Lección 5: Análisis Sintáctico LR

1) Introducción
2) Un ejemplo "intuitivo"
3) Definiciones
4) Análisis SLR
5) Construcción de un analizador SLR
6) Sobre conflictos
7) Análisis LR Canónico
8) Análisis LALR
9) Gramáticas ambiguas en Yacc
10) Recuperación de errores sintácticos en el análisis ascendente

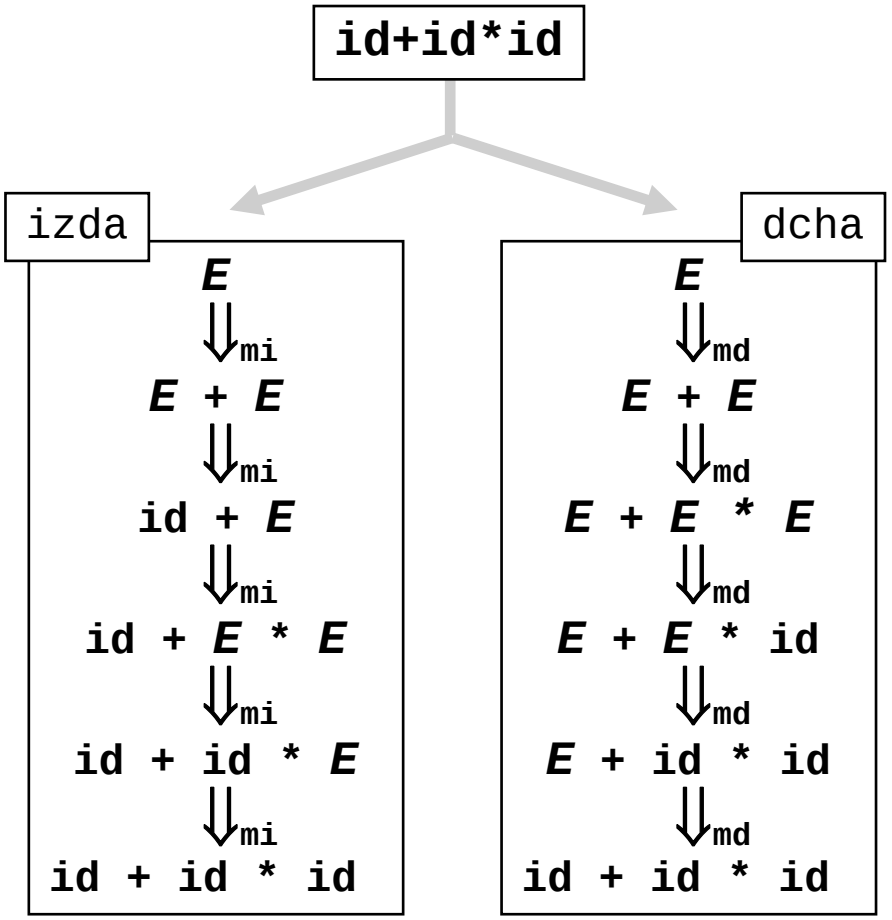
- Hemos visto algunas restricciones del AS descendente (top-down)
  - problemas con recursividad a izda.
  - no siempre son resolubles
- Una segunda forma de análisis

ascendente (bottom-up)

- Trata de construir el árbol de las hojas hacia la raíz
- Que coincide con encontrar una derivación por la derecha
- Técnica: ***desplazamiento-reducción***

$E \rightarrow E + E | E * E | (E) | -E | id$

- Recordar derivación a izda. y dcha.:
  - la primera siempre deriva el no terminal más a la izda. de la forma de frase
  - la segunda el de más a dcha.
- Recordar:



- ¿Cómo hacer derivación MD cuando los tokens llegan de izda. a dcha.?
- Idea básica:

en cuanto hayan entrado suficientes tokens para detectar una parte dcha., sustituírlos por su parte izda.

- Adecuado para ello: una pila
- Ejemplo: construir con ayuda de una pila el árbol sintáctico anterior

Un ejemplo

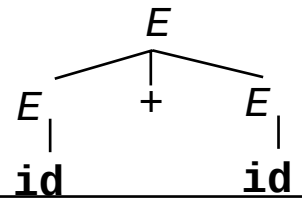
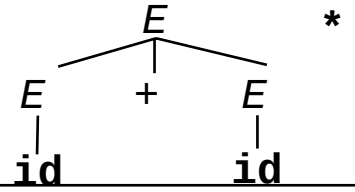
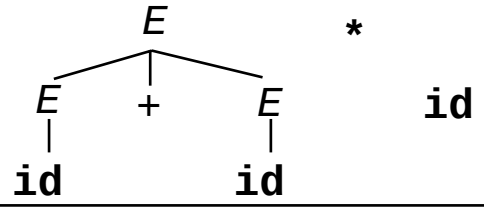
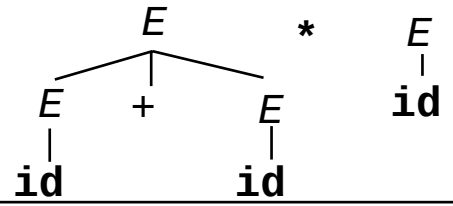
	1	2	3	4	5
$E \rightarrow$	$E + E$	$E * E$	(E)	$-E$	**id**

pila	**árbol**	**entrada**	
		id+id*id$	
id	**id**	**+id*id$**	**5**
E	*E* | **id**	**+id*id$**	
+ **E**	*E* **+** | **id**	**id*id$**	
id **+** **E**	*E* **+** | **id** **id**	***id$**	**5**
E **+** **E**	*E* **+** *E* | | **id** **id**	***id$**	**1**

Un ejemplo

1 2 3 4 5

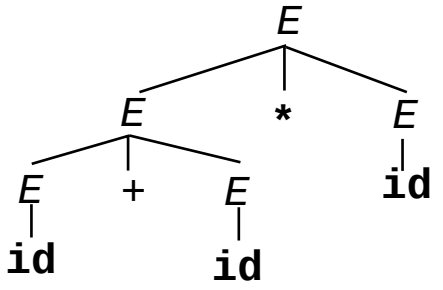
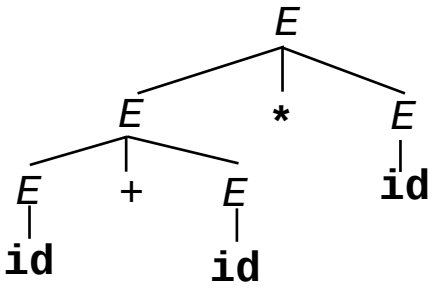
$E \rightarrow E + E | E * E | (E) | -E | \mathbf{id}$

pila	árbol	entrada	
E		***id$**	
*** E**		**id$**	
id * E		**$**	**5**
E * E		**$**	**2**

Un ejemplo

1 2 3 4 5

$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \mathbf{id}$

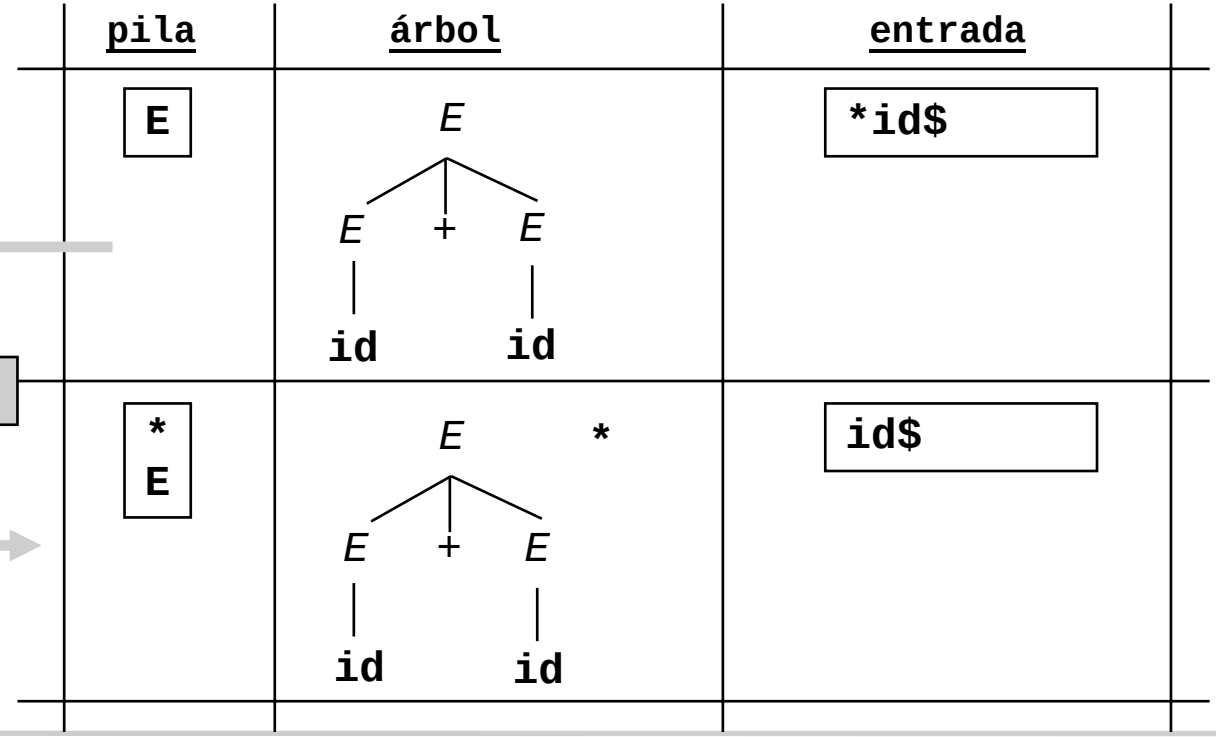
pila	**árbol**	**entrada**
E		**$**
		¡¡ EXITO !!

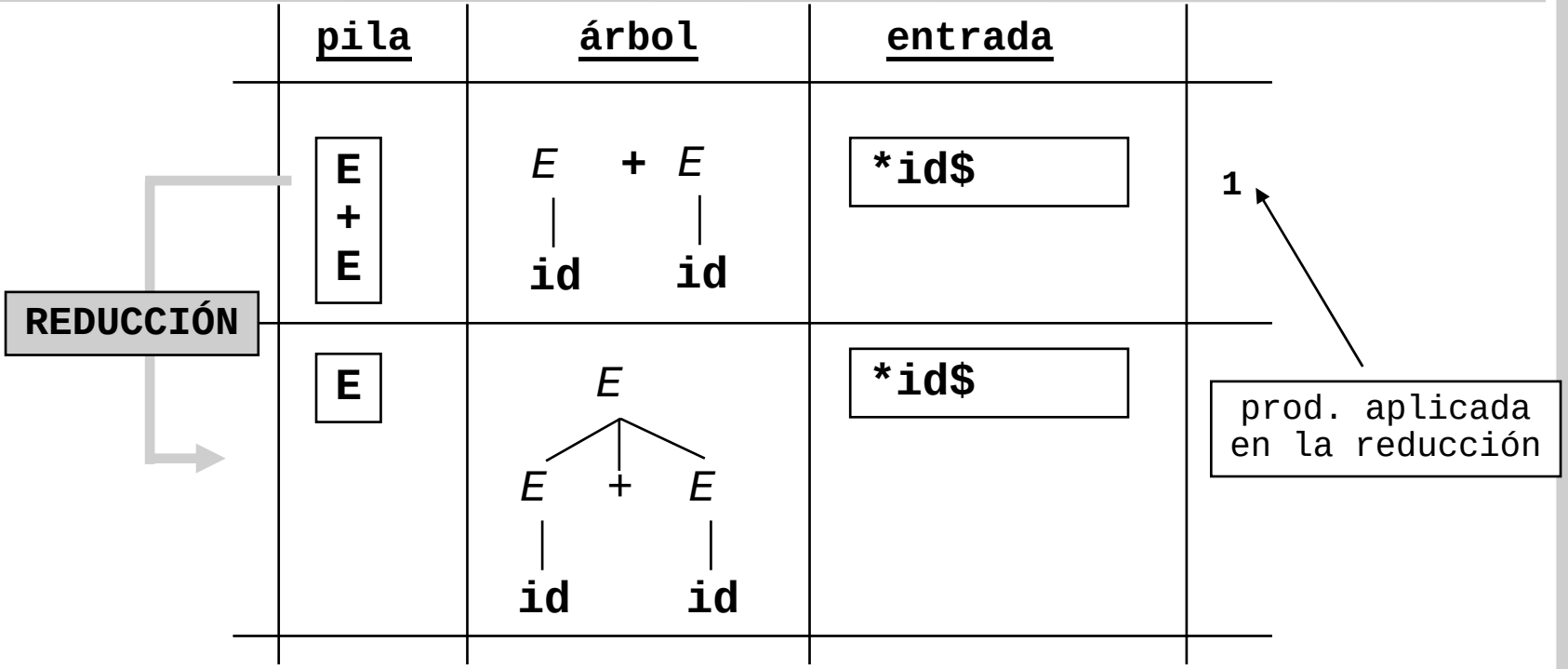
- Objetivo:

encontrar en la cima de la pila una parte derecha de producción y sustituírla por su parte izquierda

- Dos operaciones fundamentales:

DESPLAZAMIENTO





Analizadores por

DESPLAZAMIENTO y REDUCCION (D/R)

SHIFT-REDUCE parsers (S/R)

- Algunas definiciones:

desplazamiento

Operación de mover un token de la entrada a la pila

reducción

Operación de sustitución, en la parte superior de la pila, de la parte dcha. de una producción por su parte izda.

mango (asa/handle)

Parte dcha. de una producción que se encuentra en la parte superior de la pila

disparará una reducción

Aquélla para la que un analizador D/R, operando de izda. a dcha., puede reconocer los mangos que aparecen en la cima la pila

gramática LR

- Por lo tanto, una gramática será LR dependiendo de nuestra astucia para construir analizadores D/R

- Tres técnicas diferentes de construcción:
  - diferente complejidad de desarrollo
  - diferente tamaño del analizador
  - diferente potencia

analizador **SLR**

analizador **LR canónico**

analizador **LALR**

- Alternativamente:

gramática SLR (LR canónica)(LALR)

Aquélla para la que es posible construir un analizador sintáctico SLR (LR canónico)(LALR)

- Es decir:
  - sabremos que una gramática es SLR (LR canónica)(LALR)
  - si, con las técnicas que existen,

    podemos construir una analizador D/R SLR (LR canónico)(LALR)
- Estudiaremos el desarrollo de un analizador sintáctico SLR
  - más sencillo de comprender
  - técnicas para los demás "análogas"

Definiciones

prefijo viable

Prefijos de partes dcha. de producciones que pueden aparecer en la cima de la pila

- Expresión analizada parcialmente
- No debe sobrepasar por la derecha el mango
- No toda parte derecha es un mango:precedencia, asociatividad,..

- Notar que:

contenido de la pila + **resto de la entrada** = **forma de frase *md***

- Cuestiones fundamentales:
  - ¿Cómo reconocer cuándo un mango se encuentra en la pila?
  - ¿Cuántos símbolos de la pila forman el mango?
  - Si hay varias posibilidades, ¿qué producción usar en la reducción?
- Opciones:
  - a lo bestia: IMPRACTICABLE
  - "refinadamente": mediante un autómata
    - » complejo de obtener
    - » fácil de usar

- Presentamos un método que permite realizar el análisis sintáctico por desplazamiento/reducción
- Básicamente:
  - un autómata tal que cada estado guarda la información de los posibles prefijos recorridos hasta llegar a él
  - una pila de estados (guarda el prefijo recorrido)
  - a partir de los estados en la pila y del token de entrada se determinan las acciones a ejecutar (SLR(**1**))
- Pero: ¿cómo almacenar en un estado del autómata todos los símbolos gramaticales ya reconocidos pero aún no reducidos?
- Solución:

Conjuntos de "elementos" (items/configuraciones)

del análisis sintáctico LR(0)

elemento (configuración/item)

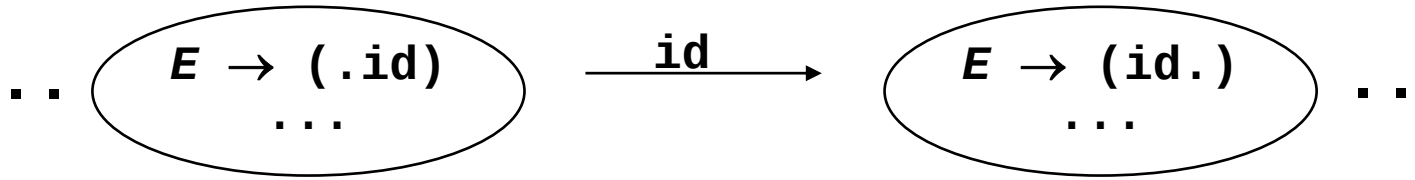
Una producción con un punto en alguna posición de su parte derecha

- Ejemplo: "**E → (id)**" puede generar las configuraciones

E → .(id)
E → (.id)
E → (id.)
E → (id).

- ¿Qué representa "***E* → (.id)**"? ¿***F* → aX.Ybc**?
  - que en algún momento, durante el análisis sintáctico, hemos llegado a un estado en el que hemos reconocido ya **(** y **aX**
  - y estamos esperando (¿deseando?) reconocer para poder aplicar una reducción **id)** y **Ybc**

- Así, si tenemos la suerte de que reconocemos "**id**" podremos cambiar de estado:



- ¿Cómo detectar el fin?
  - se amplía la gramática con un no terminal especial: **S'**
  - y una producción especial

$E \rightarrow E + E | E * E | (E) | -E | \mathbf{id}$

$S' \rightarrow E$

$E \rightarrow E + E | E * E | (E) | -E | \mathbf{id}$

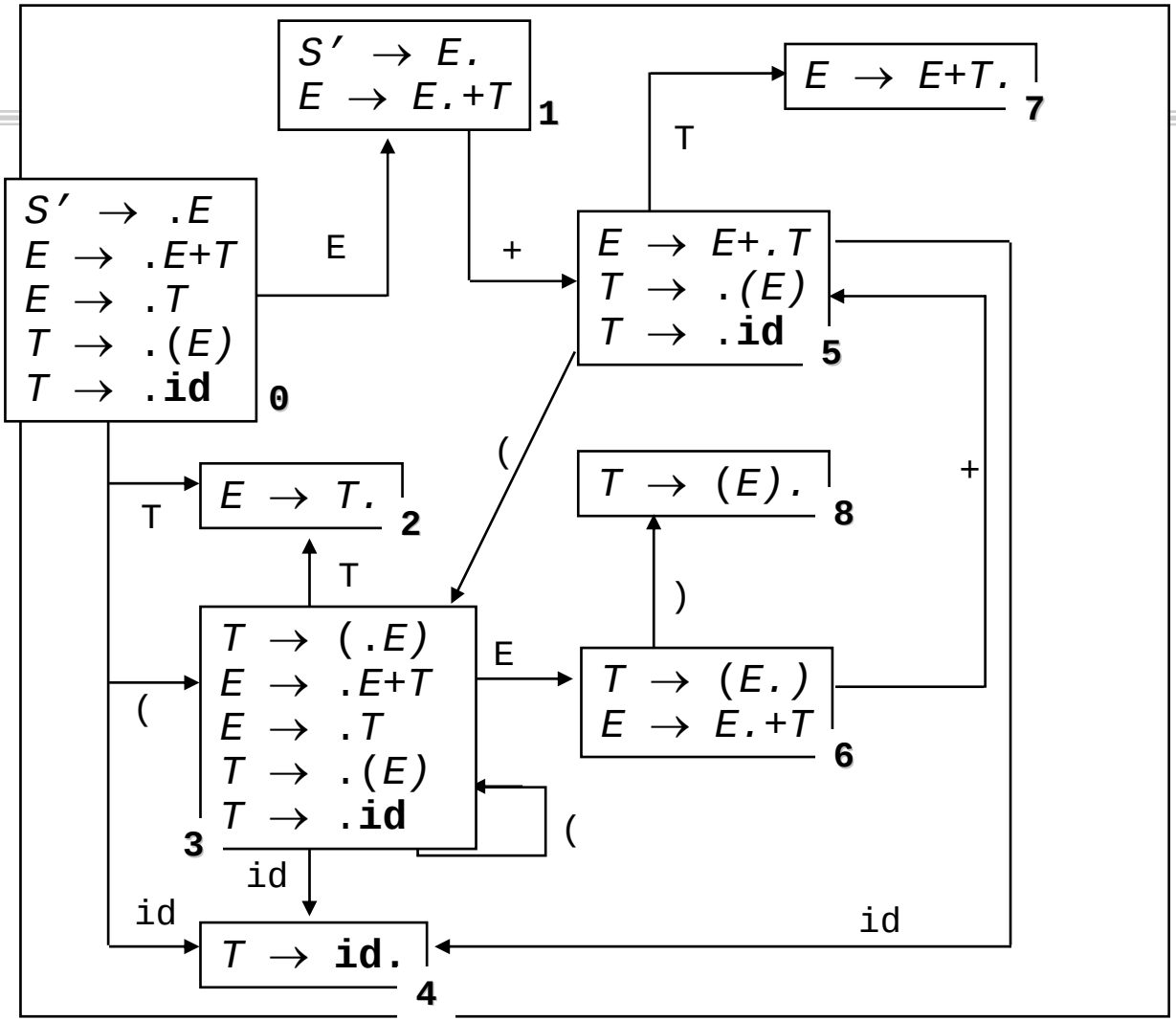
- Así, el reconocimiento habrá funcionado cuando se llegue a un estado que contenga la configuración

$$S' \rightarrow E.$$

- Vamos a ver un pequeño ejemplo de cómo funciona un reconocedor a partir del autómata
- Ya veremos cómo construir dicho autómata

$$E \rightarrow E+T$$
$$E \rightarrow T$$
$$T \rightarrow (E)$$
$$T \rightarrow \mathbf{id}$$

$$S' \rightarrow E$$
$$E \rightarrow E+T$$
$$E \rightarrow T$$
$$T \rightarrow (E)$$
$$T \rightarrow \mathbf{id}$$


S' → E.
E → E.+T
1
E → E+T.
7
T
S' → .E
E → .E+T
E → .T
T → .(E)
T → .id
0
E
+
E → E+.T
T → .(E)
T → .id
5
(
T → (E).
8
+
E → T.
2
T
T
)
T → (.E)
E → .E+T
E → .T
T → .(E)
T → .id
3
E
T → (E.)
E → E.+T
6
(
(
id
id
id
T → id.
4

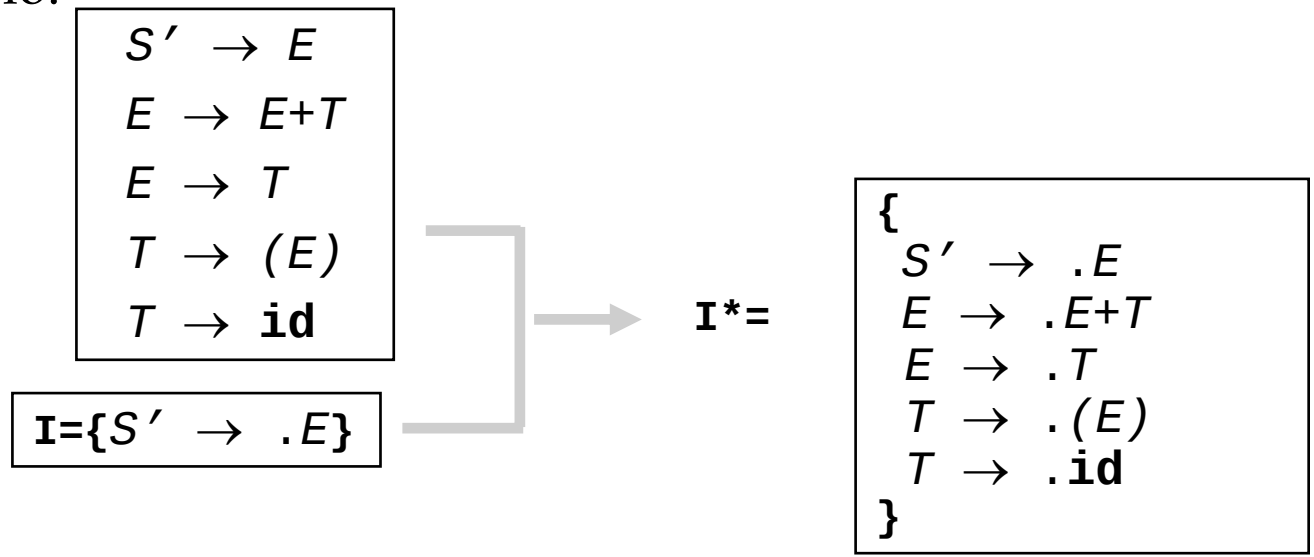
- "Pasear" por el autómata anterior para reconocer

```
(id+id)
(id+(id+id))
(id++id)
```

- La cuestión es cómo obtener el autómata anterior
- Basado en las funciones:
  - clausura (de un conj. de configuraciones)
  - sucessor (de un conj. de configuraciones)

```
Función clausura(E I:conj config.) dev (I*:conj. config.)
Pre: I  es un conj. de configuraciones pa-
    ra una gramática ampliada
    G'=(N',T,S',P')
Post: I* contiene la clausura LR(0) de I
Principio
  I*:=I
  Repetir
     Para cada A→α.Bβ ∈I*
        Para cada B→γ∈P'
          Si B→.γ ∉ I*
            entonces I*=I*∪{B→.γ }
          FSi
        FPara
     FPara
  Hasta Que no se añada nada a I*
  dev(I*)
Fin
```

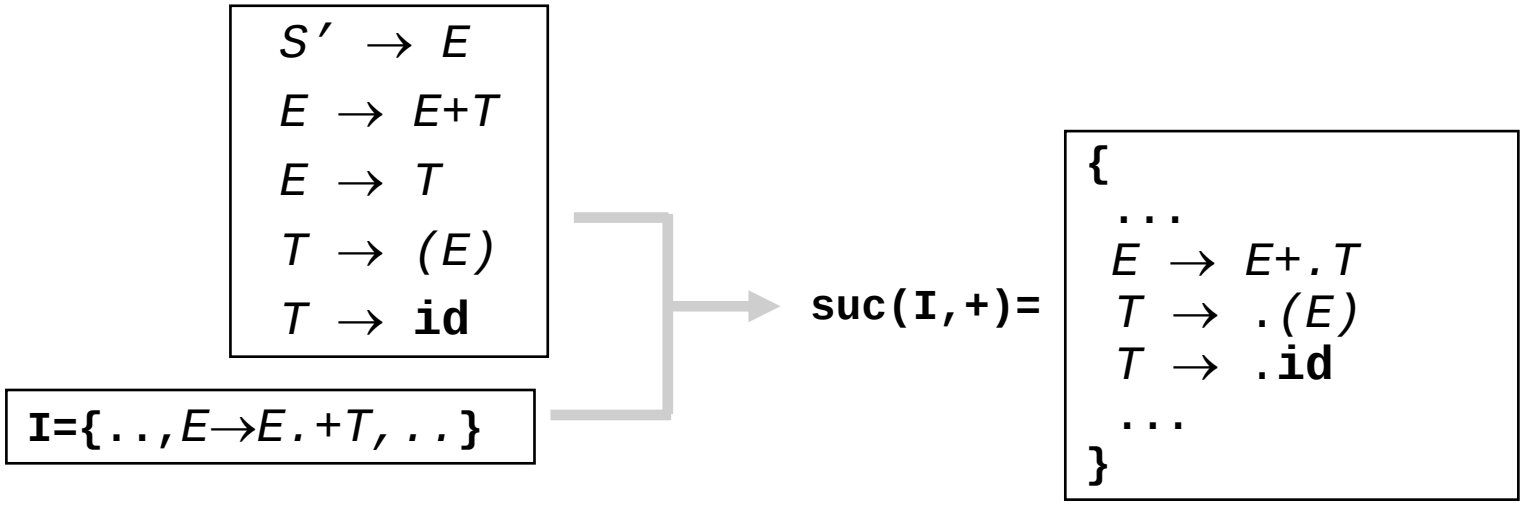
- Idea subyacente: Si en un estado tenemos la config. "A→α.Bβ", quiere decir que esperamos cualquier derivación de "Bβ", por lo que también nos veremos "contentos" con cualquier derivación de "B"
- Ejemplo:



```
Función sucesor(E I:conj config.;E X:símbolo)
                        dev (SS:conj. config.)
Pre: I  es un conj. de configuraciones para una
     gramática ampliada G'=(N',T,S',P')
     X∈T∪N
Post: S contiene los sucesores de I
      respecto del símbolo X
Variables S:conj. config.
Principio
  S:=∅
  Para cada (A→α.Xβ) ∈I
     S=S∪{A→αX.β}
  FPara
  dev(clausura(S))
Fin
```

- Idea subyacente: Si en un estado tenemos la config. "$A \rightarrow \alpha.X\beta$", y "viene" el símbolo X, pasamos a cualquier configuración "deseable/esperable" de "$A \rightarrow \alpha X.\beta$"

- Ejemplo:



- Ahora, agrupamos los conjuntos de configuraciones que representan estados del AFD que reconoce prefijos viables
- Construimos el conjunto C:
  - cada elemento de C es un conjunto de configuraciones

colección canónica de conjuntos de configuraciones LR(0)

- Proceso:
  - añadir a C el conjunto **clausura({S' → .S })**

simb. inicial original

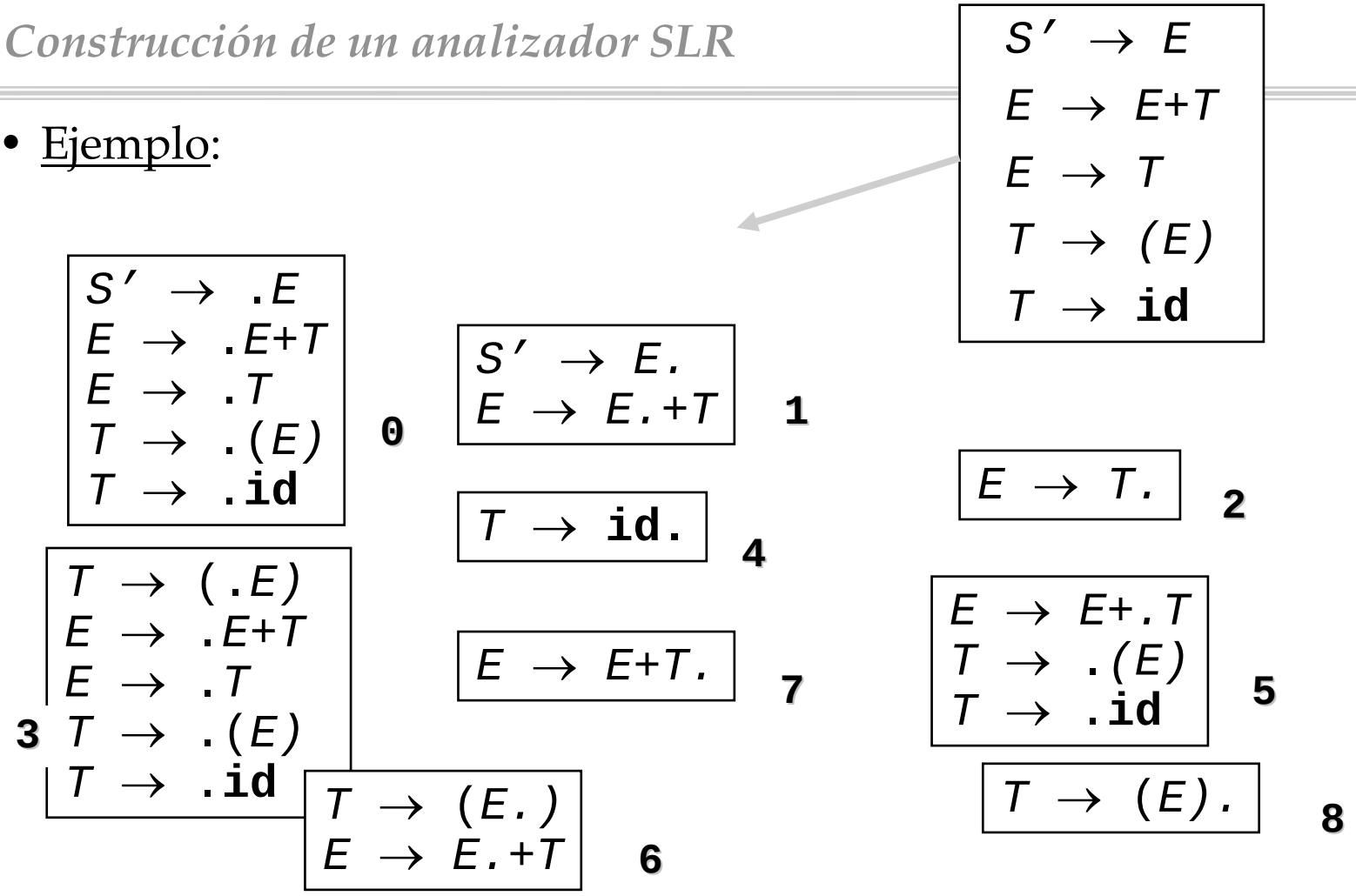
simb. inicial "amplíado"

  - añadir, para cada `I` de `C` y para cada símbolo `X`, los sucesores correspondientes
- Resultado: nodos del autómata

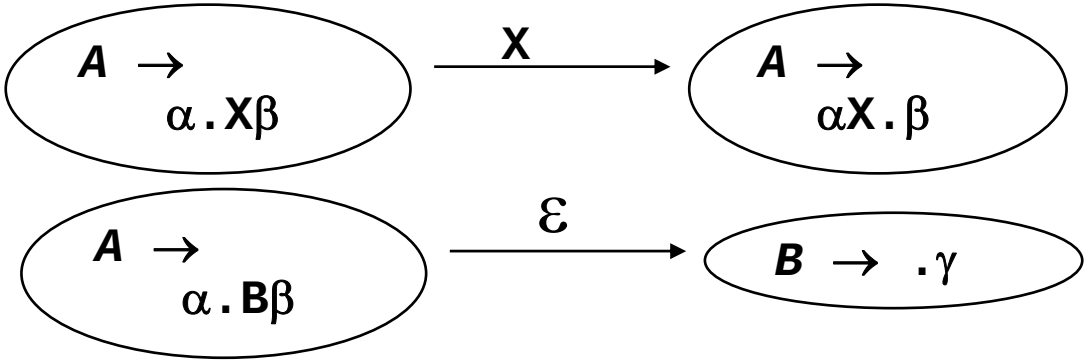
- Cálculo de la colección de canónica de conjuntos de configuraciones LR(0) para una gramática

```
Función conjLR_0(E G':gramática) dev
                (C:conj. conj. config.)
Pre: G'=(N',T,S',P')es una gramática
     ampliada
Post: C es la colección canónica de
     configuraciones LR(0) C={I0,...,In}
Variables S:conj. config.
Principio
  C:={clausura({S'→.S})}  /*nodo I0*/
  Repetir
     Para cada I∈C
        Para cada X∈N∪T
           Si (suc(I,X)<>∅)∧(suc(I,X)∉C)
             entonces C=C ∪ suc(I,X)
           FSi
        FPara
     FPara
  Hasta que no se añada nada a C
  dev(C)
Fin
```

- Ejemplo:

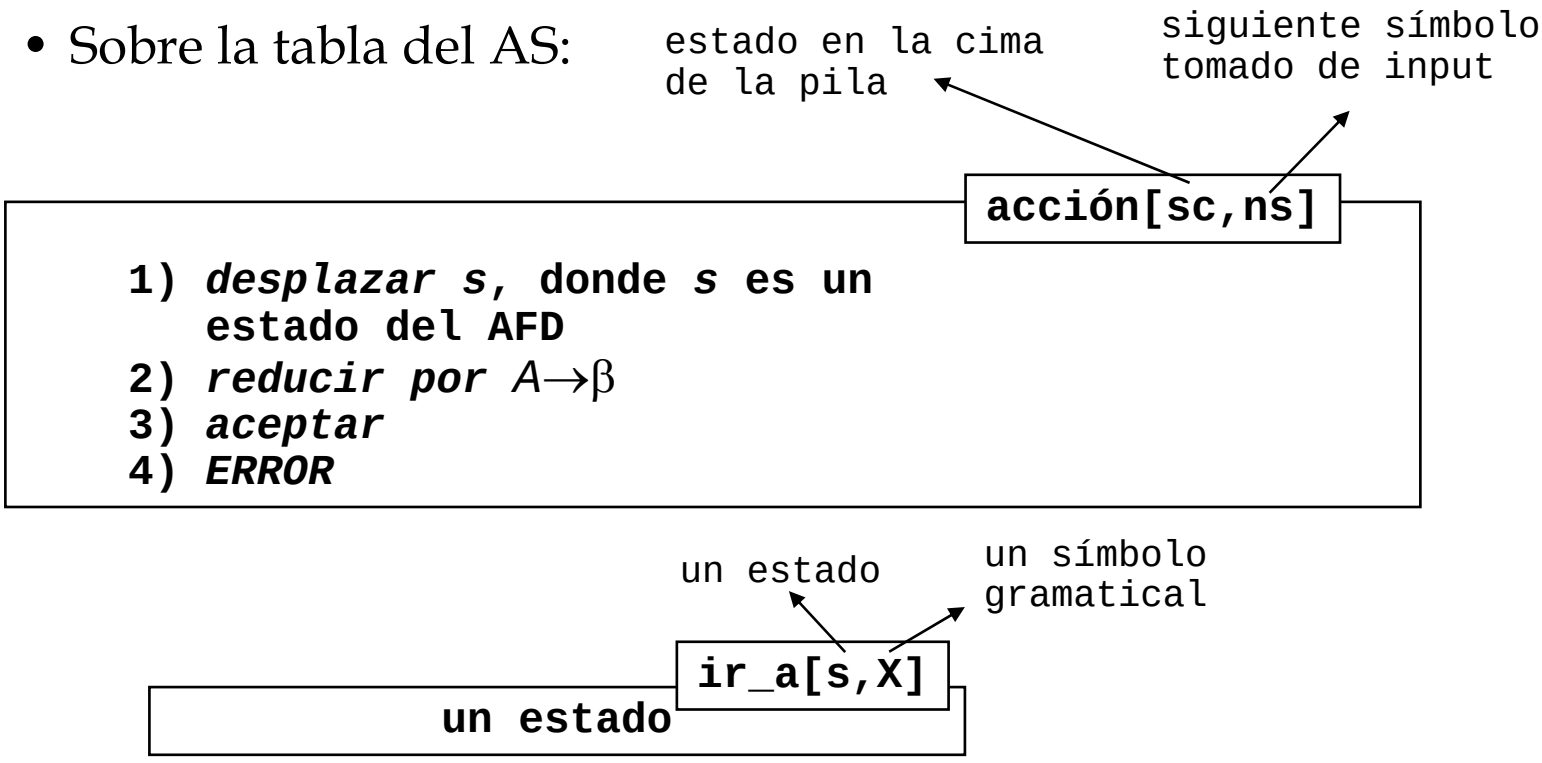

S' → E
E → E+T
E → T
T → (E)
T → id
S' → .E
E → .E+T
E → .T
T → .(E)
T → .id
0
S' → E.
E → E.+T
1
T → id.
4
E → T.
2
T → (.E)
E → .E+T
E → .T
3 T → .(E)
T → .id
E → E+T.
7
E → E+.T
T → .(E)
T → .id
5
T → (E.)
E → E.+T
6
T → (E).
8

- Ya podemos construir el AFD
  - **C** es el conjunto de nodos
  - $\mathbf{I_0}$**= clausura({*S'* → .*S*})** es el estado inicial
  - La función **sucesor()** establece exactamente los arcos del AFD
- ¿Por qué funciona?
  - tomar un AFN cuyos estados son las distintas configuraciones
  - transiciones:



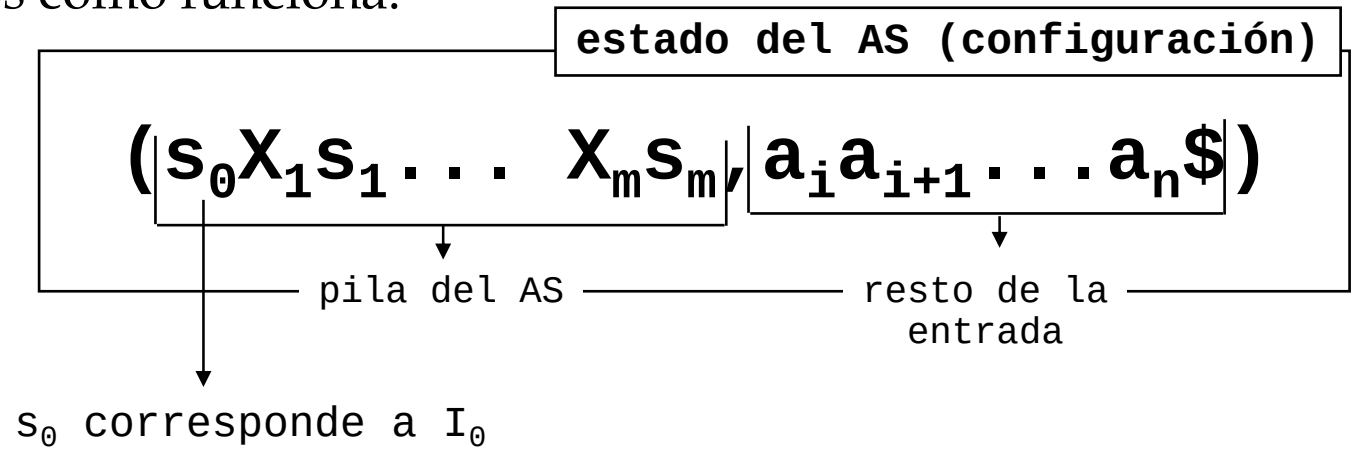
  - Transformar en AFD mediante la ε-clausura
    - » aquí se convierte en usar la nueva clausura para conjuntos
  - El resultado es el grafo de conjuntos que hemos visto
- En lo que sigue nos queda implementar algorítmicamente el método de "recorrido" del autómata que hemos presentado
- Se basa en
  - una pila para el análisis sintáctico (como vimos al principio)
  - la tabla del análisis sintáctico
    - » parte 1: tabla "**`acción`**"
    - » parte 2: tabla "**`ir_a`**"
  - parte de control

- Sobre la tabla del AS:



¡La función de transición en el autómata que reconoce los prefijos viables!

- Veamos cómo funciona:



- Donde:

$s_{i-1}X_is_i$ paso del estado s_{i-1} al s_i mediante el símbolo Xi

$X_1 \ldots X_m\ a_i \ldots a_n$ forma de frase derecha

- El siguiente movimiento del analizador se establece
  - en función a s_m y a_i
  - en función a **acción**$[s_m, a_i]$
  - en función a **ir_a[?,?]**

1) Si **acción**$[s_m, a_i]$=*desplazar s*
  - acción de desplazamiento, pasando a la siguiente configuración del AS

$$(s_0 X_1 s_1 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$)$$

$$(s_0 X_1 s_1 \ldots X_m s_m a_i s, a_{i+1} \ldots a_n \$)$$

2) Si **acción[s_m, a_i]=*reducir* $A \rightarrow \beta$**

- acción de reducción, pasando a la siguiente configuración del AS

$$(s_0 X_1 s_1 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$)$$

$$(s_0 X_1 s_1 \ldots X_{m-r} s_{m-r} A\ s, a_i a_{i+1} \ldots a_n \$)$$

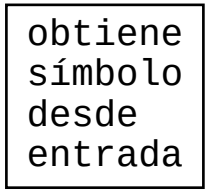
- Donde:
  - » $r=|\beta|$ $\quad s=ir\_a(s_{m-r}, A)$
- no se modifica la entrada
- nº elementos en la pila no aumenta
- sustitución de una derivación por un no terminal
  - » se "ha reducido" un árbol por su raíz
- Normalmente se informa de la producción aplicada para la reducción

3) Si **acción[s_m, a_i]=*aceptar***
   - AS terminado con éxito: la entrada es sintácticamente correcta para la gramática considerada

4) Si **acción[s_m, a_i]=*ERROR***
   - no es posible terminar el AS
   - invocar a rutinas de tratamiento de errores

- Normalmente:
  - la pila no almacena los símbolo X_i
  - aquí los ponemos por motivos de claridad
- Veamos el algoritmo de análisis SLR a partir de **acción[]** e **ir_a[]**

Construcción de un analizador SLR




```
Algoritmo LR(E acción[]:...; E ir_a[]:...)
Variables n:simbolo;P:pila
          s,s':estado
          aceptado,error:booleano
Principio
  pilaVacia(P);apilar(P,s0)
  <aceptado,error>:=<False,False>
  n:=yylex()
  Repetir
     s:=cima(P)
     Sel
       accion[s,n]=desplazar s':
           apilar(P,n);apilar(P,s')
           n:=yylex()
       accion[s,n]=aceptar:
           aceptado:=True
       accion[s,n]=ERROR:
           error:=True
       .....
     FSel
  Hasta que aceptado ∨ error
Fin
```

Construcción de un analizador SLR

```
Algoritmo LR(E acción[]:...; E ir_a[]:...)
Variables n:simbolo;P:pila
          s,s':estado
          aceptado,error:booleano
Principio
  pilaVacia(P);apilar(P,s0)
  <aceptado,error>:=<False,False>
  n:=yylex()
  Repetir
     s:=cima(P)
     Sel
       ....
       accion[s,n]=reducir A→β:
           Para i:=1 hasta 2*|β|
              desapilar(P)
           FPara
           s:=cima(P)
           apilar(P,A)
           apilar(P,ir_a[s,A])
           emitir "A→β"
     FSel
  Hasta que aceptado ∨ error
Fin
```

Construcción de un analizador SLR

1) $S' \rightarrow E$
2) $E \rightarrow E+T$
3) $E \rightarrow T$
4) $T \rightarrow (E)$
5) $T \rightarrow$ **id**

estados ***acción[]*** ***ir_a[]***

	+	(	)	id	$	E	T
0		d3		d4		1	2
1	d5				acep		
2	r3		r3		r3		
3		d3		d4		6	2
4	r5		r5		r5		
5		d3		d4			7
6	d5		d8				
7	r2		r2		r2		
8	r4		r4		r4		

Construcción de un analizador SLR

```
Alg constAccion(E G':gramática) dev
                (accion[]:....)
Pre: G'=(N',T,S',P')es una gramática
     amplíada
Post: accion es parte de la tabla del SLR
Variables I:conj. config.; i:entero
          C:conj. conj. config.
Principio
  C:=conjLR_0(G')  /*C={I0,...,In}*/
  Para i:=0 hasta n
     Para cada a∈T
       Si A→α.aβ∈ Ii ∧ suc(Ii,a)=Ij
          accion[i,a]:="desplazar j"
       FSi
     FPara
     ....
  FPara
  asignar "ERROR" a entradas vacías
  dev(accion)
Fin
```

Construcción de un analizador SLR

```
Alg constAccion(E G':gramática) dev
                (accion[]:....)
Pre: G'=(N',T,S',P')es una gramática
     amplíada
Post: accion es parte de la tabla del SLR
Variables I:conj. config.; i:entero
          C:conj. conj. config.
Principio
  ....
     Para cada A∈N'\S'
       Si A→α.∈ I_i
          Para cada a∈SIG(A)
            accion[i,a]:="reducir A→α"
          FPara
       FSi
     FPara
     Si S'→S. ∈ I_i
          accion[i,$]:="aceptar"
     FSi
  FPara
  asignar "ERROR" a entradas vacías
  dev(accion)
Fin
```

Construcción de un analizador SLR

Estado inicial del analizador:
el construído a partir de

"S'→.S"

```
Alg constIr_a(E G':gramática) dev
                    (ir_a[]:....)
Pre: G'=(N',T,S',P')es una gramática
     amplíada
     C={I0,...,In}= conjLR_0(G')
Post: ir_a es parte de la tabla del SLR
Variables i:entero
Principio
  Para i:=0 hasta n
     Para cada A∈N'
       Si suc(Ii,A)=Ij
          ir_a[i,A]:=j
       FSi
     FPara
  FPara
  asignar "ERROR" a entradas vacías
  dev(ir_a)
Fin
```

- Importante:
  - si en la ejecución del algoritmo se llega a alguna ***contradicción***
    - » el método no es aplicable para la gramática considerada
    - » no es una gramática SLR
  - si la ejecución genera una tabla
    - » se trata de una gramática SLR
- Las tablas anteriores establecen el método SLR
- ¿Qué significa "contradicción"?
  - Ordenes contradictorias
- Veamos algunos ejemplos

- Construir el autómata y las tablas del análisis SLR(1) para las siguientes gramáticas:

$S \rightarrow (\; L \;)$
$S \rightarrow \mathbf{a}$
$L \rightarrow L \; S$
$L \rightarrow \mathbf{S}$

$S \rightarrow \mathbf{a} \; R$
$S \rightarrow \mathbf{a}$
$R \rightarrow \mathbf{b}$
$R \rightarrow \varepsilon$

$S \rightarrow S \; \mathbf{a}$
$S \rightarrow \mathbf{a} \; R$
$S \rightarrow \mathbf{a}$
$R \rightarrow \mathbf{a} \; \mathbf{b}$

- ¿Son las siguientes gramáticas SLR(1)? En caso afirmativo, el funcionamiento de ambas durante en análisis, y concluir sobre cuál de las dos formas es más conveniente.

$lID \rightarrow \mathbf{id} \; lID$
$lID \rightarrow \mathbf{id}$

$lID \rightarrow lID \; \mathbf{id}$
$lID \rightarrow \mathbf{id}$

- **Ejercicio** (examen Febrero-96)

Considérese la siguiente gramática:

```
ALS →  ALS  AL
ALS →  AL
AL  →  NUM  NS  ',' NS  '\n'
NS  →  NS N
NS  →  N
```

Se desea saber si se trata de una gramática SLR o no. Para ello, responder a las siguientes preguntas.

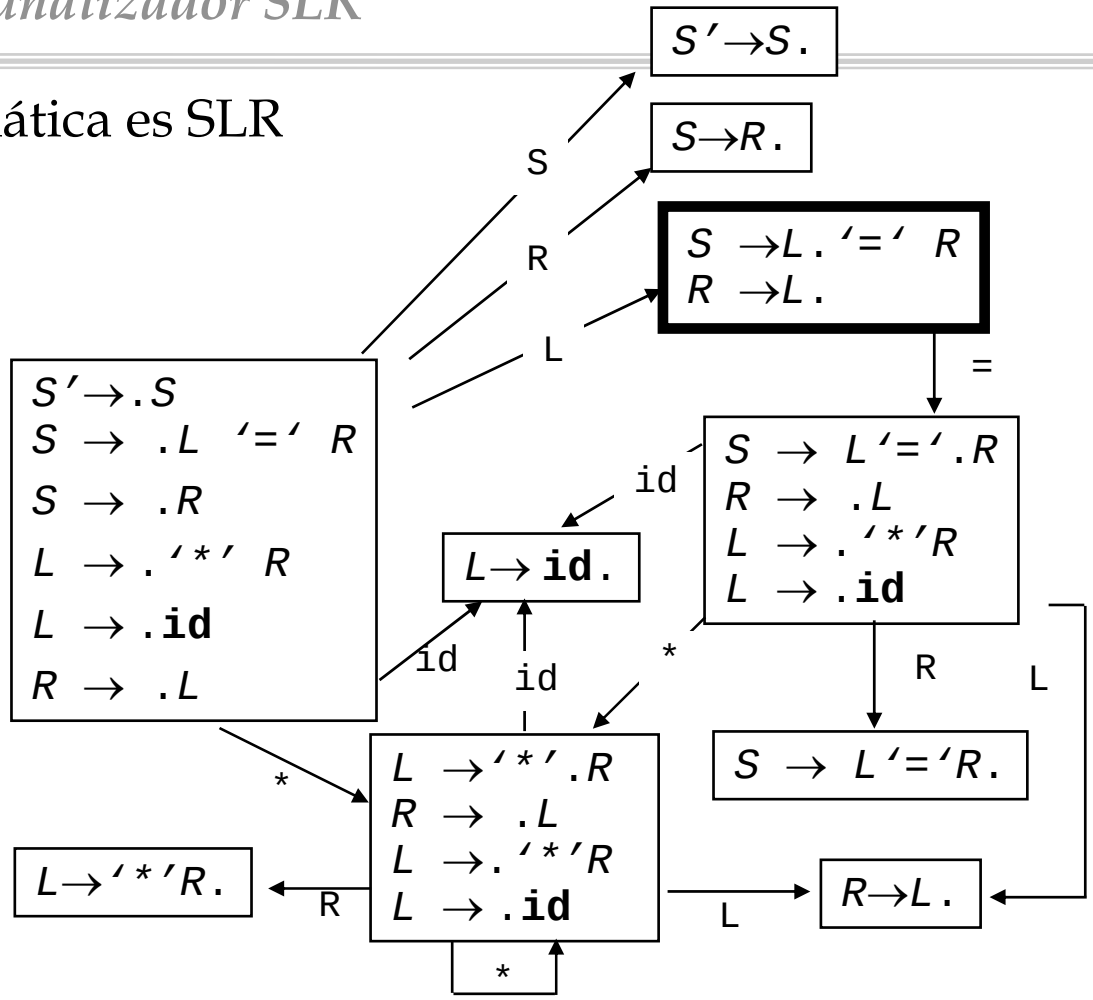
Ejercicio 4 (2 ptos.): Construir la familia canónica de conjuntos LR(0)

Ejercicio 5 (1.5 ptos.): Construir el autómata a partir de dichos conjuntos

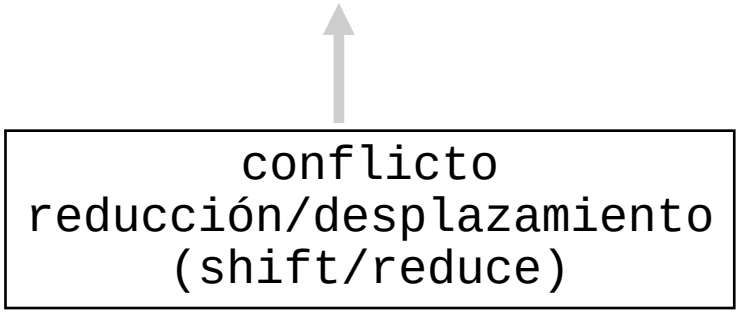
Ejercicio 6 (1.5 ptos.): ¿Se trata de una gramática SLR o no?

- Pero no toda gramática es SLR

$S' \rightarrow S$
$S \rightarrow L\ \text{`='}\ R \mid R$
$L \rightarrow \text{`*'}\ R \mid \mathbf{id}$
$R \rightarrow L$



- Si observamos el estado destacado tenemos:
  - S $\rightarrow$ L . '=' R
    accion[2,'=']="desplazar 6"
  - '=' $\in$ SIG(R)
    accion[2,'=']="reducir R $\rightarrow$ L"



Sobre conflictos

- Ya comentamos que pueden aparecer dos tipos de conflictos
  - shift/reduce
  - reduce/reduce
- Ambos corresponden a decisiones que ha de tomar el analizador:
  - **shift/reduce**: se puede tanto realizar un desplazamiento como una reducción
  - **reduce/reduce**: el analizador detecta en la cima de la pila partes derechas de más de una producción
- Habitualmente, se aplican políticas preestablecidas para su resolución:
  - **shift/reduce**: realiza el desplazamiento
  - **reduce/reduce**: reduce por la producción que se haya escrito antes

```
Yacc las hace
Notar que son análogas a las que hacía Lex
```

- Vamos a ver un método alternativo de análisis bottom-up: LR canónico
- El proceso es parecido al desarrollado para SLR
- Se basa en la noción de elemento (configuración) para el análisis LR(1)

configuración del análisis LR(1)

Es de la forma

$$A \rightarrow X_1 \ldots X_i \cdot X_{i+1} \ldots X_n, a$$

donde

1) $A \rightarrow X_1 \ldots X_i X_{i+1} \ldots X_n$ **es una producción**
2) $a \in V_t \cup \{\$\}$

- ¿Qué representa una de estas configuraciones?

$$A \rightarrow X_1 \ldots X_i \,.\, X_{i+1} \ldots X_n, \underline{a}$$

símbolo de anticipación (pre-análisis)

hemos reconocido "$X_1 \ldots X_i$"

- esperamos/deseamos reconocer "$X_{i+1} \ldots X_n$"
- cuando hayamos reconocido "$X_1 \ldots X_n$", nos podemos encontrar con el terminal "**a**"
- en un config. "$A \rightarrow \alpha \,.\, \beta, a$"
  - » si $\beta \not\Rightarrow_* \varepsilon$, "**a**" no genera ningún efecto
- una config. "$A \rightarrow \alpha \,.\, , a$" o "$A \rightarrow \alpha \,.\, \beta, a$" con $\beta \Rightarrow_* \varepsilon$, pide una reducción por "$A \rightarrow \alpha$" " <u>sólo</u> si el símbolo de preanálisis es "**a**"

- Dicho de otra forma:

Reducir por "A → α" sólo con aquellos símbolos "a" de la entrada para los que "A → α.,a" es un elemento LR(1) del estado en la cima de la pila

- Diferencia importante con SLR:
  - en SLR, la reducción se hacía siempre (es decir, para todo elemento de **SIG(A)**)
  - en LR(1), puede que la reducción sólo sea para un subconjunto propio de **SIG(A)**
- Notación compacta

$$A \rightarrow X_1 \ldots X_i \cdot X_{i+1} \ldots X_n, a_1$$
$$\ldots\ldots\ldots$$
$$A \rightarrow X_1 \ldots X_i \cdot X_{i+1} \ldots X_n, a_k$$

$$A \rightarrow X_1 \ldots X_i \cdot X_{i+1} \ldots X_n, \{a_1, \ldots, a_k\}$$

- El uso del símbolo de anticipación:
  - da a LR(1) más potencia que LR(0):
    - » parsers LR(1) son los más potentes bottom-up parsers con un único símbolo de pre-análisis
  - aumenta el tamaño de AFD:
    - » hay varias conf. LR(1) por cada posible conf. LR(0)
    - » el límite es $|V_t|$
- El proceso de construcción:
  - análogo al de LR(0), empezando con

$$\mathbf{S' \rightarrow .S, \{\$\}}$$

  - como S puede ser un no terminal, es necesaria una operación de clausura

```
Función clausura1(E I:conj config.) dev
                                (I*:conj. config.)
Pre: I  es un conj. de configuraciones pa-
    ra una gramática ampliada
    G'=(N',T,S',P')
Post: I* contiene la clausura LR(1) de I
Principio
  I*:=I
  Repetir
    Para cada "A→α.Bβ,a" ∈I*
      Para cada B→γ∈P
        Para cada b∈T∩ PRI(βa)
          Si "B→.γ ,b" ∉ I*
            entonces I*=I*∪{"B→.γ ,b" }
          FSi
        FPara
    FPara
  Hasta Que no se añada nada a I*
  dev(I*)
Fin
```

Idea subyacente

Análoga a la vista para la clausura LR(0), sólo que teniendo en cuenta el símbolo de pre-análisis

- Ejemplo:

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow \mathbf{c}C$
$C \rightarrow \mathbf{d}$

I={$S' \rightarrow .S, \{\$\}$}

I*=

{
$S' \rightarrow .S,$ {$}
$S \rightarrow .CC,$ {$}
$C \rightarrow .\mathbf{c}C,$ {c,d}
$C \rightarrow .\mathbf{d}\ ,$ {c,d}
}

```
Función sucesor1(E I:conj config.;E X:símbolo)
                        dev (SS:conj. config.)
Pre: I  es un conj. de configuraciones para una
     gramática ampliada G'=(N',T,S',P');X∈T∪N
Post: SS contiene los sucesores de I respecto de X
Variables S:conj. config.
Principio
  S:=∅
  Para cada "A→α.Xβ,a" ∈I
     S=S∪{A→αX.β,a}
  FPara
  dev(clausura1(S))
Fin
```

Idea subyacente

Si en un estado tenemos la config. "A→α.Xβ,a", y "viene" el símbolo X, pasamos a cualquier configuración "deseable/esperable" "A→αX.β,a"

- Colección de canónica de conjuntos de configura-ciones LR(1) para una gramática

```
Función conjLR_1(E G':gramática) dev
                (C:conj. conj. config.)
Pre: G'=(N',T,S',P'), una gramática ampliada
Post: C es la colección canónica de
     configuraciones LR(1)
Variables S:conj. config.
Principio
  C:=clausura1({"S'→.S,$"})
  Repetir
     Para cada I∈C
        Para cada X∈N∪T
           Si (suc1(I,X)<>∅)∧(suc1(I,X)∉C)
              entonces C=C∪ suc1(I,X)
           FSi
        FPara
     FPara
  Hasta que no se añada nada a C
  dev(C)
Fin
```

- Ejemplo:

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow \mathbf{c}C$
$C \rightarrow \mathbf{d}$

$S' \rightarrow .S, \{\$\}$
$S \rightarrow .CC, \{\$\}$
$C \rightarrow .\mathbf{c}C, \{\mathbf{c},\mathbf{d}\}$
$C \rightarrow .\mathbf{d}, \{\mathbf{c},\mathbf{d}\}$ **0**

$S \rightarrow C.C, \{\$\}$
$C \rightarrow .\mathbf{c}C, \{\$\}$
$C \rightarrow .\mathbf{d}, \{\$\}$ **2**

$C \rightarrow \mathbf{d}., \{\mathbf{c},\mathbf{d}\}$ **4**

$S' \rightarrow S., \{\$\}$ **1**

$C \rightarrow \mathbf{c}.C, \{\$\}$
$C \rightarrow .\mathbf{c}C, \{\$\}$
$C \rightarrow .\mathbf{d}, \{\$\}$ **6**

$C \rightarrow \mathbf{c}.C, \{\mathbf{c},\mathbf{d}\}$
$C \rightarrow .\mathbf{c}C, \{\mathbf{c},\mathbf{d}\}$
$C \rightarrow .\mathbf{d}, \{\mathbf{c},\mathbf{d}\}$ **3**

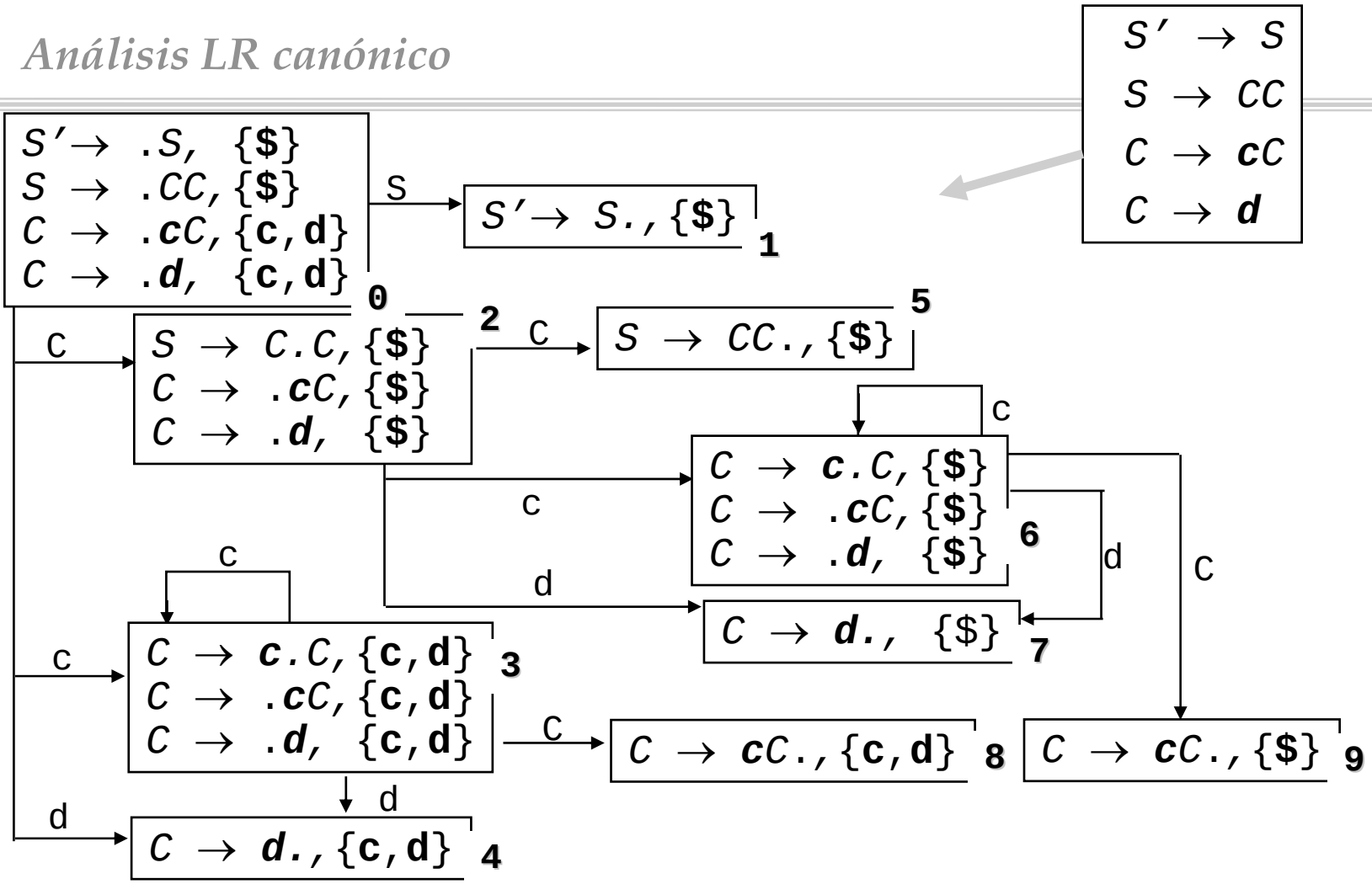
$C \rightarrow \mathbf{d}., \{\$\}$ **7**

$C \rightarrow \mathbf{c}C., \{\mathbf{c},\mathbf{d}\}$ **8**

$S \rightarrow CC., \{\$\}$ **5**

$C \rightarrow \mathbf{c}C., \{\$\}$ **9**

Análisis LR canónico



```
Función constAccion(E G':gramática) dev (accion[]:....)
Pre: G'=(N',T,S',P')es una gramática ampliada
Post: accion es parte de la tabla LR(1)
Variables I:conj. config.; i:entero;C:conj. conj. config.
Principio
  C:=conjLR_1(G')  /*C={I0,...,In}*/
  Para i:=0 hasta n
     Para cada b∈T
       Si "A→α.bβ,a"∈ Ii ∧ suc1(Ii,b)=Ij
          accion[i,b]:="desplazar j"
       FSi
     FPara
     Para cada A∈N'\S'
       Si ("A→α.,a"∈ Ii) accion[i,a]:="reducir A→α"  FSi
     FPara
     Si ("S'→ S.,$"∈ Ii)  accion[i,$]:="aceptar" FSi
  FPara
  asignar "ERROR" a entradas vacías
  dev(accion)
Fin
```

- Importante:
  - si en la ejecución del algoritmo se llega a alguna contradicción
    - » el método no es aplicable para la gramática considerada
    - » no es una gramática LR(1)
  - si la ejecución genera una tabla
    - » se trata de una gramática LR(1)
- Esta tabla junto con la siguiente establecen el método LR(1) "canónico"

Estado inicial del analizador:
el construído a partir de

"S'→.S,$"

```
Función constIr_a(E G':gramática) dev
                  (ir_a[]:....)
Pre: G'=(N',T,S',P')es una gramática
     ampliada
     C={I0,...,In}= conjLR_1(G')
Post: ir_a es parte de la tabla LR(1)
      canónico
Variables i:entero
Principio
  Para i:=0 hasta n
     Para cada A∈N'
       Si suc1(Ii,A)=Ij
          ir_a[i,A]:=j
       FSi
     FPara
  FPara
  asignar "ERROR" a entradas vacías
  dev(ir_a)
Fin
```

1) $S' \rightarrow S$
2) $S \rightarrow CC$
3) $C \rightarrow \mathbf{c}C$
4) $C \rightarrow \mathbf{d}$

- Las tablas para el ejemplo anterior son

acción[]

	c	d	$
0	d 3	d 4	
1			Aceptar
2	d 6	d 7	
3	d 3	d 4	
4	r 4	r 4	
5			r 2
6	d 6	d 7	
7			r 4
8	r 3	r 3	
9			r 3

ir_a[]

	S	C
0	1	2
1		
2		5
3		8
4		
5		
6		9
7		
8		
9		

Análisis LALR

- En general, ocurre que:
  - tablas `'accion'` e `'ir_a'` para analizadores SLR son mucho más compactas que las tablas LR(1) canónico
  - pero hay importantes estructuras que no pueden ser resueltas con SLR
- Análisis LALR(1):
  - más compacto que LR(1) (mismo núm. estados que SLR(1))
  - menos potente que LR(1) canónico
  - buen compromiso `potencia/tamaño` para los lenguajes "frecuentes"
  - propuesto por DeRemer en 1969
- Espacio para Pascal:
  - SLR y LALR varios cientos de estados
  - LR(1) canónico varios miles de estados

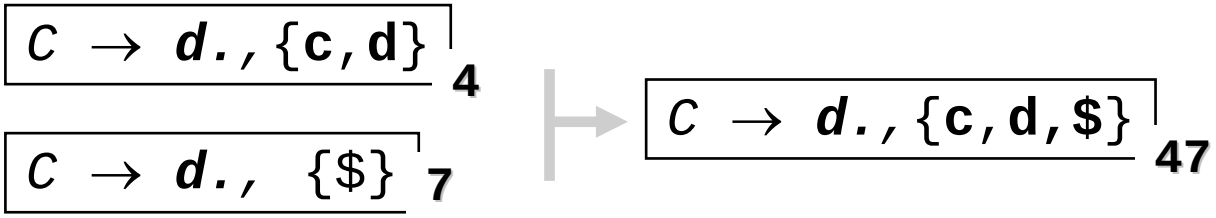
- Sea el estado LR(1)

```
s={ A → a.,{b,c},
    B → a.,{d}
  }
```

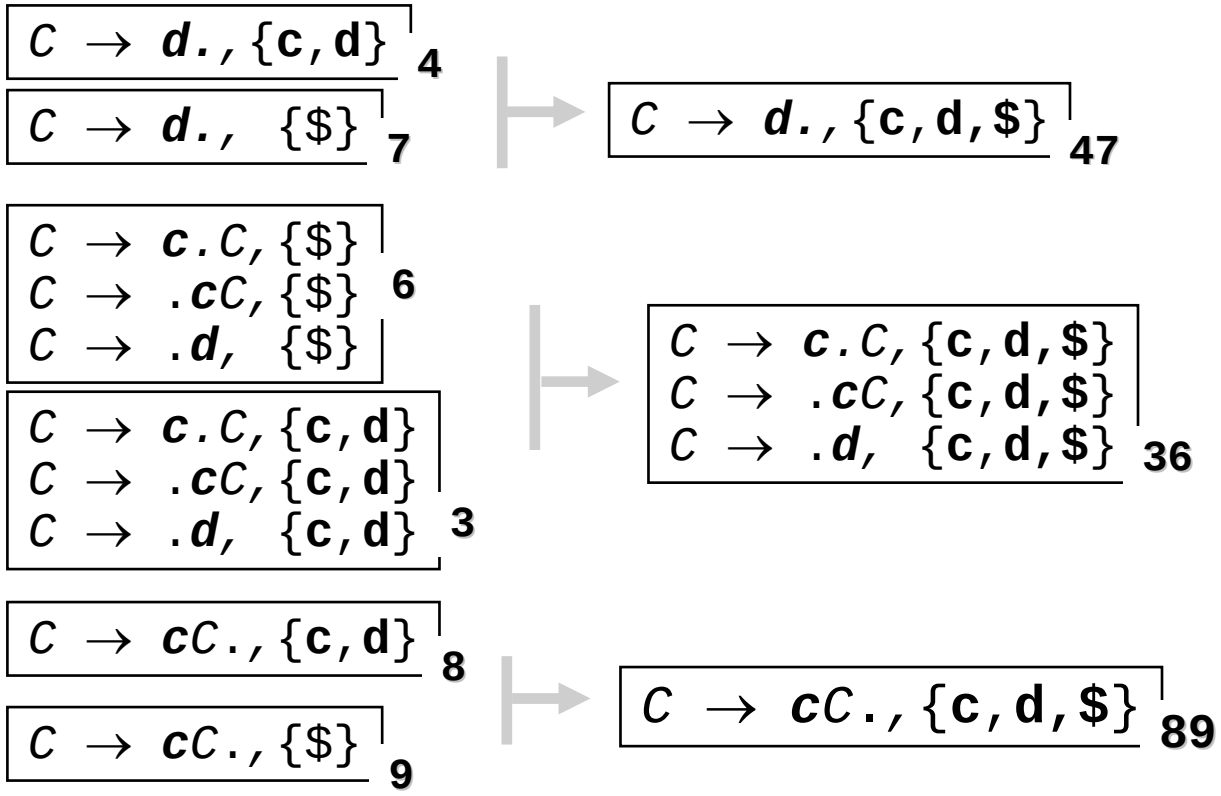
- Se le puede hacer corresponder el estado SLR(1)

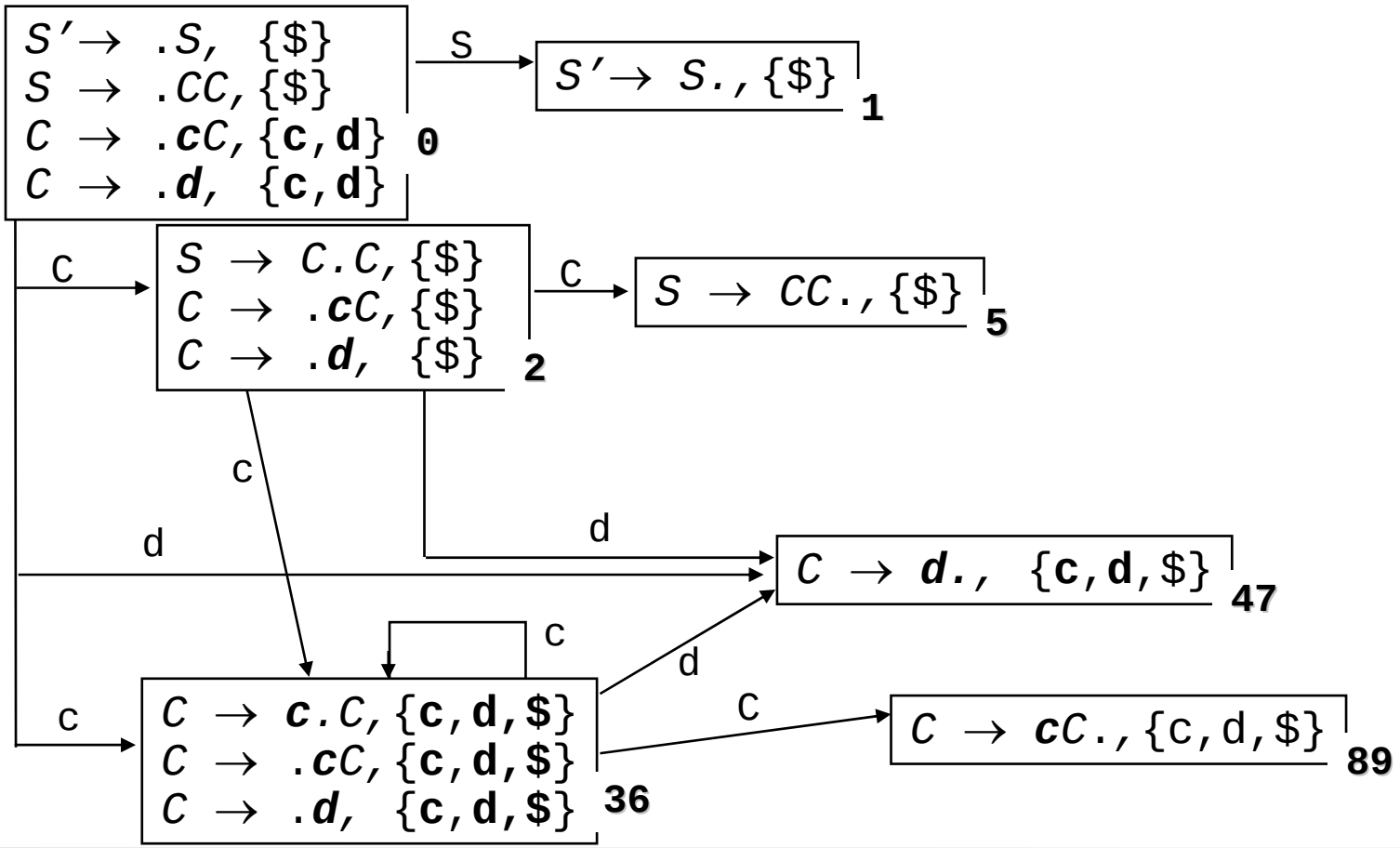
```
s={ A → a.,B → a. }
```

- Diremos que ***S es el corazón de S***
- En el momento de la generación de un nuevo nodo:
  - si hay uno con el mismo corazón
    - » fusionarlos, haciendo la unión de los conjuntos de preanálisis
    - » poner los arcos correspondientes



- En el ejemplo anterior:




S'→ .S, {$}
S → .CC, {$}
C → .cC, {c,d}
C → .d, {c,d}
0
S
S'→ S., {$}
1
c
S → C.C, {$}
C → .cC, {$}
C → .d, {$}
2
c
S → CC., {$}
5
c
d
d
C → d., {c,d,$}
47
c
d
c
C → c.C, {c,d,$}
C → .cC, {c,d,$}
C → .d, {c,d,$}
36
c
C
C → cC., {c,d,$}
89

- Yacc realiza el análisis LALR

```
yacc -v lG.y
```

```
S : CC ;
C : c C ;
C : d ;
```

```
state 0
  $accept : _S $end

  c  shift 3
  d  shift 4
  .  error

  S  goto 1
  C  goto 2
state 1
  $accept :  S_$end

  $end  accept
  .  error
state 2
  S :  C_C

  c  shift 3
  d  shift 4
  .  error
  C  goto 5
```

```
state 3
  C :  c_C

  c  shift 3
  d  shift 4
  .  error

  C  goto 6
state 4
  C :  d_    (3)

  .  reduce 3
state 5
  S :  C C_    (1)

  .  reduce 1

state 6
  C :  c C_    (2)

  .  reduce 2
```

bison -r itemset lG.y

```
state 0
 0 $accept: . S $end
 1 S: . C C
 2 C: . c C
 3  | . d
 c  shift, and go to state 1
 d  shift, and go to state 2
 S  go to state 3
 C  go to state 4

state 1
 2 C: . c C
 2  | c . C
 3  | . d
 c  shift, and go to state 1
 d  shift, and go to state 2
 C  go to state 5

state 2
 3 C: d .
 $default  reduce using rule 3 (C)
```

```
state 3
 0 $accept: S . $end
 $end  shift, and go to state 6

state 4
 1 S: C . C
 2 C: . c C
 3  | . d
 c  shift, and go to state 1
 d  shift, and go to state 2
 C  go to state 7

state 5
 2 C: c C .
 $default  reduce using rule 2 (C)

state 6
 0 $accept: S $end .
 $default  accept

state 7
 1 S: C C .
 $default  reduce using rule 1 (S)
```

- ¿LL(1) ó LALR(1)?
  - simplicidad:
    - » ambos métodos son simples, pero LL(1) es más intuitivo
    - » es más fácil escribir una gramática que sea LALR(1)
    - » la mayoría de las construcciones habituales son LL(1)
  - generalidad: la mayoría de las gramáticas LL(1) son LALR(1)
  - tratamiento de "símbolos de acción" (para el tratamiento semántico):
    - » LL(1) permite ponerlos en cualquier parte de las reglas
    - » LALR(1) sólo al final
      - hay apaños (pero pueden generar conflictos)
  - recuperación de errores:
    - » LL(1) tiene en la pila lo que deseamos encontrar
    - » LALR(1) contiene lo que hemos encontrado

- tamaño del analizador: se ha calculado que, para los lenguajes habituales,

$$|\text{LALR}(1)| = 2^* |\text{LL}(1)|$$

- En conclusión:
  - LL(1) parece tener más ventajas, si la gramática se puede hacer fácilmente LL(1)
  - hay que manejar bien las dos técnicas
  - LL(1) es una mejor primera aproximación
  - todo depende de la disponibilidad de buenas herramientas
- En cuanto a la potencia de los métodos:
  - LR(0) < SLR(1) < LALR(1) < LR(1)
  - LL(1) < LR(1)

Gramáticas ambiguas en Yacc

- ¿Qué pasa cuando la gramática en ambigua?
  - Yacc aplica dos reglas:
    - » ante un conflicto red/desp, desplazar
    - » ante un conflicto red/red, reducir por la producción que se haya escrito antes
- ¿Si estas reglas no corresponden a lo que queremos hacer?
- Yacc da la siguiente posibilidad:
  - se puede asociar a cada terminal una precedencia:
    - » cuanto más tarde se declare, mayor precedencia
  - se puede asociar a cada terminal una asociatividad
    - » **`%left, %right, %nonassoc`**
  - se puede asociar a cada producción una precedencia y asociatividad
    - » la de su terminal más a la dcha.

- En caso de conflicto red/desp
  - si la producción tiene mayor precedencia que el token de la entrada, reducir
  - si tienen igual precedencia, pero la asociatividad de la producción es por la izda. reducir
  - en los demás casos, desplazar

	por defecto	con prec.
-1+2	-3	1
2*3+4	14	10
-1+-2*3	5	-7

```
%token id
%left '+' '-'
%left '*' '/'
%right MENOS
%%
E:
   E '+' E
 | E '-' E
 | E '*' E
 | E '/' E
 | '(' E ')'
 | '-' E %prec MENOS
 | id
;
%%
```

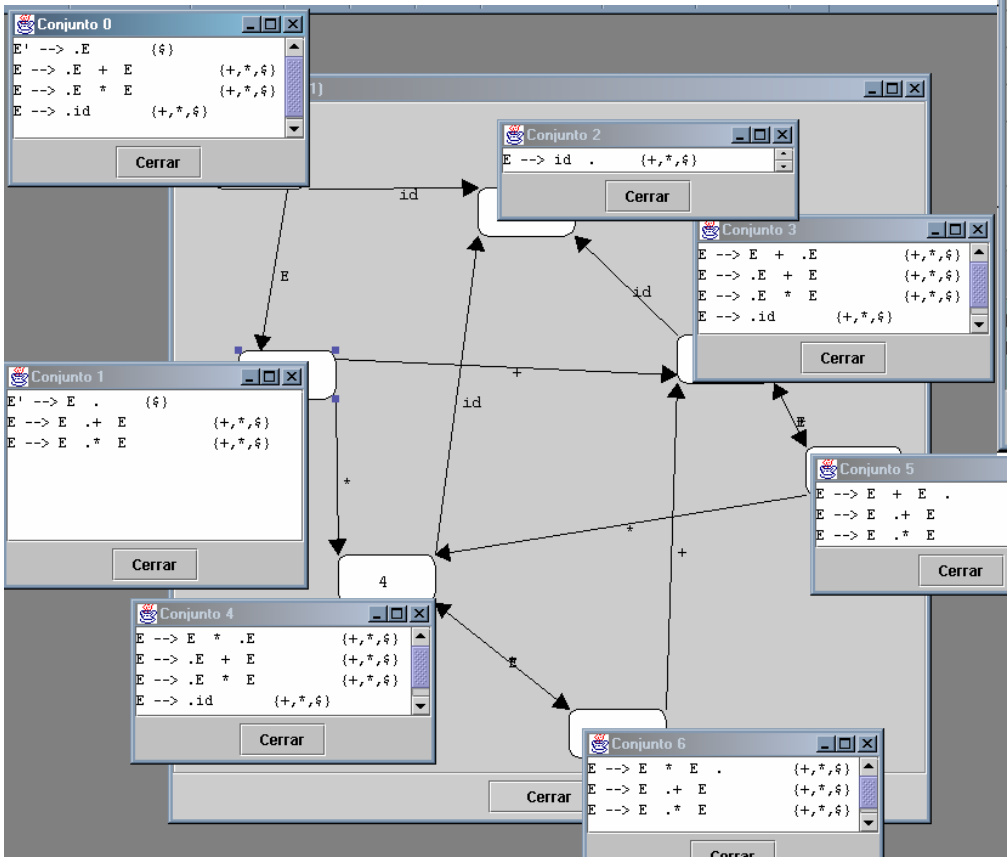
Gramáticas ambiguas en Yacc

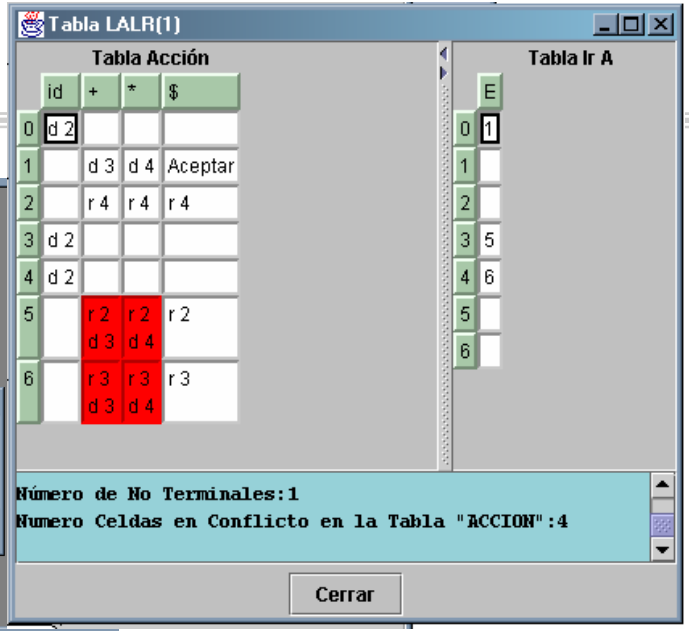
"ACTION" Table

	id	+	-	*	/	(	)	$
0	d 4		d 3			d 2		
1		d 5	d 6	d 7	d 8			Accept
2	d 4		d 3			d 2		
3	d 4		d 3			d 2		
4		r 8	r 8	r 8	r 8		r 8	r 8
5	d 4		d 3			d 2		
6	d 4		d 3			d 2		
7	d 4		d 3			d 2		
8	d 4		d 3			d 2		
9		d 5	d 6	d 7	d 8		d 15	
10		r 7 d 5	r 7 d 6	r 7 d 7	r 7 d 8		r 7	r 7
11		r 2 d 5	r 2 d 6	r 2 d 7	r 2 d 8		r 2	r 2
12		r 3 d 5	r 3 d 6	r 3 d 7	r 3 d 8		r 3	r 3
13		r 4 d 5	r 4 d 6	r 4 d 7	r 4 d 8		r 4	r 4
14		r 5 d 5	r 5 d 6	r 5 d 7	r 5 d 8		r 5	r 5
15		r 6	r 6	r 6	r 6		r 6	r 6

```
.....
***** Configuration Set 10 *****
E --> -  E  . {/,-,+,*,),$}
E --> E  .+  E       {/,-,+,*,),$}
E --> E  .-  E       {/,-,+,*,),$}
E --> E  .*  E       {/,-,+,*,),$}
E --> E  ./  E       {/,-,+,*,),$}
Edge to 5 labelled '+'
Edge to 6 labelled '-'
Edge to 7 labelled '*'
Edge to 8 labelled '/'
***** Configuration Set 11 *****
E --> E  +  E  .     {/,-,+,*,),$}
E --> E  .+  E       {/,-,+,*,),$}
E --> E  .-  E       {/,-,+,*,),$}
E --> E  .*  E       {/,-,+,*,),$}
E --> E  ./  E       {/,-,+,*,),$}
Edge to 5 labelled '+'
Edge to 6 labelled '-'
Edge to 7 labelled '*'
Edge to 8 labelled '/'
***** Configuration Set 12 *****
.....
```

Gramáticas ambiguas en Yacc






```
1) E' --> E
2) E  --> E  +  E
3) E  --> E  *  E
4) E  --> id
```

- Yacc implementa un método propio de recuperación de errores mediante la adición de "**producciones de error**“
  - producciones especiales que incorporan el terminal ficticio "***error***“
- Ejemplo: un error común en Pascal es dejarse el ‘;’ que sigue al identificador del programa

```
programa:
     tkPROGRAM tkID ';' parteDecs bloqueInst '.' ;
```

- Por lo que la gramática se puede transformar en

```
programa:
    tkPROGRAM tkID ';' parteDecs bloqueInst '.'
|   tkPROGRAM tkID error parteDecs bloqueInst '.'
;
```

  - ***error*** es un token especial definido por la herramienta

- Método de recuperación aplicado una vez detectado un error:
  - se invoca el procedimiento `yyerror()`
    - » Éste se encuentra definido en la librería de Yacc (recordar que se incluye mediante la opción "`-ly`")
    - » Salvo que se redefina, dicho procedimiento escribe un mensaje y aborta la ejecución. Por lo tanto, si deseamos llevar a cabo algún tipo de recuperación del error deberemos redefinirlo.
  - se pone como siguiente token de la entrada el token ficticio "**error**"
  - se eliminan estados de la pila, sucesivamente, hasta llegar a un estado capaz de ejecutar un *desplazamiento* con un token "error"
    - » eventualmente, puede llegar a vaciar la pila, terminando el análisis en una situación de error

- una vez ejecutado dicho desplazamiento, se consulta de nuevo la tabla acción para ver si existe alguna acción para el nuevo estado alcanzado y el token de la entrada (que es el mismo que generó el primer error)
  - » si se produce un nuevo error se ejecuta una secuencia de eliminaciones de tokens de la entrada hasta encontrar uno que pueda seguir a un token de error, lo que normalmente finalizará con una reducción por la producción de error, abandonando el estado de error
- el analizador considerará que ha salido de la condición de error detectada cuando sea capaz de desplazar al menos tres tokens consecutivos sin detectar un nuevo error sintáctico
  - » Se puede evitar este comportamiento mediante el uso de la acción "*yyerrok*" (consultar documentación sobre Yacc)

- Lo más complicado es establecer cuáles son las producciones de error interesantes para un lenguaje dado. En [ScFr 85] se indican las siguientes pautas sobre en qué partes del análisis sintáctico conviene introducir producciones de error:
  - tan cerca como sea posible del símbolo inicial de la gramática (esto evitaría que en la ejecución del paso 3 se llegue a vaciar la pila)
  - tan cerca como sea posible de cada símbolo terminal (esto evitaría eliminar muchos tokens de la entrada en la ejecución del paso 4)
  - de manera que la adición de las producciones de error no introduzca conflictos (shift/reduce o reduce/reduce)

Schreiner A., Friedman H. Jr.

Introduction to Compiler Construction with Unix

Prentice-Hall, 1985

Recuperación de errores sintácticos en el análisis ascendente. El caso de Yacc

- Parece claro que algunas de las pautas anteriores son excluyentes entre sí
- En cualquier caso, en [ScFr 85] se recomienda añadir producciones error:
  - en cada construcción recursiva
  - evitando que el token error aparezca al final de la producción
  - introduciendo dos producciones error en cada lista no vacía de elementos: una para los problemas al principio de la lista y otra para los problemas al final de la lista
  - en la rama vacía de las listas que puedan ser vacías

- Algunos ejemplos interesantes:

0 ó más

```
x:
   {/*vacía*/}
|  x y   {yyerrok;}
|  x  error
;
```

1 ó más con separadores

```
x:
   y
|  x T y          {yyerrok;}
|  error
|  x error
|  x error y      {yyerrok;}
|  x T error
;
```

1 ó más

```
x:
   y
|  x y          {yyerrok;}
|  error
|  x error
;
```

```
inst:
   ....
|  error  ';' /*en caso de
               error, saltará
               hasta encontrar
               ';'*/
;
```