ANÁLISIS COMPARATIVO DE LENGUAJES

M. Cs. Patricia Roggero

Universidad Nacional de San Luis

Año 2021

Aspectos Formales de los Lenguajes de Programación – Segunda Parte

Notas de Clase

Capítulos III – Programming Languages – Design and Implementation – Terrence Pratt

Capítulos III Concepts of Programming Languages – Robert Sebesta

Apunte de la cátedra.

Gramáticas

Gramáticas: es un modelo matemático que permite generar a través de reglas sintácticas o gramaticales, cadenas miembros de un leguaje específico.

Formalmente:

Definición: formalmente una gramática G es una 4-upla $G = (N, \Sigma, P, S)$ donde:

$N =$ es el conjunto finito de símbolos no terminales.

$\Sigma =$ es el conjunto finito de símbolos terminales o alfabeto.

$P =$ es el conjunto finito de reglas o producciones,

$P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$

$S =$ es un símbolo especial llamado símbolo distinguido.

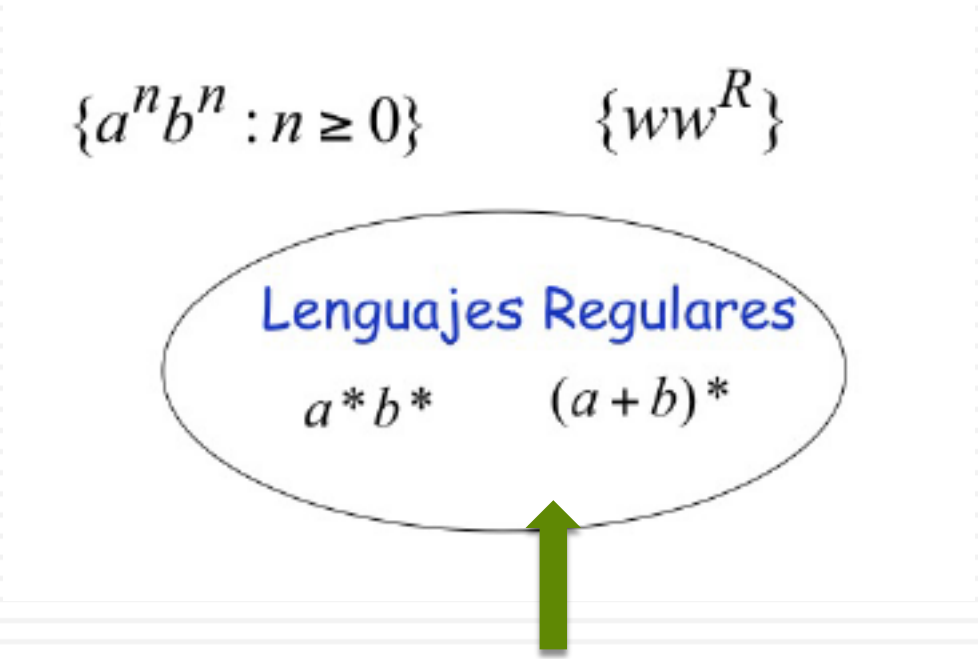
Las restricciones impuestas al conjunto P, dan origen a diferentes tipos de gramáticas, para generar los diferentes tipos de lenguajes (Jerarquía de Chomsky).

¿Qué gramáticas nos interesan?

- Las gramáticas que nos permitan describir lenguajes de programación.
- Los lenguajes de programación, recordando la Jerarquía de Chomsky, son lenguajes Tipo 2 o Libres del Contexto.

Intuitivamente

- Veamos la siguiente gráfica:

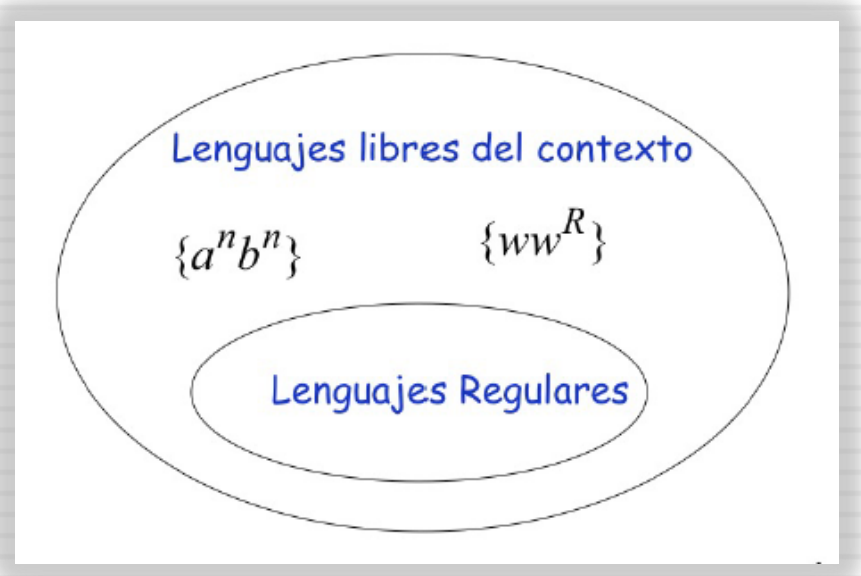


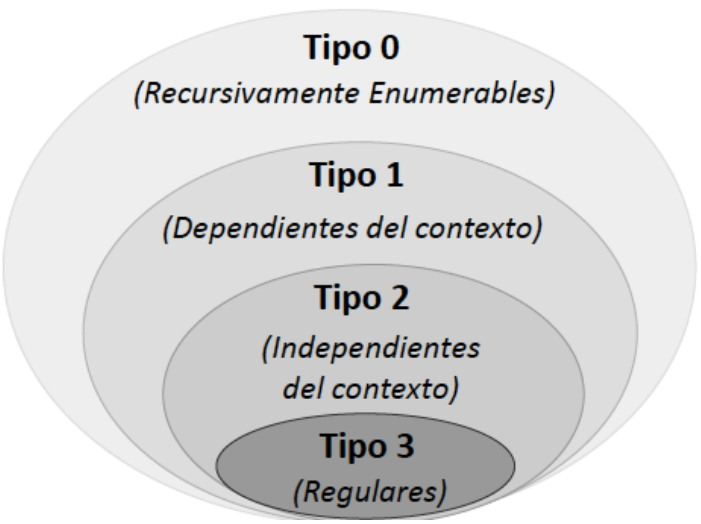
¿Estos lenguajes, serán regulares?

Estos lenguajes son los denotados por Expresiones Regulares

Intuitivamente

- Estos lenguajes se denominan Lenguajes Libres del Contexto (LLC):





Lenguajes Libres del Contexto

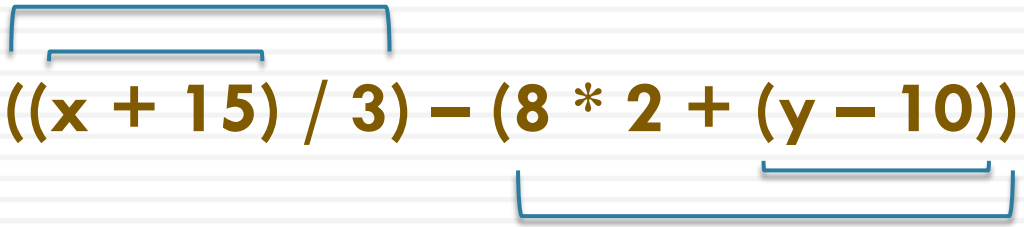
- **Características:**

  - Estos lenguajes pueden ser generados por gramáticas libres del contexto, la cual es una *notación recursiva natural* para este tipo de lenguajes.
  - La principal característica de este tipo de lenguajes es que las cadenas contienen símbolos en modalidad espejo, aunque también se engloban las características propias de los lenguajes Tipo 3 (recordar $L_3 \subset L_2$).

Lenguajes Libres del Contexto

- *En los siguientes ejemplos, propios de los lenguajes de programación y de las matemáticas, podemos apreciar esta característica de anidamientos, sincronización o modalidad espejo:*

- Uso de paréntesis en expresiones aritméticas:



- Uso de paréntesis y corchetes en expresiones tipo C:

Lenguajes Libres del Contexto

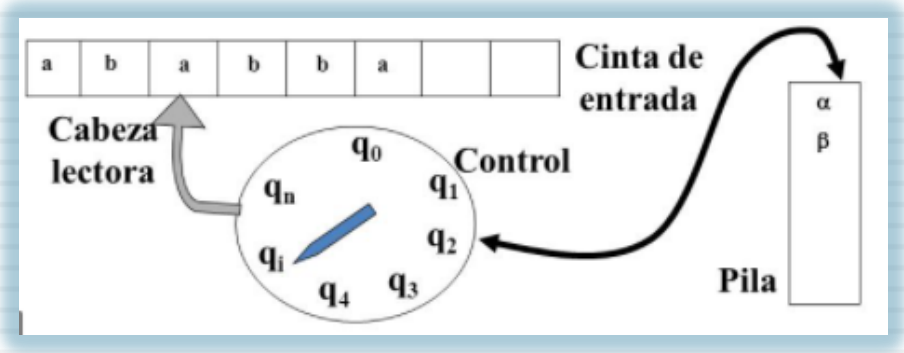
¿Qué tipo de dispositivos permiten describir estos lenguajes?

Gramática Libre del Contexto (GLC)

$$G = (N , \Sigma , P , S)$$

- N: Conjunto de símbolos No Terminales
- Σ: Conjunto de símbolos Terminales o alfabeto
- P: Conjunto de producciones
- S: Símbolo de comienzo o distinguido

Autómata Push-Down (APD)



Gramáticas Libres del Contexto (GLC)

Ejemplo: el lenguaje inglés

$\langle sentence \rangle \rightarrow \langle noun\_phrase \rangle \ \langle predicate \rangle$

$\langle noun\_phrase \rangle \rightarrow \langle article \rangle \ \langle noun \rangle$

$\langle predicate \rangle \rightarrow \langle verb \rangle$

$\langle article \rangle \rightarrow a$

$\langle article \rangle \rightarrow the$

$\langle noun \rangle \rightarrow cat$

$\langle noun \rangle \rightarrow dog$

$\langle verb \rangle \rightarrow runs$

$\langle verb \rangle \rightarrow walks$

Originalmente las GLC fueron concebidas por Noam Chomsky para describir lenguajes naturales. Esta idea fue utilizada, posteriormente, en el contexto de las Cs. de la Computación.

$$\begin{aligned}
\langle sentence \rangle &\Rightarrow \langle noun_phrase \rangle \ \langle predicate \rangle \\
&\Rightarrow \langle noun_phrase \rangle \ \langle verb \rangle \\
&\Rightarrow \langle article \rangle \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow the \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow the \ dog \ \langle verb \rangle \\
&\Rightarrow the \ dog \ walks
\end{aligned}$$

Usando las reglas se obtiene la oración **"the dog walk"**

Gramáticas Libres del Contexto (GLC)

Definición: Una gramática $G = (N, \Sigma, P, S)$ es libre de contexto (GLC), si sus producciones en P tienen la siguiente forma: $A \rightarrow \alpha$, con $A \in N$ y $\alpha \in (N \cup \Sigma)^*$.

El nombre *"libre de contexto"* se debe a que cada una de las producciones pueden ser aplicadas independientemente del contexto en donde aparezca un no terminal en una *forma sentencial*.

Su importancia radica en que permiten describir los aspectos sintácticos de los lenguajes de programación. Por lo tanto tienen un rol central en el contexto de compiladores.

Las GLC también son llamadas Gramáticas Tipo 2.

Gramáticas Libres del Contexto (GLC)

- Se utilizan las letras mayúsculas para expresar los símbolos no terminales.
- Para los símbolos del alfabeto Σ se usan las letras minúsculas, números, símbolos en general que pueden ser delimitadores, símbolos de puntuación, etc.
- El símbolo distinguido forma parte del conjunto de no terminales.

GLC

Gramática: $S \rightarrow aSb$

$S \rightarrow \lambda$

Derivación de la sentencia: ab

$$S \Rightarrow aSb \Rightarrow ab$$

$S \rightarrow aSb$ $\qquad$ $S \rightarrow \lambda$

$L = \{a^n b^n / n \geq 0\}$

Derivación de la sentencia: $aabb$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

$S \rightarrow aSb$ $\qquad$ $S \rightarrow \lambda$

El símbolo → se utiliza para expresar las reglas, en forma práctica y se lee produce.

El símbolo $\Rightarrow$ *se lee deriva. Representa la relación deriva, que permite derivar cadenas a partir del símbolo distinguido y aplicando las reglas de la GLC.*

Ejemplo GLC

Construir una GLC que genere el lenguaje $L = \{ww^R \ / \ w \in \{a, b\}^*\}$, observemos que las cadenas que pertenecen a este lenguaje tienen la forma:

$\underbrace{abb}_{w}\ \underbrace{bba}_{w^R},\ \underbrace{babb}_{w}\ \underbrace{bbab}_{w^R},\ \underbrace{baa}_{w}\ \underbrace{aab}_{w^R},\ \underbrace{baabbb}_{w}\ \underbrace{bbbaab}_{w^R}$, etc.

Para construir una gramática que genere este lenguaje, debemos observar como se aparean los símbolos, si miramos la primer cadena vemos que la primera *a* se corresponde con la última *a*, la *b* que sigue lo hace con la *b* anterior a la última *a*, y así siguiendo. De manera similar ocurre en cualquier cadena que pertenece a este lenguaje, corroborarlo.

Ejemplo (Cont.)

Observemos la sincronización que se da en las cadenas de este lenguaje:

Entonces una gramática que genere este lenguaje puede ser la siguiente:

$$G = \langle \{S\}, \{a, b\}, P, S \rangle$$
$$P = \{S \rightarrow aSa \mid bSb \mid \lambda\}$$

Determinar, usando la relación deriva ($\Rightarrow$), si la gramática construida, genera las cadenas miembros del lenguaje.

Ejemplos GLC

Construir una GLC que genere el lenguaje $L = \{w \in \{a, b\}^* \ / \ |w|_a = |w|_b\}$, observemos que las cadenas que pertenecen a este lenguaje tienen la forma: *aaabbbababb*, *aabb*, *bababababa*, *bbaaaabb*, etc.

Para pensar como construir una GLC que genere este lenguaje tenemos que observar que, cada vez que se genere una *a* debe generarse una *b* y viceversa, el número de veces que querramos, por lo tanto la gramática podría ser:

Ejemplo (Cont.)

$$G = \langle \{S\}, \{a, b\}, P, S \rangle$$
$$P = \{S \rightarrow aSb \mid bSa \mid \lambda\}$$

Determinar, usando la relación deriva, si la gramática construida, genera las cadenas miembros del lenguaje.

Genera todas las cadenas posibles que pertenecen al lenguaje?

Por ejemplo, permite generar cadenas como: **abba, aabbbbaa, baab, etc.** ? No, no se pueden generar cadenas como estas, por lo tanto es necesario revisar la gramática y obtener las reglas Que permitan generar todas las cadenas que pertenecen al lenguaje.

Ejemplo (Cont.)

Posibles soluciones:

1)

$S \rightarrow aSbS \mid bSaS \mid \lambda$

2)

$S \rightarrow aSb \mid bSa \mid SS \mid \lambda$

Corroborarlo!

Ejercicios

Construir una GLC para cada uno de los siguientes lenguajes:

- $L_1 = \{a^n\ b^m\ c^{n+m} \ / \ n, m \geq 0\} = \{\lambda,$ abbccc, ac, bc, aabccc, ….$\}$

  S -> aSc / C

  C -> bCc / λ

- $L_2 = \{a^i\ b^k\ c\ b^k\ a^i \ / \ i, k > 0\}$

Derivaciones en una GLC

Las producciones de una gramática se utilizan para inferir que ciertas cadenas están en el lenguaje generado por dicha gramática.
Como ya vimos, el proceso de derivar cadenas requiere de la definición de la **relación deriva** $\Rightarrow$.

Lenguaje generado por una GLC

Sea $G = (N, \Sigma, P, S)$ una GLC, entonces cualquier cadena $\alpha \in (N \cup \Sigma)^*$ tal que $S \overset{*}{\Rightarrow} \alpha$ es una forma sentencial. Notar que el lenguaje $L(G)$ está formado por las formas sentenciales que están en Σ^*, es decir que consisten solamente de símbolos terminales.

Lenguaje generado por una GLC

El lenguaje generado por una gramática $G = (N, \Sigma, P, S)$ (denotado $L(G)$) es: $L(G) = \{w \in \Sigma^* / S \overset{*}{\Rightarrow} w\}$.

Criterios de selección

Para obtener las cadenas generadas por una gramática a partir de la aplicación de sus reglas, es posible tomar la decisión de hacerlo en base a 2 criterios:

- Elegir siempre el no terminal de más a la izquierda. La forma sentencial así obtenida se denomina forma sentencial izquierda.

- Elegir siempre el no terminal de más a la derecha. La forma sentencial así obtenida se denomina forma sentencial derecha.

Ejemplo criterios de selección

Dada la siguiente gramática:

$$E \rightarrow E + E | E * E | (E) | I$$
$$I \rightarrow a | b | c$$

Obtener una secuencia de derivaciones de más a la izquierda y una de más a la derecha respectivamente para la cadena **w = a + b * c:**

Derivación de más a la izquierda:

$$E \Rightarrow \underline{E} + E \Rightarrow \underline{I} + E \Rightarrow a + \underline{E} \Rightarrow a + \underline{E} * E \Rightarrow a + \underline{I} * E \Rightarrow$$
$$a + b * \underline{E} \Rightarrow a + b * \underline{I} \Rightarrow a + b * c$$

Derivación de más a la derecha:

$$E \Rightarrow E + \underline{E} \Rightarrow E + E * \underline{E} \Rightarrow E + E * \underline{I} \Rightarrow E + \underline{E} * c \Rightarrow E + \underline{I} * c \Rightarrow$$
$$\underline{E} + b * c \Rightarrow \underline{I} + b * c \Rightarrow a + b * c$$

Árbol de derivación o sintáctico

Es un recurso alternativo para representar las derivaciones de las gramáticas Tipo 2. Este es un concepto muy importante en el contexto de análisis sintáctico (etapa importante en el proceso de compilación) de lenguajes de programación.

Definición: Sea $G = (N, \Sigma, P, S)$ una GLC, luego un árbol es un *Árbol de Derivación* para G si:

Árbol de derivación

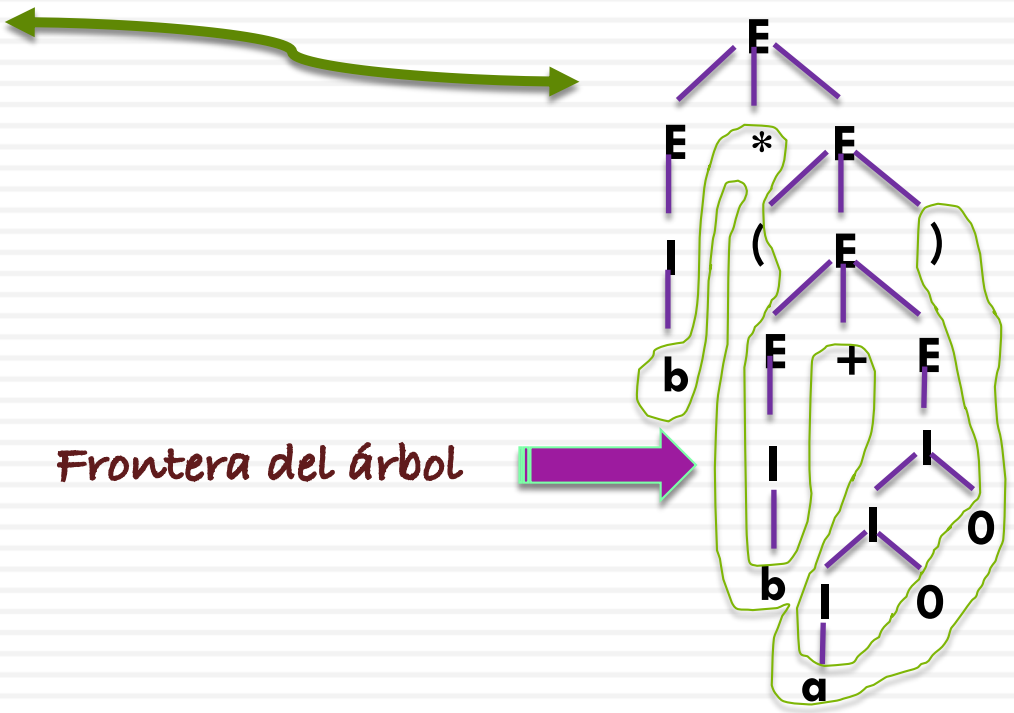
1. Cada nodo interior está rotulado por un símbolo no terminal.
2. Cada hoja es rotulada por un símbolo de $(N \cup \Sigma) \cup \{\lambda\}$. No obstante si el nodo n tiene rótulo λ, luego n es una hoja y es el único descendiente de su padre. (Para las producciones del tipo $A \rightarrow \lambda$)
3. Si un nodo interior n tiene rótulo A y los vértices $n_1, n_2, ..., n_k$ son los descendientes de n, de izquierda a derecha, con rótulos $X_1, X_2, ..., X_k$ respectivamente, luego $A \rightarrow X_1 X_2 ... X_k \in P$

Ejemplo Árbol de derivación

Dada una gramática que genera ciertas expresiones aritméticas

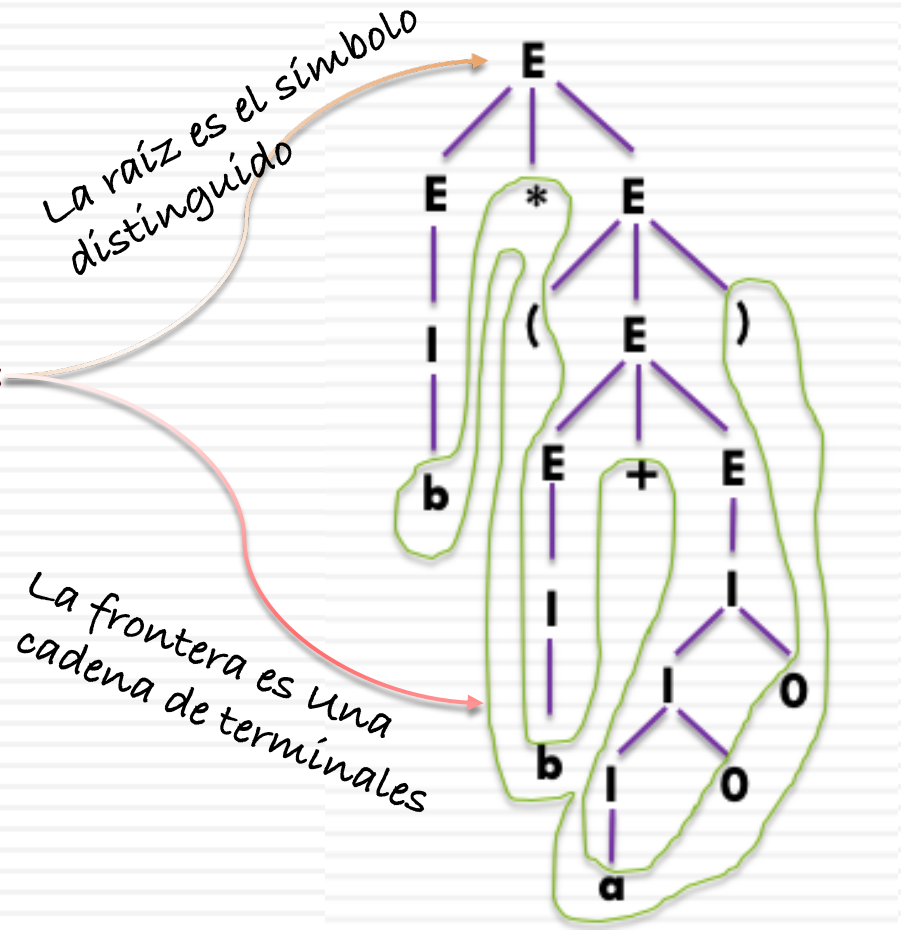
$$E \rightarrow I \mid E + E \mid E * E \mid (E)$$
$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

y la cadena **w = b * (b + a00)**, obtenemos el siguiente Árbol de Derivación:



Árbol de derivación

Los árboles de derivación que nos importan son aquellos en los que:



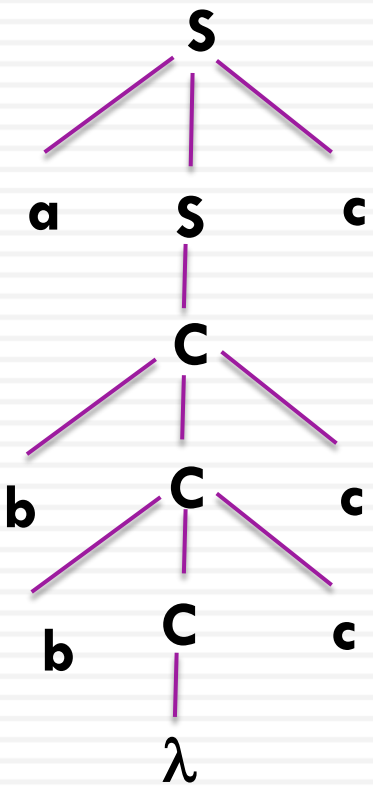
Ejemplo: construir el árbol de derivación para w = abbccc

Dada la siguiente GLC:

S -> aSc / C

C -> bCc / λ

genera → $L_1 = \{a^n b^m c^{n+m} / n, m \geq 0\}$



Backus Naur Form (BNF)

Las GLC se escriben frecuentemente utilizando una notación conocida como BNF, que es un metalenguaje, porque se usa para describir otro lenguaje.

Este tipo de notación, es la técnica más común para describir lenguajes de programación.

En esta notación se utilizan comúnmente las siguientes convenciones:

- Los no terminales se escriben entre paréntesis angulares <>.
- Los terminales se representan con cadenas de caracteres sin corchetes angulares.
- El lado izquierdo de cada regla debe tener únicamente un no terminal (ya que es una gramática libre del contexto).
- El símbolo ::=, que se lee se define como o se reescribe como, se utiliza en lugar de →.

Ejemplos

BNF para el lenguaje cuyas cadenas son números:

```
<number> ::= <digit> | <number> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

BNF para el lenguaje cuyas cadenas consisten en paréntesis anidados:

< *cadena* >::=< *cadena* >< *parentesis* > / < *parentesis* >
< *parentesis* >::= (< *cadena* >)/()

< *cadena* >::=< *cadena* >< *parentesis* > / < *parentesis* >
< *parentesis* >::= (< *cadena* >)/()

<cadena> => <cadena> <parentesis> =>

<parentesis> <parentesis> => (<cadena>) <parentesis> =>

(<cadena> <parentesis>) <parentesis> =>

(<parentesis> <parentesis>) <parentesis> =>

(() <parentesis>) <parentesis> => (() ()) <parentesis> =>

(() ()) ()

Ejemplos (Cont.)

$$
\begin{aligned}
&< \textit{expresion} >::=< \textit{expresion} > + < \textit{term} > \mid < \textit{term} > \\
&< \textit{term} >::=< \textit{term} > * < \textit{factor} > \mid < \textit{factor} > \\
&< \textit{factor} >::= (< \textit{expresion} >) \mid < \textit{iden} > \mid < \textit{entero} > \\
&< \textit{iden} >::=< \textit{letra} > \mid < \textit{iden} >< \textit{letra} > \mid < \textit{iden} >< \textit{digito} > \\
&< \textit{entero} >::=< \textit{digito} > \mid < \textit{entero} >< \textit{digito} > \\
&< \textit{letra} >::= A|B|...|Z \\
&< \textit{digito} >::= 0|1|2|...|9
\end{aligned}
$$

¿Qué cadenas permite describir esta BNF?

Podemos usar árboles de derivación o la relación deriva para daterminar que cadenas se pueden generar.

Ejemplo BNF para un lenguaje de programación hipotético

```
<Programa>                ::= <ListaDeFunciones>
<ListaDeFunciones>        ::= <Función> | <ListaDeFunciones> <Función>
<Función>                 ::= FUNC <Variable> ( <ListaDeParámetros> ) <Sentencia>
<ListaDeParámetros>       ::= <ListaDeVariables> | ε
<ListaDeVariables>        ::= <Variable> | <ListaDeVariables> , <Variable>
<Variable>                ::= <Letra> | <Variable> <Alfanumérico>
<Alfanumércio>            ::= <Letra> | <Dígito>
<Letra>                   ::= a | b | ... | y | z
<Dígito>                  ::= 0 | 1 | ... | 8 | 9
<Sentencia>               ::= <SentenciaDeAsignación> | <SentenciaDeRetorno> |
                              <SentenciaDeImpresión> | <SentenciaNula> |
                              <SentenciaCondicional> | <SentenciaWhile> | <Bloque>
<SentenciaDeAsignación>   ::= <Variable> := <Expresión>
<Expresión>               ::= <Expresión> <OperadorBinario> <Expresión> |
                              <OperadorUnario> <Expresión> | ( <Expresión> ) | <Entero> |
                              <Variable> | <Variable> (<ListaDeArgumentos> )
<OperadorBinario>         ::= + | − | * | /
<OperadorUnario>          ::= −
<Entero>                  ::= <Dígito> | <Entero> <Dígito>
<ListaDeArgumentos>       ::= <ListaDeExpresiones> | ε
<ListaDeExpresiones>      ::= <Expresión> | <Expresión> , <ListaDeExpresiones>
<SentenciaDeImpresión>    ::= PRINT <ListaDeImpresión>
<ListaDeImpresión>        ::= <ElementoDeImpresión> |
                              <ListaDeImpresión> , <ElementoDeImpresión>
<ElementoDeImpresión>     ::= <Expresión> | “<Texto> ”
<Texto>                   ::= <Carácter> | <Carácter> <Texto>
<Carácter>                ::= <CarácterImprimible> | <CarácterEscapado>
<CarácterImprimible>      ::= Cualquier carácter ASCII imprimible
<CarácterEscapado>        ::= \n
<SentenciaDeRetorno>      ::= RETURN <Expresión>
<SentenciaNula>           ::= CONTINUE
<SentenciaCondicional>    ::= IF <Expresión> THEN <Sentencia> FI |
                              IF <Expresión> THEN <Sentencia> ELSE <Sentencia> FI
<SentenciaWhile>          ::= WHILE <Expresión> DO <Sentencia> DONE
<Bloque>                  ::= { <ListaDeDeclaraciones> <ListaDeSentencias> }
<ListaDeDeclaraciones>    ::= <Declaración> <ListaDeDeclaraciones> | ε
<Declaración>             ::= VAR <ListaDeVariables>
<ListaDeSEntencias>       ::= <Sentencia> | <ListaDeSentencias> <Sentencia>
```

BNF Extendida (BNFE)

La BNFE surge a partir de algunas extensiones realizadas a la BNF, las cuales no afectan la potencia descriptiva, sino que incrementan su legibilidad y facilidad de escritura. Las extensiones son las siguientes:

- Si un elemento es opcional se encierra entre [ ].
  Ejemplo:
  < *selec* >::= *if*(< *exp* >) < *sent* > [*else* < *sent* >]

- Para la elección de alternativas se usa | y opcionalmente también se pueden usar ( ).
  Ejemplo:
  < *for* >::= *for* < *var* >:=< *expresion* > (*to*|*downto*)
  < *expresion* > *do* < *sentencia* >
- Una secuencia arbitraria se encierra entre {}.
  Ejemplo:
  < *lista – ident* >::=< *identificador* > {, < *identificador* >}

Ejemplo

BNF

$< entero - con - signo >::=< entero > | < signo >< entero >$

$< entero >::=< digito > | < digito >< entero >$

$< signo >::= +|-$

$< digito >::= 0|1|2|...|9$

BNF extendida

$< entero - con - signo >::= [+|-] < digito > \{< digito >\}$

Ejemplo

BNFE que permite describir el Lenguaje de programación LISP:

$$
\begin{aligned}
&\langle \textit{expressions} \rangle && ::= \langle \textit{atomic} - \textit{symbol} \rangle \mid \\
&&& (\langle \textit{expressions} \rangle . \langle \textit{expressions} \rangle) \mid \langle \textit{list} \rangle \\
&\langle \textit{list} \rangle && ::= \{(\langle \textit{expressions} \rangle)\} \\
&\langle \textit{atomic} - \textit{symbol} \rangle && ::= \langle \textit{letter} \rangle\langle \textit{atom} - \textit{part} \rangle \\
&\langle \textit{atom} - \textit{part} \rangle && ::= \langle \textit{letter} \rangle\langle \textit{atom} - \textit{part} \rangle \mid \\
&&& \langle \textit{digit} \rangle\langle \textit{atom} - \textit{part} \rangle \\
&\langle \textit{letter} \rangle && ::= a|b|...|z \\
&\langle \textit{digit} \rangle && ::= 0|1|...|9
\end{aligned}
$$

Ejemplo

```
<translation-unit> ::= {<external-declaration>}*

<external-declaration> ::= <function-definition>
                         | <declaration>

<function-definition> ::= {<declaration-specifier>}* <declarator> {<declaration>}* <compound-statement>

<declaration-specifier> ::= <storage-class-specifier>
                          | <type-specifier>
                          | <type-qualifier>

<storage-class-specifier> ::= auto
                            | register
                            | static
                            | extern
                            | typedef

<type-specifier> ::= void
                   | char
                   | short
                   | int
                   | long
                   | float
                   | double
                   | signed
                   | unsigned
                   | <struct-or-union-specifier>
                   | <enum-specifier>
                   | <typedef-name>

<struct-or-union-specifier> ::= <struct-or-union> <identifier> { {<struct-declaration>}+ }
                              | <struct-or-union> { {<struct-declaration>}+ }
                              | <struct-or-union> <identifier>

<struct-or-union> ::= struct
                    | union
```

¿DUDAS?