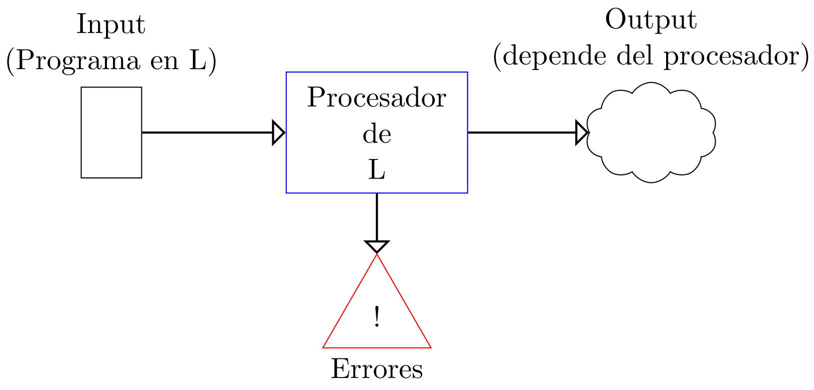
Procesadores de Lenguajes

David Antuña Rodríguez

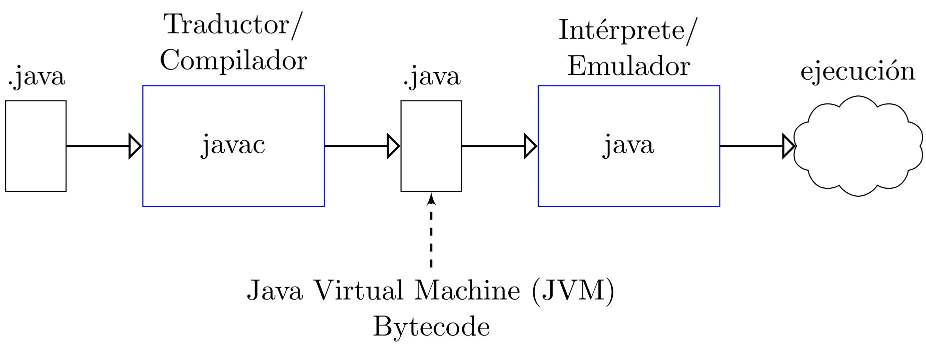
Contenidos

1 Introducción

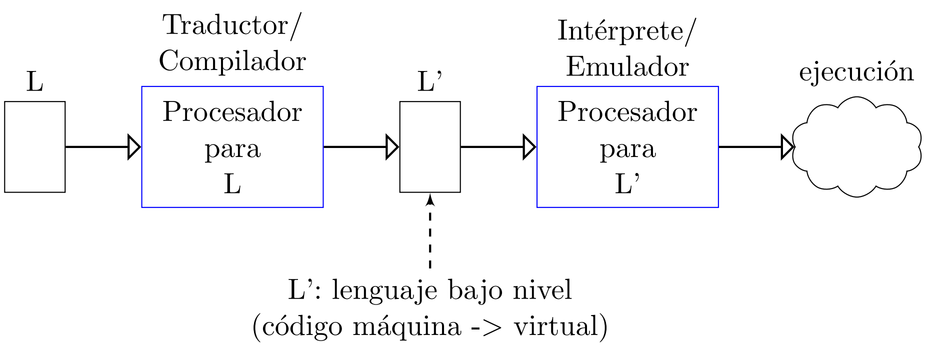


Compilador: Genera una representación en otro L (Alto nivel -> Bajo nivel).
Intérprete: Ejecuta una representación.

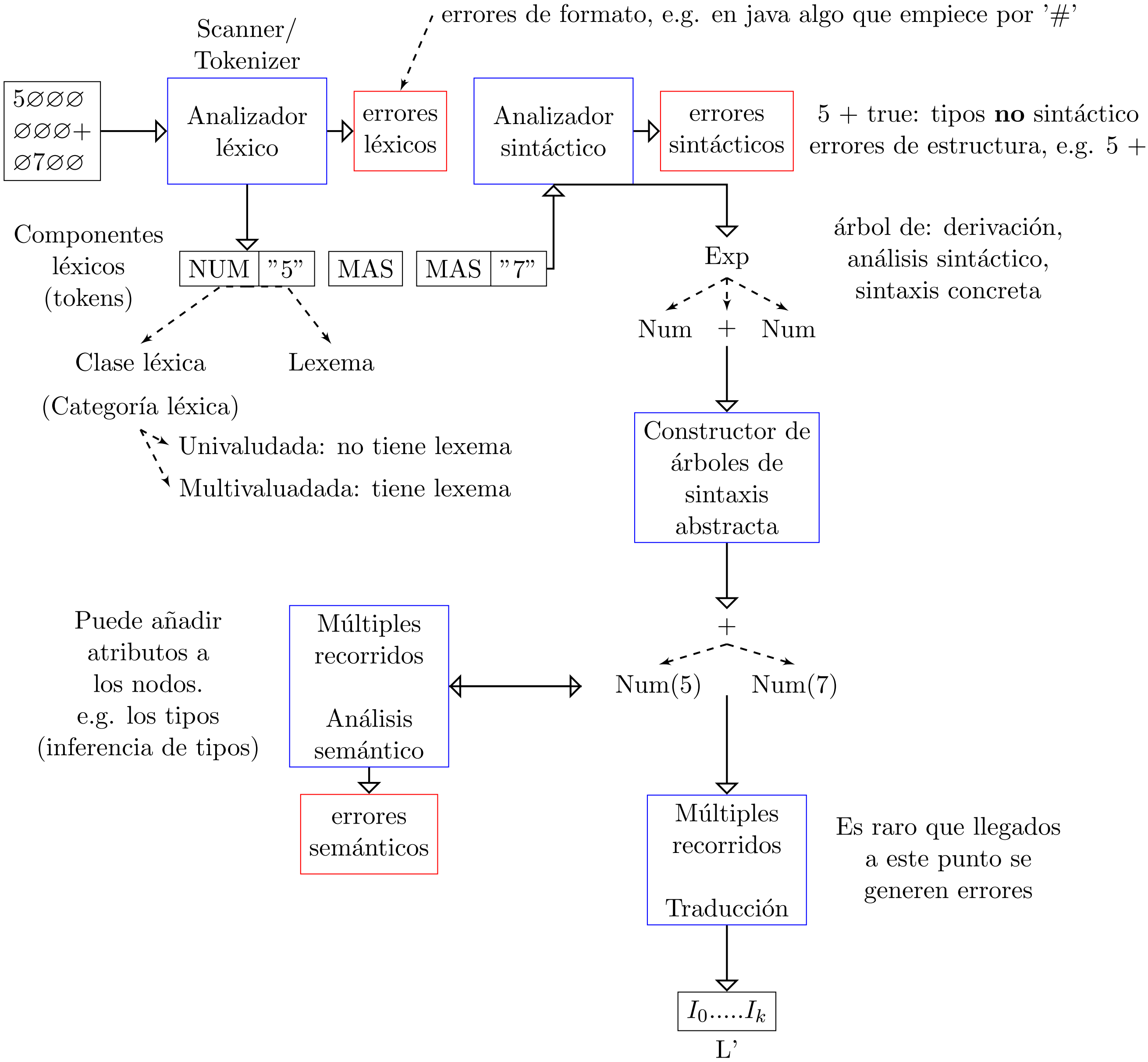
Java



Abstracción

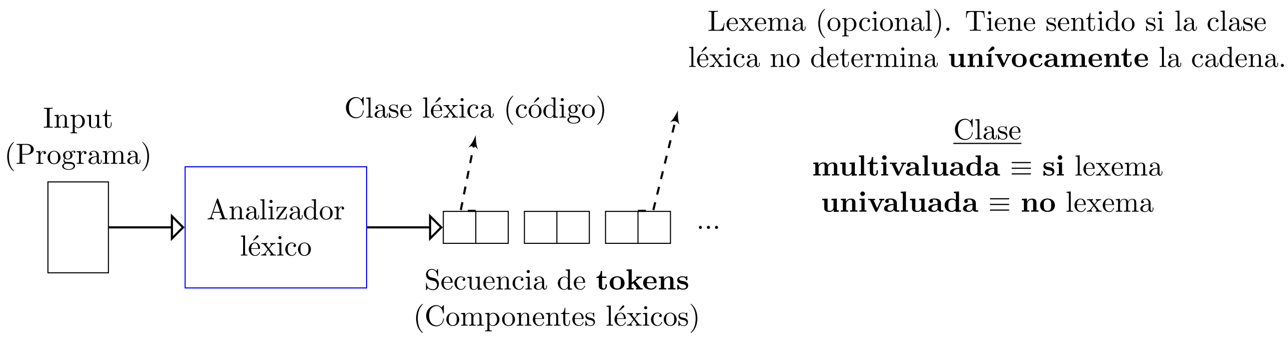


1.1 Traductor

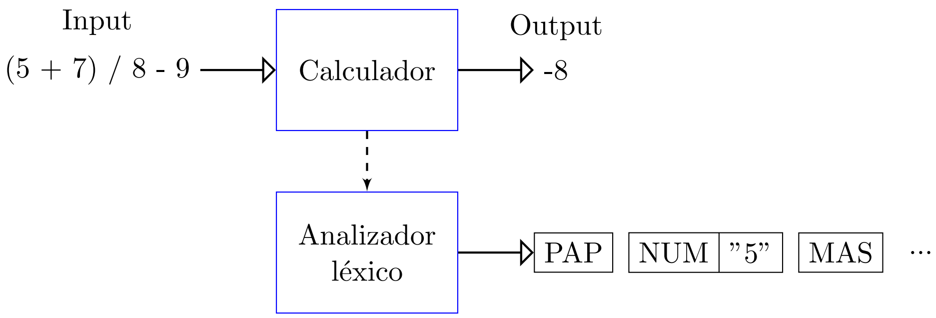


1.2 Intérprete

2 Análisis léxico



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 ??.

- 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 | y $\bullet$.
  - $\bullet$ tiene mayor prioridad que |.
  - $\bullet$ 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| ϵ). 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 ≡ $\underline{\text{Letra}}$ ($\underline{\text{Letra}}$ | $\underline{\text{Dig}}$)∗
Letra ≡ [a-z, A-Z, _]
Dig ≡ [0-9]

e.g. Números enteros

(∗) LENT ≡ $\underline{\text{Signo}}$? (0 | $\underline{\text{DPos}}$ $\underline{\text{Dig}}$∗)
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 ≡ $\underline{\text{LENT}}$ ($\underline{\text{PDEC}}$ | $\underline{\text{PEXP}}$ | $\underline{\text{PDEC}}$ $\underline{\text{PEXP}}$)
PDEC ≡ \. ($\underline{\text{Dig}}$∗ $\underline{\text{DPos}}$ | 0)
PEXP ≡ (E | e) $\underline{\text{LENT}}$

LENT ≡ $\underline{\text{Signo}}$? (0 | $\underline{\text{DPos}}$ $\underline{\text{Dig}}$∗)
Signo ≡ [\+, \-]
DPos ≡ [1-9]
Dig ≡ [0-9]

Ignorables

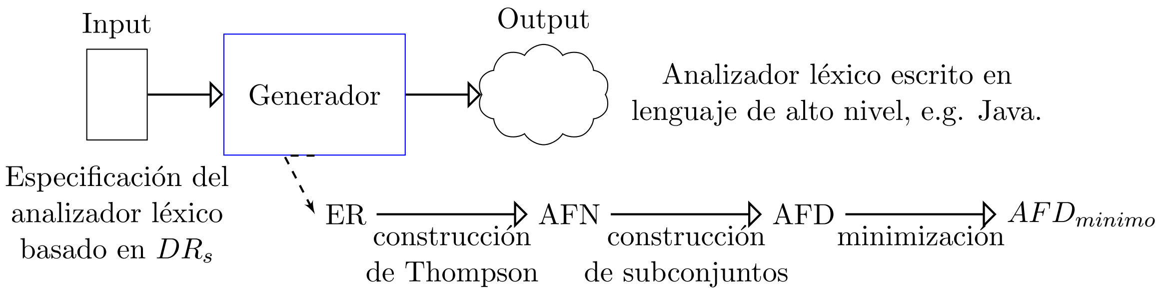
[I] SEP ≡ [' ', \t, \n, \r, \b], posibles separadores.
[I] COM ≡ # [^ \n]∗ \n, e.g. # Esto es un comentario.\n

e.g DR de **??**

(∗) 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



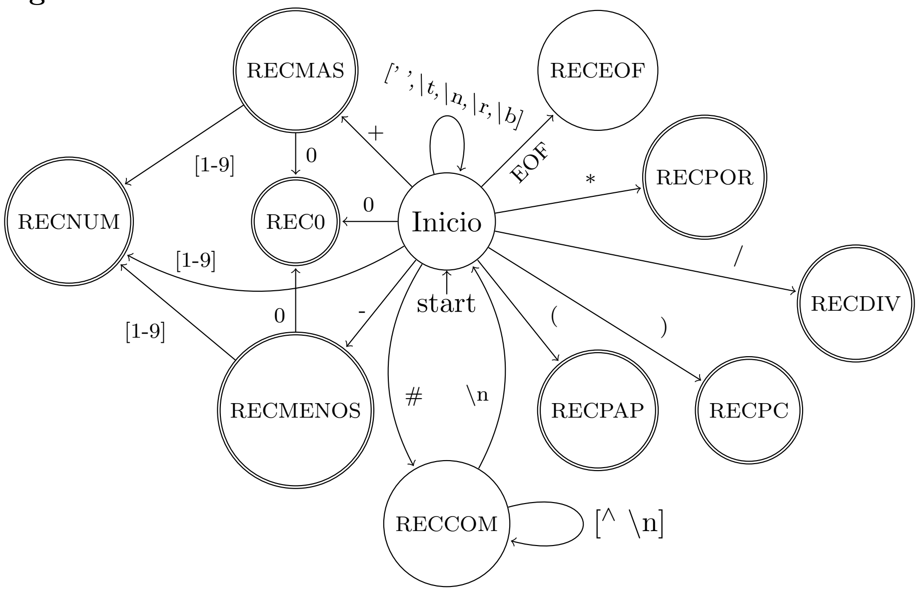
2.3.2 Implementación manual

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 **??**



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 $\in T_0$
    Estado ← SigEstado
    lexema ← lexema + sigCar
      (lexema solo cambia en estados no ignorables)
    Actualizar sigCar
  si no si sigCar $\in T_1$
    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.

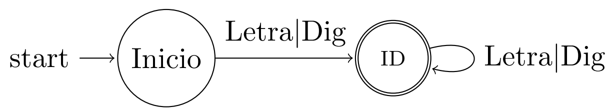
EVALUA
5 + x
DONDE
x = 27

Especificación léxica

...

(∗) ID ≡ $\underline{\text{Letra}}$ ($\underline{\text{Letra}}$ | $\underline{\text{Dig}}$)∗
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:

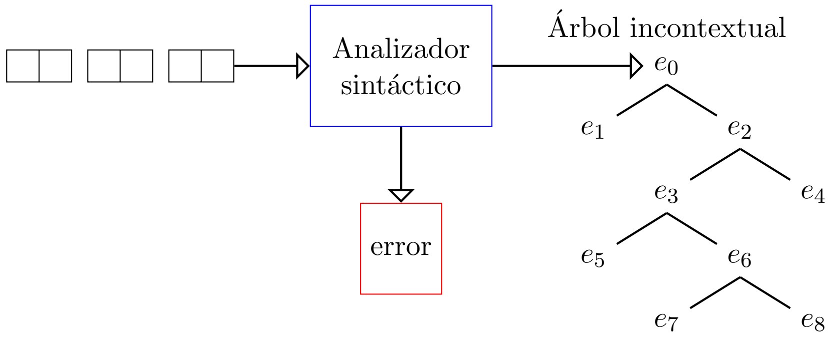
Configuracion Puede incluir código en Java
Definiciones regulares
Patrones - acciones

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
$\mathrm{N} \Rightarrow \mathrm{NB} \Rightarrow \mathrm{NBB} \Rightarrow \mathrm{N1B} \Rightarrow \mathrm{B1B} \Rightarrow \mathrm{B10} \Rightarrow 010$

Derivación mas a la izquierda
$\mathrm{N} \Rightarrow \mathrm{NB} \Rightarrow \mathrm{NBB} \Rightarrow \mathrm{BBB} \Rightarrow \mathrm{0BB} \Rightarrow \mathrm{01B} \Rightarrow 010$

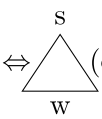
Derivación mas a la derecha
$\mathrm{N} \Rightarrow \mathrm{NB} \Rightarrow \mathrm{N0} \Rightarrow \mathrm{NB0} \Rightarrow \mathrm{N10} \Rightarrow \mathrm{B10} \Rightarrow 010$

3.1.2 Árbol de análisis sintáctico

La idea es obtener una representación única para cada sentencia.

Árboles

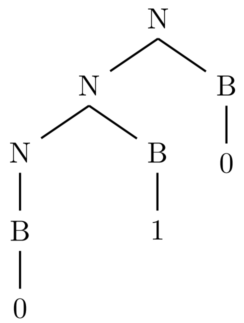
- La raíz está etiquetada con el símbolo inicial.
- Están ordenados, hay un órden en los hijos de los nodos (primer hijo, segundo hijo...).
- Los nodos internos están etiquetados por no terminales.
- Los nodos hoja están etiquetados por terminales o por ϵ.
- Si un nodo está etiquetado por A y sus hijos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

w $\in L(G) \Leftrightarrow$ 

(es un árbol de análisis sintáctico)

e.g.

N $\to$ B
N $\to$ NB
B $\to$ 0
B $\to$ 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 [...][...]

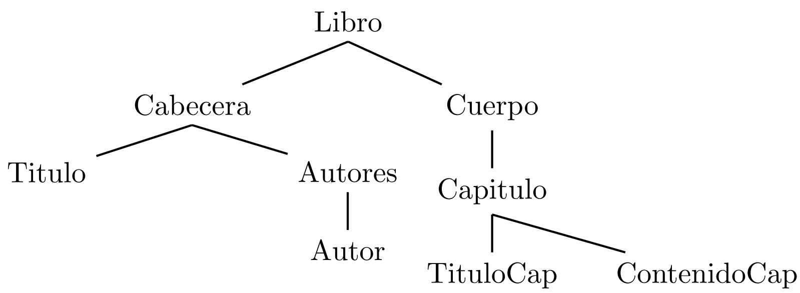


Figure 3.2.1.1 : Árbol informal.

Clases léxicas
Autores
Capitulo
CAP
CCIE
Texto

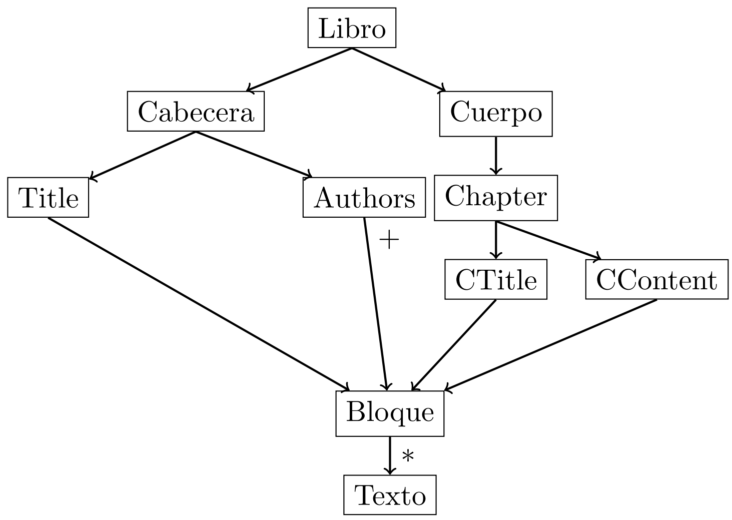


Figure 3.2.1.2 : Árbol formal.

3.2.2 Patrones para secuencias

- Secuencia de 1 o más I_s, separados por □.
  e.g. I | I □ I | I □ I □ I...

  LI → I
  LI → LI □ I

  e.g.
  NUM → B
  NUM → NUM , B

- Secuencia de 0 o más I_s, separados por □.
  S → ε
  S → LI
  LI → I
  LI → LI □ I

  e.g. Declaraciones
  Decs → ε
  Decs → LDEC
  LDEC → Dec
  LDEC → LDEC ; Dec

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

- Secuencia de 1 o más I_s.
  LI → I
  LI → LI I

- Secuencia de 0 o más I_s.
  S → ε
  S → LI
  LI → I
  LI → LI I

También es posible usar un terminador en lugar de un separador para cada item.
e.g. $\underline{\text{I □}}$ $\underline{\text{I □}}$ $\underline{\text{I □}}$... donde I □ es un item.

- Secuencia de 1 o más I_s terminados en □.
  LI → BI
  LI → LI BI
  BI → I □

- Secuencia de 0 o más I_s terminados en □.
  S → ε
  S → LI

LI → BI
LI → LI BI
BI → I □

e.g. Libro

Libro → Cabecera Cuerpo
Cabecera → Title Authors
Title → $\underline{\text{Titulo}}$ Bloque
Authors → $\underline{\text{Autores}}$ LBloques
LBloques → Bloque
LBloques → LBloques Bloque
Cuerpo → LChapter
LChapter → Chapter
LChapter → LChapter Chapter

Chapter → CTitle CContent
CTitle → $\underline{\text{Capitulo}}$ Bloque
CContent → Bloque
Bloque → [CTexto]
CTexto → ε
CTexto → LTexto
LTexto → Texto
LTexto → LTexto Texto

3.2.3 Patrones para operadores

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

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

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

- Operadores binarios infijos.
  Pueden asociar a izquierdas
  e.g. $\underline{\underline{5+6}+7}$

  O a derechas.
  e.g. $\underline{5+\underline{6+7}}$

  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

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

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

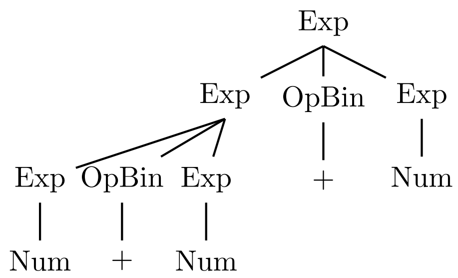
Operador	Aridad	Tipo	Prioridad	Asociatividad
+	2		0	izquierda
−	2		0	derecha
*,/	2		1	izquierda
−	1	prefijo	2	si

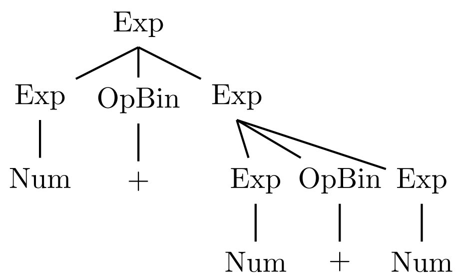
$\underline{5+\underline{\underline{6+7}}}$

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

Exp → Variable
Exp → Numero
Exp → OpUn Exp
Exp → Exp OpBin Exp
OpUn → −
OpBin → +
OpBin → −
OpBin → ∗
OpBin → /

Num + Num + Num



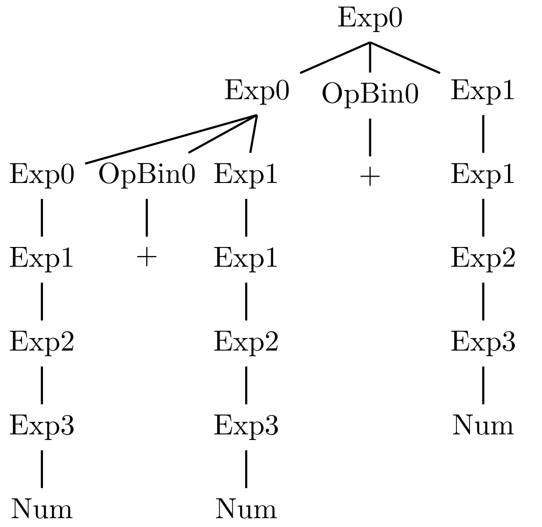


Como hemos encontrado dos posibles árboles de derivación la gramática es ambigua y no vale.

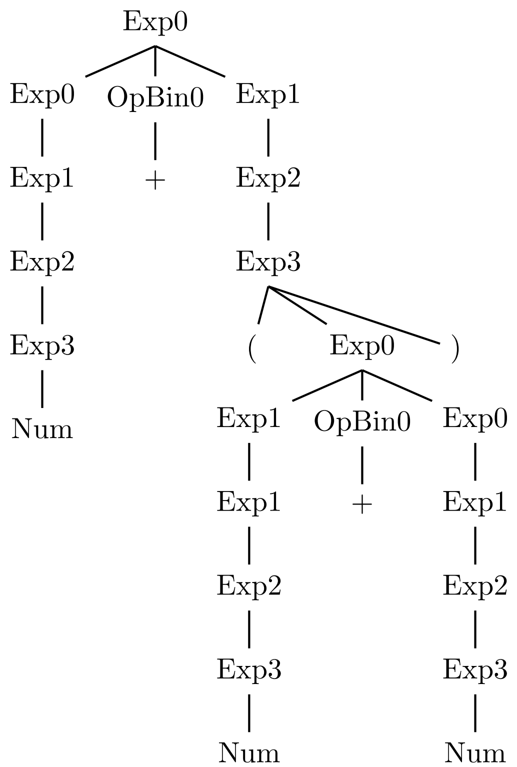
Cada nivel de prioridad tendrá su propio terminal.
Si crece a izquierdas reduzco el nivel de prioridad de la derecha, si es a derechas baja el de la izquierda y si no asocia ambos aumentan.

Exp0 → Exp0 OpBin0 Exp1
Exp0 → Exp1
Exp1 → Exp1 OpBin1 Exp2
Exp1 → Exp2
Exp2 → Exp2 OpBin2 Exp3
Exp2 → Exp3
Exp3 → (Exp0)
Exp3 → Num
Exp3 → Var
OpBin0 → +
OpBin0 → −
OpBin1 → ∗
OpBin1 → /
OpUn2 → −

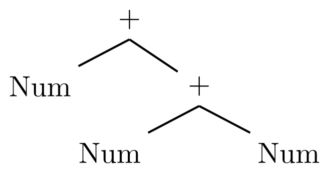
Num + Num + Num



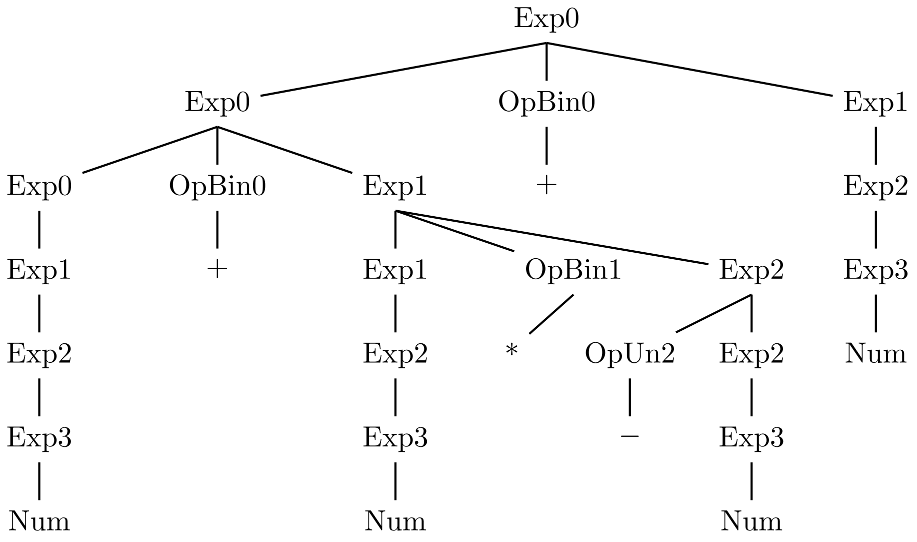
Num + (Num + Num)



El árbol de la izquierda sería equivalente al siguiente.



$Num - Num * -Num + Num$



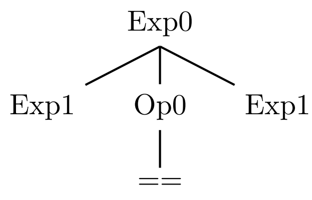
e.g. Lenguaje

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

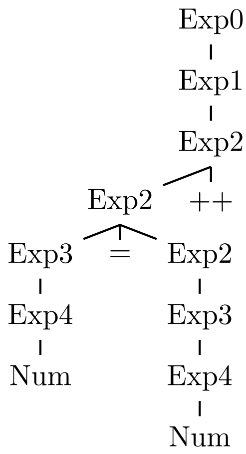
Operador	Prioridad	Aridad	Tipo	Asociatividad
==, !=	0	2		no
+	1	2		izq.
$*$	1	1	prefijo	no
=	2	2		der.
++	2	1	postfijo	si
$\sim$	3	1	prefijo	si
<<	3	1	postfijo	no

Exp0 → Exp1 Op0 Exp1
Exp0 → Exp1
Exp1 → Exp1 + Exp2
Exp1 → $*$ Exp2
Exp1 → Exp2
Exp2 → Exp3 = Exp2
Exp2 → Exp2 ++
Exp2 → Exp3
Exp3 → $\sim$ Exp3
Exp3 → Exp4 <<
Exp3 → Exp4
Exp4 → Num
Exp4 → Id
Exp4 → (Exp0)
Op0 → ==
Op0 → !=

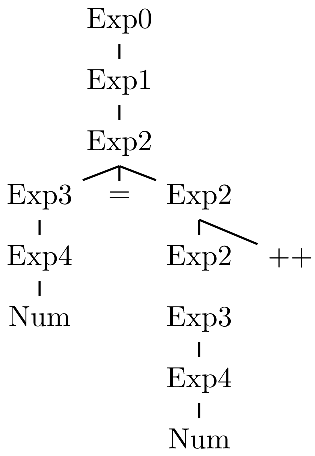
Num == Num == Num, está prohibido.



5 = 6 ++

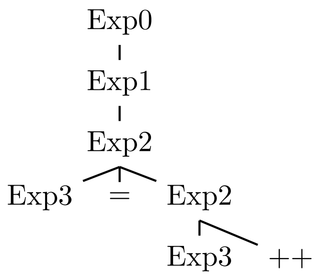


5 = 6 ++



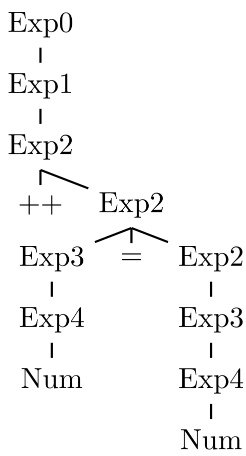
Supongamos que el operador ++ no asocia.
Ahora, Exp2 → Exp3 ++

5 = 6 ++



Supongamos que el operador ++ es prefijo y asocia.
Ahora, Exp2 → ++ Exp2

5 = 6 ++



- 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.

Exp0 $\rightarrow$ Exp0 $\rightarrow$ Exp1
Exp0 $\rightarrow$ Exp1 $\sim$ Exp1
Exp0 $\rightarrow$ # Exp1
Exp0 $\rightarrow$ Exp1
Exp1 $\rightarrow$ Exp1 @ Exp2
Exp1 $\rightarrow$ Exp2 $\wedge$ Exp2
Exp1 $\rightarrow$ Exp2
Exp2 $\rightarrow$ Exp2 $*$
Exp2 $\rightarrow$ Exp3
Exp3 $\rightarrow$ Num
Exp3 $\rightarrow$ (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 Instruncciones **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

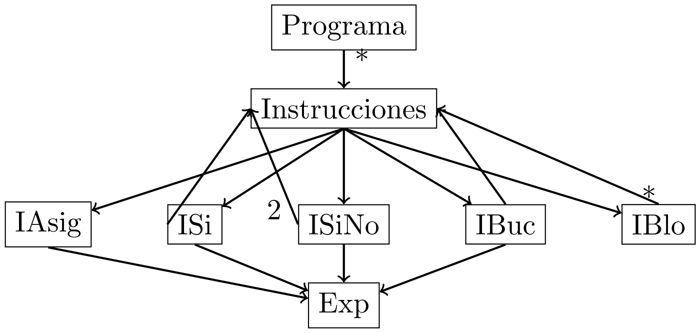
  Binarios, más prioridad que los aritméticos y asocian a derechas.

- Unarios. !, -

  Prefijos, más prioritarios que los lógicos y asocian.

Se pueden utilizar paréntesis, (...).

Un posible programa sería:
    x $\leftarrow$ 25
    WHILE x $<=$ 50 **DO**
        x $\leftarrow$ x + 1
    OD



Programa → LI
LI → I
LI → LI ; I
I → IAsig
I → ISi
I → ISino
I → IBuc
I → IBlo
IAsig → Var ⇐ Exp0
ISi → IF Exp0 I FI
ISino → IF Exp0 I ELSE I FI
IBuc → WHILE Exp0 DO I OD
IBlo → { S }
S → ε
S → LI

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

Exp0 → Exp1 Op0 Exp1
Exp0 → Exp1
Exp1 → Exp1 Op1 Exp2
Exp1 → Exp2
Exp2 → Exp3 Op2 Exp2
Exp2 → Exp3
Exp3 → Op3 Exp3
Exp3 → Exp4
Exp4 → (Exp0)
Exp4 → Num
Exp4 → Var

OpX se corresponde con los operadores del nivel X.

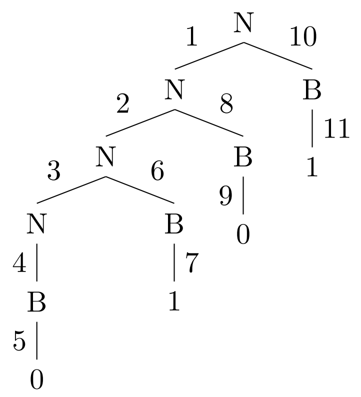
3.3 Implementación de anazlizadores descencentes

Vamos a ver analizadores descencentes (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

$N \to N\ B$
$N \to B$
$B \to 0$
$B \to 1$



$N \Rightarrow NB \Rightarrow NBB \Rightarrow NBBB \Rightarrow BBBB \Rightarrow 0BBB \Rightarrow 01BB \Rightarrow 010B \Rightarrow 0101$

<table>
<tr><th>Pila de análisis (←)</th><th>Entrada</th><th>Movimiento</th></tr>
<tr><td>N</td><td>|0101</td><td>Expandir N → N B</td></tr>
<tr><td>NB</td><td>|0101</td><td>Expandir N → N B</td></tr>
<tr><td>NBB</td><td>|0101</td><td>Expandir N → N B</td></tr>
<tr><td>NBBB</td><td>|0101</td><td>Expandir N → B</td></tr>
<tr><td>BBBB</td><td>|0101</td><td>Expandir B → 0</td></tr>
<tr><td>0BBB</td><td>|0101</td><td>Desplazar 0</td></tr>
<tr><td>BBB</td><td>0|101</td><td>Expandir B → 1</td></tr>
<tr><td>1BB</td><td>0|101</td><td>Desplazar 1</td></tr>
<tr><td>BB</td><td>01|01</td><td>Expandir B → 0</td></tr>
<tr><td>0B</td><td>01|01</td><td>Desplazar 0</td></tr>
<tr><td>B</td><td>010|1</td><td>Expandir B → 1</td></tr>
<tr><td>1</td><td>010|1</td><td>Desplazar 1</td></tr>
<tr><td></td><td>0101|</td><td>Aceptar</td></tr>
</table>

Gramática

- N' → N ⊢
- N → B R
- R → B R
- R → ε
- B → 0
- B → 1

Los no terminales se interpretan como funciones que reconocen partes de la entrada.

Las decisiones se basan en un token, el algoritmo siempre lleva un token adelantado.

Los terminales consumen el token si coincide y rechazan la entrada si no.

Para asegurarse de que la cadena pertenece y no solo un prefijo siempre se añade una producción adicional, N', que llama al simbolo inicial y luego reconoce el fin de fichero.

Pseudocódigo

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

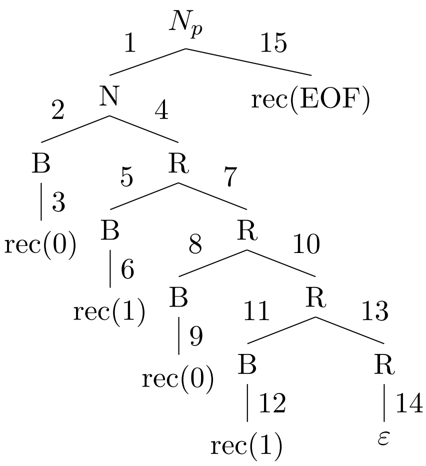
```
N() {
    B();
    R();
}
```

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

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

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

e.g. 0101⊢

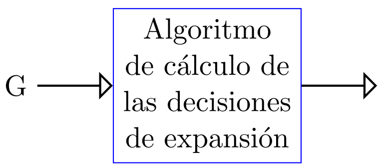


¿Funciona este método para cualquier gramática?

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

A $\rightarrow$ 0 X	L $\rightarrow$ L i
A $\rightarrow$ 0 Y	L $\rightarrow$ i

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



Para A
  si el sig. token es C_0... C_n aplica A $\rightarrow \alpha_0$
  ...
Para B
  ...

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, $\varepsilon \in$ PRIM(α) si $\alpha \Rightarrow^* \varepsilon$.
e.g.

N $\rightarrow$ B R
R $\rightarrow$ B R
R $\rightarrow \varepsilon$
B $\rightarrow$ 0
B $\rightarrow$ 1

PRIM(N) = {0, 1}
PRIM(B R) = {0, 1}

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

PRIM(α)-{ε} $\subseteq$ Dir(A $\rightarrow \alpha$)

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

$\Gamma_0 \oplus \Gamma_1 = (\Gamma_0 - \{\varepsilon\}) \cup \Gamma_1$ si $\varepsilon \in \Gamma_0$
$(\Gamma_0 - \{\varepsilon\}) \cup \emptyset$ c.o.c.

e.g.
{a, b} $\oplus$ {c, d} = {a, b}
{a, b, ε} $\oplus$ {c, d} = {a, b, c, d}
{a, b, ε} $\oplus$ {c, d, ε} = {a, b, c, d, ε}

Sea **a** un terminal y **A** un no terminal.
PRIM(a) = {a}

PRIM(ε) = {ε}
PRIM(A) = $\cup$ PRIM(α_i)   donde {α | A $\rightarrow$ α}

En general:

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

e.g.
PRIM(B R) = PRIM(B) $\oplus$ PRIM(R)

e.g.

N $\rightarrow$ B R
R $\rightarrow$ B R
R $\rightarrow$ ε
B $\rightarrow$ 0
B $\rightarrow$ 1

PRIM(N) = PRIM(B) $\oplus$ PRIM(R)
PRIM(R) = PRIM(B) $\oplus$ PRIM(R) $\cup$ PRIM(ε)
PRIM(B) = PRIM(0) $\cup$ PRIM(1)

Por tanto:
PRIM(B) = {0, 1}
PRIM(R) = {0, 1} $\cup$ {ε}
PRIM(N) = {0, 1}

Los primeros no resuelven R $\rightarrow$ ε porque el cuerpo de la regla se puede anular, necesitamos más terminales.

Introducimos el operador siguientes de A, SIG(A).
El operador se define como el conjunto de posibles terminales que siguen en las sentencias del lenguaje los fragmentos derivados de A.

$\vdash$ $\in$ SIG(S), donde S es el símbolo inicial.
Dada la producción A $\rightarrow$ αBβ sabemos que:

- PRIM(β) - {ε} $\subseteq$ SIG(B)
- Si $\varepsilon \in$ PRIM(β) $\Rightarrow$ SIG(A) $\subseteq$ SIG(B)

e.g.

Obj $\rightarrow$ Desig RParams
RParams $\rightarrow$ (LParams)
RParams $\rightarrow$ ε

SIG(Desig) = PRIM(RParams) - {ε} $\cup$ SIG(Obj)
SIG(Obj) = {;}
PRIM(RParams) = {(, ε}
SIG(Desig) = {(, ;}

e.g.

N → B R
R → B R
R → ε
B → 0
B → 1

$$
\begin{aligned}
\text{SIG(N)} &= \{\vdash\} \\
\text{SIG(R)} &= \text{PRIM}(\varepsilon) \oplus \text{SIG(N)} \cup \text{PRIM}(\varepsilon) \oplus \text{SIG(R)} \\
&= \{\varepsilon\} \oplus \text{SIG(N)} \cup \{\varepsilon\} \oplus \text{SIG(R)} \\
&= \text{SIG(N)} \cup \text{SIG(R)} \\
&= \{\vdash\} \cup \text{SIG(R)} \\
&= \{\vdash\} \\
\text{SIG(B)} &= \text{PRIM(R)} \oplus \text{SIG(N)} \cup \text{PRIM(R)} \oplus \text{SIG(R)} \\
&= \{0,\ 1,\ \varepsilon\} \oplus \text{SIG(N)} \cup \{0,\ 1,\ \varepsilon\} \oplus \text{SIG(R)} \\
&= \{0,\ 1,\ \vdash\} \cup \{0,\ 1,\ \vdash\}
\end{aligned}
$$

Revisemos ahora los el conjunto de directores, teniamos que:

DIR(A → α) = PRIM(α) - $\{\varepsilon\}$

Pero ahora al anularse podemos definir qué ocurre, la formula quedaria:

DIR(A → α) = PRIM(α) $\oplus$ SIG(A)

e.g.

N → B R
R → B R
R → ε
B → 0
B → 1

$$
\begin{aligned}
\text{DIR(N} \to \text{B R)} &= \text{PRIM(B R)} \oplus \text{SIG(N)} \\
&= \text{PRIM(B)} \oplus \text{PRIM(R)} \oplus \text{SIG(N)} \\
&= \{0,\ 1\} \\
\text{DIR(R} \to \text{B R)} &= \{0,\ 1\} \\
\text{DIR(R} \to \varepsilon) &= \text{PRIM}(\varepsilon) \oplus \text{SIG(R)} \\
&= \{\varepsilon\} \oplus \text{SIG(R)} \\
&= \{\vdash\} \\
\text{DIR(B} \to 0) &= \{0\} \\
\text{DIR(B} \to 1) &= \{1\}
\end{aligned}
$$

Sea G una gramática LL1 de la forma
A → α
A → β
siempre se cumple que DIR(A → α) ∩ DIR(A → β) = ∅

Revisemos ahora el pseudocódigo de R.

```
R() {
    if sigToken ∈ {0,1} then
        B();
        R();
    elif sigToken ∈ {⊢} then
        //TODO
    else
        error({0, 1, ⊢})
    fi
}
```

```
error(Γ) {
    print Token inesperado sigToken, se esperaba Γ.
}
```

e.g.

$L \to [\ E_s\ ]$
$E_s \to LI_s$
$E_s \to \varepsilon$
$LI_s \to \mathrm{I}\ RLI_s$
$RLI_s \to\ ,\ \mathrm{I}\ RLI_s$
$RLI_s \to \varepsilon$
$\mathrm{I} \to \mathrm{a}$
$\mathrm{I} \to \mathrm{L}$

[a, a, [ ], [a, a]]

$$
\begin{aligned}
\mathrm{PRIM}(\mathrm{L}) &= \mathrm{PRIM}([) \oplus \mathrm{PRIM}(E_s) \oplus \mathrm{PRIM}(]) \\
&= \{[\} \\
\mathrm{PRIM}(E_s) &= \mathrm{PRIM}(LI_s) \cup \mathrm{PRIM}(\varepsilon) \\
&= \{\mathrm{a},\ [,\ \varepsilon\} \\
\mathrm{PRIM}(LI_s) &= \mathrm{PRIM}(\mathrm{I}) \oplus \mathrm{PRIM}(RLI_s) \\
&= \{\mathrm{a},\ [\} \\
\mathrm{PRIM}(RLI_s) &= \mathrm{PRIM}(,) \oplus \mathrm{PRIM}(\mathrm{I}) \oplus \mathrm{PRIM}(RLI_s) \cup \mathrm{PRIM}(\varepsilon) \\
&= \{,\ ,\ \varepsilon\} \\
\mathrm{PRIM}(\mathrm{I}) &= \mathrm{PRIM}(\mathrm{a}) \cup \mathrm{PRIM}(\mathrm{L}) \\
&= \{\mathrm{a},\ [\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{SIG}(\mathrm{L}) &= \{\vdash\} \cup \mathrm{PRIM}(\varepsilon) \oplus \mathrm{SIG}(\mathrm{I}) \\
&= \{\vdash,\ ,\ ,[\} \\
\mathrm{SIG}(E_s) &= \mathrm{PRIM}(]) \oplus \mathrm{SIG}(\mathrm{L}) \\
&= \{]\} \\
\mathrm{SIG}(LI_s) &= \mathrm{PRIM}(\varepsilon) \oplus \mathrm{SIG}(E_s) \\
&= \{]\} \\
\mathrm{SIG}(RLI_s) &= \mathrm{PRIM}(\varepsilon) \oplus \mathrm{SIG}(LI_s) \cup \mathrm{PRIM}(\varepsilon) \oplus \mathrm{SIG}(RLI_s) \\
&= \{]\} \\
\mathrm{SIG}(\mathrm{I}) &= \mathrm{PRIM}(RLI_s) \oplus \mathrm{SIG}(LI_s) \cup \mathrm{PRIM}(RLI_s) \oplus \mathrm{SIG}(RLI_s) \\
&= \{,,\ ]\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{DIR}(\mathrm{L} \to [\ E_s\ ]) &= \mathrm{PRIM}([\ E_s\ ]) \oplus \mathrm{SIG}(\mathrm{L}) \\
&= \mathrm{PRIM}([) \oplus \mathrm{PRIM}(E_s) \oplus \mathrm{PRIM}(]) \oplus \mathrm{SIG}(\mathrm{L}) \\
&= \{[\} \\
\mathrm{DIR}(E_s \to LI_s) &= \{\mathrm{a},\ [\} \\
\mathrm{DIR}(E_s \to \varepsilon) &= \{]\} \\
\mathrm{DIR}(LI_s \to \mathrm{I}\ RLI_s) &= \{\mathrm{a},\ [\} \\
\mathrm{DIR}(RLI_s \to\ ,\ \mathrm{I}\ RLI_s) &= \{,\} \\
\mathrm{DIR}(RLI_s \to \varepsilon) &= \{]\} \\
\mathrm{DIR}(\mathrm{I} \to \mathrm{a}) &= \{\mathrm{a}\} \\
\mathrm{DIR}(\mathrm{I} \to \mathrm{L}) &= \{[\}
\end{aligned}
$$

3.3.1 Resolución por aproximaciones sucesivas

La resolución por sustitución nos ha servido hasta ahora pero es un problema ante gramáticas con múlitples equiaciones del tipo:

$$\overrightarrow{x} = \Phi\ [\overrightarrow{x}],\ \overrightarrow{x}\ in\ \mathrm{U}\ \text{con U finito.}$$
$$\overrightarrow{x}\ \text{es un punto fijo de}\ \Phi$$

Planteamos ahora como método la resolución por aproximaciones sucesivas.

$$\overrightarrow{x}_0\colon\ \overrightarrow{x}_{0i} = \emptyset, \text{ para cada i.}$$
$$\overrightarrow{x}_{n+1} = \Phi(\overrightarrow{X}_n)$$

Si $\overrightarrow{x}_n$ tal que $\Phi(\overrightarrow{X}_n) = \overrightarrow{x}_n$, entonces $\overrightarrow{x}_n$ satisface $\overrightarrow{x} = \Phi[\overrightarrow{X}]$.
$\overrightarrow{x}_n$ es el mínimo punto fijo.

e.g.

L → [I_s]
$I_s \to LI_s$
$I_s \to \varepsilon$
$LI_s \to$ I RLI_s
$RLI_s \to$, I RLI_s
$RLI_s \to \varepsilon$
I → a
I → L

PRIM(L) = PRIM([) ⊕ PRIM(I_s) ⊕ PRIM(])
PRIM(I_s) = PRIM(LI_s) ∪ PRIM(ε)
PRIM(LI_s) = PRIM(I) ⊕ PRIM(RLI_s)
PRIM(RLI_s) = PRIM(,) ⊕ PRIM(]) ⊕ PRIM(RLI_s) ∪ PRIM(ε)
PRIM(I) = PRIM(a) ∪ PRIM(L)

Iteración	0	1	2	3	4	5
PRIM(L)	∅	[	[	[	[	[
PRIM(I_s)	∅	ε	ε	a ε	a [ε	a [ε
PRIM(LI_s)	∅	∅	a	a [	a [	a [
PRIM(RLI_s)	∅	, ε	, ε	, ε	, ε	, ε
PRIM(I)	∅	a	a [	a [	a [	a [

Como en la iteración 5 ya no se han producido cambios, se ha estabilizado, sabemos que hemos alcanzado el mínimo punto fijo.

Los siguientes se pueden calcular con el mismo tipo de algoritmo.

3.3.2 Transformación de gramáticas

Con los métodos que conocemos tenemos problemas para implementar gramáticas como
$E_0 \to E_0 + E_1$
$E_0 \to E_1$
porque
PRIM(E_1) ⊆ PRIM(E_0), por la regla (2)
PRIM(E_0) − {ε} ⊆ DIR($E_0 \to E_0 + E_1$)
PRIM(E_1) − {ε} ⊆ DIR($E_0 \to E_1$)
PRIM(E_1) − {ε} ⊆ DIR($E_0 \to E_0 + E_1$)

Por las últimas dos conclusiones vemos un problema, los directores no son disjuntos, que nos impide decidir.

Sea G una gramática de la forma
A → $\alpha\ \beta_0$
...
A → $\alpha\ \beta_n$

podemos crear G', la gramática transformada, sacando el factor común

A $\rightarrow \alpha$ RA
RA $\rightarrow \beta_0$
...
RA $\rightarrow \beta_n$

e.g.

$E_0 \rightarrow E_1 + E_0$
$E_0 \rightarrow E_1$

$E_0 \rightarrow E_1\ R_0$
$R_0 \rightarrow +\ E_0$
$R_0 \rightarrow \varepsilon$

Veamos ahora como tratar la recursón. Sea G una gramática de la forma

A $\rightarrow$ A α_0
...
A $\rightarrow$ A α_n
A $\rightarrow \beta_0$
...
A $\rightarrow \beta_n$

podemos crear G' cambiando el tipo de recursión, de izquierdas a derechas.

A $\rightarrow \beta_0\ R_A$
...
A $\rightarrow \beta_n\ R_A$
$R_A \rightarrow \alpha_0\ R_A$
...
$R_A \rightarrow \alpha_n\ R_A$
$R_A \rightarrow \varepsilon$

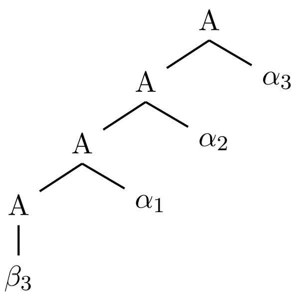


Figure 3.3.2.1 : Árbol de G.

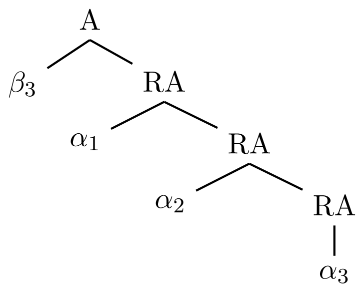


Figure 3.3.2.2 : Árbol de G'.

e.g.

$E_0 \rightarrow E_0 + E_1$
$E_0 \rightarrow E_1$
$E_1 \rightarrow E_2 * E_1$
$E_1 \rightarrow E_2$
$E_2 \rightarrow$ num
$E_2 \rightarrow (\ E_0\ )$

$E_0 \rightarrow E_1\ R_0$
$R_0 \rightarrow +\ E_1\ R_0$
$R_0 \rightarrow \varepsilon$
$E_1 \rightarrow E_2\ R_1$
$R_1 \rightarrow *\ E_1$
$R_1 \rightarrow \varepsilon$
$E_2 \rightarrow$ num
$E_2 \rightarrow (\ E_0\ )$

Recursión a izquierdas
Factor común

Por tanto la especificación se haria utilizando G pero implementamos G'.

Calculamos los primeros de G'
$\mathrm{PRIM}(E_0) = \mathrm{PRIM}(E_1) \oplus \mathrm{PRIM}(R_0)$
$\mathrm{PRIM}(R_0) = \mathrm{PRIM}(+) \oplus \mathrm{PRIM}(E_1) \oplus \mathrm{PRIM}(R_0) \cup \mathrm{PRIM}(\varepsilon)$
$\mathrm{PRIM}(E_1) = \mathrm{PRIM}(E_2) \oplus \mathrm{PRIM}(R_1)$
$\mathrm{PRIM}(R_1) = \mathrm{PRIM}(*) \oplus \mathrm{PRIM}(E_1) \cup \mathrm{PRIM}(\varepsilon)$
$\mathrm{PRIM}(E_2) = \mathrm{PRIM}(\mathrm{num}) \cup \mathrm{PRIM}(() \oplus \mathrm{PRIM}(E_0) \oplus \mathrm{PRIM}())$

	0	1	2	3	4
$\mathrm{PRIM}(E_0)$	Ø	Ø	Ø	num (	num (
$\mathrm{PRIM}(R_0)$	Ø	$+\ \varepsilon$	$+\ \varepsilon$	$+\ \varepsilon$	$+\ \varepsilon$
$\mathrm{PRIM}(E_1)$	Ø	Ø	num (	num (	num (
$\mathrm{PRIM}(R_1)$	Ø	$*\ \varepsilon$	$*\ \varepsilon$	$*\ \varepsilon$	$*\ \varepsilon$
$\mathrm{PRIM}(E_2)$	Ø	num (	num (	num (	num (

Ahora los siguientes de G'
$\mathrm{SIG}(E_0) = \{\vdash\} \cup \mathrm{PRIM}()) \oplus \mathrm{SIG}(E_2)$
$\mathrm{SIG}(R_0) = \mathrm{PRIM}(\varepsilon) \oplus \mathrm{SIG}(E_0) \cup \mathrm{PRIM}(\varepsilon) \oplus \mathrm{SIG}(R_0)$
$\mathrm{SIG}(E_1) = \mathrm{PRIM}(R_0) \oplus \mathrm{SIG}(E_0) \cup \mathrm{PRIM}(R_0) \oplus \mathrm{SIG}(R_0)$
$\mathrm{SIG}(R_1) = \mathrm{PRIM}(\varepsilon) \oplus \mathrm{SIG}(E_1)$
$\mathrm{SIG}(E_2) = \mathrm{PRIM}(R_1) \oplus \mathrm{SIG}(E_1)$

	0	1	2	3	4
$\mathrm{SIG}(E_0)$	Ø	$\vdash$)	$\vdash$)	$\vdash$)	$\vdash$)
$\mathrm{SIG}(R_0)$	Ø	Ø	$\vdash$)	$\vdash$)	$\vdash$)
$\mathrm{SIG}(E_1)$	Ø	+	+ $\vdash$)	+ $\vdash$)	+ $\vdash$)
$\mathrm{SIG}(R_1)$	Ø	Ø	+	+ $\vdash$)	+ $\vdash$)
$\mathrm{SIG}(E_2)$	Ø	$*$	$*$ +	$*$ + $\vdash$)	$*$ + $\vdash$)

Por último empleamos los siguientes y los primeros para calcular los directores.
$\mathrm{DIR}(E_0 \to E_1\ R_0) = \{\mathrm{num}, (\}$
$\mathrm{DIR}(R_0 \to +\ E_1\ R_0) = \{+\}$
$\mathrm{DIR}(R_0 \to \varepsilon) = \{\vdash, (\}$
$\mathrm{DIR}(E_1 \to E_2\ R_1) = \{\mathrm{num}, (\}$
$\mathrm{DIR}(R_1 \to *\ E_1) = \{*\}$
$\mathrm{DIR}(R_1 \to \varepsilon) = \{+, \vdash,)\}$
$\mathrm{DIR}(E_2 \to \mathrm{num}) = \{\mathrm{num}\}$
$\mathrm{DIR}(E_2 \to (\ E_0\ )) = \{(\}$

Los directores para las gramáticas con múltiples producciones son disjuntos, por lo que no vamos a tener problemas de decisión.

3.3.3 Tabla de análisis sintáctico descendente

Sea G una gramática de la forma

$N \to B\ RN$	$DIR(N \to B\ RN) = \{0, 1\}$
$RN \to B\ RN$	$DIR(RN \to B\ RN) = \{0, 1\}$
$RN \to \varepsilon$	$DIR(RN \to \varepsilon) = \{\vdash\}$
$B \to 0$	$DIR(B \to 0) = \{0\}$
$B \to 1$	$DIR(B \to 1) = \{1\}$

La tabla de análisis de G sería la siguiente.

	0	1	$\vdash$
N	$N \to B\ RN$	$N \to B\ RN$	
RN	$RN \to B\ RN$	$RN \to B\ RN$	$RN \to \varepsilon$
B	$B \to 0$	$B \to 1$	

Cada celda indica qué producción aplicar al detectar cierto símbolo, si una celda está vacía daría error.

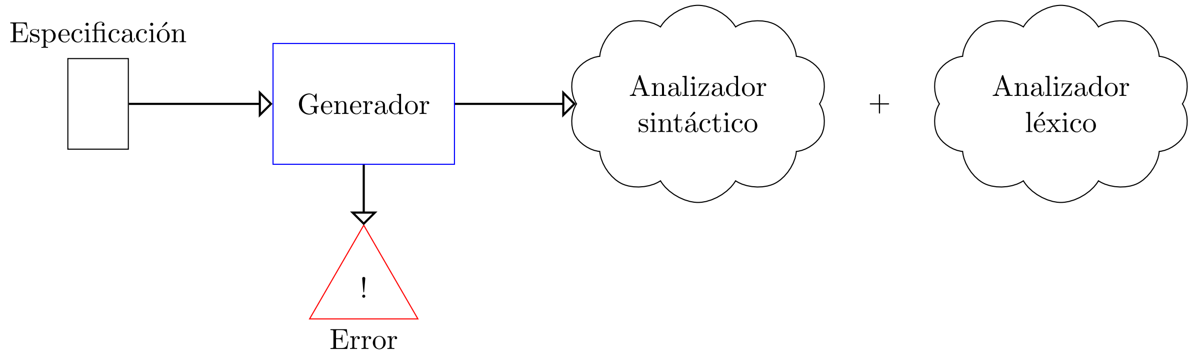
La gramática es LL(1) si en cada celda hay a lo sumo una producción.

Se puede implementar un programa genérico que utilice la tabla para implementar el analizador, se llama analizador sintáctico descendente predictivo no-recursivo, o dirigido por tabla.

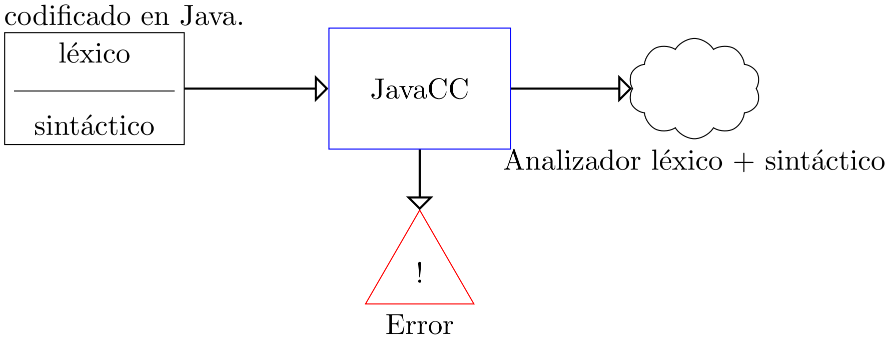
e.g. Queremos reconocer $010\vdash$, utilizando la tabla de análisis de G.

Pila ($\leftarrow$)	Entrada	Acción
N	$010\vdash$	$N[0] \equiv N \to B\ RN$
B RN	$010\vdash$	$B[0] \equiv B \to 0$
0 RN	$010\vdash$	Consume 0
RN	$10\vdash$	$RN[1] \equiv RN \to B\ RN$
B RN	$10\vdash$	$B[1] \equiv B \to 1$
1 RN	$10\vdash$	Consume 1
RN	$0\vdash$	$RN[0] \equiv RN \to B\ RN$
B RN	$0\vdash$	$B[0] \equiv B \to 0$
0 RN	$0\vdash$	Consume 0
RN	$\vdash$	$RN[\vdash] \equiv RN \to \varepsilon$
	$\vdash$	Pila vacía $\to$ aceptar

3.3.4 Implementación mediante herramientas



En nuestro caso el generador será JavaCC que dará lugar al analizador sintáctico codificado en Java.



Soporta gramáticasen formato EBNF (Extended BNF) aunque no lo vamos a usar. BNF (Backus Normal Form) es el formato de reglas gramaticales que hemos empleado hasta ahora.

e.g. BNF

$E_0 \rightarrow E_1$ ((\+ | \-) RE_0)$*$

Para compilar usar: java -cp javacc.jar javacc input.jj
Generara varios ficheros, *TokenManager.java es el analizador.

Esta herramiento soporta gramáticas LL(k), pueden tener k símbolos adelantados. En el caso de nuestras gramáticas la k es 1.

LL($*$) son gramáticas que pueden utilizar cadenas arbitrariamente largas, la herramienta usada para esta tipología es ANTLR.

3.4 Análisis ascendente

Sea G
N → N B
N → B
B → 0
B → 1

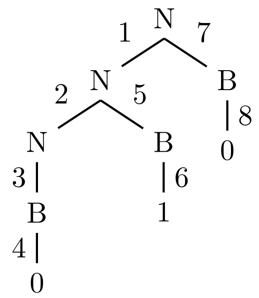


Figure 3.4.1 : Descendente

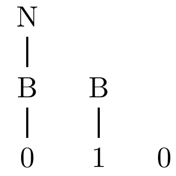
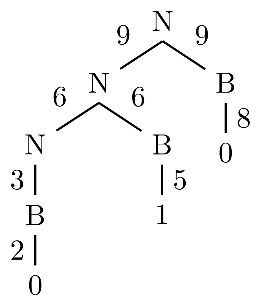


Figure 3.4.2 : Ascendente

El modelo que vamos a seguir es el analizador por desplazamiento-reducción.

N	Pila (←)	Entrada	Acción
0		010⊢	
1	0	10⊢	desplazamiento
2	B	10⊢	reducción B → 0
3	N	10⊢	reducción N → B
4	N 1	0⊢	desplazamiento
5	N B	0⊢	reducción B → 1
6	N	10⊢	reducción N → N B
7	N 0	⊢	desplazamiento
8	N B	⊢	reducción B → 0
9	N	⊢	reducción N → N B

La entrada se ha reducido al símbolo inicial por tanto aceptamos la cadena. El siguiente árbol muestra el orden de expansión, los numeros indican el N de la tabla que lo ha generado.



Veamos ahora una derivación más a la derecha de la sentencia.
N ⇒ NB ⇒ N0 ⇒ NB0 ⇒ N10 ⇒ B10 ⇒ 010
La secuencia, leida de derecha a izquierda, se corresponde con la construcción ascendente del árbol.

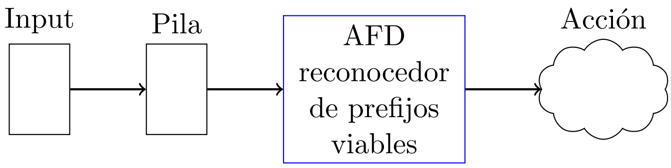
Los posibles contenidos de la pila se llaman prefijo viable.

- Asidero: sufijo del prefijo viable que se sustituye en una reducción.

e.g. n + n + n
Los asideros serán los marcados de color rojo.

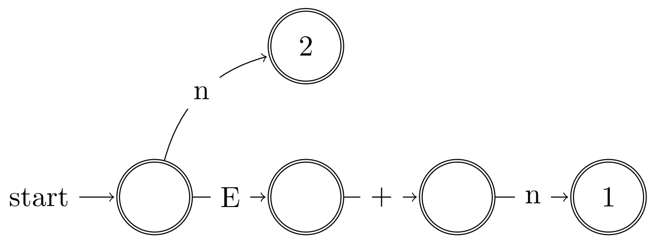
	PILA
	n
	E
E → E + n	E +
E → n	E + n
	E
	E +
	E + n
	E

El conjunto de prefijos viables para una gramática G es un conjunto regular, es decir, puede ser reconocido por una AFD.



e.g. AFD

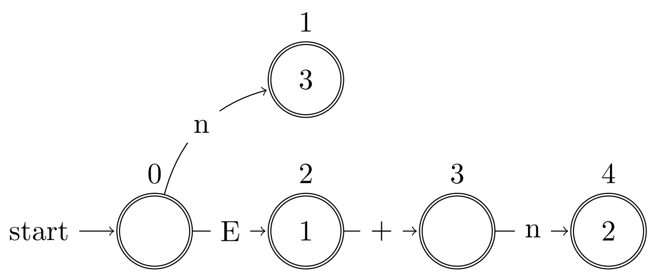
(1) E → E + n
(2) E → n



El número en un estado indica cuál de las reglas reducirá, los estados sin número representan la acción de desplazar.

e.g. AFD

(1) E' → E ⊢
(2) E → E + n
(3) E → n



	⊢	n	+	E
0		d 1		2
1	r E → E + n	r E → E + n	r E → E + n	
2	a		d 3	
3		d 4		
4	r E → E + n	r E → E + n	r E → E + n	

Solo los terminales dictan acciones, por eso la columna de la E solo tiene transiciones. Las casillas vacías en la tabla indican los errores sintácticos.

Pila	Entrada	Estado AFD	Acción
	n + n + n ⊢	0	desplazar
n	+ n + n ⊢	1	reducir E → n
E	+ n + n ⊢	2	desplazar
E +	n + n ⊢	3	desplazar
E + n	+ n ⊢	4	reducir E → E + n
E	+ n ⊢	2	desplazar
E +	n ⊢	3	desplazar
E + n	⊢	4	reducir E → E + n
E	⊢	2	aceptar

Estos autómatas pueden optimizarse, ahora la pila tiene siempre en la cima el estado en el que está. Para que el cambio se aprecie mejor los estados se marcaran en rojo.

Pila	Entrada	Estado AFD	Acción
0	n + n + n ⊢	0	desplazar
0 n 1	+ n + n ⊢	1	reducir E → n
0 E 2	+ n + n ⊢	2	desplazar
0 E 2 + 3	n + n ⊢	3	desplazar
0 E 2 + 3 n 4	+ n ⊢	4	reducir E → E + n
0 E 2	+ n ⊢	2	desplazar
0 E 2 + 3	n ⊢	3	desplazar
0 E 2 + 3 n 4	⊢	4	reducir E → E + n
0 E 2	⊢	2	aceptar

Para que siga funcionando las reducciones deberan desapilar varios elementos.

0 n 1 → 0 → 0 E 2

3.4.1 Métodos de construcción de AFDs

Concretamente queremos implementar AFDs que reconozcan prefijos viables.

En las tablas puede haber más de una acción por celda, se denomina conflicto, esto provocará que el análisis no sea determinista y el método no funcionará.

Los métodos son los siguientes, de menos a más genéricos.

- LR(0) $\rightarrow$ gramáticas LR(0)
- SLR(1) $\rightarrow$ gramáticas SLR(1)
- LALR(1) $\rightarrow$ gramáticas LALR(1)
- LR(1) $\rightarrow$ gramáticas LR(1)

De todos ellos el más empleado es LALR(1).

Existen dos tipos de conflictos.

- Desplazamiento-reducción.
  Si en una celda nos encontramos como acciones posibles:
  d X
  r A $\rightarrow \alpha$
- Reducción-reducción.
  Si en una celda nos encontramos como acciones posibles:
  r A $\rightarrow \alpha$
  r B $\rightarrow \beta$

Gramática LR(0), es posible determinar que acción realizar únicamente a partir del prefijo viable.

A $\rightarrow$ a
A $\rightarrow$ b
Prefijos viables $\equiv \{\varepsilon$, A, a, b$\}$

Gramática LR(1), emplea el prefijo viable y un símbolo anticipado de la entrada.

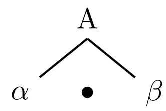
A $\rightarrow$ a
A $\rightarrow$ a b
Es LR(1) pero no LR(0).

3.4.1.1 Método LR(0)

A $\rightarrow \alpha \bullet \beta$, donde α y β se denominarán elementos LR(0).
En el contexto de una estructura A, como la descrita previamente.

- Hemos "visto" α

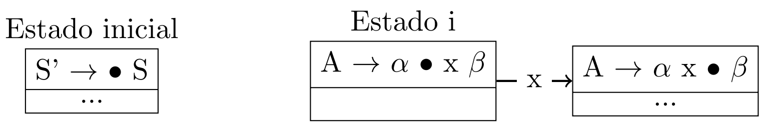
- Es posible ver β y, potencialmente, reducir por A $\rightarrow$ α β



Los estados del AFD LR(0) son conjuntos de elementos LR(0), cada conjunto será válido para cierto prefijo viable.

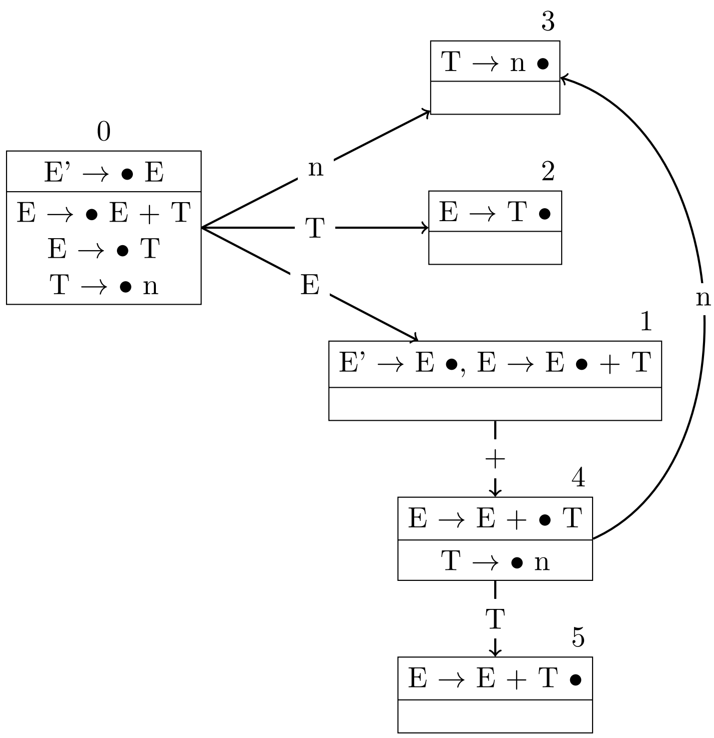
Para contruirlo.

- Partimos de una serie de elementos iniciales, denominados elementos nucleares.
  $A_0 \rightarrow \alpha_0\ \beta_0\ ...\ A_n \rightarrow \alpha_n\ \beta_n$
- Siguiendo la regla de cierre habrá otros elementos, denominados elementos no nucleares.
  A $\rightarrow$ α • B β $\Rightarrow$ B $\rightarrow$ • γ



... serán las posibles producciones para continuar el análisis

e.g.



E' $\rightarrow$ E
E $\rightarrow$ E + T
E $\rightarrow$ T
T $\rightarrow$ n

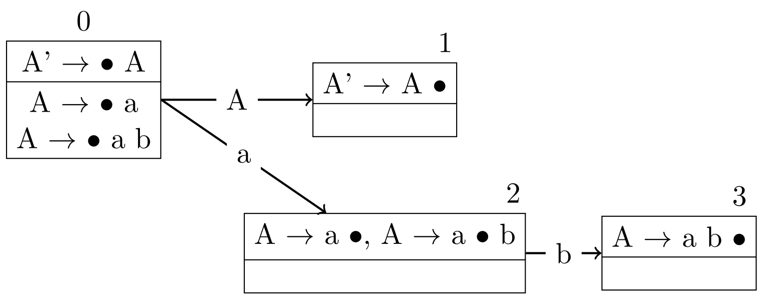
Del automata anterior podemos obtener la siguiente tabla.

	n	⊢	+	E	T
0	d 3			1	2
1		d 4	a		
2	r E → T	r E → T	r E → T		
3	r T → n	r T → n	r T → n		
4	d 3				5
5	r E → E + T	r E → E + T	r E → E + T		

Una gramática LR(0) es aquella para la cual el método LR(0) genera tablas sin conflictos. Entendemos conflicto como una celda no definida de manera unívoca.

e.g. Conflicto desplazamiento-reducción

A' → A
A → a
A → a b



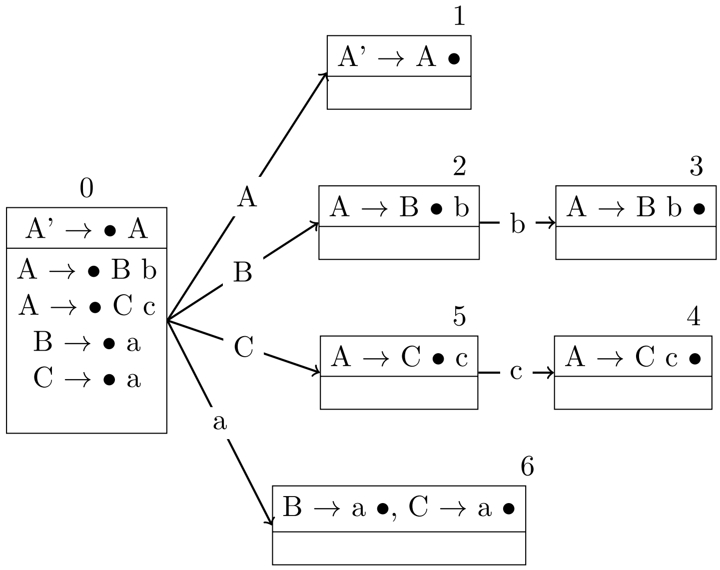
El problema se da en el estado 2 porque tiene dos posibles acciones, conflicto de desplazamiento-reducción.

	a	b	⊢	A
0	d 2			1
1			a	
2	r A → a	r A → a d 3	r A → a	
3	r A → a b	r A → a b	r A → a b	

La gramática no es LR(0).

e.g. Conflicto reducción-reducción

A’ → A
A → B b
A → C c
B → a
C → a



	a	b	c	⊢	A	B	C
0	d 6				1	2	5
1				a			
2		d 3					
3	r A → B b	r A → B b	r A → B b	r A → B b			
4	r A → C c	r A → C c	r A → C c	r A → C c			
5			d 4				
6	r B → A r C → a	r B → A r C → a	r B → A r C → a	r B → A r C → a			

3.4.1.2 Método SLR(1)

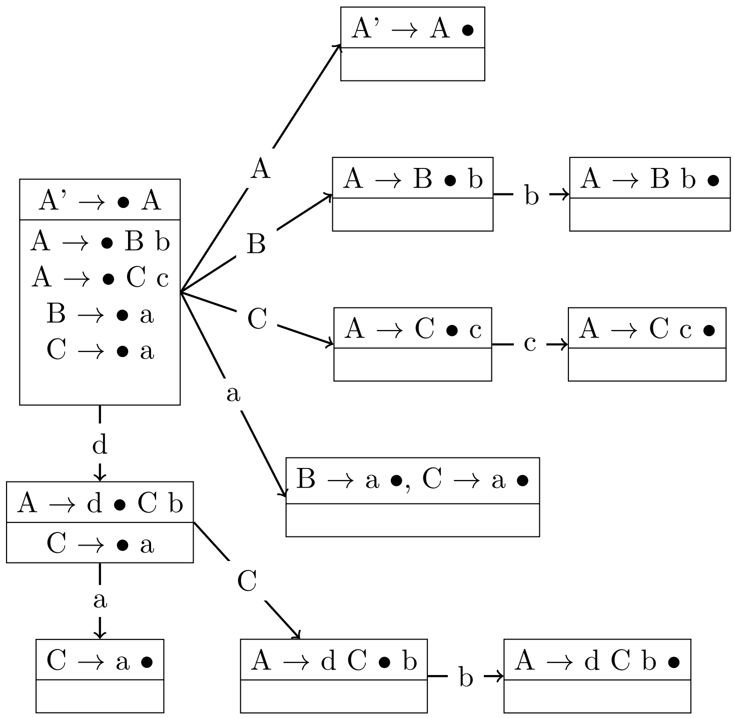
Consiste en utilizar los siguientes en prefijos viables para no realizar reducciones inútiles. Al aplicarlo la tabla anterior quedaria de la siguiente manera.

	a	b	c	⊢	A	B	C
0	d 6				1	2	5
1				a			
2		d 3					
3				r A → B b			
4				r A → C c			
5			d 4				
6		r B → A	r C → a				

No hay conflictos por lo que la gramática es SLR(1).

e.g. La siguiente gramática supone un problema incluso para SLR(1), no se suelen encontrar casos de este tipo.

A' → A
A → d C b
A → B b
A → C c
B → a
C → a

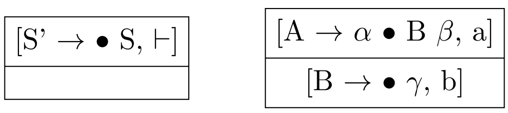


Como hay un conflicto de reducción-reducción la gramática no es SLR(1).

3.4.1.3 Método LR(1)

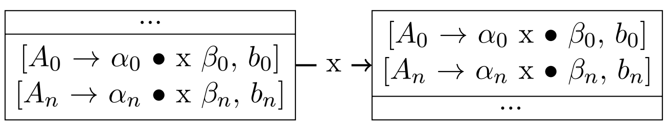
En este método también se indica el símbolo por el cual se aplica la reducción.

$$[\mathrm{A} \to \alpha \bullet \beta,\ \mathrm{a}]$$



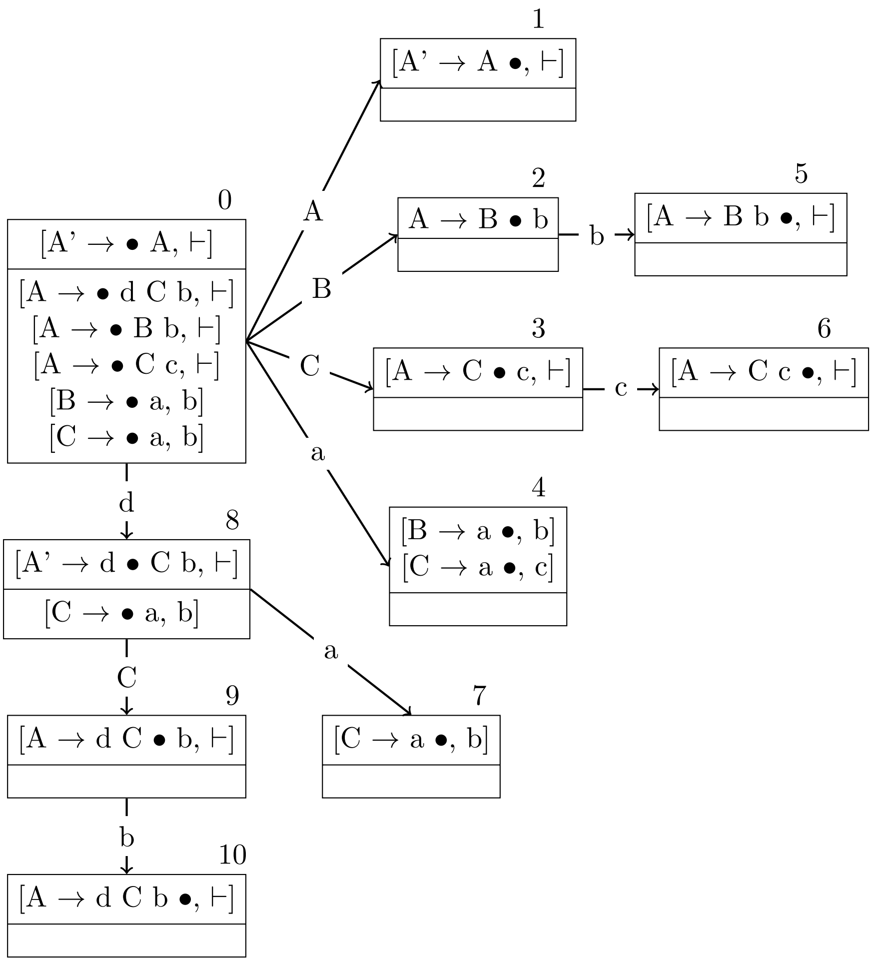
Se expandiran producciones en un nodo para toda producción de la forma B → γ y para todo b ∈ PRIM(B) ⊕ {a}.

e.g.



e.g.

A’ → A
A → d C b
A → B b
A → C c
B → a
C → a



	a	b	c	d	⊢	A	B	C
0	d 4			d 8		1	2	3
1					a			
2		d 5						
3			d 6					
4		r B → a	r C → a					
5					r A → B b			
6					r A → C c			
7		r C → a						
8	d 7							9
9		d 10						
10					r A → d C B			

Como la tabla no tiene conflictos podemos concluir que esta gramática es LR(1).

En el ejemplo no ha ocurrido pero el número de estados podría verse incrementado

[C $\rightarrow$ a $\bullet$, b] [C $\rightarrow$ a $\bullet$, $\vdash$]

Como puede verse ambos estados coincidirian si utilizaramos el método LR(0).

3.4.1.4 Método LALR(1)

Para mejorar el método anterior podemos fusionar los estados.

[C $\rightarrow$ a $\bullet$, b]
[C $\rightarrow$ a $\bullet$, $\vdash$]

Como lo único que se ha realizado ha sido dicha fusión podemos asegurar que toda gramática LALR(1) es necesariamente LR(1).

Sin embargo, al realizar dicha fusión pueden aparecer conflictos por lo que no toda gramática LR(1) tiene que ser LALR(1).

- Conflicto desplazamiento-reducción.

  [A $\rightarrow \alpha \bullet$ a β, -]
  [B $\rightarrow \gamma \bullet$, -]

  [A $\rightarrow \alpha \bullet$ a β, -]
  [B $\rightarrow \gamma \bullet$, a]

  [A $\rightarrow \alpha \bullet$ a β, -]
  [A $\rightarrow \alpha \bullet$ a β, -]
  [B $\rightarrow \gamma \bullet$, a]
  [B $\rightarrow \gamma \bullet$, -]

  No pueden aparecer conflictos NUEVOS de este tipo, pero si los habia en LR(1) seguiran estando.

- Conflicto reducción-reducción.

  Pueden aparecer conflictos nuevos de este tipo.

  [A $\rightarrow \alpha \bullet \beta$, a]
  [A $\rightarrow \alpha \bullet \beta$, b]
  [A $\rightarrow \alpha \bullet \beta$, c]
  Lo podemos escribir de la siguiente forma.
  [A $\rightarrow \alpha \bullet \beta$, {a, b, c}]

$[A \rightarrow \alpha \bullet, \psi_0]$ $[B \rightarrow \beta \bullet, \psi_1]$

$\psi_0 \cap \psi_1 = \emptyset$
$\psi_0' \cap \psi_1' = \emptyset$
$(\psi_0 \cup \psi_0') \cap (\psi_1 \cup \psi_1') \neq \emptyset$

$[A \rightarrow \alpha \bullet, \psi_0']$ $[B \rightarrow \beta \bullet, \psi_1']$

Por tanto al fusionar pueden aparecer conflictos de este tipo.

e.g.

B → A b
C → B b
C → A c
C → B b
A → a
B → a

[A → a •, b] [B → a •, c]

[A → a •, c] [B → a •, b]

Podemos fusionar ambos estados dando lugar al siguiente.

[A → a •, {b, c}] [B → a •, {b, c}]

Al fusionarlos se han generado conflictos, probando así que no toda gramática LR(1) es LALR(1).

TODO: Pasar automata ejemplo grande.

Construcción directa del autómata LALR(1)

$\#_{n,m}$ ≡ preanálisis del elemento m del estado n.

L’ → L
L → [E_s]
E_s → LE_s
E_s → ε
LE_s → LE_s , E
LE_s → E
E → a
E → L

0

0 [L’ → • L, ⊢]
1 [L → • [E_s], $\#_{0,0}$]

1

0 [L’ → L •, $\#_{0,0}$]

2

0 [L → [• E_s], {$\#_{0,0}$, $\#_{2,7}$, $\#_{9,3}$}]
1 [E_s → • LE_s, {]}] 2 [E_s → •, {]}] 3 [LE_s → • LE_s , E, {’,’, $\#_{2,1}$}] 4 [LE_s → • E, {’,’, $\#_{2,1}$}] 5 [E → • a, {$\#_{2,4}$}] 6 [E → • L, {$\#_{2,4}$}] 7 [L → • [E_s], {$\#_{2,6}$}]

3

0 [L → [E_s •], {$\#_{2,0}$}]

4

0 [L → [E_s] •, {$\#_{3,0}$}]

5

0 [E_s → LE_s •, {$\#_{2,1}$}] 1 [LE_s → LE_s • , E , {$\#_{2,3}$}]

6

0 [LE_s → E •, {$\#_{2,4}$}]

7

0 [E → a •, {$\#_{2,5}$, $\#_{9,1}$}]

8

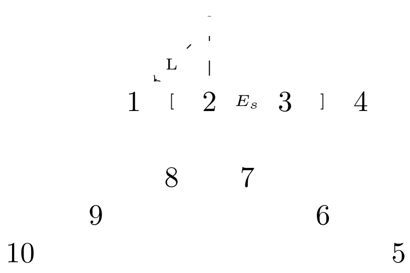
0 [E → L •, {$\#_{2,6}$, $\#_{9,2}$}]

9

0 [LE_s → LE_s , • E, {$\#_{5,1}$}]
1 [E → • a, {$\#_{9,0}$}] 2 [E → • L, {$\#_{9,0}$}] 3 [L → • [E_s], {$\#_{9,2}$}]

10

0 [LE_s → LE_s , E •, {$\#_{9,0}$}]



4 Semántica estática

5 Traducción