to CaA \end{array}$$

   Muestra un **árbol de derivación** para la **cbb**. Obtén una **gramática propia** equivalente (puede resolverse a mano y comparar el resultado con el que obtiene Jflap).

8. Obtén una gramática sin reglas unitarias equivalente a: $\begin{cases} E \to E + T \mid T \\ T \to T * F \mid F \\ F \to i \mid (E) \end{cases}$

9. (!) Obtén una GLC para cada uno de los siguientes lenguajes y justifica que la gramática obtenida es correcta, razonando por qué se incluye cada regla:

   a) $\{a^i b^j c^k \mid (i \neq j) \vee (j \neq k)\}$

   b) Consideremos las cadenas que consisten en paréntesis y corchetes balanceados. Estas expresiones aparecen inmersas en otras más complejas en algunos lenguajes de programación como C, por ejemplo: $f(a[i] * (b[i][j], c[g(x)]), d[i])$. Si eliminamos todo menos los paréntesis y los corchetes se obtiene la cadena de paréntesis y corchetes balanceados: ([](([][][()])[]). Se pide una gramática para el lenguaje formado por las cadenas de paréntesis y corchetes balanceados.

10. (!) Dada la gramática G con reglas: $S \to AA, \quad A \to aSa \mid a$

    a) Demuestra que G es ambigua.

    b) Demuestra que existe una gramática equivalente a G que no es ambigua.
    Indicación: es necesario deducir el lenguaje $L(G)$. Una vez descubierto el conjunto de cadenas que genera la gramática se procede a mostrar una nueva gramática equivalente G' sin ambigüedad. Muestra un argumento convincente de que la gramática obtenida es equivalente a la original y no es ambigua.

7.3. Preguntas tipo test

Indica si las siguientes afirmaciones son verdaderas o falsas. Inténtese justificar la verdad o falsedad de cada afirmación.

1. La gramática cuyas reglas son $S \to \lambda \mid aSa \mid bSb$ es correcta para generar el lenguaje de los palíndromos con alfabeto $\{a, b\}$.

2. Si un lenguaje L es libre del contexto entonces también lo es el lenguaje reflejo L^R.

3. Si G es λ-libre entonces $\lambda \notin L(G)$.

4. Si G no tiene reglas unitarias entonces es libre de ciclos.

5. Cualquier lenguaje regular puede ser generado por una gramática libre del contexto.