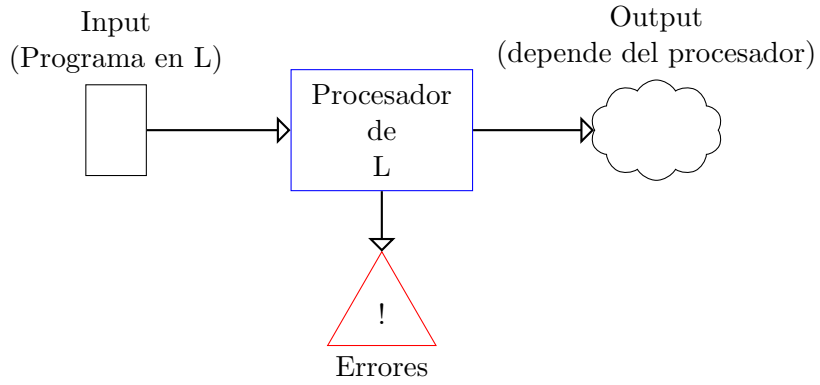
Procesadores de Lenguajes

David Antuña Rodríguez

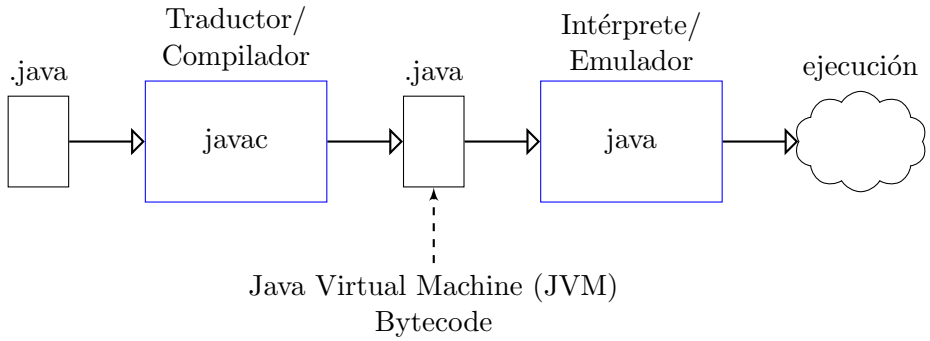
Contenidos

1 Introducción

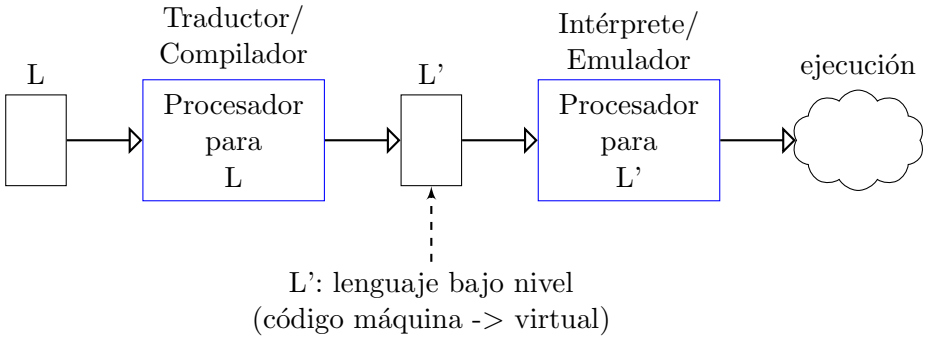


Compilador: Genera una representación en otro L (Alto nivel -> Bajo nivel).
Intérprete: Ejecuta una representación.

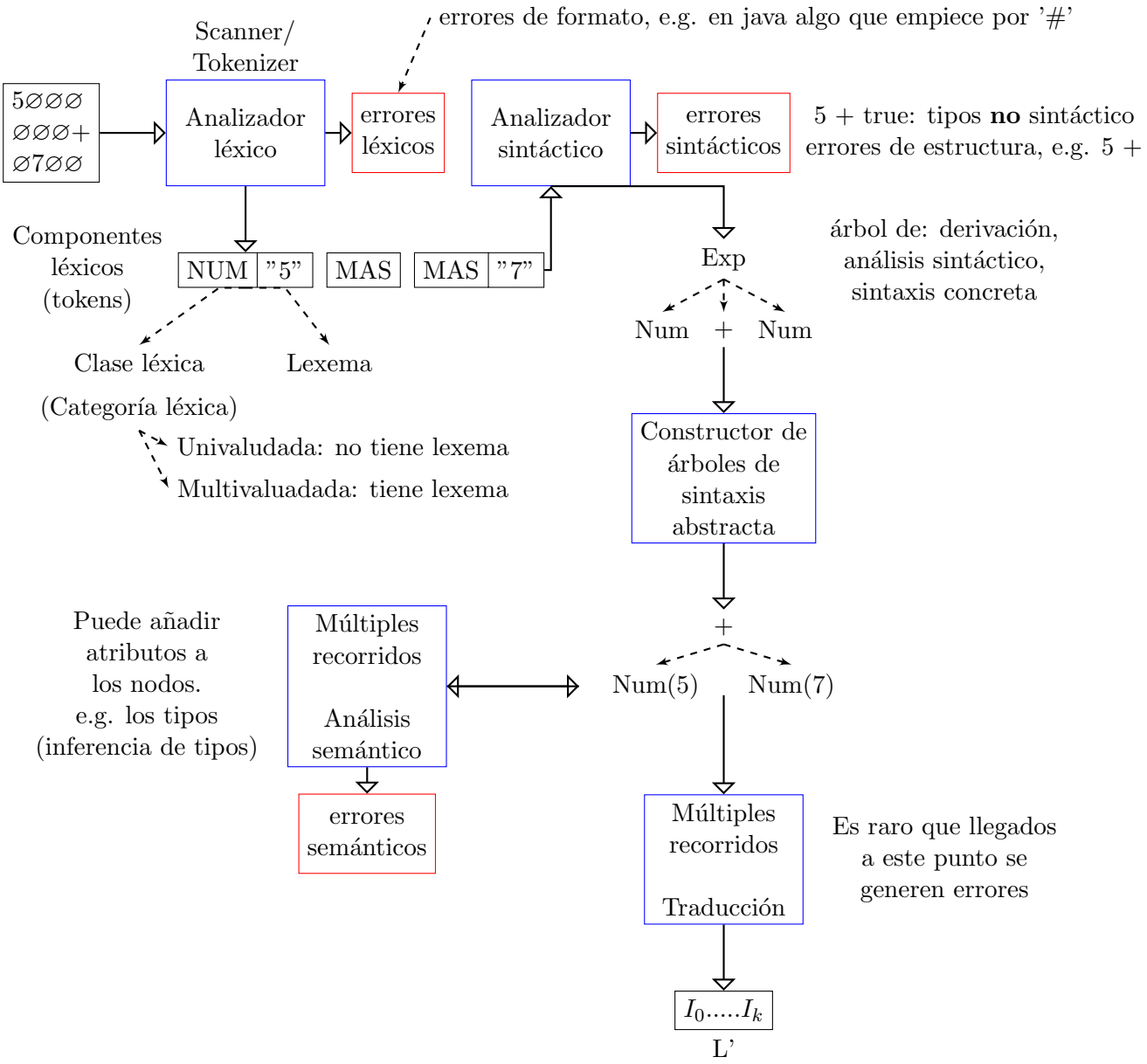
Java



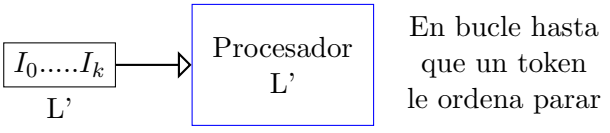
Abstracción



1.1 Traductor

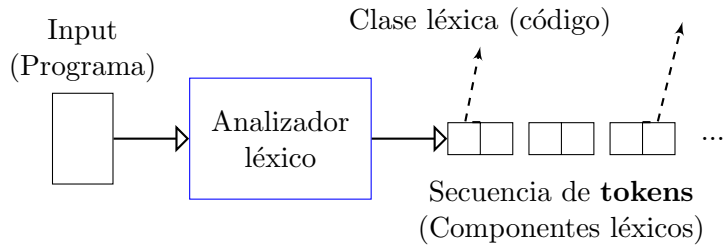


1.2 Intérprete



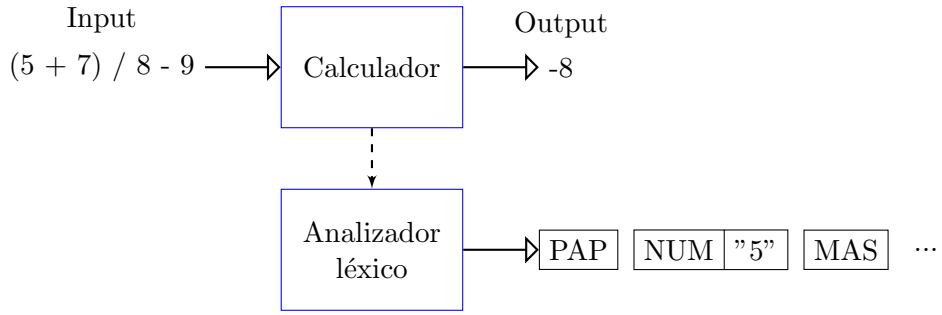
2 Análisis léxico

Lexema (opcional). Tiene sentido si la clase léxica no determina **unívocamente** la cadena.



Clase
multivaluada ≡ **si** lexema
univaluada ≡ **no** lexema

2.1 e.g. prototípico



Una expresión aritmética es...

- ...un **número entero**.
- ...una **expresión** seguida de un **operador** seguido de otra **expresión**.
- ...**(** seguido de una **expresión** seguido de **)**.

Los operadores son: +, -, * y /.

2.2 Método de desarrollo

2.2.1 Identificación de las clases léxicas

Normalmente:

- Los **signos** (puntos, operadores...) conviene que sean clases **univaluadas**.
- **Variables** (NUM,LETRA...) son **multivaluadas**.
- **Palabras reservadas** (bool, int...) son **univaluadas**.

Clases léxicas de apartado 2.1.

- PAP (paréntesis apertura)
- PCIERRE (paréntesis cierre)
- NUM
- MAS
- MENOS
- POR
- DIV

e.g.

```
class C {
    int x;
    ....
```

- class ≡ univaluada
- C ≡ id
- { ≡ univaluada
- int ≡ univaluada
- x ≡ id

2.2.2 Describir las clases léxicas

Cada clase léxica es un lenguaje formal de los posibles lexemas.
Suposición: son lenguajes regulares, por tanto:

- Se pueden describir mediante ER_s (expresiones regulares).
- Se pueden reconocer utilizando AFD_s (autómatas finitos deterministas).

e.g. NENT

- Descripción informal.

  Empieza con un signo (+ o -) opcional.
  A continuación aparecen uno o más dígitos.
  e.g. +5, +007, 000, -0, -08, -280.

- Descripción formal.

  (\+ | \-)?[0-9]+

Notación

- ER sobre el alfabeto Σ (e.g. UNICODE, ASCII...).
  - Una "letra" de Σ, e.g. a. Denota {a}
  - Si E_0 y E_1 son ER_s, también lo son:
    - $(E_0 \mid E_1)$. Denota la unión, $L(E_0) \cup L(E_1)$.
    - $(E_0 \bullet E_1)$. Denota la concatenación, $\{W_0W_1 \mid W_0 \in L(E_0), W_1 \in L(E_1)\}$.
    - (E_0*). Denota $\{\epsilon, W_0, W_0W_1, W_0W_1W_2... \mid W_0, W_1, W_2, ... \in L(E_0)\}$.
  - ϵ denota el lenguaje formado por la cadena vacía.
- Convenios.
  - $*$ tiene mayor prioridad que $\mid$ y $\bullet$.
  - $\bullet$ tiene mayor prioridad que $\mid$.
  - $\bullet$ se puede omitir.
  - Los ( ) se usan para cambiar prioridades.
  - Conjuntos de caracteres, e.g. [0-9,a] es el conjunto formado por los dígitos y la a.
    - $[e_0, ..., e_n]$ donde e_i puede ser...
      - ...una letra.
      - ...a-b, conjunto de caracteres comprendidos entre a y b.
    - Conjuntos complementados $[\hat{}e_0, ..., e_n]$.
      e.g. [^0-9, a-z] es el conjunto de caracteres que no son dígitos ni letras minúsculas.
    - E+ $\equiv$ EE$*$. Aparece una o más veces pero mínimo una.
    - E? $\equiv$ (E$\mid \epsilon$). Aparece o no, es opcional.
    - \. La forma de escape.

e.g. Identificadores

- Pueden contener letras, dígitos y _.
- Empiezan por letra o _.
- Van seguidos de una secuencia de 0 o más caracteres válidos.

[a-z, A-Z, _] [a-z, A-Z, _, 0-9]$*$

e.g. NENT, como en el ejemplo anterior pero sin 0_s a la izquierda.
[\+, \-]?([1-9] [0-9]$*$ | 0)

Para mayor claridad vamos a utilizar DR_s (definiciones regulares) en lugar de ER_s.

- $(*) \equiv$ clase léxica, marca cual es la definición principal.
- $[I] \equiv$ ignorables, se utiliza para definir que caracteres no se han de tener en cuenta.
- Las palabras subrayadas corresponden a definiciones auxiliares.

e.g. Identificadores

(∗) IDEN ≡ Letra (Letra | Dig)∗
Letra ≡ [a-z, A-Z, _]
Dig ≡ [0-9]

e.g. Números enteros

(∗) LENT ≡ Signo? (0 | DPos Dig∗)
Signo ≡ [\+, \-]
DPos ≡ [1-9]
Dig ≡ [0-9]

e.g. Literales reales

- Empiezan por un entero.
- Continúan por...
  - ...una parte decimal.
  - ...una parte exponencial.
  - ...una parte decimal seguida de una exponencial.
- Parte decimal. '.' seguido de una secuencia de uno o más dígitos, sin ceros superfluos a la derecha.
- Parte exponencial. E o e seguida de un entero.

  e.g. +5.7E-28

(∗) LREAL ≡ LENT (PDEC | PEXP | PDEC PEXP)
PDEC ≡ \. (Dig∗ DPos | 0)
PEXP ≡ (E | e) LENT

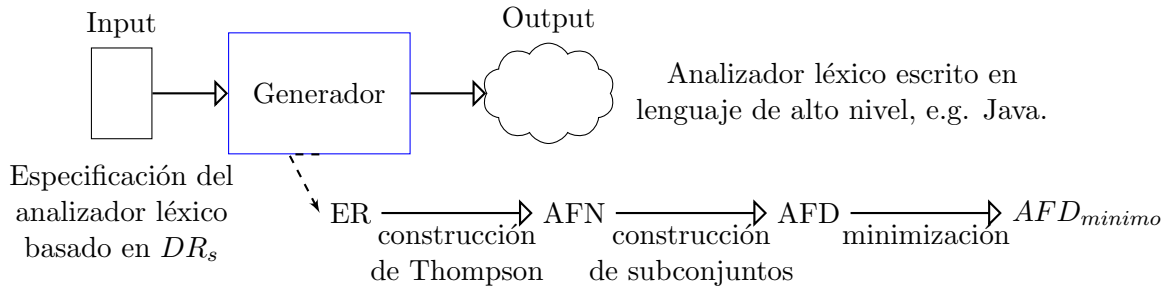
LENT ≡ Signo? (0 | DPos Dig∗)
Signo ≡ [\+, \-]
DPos ≡ [1-9]
Dig ≡ [0-9]

Ignorables
[*I*] SEP ≡ [' ', \t, \n, \r, \b], posibles separadores.
[*I*] COM ≡ # [^ \n]∗ \n, e.g. # Esto es un comentario.\n

e.g DR de apartado 2.1
(∗) PAP ≡ \(
(∗) PCIERRE ≡ \)
(∗) NUM ≡ Signo? (0 | DPos Dig∗)
Signo ≡ [\+, \-]
DPos ≡ [1-9]
Dig ≡ [0-9]
(∗) MAS ≡ \+
(∗) MENOS ≡ \-
(∗) POR ≡ \∗
(∗) DIV ≡ \/
[*I*] SEP ≡ [' ', \t, \n, \r, \b]
[*I*] COM ≡ # [^ \n]∗ \n

2.3 Implementación

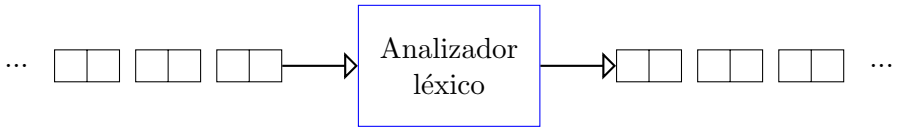
2.3.1 Generadores de analizadores léxicos



2.3.2 Implementación manual

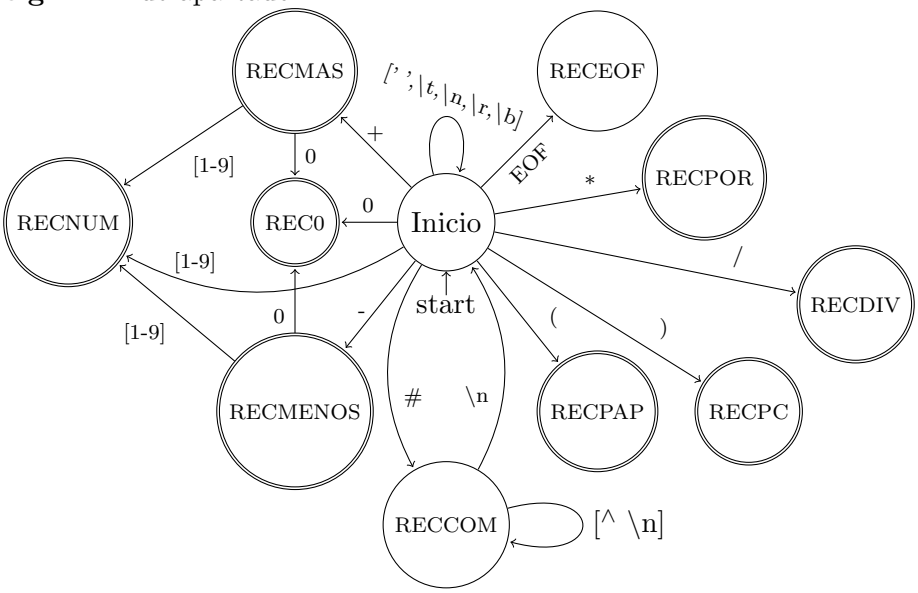
Se implementa utilizando AFD_s donde las transiciones a estados de error no se dibujan, son implicitas.

El AFD no reconoce la entrada completamente sino que utiliza una **arquitectura pull**, solicita los token uno a uno y los va procesando.



Siempre hay que tener una clase léxica que represente el EOF.

e.g. AFD de apartado 2.1



Inicialización: Conseguir el primer caracter en sigCar.

SigToken
Estado ← EstadoInicial
lexema ← ""
loop {
switch(Estado) {
case S_0
...
case S_n
}
}

case S_i
si sigCar ∈ T_0
Estado ← SigEstado
lexema ← lexema + sigCar
(lexema solo cambia en estados no ignorables)
Actualizar sigCar
si no si sigCar ∈ T_1
Estado ← SigEstado
lexema ← lexema + sigCar
Actualizar sigCar
...
si no
//Si es un estado final
return token
//Si no es un estado final
error

Dos variables extra, fila y columna, para poder dar información extra en el mensaje de error.

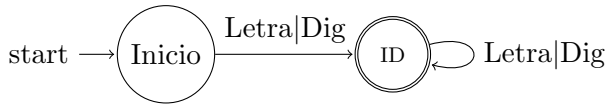
Codificación parcial (pseudocódigo)

```
Estado ← Inicio
lexema ← ""
loop {
   switch (Estado) {
   Inicio: if sigCar = ( then transita(RECPAP)
           else if ...
           ...
           else if sigCar = # textbfthen transitaIgnorando(RECCOM)
           else if sigCar ∈ [1-9] then transita(RECNUM)
           else error()

   RECNUM: if sigCar ∈ [1-9] then transita(RECNUM)
                else return token(NUM, lexema)

   ...

   RECCOM: if sigCar ≠ \n then transitaIgnorando(RECCOM)
                else transitaIgnorando(Inicio)
   }
}


transita(S) {
   Estado ← S
   lexema ← lexema + sigCar
   Actualiza sigCar
}


transitaIgnorando(S) {
   Estado ← S
   Actualiza sigCar
}
```

e.g.

	Especificación léxica
	...
EVALUA 5 + x DONDE x = 27	(∗) ID ≡ Letra (Letra | Dig)∗ Letra ≡ [a-z,A-Z,_] Dig ≡ [0-9] (∗) EVALUA ≡ [E,e][V,v][A,a][L,l][U,u][A,a] (∗) DONDE ≡ [D,d][O,o][N,n][D,d][E,e]

En el return que reconoce los id se comprueba si el lexema es una palabra reservada, de serlo se devuelve la palabra reservada en lugar del id.

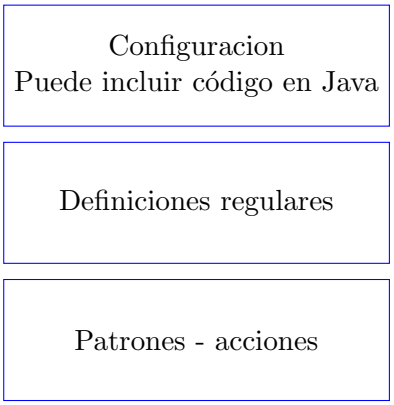


Hay un ejemplo de codificación en el campus, tiene una errata en el diagrama de transiciones.

2.3.3 Implementación mediante herramientas

La herramienta que vamos a utilizar es **JLex**, hay un ejemplo en el CV.

Formato de JLex:

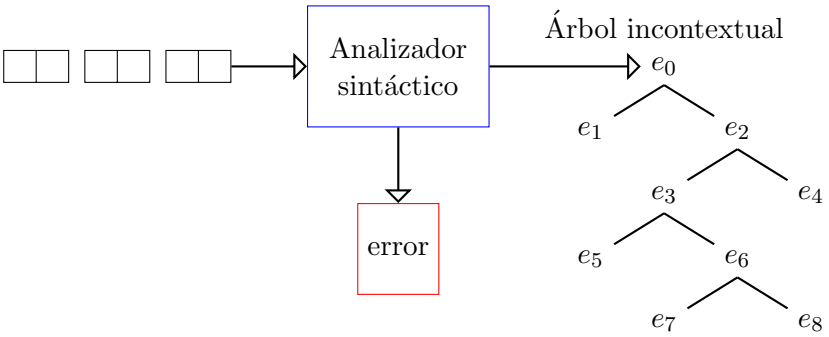


Para que una DR pueda referirse a otras tienen que haberse definido antes, lo que en las DR_s subrayamos tiene que ir entre {}, hacen sustituciones literales (no ponen paréntesis).

Las palabras reservadas se ponen sin mas, el orden de reconocimiento equivale al de aparición en el fichero, en la parte de los return.

Compilar: java -cp jlex.jar JLex.Main input(DR_s)

3 Análisis sintáctico



Hipótesis: Las cadenas de clases léxicas generadas por el analizador léxico forman un lenguaje incontextual.

3.1 Recordatorio

3.1.1 Gramática

Sea una GI (Gramática Incontextual), GI(N,T,P,S):

- N $\equiv$ alfabeto de no terminales $\Rightarrow$ clases sintácticas.
- T $\equiv$ alfabeto de terminales $\Rightarrow$ clases léxicas.
- P $\equiv$ conjunto de reglas de la forma A $\rightarrow \alpha$, donde A $\in$ N y $\alpha \in$ (NUT)$*$.
- S $\in$ N y es el símbolo inicial.

Sea G $\equiv$ (N,T,P,S) $\rightarrow$ G denota un lenguaje L(G)

- Relación de derivación $\Rightarrow_G$ ($0 \Rightarrow$ si G se sobreentiende) $\Rightarrow \subseteq$ (NUT)$*$ x (NUT)$*$
  αA$\beta \wedge$ A $\rightarrow \gamma \in$ P entonces αA$\beta \Rightarrow \alpha\gamma$B
- Se considera $\Rightarrow_*$ aplicar cero o más veces $\Rightarrow$.

L(G) = {w $\in$ T$*$ | S $\Rightarrow_*$ w}

e.g. Número binario
N = {N,B}
T = {0,1}
P = N $\rightarrow$ B
N $\rightarrow$ NB
B $\rightarrow$ 0
B $\rightarrow$ 1
S = N

Esto sería equivalente a dar tan solo las reglas(P), donde:

- El símbolo azul de la primera regla es el símbolo inicial(S).
- El conjunto de símbolos azules son los no terminales(N).
- El conjunto de símbolos rojos son los terminales(T).

La derivación se denomina *mas a la izquierda* si siempre se reescribe el no terminal que está mas a la izquierda, equivalente para *mas a la derecha*.

Derivación
N ⇒ NB ⇒ NBB ⇒ N1B ⇒ B1B ⇒ B10 ⇒ 010

Derivación mas a la izquierda
N ⇒ NB ⇒ NBB ⇒ BBB ⇒ 0BB ⇒ 01B ⇒ 010

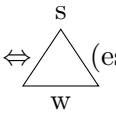
Derivación mas a la derecha
N ⇒ NB ⇒ N0 ⇒ NB0 ⇒ N10 ⇒ B10 ⇒ 010

3.1.2 Árbol de análisis sintáctico

La idea es obtener una representación única para cada sentencia.

Árboles

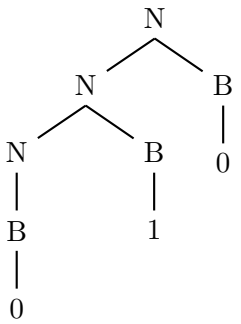
- La raíz está etiquetada con el símbolo inicial.
- Están ordenados, hay un órden en los hijos de los nodos (primer hijo, segundo hijo...).
- Los nodos internos están etiquetados por no terminales.
- Los nodos hoja están etiquetados por terminales o por ϵ.
- Si un nodo está etiquetado por A y sus hijos por α entonces A $\rightarrow \alpha \in$ P.

w $\in L(G) \Leftrightarrow$ 

(es un árbol de análisis sintáctico)

e.g.

N → B
N → NB
B → 0
B → 1



Cada nodo es equivalente a un array en el que se referencian sus hijos en orden.

3.2 Especificación sintáctica

3.2.1 Determinar las clases sintácticas

¿Cómo? A partir de la especificación informal.

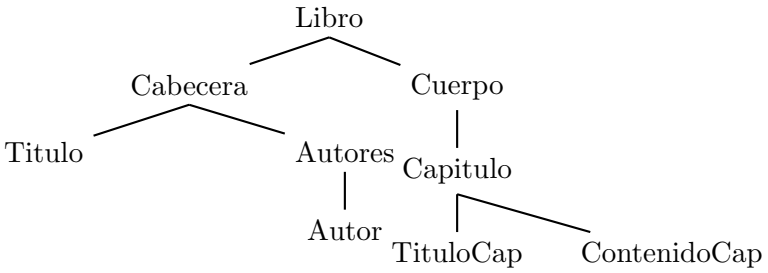
De arriba a abajo:

- Identificar las clases complejas.
- Descomponerlas en clases más simples hasta llegar a las clases léxicas.

Descomposición → Las clases resultantes deben estar al mismo nivel, el más alto posible.

e.g. L que describe libros

Titulo [...]
Autores [Autor]...[Autor]
Capitulo [...][...]
...
Capitulo [...][...]



4 Semántica estática

5 Traducción