Procesadores de lenguaje
Tema 3 – Análisis sintáctico (Parte I)

Salvador Sánchez Alonso, Daniel Rodríguez García
Departamento de Ciencias de la Computación
Universidad de Alcalá

Resumen

- Introducción
  - Conceptos básicos
  - Tipos de analizadores
- Gramáticas independientes del contexto.
- Recursividad y ambigüedad.
- Tipos de análisis sintáctico.
- Análisis sintáctico descendente recursivo con retroceso.
- Análisis sintáctico descendente predictivo LL(1).

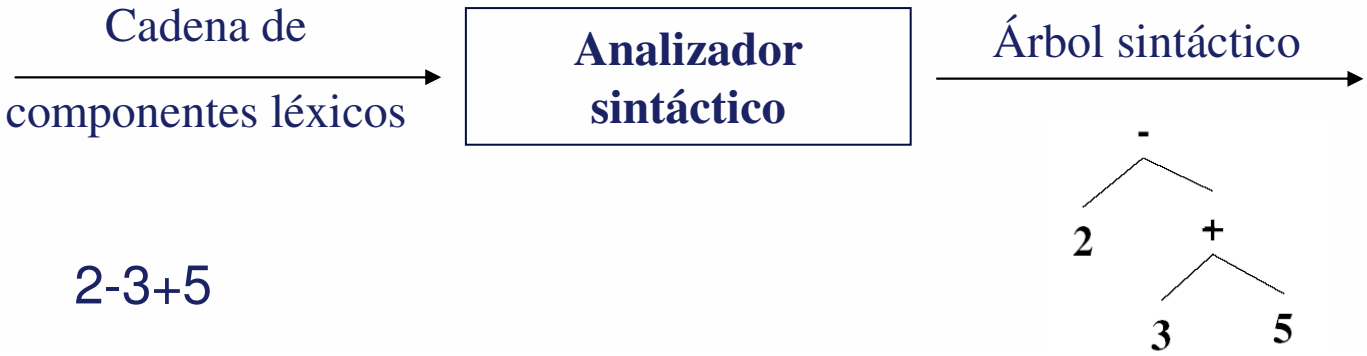
Función del analizador sintáctico

- El analizador sintáctico (*parser*) construye una representación intermedia del programa analizado.
  - Construye un árbol de análisis a partir de los componentes léxicos que recibe, aplicando las producciones de la gramática con el objeto de comprobar la corrección sintáctica de las frases.
- Comprobar que el orden en que el analizador léxico le va entregando los *tokens* es válido:
  - Para ello verifica que la cadena pueda ser generada por la gramática del lenguaje fuente.
- Informar acerca de los errores de sintaxis, recuperándose de los mismos (si es posible) para continuar procesando la entrada.

Función del analizador sintáctico

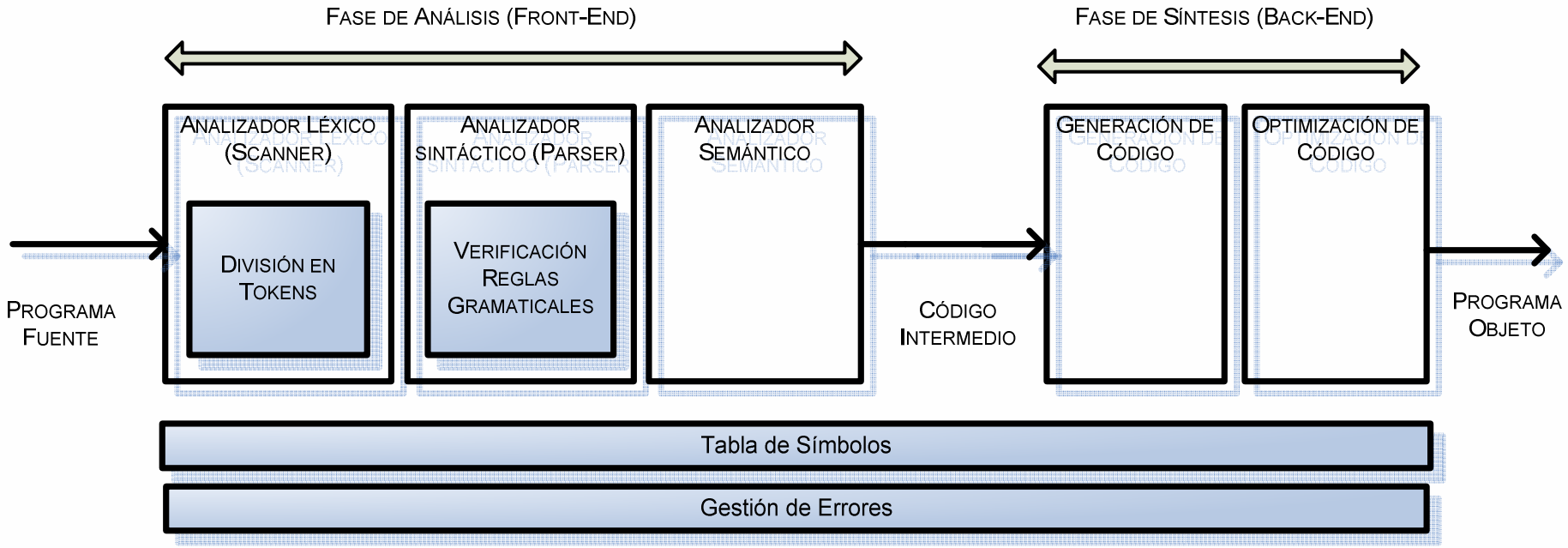
- La salida del analizador sintáctico es una representación en forma de árbol sintáctico de la cadena de componentes léxicos producida por el analizador léxico.

$$E \rightarrow E + E$$
$$E \rightarrow E - E$$
$$E \rightarrow n$$

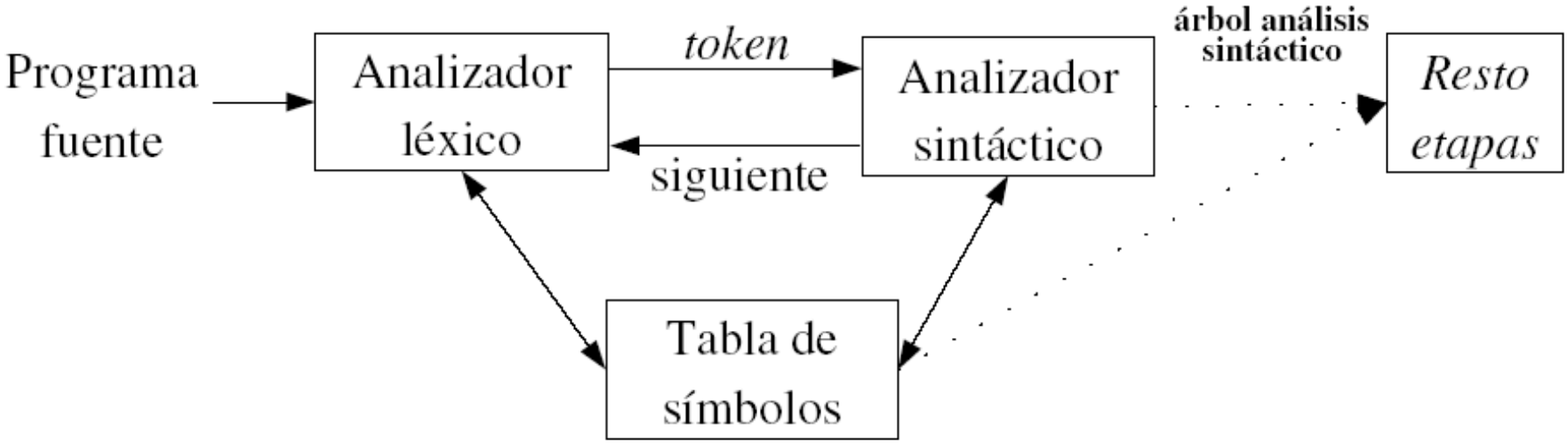


2-3+5

Situación en el modelo del compilador



Situación en el modelo del compilador (más detallado)



Criterios a cumplir

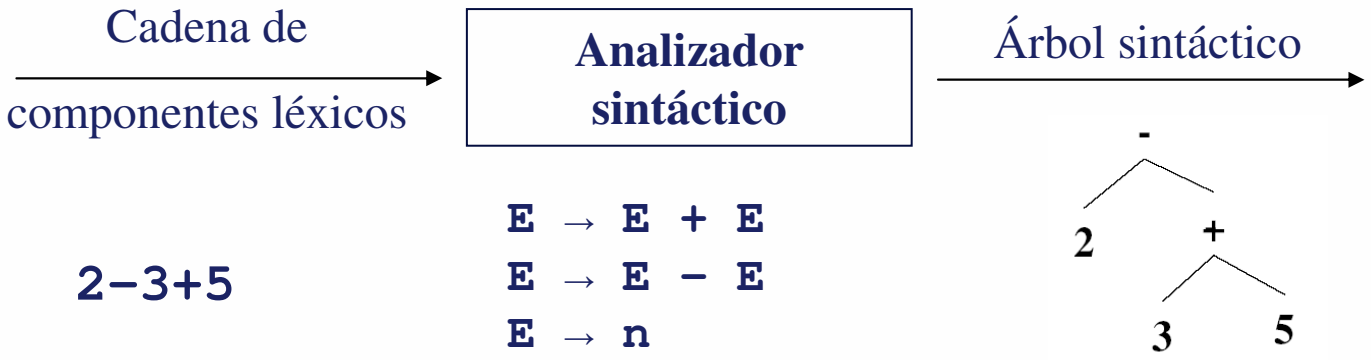
- Eficiencia: en lo posible el tiempo de análisis debe ser proporcional al tamaño del archivo.
- La acción a realizar debe decidirse conociendo un número limitado de componentes léxicos de la entrada.
- Deben evitarse los retrocesos, para lo cual las acciones a realizar se deben poder predecir con exactitud.

Implementación

- De manera similar a lo que ocurre en el léxico, existen dos opciones.
- Utilizando un lenguaje de programación, a través de una serie de técnicas que se explicarán mas adelante.
  - Complejidad, pero mayor control sobre su eficiencia.
- Utilizando un generador de analizadores sintácticos, como CUP (Java), YACC o Bison (C/C++ Gnu).
  - Sencillez, aunque el código generado es más difícil de mantener se tiene un menor control sobre la eficiencia del código generado.

Ejemplo de analizador sintáctico

- Dada la expresión: “2 – 3 + 5”
- El analizador sintáctico utiliza las reglas de producción de la gramática para construir el árbol sintáctico:



Tipos de analizador sintáctico

- Para comprobar si una cadena pertenece al lenguaje generado por una gramática, los analizadores sintácticos construyen una representación en forma de árbol de 2 posibles maneras:
  - Analizadores sintácticos descendentes (*Top-down*)
    - Construyen el árbol sintáctico de la raíz (arriba) a las hojas (abajo).
    - Parten del símbolo inicial de la gramática (axioma) y van expandiendo producciones hasta llegar a la cadena de entrada.
  - Analizadores sintácticos ascendentes (*Bottom-up*)
    - Construyen el árbol sintáctico comenzando por las hojas.
    - Parten de los terminales de la entrada y mediante reducciones llegan hasta el símbolo inicial.
- En ambos casos se examina la entrada de izquierda a derecha, analizando los testigos o tokens de entrada de uno en uno.

AQUI

- Aqui

Gramática independiente del contexto

- La sintaxis de un lenguaje se especifica mediante las denominadas gramáticas independientes del contexto.
- Ventajas de las gramáticas:
  - Describen de forma natural la estructura jerárquica de las construcciones de los lenguajes de programación.
  - Facilitan la construcción de *analizadores sintácticos* eficientes.
  - Si está adecuadamente diseñada, impone una estructura al lenguaje que posteriormente resulta útil para su traducción a código objeto y para la detección de errores.
  - Facilitan la extensión (ampliación con nuevas construcciones) del lenguaje.
- Para cualquier gramática independiente del contexto se puede construir un analizador sintáctico.

Gramática independiente del contexto

- Componentes de una gramática: *G = (Vt, Vn, S, P)*
  - Símbolos terminales (componentes léxicos), *Vt*
  - Símbolos no-terminales, *Vn*
  - Producciones, formadas por no-terminales y terminales, *P*
  - Símbolo inicial o axioma, *S*

- Una gramática se describe
  - Mostrando una lista de sus producciones.
    - Una producción consta de un símbolo no-terminal (parte izquierda), una flecha, y una secuencia de símbolos terminales y no-terminales (parte derecha).

$$\mathtt{E} \rightarrow \mathtt{E} + \mathtt{E}$$

$$\mathtt{E} \rightarrow \mathtt{E} - \mathtt{E}$$

$$\mathtt{E} \rightarrow \mathtt{n}$$

  - Usando la notación Backus-Naur Form (BNF) o Extended BNF (EBNF)

Lenguaje definido por una gramática

- Lenguaje definido por una gramática: conjunto de cadenas de componentes léxicos **derivadas** del símbolo inicial de la gramática.
  - $L(G) = \{s \mid exp \rightarrow^* s\}$
- Por ejemplo:
  - $E \rightarrow (E) \mid a$
    - $L(G) = \{(a), ((a)), (((a))), \ldots\}$
  - $E \rightarrow E + a \mid a$
    - $L(G) = \{a, a+a, a+a+a, \ldots\}$

Ejemplos

- Especificación de la sintaxis de un bloque en lenguaje C mediante una gramática independiente del contexto:

```
Bloque → { Sentencias }

Sentencias → Lista_Sentencias | Є

Lista_Sentencias → Lista_Sentencias ; sentencia
                 | sentencia
```

- Nota: Al igual que en las expresiones regulares, Є (épsilon) denota la cadena vacía

Ejemplos

- Si denominamos E a una expresión aritmética simple, la gramática que especifica la sintaxis de las expresiones entre dígitos sería:

```
E → E operador E
E → dígito
operador → + | - | * | /
dígito → 0|1|2|3|4|5|6|7|8|9
```


BNF - Backus-Naur Form

- Notación alternativa para la especificación de producciones (BNF – *Backus-Naur Form*).
- Representación:
  - **::=** significa “se define como”
  - | significa "or lógico"
  - **< >** encierran los no-terminales
  - Los **terminales** se escriben tal y como son
- Ejemplo:

```
<identificador>::=<letra>|<identificador>[<letra>|<dígito>]
```


Árbol gramatical

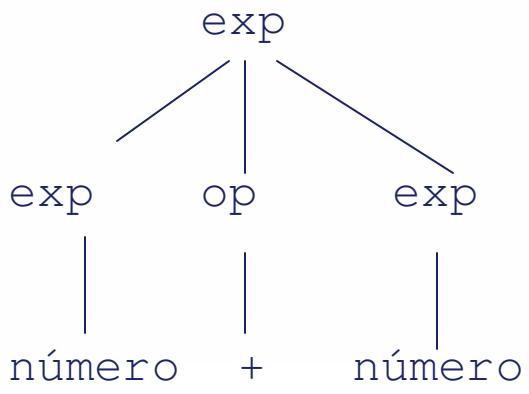
- Un árbol gramatical correspondiente a una derivación es un árbol etiquetado en el cual los nodos interiores están etiquetados por no terminales, los nodos hoja están etiquetados por terminales, y los hijos de cada nodo interno representan el reemplazo del no terminal asociado en un paso de la derivación

exp → exp op exp

exp → numero

op → +

op → -



Derivaciones

- Se denomina **derivación** a la sucesión de una o más producciones:

$$A_1 \Rightarrow A_2 \Rightarrow A_3 \Rightarrow \ldots \Rightarrow A_n \text{ o también } A_1 \overset{*}{\Rightarrow} A_n$$

- Una cadena de componentes léxicos es considerada válida si existe una derivación en la gramática del lenguaje fuente que parta del símbolo inicial y que tras aplicar las producciones a los no terminales, genere la frase a reconocer.
  - Válida: 5 + 3 + 6
  - No válida: -6 + 1

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow n$$
$$E \rightarrow (E)$$

Derivaciones

- Las derivaciones pueden ser por la izquierda o por la derecha (canónicas).

- Derivación por la izquierda: sólo el no-terminal de más a la izquierda de cualquier forma de frase se sustituye en cada paso:

```
E → E + E | E * E | (E) | -E | id

E => -E => -(E) => -(E+E) => -(id+E) => -(id+id)
```


Recursividad

- Permite expresar iteración utilizando un número pequeño de reglas de producción.
- La estructura de las producciones recursivas es:
  - Una o más reglas no recursivas que se definen como caso base.
  - Una o más reglas recursivas que permiten el crecimiento a partir del caso base.
- Ejemplo: Estructura de un tren formado por una locomotora y un número de vagones cualquiera detrás.

Recursividad

- Solución 1 (no recursiva):

  `tren → locomotora`

  `tren → locomotora vagón`

  `tren → locomotora vagón vagón ...`

- Solución 2 (recursiva):
  - Regla base: **`tren → locomotora`**
  - Regla recursiva : **`tren → tren vagón`**
    - La regla recursiva permite el crecimiento ilimitado

Recursividad

- Una gramática se dice que es recursiva si en una derivación de un símbolo no-terminal aparece dicho símbolo en la parte derecha: $\mathbf{A} \overset{*}{\Rightarrow} \mathbf{aAb}$
- Tipos de recursividad:
  - por la **izquierda**: Problemática para el análisis descendente, funciona bien en los analizadores ascendentes.

$$\mathbf{A} \overset{*}{\Rightarrow} \mathbf{Ab}$$

  - por la **derecha**: Utilizada para el análisis descendente.

$$\mathbf{A} \overset{*}{\Rightarrow} \mathbf{aA}$$

  - por **ambos lados**: No se utiliza porque produce gramáticas ambiguas.

Ambigüedad

- Una gramática es **ambigua** si el lenguaje que define contiene alguna cadena que pueda ser generada por más de un árbol sintáctico distinto aplicando las producciones de la gramática.

- Para la mayoría analizadores sintácticos es preferible que la gramática no sea ambigua
  - Es complicado conseguir en todos los casos la misma representación intermedia.
  - El analizador resultante puede no ser tan eficiente

- Es posible, mediante ciertas restricciones, garantizar la no ambigüedad de una gramática.

Ambigüedad

- Algunos generadores automáticos (como Bison-YACC) son capaces de manejar gramáticas ambiguas, no obstante se deben proporcionar reglas para evitar la ambigüedad y generar un único árbol sintáctico.
- Algunas reglas:

No permitir ciclos:

$$S \rightarrow A$$
$$S \rightarrow a$$
$$A \rightarrow S$$

Suprimir reglas que ofrezcan caminos alternativos:

$$S \rightarrow A$$
$$S \rightarrow B$$
$$A \rightarrow B$$

Evitar producciones recursivas en las que las variables no recursivas de la producción puedan derivar a la cadena vacía:

$$S \rightarrow H\ R\ S$$
$$S \rightarrow s$$
$$H \rightarrow h \mid \epsilon$$
$$R \rightarrow r \mid \epsilon$$

Ambigüedad

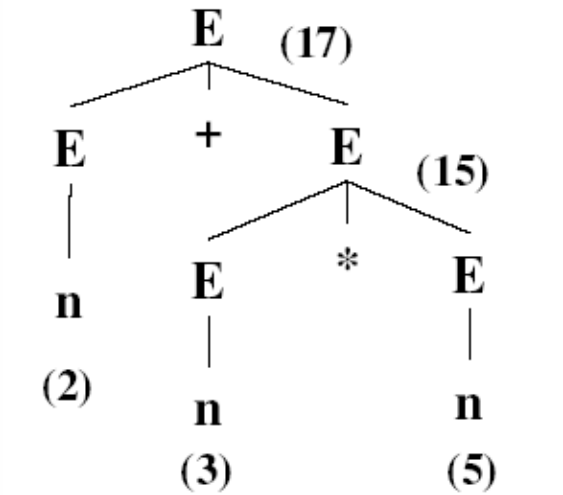
- Ejemplo de gramática ambigua:

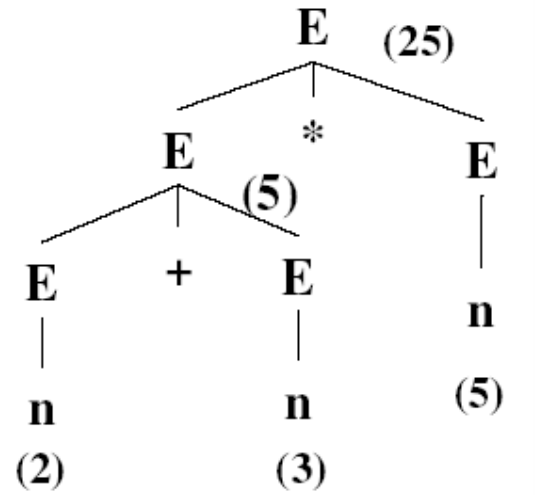
$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow n$

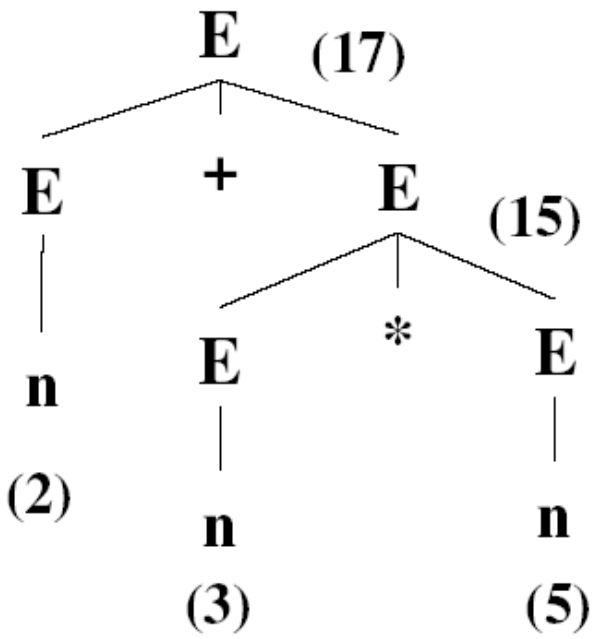
$E \rightarrow (E)$





Ambigüedad

- Posibilidad 1:



Gramática

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow n$

$E \rightarrow (E)$

Producción

$E \rightarrow E + E$

$\rightarrow n + E$

$\rightarrow n + E * E$

$\rightarrow n + n * E$

$\rightarrow n + n * n$

Ambigüedad

- Posibilidad 2:

Gramática

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow n$

$E \rightarrow (E)$

Producción

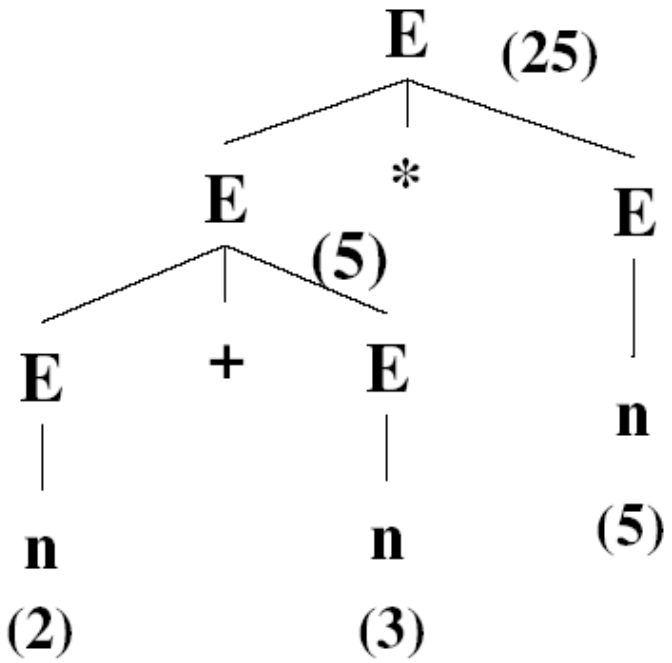
$E \rightarrow E * E$

$\rightarrow E + E * E$

$\rightarrow n + E * E$

$\rightarrow n + n * E$

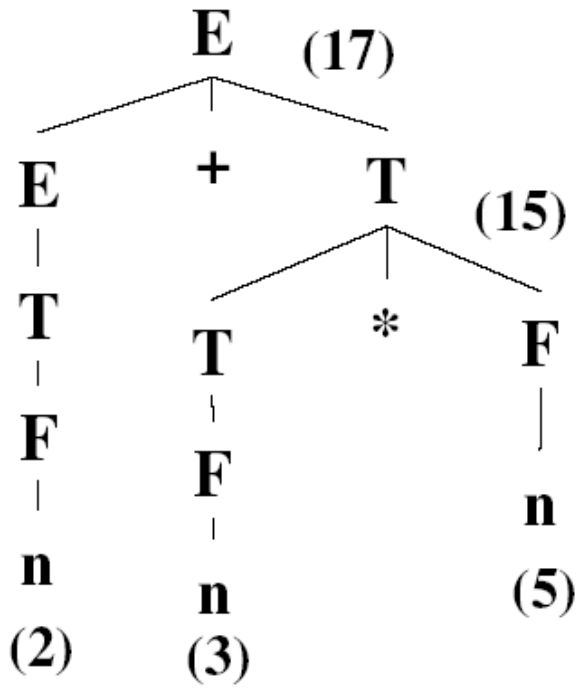
$\rightarrow n + n * n$



Ambigüedad

- Solución a la ambigüedad:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid n$$



Ambigüedad

- En resumen:
  - Ambigüedad implica que a una misma sentencia se le pueden asignar significados (semánticas) diferentes.
  - El algoritmo para poder crear un árbol sintáctico para una gramática ambigua necesita de prueba y retroceso.
  - Si un lenguaje es ambiguo existirán varios significados posibles para el mismo programa, y por tanto el compilador podría generar varios códigos máquina diferentes para un mismo código fuente.
  - Un análisis sintáctico determinista (sin posibilidad de elegir entre varias opciones) es más eficiente.
- **Conclusión**: las gramáticas de los lenguajes de programación no deben ser ambiguas.

Métodos de análisis sintáctico

- El **análisis sintáctico** comprueba que la entrada cumple las condiciones impuestas por la gramática. Dos tipos:
  - Análisis Sintáctico Descendente (ASD). Parten del símbolo inicial de la gramática (axioma) y van expandiendo producciones hasta llegar a la cadena de entrada.
  - Análisis Sintáctico Ascendente. Parten de los terminales de la entrada y mediante reducciones llegan hasta el símbolo inicial.
- Notas:
  - Los métodos descendentes se pueden implementar más fácilmente sin necesidad de utilizar generadores automáticos.
  - Los métodos ascendentes pueden manejar una mayor gama de gramáticas por lo que los generadores automáticos suelen utilizarlos.
  - Para cualquier gramática independiente de contexto hay un analizador sintáctico "general" que toma como máximo un tiempo de $O(n^3)$ para realizar el análisis de una cadena de n componentes léxicos. Se puede conseguir un análisis lineal $O(n)$ para la mayoría de lenguajes de programación.

Análisis Sintáctico Descendente - ASD

- Métodos que parten del axioma y, mediante derivaciones por la izquierda, tratan de encontrar la entrada.
- Existen dos formas de implementarlos:
  - Análisis descendente recursivo
    - Es la manera más sencilla, implementándose con una función recursiva aprovechando la recursividad de la gramática.
  - Análisis descendente predictivo
    - Para aumentar la eficiencia, evitando los retrocesos, se predice en cada momento cuál de las reglas sintácticas hay que aplicar para continuar el análisis
- En la práctica apenas se emplea el recursivo (o con retroceso) debido a diversos inconvenientes.

ASD recursivo

- Mediante un método de **backtracking** se van probando todas las opciones de expansión para cada no-terminal de la gramática hasta encontrar la correcta.
- Cada retroceso en el árbol sintáctico tiene asociado un retroceso en la entrada: se deben eliminar todos los terminales y no terminales correspondientes a la producción que se “elimina” del árbol.
- Si el terminal obtenido como consecuencia de probar con una opción de las varias de una producción no coincide con el *componente léxico* leído en la entrada, hay que retroceder.

Algoritmo

1. Se colocan las reglas en orden, de forma que si la parte derecha de una producción es prefijo de otra, esta última se sitúa detrás.
2. Se crea el nodo inicial con el axioma y se considera nodo activo.
3. Para cada nodo activo A:
   - Si A es un no-terminal, se aplica la primera producción asociada a A.
     - El nodo activo pasa a ser el hijo izquierdo.
     - Cuando se terminan de tratar todos los descendientes, el siguiente nodo activo es el siguiente hijo por la izquierda.
   - Si A es un terminal:
     - Si coincide el símbolo de la entrada, se avanza el puntero de entrada y el nodo activo pasa a ser el siguiente "hermano" de A.
     - Si no, se retrocede en el árbol hasta el anterior no-terminal (y en la entrada si se ha reconocido algún componente léxico mediante la producción que se elimina) y se prueba la siguiente producción.
       - Si no hay más producciones para probar, se retrocede hasta el anterior no-terminal y se prueba con la siguiente opción de éste.
4. Si se acaban todas las opciones del nodo inicial, error sintáctico. Si por el contrario se encuentra un árbol para la cadena de entrada, éxito.

Ejemplo

- Comprobar si **ccd** pertenece al lenguaje de la gramática:

  S → **c** X **d**

  X → **c k** | **c**

Entrada	Pila	Regla a aplicar	Se obtiene en la pila	Se empareja con la entrada	Queda en la pila	Queda en la entrada
c c d	S	S → **c X d**	c X d	c	X d	c d
c d	**X d**	X → **c k**	c k d	c	k d	d
d	k d	d y k no se emparejan, hay que deshacer el último paso (2) y probar la siguiente regla				
c d	**X d**	**X → c**	c d	c	d	d
d	d			d		

ASD Recursivo – Implementación

```
exp → exp op suma term | term
opsuma → + | -
term →  term opmult factor |
  factor
opmult → *
Factor → ( exp )| num
```

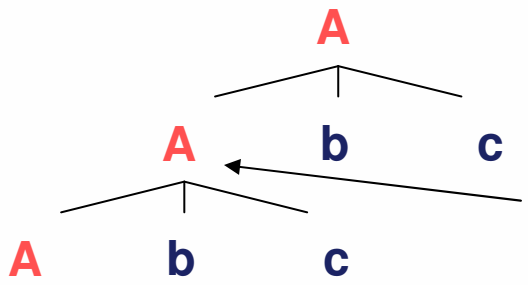
```
procedure factor;
begin
 case token of
 (: match ( "(");
    exp;
    match (")");
 num: match (num);
 else error
 endCase
end

procedure match(expecToken)
begin
 if token=expecTok then
  getToken;
 else
  error
end
```


ASD recursivo - Problemas

- No puede tratar gramáticas con recursividad a izquierdas.

A→Abc



Entra en un bucle infinito

- Acaba la ejecución cuando se encuentra el primer error
  - Difícil proporcionar mensajes más elaborados que "correcto" o "incorrecto", como por ejemplo especificar dónde se ha encontrado el error.
- Aunque la programación es simple, utiliza muchos recursos
  - Como consecuencia del retroceso necesita almacenar los componentes léxicos ya reconocidos por si es necesario volverlos a tratar.
- Cuando un analizador sintáctico se utiliza para comprobar la semántica y generar código, cada vez que se expande una regla, se ejecuta una acción semántica. Al retroceder esa regla o producción se deben deshacer las acciones semánticas, lo que no es fácil ni siempre posible.

Eliminación de la recursividad por la izquierda - Recursión inmediata

- Si la gramática recursiva tiene la forma siguiente:
  - $A \rightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid \ldots \mid A\,\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$
    - A es el no-terminal recursivo.
    - α_i , partes derechas de las reglas recursivas del no terminal A.
    - β_i , partes derechas de las reglas no recursivas del no terminal A.
- La gramática no recursiva equivalente será:
    - $A \rightarrow \beta_1\,A' \mid \beta_2\,A' \mid \ldots \mid \beta_m\,A'$
    - $A' \rightarrow \alpha_1\,A' \mid \alpha_2\,A' \mid \ldots \mid \alpha_n\,A' \mid \varepsilon$
- Ej.: **exp → exp + term | exp – term | term**
  - La gramática sin recursión por la izquierda sería:

```
exp  → term exp'
exp' → + term exp' | - term exp' | ε
```


Eliminación de la recursividad por la izquierda - Recursión indirecta

- Para eliminar la recursividad indirecta se debe encontrar el elemento conflictivo y sustituirlo por su definición. Ej.:

  S → A **a** | **b**

  A → A **c** | S **d** | ε (**A→Sd** es recursiva por S→A**a**)

- Sustituimos

  S → A **a** | **b**

  A → A **c** | A **a d** | **b d** | ε

- Y eliminamos la recursión inmediata como antes:

  S → A **a** | **b**

  A → **b d** A′ | A′

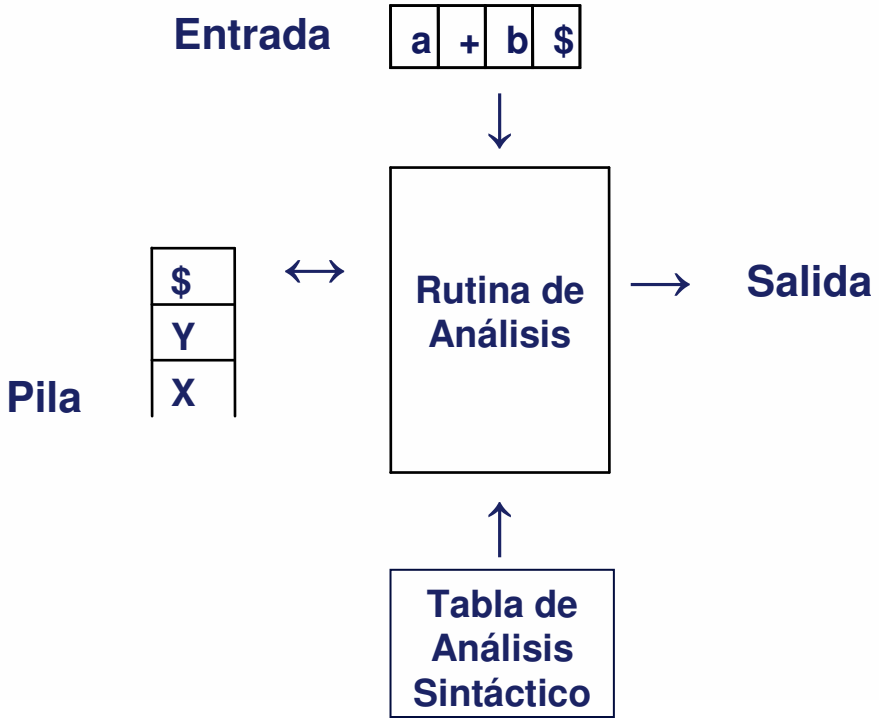
  A′ → **c** A′ | **a d** A′ | ε

ASD predictivo

- Intentan predecir la siguiente construcción a aplicar leyendo uno o más *componentes léxicos* por adelantado
  - "Saben" con exactitud qué regla deben expandir para llegar a la entrada
- Este tipo de analizadores se denomina **LL(k)**
  - Leen la entrada de izquierda a derecha (Left to rigth)
  - Aplican derivaciones por la izquierda para cada entrada (Left)
  - Utilizan **k** componentes léxicos de la entrada para predecir la dirección del análisis.
- Están formados por:
  - Un buffer para la entrada.
  - Una pila de análisis.
  - Una tabla de análisis sintáctico.
  - Una rutina de control.

Componentes de un ASD predictivo

- En función de la entrada, de la tabla de análisis y de la pila decide la acción a realizar.



Posibles acciones:

– Aceptar la cadena.
– Aplicar producción.
– Pasar al siguiente símbolo de la entrada.
– Notificar un error.

Tabla de análisis sintáctico

- Se trata de una matriz **M [V_n, V_t]** donde se representan las producciones a expandir en función del estado actual del análisis y del símbolo de la entrada.
- Las entradas en blanco indican errores

	a	b	c	d	e	$
S	S→BA			S→BA		
A		A→ bSC			A→ϵ	A→ ϵ
B	B→DC			B→ DC		
C		C→ ϵ	C→cDC		C→ϵ	C→ ϵ
D	D→ a			D → dSe		

Ejemplo

Gramática		Cadena entrada	Símbolo en la entrada	Regla a aplicar *	Pila	Derivaciones aplicadas
A → aBc A → xC A → B B → bA C → c	1	babxcc	b	A →B	B	B
	2	babxcc	b	B→ bA	bA	bA
	3	abxcc	a	A→ aBc	aBc	baBc
	4	bxcc	b	B →bA	bAc	babAc
	5	xcc	x	A → xC	xCc	babxCc
	6	cc	c	C → c	cc	babxcc
	7	c	c		c	babxcc
	8					babxcc

* La regla a aplicar vendría dada por la tabla de análisis sintáctico.

Analizador LL(1)

- Una gramática es LL(1) si la tabla de análisis sintáctico asociada tiene como máximo una producción en cada entrada de la tabla
  - Es decir, sólo se necesita mirar el primer token (testigo) de la entrada para decidir que regla aplicar
- El análisis descendente más común es LL(1) pues con *k>1* la tabla de análisis es mucho más complicada
  - No se usan LL(2), LL(3), etc. en la práctica
- Sin embargo, no todas las gramáticas se pueden analizar mediante métodos descendentes predictivos

Analizador LL(1)

- Para determinar si a una gramática se le puede aplicar el método LL(1), debe comprobarse que:
  - No es recursiva por la izquierda
  - Si existe un conjunto de reglas de la forma $A \rightarrow \alpha_1 | \alpha_2 | ... | \alpha_n$, es posible decidir la opción a escoger mirando un *token* de la entrada, i.e., dos productores de un mismo terminal no pueden dar lugar al mismo testigo (simbolo terminal) inicial.

Conjuntos de Predicción

- Para construir la tabla de análisis y probar si la gramática es LL(1) se construyen dos conjuntos:
  - *Primero*
  - *Siguiente*

- Los conjuntos Primeros se calculan para todos los símbolos de la gramática, terminales y no terminales.

- Los conjuntos Siguiente se definen sólo para los no terminales.

Analizador LL(1) – Conjunto *Primero*

- Dado un símbolo de la gramática A, su conjunto **Primero** es el conjunto de terminales por los que comienza cualquier frase que se genere a partir de A mediante derivaciones por la izquierda de las producciones.

  $a \in \text{Primero}(\alpha_p)$ si $a \in (Vt \cup \varepsilon) \mid \alpha_p \rightarrow^* a\,\beta$

- Algoritmo de cálculo del conjunto primero:
  - Si $A \in Vt \Rightarrow \text{Prim}(A) = A$
  - Si $\exists\, A \rightarrow \varepsilon \Rightarrow \varepsilon \in \text{Prim}(A)$
  - Si $\exists\, A \rightarrow Y_1 Y_2 \ldots Y_n \Rightarrow \{\text{Prim}(Y_1) - \varepsilon\} \in \text{Prim}(A)$
  - Si $\varepsilon \in \text{Prim}(Y_1) \Rightarrow \{\text{Prim}(Y_2) - \varepsilon\} \in \text{Prim}(A)$
  - Si $\varepsilon \in \text{Prim}(Y_2) \Rightarrow \{\text{Prim}(Y_3) - \varepsilon\} \in \text{Prim}(A)$
  - …
  - Si $\forall i, \varepsilon \in \text{Prim}(Y_i) \Rightarrow \varepsilon \in \text{Prim}(A)$

Ejemplo – Conjunto Primero

A→ BC

B→ ϵ | m

C→ ϵ | s

SOLUCIÓN:

Primero(A) = {m, s, ϵ}

Primero(B) = {m, ϵ}

Primero(C) = {s, ϵ}

Ejemplo – Conjunto Primero

Gramática	***Primeros* de la regla**	***Primeros* del no terminal**
expr ⟶ expr + expr1 expr ⟶ expr - expr1 expr ⟶ expr1	{ (, *num*} { (, *num*} { (, *num*}	De entrada, *Primeros*(*expr*) = {∅} *Primeros* (*expr*) = *Primeros*(*expr*) U *Primeros*(*expr1*) = { (, *num*}
expr1 ⟶ expr2 *expr1 expr1 ⟶ expr2 /expr1 expr1 ⟶ expr2	{ (, *num*} { (, *num*} { (, *num*}	*Primeros* (*expr1*) = *Primeros*(*expr2*) = { (, *num*}
expr2 ⟶ expr3 ^expr2 expr2 ⟶ expr3	{ (, *num*} { (, *num*}	*Primeros* (*expr2*) = *Primeros*(*expr3*) = { (, *num*}
expr3 ⟶ (expr) expr3 ⟶ num	{ (} { *num*}	*Primeros* (*expr3*) = { (, *num*}

Conjunto *Siguientes*

- Dado un símbolo **A**, su conjunto **Siguiente** es el conjunto de terminales que aparecen inmediatamente después de A en alguna sentencia por la que se pasa al realizar una derivación por la izquierda.

  $a \in$ Siguientes(A) si $a \in (Vt \cup \{\$\}) \mid \exists\ S \Rightarrow^* \ldots Aa \ldots$

- Algoritmo:
  - Siguiente (S) = $, siendo S es el axioma.
  - Para cada símbolo no-terminal **A** de la gramática, añadir elementos aplicando las siguientes reglas:
  - Si $\exists\ X \rightarrow \alpha AB \Rightarrow \{Primero(B) - \varepsilon\} \in$ Siguiente(A)
  - Si $(\exists\ B \rightarrow \alpha A) \vee (B \rightarrow \alpha AC \wedge \varepsilon \in$ Primero(C)) $\Rightarrow$ Siguiente(B)$\in$ Siguiente (A)
- ε nunca es elemento de un conjunto siguiente.

Ejemplo Conjunto Siguiente

S → aBCd

B → bb

C → cc

- Solución:

  Prim(S)={a}, Prim(B)={b}, Prim(C)={c}

  Sig(S)={$}, Sig(B)={c}, Sig(C)={d}

Ejemplo Conjunto Siguiente

Gramática	Conjunto *Primeros*	Conjunto *Siguientes*
expr ⟶ expr + expr1 expr ⟶ expr - expr1 expr ⟶ expr1	*Primeros*(*expr*) = { (, *num*}	*Siguientes* (*expr*) = { +, -,), $ }
expr1 ⟶ expr2 * expr1 expr1 ⟶ expr2 | expr1 expr1 ⟶ expr2	*Primeros*(*expr1*) = { (, *num*}	*Siguientes* (*expr1*) = *Siguientes*(*expr*) = { +, -,), $ }
expr2 ⟶ expr3 ^ expr2 expr2 ⟶ expr3	*Primeros*(*expr2*) = { (, *num*}	*Siguientes* (*expr2*) = *Siguientes*(*expr1*) U { *, / } = { +, -,), $, *, / }
expr3 ⟶ (expr) expr3 ⟶ num	*Primeros*(*expr3*) = { (, *num*}	*Siguientes* (*expr3*) = *Siguientes*(*expr2*) U{ ^ } = { +, -,), $, *, /, ^ }

Comprobar si una gramática es LL(1)

- Una gramática es LL(1) si y sólo si
  - no es recursiva a izquierdas y
  - Si contiene producciones A → $\alpha_1 | \alpha_2 | \ldots | \alpha_n$, se cumple que no hay dos partes derechas que comiencen por un mismo terminal:

$$\text{Primero}(\alpha_i) \cap \text{Primero}(\alpha_j) = \{\}, \forall\ i \neq j$$

    - A lo sumo un α_i puede derivar la cadena vacía ϵ.
    - Si un no terminal $\alpha_i \overset{*}{\Rightarrow} \epsilon$, entonces no hay otro α_j que comience por un elemento que esté en Siguiente(A):

$$\text{Primero}(\alpha_j) \cap \text{Siguiente}(A) = \emptyset$$

Modificar Gramáticas para LL(1)

- Cuando se da alguno de estos casos hay que modificar la gramática:
  - Eliminando la recursión por la izquierda
  - Sacando algún factor común por la izquierda

Gramática original	**Gramática modificada**
$A \rightarrow A\ a \mid bB$	$A \rightarrow bB\ A'$ $A' \rightarrow aA' \mid \varepsilon$ se elimina la recursividad
$B \rightarrow b\ c \mid b\ b \mid b$	$B \rightarrow b\ B'$ $B' \rightarrow c \mid b \mid \varepsilon$ se saca los factores común y se agrupa

Construcción de la tabla de análisis

1. Para cada producción de la forma **A** → **α** aplicar las reglas 2 y 3.

2. Para todo terminal "t", tal que **t** ∈ **Primero(α)** hacer:

$$M\,[V_n, V_t] = A \rightarrow \alpha$$

3. Si **ε** ∈ **Primero(α)** hacer:

$$\forall\ t \in \text{Siguiente}(A),\ M\,[V_n, V_t] = A \rightarrow \alpha$$

4. Las entradas no marcadas (vacías) corresponden a errores sintácticos.

Analizador LL(1) - Ejemplo

1. Comprobar si la siguiente gramática es LL(1) y construir la tabla, calculando todos los conjuntos Siguiente y Primero.

$$S \rightarrow cA$$
$$A \rightarrow aB$$
$$B \rightarrow b \mid \epsilon$$

2. Reconocer la cadena "**`cab`**" con el analizador construido.

Ejemplo – Solución

Pasos para ver si es LL(1):

1.- No es recursiva a izquierdas.
2.- $B \rightarrow b \mid \epsilon$
  2.1. $Prim(b) \cap Prim(\epsilon) = \emptyset$
  2.2. Si $\epsilon \in Prim(\epsilon) \Rightarrow Prim(b) \cap Sig(B) = \emptyset$

Conjuntos primero:

$Prim(S)=\{c\}$ $Prim(A)=\{a\}$ $Prim(B)=\{b, \epsilon\}$

Conjuntos siguiente:

$Sig(S)=\{\$\}$ $Sig(A)=Sig(S)=\{\$\}$ $Sig(B)=Sig(A)=\{\$\}$

Solución

Tabla de análisis:

	c	a	b	$
S	S → cA			
A		A → aB		
B			B → b	B → ϵ

Solución

Reconocer la cadena “cab”

$S		cab$
$Ac	Reconocimiento	cab$
$A	A→aB	ab$
$Ba	Reconocimiento	ab$
$B	B→b	b$
$b	Reconocimiento	b$
$	éxito	$

Bibliografía

- *Compiladores: principios, técnicas y herramientas*. A.V. Aho, R. Sethi, J.D. Ullman. Addison-Wesley Iberoamerica. 1990.

- *Construcción de compiladores. Principios y práctica.* Kenneth C. Louden. Thomson-Paraninfo. 2004.