Sintaxis y Semántica del lenguaje.

Tema: Gramáticas Libres de Contexto

Gramáticas libres de contexto (GLC)

- Las gramáticas libres de contexto se caracterizan por presentar un solo símbolo no terminal del lado izquierdo de una regla de producción y del lado derecho una combinación de dos o más símbolos terminales y/o no terminales.

  A → ab <A>::=ab

  A → CbD <A>::=<C>b<D> en BNF

- Para aplicar una derivación en una gramática libre de contexto un símbolo no terminal V se reemplaza por un símbolo W, sin tener en cuenta el contexto de uso.

- Uso: lenguaje de programación - HTML - XML

EJEMPLO:

<dig> ::= 0 | 1 | 2.... | 9

<nro>::= <dig> | <nro><dig>

::= → se lee 'se define como'

| → se lee 'or'

< > → representa un símbolo no terminal

< A > < B > → se lee 'A seguido de B'

Gramática para un identificador

Pensemos en el formato del nombre de una variable:

x, suma, valor1, porc2A

Puede tener una sola letra, varias letras seguidas, o letras con números; pero no puede comenzar con un número. Entonces definimos su gramática de la siguiente forma:

<id> ::= <letra>

<id> ::= <id><letra> /*regla recursiva para repetir letras

<id> ::= <id><dig> /*regla recursiva a izq, para evitar que comience con un número

<dig> ::= 0 | 1 | 2.... | 9

<letra> ::= a | b | ....z | A | .. | Z

Gramática para un identificador

<id> ::= <letra>

<id> ::= <id><letra>

<id> ::= <id><dig>

<dig> ::= 0 | 1 | 2.... | 9

<letra> ::= a | b | ....z | A | .. | Z

Esta misma gramática se puede expresar en forma abreviada:

<id> ::= <letra> |<id><letra> |<id><dig>

<dig> ::= 0 | 1 | 2.... | 9

<letra> ::= a | b | ....z | A | .. | Z

Gramática para un identificador

RP= { <dig> ::= 0 | 1 | 2.... | 9

<letra> ::= a | b | ....z | A | .. | Z

<id> ::= <letra> | <id><letra> | <id><dig> }

T= { 0,1,2,3,4,5,6,7,8,9,a,b,c,.....}

NT= { <dig>, <letra>, <id> }

S: <id>

G={ RP, T, NT, S}

Verifiquemos si **suma** es un identificador válido para la gramática propuesta:

<dig> ::= 0 | 1 | 2.... | 9
<letra> ::= a | b | ....z | A | .. | Z
<id> ::= <letra> | <id><letra> | <id><dig>

Partiendo del start symbol y reemplazando a izquierda:

S:<id> → <id> <letra> suma
reemplazando <id> → <id><letra> <letra> suma
→ <id><letra> <letra><letra> suma
→ <letra> <letra><letra><letra> suma
→ s <letra> <letra> <letra>
→ s u <letra> <letra>
→ s u m <letra>
→ s u m a

podemos derivar suma.

¿Si el reemplazo se hace a derecha, qué ocurre?

<dig> ::= 0 | 1 | 2.... | 9
<letra> ::= a | b | ....z | A | .. | Z
<id> ::= <letra> | <id><letra> | <id><dig>

S: <id> → <id> <letra> sum a
→ <id> a

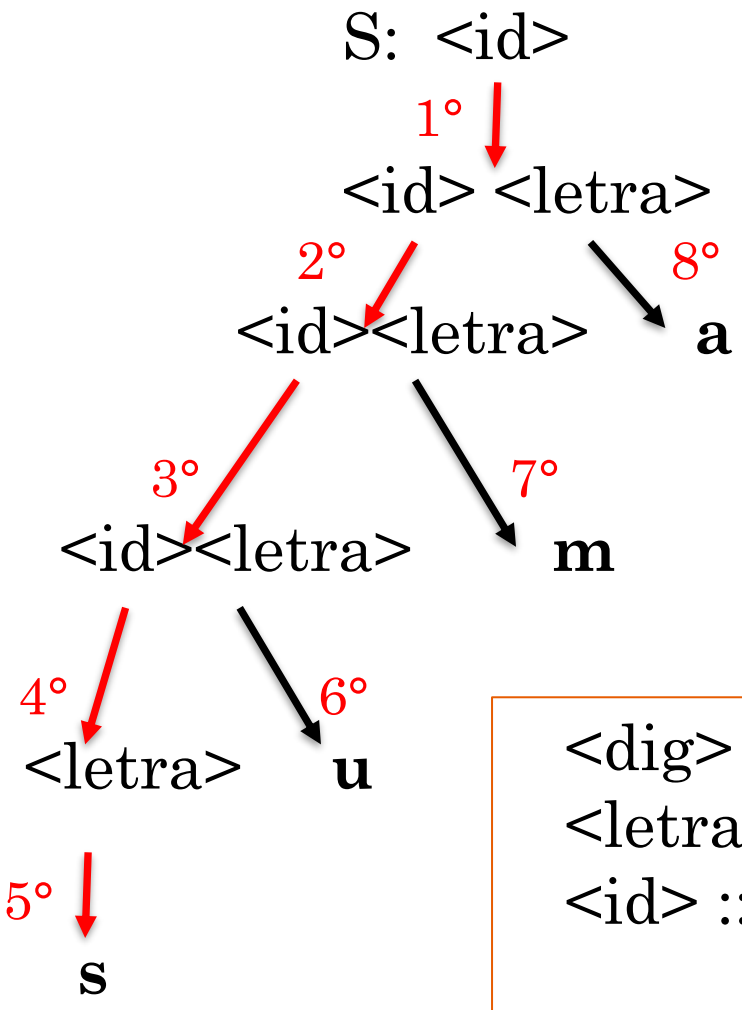
→ <id> <letra> a su m a
→ <id> m a

→ <id> <letra> m a s u ma
→ <id> u m a

→ <letra> u m a s uma
→ **s u m a**

Árbol de derivación (izquierda)

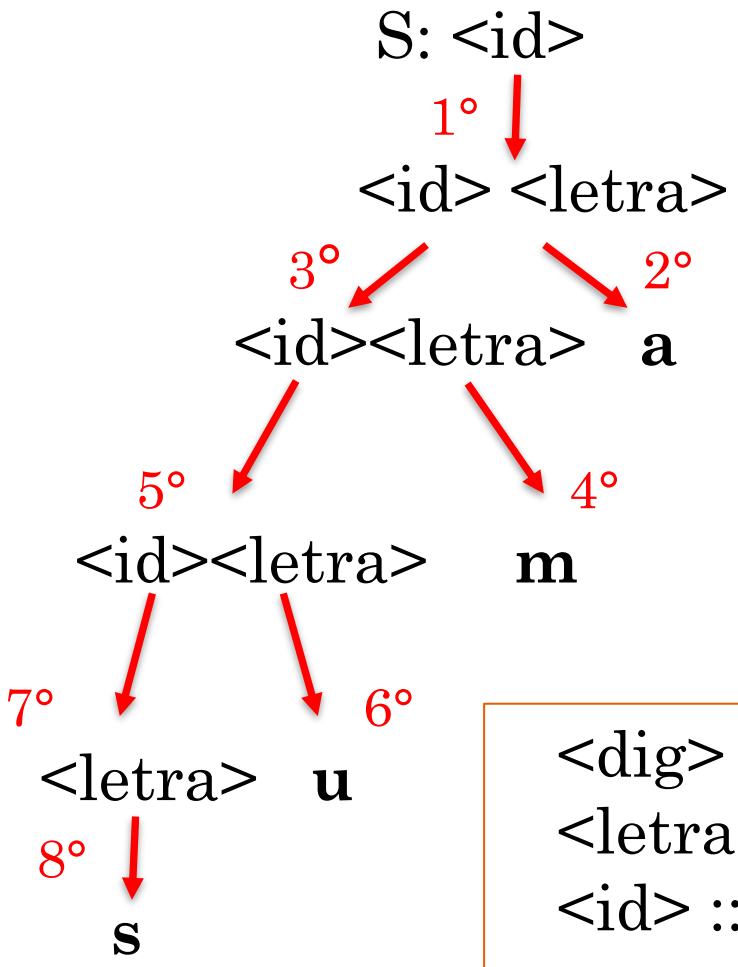
Sentencia: **suma**



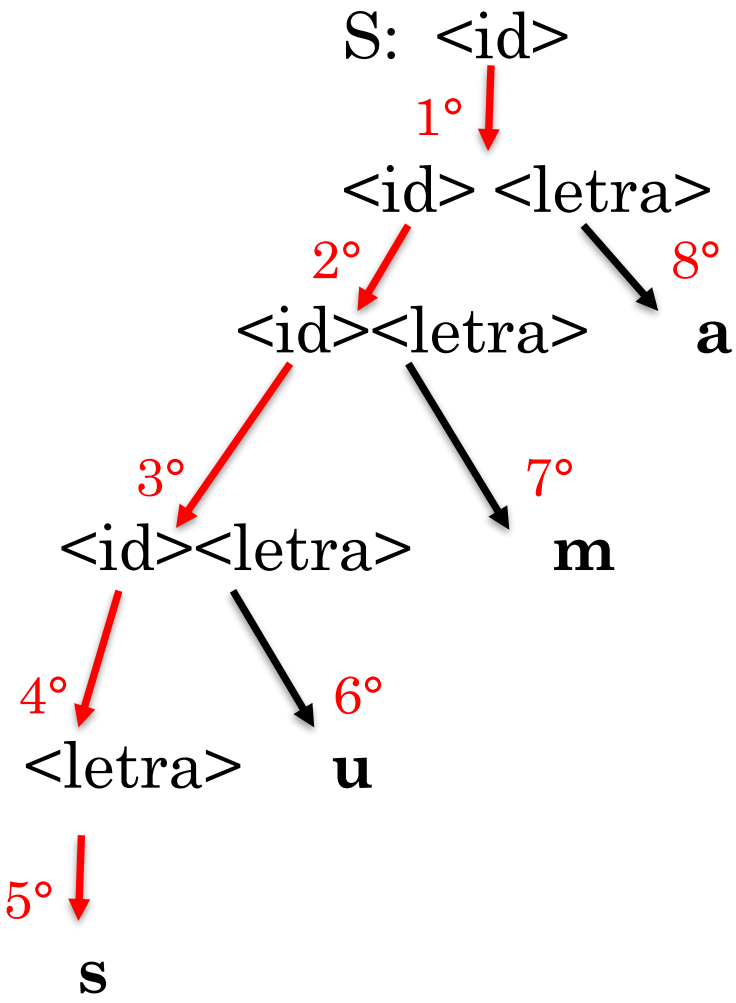
<dig> ::= 0 | 1 | 2.... | 9
<letra> ::= a | b | ....z
<id> ::= <letra> | <id><letra> |
<id><dig>

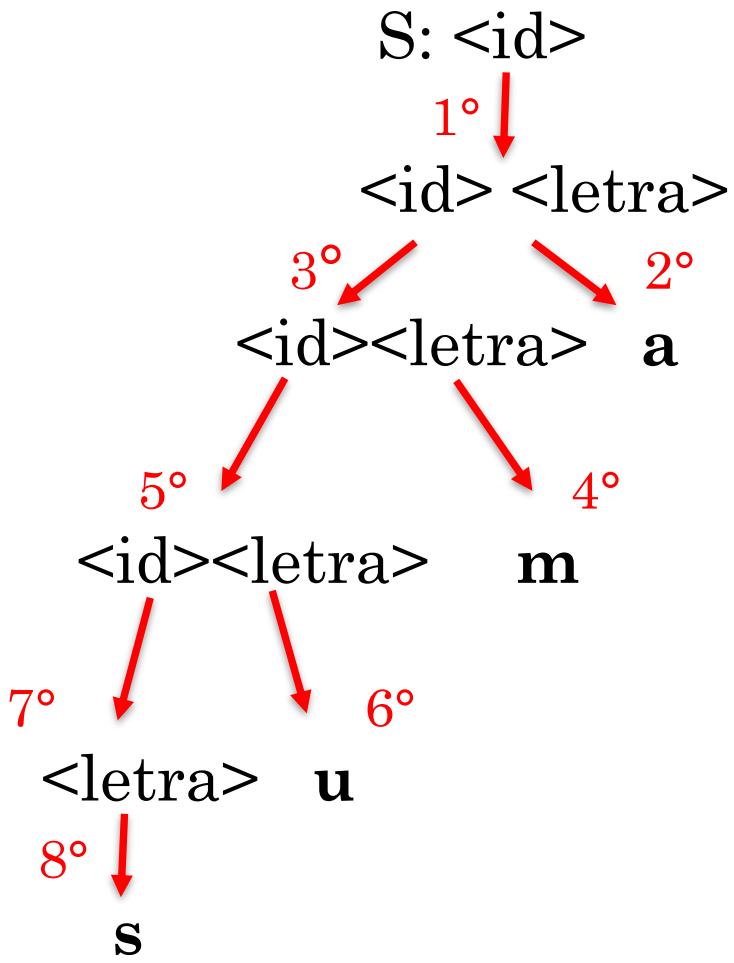
Árbol de derivación (derecha)

Sentencia: **suma**



<dig> ::= 0 | 1 | 2.... | 9
<letra> ::= a | b | ....z
<id> ::= <letra> | <id><letra> |
<id><dig>

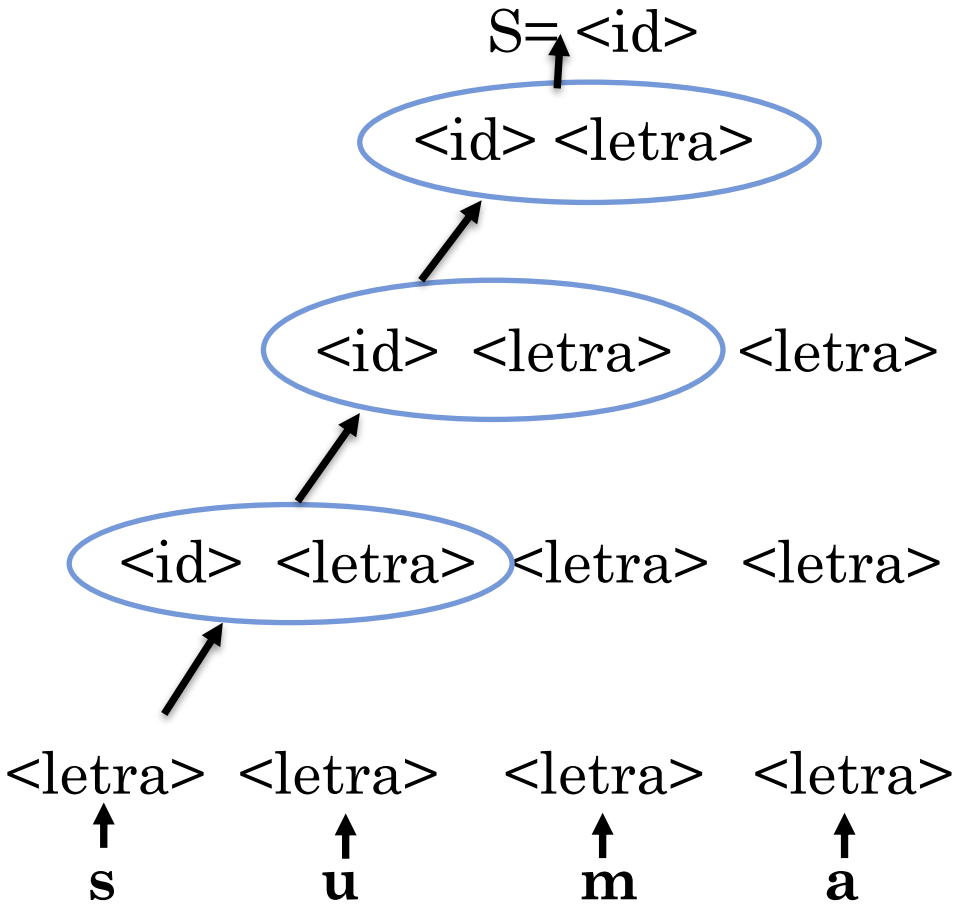




El árbol de derivación a derecha debe coincidir con el de derivación a izquierda

Árbol de derivación: diseño bottom up

<dig> ::= 0 | 1 | 2.... | 9
<letra> ::= a | b | ....z
<id> ::= <letra> | <id><letra> | <id><dig>



Ambigüedad

Cuando hay más de un árbol de derivación para una misma gramática, la gramática se dice que es ambigua. Para evitarlo debemos agregar nuevos no terminales que eliminen la ambigüedad.

Analicemos la siguiente gramática correspondiente a una expresión aritmética.

<asig>::= <id>:=<exp>

<id>::= a| b| …|z

<exp>::= <exp> + <exp> |<exp> * <exp>|<id>

Vamos a probar esta gramática con la siguiente expresión: a:= b + c * d

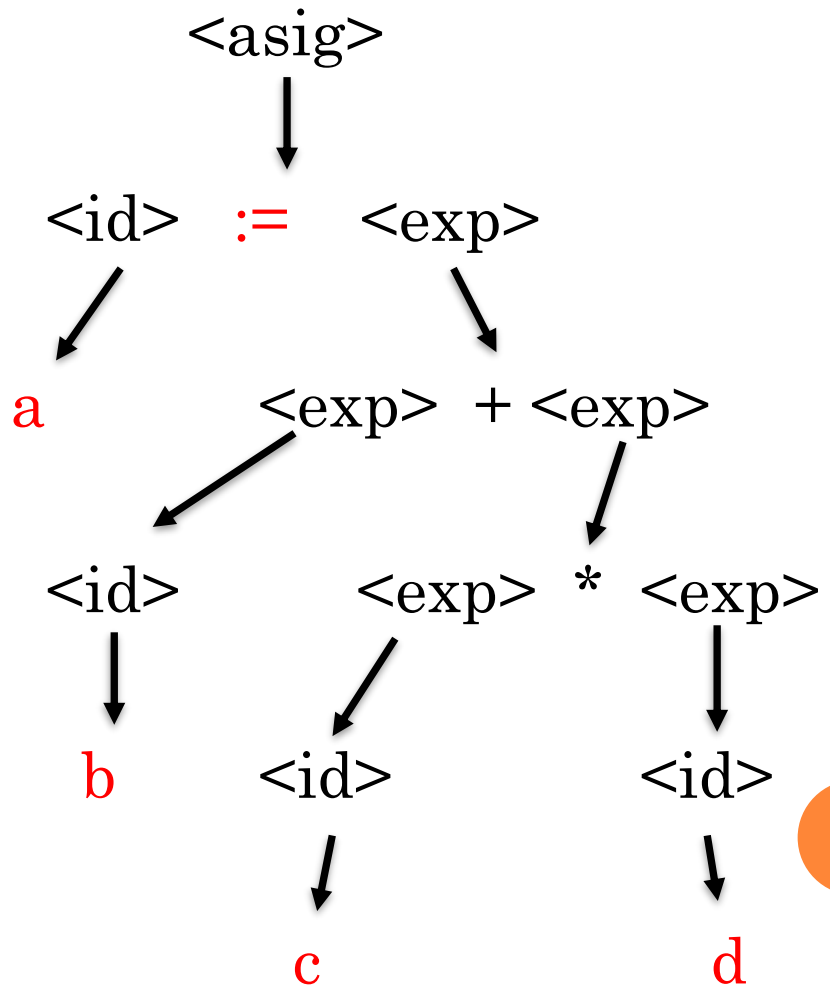
Lo primero que vamos a hacer es el árbol de derivación.

<asig>::= <id>:=<exp>

<id>::= a | b | … | z

<exp>::= <exp> + <exp> |<exp> * <exp> |<id>

A partir de la gramática podemos construir el siguiente árbol de derivación:



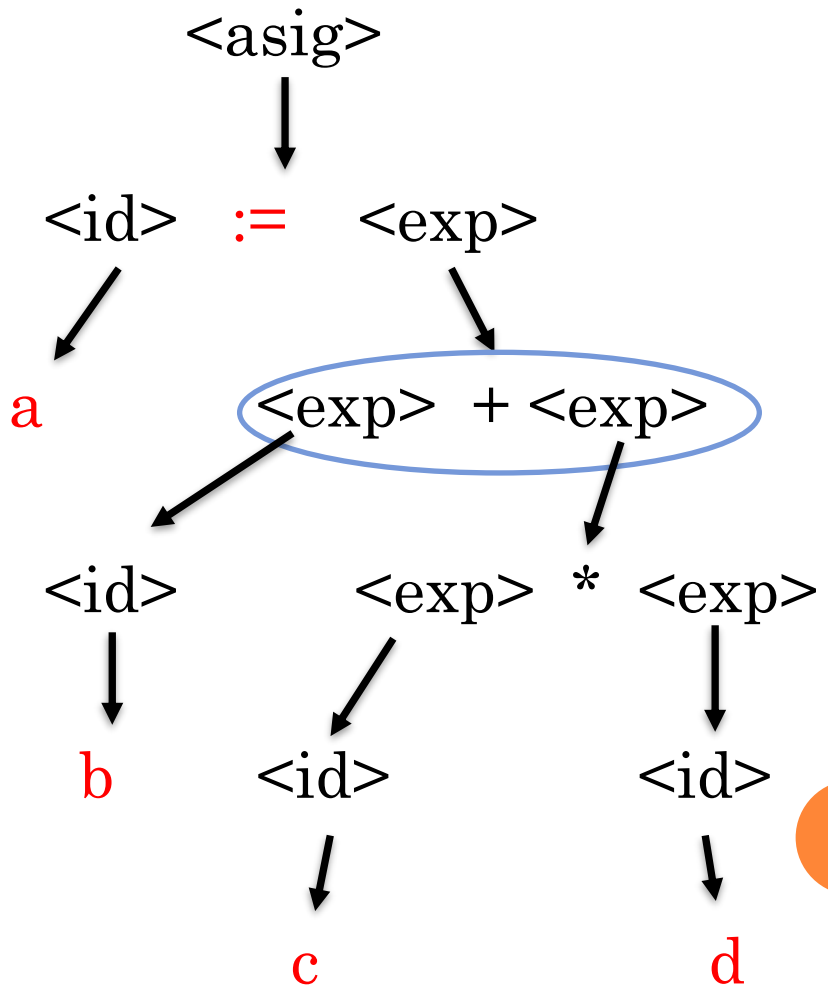
Probamos la gramática con la siguiente expresión: a:= b + c * d

<asig>::= <id>:=<exp>

<id>::= a | b | … | z

<exp>::= <exp> + <exp> | <exp> * <exp> | <id>

Obtuvimos este árbol de derivación porque en la definición de expresión consideramos la primer opción.



<asig>::= <id>:=<exp>

<id>::= a | b | … | z

<exp>::= <exp> + <exp> | <exp> * <exp> | <id>

Pero nada nos impide entrar en la segunda opción.

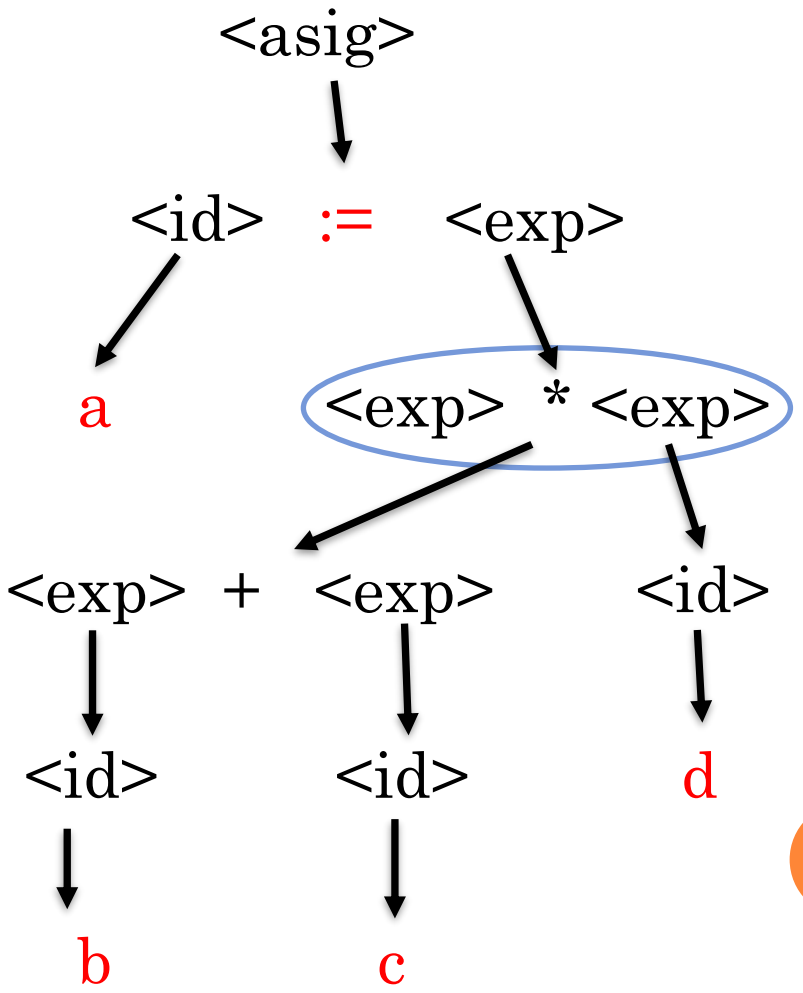
Vamos a construir el árbol de derivación ingresando en la segunda definición de expresión..

<asig>::= <id>:=<exp>

<id>::= a | b | … | z

<exp>::= <exp> + <exp> | <exp> * <exp> | <id>

Podemos observar que se obtiene otro árbol de derivación, diferente



Esto no debe pasar, por eso podemos concluir que esta gramática es ambigua.

Para sacar la ambigüedad se agregan más símbolos no terminales y se aumenta el nivel de abstracción, tratando de ser lo más general posible en cada regla de producción

<asig>::= <id>:=<exp>

<id>::= <letra> | <id> <letra> | <id> <dig>

<dig>::= 0|…|9

<nro>::= <dig> | <nro> <dig>

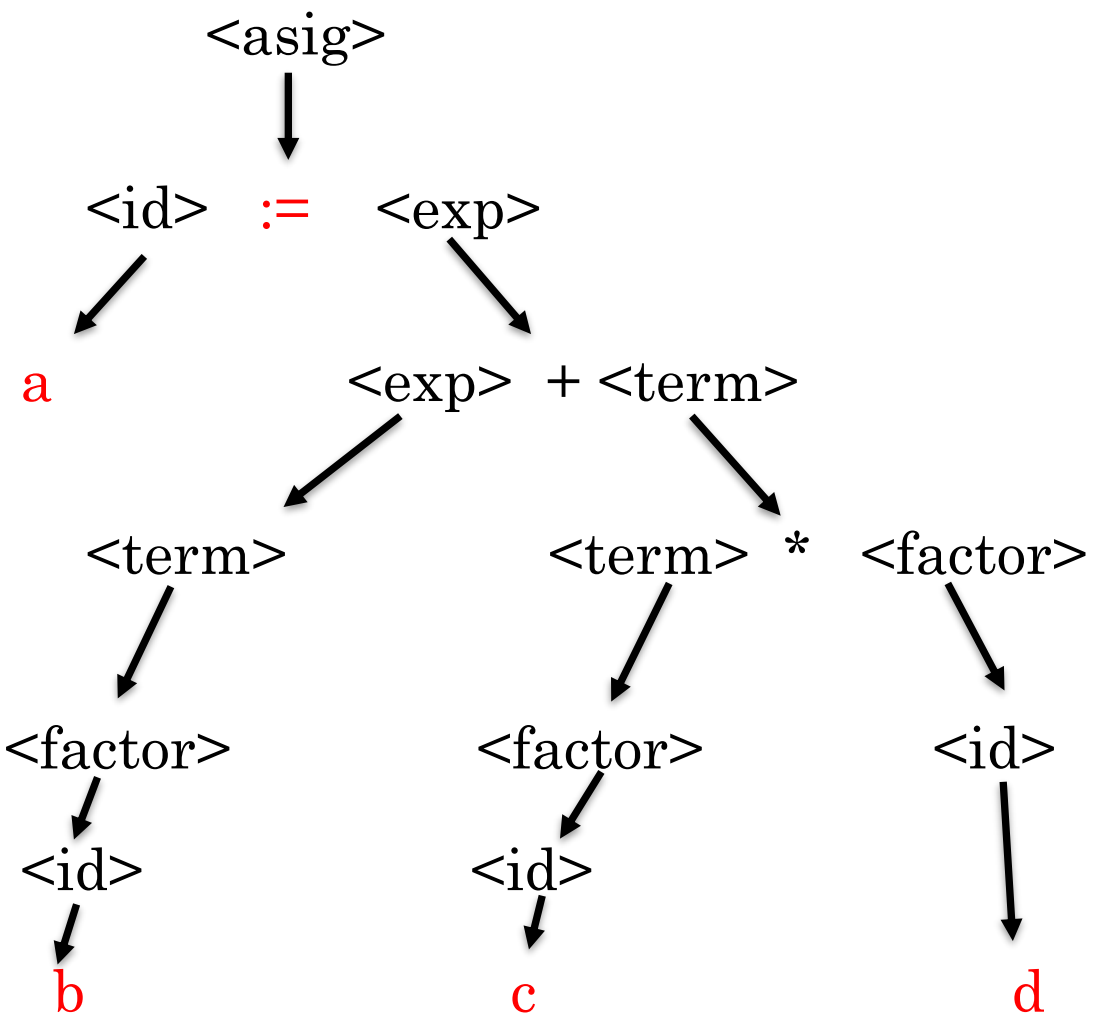
<letra>::= a| b| …|z

<exp>::= <exp> + <term> | <exp> - <term> | <term>

<term>::= <term> * <factor>|<term> / <factor>| <factor>

<factor>::= <id> | <exp> | <nro>

Volvamos a verificar la expresión: a:= b + c * d



Esta es la gramática correcta de expresión.
No tiene dos árboles de derivación.

Resumiendo:

- Dado una alfabeto $\sum$, el lenguaje L generado por la gramática G es el conjunto de todas las palabras derivables a partir de G.
- Si la gramática genera más palabras que las que pertenecen a L, entonces G es incorrecta.
- Si la gramática genera menos palabras, entonces G es incompleta.
- Si hay más de un árbol de derivación para una palabra, entonces G es ambigua.

EJEMPLO 1:

Definir la gramática para validar la declaración de un registro en Pascal.

Recordemos el formato:

```
reg=record
      campo1:tipo;
      campo2,campo3: tipo;
end;
```

<registro>::= <id>=record<listacampos> end;
<listacampos>::=<uncampo>| <listacampos><uncampo>
<uncampo>::= <listaiden>:<tipo>;
<listaiden>::=<id> | <listaiden>,<id>
<id>::= <letra> |<id><letra>| <id><dig>
<letra> ::= a | b | ….z | A | .. | Z
<dig>::=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

EJEMPLO 2:

Definir la gramática para validar la declaración de variables en C.

Recordemos el formato:

```
int v1;
int v1,v2;
float  d1,d2=4,5;
string s1="sol",s2="hola";
```

<decl_var>::= <unadecl>| <decl_var><unadecl>
<unadecl>::= <tipo> <listaDec>;
<listaDec>::=<listasimple>|<listaCombinada>
<listasimple>::=<listaiden>|<listaAsig>
<listaiden>::=<id> | <listaiden>,<id>
<listaAsig>::=<unaAsig>|<listaAsig>,<unaAsig>
<unaAsig>::=<id>=<nro>|<id>=<id>
<listaCombinada>::= <listasimple>|<listacombinada>,<listasimple>
<nro>::=<dig>| <nro><dig>
<id>::= <letra> | <id><letra>| <id><dig>
<dig>::=0|1|2|3|4|5|6|7|8|9
<letra> ::= a|b|….z|A|..|Z

EJEMPLO 3:

Definir la gramática para validar el encabezamiento de una función en Python.

Recordemos el formato:

def nombre (argumentos): o def nombre():

<función>::= def <id> (<listaParam>): | def <id>( ):
<listaParam>::= self | self, <listaArg> | <listaArg>
<listaArg>::= <id>| <listaArg>,<id>
<id>::= <letra> | <id>_<letra>| <id>_<dig>|<id><letra>|
<id><dig>
<dig>::=0|1|2|3|4|5|6|7|8|9
<letra> ::= a|b|….z|A|..|Z

Observación: Tener en cuenta que en Python los identificadores pueden llevar un guión bajo. En los métodos de instancia o de clase se usa la pseudovariable self como primer argumento del mismo.

EJEMPLO 4:

Definir la gramática para validar la sentencia import en Python.

Recordemos el formato:
import nombre o
import nombre1,nombre2

<import>::= import <listamodulos>
<listamodulos>::=<unmodulo> | <unmodulo>,<listamodulos>
<unmodulo>::= os | sys | math | <id> | ……
<id>::= <letra> | <id>_<letra> | <id>_<dig> | <id><letra> |
<id><dig>
<dig>::=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letra> ::= a | b | ….z | A | .. | Z

EJERCICIO:

Escribir la gramática correspondiente a un programa en Pascal como el del ejemplo.

```
Ej: program ejemplo;
    var
    a,b;integer
    begin
    a:=5;
    b:=10;
    ....
    ....
    if a>b then
            a:=a+1
            else
            a:=a+ b;
    end.
```

EJERCICIO:

A partir de la descripción BFN el compilador valida cada una de las sentencias del programa, realizando el análisis léxico y sintáctico correspondiente a L.

Ej: program ejemplo;
var a,b;integer;
nom:string;
begin
....
....
end.

Tener en cuenta que tiene el siguiente formato:

- S:<programa>::= program <id>; <cuerpo>
  <cuerpo>::= <declaraciones> begin <listasent> end.
  <declaraciones>::= var <listadecl>
  <listadecl>::=<unadecl>; |<listadecl><unadecl>
  <unadecl>::= <listaid> : <tipo>
  <listaid>::=<id> |<listaid>,<id>
  <listasent>::= <sent> | <listasent>; <sent>
  <sent>::= <if> | <while>| ……
  …..