
Análisis Sintáctico (I)

Diseño de Lenguajes de Programación

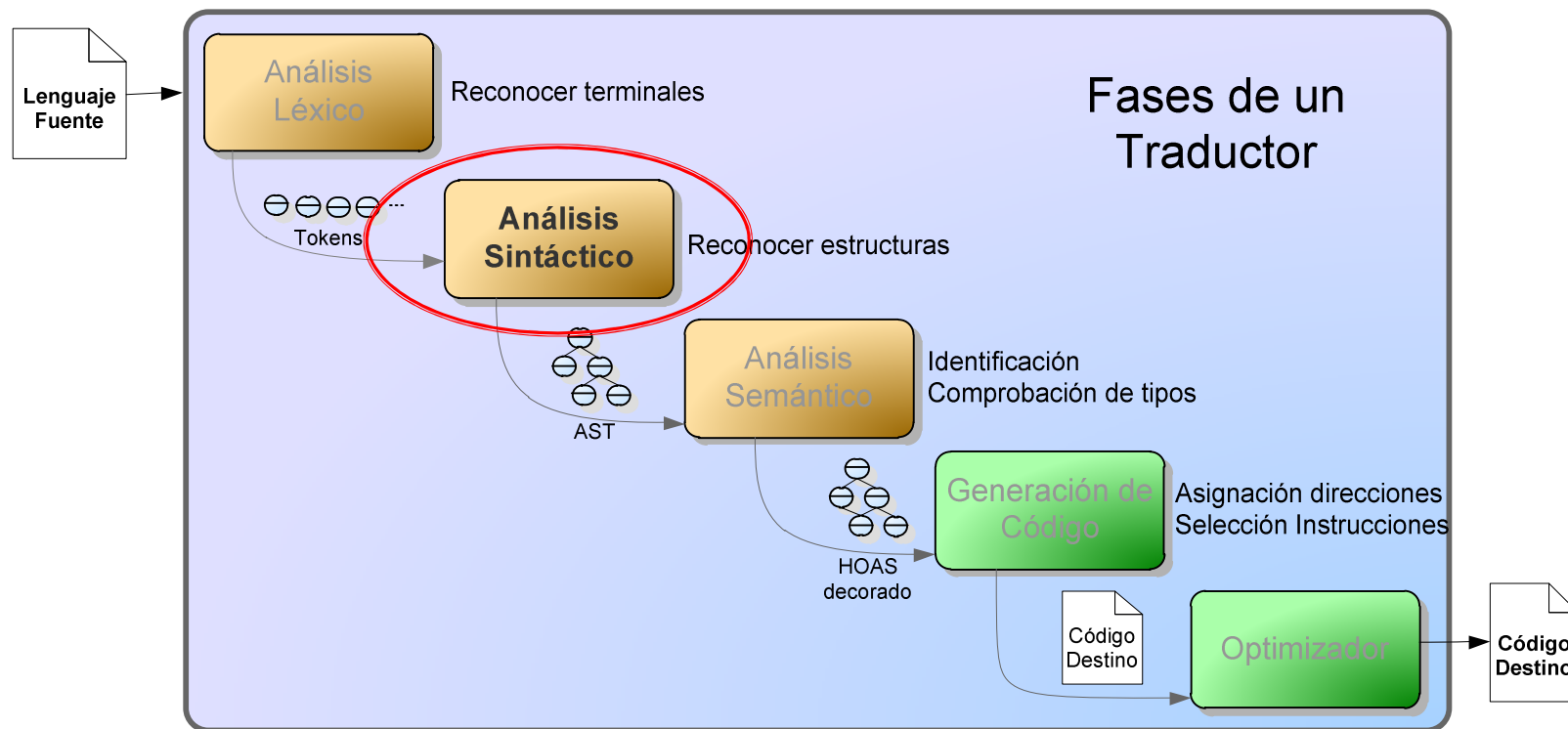
Ingeniería Informática

Universidad de Oviedo

(v1.12)

Raúl Izquierdo Castanedo

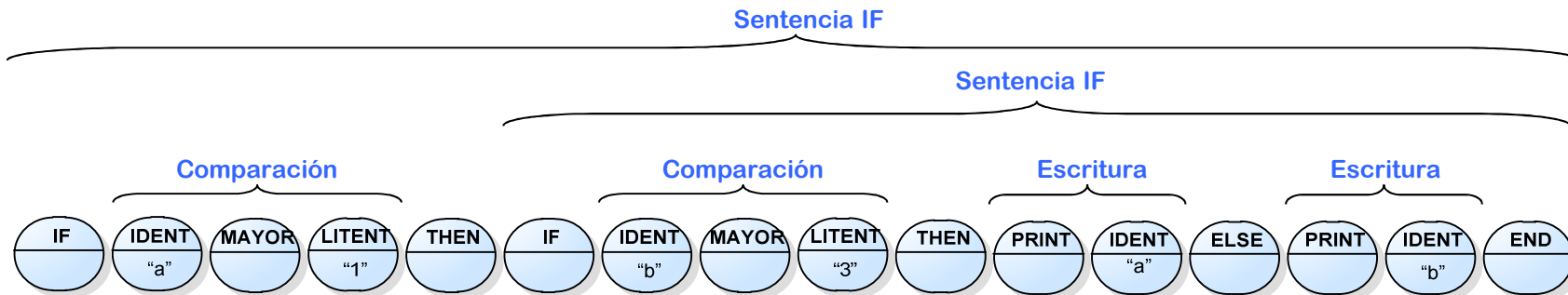
Análisis Sintáctico



Funciones

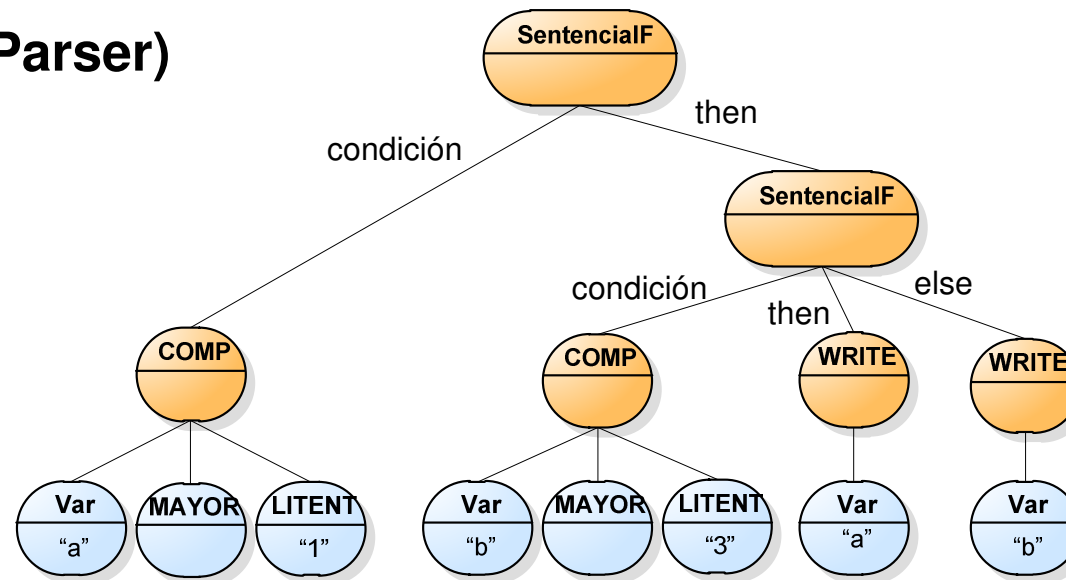
1. Reconocer estructuras (Analizador)

Esta clase y la siguiente



2. Crear AST (Parser)

Clase 3 del sintáctico



Metalinguaje Sintáctico

Análisis Sintáctico

Metalinguajes

Lenguaje

$3 * (4 + 5)$

Definición en lenguaje natural

- Léxico: números, paréntesis, suma y producto
- Sintáctico
 - Las expresiones utilizan notación infija
 - El número de paréntesis abierto \geq cerrados. Deben ser iguales al finalizar

Gramáticas Libres de Contexto (GLC)

Gramática

$G = \{VT, VN, s, P\}$

VT = Símbolos terminales

- Tokens

VN = Símbolos no-terminales

- Estructuras del lenguaje

$s \in VN$

- Estructura que comprende a todas las demás

P = Reglas de producción

- $n \rightarrow \alpha$ / $n \in VN, \alpha \in (VN \cup VT)^*$
- Indican la composición de las estructuras

Ejemplo

$G = \{VT, VN, programa, P\}$

VT = { ident if = () entero + * }

VN = { programa, instrucciones, instr, expr }

P = {

programa \rightarrow instrucciones

instrucciones \rightarrow instr

instrucciones \rightarrow instrucciones instr

instr \rightarrow ident = expr

instr \rightarrow ident (expr)

instr \rightarrow if expr then instr else instr

expr \rightarrow entero

| expr + expr

| expr * expr

}

¿Válido " $a = b = 0$ "?
¿else opcional?
¿if anidados?

Notación que usaremos

Notación de una GLC

```
G = {VT, VN, programa, P}

VT = { ident if = ( ) entero + * }
VN = { programa, instrucciones,
      instr, expr }
P = {
  programa → instrucciones
  instrucciones → instr
  instrucciones → instrucciones instr

  instr → ident = expr
  instr → ident ( expr )
  instr → if expr then instr else instr

  expr → entero
        | expr + expr
        | expr * expr
}
```

Notación simplificada para una GLC

Solo se indicará P. El resto se deduce:

- VT serán los símbolos en mayúsculas y los caracteres no alfanuméricos
- VN serán los símbolos en minúsculas
- s es antecedente de la primera regla

```
programa: instrucciones
instrucciones: instr
              | instrucciones instr
```

```
instr: IDENT = expr
      | IDENT ( expr )
      | IF expr THEN instr
        ELSE instr
```

```
expr: ENTERO
     | expr + expr
     | expr * expr
```

Definiciones

Definiciones (I)

Transformación (o paso de derivación)

- Sea

$$G = \{V_T, V_N, s, P\}$$

- Se dice que $\beta\alpha\gamma$ es una transformación de $\beta n\gamma$

$$\beta \mathbf{n} \gamma \Rightarrow \beta \mathbf{\alpha} \gamma \quad (\text{con } \beta n\gamma \text{ y } \beta\alpha\gamma / n \in V_N \text{ y } \beta, \gamma, \alpha \in V^*)$$

- Si existe

$$(n \rightarrow \alpha) \in P$$

Ejemplo

$$X \mathbf{a} b Z \Rightarrow X \mathbf{Y} \mathbf{a} b Z$$

GLC

$$s \rightarrow a b Z$$

$$a \rightarrow X a$$

$$\mathbf{a} \rightarrow \mathbf{Y} \mathbf{a}$$

$$a \rightarrow$$

$$b \rightarrow W b$$

$$b \rightarrow$$

Definiciones (II)

Derivación

- α_n es una derivación de α_1 si se obtiene aplicando una o más transformaciones

$$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$$

$$\alpha_1 \xRightarrow{*} \alpha_n$$

$$X \ a \ b \ Z \Rightarrow X \ \mathbf{Y} \ \mathbf{a} \ b \ Z \Rightarrow X \ Y \ b \ Z \Rightarrow X \ Y \ \mathbf{W} \ \mathbf{b} \ Z$$

□ Por tanto

$$X \ a \ b \ Z \xRightarrow{*} X \ Y \ W \ b \ Z$$

- Cadena anulable

$$\alpha \xRightarrow{*} \lambda$$

GLC

$$s \rightarrow a \ b \ Z$$

$$a \rightarrow X \ a$$

$$a \rightarrow Y \ a$$

$$a \rightarrow$$

$$b \rightarrow W \ b$$

$$b \rightarrow$$

Definiciones (III)

Sentencia

- Sea s el símbolo inicial de la gramática
- α será una sentencia si cumple

$$s \xRightarrow{*} \alpha$$

y $\alpha \in VT^*$

- Ejemplo

$$\begin{aligned} s &\Rightarrow \mathbf{a} \mathbf{b} \mathbf{Z} \Rightarrow \mathbf{X} \mathbf{a} \mathbf{b} \mathbf{Z} \Rightarrow \mathbf{X} \mathbf{b} \mathbf{Z} \\ &\Rightarrow \mathbf{X} \mathbf{W} \mathbf{b} \mathbf{Z} \Rightarrow \mathbf{X} \mathbf{W} \mathbf{Z} \end{aligned}$$

$$s \xRightarrow{*} \mathbf{X} \mathbf{W} \mathbf{Z}$$

GLC

$$s \rightarrow a \ b \ Z$$

$$a \rightarrow X \ a$$

$$a \rightarrow Y \ a$$

$$a \rightarrow$$

$$b \rightarrow W \ b$$

$$b \rightarrow$$

Definiciones (IV)

Lenguaje

$$G = \{VT, VN, s, P\}$$

$$L(G) = \{t \in VT^* \mid s \xRightarrow{*} t\}$$

GLC

$$s \rightarrow X a$$

$$a \rightarrow Y b$$

$$a \rightarrow \lambda$$

$$b \rightarrow W$$

$$b \rightarrow \lambda$$

$$L(GLC) = \{ \quad \quad \quad \}$$

Objetivo

Nuestro objetivo no es generar el lenguaje

Nuestro objetivo

X Y W

□ ¿Pertenece?

$s \rightarrow X a$

$a \rightarrow Y b$

$a \rightarrow \lambda$

$b \rightarrow W$

$b \rightarrow \lambda$

□ Buscar derivación

$s \Rightarrow \mathbf{X a} \Rightarrow X \mathbf{Y b} \Rightarrow X Y \mathbf{W}$

Equivalencia (I)

Sean las siguientes GLC

$$\begin{array}{l} 1) \quad s \rightarrow s + s \\ \quad \quad | \quad X \end{array}$$

$$\begin{array}{l} 2) \quad s \rightarrow X \text{ mt} \\ \quad \text{mt} \rightarrow + X \text{ mt} \\ \quad \text{mt} \rightarrow \lambda \end{array}$$

Equivalencia (II)

Dado un lenguaje hay INFINITAS gramáticas *equivalentes*

- *Pero no todas tienen las mismas propiedades*

Árbol de Análisis Sintáctico (árbol concreto)

Árbol de análisis gramatical o sintáctico

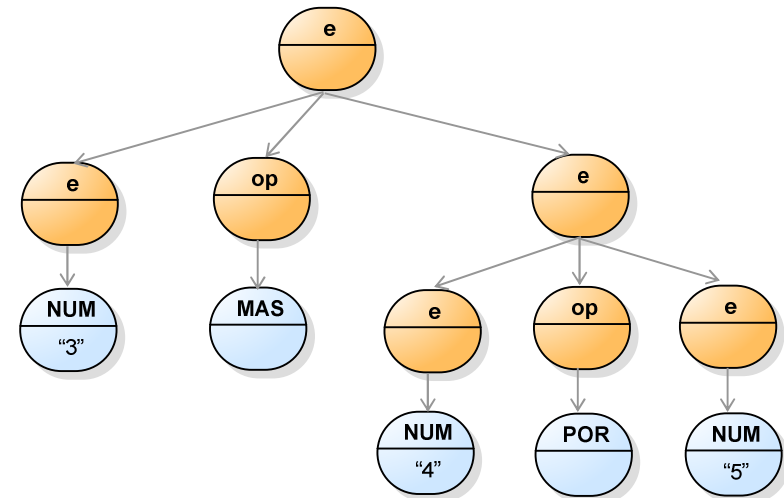
- Registro de las transformaciones realizadas en una derivación
- Muestra las estructuras encontradas en la entrada

Se construye a la vez que se realiza la derivación

- Ejemplo. Derivación a Izquierda de '3 + 4 * 5'

$$\begin{aligned} & \Rightarrow \overbrace{e \quad op \quad e}^e \\ & \Rightarrow \mathbf{NUM} \quad op \quad e \\ & \Rightarrow NUM \quad + \quad e \\ & \Rightarrow NUM \quad + \quad \overbrace{e \quad op \quad e}^e \\ & \Rightarrow NUM \quad + \quad \mathbf{NUM} \quad op \quad e \\ & \Rightarrow NUM \quad + \quad NUM \quad * \quad e \\ & \Rightarrow NUM \quad + \quad NUM \quad * \quad \mathbf{NUM} \end{aligned}$$

$$\begin{aligned} e &\rightarrow e \quad op \quad e \\ e &\rightarrow \mathbf{NUM} \\ op &\rightarrow + \\ op &\rightarrow * \end{aligned}$$



Formato

- El nodo raíz es s
- Nodos hoja $\subset VT$
- Nodos internos $\subset VN$
- Si $hijos(n) = \{x, y, z\}$ entonces ' $n \rightarrow x \ y \ z$ ' $\in P$

Ambigüedad

Gramática Ambigua

GLC

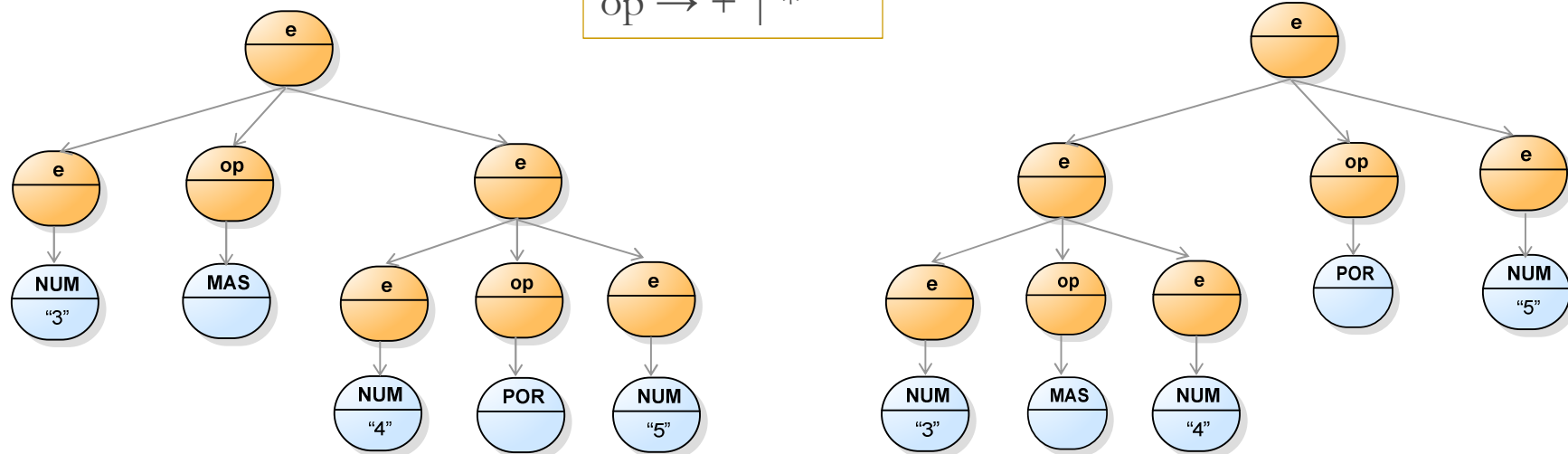
$e \rightarrow \text{NUM}$

$e \rightarrow e \text{ op } e$

$\text{op} \rightarrow + \mid *$

Entrada

$3 + 4 * 5$



- ❑ Cada sentencia debe tener un solo árbol concreto
 - ¿Por qué?
- ❑ ¿Se pueden detectar?
- ❑ Lenguajes intrínsecamente ambiguos

Resumen

Dada una cadena se quiere saber si es válida (si pertenece al lenguaje)

- Hay que buscar una derivación
 - Si no se encuentra
 - Cadena no válida
 - Si se encuentra
 - En cada paso de derivación se enlaza el no-terminal con los símbolos que lo han sustituido
 - Se obtiene así la estructura de la cadena (árbol concreto)
 - La GLC debe ser no-ambigua para que sólo exista un árbol

Creación de Gramáticas

Creación de Gramáticas

Análisis Sintáctico

- Para saber si una sentencia pertenece al lenguaje hace falta una gramática
- ¿Cómo se crea una gramática?
 - Identificando las construcciones básicas en el lenguaje

Creación de Gramáticas

Construcciones básicas

- Secuencias
 - Indican el orden el que deben aparecer los componentes de la estructura

```
asignación: IDENT = expr ';' ;
```

- Listas
 - Indican que la estructura se forma repitiendo otra estructura

```
instrucciones: instrucción  
              | instrucciones instrucción
```

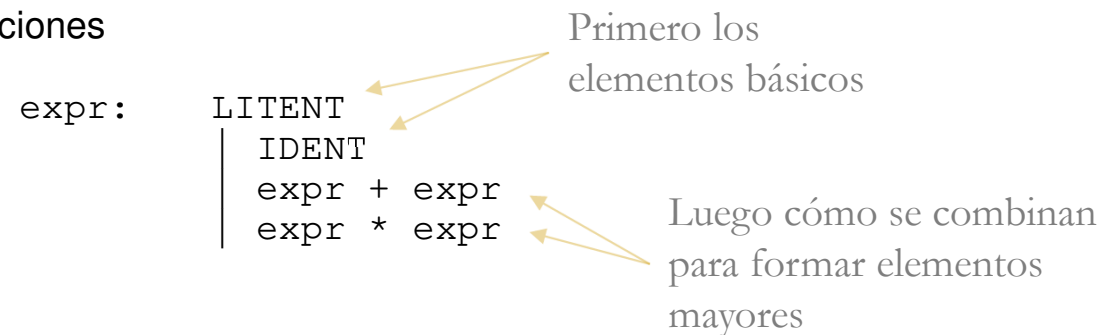
- Forman un árbol desequilibrado a izquierda o a derecha

- Composiciones

```
expr: LITENT  
      | IDENT  
      | expr + expr  
      | expr * expr
```

Primero los elementos básicos

Luego cómo se combinan para formar elementos mayores



Ejercicio E1

Hacer una GLC que genere el siguiente lenguaje

- Un conjunto está formado por uno o más elementos entre paréntesis
- Cada elemento puede ser un número u otro conjunto
- Los números están formados por dígitos de 1 al 3
- Los números están formados por un número impar de dígitos y son capicúa

(131)

(3 222 12321)

(12121 2 (333) (2 3 1111111))

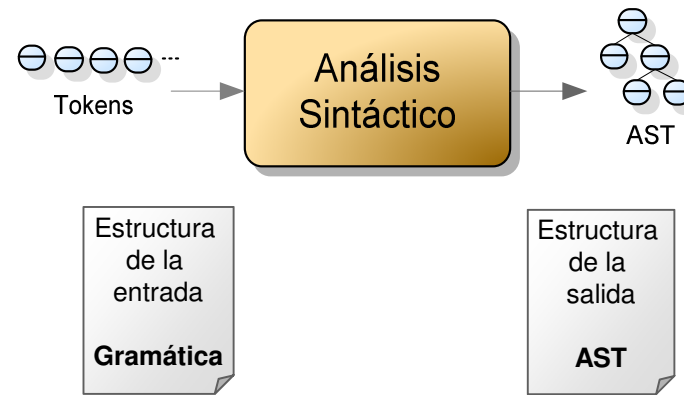
(2132312 (((1 2) 2 131) 1 2) 3322233)

Estrategias de Parsing

Estrategias de Parsing

Hasta ahora se han visto los interfaces de entrada y de salida

- El *qué* hace



- Técnicas para el *cómo*
 - Análisis Descendente
 - **Análisis Ascendente**

Análisis Ascendente

Análisis Ascendente

■ Gramática

$s \rightarrow A a b E$

$a \rightarrow a B C$

$a \rightarrow B$

$b \rightarrow D$

$b \rightarrow a B C$

■ Reconocer A B B C D E

Pivote

A B B C D E
 $\Leftarrow A \mathbf{a} B C D E$
 $\Leftarrow A \mathbf{a} D E$
 $\Leftarrow A a \mathbf{b} E$
 $\Leftarrow \mathbf{s}$

■ ¿Qué pivote elegir?

A B B C D E
 $\Leftarrow A \mathbf{a} B C D E$
 $\Leftarrow A a \mathbf{a} C D E$
 $\Leftarrow \dots$

■ ¿Qué regla a aplicar para el pivote?

A B B C D E
 $\Leftarrow A \mathbf{a} B C D E$
 $\Leftarrow A \mathbf{b} D E$
 $\Leftarrow \dots$

Análisis Ascendente

Análisis con Retroceso

- Combina pivotes y reglas

Reducción por Desplazamiento

- En cada paso se asegura pivote y/o regla que llevan a la cadena inicial
 - Solo puede hacer con ciertas GLC
- Basado en una pila
 - Comienza vacía
 - Operaciones
 - Shift: Introducir token
 - Reduce. Dada $n \rightarrow \beta$ extrae β **de la cima** e introduce n
 - Si la entrada pertenece al lenguaje
 - Quedará sólo s en la pila
 - En la entrada solo estará $\$$

Reducción por Desplazamiento

Ejemplo

$s \rightarrow \text{exp}$

$\text{exp} \rightarrow \text{exp} + \text{LITENT}$

$\text{exp} \rightarrow \text{LITENT}$

Entrada: 5 + 4

1) shift

2) $\text{exp} \rightarrow \text{LITENT}$

3) shift

4) shift

5) $\text{exp} \rightarrow \text{exp} + \text{LITENT}$

6) $s \rightarrow \text{exp}$

	5 + 4 \$
LITENT	+ 4 \$
exp	+ 4 \$
exp +	4 \$
exp + LITENT	\$
exp	\$
s	\$

Reducción por Desplazamiento

Ejemplo

$s \rightarrow \text{par}$

$\text{par} \rightarrow (\text{par}) \text{par}$

$\text{par} \rightarrow \lambda$

Entrada: $()$

1) shift

2) $\text{par} \rightarrow \lambda$

3) shift

4) $\text{par} \rightarrow \lambda$

5) $\text{par} \rightarrow (\text{par}) \text{par}$

6) $s \rightarrow \text{par}$

	() \$
() \$
(par) \$
(par)	\$
(par) par	\$
par	\$
s	\$

En la pila
siempre está λ

Reducción por Desplazamiento

Gramáticas LR(k)

- Gramáticas que pueden ser reconocidas mediante el algoritmo de reducción por desplazamiento
 - Cuando se puede determinar la acción a realizar (shift o reduce) ante cualquier estado de la pila
 - Trabajaremos con LR(1)
- ¿Cómo se comprueba si una gramática es LR(1)?

Gramáticas LR(1)

Ejemplo 1

$s \rightarrow a \ b$
 $a \rightarrow \text{LITENT}$
 $b \rightarrow \text{LITENT}$

■ ¿Determinista?

	25	32	\$
LITENT	32	\$	
a	32	\$	
a LITENT	\$		

Ejemplo 2

$s \rightarrow a \mid b$
 $a \rightarrow \text{LITENT}$
 $b \rightarrow \text{LITENT}$

□ Conflicto reduce/reduce (r/r)

Gramáticas LR(1)

Ejemplo 1

$s \rightarrow a \text{ RETURN} \mid b \text{ PRINT}$
 $a \rightarrow \text{LITENT}$
 $b \rightarrow \text{LITENT}$

LITENT

? \$

- Por tanto, que haya reglas con la misma parte derecha no significa que la GLC no pueda ser LR(1)

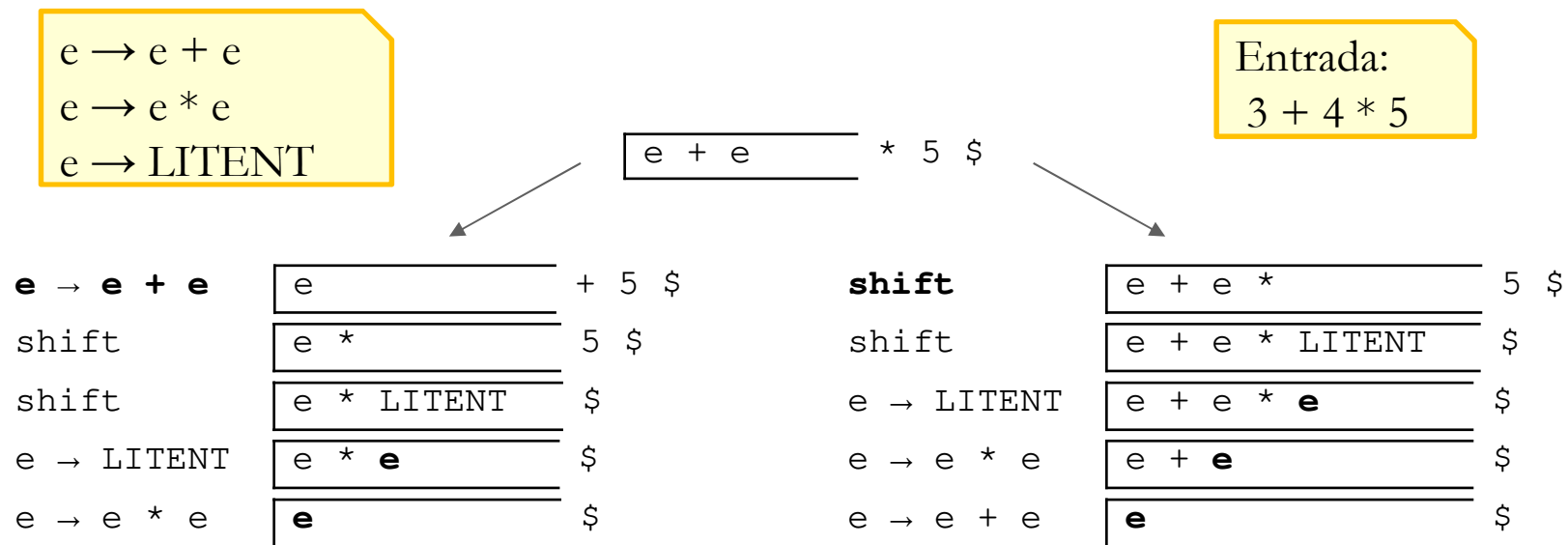
Ejemplo 2

$s \rightarrow a \text{ RETURN RETURN} \mid b \text{ RETURN PRINT}$
 $a \rightarrow \text{LITENT}$
 $b \rightarrow \text{LITENT}$

Gramáticas LR(1)

No haber reglas con misma parte derecha ¿implica LR(1)?

- No habrá conflictos r/r
 - Pero puede haber otro tipo de conflicto (el más habitual)



- Conflicto shift/reduce (s/r)
 - En un estado no se sabe cuál de las dos acciones realizar
 - Por definición no pueden ser LR(K)
- Una GLC ambigua no es LR(K)

Ejercicio E2

Dibujar la traza del reconocimiento de la siguiente entrada con cada gramática

```
int a, b;
```

■ Versión RD

```
s: defs
defs: def defs |  $\lambda$ 
def: tipo ids ';'
tipo: INT
      | DOUBLE
ids: ID
     | ID ',' ids
```

■ Versión RI

```
s: defs
defs: defs def |  $\lambda$ 
def: tipo ids ';'
tipo: INT
      | DOUBLE
ids: ID
     | ids ',' ID
```

YACC

Análisis Sintáctico Ascendente

Yacc

Yet Another Compiler Compiler

- Herramienta para generar analizadores sintácticos para gramáticas LR(1)
 - Creada por *Stephen C. Johnson* en AT & T
 - Herramientas incorporadas en Unix
- Existen multitud de versiones
 - Usaremos byaccj
 - Berkeley Yacc para Java
- Qué ofrece la herramienta
 - Analiza Gramática
 - Genera Reconocedor
 - Permite ampliar a Parser

Determinar si una GLC es LR(1)

Fichero *ejemplo.y*

```
%token GATO PERRO
```

Declarar
tokens así...

```
%%
```

```
// Comentarios como en C++
```

```
oracion: sujeto verbo objeto '.';
```

```
sujeto: "TODO" nombre  
      | 'EL' nombre  
      | "PEPE"  
      ;
```

... o así...

... o así.

```
nombre: GATO | PERRO;
```

```
verbo: 'MUERDE' | "TREPA" 'A';
```

Reglas
acabadas en
punto y coma

```
objeto: 'LA' 'CORTINA'  
       | 'UN' 'ARBOL'  
       ;
```

No se pone λ

```
%%
```

Análisis

```
c:\>yacc ejemplo.y
```

- Si la gramática es LR(1) no imprime nada
- Si no lo es, indicará los conflictos
 - ❑ Shift/Reduce
 - ❑ Reduce/Reduce

Tres secciones

Generar código del Analizador

```
c:\>yacc -J ejemplo.y
```

```
%token GATO PERRO

%%
oracion: sujeto verbo objeto '.';
sujeto: "TODO" ...
...
%%
private Yylex lex;
Parser(Yylex lex, boolean debug) {
    this(debug);
    this.lex = lex;
}

int actual;
int yylex() {
    try {
        actual = lex.yylex();
        return actual;
    } catch (Exception e) {
        return -1;
    }
}

void yyerror(String s) {
    Sop("En "+lex.line()+":"+lex.column()
        + "Token " + actual
        + "lexema " + lex.lexeme());
}
```

```
class Parser {
    public short GATO = 257;
    public short PERRO = 258;
    public short TODO = 259;
    ...

    void yyparse() { // Analizador
        Mientras no símbolo inicial ni error {
            Mirar token actual (yylex)
            Según estado pila y token actual
            shift, reduce o error (yyerror)
        }
    }

    private Yylex lex;
    Parser(Yylex lex, boolean debug) {
        ...
        int yylex() { ... }
        void yyerror(String s) { ... }
    }
}
```

Las invoca pero no las genera

Invocación del Analizador

Añadir Léxico

```
...
%%
TODO      { return Parser.TODO; }
EL        { return Parser.EL; }
PEPE      { return Parser.PEPE; }
GATO      { return Parser.GATO; }
PERRO     { return Parser.PERRO; }
MUERDE    { return Parser.MUERDE; }
TREPA     { return Parser.TREPA; }
A         { return 'A'; }
LA        { return Parser.LA; }
CORTINA   { return Parser.CORTINA; }
UN        { return Parser.UN; }
ARBOL     { return Parser.ARBOL; }
\.        { return '.'; }

[ \n\r\t] { }
.         { Sop("Error " + yytext()); }
```

Ya no se
utiliza *Tokens*

Añadir Main

```
public class Main {
    public static void main(String[] args)
    {
        Yylex lex = new Yylex(System.in);
        Parser parser=new Parser(lex, false);

        if (parser.yyparse() == 0)
            Sop("Pertenece al lenguaje");
        else
            Sop("No pertenece al lenguaje");
    }
}
```

Se Compila y se obtiene Reconocedor

- Dada una entrada dirá si pertenece al lenguaje o no

EL GATO TREPA A LA CORTINA

TODO PERRO MUERDE

TODO PROFESOR MUERDE // No...



Soluciones

Solución E2 (I)

Versión RD

%%				int a, b; \$
s: defs;	shift	INT		int a, b; \$
defs: def defs	tipo → INT	tipo		a, b; \$
	shift	tipo ID		, b; \$
;				
def: tipo ids ';' ;	shift	tipo ID ,		b; \$
tipo: 'INT'	shift	tipo ID , ID		; \$
'DOUBLE' ;	ids → ID	tipo ID , ids		; \$
ids: 'ID'	ids → ID ',' ids	tipo ids		; \$
'ID' ',' ids;	shift	tipo ids ;		\$
%%				
	def → tipo ids ';' ;	def		\$
	defs → λ	def defs		\$
	defs → def defs	defs		\$
	s → defs	s		\$

Solución E2 (II)

Versión RI

%%			
s: defs;			int a, b; \$
	defs → λ	defs	int a, b; \$
defs: defs def	shift	defs INT	a, b; \$
;	tipo → INT	defs tipo	a, b; \$
def: tipo ids ';' ;	shift	defs tipo ID	, b; \$
tipo: 'INT'	ids → ID	defs tipo ids	, b; \$
'DOUBLE' ;	shift	defs tipos ids ,	b; \$
ids: 'ID'	shift	defs tipos ids , ID	; \$
ids ',' 'ID';			
%%	ids → ids ',' ID	defs tipos ids	; \$
	shift	defs tipos ids ;	\$
	def → tipo ids ';' ;	defs def	\$
	defs → defs def	defs	\$
	s → defs	s	\$