

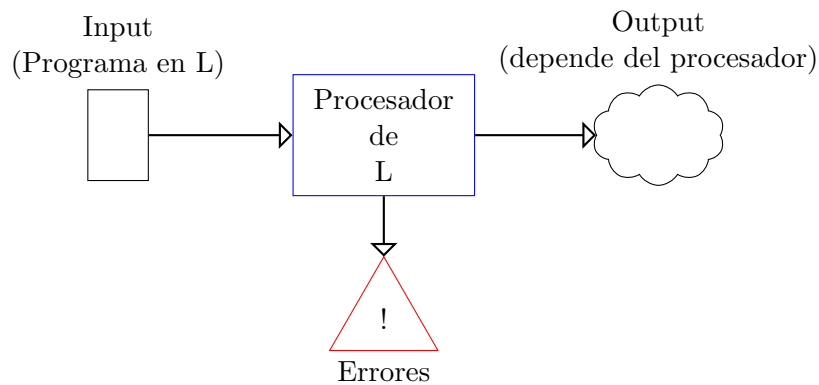
Procesadores de Lenguajes

David Antuña Rodríguez

Contenidos

1	Introducción	1
1.1	Traductor	2
1.2	Intérprete	2
2	Análisis léxico	3
2.1	e.g. prototípico	3
2.2	Método de desarrollo	3
2.2.1	Identificación de las clases léxicas	3
2.2.2	Describir las clases léxicas	4
2.3	Implementación	7
2.3.1	Generadores de analizadores léxicos	7
2.3.2	Implementación manual	7
2.3.3	Implementación mediante herramientas	10
3	Análisis sintáctico	11
3.1	Recordatorio	11
3.1.1	Gramática	11
3.1.2	Árbol de análisis sintáctico	12
3.2	Especificación sintáctica	13
3.2.1	Determinar las clases sintácticas	13
3.2.2	Patrones para secuencias	14
3.2.3	Patrones para operadores	15
3.3	Implementación	23
4	Semántica estática	26
5	Traducción	26

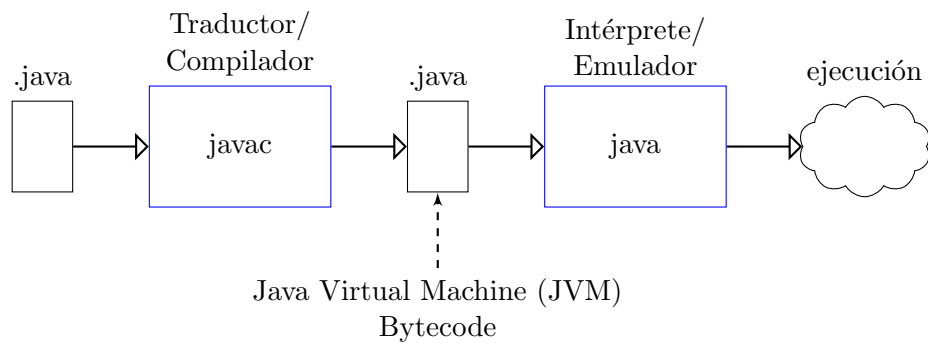
1 Introducción



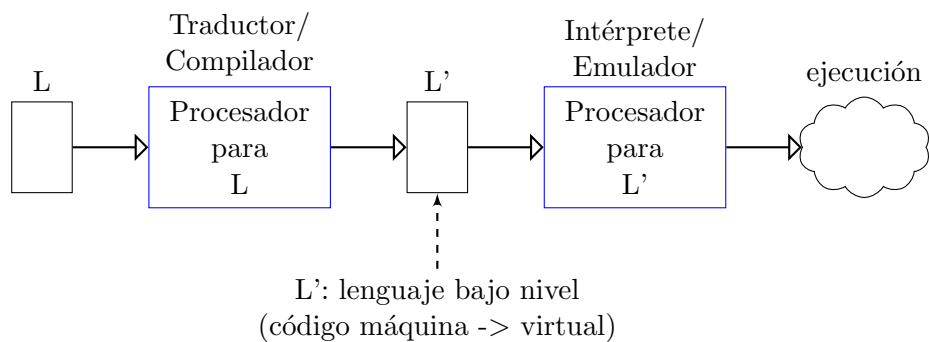
Compilador: Genera una representación en otro L (Alto nivel -> Bajo nivel).

Intérprete: Ejecuta una representación.

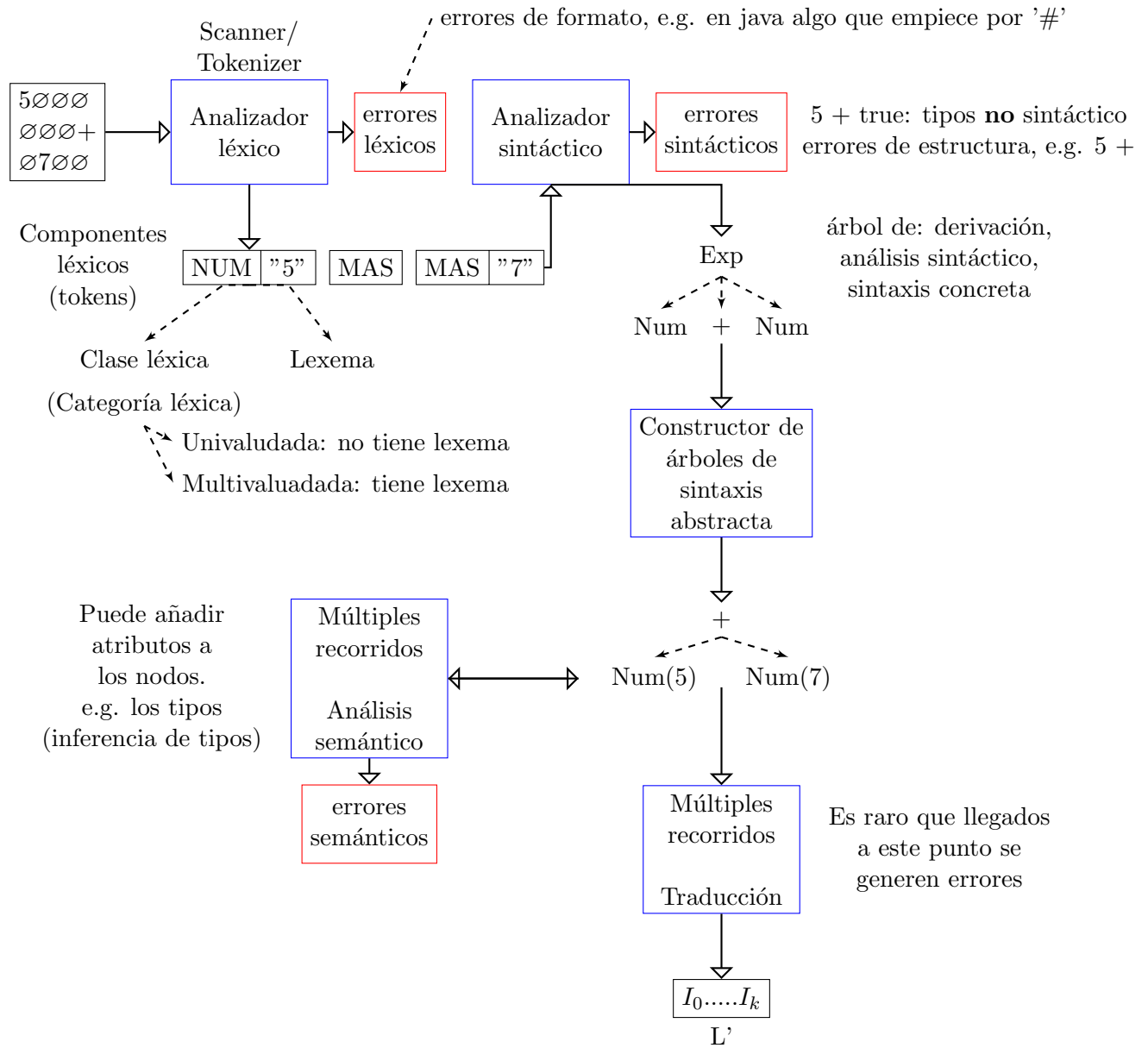
Java



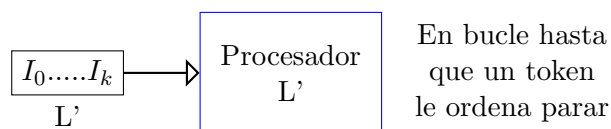
Abstracción



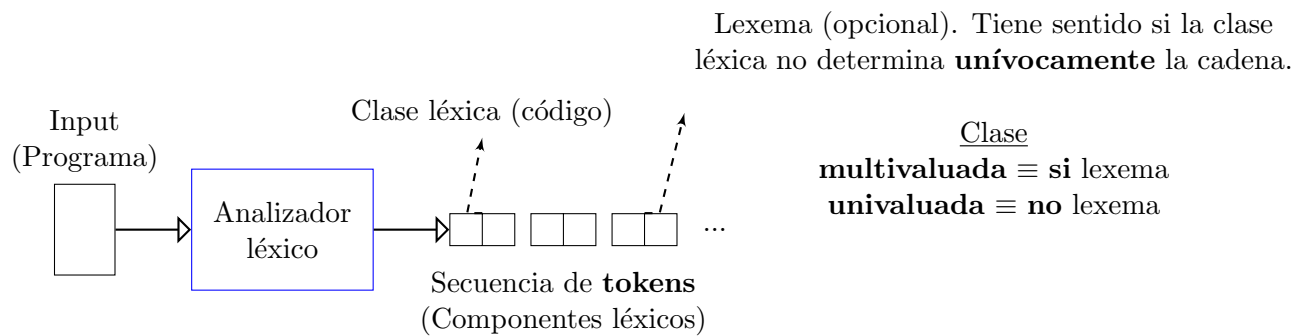
1.1 Traductor



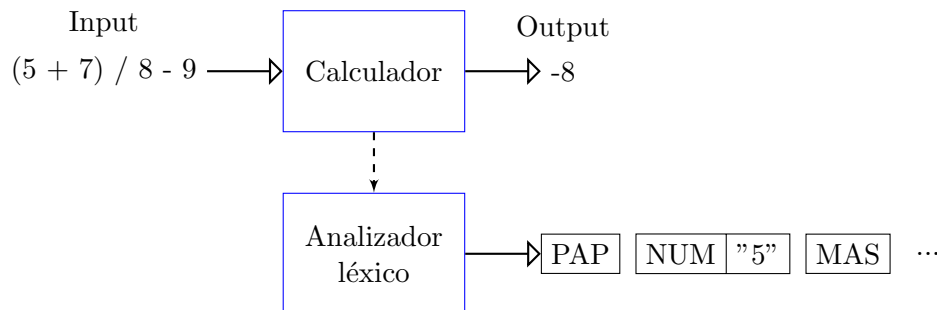
1.2 Intérprete



2 Análisis léxico



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 \equiv univaluada
- C \equiv id
- { \equiv univaluada
- int \equiv univaluada
- x \equiv 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.
 $(\backslash+ | \backslash-)?[0-9]^+$

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 $\{\epsilon, W_0, W_0W_1, W_0W_1W_2\ldots \mid W_0, W_1, W_2, \ldots \in L(E_0)\}$.
- 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.
 - * $[e_0, \ldots, e_n]$ donde e_i puede ser...
 - ...una letra.
 - ...a-b, conjunto de caracteres comprendidos entre a y b.
 - * Conjuntos complementados $[\wedge e_0, \ldots, e_n]$.
e.g. $[\wedge 0-9, a-z]$ 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.
 - * \backslash . 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.

$[\backslash +, \backslash -]^?([1-9] [0-9]^* \mid 0)$

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

$(*) \text{ IDEN} \equiv \underline{\text{Letra}} (\underline{\text{Letra}} \mid \underline{\text{Dig}})^*$
 $\text{Letra} \equiv [\text{a-z}, \text{A-Z}, _]$
 $\text{Dig} \equiv [0-9]$

e.g. Números enteros

$(*) \text{ LENT} \equiv \underline{\text{Signo}}? (0 \mid \underline{\text{DPos}} \underline{\text{Dig}}^*)$
 $\text{Signo} \equiv [\backslash+, \backslash-]$
 $\text{DPos} \equiv [1-9]$
 $\text{Dig} \equiv [0-9]$

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

$(*) \text{ LREAL} \equiv \underline{\text{LENT}} (\underline{\text{PDEC}} \mid \underline{\text{PEXP}} \mid \underline{\text{PDEC}} \underline{\text{PEXP}})$
 $\text{PDEC} \equiv \backslash. (\underline{\text{Dig}}^* \underline{\text{DPos}} \mid 0)$
 $\text{PEXP} \equiv (\text{E} \mid \text{e}) \underline{\text{LENT}}$
 $\text{LENT} \equiv \underline{\text{Signo}}? (0 \mid \underline{\text{DPos}} \underline{\text{Dig}}^*)$
 $\text{Signo} \equiv [\backslash+, \backslash-]$
 $\text{DPos} \equiv [1-9]$
 $\text{Dig} \equiv [0-9]$

Ignorables

[I] SEP $\equiv [\text{'}, \backslash t, \backslash n, \backslash r, \backslash b]$, posibles separadores.

[I] COM $\equiv \# [\wedge \backslash n]^* \backslash n$, e.g. $\#$ Esto es un comentario. $\backslash n$

e.g DR de apartado 2.1

(*) PAP $\equiv \backslash ($

(*) PCIERRE $\equiv \backslash)$

(*) NUM $\equiv \text{Signo? } (0 \mid \underline{\text{DPos}} \underline{\text{Dig}}^*)$

Signo $\equiv [\backslash +, \backslash -]$

DPos $\equiv [1-9]$

Dig $\equiv [0-9]$

(*) MAS $\equiv \backslash +$

(*) MENOS $\equiv \backslash -$

(*) POR $\equiv \backslash *$

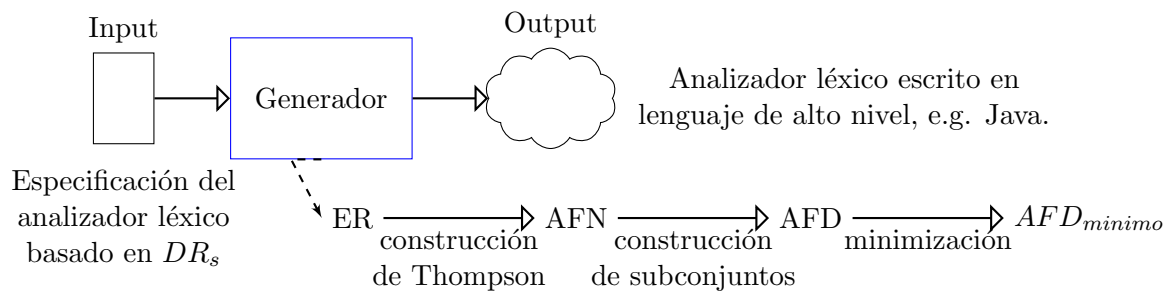
(*) DIV $\equiv \backslash /$

[I] SEP $\equiv [\text{'}, \backslash t, \backslash n, \backslash r, \backslash b]$

[I] COM $\equiv \# [\wedge \backslash n]^* \backslash n$

2.3 Implementación

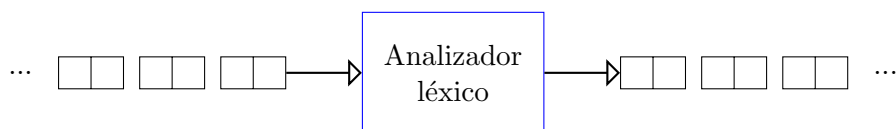
2.3.1 Generadores de analizadores léxicos



2.3.2 Implementación manual

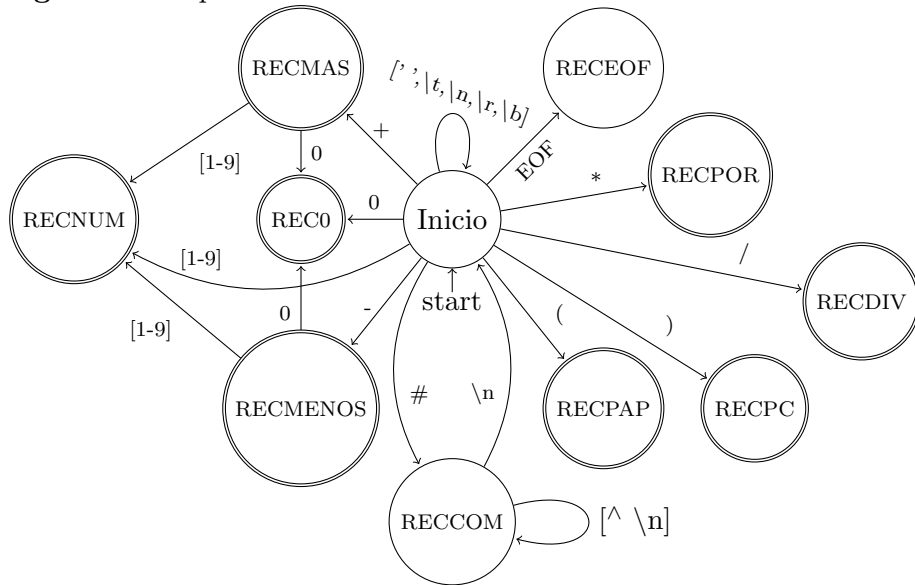
Se implementa utilizando AFD_s donde las transiciones a estados de error no se dibujan, son implícitas.

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  $\leftarrow$  EstadoInicial
lexema  $\leftarrow$  ""
loop {
    switch(Estado) {
        case  $S_0$ 
        ...
        case  $S_n$ 
    }
}

case  $S_i$ 
si sigCar  $\in T_0$ 
    Estado  $\leftarrow$  SigEstado
    lexema  $\leftarrow$  lexema + sigCar
    (lexema solo cambia en estados no ignorables)
    Actualizar sigCar
si no si sigCar  $\in T_1$ 
    Estado  $\leftarrow$  SigEstado
    lexema  $\leftarrow$  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 = # textbf then 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
}

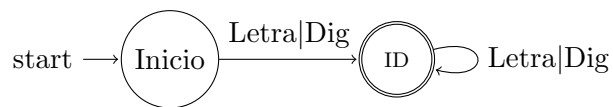
transitaIgnorando(S) {
  Estado ← S
  Actualiza sigCar
}
```

e.g.

Especificación léxica

EVALUA	(*) $ID \equiv \underline{\text{Letra}} (\underline{\text{Letra}} \mid \underline{\text{Dig}})^*$
5 + x	$\text{Letra} \equiv [\text{a-z}, \text{A-Z}, -]$
DONDE	$\text{Dig} \equiv [0-9]$
x = 27	(*) $\text{EVALUA} \equiv [\text{E}, \text{e}][\text{V}, \text{v}][\text{A}, \text{a}][\text{L}, \text{l}][\text{U}, \text{u}][\text{A}, \text{a}]$
	(*) $\text{DONDE} \equiv [\text{D}, \text{d}][\text{O}, \text{o}][\text{N}, \text{n}][\text{D}, \text{d}][\text{E}, \text{e}]$

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.

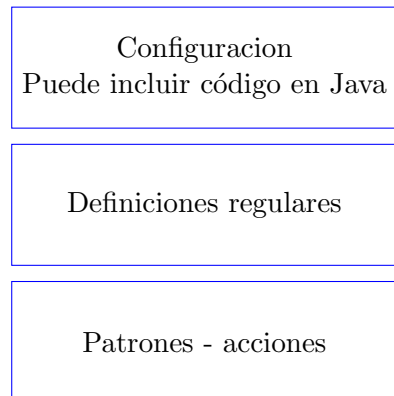


Hay un ejemplo de codificación en el campus, tiene una errata en el diagrama de transiciones.

2.3.3 Implementación mediante herramientas

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

Formato de JLex:

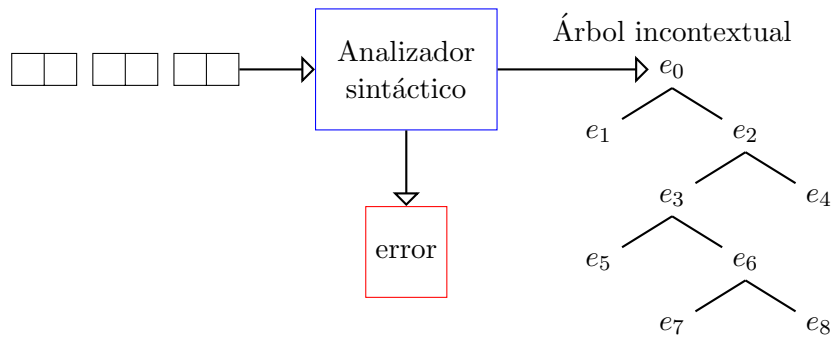


Para que una DR pueda referirse a otras tienen que haberse definido antes, lo que en las DR_s subrayamos tiene que ir entre {}, hacen sustituciones literales (no ponen paréntesis).

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) $\Rightarrow \subseteq (NUT)^* \times (NUT)^*$
 $\alpha A \beta \wedge A \rightarrow \gamma \in P$ entonces $\alpha A \beta \Rightarrow \alpha \gamma \beta$
- Se considera \Rightarrow_* aplicar cero o más veces \Rightarrow .

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

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

$N \Rightarrow NB \Rightarrow NBB \Rightarrow N1B \Rightarrow B1B \Rightarrow B10 \Rightarrow 010$

Derivación mas a la izquierda

$N \Rightarrow NB \Rightarrow NBB \Rightarrow BBB \Rightarrow 0BB \Rightarrow 01B \Rightarrow 010$

Derivación mas a la derecha

$N \Rightarrow NB \Rightarrow N0 \Rightarrow NB0 \Rightarrow N10 \Rightarrow B10 \Rightarrow 010$

3.1.2 Árbol de análisis sintáctico

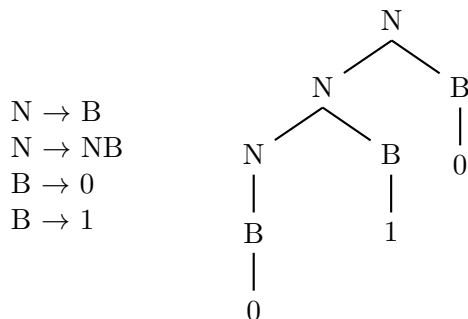
La idea es obtener una representación única para cada sentencia.

Árboles

- La raíz está etiquetada con el símbolo inicial.
- Están ordenados, hay un orden 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 $A \rightarrow \alpha \in P$.

$w \in L(G) \Leftrightarrow \begin{array}{c} s \\ \triangle \\ w \end{array} \text{ (es un árbol de análisis sintáctico)}$

e.g.



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 [...]...]

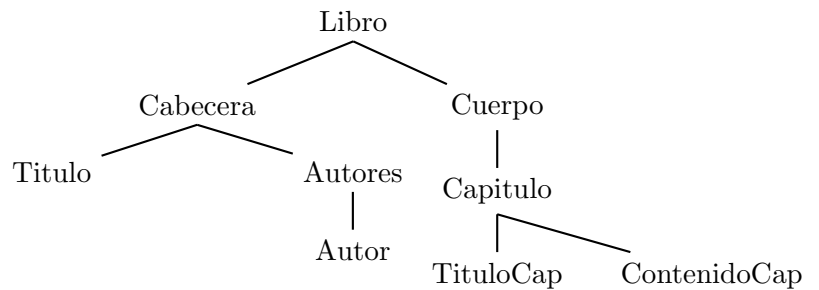


Figure 3.2.1.1 : Árbol informal.

Clases léxicas
Autores
Capitulo
CAP
CCIE
Texto

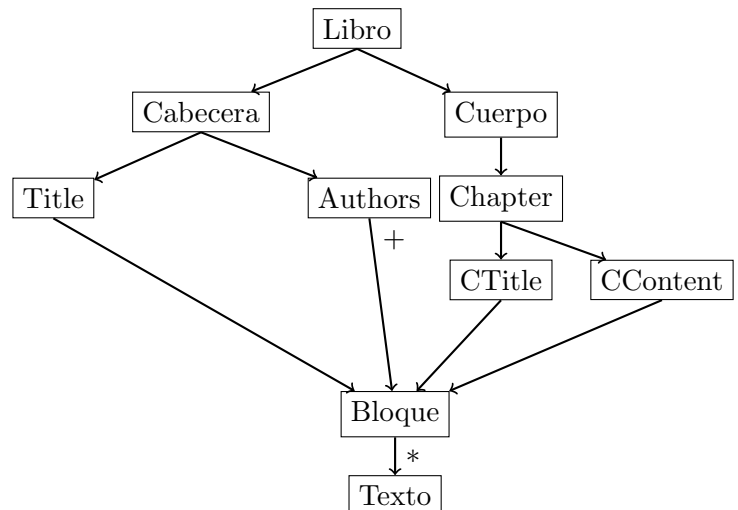


Figure 3.2.1.2 : Árbol formal.

3.2.2 Patrones para secuencias

- Secuencia de 1 o más I_s , separados por \square .

e.g. $I \mid I \square I \mid I \square I \square I \dots$

$LI \rightarrow I$

$LI \rightarrow LI \square I$

e.g.

$NUM \rightarrow B$

$NUM \rightarrow NUM , B$

- Secuencia de 0 o más I_s , separados por \square .

$S \rightarrow \varepsilon$

$S \rightarrow LI$

$LI \rightarrow I$

$LI \rightarrow LI \square I$

e.g. Declaraciones

$Decs \rightarrow \varepsilon$

$Decs \rightarrow LDEC$

$LDEC \rightarrow Dec$

$LDEC \rightarrow LDEC ; Dec$

Como caso particular de los patrones anteriores tenemos aquellos en que \square es ε , no están separados por nada.

- Secuencia de 1 o más I_s .

$LI \rightarrow I$

$LI \rightarrow LI I$

- Secuencia de 0 o más I_s .

$S \rightarrow \varepsilon$

$S \rightarrow LI$

$LI \rightarrow I$

$LI \rightarrow LI I$

También es posible usar un terminador en lugar de un separador para cada ítem.

e.g. $I \square I \square I \square I \square \dots$ donde $I \square$ es un ítem.

- Secuencia de 1 o más I_s terminados en \square .

$LI \rightarrow BI$

$LI \rightarrow LI BI$

$BI \rightarrow I \square$

- Secuencia de 0 o más I_s terminados en \square .

$S \rightarrow \varepsilon$

$S \rightarrow LI$

$LI \rightarrow BI$
 $LI \rightarrow LI BI$
 $BI \rightarrow I \square$

e.g. Libro

Libro \rightarrow Cabecera Cuerpo
 Cabecera \rightarrow Title Authors
 Title \rightarrow Titulo Bloque
 Authors \rightarrow Autores LBloques
 LBloques \rightarrow Bloque
 LBloques \rightarrow LBloques Bloque
 Cuerpo \rightarrow LChapter
 LChapter \rightarrow Chapter
 LChapter \rightarrow LChapter Chapter

Chapter \rightarrow CTitle CContent
 CTitle \rightarrow Capitulo Bloque
 CContent \rightarrow Bloque
 Bloque \rightarrow [CTexto]
 CTexto $\rightarrow \varepsilon$
 CTexto \rightarrow LTexto
 LTexto \rightarrow Texto
 LTexto \rightarrow LTexto Texto

3.2.3 Patrones para operadores

La prioridad indica el orden en que los operadores se evalúan, si dos tienen la misma prioridad se consulta la asociatividad, a izquierdas o a derechas.

e.g.

$\underline{\underline{5 + 6 - 7 * 8 + 9}}$ * es el más prioritario.
 + y - son igual de prioritarios pero asocian a izquierdas.

Cada operador tiene un nivel de prioridad, pueden existir múltiples operadores con un mismo nivel.

- Operadores binarios infijos.
 Pueden asociar a izquierdas
e.g. $\underline{\underline{5 + 6 + 7}}$
 O a derechas.
e.g. $\underline{\underline{5 + 6 + 7}}$
 O no asociar, no está permitido encadenar operadores.
e.g. $5 + 6 + 7$
- Operadores unarios prefijos.
e.g. -5
 Pueden ser...
 - ...**asociativos**, pueden encadenarse varios.
 - ...**no asociativos**, no pueden encadenarse.
 Asocian siempre a derechas.

- Operadores unarios postfijos.

e.g. $5+$, $x++$

Pueden ser...

- ...**asociativos**, pueden encadenarse varios.
- ...**no asociativos**, no pueden encadenarse.

Asocian siempre a izquierdas.

e.g. Lenguaje

- Numeros y variables.

- $+$, $-$, $*$, $/$, $-$ unario.

Operador	Aridad	Tipo	Prioridad	Asociatividad
$+, -$	2		0	izquierda
$*, /$	2		1	izquierda
$-$	1	prefijo	2	si

$$\underline{\underline{5 + 6 - \underline{\underline{-7 * 8 + 9}}}}$$

Si existen varios operadores con la misma prioridad pero que asocian distinto porque existen multiples interpretaciones.

Operador	Aridad	Tipo	Prioridad	Asociatividad
$+$	2		0	izquierda
$-$	2		0	derecha
$*, /$	2		1	izquierda
$-$	1	prefijo	2	si

$$\underline{\underline{5 + 6 + 7}}$$

$$\underline{\underline{5 + 6 + 7}}$$

$\text{Exp} \rightarrow \text{Variable}$

$\text{Exp} \rightarrow \text{Numero}$

$\text{Exp} \rightarrow \text{OpUn Exp}$

$\text{Exp} \rightarrow \text{Exp OpBin Exp}$

$\text{OpUn} \rightarrow -$

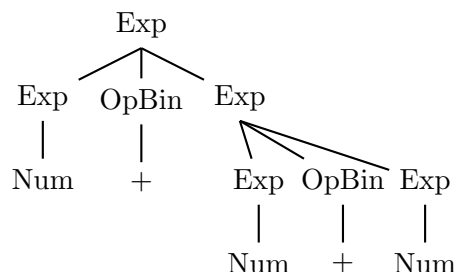
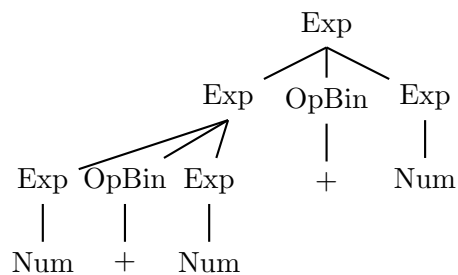
$\text{OpBin} \rightarrow +$

$\text{OpBin} \rightarrow -$

$\text{OpBin} \rightarrow *$

$\text{OpBin} \rightarrow /$

$\text{Num} + \text{Num} + \text{Num}$



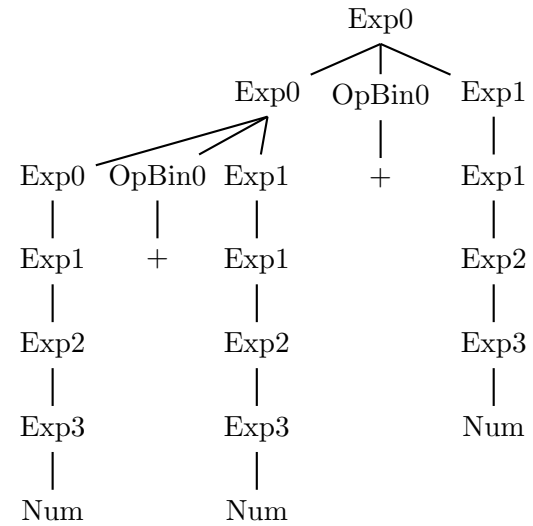
Como hemos encontrado dos posibles árboles de derivación la gramática es ambigua y no vale.

Cada nivel de prioridad tendrá su propio terminal.

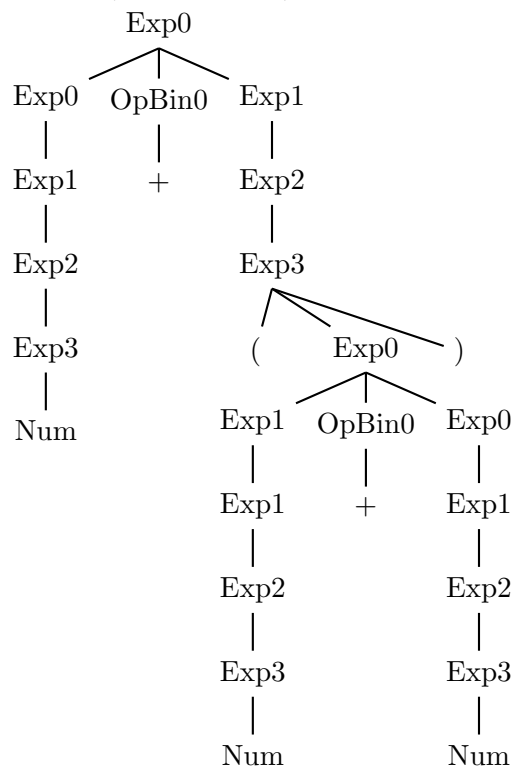
Si crece a izquierdas reduzco el nivel de prioridad de la derecha, si es a derechas baja el de la izquierda y si no asocia ambos aumentan.

$\text{Exp0} \rightarrow \text{Exp0 OpBin0 Exp1}$
 $\text{Exp0} \rightarrow \text{Exp1}$
 $\text{Exp1} \rightarrow \text{Exp1 OpBin1 Exp2}$
 $\text{Exp1} \rightarrow \text{Exp2}$
 $\text{Exp2} \rightarrow \text{Exp2 OpBin2 Exp3}$
 $\text{Exp2} \rightarrow \text{Exp3}$
 $\text{Exp3} \rightarrow (\text{Exp0})$
 $\text{Exp3} \rightarrow \text{Num}$
 $\text{Exp3} \rightarrow \text{Var}$
 $\text{OpBin0} \rightarrow +$
 $\text{OpBin0} \rightarrow -$
 $\text{OpBin1} \rightarrow *$
 $\text{OpBin1} \rightarrow /$
 $\text{OpUn2} \rightarrow -$

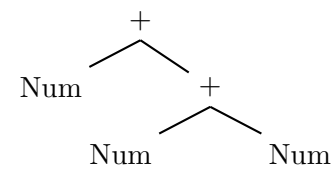
$\text{Num} + \text{Num} + \text{Num}$



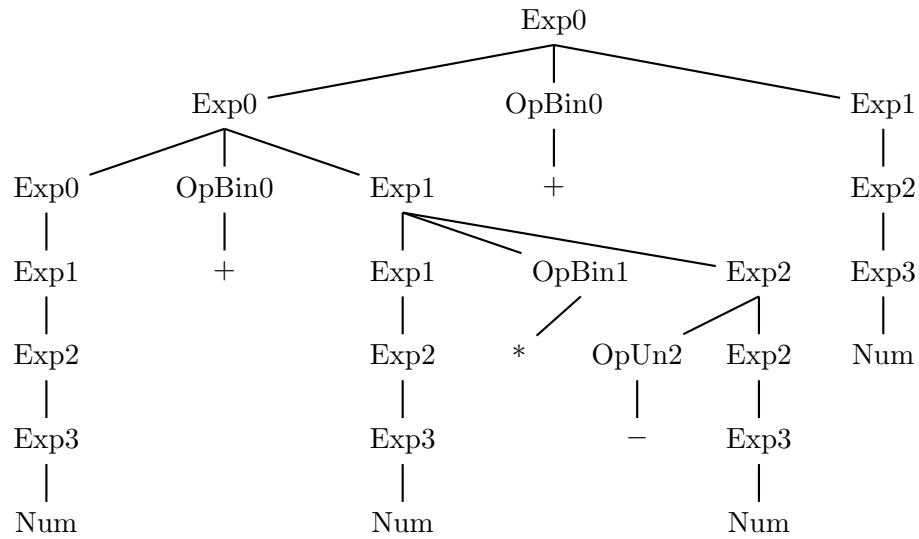
$\text{Num} + (\text{Num} + \text{Num})$



El árbol de la izquierda sería equivalente al siguiente.



$Num - Num * -Num + Num$



e.g. Lenguaje

Expresiones básicas: números e identificadores, se pueden usar ().

Operador	Prioridad	Aridad	Tipo	Asociatividad
==, !=	0	2		no
+	1	2		izq.
*	1	1	prefijo	no
=	2	2		der.
++	2	1	postfijo	si
~	3	1	prefijo	si
<<	3	1	postfijo	no

$Exp0 \rightarrow Exp1 Op0 Exp1$

$Exp0 \rightarrow Exp1$

$Exp1 \rightarrow Exp1 + Exp2$

$Exp1 \rightarrow * Exp2$

$Exp1 \rightarrow Exp2$

$Exp2 \rightarrow Exp3 = Exp2$

$Exp2 \rightarrow Exp2 ++$

$Exp2 \rightarrow Exp3$

$Exp3 \rightarrow \sim Exp3$

$Exp3 \rightarrow Exp4 <<$

$Exp3 \rightarrow Exp4$

$Exp4 \rightarrow Num$

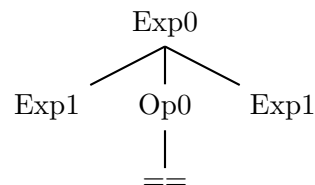
$Exp4 \rightarrow Id$

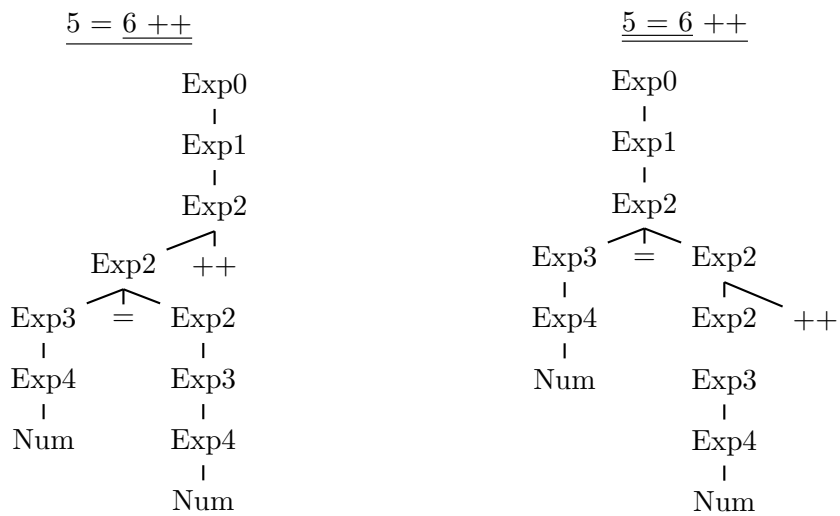
$Exp4 \rightarrow (Exp0)$

$Op0 \rightarrow ==$

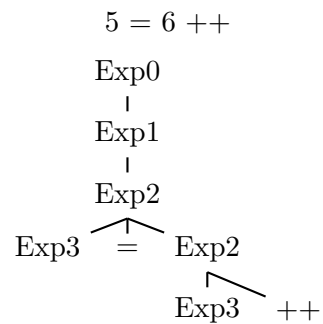
$Op0 \rightarrow !=$

$Num == Num == Num$, está prohibido.

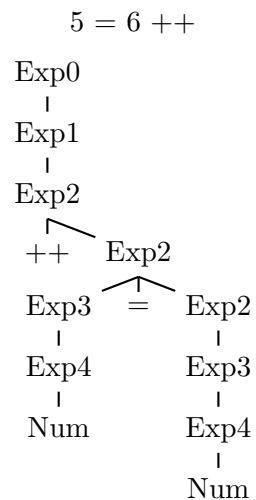




Supongamos que el operador ++ no asocia.
 Ahora, $\text{Exp2} \rightarrow \text{Exp3} ++$



Supongamos que el operador ++ es prefijo
 y asocia.
 Ahora, $\text{Exp2} \rightarrow ++ \text{Exp2}$



- Los operadores prefijos que asocian son incompatibles con los operadores que asocian a izquierdas.

- Los operadores postfijos que asocian son incompatibles con los operadores que asocian a derechas.

e.g. Dada g obtenemos su tabla de operadores.

$\text{Exp0} \rightarrow \text{Exp0} \rightarrow \text{Exp1}$

$\text{Exp0} \rightarrow \text{Exp1} \sim \text{Exp1}$

$\text{Exp0} \rightarrow \# \text{Exp1}$

$\text{Exp0} \rightarrow \text{Exp1}$

$\text{Exp1} \rightarrow \text{Exp1} @ \text{Exp2}$

$\text{Exp1} \rightarrow \text{Exp2} \wedge \text{Exp2}$

$\text{Exp1} \rightarrow \text{Exp2}$

$\text{Exp2} \rightarrow \text{Exp2} *$

$\text{Exp2} \rightarrow \text{Exp3}$

$\text{Exp3} \rightarrow \text{Num}$

$\text{Exp3} \rightarrow (\text{Exp0})$

Operador	Prioridad	Aridad	Tipo	Asociatividad
\rightarrow	0	2		izq.
\sim	0	2		no
$\#$	0	1	prefijo	no
$@$	1	2		izq.
\wedge	1	2		no
$*$	2	1	postfijo	si

Supongamos que en un mismo nivel de prioridad existen:

- $+$, binario que asocia a izquierdas.
- $-$, unario prefijo que no asocia.
- $*$, unario postfijo que asocia.

Hay problemas con estos operadores por conflictos de crecimiento, si un operador crece a izquierdas y el otro a derechas los árboles que generan son incompatibles.

No	A izquierdas	A derechas	Si	Estado
$+, -, *$				Ok
$+, -$			$*$	Ok
$+, *$			$-$	Ok
$-, *$	$+$			Ok
$-, *$		$+$		Ok
$-$	$+$		$*$	Ok
$*$		$+$	$-$	Ok
	$+$		$*, -$	X
$-$		$+$	$*$	X
$*$	$+$		$-$	X
		$+$	$*, -$	X

e.g. Lenguaje, especificación completa.

Un programa es una secuencia de una o más instrucciones separadas por $;$.

Se dispone de los siguientes tipos de instrucciones:

- Asignación.

Una variable seguida de \leftarrow seguida de una expresión.

- Si.

IF expresión Instrucciones **FI**

- Si-Sino.

IF expresión

I_0

ELSE

I_1

FI

- Bucle.

WHILE expresion **DO**

I

OD

- Bloque.

{ Secuencia de I_s separadas por ; }

Las expresiones básicas las componen números (literales) y variables.

Se dispone de los siguientes operadores:

- Comparación. $<$, $>$, $==$, $!=$, $<=$, $>=$

Binarios, menor nivel de prioridad y no asocian.

- Aritméticos. $+$, $-$, $*$, $/$

Binarios, más prioridad que los de comparación y asocian a izquierdas.

- Lógicos. and , or

Binarios, más prioridad que los aritméticos y asocian a derechas.

- Unarios. $!$, $-$

Prefijos, más prioritarios que los lógicos y asocian.

Se pueden utilizar paréntesis, (...).

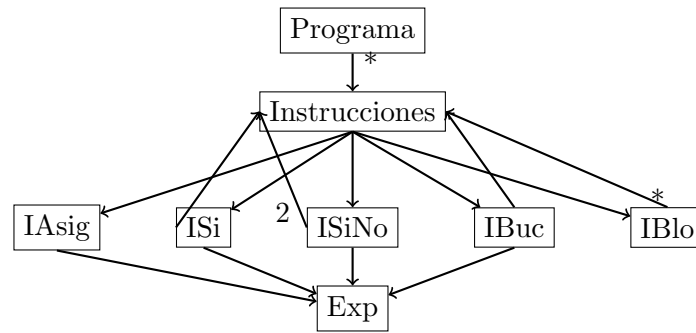
Un posible programa sería:

$x \leftarrow 25$

WHILE $x \leq 50$ **DO**

$x \leftarrow x + 1$

OD



Operador	Prioridad	Aridad	Tipo	Asociatividad
<, >, ==,				
!=, <=, >=	0	2		no
+, -, *, /	1	2		izq.
and, or	2	2		der.
!, -	3	1	prefijo	si

Programa \rightarrow LI

LI \rightarrow I

LI \rightarrow LI ; I

I \rightarrow IAsig

I \rightarrow ISi

I \rightarrow ISino

I \rightarrow IBuc

I \rightarrow IBlo

IAsig \rightarrow Var \leftarrow Exp0

ISi \rightarrow IF Exp0 I FI

ISino \rightarrow IF Exp0 I ELSE I FI

IBuc \rightarrow WHILE Exp0 DO I OD

IBlo \rightarrow { S }

S \rightarrow ε

S \rightarrow LI

Exp0 \rightarrow Exp1 Op0 Exp1

Exp0 \rightarrow Exp1

Exp1 \rightarrow Exp1 Op1 Exp2

Exp1 \rightarrow Exp2

Exp2 \rightarrow Exp3 Op2 Exp2

Exp2 \rightarrow Exp3

Exp3 \rightarrow Op3 Exp3

Exp3 \rightarrow Exp4

Exp4 \rightarrow (Exp0)

Exp4 \rightarrow Num

Exp4 \rightarrow Var

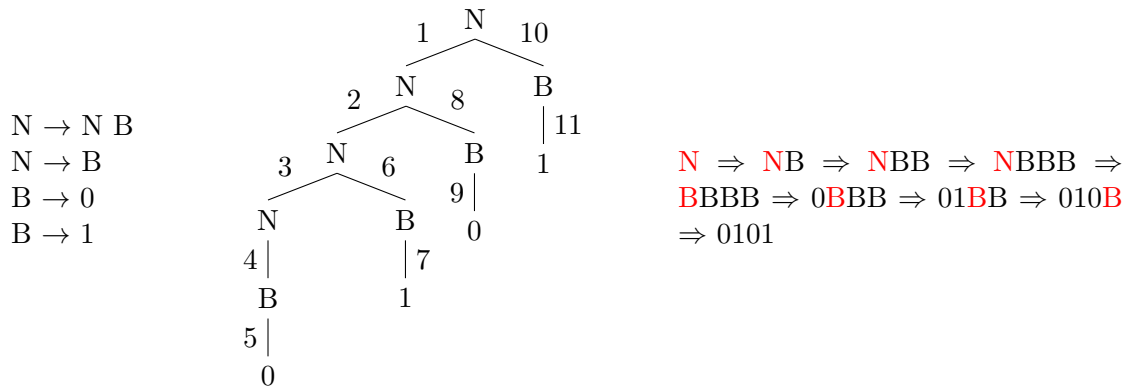
OpX se corresponde con los operadores del nivel X.

3.3 Implementación

Vamos a ver analizadores descendentes (top-down parsers), concretamente implementaremos analizadores sintácticos predictivos recursivos descendentes.

Los árboles expanden de izquierda a derecha equivalente a una derivación más a la izquierda, los algoritmos que veremos se basan en este tipo de derivación.

e.g. 0101



Pila de analisis (\leftarrow)	Entrada	Movimiento
N	0101	Expandir $N \rightarrow N B$
NB	0101	Expandir $N \rightarrow N B$
NBB	0101	Expandir $N \rightarrow N B$
NBBB	0101	Expandir $N \rightarrow B$
BBBB	0101	Expandir $B \rightarrow 0$
0BBB	0101	Desplazar 0
BBB	0 01	Expandir $B \rightarrow 1$
1BB	0 01	Desplazar 1
BB	01 01	Expandir $B \rightarrow 0$
0B	01 01	Desplazar 0
B	010 1	Expandir $B \rightarrow 1$
1	010 1	Desplazar 1
	0101	Aceptar

Gramática	Los no terminales se interpretan como funciones que reconocen partes de la entrada.
$N' \rightarrow N \vdash$	Las decisiones se basan en un token, el algoritmo siempre lleva un token adelantado.
$N \rightarrow B R$	Los terminales consumen el token si coincide y rechazan la entrada si no.
$R \rightarrow B R$	
$R \rightarrow \varepsilon$	
$B \rightarrow 0$	Para asegurarse de que la cadena pertenece y no solo un prefijo siempre se añade una producción adicional, N' , que llama al símbolo inicial y luego reconoce el fin de fichero.
$B \rightarrow 1$	

Pseudocódigo

```

Np() {
  N();
  rec(EOF);
}

N() {
  B();
  R();
}

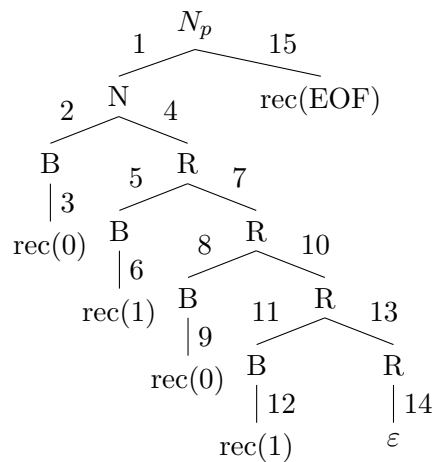
B() {
  if sigToken == 0 then
    rec(0);
  else
    rec(1);
  fi
}

R() {
  if sigToken ∈ {0,1} then
    B();
    R();
  fi
}

rec(T) {
  if sigToken == T then
    sigToken ← sigToken();
  else
    error
  fi
}

```

e.g. 0101-

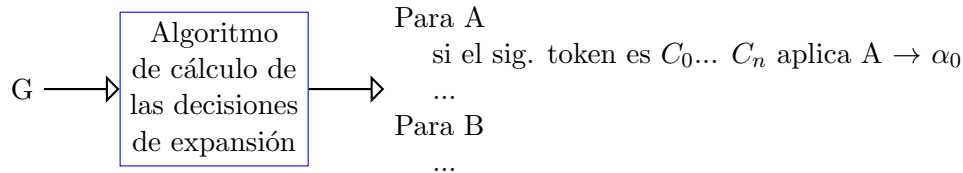


¿Funciona este método para cualquier gramática?

Como se puede ver con un único token no es posible decidir que producción usar.

$$\begin{array}{ll} A \rightarrow 0 X & L \rightarrow L i \\ A \rightarrow 0 Y & L \rightarrow i \end{array}$$

En general la recursión a izquierdas es problemática para esta metodología. La solución es transformar estas gramáticas con métodos que veremos más adelante.



Con este método podemos implementar gramáticas LL(1).

Si tenemos $A \rightarrow \alpha$ tenemos que preguntarnos por qué token pueden empezar las cadenas de α y comprobar sigToken.

Vamos a definir una función que emplearemos más adelante:

PRIM(α), primeros de α .

Son todos los terminales que pueden comenzar sentencias derivables desde α . Además, $\varepsilon \in \text{PRIM}(\alpha)$ si $\alpha \Rightarrow^* \varepsilon$.

e.g.

$$\begin{array}{ll} N \rightarrow B R & \\ R \rightarrow B R & \\ R \rightarrow \varepsilon & \text{PRIM}(N) = \{0, 1\} \\ B \rightarrow 0 & \text{PRIM}(B R) = \{0, 1\} \\ B \rightarrow 1 & \end{array}$$

Luego $A \rightarrow \alpha$ se podrá aplicar durante la expansión si el siguiente símbolo $\in \text{PRIM}(\alpha) - \{\varepsilon\}$. Al conjunto de estos símbolos se le denomina directores, $\text{Dir}(A \rightarrow \alpha)$.

$$\text{PRIM}(\alpha) - \{\varepsilon\} \subseteq \text{Dir}(A \rightarrow \alpha)$$

Introducimos un operador, concatenación en modulo ε denotada por el símbolo \oplus .

$$\Gamma_0 \oplus \Gamma_1 = \begin{array}{ll} (\Gamma_0 - \{\varepsilon\}) \cup \Gamma_1 & \text{si } \varepsilon \in \Gamma_0 \\ (\Gamma_0 - \{\varepsilon\}) \cup \emptyset & \text{c.o.c.} \end{array}$$

e.g.

$$\begin{array}{l} \{a, b\} \oplus \{c, d\} = \{a, b\} \\ \{a, b, \varepsilon\} \oplus \{c, d\} = \{a, b, c, d\} \\ \{a, b, \varepsilon\} \oplus \{c, d, \varepsilon\} = \{a, b, c, d, \varepsilon\} \end{array}$$

Sea **a** un terminal y **A** un no terminal.

$$\text{PRIM}(a) = \{a\}$$

$$\begin{aligned}\text{PRIM}(\varepsilon) &= \{\varepsilon\} \\ \text{PRIM}(A) &= \cup \text{PRIM}(\alpha_i) \quad \text{donde } \{\alpha \mid A \rightarrow \alpha\}\end{aligned}$$

En general:

$$\text{PRIM}(X_0 \dots X_n) = \text{PRIM}(X_0) \oplus \text{PRIM}(X_1) \oplus \dots \oplus \text{PRIM}(X_n)$$

e.g.

$$\text{PRIM}(B \ R) = \text{PRIM}(B) \oplus \text{PRIM}(R)$$

e.g.

$$\begin{array}{ll} N \rightarrow B \ R & \\ R \rightarrow B \ R & \text{PRIM}(N) = \text{PRIM}(B) \oplus \text{PRIM}(R) \\ R \rightarrow \varepsilon & \text{PRIM}(R) = \text{PRIM}(B) \oplus \text{PRIM}(R) \cup \text{PRIM}(\varepsilon) \\ B \rightarrow 0 & \text{PRIM}(B) = \text{PRIM}(0) \cup \text{PRIM}(1) \\ B \rightarrow 1 & \end{array}$$

4 Semántica estática

5 Traducción