Análisis sintáctico

Dpto. de Ingeniería de la Información y las Comunicaciones

Facultad de Informática
UNIVERSIDAD DE MURCIA

Índice: secciones principales

Objetivo del analizador sintáctico

- **Comprueba si la secuencia de tokens proporcionada por el a. léxico puede ser generada por la gramática del lenguaje fuente.**
- Debe **informar y recuperarse de los errores sintácticos**, que ocurren frecuentemente, para poder continuar procesando el resto de la entrada.
- La salida suele ser el **árbol sintáctico**.
- Puede realizar otras tareas también como:
  - **Recoger información sobre tokens en la tabla de símbolos**.
  - **Realizar analisis semántico, como la verificación de tipos**.
  - **Generar código intermedio**.

Manejo y recuperación de errores sintácticos

Está muy centrado en esta fase porque...

→ ...**muchos** de los **errores** de compilación **son sintácticos**.

→ ...existen **métodos muy eficientes** para detectarlos aquí.

Objetivos del manejador de errores:

1. **Informar** de su presencia **con claridad y exactitud**.
2. **Recuperarse con rapidez**. **Posibles estrategias**:
   - En *modo pánico*.
   - A nivel de frase.
   - De producciones de error.
   - De corrección global.
3. **No retrasar el procesamiento de programas correctos**.

Fundamentos teóricos del análisis sintáctico: GLC y derivaciones izquierdas y derechas

Una gramática libre de contexto (GLC) $G = (V_N, V_T, S, P)$...

...es aquella que tiene las producciones de la forma $A \to \alpha$, $A \in V_N$, $\alpha \in (V_N \cup V_T)^*$.

Una derivación más a la izquierda (m.i.)/derecha (m.d.) de una GLC para una forma sentencial (FS)...

...es una derivación que comienza con el símbolo inicial, acaba en la FS, y en cada derivación directa se aplica una regla de producción para la variable más a la izquierda / derecha de la cadena que se está derivando. Entonces, la FS es una **forma sentencial izquierda** (**derecha**).

Fundamentos teóricos del análisis sintáctico: pivotes

Pivote

A una derivación más a la derecha de longitud n

$$S \Rightarrow_{m.d.} \gamma_1 \Rightarrow_{m.d.} \gamma_2 \Rightarrow_{m.d.} \cdots \Rightarrow_{m.d.} \gamma_n$$

le corresponde una reducción por la izquierda

$$\gamma_n \Rightarrow^{R}_{m.i.} \gamma_{n-1} \Rightarrow^{R}_{m.i.} \cdots \Rightarrow^{R}_{m.i.} S$$

donde en cada $\gamma_i \Rightarrow^{R} \gamma_{i-1}$ se sustituye la parte derecha de una regla por la izquierda.

Si $\gamma_i = \alpha_1\beta\alpha_2$ y $\gamma_{i-1} = \alpha_1 A\alpha_2$ entonces $A \to \beta \in P$.

β es el **pivote** de la forma sentencial.

Fundamentos teóricos del análisis sintáctico: pivotes

Ejemplo.- Pivote para una gramática con las siguientes reglas:

$$
\begin{array}{lcl}
S & \rightarrow & z\ A\ B\ z \\
A & \rightarrow & a\ A\ a \\
 & | & a \\
B & \rightarrow & b\ B\ b \\
 & | & b
\end{array}
$$

El pivote de la forma sentencial derecha $zaaAaabbbz$
es aAa, puesto que:
$S \Rightarrow^* zaAabbbz \Rightarrow zaaAaabbbz$.

El **pivote** de una FS derecha **no** tiene por qué ser **único si la gramática es ambigua**.

Ejemplo: $E \rightarrow E + E | E * E | (E) | i$

$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * i$ El pivote es i

$E \Rightarrow E * E \Rightarrow E * i \Rightarrow E + E * i$ El pivote es $E + E$

Fundamentos teóricos del análisis sintáctico: árboles de derivación

Un árbol de derivación...

...se construye creando un nodo raíz etiquetado con S. Éste tendrá nodos hijos etiquetados cada uno con el símbolo de la cadena que se corresponde con la parte derecha de una regla de producción para S.

- Cada nodo etiquetado con un símbolo de V_N se expande como S.
- Cada nodo etiquetado con un símbolo de V_T será una hoja.

Fundamentos teóricos del análisis sintáctico: árboles de derivación

Gramática:

$E \rightarrow E + E$

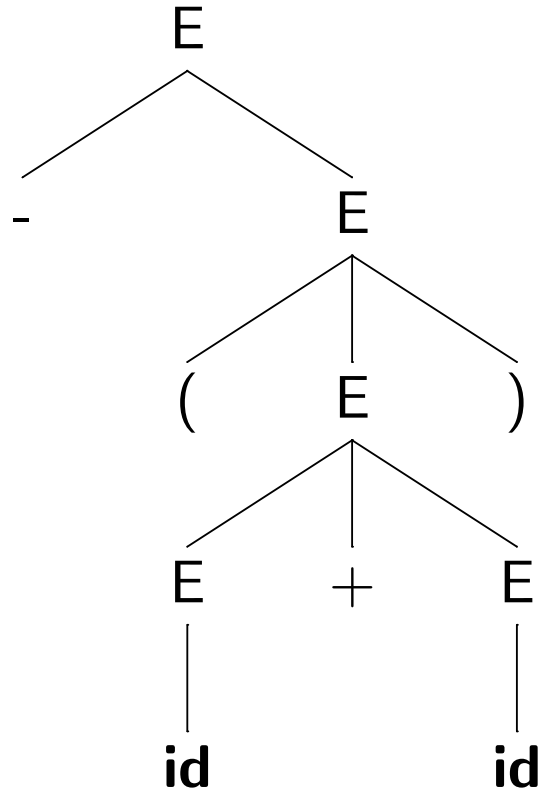
$E \rightarrow E * E$

$E \rightarrow -E$

$E \rightarrow (E)$

$E \rightarrow \mathbf{id}$

Árbol de derivación de -(**id**+**id**):



$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Fundamentos teóricos del análisis sintáctico: árboles de derivación

- En una g.l.c., una secuencia de derivaciones $S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow ... \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \equiv w$ puede representarse mediante un **árbol de derivación**.
- A un mismo árbol pueden corresponderle derivaciones distintas, pero a una derivación le corresponde un sólo árbol.
- Cada nodo etiquetado con una variable, tendrá asociado un árbol que parte de él y refleja la porción de cadena derivable a partir de esa variable.

Fundamentos teóricos del análisis sintáctico: ambigüedad

Una sentencia w ($S \Rightarrow^* w$) de una gramática l.c. es ambigua...

...si existen para ella dos o más árboles de derivación distintos o si podemos derivar la sentencia mediante dos o más derivaciones m.i. (o m.d.) distintas.

Una gramática es ambigua...

...si tiene al menos una sentencia ambigua.

- El proceso de decidir si una gramática es ambigua **no es algorítmico**.
- La ambigüedad no es deseable para gramáticas que definen *lenguajes de programación*.
  - → A veces puede eliminarse encontrando una gramática equivalente no ambigua (**lenguaje no ambiguo**).
  - → Otras veces no (**lenguajes inherentemente ambiguos**).
    Ejemplo: $L = \{a^n b^n c^m d^m / m, n \geq 1\} \cup \{a^n b^m c^m d^n / m, n \geq 1\}$, en el que todas las sentencias de la forma $a^i b^i c^i d^i$ son ambiguas.

El problema de la ambigüedad en el análisis sintáctico

→ **Debemos evitar la ambigüedad**, pues **cada árbol** de derivación para una misma sentencia, implica **significados diferentes** para ella.

→ **Un mismo programa podría calcular cosas distintas al encontrar dos árboles de derivación diferentes en el análisis**.

Ejemplo: ambigüedad en las operaciones aritméticas

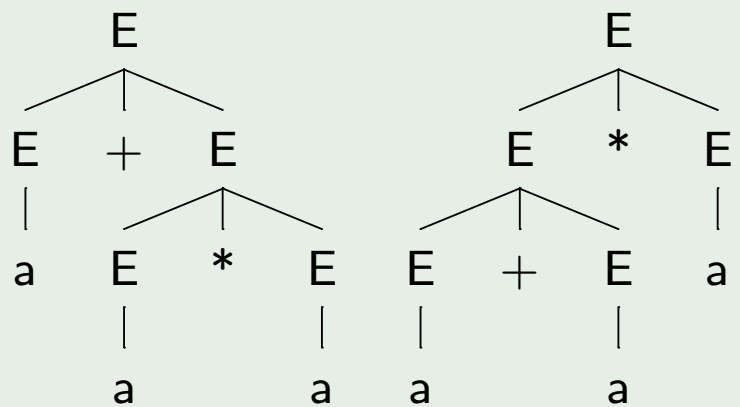
◂ LengNoAmb

Gramática no ambigua:

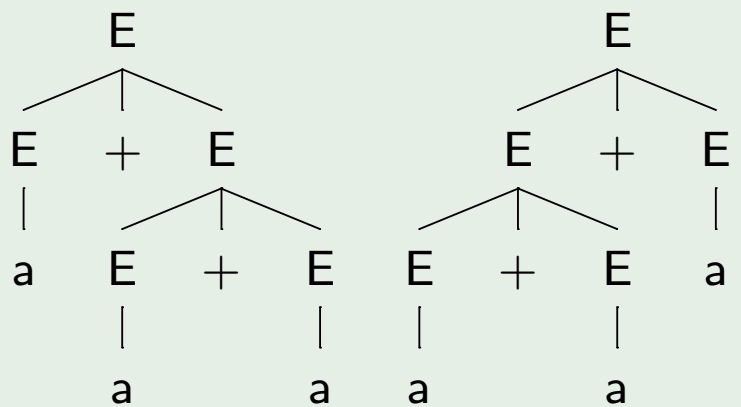
$$E \to E + E \mid E * E \mid a \qquad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to a$$

Mayor **precedencia** de $*$.
Asociatividad por la izquierda.

$a + a * a$



$a + a + a$



El problema de la ambigüedad en el análisis sintáctico

Ejemplo: ambigüedad en la sentencia `if-then-else`

Gramática:

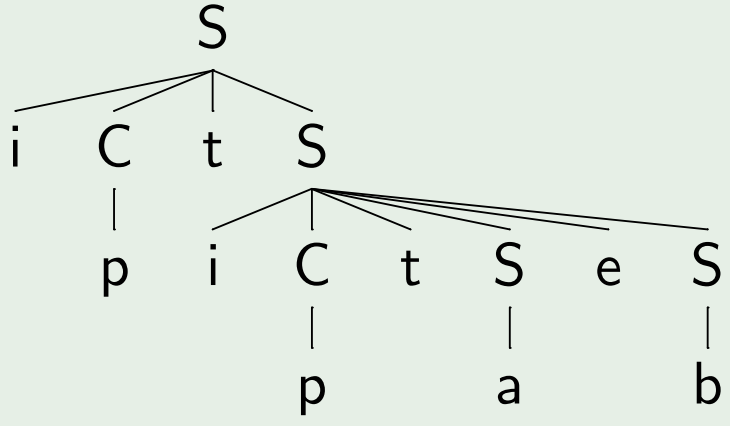
$$S \rightarrow i\ C\ t\ S\ e\ S \mid i\ C\ t\ S \mid a \mid b$$
$$C \rightarrow p$$

dónde i=if, p=predicado y t=then

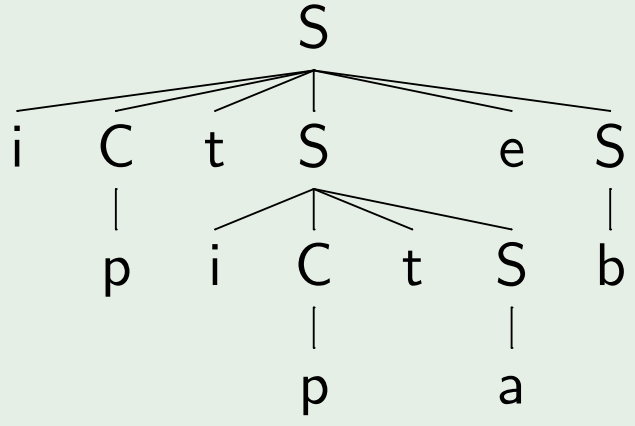
Sentencia: i p t i p t a e b

Árboles:

(válido)



(no válido)

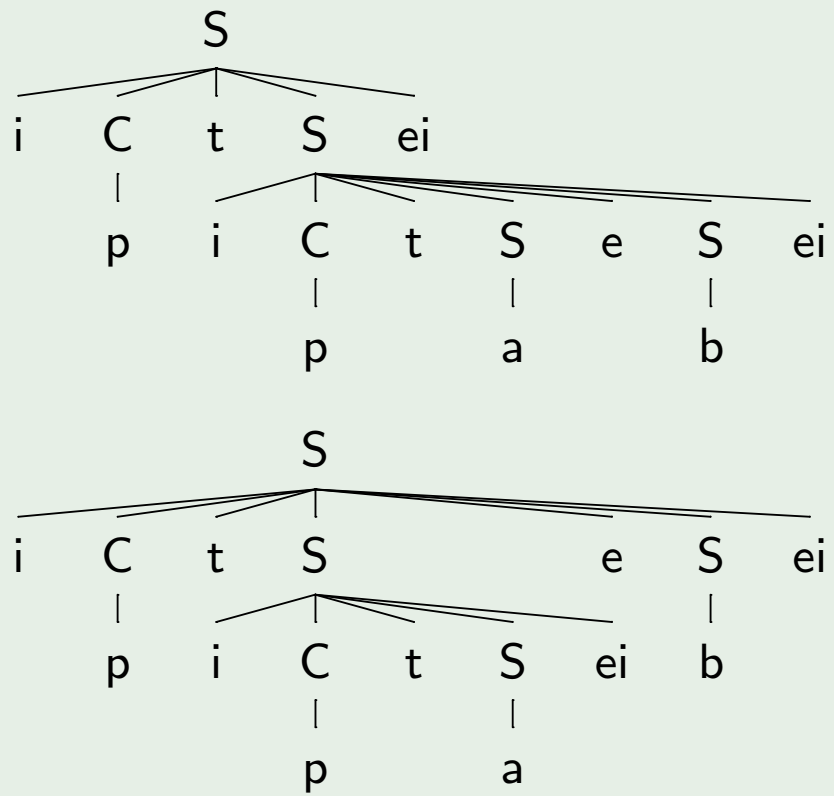


El problema de la ambigüedad en el análisis sintáctico

Ejemplo: ambigüedad en la sentencia `if-then-else`. Soluciones:

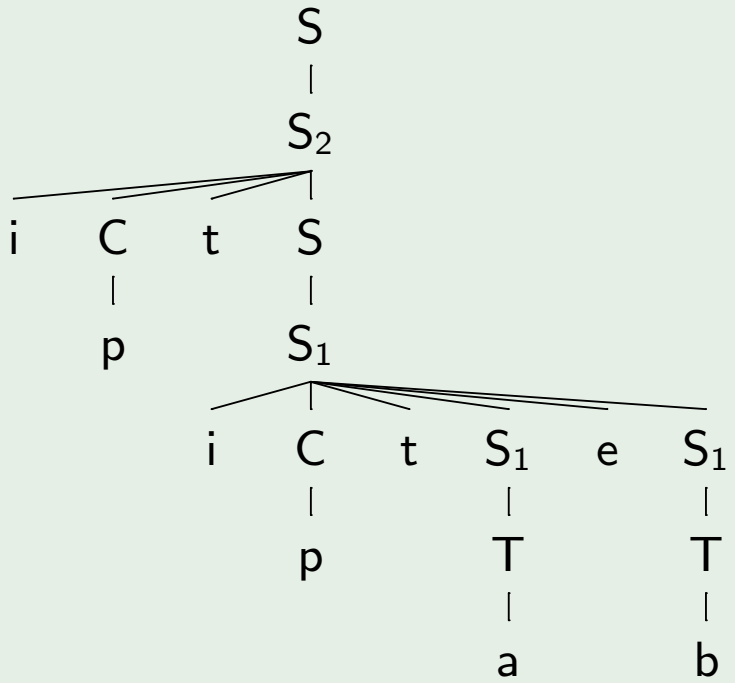
Transformar la gramática añadiendo `endif` .

$S \to i\ C\ t\ S\ e\ S\ ei \mid i\ C\ t\ S\ ei \mid a \mid b$
$C \to p$



Transformar la gramática en otra equivalente no ambigua.

$S \to S_1 \mid S_2$
$S_1 \to i\ C\ t\ S_1\ e\ S_1 \mid T$
$S_2 \to i\ C\ t\ S \mid i\ C\ t\ S_1\ e\ S_2$
$C \to p$
$T \to a \mid b$



Fundamentos teóricos del análisis sintáctico: A. de Pila

Un Autómata con Pila, M,...

...es una 7-upla $(Q, V, \Sigma, \delta, q_0, z_0, F)$ donde:

- Q es un **conjunto finito de estados**.
- V es el **alfabeto de entrada**.
- Σ es el **alfabeto de la pila**.
- q_0 es el **estado inicial**.
- z_0 es el **símbolo inicial de la pila**.
- $F \subseteq Q$ es el **conjunto de estados finales**.
- δ es la **función de transición**:

$$\delta : Q \times (V \cup \{\lambda\}) \times \Sigma \rightarrow P(Q \times \Sigma^*)$$

Fundamentos teóricos del análisis sintáctico: A. de Pila

El autómata será no determinista: dado un estado q, un símbolo del alfabeto de entrada a y otro del alfabeto de la pila z, el autómata puede pasar a distintos estados p_j y reemplazar el tope de la pila z por distintas cadenas, avanzando o no la cabeza lectora una posición:

$$\delta(q, a, z) = \{(p_1, \gamma_1), (p_2, \gamma_2), ..., (p_m, \gamma_n)\}.$$

El lenguaje puede ser aceptado...

...**por estado final**:

$L(M) = \{w \in V^* / (q_0, w, z_0) \vdash_M^* (q_f, \lambda, \gamma), q_f \in F \wedge \gamma \in \Sigma^*\}$

...**por pila vacía**:

$L(M) = \{w \in V * / (q_0, w, z_0) \vdash_M^* (q, \lambda, \lambda), q \in Q\}$

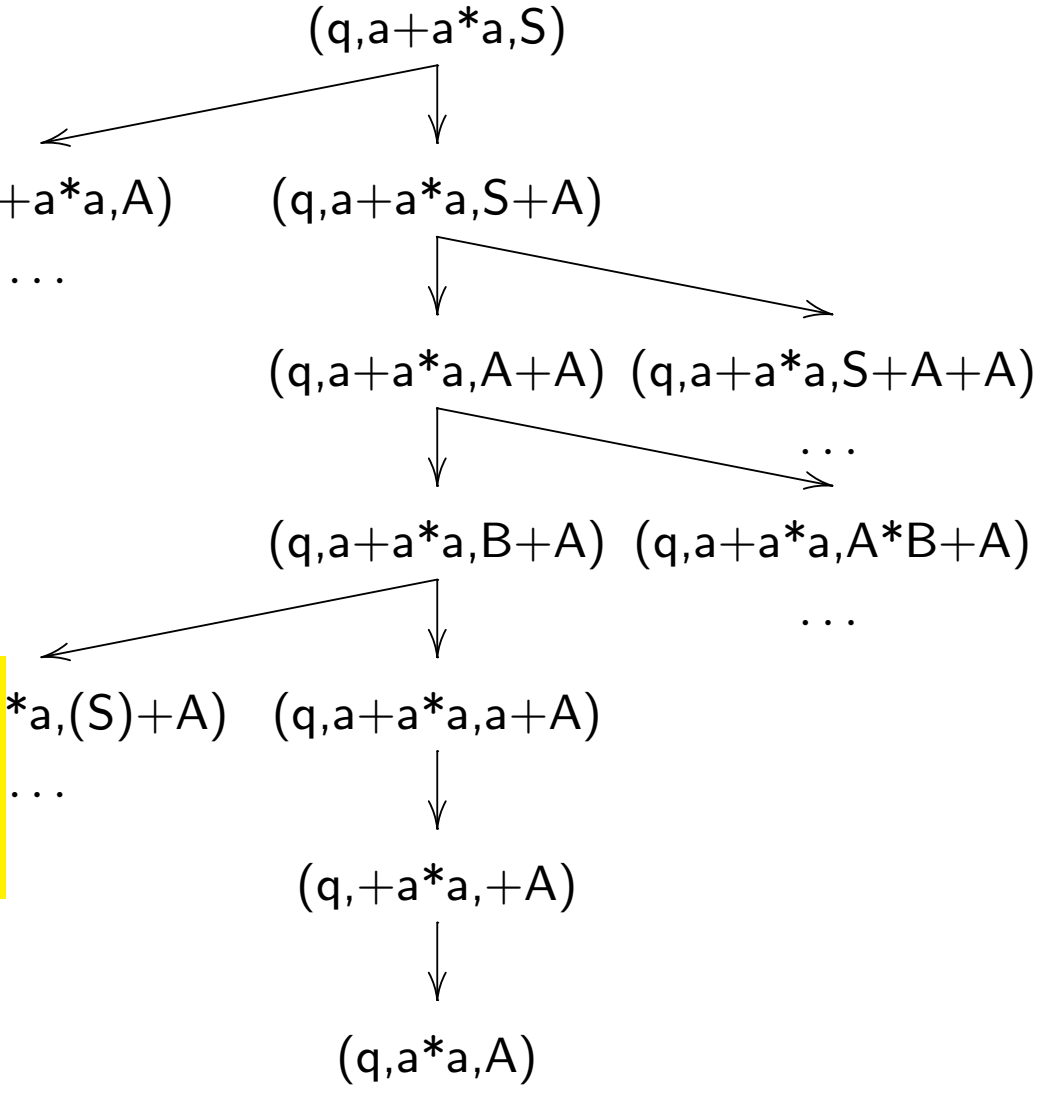
Fundamentos teóricos del análisis sintáctico: A. de Pila

Ejemplo: **cálculos** de reconocimiento de la cadena $a + a * a$ (1):

Gramática:
$S \to S + A | A$
$A \to A * B | B$
$B \to (S) | a$

Autómata:
$M = (\{q\}, \{a, +, *, (,)\},$
$\{a, +, *, (,), A, B, S\}, \delta, q, S, \emptyset)$

$\delta(q, \lambda, S) = \{(q, S + A), (q, A)\}$
$\delta(q, \lambda, A) = \{(q, A * B), (q, B)\}$
$\delta(q, \lambda, B) = \{(q, (S)), (q, a)\}$
$\delta(q, x, x) = \{(q, \lambda)\}, \forall x \in \{a, +, *, (,)\}$



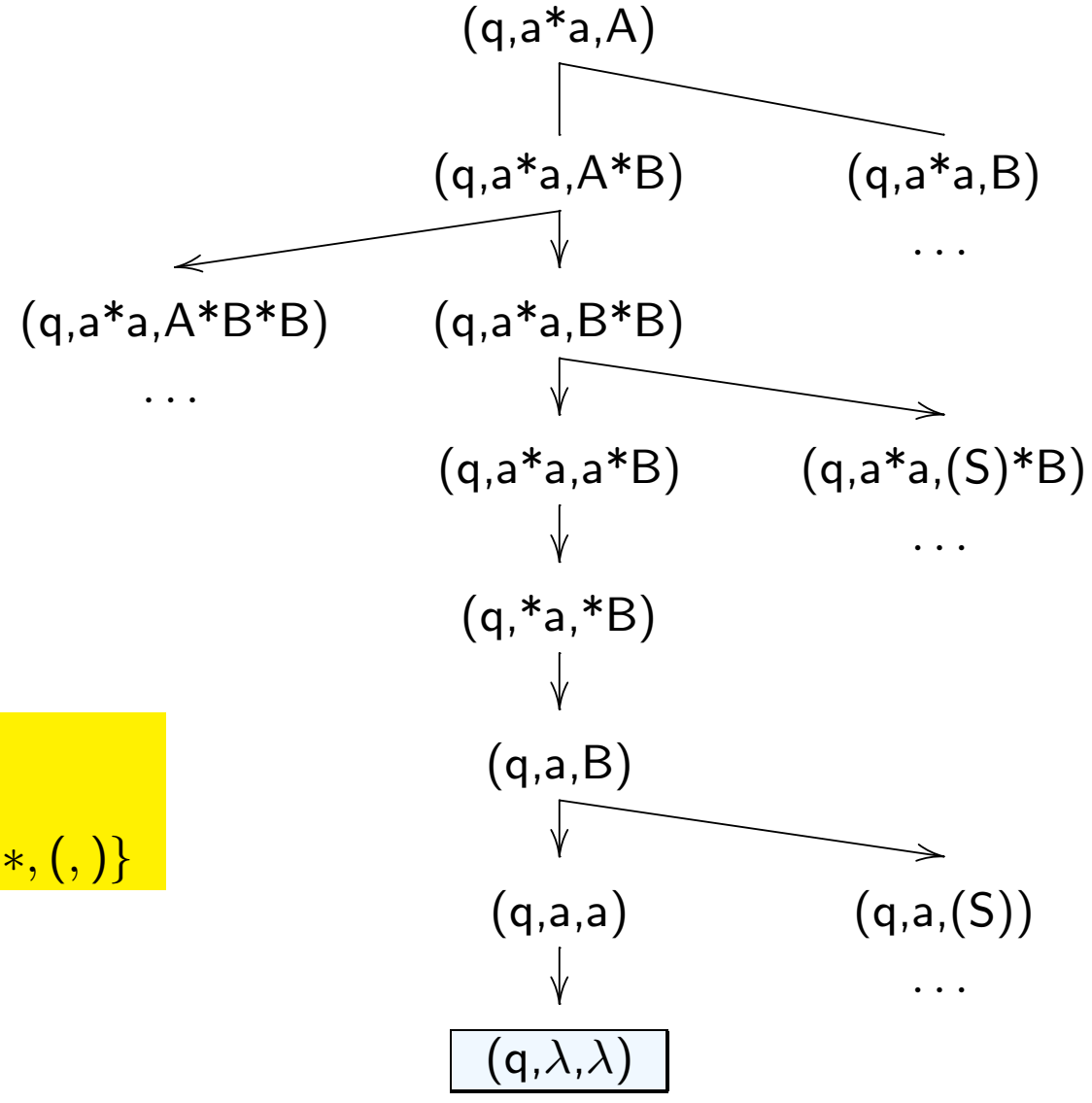
Fundamentos teóricos del análisis sintáctico: A. de Pila

Ejemplo: **cálculos** de reconocimiento de la cadena $a + a * a$ (2):

Gramática:
$S \to S + A | A$
$A \to A * B | B$
$B \to (S) | a$

Autómata:
$M = (\{q\}, \{a, +, *, (,)\},$
$\{a, +, *, (,), A, B, S\}, \delta, q, S, \emptyset)$

$\delta(q, \lambda, S) = \{(q, S + A), (q, A)\}$
$\delta(q, \lambda, A) = \{(q, A * B), (q, B)\}$
$\delta(q, \lambda, B) = \{(q, (S)), (q, a)\}$
$\delta(q, x, x) = \{(q, \lambda)\}, \forall x \in \{a, +, *, (,)\}$



Características no libres de contexto de los lenguajes de programación

Algunas construcciones sintácticas no pueden ser especificadas con gramáticas libres de contexto $\rightarrow$ Se dejan para la fase de análisis semántico:

- $\rightarrow$ $L_1 = \{wcw/w \in (a|b)^*\}$ *declaración de identificadores antes de su uso*.
- $\rightarrow$ $L_2 = \{a^n b^m c^n d^m / n \geq 1 \ \wedge \ m \geq 1\}$ *coincidencia de nº de parámetros en declaración y en su uso*.

Análisis sintáctico descendente y ascendente

Los métodos de análisis pueden clasificarse en:

- **Descendentes** : Se construye un árbol sintáctico a partir de la raíz, etiquetada con el *símbolo inicial* de la gramática, hacia abajo (*derivaciones por la izquierda*), hasta llegar a las hojas del árbol, etiquetadas con los *símbolos terminales*.
- **Ascendentes** : Se parte de las hojas y se intenta construir el árbol hacia arriba (*reducciones por la izquierda*), hasta llegar al símbolo inicial de la gramática.

→ Entre los métodos ascendentes y descendentes existen algunos que son **generales**, es decir, que *pueden analizar cualquier gramática libre de contexto*.

→ Sin embargo, **los más eficientes**, usados normalmente en compiladores, *trabajan sólo con un subconjunto de gramáticas* (como **LL** y **LR**) aunque son lo suficientemente expresivos como para describir a la mayoría de los lenguajes de programación.

Conjuntos PRIMERO y SIGUIENTE

PRIMERO

$$PRIMERO(\alpha) = \{a \in V_T / \alpha \Rightarrow^* a\beta\} \cup \{if\ \alpha \Rightarrow^* \lambda\ then\ \{\lambda\}\ else\ \emptyset\}$$

PRIMERO(X)

Aplíquense las reglas siguientes hasta que no se puedan añadir más terminales o λ a ningún conjunto PRIMERO:

1. Si X es **terminal**, entonces PRIMERO(X) es {X}.
2. Si X es λ, entonces PRIMERO(λ) es {λ}.
3. Si X es **no terminal** y X $\rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$ son producciones de la gramática, entonces PRIMERO(X) = PRIMERO(α_1) $\cup \cdots \cup$ PRIMERO(α_k).

PRIMERO(α_i)=PRIMERO($X_1 X_2 \ldots X_n$)

1. Añádanse a PRIMERO($X_1 X_2 \ldots X_n$) los símbolos distintos de λ de PRIMERO(X_1)
2. Póngase a en PRIMERO($X_1 X_2 \ldots X_n$) si, para alguna i ($2 \leq i \leq n$), a está en PRIMERO(X_i) y $X_1 \ldots X_{i-1} \Rightarrow^* \lambda$.
3. Añádase λ a PRIMERO($X_1 X_2 \ldots X_n$) si para toda i, PRIMERO(X_i) contiene λ.

Conjuntos PRIMERO y SIGUIENTE

SIGUIENTE

$SIGUIENTE(A) = \{a \in V_T / S \Rightarrow^+ ...\alpha A a \beta\} \cup \{if\ S \Rightarrow^+ \alpha A\ then\ \{\$\}\ else\ \emptyset\}$

SIGUIENTE(A)

Aplíquense las reglas siguientes hasta que no se pueda añadir nada más a ningún conjunto SIGUIENTE.

1. Póngase $ en SIGUIENTE(A), si A es el símbolo inicial. $ es el delimitador derecho de la entrada.
2. Si hay una producción B$\rightarrow$ αAβ, entonces todo lo que esté en PRIMERO(β) excepto λ se pone en SIGUIENTE(A).
3. Si hay una producción B$\rightarrow$ αA o una producción B$\rightarrow$ αAβ, donde PRIMERO(β) contenga λ (es decir, $\beta \Rightarrow^* \lambda$), entonces todo lo que esté en SIGUIENTE(B) se pone en SIGUIENTE(A).

Conjuntos PRIMERO y SIGUIENTE

Ejemplo de cálculo de PRIMERO y SIGUIENTE

$$
\begin{array}{lcl}
A & \rightarrow & B\ e \mid a \\
B & \rightarrow & C\ D \mid b \\
C & \rightarrow & c \mid \lambda \\
D & \rightarrow & d \mid \lambda
\end{array}
$$

$PRIMERO(A) = \{a, b, c, d, e\}$
$PRIMERO(B) = \{b, c, d, \lambda\}$
$PRIMERO(C) = \{c, \lambda\}$
$PRIMERO(D) = \{d, \lambda\}$
$SIGUIENTE(A) = \{\$\}$
$SIGUIENTE(B) = \{e\}$
$SIGUIENTE(C) = \{d, e\}$
$SIGUIENTE(D) = \{e\}$

Análisis ascendente predictivo: gramáticas LR

→Es un **método de análisis ascendente sin retroceso** que opera por **reducción-desplazamiento** ("*shift-reducing*").
→No sirve para cualquier gramática libre de contexto, aunque **es más potente que el análisis predictivo LL**, puesto que $GramLL \subset GramLR$.

Significado de la notación LR(k)

L : La entrada se examina de **izquierda a derecha** (*left to right*).

R : Se realiza la **derivación más a la derecha** en orden inverso (*right most derivation*).

k : Requiere **k tokens de anticipación** para decidir la siguiente acción.

Sólo estudiaremos el análisis **LR(1)** .

Modo de operar por reducción-deplazamiento:

Si $S \Rightarrow^{+}_{m.d.} w$, se puede decir que w **reduce por la izquierda a** S:

$$S = \gamma_0 \Rightarrow_{m.d.} \gamma_1 \Rightarrow_{m.d.} \gamma_2 \Rightarrow_{m.d.} \cdots \Rightarrow_{m.d.} \gamma_{n-1} \Rightarrow_{m.d} \gamma_n = w.$$

γ_i reduce a γ_{i-1} y, para reducirlo, se busca el pivote de γ_i (p.e. β) y se sustituye por un A tal que $A \to \beta \in P$.

Análisis ascendente predictivo

- Con el análisis LR se reconocerán $w \in L(G)$ y se obtendrá la secuencia de reglas de producción aplicable para derivarlas.
- Eso se hará con ayuda de una **tabla de análisis**.

Técnicas de construcción de la tabla de análisis LR

- **Método SLR**: Es el más fácil y el menos potente.
- **Método LR-canónico**: Es el más potente, pero el más costoso.
- **Método LALR**: De compromiso entre los anteriores.

Analizadores y gramáticas SLR, LALR y LR-canónicos

- Cuando un **analizador LR** utiliza una tabla SLR, LR-canónica o LALR, se llama, respectivamente, **analizador SLR** , **LR-canónico** o **LALR** .
- La **gramática** que genera el lenguaje reconocido por él, se dice que es, también respectivamente, **gramática SLR** , **LR-canónica** o **LALR** .

Pueden surgir **conflictos** al intentar construir la tabla de análisis.

Algoritmo de análisis ascendente predictivo

Independientemente del tipo de tabla de análisis que utilicemos, el método de análisis LR siempre es el mismo.

- Llamamos **configuración** del analizador LR a un par $I = (s_0 X_1 s_1 X_2 s_2 ... X_m s_m, a_i a_{i+1} ... a_n \$)$

  $X_i \in V_N \cup V_T$

  s_i son estados.

- El **siguiente movimiento** lo determinan a_i, s_m y $accion[s_m, a_i]$, de forma que:

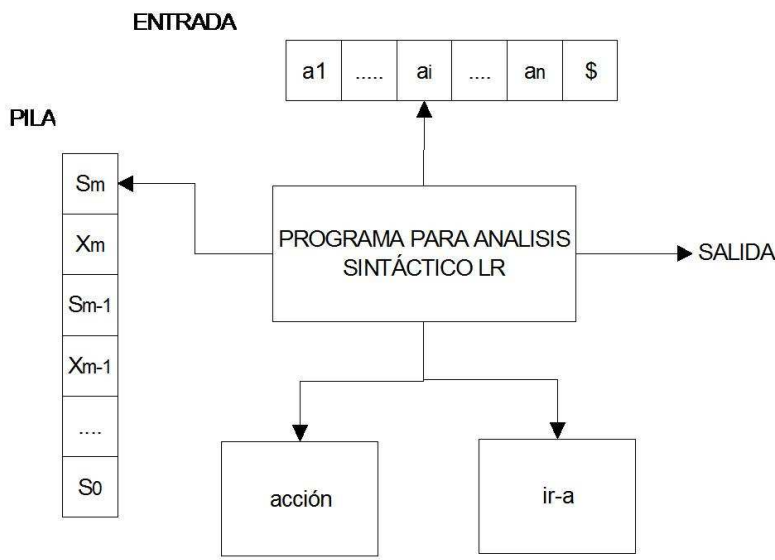
  - Si $accion[s_m, a_i]$ =desp s $\Rightarrow (s_0 X_1 s_1 X_2 s_2 ... X_m s_m a_i s, a_{i+1} ... a_n \$)$
  - Si $accion[s_m, a_i]$ =red $A \to \beta \Rightarrow (s_0 X_1 s_1 ... X_{m-r} s_{m-r} A s, a_i ... a_n \$)$
    con $ir\_a[s_{m-r}, A] = s$ y $|\beta| = r$. Salida $A \to \beta$
  - Si $accion[s_m, a_i]$ =acc$\Rightarrow$ Para
  - Si $accion[s_m, a_i]$ =error$\Rightarrow$Mensaje y recuperación

Los símbolos de la gramática incluidos en la pila no son imprescindibles. Aparecen sólo por una cuestión de claridad.

Algoritmo de análisis ascendente predictivo

Algoritmo Análisis sintáctico LR
Entrada Una cadena de entrada w y una tabla de análisis sintáctico LR con las funciones acción e ir_a para G.
Salida Si w está en $L(G)$, un análisis sintáctico ascendente de w, de lo contrario, *error*.
Método Inicialmente, la pila contiene el estado inicial s_0 y $w\$$ está en el *buffer* de entrada.

```
Apuntar ae al primer símbolo de w$;
while(1) {
    Sea s el estado en la cima de la pila
        y a el símbolo apuntado por ae;
    if (acción[s,a]==desplazar s´) {
        meter as´ en la cima de la pila;
        avanzar ae al siguiente símbolo
            de entrada y asignar a a;
    }
    else if (acción[s,a]==reducir A→β ) {
        sacar 2*|β| símbolos de la pila;
        sea s' el estado de la cima de la pila
        meter A y después ir_a[s´,A] en el tope;
        emitir la producción A→β
    }
    else if (acción[s,a]==aceptar)
        return
    else error();
}
```



Construcción de las tablas de análisis: tabla SLR

Definiciones básicas

- Se dice que α es **forma sentencial derecha** si $S \Rightarrow^*_{m.d.} \alpha$.
- Un **prefijo viable** es un prefijo de una **forma sentencial derecha** que puede aparecer en la pila de un analizador LR.
- Dada una GLC G, llamamos **item LR(0)** o **item** de la gramática, a cualquier producción de G con un punto en la parte derecha de la regla.

  Intuitivamente, un *item* indica la porción de la regla que se ha usado o reconocido en un momento dado durante el proceso de análisis.
- Los *items* se agrupan en conjuntos que forman la **colección LR(0)** . Cada conjunto representa un estado de un AFD que **reconoce los prefijos viables** de G y **servirá para construir la tabla de análisis SLR**.
- Para construir la colección LR(0) necesitamos definir lo que se conoce como **gramática aumentada** , y las operaciones de **clausura** y **goto** .

Construcción de las tablas de análisis: tabla SLR

Gramática aumentada

Si G es una GLC con símbolo inicial S, la **gramática aumentada** G' para G se construye añadiendo la producción $S' \rightarrow S$, siendo el nuevo símbolo inicial S'.

Operación CLAUSURA

```
ConjuntoDeItems CLAUSURA(I){
    J=I;
    repeat
        for (cada elemento A→ α·Bβ  en J)
            for (cada producción B→ γ de G)
                if (B→ ·γ no está en J)
                    añadir B→ ·γ a J;
    until no se puedan añadir más elementos a J;
    return J;
}
```

Operación GOTO

```
ConjuntoDeItems GOTO(I,X){
    Inicializar J al conjunto vacío;
    for (cada ítem A→ α·Xβ en I)
        añadir A→ αX·β a J;
    return CLAUSURA(J);}
```

Construcción de las tablas de análisis: tabla SLR

Construcción de la colección LR(0)

```
ColeccionLR(0) ITEMS(G'){
    C=CLAUSURA({[S'→ ·S]});
    repeat
        for (cada conjunto de elementos I en C)
            for (cada símbolo gramatical X)
                if (GOTO(I,X) no está vacío y no está en C)
                    añadir GOTO(I,X) a C;
    until no se puedan añadir más conjuntos de elementos a C;
    return C;}
```

Colección LR(0) para la gramática de expresiones aritméticas

Gramática aumentada:

$$E' \to E$$
$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

◂ Items
◂ AFD
◂ TablaSLR
◂ Simul

Construcción de las tablas de análisis: tabla SLR

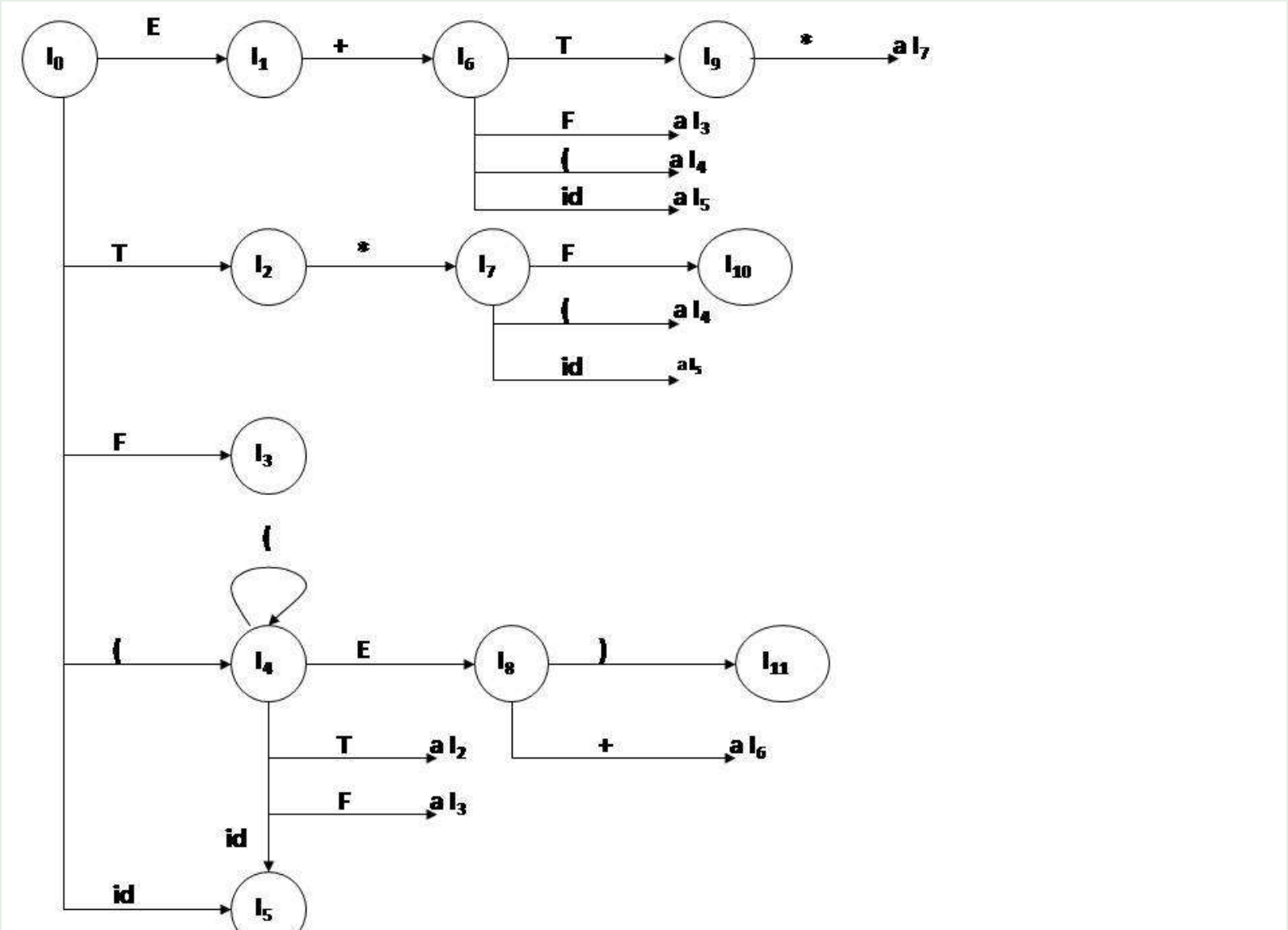
Colección LR(0) para la gramática de expresiones aritméticas

◂ GrEA

$CLAUSURA(\{[E' \to \cdot E]\}) = I_0 = \{E' \to \cdot E, E \to \cdot E + T, E \to \cdot T, T \to \cdot T * F,$
$T \to \cdot F, F \to \cdot(E), F \to \cdot id\}$
$GOTO(I_0, E) = I_1 = \{E' \to E\cdot, E \to E \cdot + T\}$
$GOTO(I_0, T) = I_2 = \{E \to T\cdot, T \to T \cdot * F\}$
$GOTO(I_0, F) = I_3 = \{T \to F\cdot\}$
$GOTO(I_0, () = I_4 = \{F \to (\cdot E), E \to \cdot E + T, E \to \cdot T, T \to \cdot T * F, T \to \cdot F,$
$F \to \cdot(E), F \to \cdot id\}$
$GOTO(I_0, id) = I_5 = \{F \to id\cdot\}$
$GOTO(I_1, +) = I_6 = \{E \to E + \cdot T, T \to \cdot T * F, T \to \cdot F, F \to \cdot(E), F \to \cdot id\}$
$GOTO(I_2, *) = I_7 = \{T \to T * \cdot F, F \to \cdot(E), F \to \cdot id\}$
$GOTO(I_4, E) = I_8 = \{F \to (E\cdot), E \to E \cdot + T\}$
$GOTO(I_4, T) = I_2$
$GOTO(I_4, F) = I_3$
$GOTO(I_4, () = I_4$
$GOTO(I_4, id) = I_5$
$GOTO(I_6, T) = I_9 = \{E \to E + T\cdot, T \to T \cdot * F\}$
$GOTO(I_6, F) = I_3$
$GOTO(I_6, () = I_4$
$GOTO(I_6, id) = I_5$
$GOTO(I_7, F) = I_{10} = \{T \to T * F\cdot\}$
$GOTO(I_7, () = I_4$
$GOTO(I_7, id) = I_5$
$GOTO(I_8,)) = I_{11} = \{F \to (E)\cdot\}$
$GOTO(I_8, +) = I_6$
$GOTO(I_9, *) = I_7$

Construcción de las tablas de análisis: tabla SLR

Diagrama de transiciones del AFD D para los prefijos viables de GrEA



Construcción de las tablas de análisis: tabla SLR

Item válido para un prefijo viable

Decimos que un **item** $A \to \beta_1 \cdot \beta_2$ es **válido** para un prefijo viable $\alpha\beta_1$, si existe una derivación:

$$S' \Rightarrow^*_{m.d.} \alpha A x \Rightarrow_{m.d.} \alpha\beta_1\beta_2 x$$

- → Si $A \to \beta_1 \cdot \beta_2$ es válido para $\alpha\beta_1$ indica si debemos desplazar o reducir cuando $\alpha\beta_1$ aparezca en la pila, según sea β_2 (si $\beta_2 \neq \lambda$ se desplaza y si $\beta_2 = \lambda$ se reduce).
- → El conjunto de items válidos para un prefijo viable γ, es el conjunto I_i de la colección LR canónica que corresponde al estado i del AFD. Este estado se alcanza partiendo del estado inicial y recorriendo el camino dado por los símbolos de γ.

Ejemplo

Observando el AFD anterior (◂ AFD) vemos que $E + T*$ es prefijo viable.
Partiendo de I_0, el AFD se encontrará en I_7 después de haber leído $E + T*$. I_7 contiene:

$$T \to T * \cdot F$$
$$F \to \cdot(E)$$
$$F \to \cdot id$$

que son los items válidos para $E + T*$:

$$E' \Rightarrow E \Rightarrow E + T \Rightarrow E + T * F$$

$$E' \Rightarrow E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * (E)$$

$$E' \Rightarrow E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * id$$

Construcción de las tablas de análisis: tabla SLR

Algoritmo **Construcción de una tabla de análisis sintáctico SLR**
Entrada Una gramática aumentada G'.
Salida Las funciones *acción* e *ir_a* de la tabla de análisis sintáctico SLR para G'.
Método

1. Constrúyase $C = \{I_0, I_1, ..., I_n\}$, la colección de conjuntos de elementos LR(0) para G', y el AFD que reconoce los prefijos viables.
2. El estado i se construye a partir de I_i. Las acciones de análisis sintácti para el estado i se determinan como sigue:
   a) Si $[A \to \alpha \cdot \ a\beta]$ está en I_i y $GOTO(I_i, a) = I_j$, entonces asígnese ''desplazar j'' a $accion[i, a]$. Aquí a debe ser un terminal.
   b) Si $[A \to \alpha\cdot]$ está en I_i, entonces asígnese ''reducir $A \to \alpha$'' a $accion[i,$ para todo a en $SIGUIENTE(A)$; aquí A no puede ser S'.
   c) Si $[S' \to \ S\cdot]$ está en I_i, entonces asígnese ''aceptar'' a $accion[i, \$]$.

Si las reglas anteriores generan acciones contradictorias, se dice que la gramática no es SLR(1). El algoritmo no consigue en este caso producir un AS.

3. Las transiciones *ir_a* para el estado i se construyen para todos los NT A utilizando la regla: si $GOTO(I_i, A) = I_j$, entonces $ir\_a[i, A] = j$.
4. Todas las entradas no definidas por las reglas 2 y 3 se consideran ''err
5. El estado inicial del analizador es el construido a partir del conjunto de elementos que contiene $[S' \to \cdot \ S]$.

Construcción de las tablas de análisis: tabla SLR

Construcción de la tabla SLR para la gramática de expresiones aritméticas

◂ GrEA

Tabla de análisis SLR:

ESTADO	accion						ir_a		
	id	+	*	(	)	$	E	T	F
0	$d5$			$d4$			1	2	3
1		$d6$				*aceptar*			
2		$r2$	$d7$		$r2$	$r2$			
3		$r4$	$r4$		$r4$	$r4$			
4	$d5$			$d4$			8	2	3
5		$r6$	$r6$		$r6$	$r6$			
6	$d5$			$d4$				9	3
7	$d5$			$d4$					10
8		$d6$			$d11$				
9		$r1$	$d7$		$r1$	$r1$			
10		$r3$	$r3$		$r3$	$r3$			
11		$r5$	$r5$		$r5$	$r5$			

d_i significa desplaza-estado i.
r_j significa reduce-producción nº j.

Construcción de las tablas de análisis: tabla SLR

Simulación del algoritmo LR para la gramática de expresiones aritméticas

◂ GrEA

Simulación del algoritmo LR con la tabla SLR ◂ TablaSLR y la entrada $id * id + id\$$:

PilaEstado	Entrada	Accion
0	$id * id + id\$$	$d5$
0 id 5	$*id + id\$$	$r6 : F \to id$
0 F 3	$*id + id\$$	$r4 : T \to F$
0 T 2	$*id + id\$$	$d7$
0 T 2 $*$ 7	$id + id\$$	$d5$
0 T 2 $*$ 7 id 5	$+id\$$	$r6 : F \to id$
0 T 2 $*$ 7 F 10	$+id\$$	$r3 : T \to T * F$
0 T 2	$+id\$$	$r2 : E \to T$
0 E 1	$+id\$$	$d6$
0 E 1 $+$ 6	$id\$$	$d5$
0 E 1 $+$ 6 id 5	$\$$	$r6 : F \to id$
0 E 1 $+$ 6 F 3	$\$$	$r4 : T \to F$
0 E 1 $+$ 6 T 9	$\$$	$r1 : E \to E + T$
0 E 1	$\$$	$aceptar$

Construcción de las tablas de análisis: tabla SLR

Una gramática SLR no puede ser ambigua, sin embargo, no toda gramática no ambigua es SLR.

Ejemplo

Gramática aumentada G':

$$S' \to S$$
$$(1)\ S \to L = R$$
$$(2)\ S \to R$$
$$(3)\ L \to * R$$
$$(4)\ L \to id$$
$$(5)\ R \to L$$

Construcción de las tablas de análisis: tabla LR-Canónica

El método LR-canónico es más general que el SLR, ya que en algunos casos resuelve algún conflicto que puede darse al intentar construir una tabla de análisis SLR. La razón es que maneja una información más precisa que ayuda a decidir cuándo se deben aplicar reducciones.

Definiciones básicas

- Llamamos **item LR(1)** de una gramática aumentada G', a un elemento de la forma $[A \to \alpha \cdot \beta, a]$ donde $A \to \alpha\beta \in P$ y $a \in V_T \cup \{\$\}$.
- Decimos que un **item LR(1)** $[A \to \beta_1 \cdot \beta_2, a]$ es **válido** para el prefijo viable $\alpha\beta_1$, si:

  $$S' \Rightarrow^*_{m.d.} \alpha A x \Rightarrow_{m.d.} \alpha\beta_1\beta_2 x,\ x \in V_T^*$$

  y (a es el primer símbolo de x) o (x es λ y $a = \$$).

Ahora, igual que se hizo antes, hay que construir una colección de ítems LR(1) válidos para cada prefijo viable de la gramática. Para ello se modifican ligeramente las definiciones de CLAUSURA y GOTO.

Construcción de las tablas de análisis: tabla LR-Canónica

Operación CLAUSURA

```
ConjuntoDeItems CLAUSURA(I){
          J=I;
    repeat
        for (cada elemento [A→ α·Bβ,a] en J)
            for (cada producción B→ γ en G´)
                for (cada terminal b en PRIMERO(βa)
                     tal que [B→ ·γ,b] no esté en J)
                            añadir [B→ ·γ,b] a J;
    until no se puedan añadir más elementos a J;
    return J;
}
```

Operación GOTO

```
ConjuntoDeItems GOTO(I,X){
    Inicializar J al conjunto vacío;
    for (cada ítem [A→ α·Xβ,a] en I)
        añadir [A→ αX·β,a] a J;
    return CLAUSURA(J);
}
```

Construcción de las tablas de análisis: tabla LR-Canónica

Construcción de la colección LR(1)

```
ColeccionLR(1) ITEMS(G´){
    C=CLAUSURA({[S'→·S,$]});
    repeat
        for (cada conjunto de elementos I en C)
            for (cada símbolo gramatical X)
                if (GOTO(I,X) no está vacío y no está en C)
                    añadir GOTO(I,X) a C;
    until no se puedan añadir más conjuntos de elementos a C;
    return C;
}
```

Construcción de las tablas de análisis: tabla LR-Canónica

Algoritmo Construcción de la tabla de análisis sintáctico LR-canónico
Entrada Gramática aumentada G'.
Salida Tabla de análisis *acción* e *ir_a* para G'.
Método

1. Constrúyase la colección LR(1), $C = \{I_0, I_1, ..., I_n\}$, y el AFD.
2. Constrúyase el estado i del analizador a partir de I_i de la siguiente form
   (parte de *acción*):
   a) Si $[A \to \alpha \cdot \ a\beta, b] \in I_i$, $GOTO(I_i, a) = I_j$, $a \in V_T$, entonces asignar ''desplazar j'' a $accion[i, a]$.
   b) Si $[A \to \alpha\cdot, a] \in I_i$, $A \neq S'$, entonces asignar ''reducir $A \to \alpha$'' a $accion[i, a]$.
   c) Si $[S' \to S\cdot, \$] \in I_i$, entonces asignar ''aceptar'' a $accion[i, \$]$.

Si no hay conflictos, la gramática será LR(1).
En otro caso, el algoritmo falla.

3. Determinar las transiciones para el estado i, de la siguiente forma
   (parte *ir_a*):
   Si $GOTO(I_i, A) = I_j$, entonces $ir\_a[i, A] = j$.
4. Las entradas no definidas en 2 y 3 serán entradas de ''error''.
5. El estado inicial del analizador es 0, construido a partir del conjunto d
   ítems que contiene a $[S' \to \cdot\ S, \$]$.

Construcción de las tablas de análisis: tabla LR-Canónica

Cáculo de la colección LR(1)

Sea la gramática aumentada:

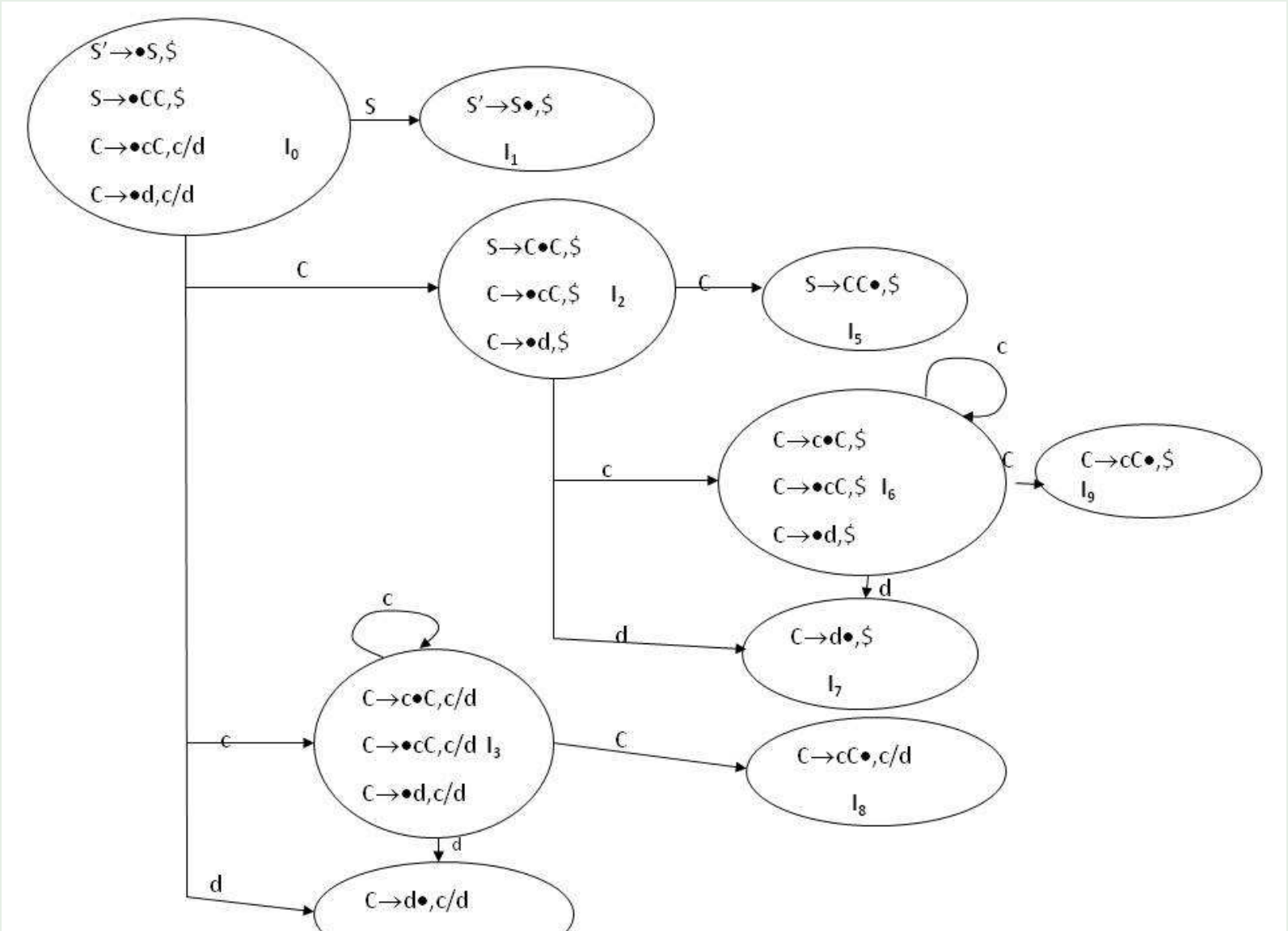
$$S' \to S$$
$$S \to C\ C$$
$$C \to c\ C$$
$$C \to d$$

◂ AFD-LR-C ◂ AFD-LALR ◂ Tabla-LR-C ◂ Tabla-LALR

$$I_0 = \{[S' \to \cdot S, \$], [S \to \cdot CC, \$], [C \to \cdot cC, c/d], [C \to \cdot d, c/d]\}$$
$$GOTO(I_0, S) = I_1 = \{[S' \to S\cdot, \$]\}$$
$$GOTO(I_0, C) = I_2 = \{[S \to C \cdot C, \$], [C \to \cdot cC, \$], [C \to \cdot d, \$]\}$$
$$GOTO(I_0, c) = I_3 = \{[C \to c \cdot C, c/d], [C \to \cdot cC, c/d], [C \to \cdot d, c/d]\}$$
$$GOTO(I_0, d) = I_4 = \{[C \to d\cdot, c/d]\}$$
$$GOTO(I_2, C) = I_5 = \{[S \to CC\cdot, \$]\}$$
$$GOTO(I_2, c) = I_6 = \{[C \to c \cdot C, \$], [C \to \cdot cC, \$], [C \to \cdot d, \$]\}$$
$$GOTO(I_2, d) = I_7 = \{[C \to d\cdot, \$]\}$$
$$GOTO(I_3, C) = I_8 = \{[C \to cC\cdot, c/d]\}$$
$$GOTO(I_3, c) = I_3$$
$$GOTO(I_3, d) = I_4$$
$$GOTO(I_6, C) = I_9 = \{[C \to cC\cdot, \$]\}$$
$$GOTO(I_6, c) = I_6$$
$$GOTO(I_6, d) = I_7$$

Construcción de las tablas de análisis: tabla LR-Canónica

Diagrama de transiciones del AFD D para los prefijos viables de GrCC



Construcción de las tablas de análisis: tabla LR-Canónica

El AFD se construye como en el método SLR. Si tenemos un prefijo viable γ, partiendo del estado inicial I_0, y recorriendo los arcos del AFD etiquetados con los símbolos de γ, se llegaría a un estado I_i, que contiene los items LR(1) válidos para dicho prefijo γ.

Tabla de análisis LR-canónica para la gramática

◂ GrCC

ESTADO	accion			ir_a	
	c	d	$	S	C
0	*d*3	*d*4		1	2
1			*aceptar*		
2	*d*6	*d*7			5
3	*d*3	*d*4			8
4	*r*3	*r*3			
5			*r*1		
6	*d*6	*d*7			9
7			*r*3		
8	*r*2	*r*2			
9			*r*2		

Construcción de las tablas de análisis: tabla LALR

→ El **método LALR** es **menos potente que** el **LR-canónico** y **más potente que** el **SLR**.

→ **El tamaño de la tabla de análisis LALR es igual que el de una SLR**. Esto hace que sea un método muy utilizado en la práctica.

Tabla LALR

- **Para construir una tabla de análisis LALR** partimos de la colección de conjuntos de items LR(1) y del AFD, como en el LR-canónico. **Agrupamos los estados con las mismas producciones punteadas en los items y que sólo difieren en los símbolos de anticipación**.
- Modificando las transiciones de forma conveniente, se llegaría a otro *AFD simplificado*, a partir del cual se construye la tabla LALR, exactamente igual que la tabla LR-canónica.
- Los nuevos estados del AFD constituyen la **colección LALR** .
- Si la tabla resultante no contiene conflictos, la **gramática** es **LALR** .

Construcción de las tablas de análisis: tabla LALR

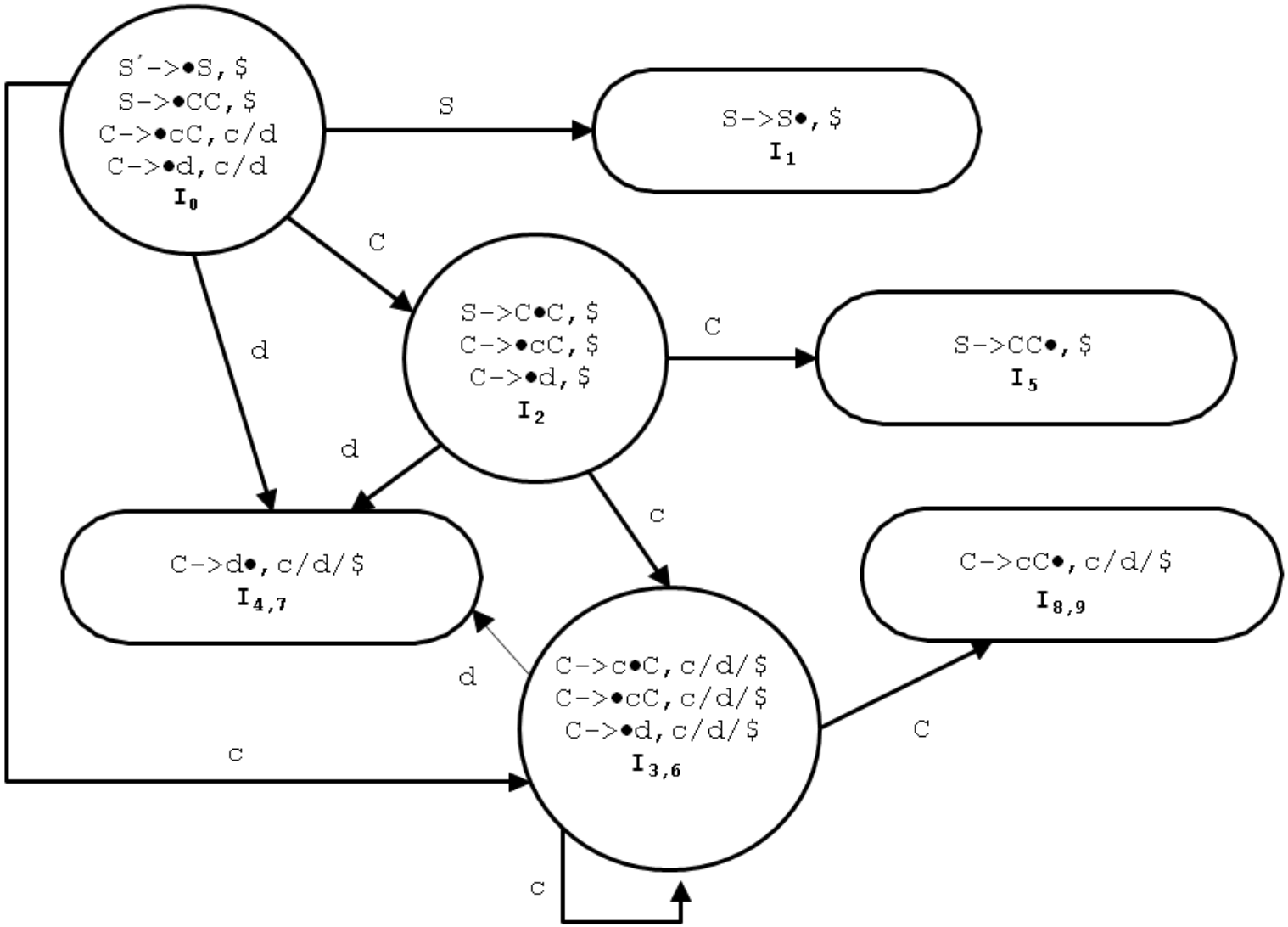
Algoritmo Construcción de la tabla de análisis sintáctico LALR
Entrada Gramática aumentada G'.
Salida Tabla de análisis *acción* e *ir_a* para G'.
Método

1. Constrúyase la colección LR(1), $C = \{I_0, I_1, ..., I_n\}$, y el AFD.
2. Encuéntrense los conjuntos de ítems en los que los primeros componentes de todos sus items coinciden, y únanse en un único estado que sustituya a los conjuntos coincidentes.
3. Sea $C = \{J_0, J_1, \ldots, J_m\}$ la nueva colección de ítems obtenida.
   Construir las acciones para el estado i a partir del conjunto J_i como para la tabla LR-canónica.
   Si no existe ningún conflicto en las acciones, la gramática es $LALR(1)$.
   En caso contrario, el algoritmo no consigue producir un analizador sintáctico.
4. Las transiciones *ir_a* para el estado i se determinan de la siguiente form
   Si $J_i = I_1 \cup I_2 \cup \ldots I_k$, entonces $Goto(I_1, X)$, $Goto(I_2, X)$, ..., $Goto(I_k, X)$ son estados de $LR(1)$ coincidentes en el primer componente de todos sus ítems.
   Sea J_t el estado resultante de la fusión de estos estados en el paso 3.
   Entonces definimos $Goto(J_i, X) = J_t$.

Construcción de las tablas de análisis: tabla LALR

Colección LALR y AFD para la gramática

◂ GrCC



Construcción de las tablas de análisis: tabla LALR

Tabla LALR para la gramática

◂ GrCC

ESTADO	accion			ir_a	
	c	d	$	S	C
0	*d*36	*d*47		1	2
1			*aceptar*		
2	*d*36	*d*47			5
36	*d*36	*d*47			89
47	*r*3	*r*3	*r*3		
5			*r*1		
89	*r*2	*r*2	*r*2		

Construcción de las tablas de análisis: tabla LALR

Conflictos en tablas LALR

Puede que tengamos una **gramática LR-canónica para la que el método LALR produce conflictos en la tabla de análisis**. Se puede comprobar que **no se originan conflictos shift-reduce** en el método LALR **si la gramática es LR-canónica**, pero **sí pueden aparecer nuevos conflictos reduce-reduce**.

Gramática que es LR-canónica pero no LALR

$$
\begin{aligned}
&S' \to S \\
&S \to aAd \mid bBd \mid aBe \mid bAe \\
&A \to c \\
&B \to c
\end{aligned}
$$

Gramática que es LALR pero no SLR

$$
\begin{aligned}
&S' \to S \\
&S \to L = R \mid R \\
&L \to *R \mid id \\
&R \to L
\end{aligned}
$$

Ambigüedad en el análisis LR

Una gramática ambigua no puede ser LR. Por tanto, para analizar por el método LR el lenguaje generado por ella, debemos eliminar la ambigüedad para obtener una gramática equivalente, y después comprobar si es LR.

Ejemplo

La gramática

$$S \to iCtS \mid iCtSeS \mid a$$
$$C \to p$$

se puede transformar para que no sea ambigua:

$$S \to S_1 \mid S_2$$
$$S_1 \to iCtS_1eS_1 \mid a$$
$$S_2 \to iCtS \mid iCtS_1eS_2$$
$$C \to p$$

- → Esta gramática, aunque no es LL(1), es LR.
- → Sin embargo, la tabla que se obtiene a partir de ella es demasiado extensa.
- → Una solución es partir de la gramática ambigua y tratar de resolver el conflicto shift-reduce que se produce en la tabla resultante.

Ambigüedad en el análisis LR

Colección LR(0), tabla SLR y simulación con la gramática del `if` anidado

En lugar de utilizar la gramática de partida, utilizamos una versión **simplificada** y **aumentada**:

$$S' \to S \qquad S \to iSeS \mid iS \mid a$$

Colección LR(0):

$I_0 = \{S' \to \cdot S, S \to \cdot iSeS, S \to \cdot iS, S \to \cdot a\}$
$I_1 = \{S' \to S\cdot\}$
$I_2 = \{S \to i \cdot SeS, S \to i \cdot S, S \to \cdot iSeS, S \to \cdot iS, S \to \cdot a\}$
$I_3 = \{S \to a\cdot\}$
$I_4 = \{S \to iS \cdot eS, S \to iS\cdot\}$
$I_5 = \{S \to iSe \cdot S, S \to \cdot iSeS, S \to \cdot iS, S \to \cdot a\}$
$I_6 = \{S \to iSeS\cdot\}$

Tabla SLR:

ESTADO	accion				ir_a
	i	e	a	$	S
0	d2		d3		1
1				aceptar	
2	d2		d3		4
3		r3		r3	
4		d5/r2		r2	
5	d2		d3		6
6		r1		r1	

Ejecución para la entrada *iiaea*$:

PilaEstado	Entrada
0	*iiaea*$
0*i*2	*iaea*$
0*i*2*i*2	*aea*$
0*i*2*i*2*a*3	*ea*$
0*i*2*i*2*S*4	*ea*$
0*i*2*i*2*S*4*e*5	*a*$
0*i*2*i*2*S*4*e*5*a*3	$
0*i*2*i*2*S*4*e*5*S*6	$
0*i*2*S*4	$
0*S*1	$

Tratamiento y recuperación de errores en el análisis LR

- Se detecta un error en el análisis LR cuando se consulta en la tabla de análisis una casilla de acción que está vacía.
- Cualquiera de los métodos LR es capaz de detectar un error antes de meter el símbolo erróneo en la pila.
- Con el **método LR-canónico** el error se detecta inmediatamente. Con los **métodos LALR o SLR** pueden hacerse varias reducciones antes de detectar el error, pero nunca desplazarán un símbolo erróneo a la pila.

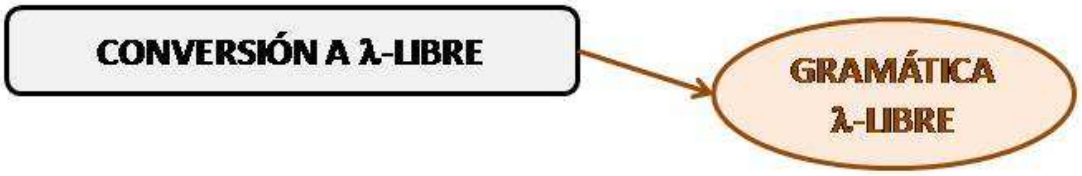
Técnicas

Con el tratamiento **a nivel frase** , se rellenan las casillas vacías de la tabla con llamadas a rutinas de error, aunque es más fácil corregir los errores **en "modo pánico"** , saltando los símbolos hasta que se pueda seguir con el análisis:

- Sacar símbolos de la pila hasta que aparece en el tope un estado s, tal que $ir\_a[s, A] = s'$ para alguna variable A.
- Introducir As' en la pila.
- Descartar símbolos de entrada hasta encontrar un a seguidor legal de A.

Algoritmos de transformación de gramáticas

Consultar los algoritmos en los apuntes de la asignatura.



Eliminación de la recursividad izquierda inmediata

Entrada: una gramática λ-libre $G = (V_N, V_T, S, P)$ con una variable A recursiva por la izquierda.
Salida: una gramática equivalente $G' = (V'_N, V_T, S, P')$ sin recursividad izquierda en A.

1. Ordenar las producciones de A:
   $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$;
2. Añadir una variable nueva A' a V_N;
3. Reemplazar en P las A-producciones por las siguientes nuevas reglas de producción:
   $A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \mid \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$;
   $A' \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A'$;
4. Devolver (G'), con los componentes de G modificados por los pasos anteriores.

Eliminación de la recursividad izquierda

Entrada: una gramática $G = (V_N, V_T, S, P)$ propia.
Salida: una gramática equivalente $G' = (V'_N, V_T, S, P')$ sin recursividad por la izquierda.

1. Ordenar los no terminales $V_N = \{A_1, A_2, \ldots, A_n\}$, donde $A_1 = S$.
2. Inicializar $V'_N \leftarrow V_N$ y $P' \leftarrow P$;
3. REPETIR desde $i = 1$ HASTA n:
   - 3.1. Si hay recursión inmediata en las reglas para A_i entonces:
     - 3.1.1 $V'_N \leftarrow V'_N \cup \{A'_i\}$;
     - 3.1.2 Sustituir $A_i \rightarrow A_i\alpha_1 \mid A_i\alpha_2 \mid \ldots \mid A_i\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ en P' por:
       $A_i \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \mid \beta_1 A'_i \mid \beta_2 A'_i \mid \ldots \mid \beta_n A'_i$;
       $A'_i \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A'_i \mid \alpha_2 A'_i \mid \ldots \mid \alpha_m A'_i$;
   - 3.2. Si $i < n$ entonces REPETIR desde $j = 1$ HASTA i:
     - 3.2.1. Sustituir cada producción $A_{i+1} \rightarrow A_j\alpha$ en P' por:
       $A_{i+1} \rightarrow \gamma_1\alpha \mid \gamma_2\alpha \mid \ldots \mid \gamma_p\alpha$;
       donde $A_j \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_p$.
4. Devolver (G'), con los componentes de V'_N y P' modificados por los pasos anteriores.

Factorización izquierda

Entrada: una gramática libre de contexto $G = (V_N, V_T, S, P)$.
Salida: una gramática equivalente $G' = (V'_N, V_T, S, P')$ sin factores comunes por la izquierda en las reglas de producción.

1. REPETIR por cada no terminal $A \in V_N$
   - 1.1. Encontrar el prefijo α más largo común a dos o más de sus alternativas:
     $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$;
     donde cada γ_i $(1 \leq i \leq k)$ representa una alternativa que no comienza con α.
   - 1.2. Si $\alpha \neq \lambda$:
     - 1.2.1. Añadir un nuevo no terminal A' a V_N;
     - 1.2.2. Sustituir en P las A-producciones por estas otras:
       $A \to \alpha A' \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$;
       $A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$;

   HASTA que no haya dos o más alternativas para A con un prefijo común $\alpha \neq \lambda$.
2. Devolver (G'), con los componentes de G modificados por los pasos anteriores.

Análisis descendente predictivo

Los **métodos generales**, válidos para cualquier GLC, tienen **inconvenientes**:

- Son muy *costosos en tiempo*.
- No son adecuados para corregir *errores sintácticos*.

Los métodos de **análisis descendente predictivo** se aplican para analizar sentencias generadas por **gramáticas LL(1)** . Tienen importantes **ventajas**:

- El algoritmo tiene *orden de complejidad lineal*.
- *Analiza de forma determinista una sentencia examinando sólo una vez, de izquierda a derecha, la secuencia de tokens*.

Para no hacer retroceso será necesario...

...dado un terminal de entrada a y una variable A de la gramática que debe ser expandida, conocer, en cada momento, cuál de las alternativas para las A-producciones:

$$A \rightarrow \alpha_1|\alpha_2|...|\alpha_k$$

debe aplicarse.

Gramáticas LL(1)

Significado de la notación LL(1)

L : La entrada se examina **de izquierda a derecha** (*left to right*).

L : Se realiza la **derivación más a la izquierda** (*leftmost derivation*).

1 : Sólo se necesita examinar **un token de anticipación** para decidir qué regla aplicar.

Cuando se necesitan k símbolos de anticipación, la gramática se llama **LL(k)** .

→ El **saber si un lenguaje es LL(1)** es un **problema indecidible**.

→ Transformaciones sobre gramáticas que **a veces** conducen a gramáticas equivalentes LL(1):

- **Eliminar la ambigüedad**, si se puede.
- **Eliminar la recursividad por la izquierda**.
- **Factorizar** la gramática por la izquierda.

Construcción de la tabla de análisis en el ADPNR

Función predict

Para cada producción de una g.l.c. $G = (V_N, V_T, S, P)$:
$predict(A \to \alpha) = if\ \lambda \in PRIMERO(\alpha)$
$\quad then\ (PRIMERO(\alpha) - \{\lambda\} \cup\ SIGUIENTE(A))$
$\quad else\ PRIMERO(\alpha).$

Tabla de análisis

$T : V_N \times V_T \cup \{\$\} \to P(P)$
Cada casilla de la tabla se forma de la siguiente manera:
$T[A, a] = \{A \to \alpha / a \in predict(A \to \alpha)\}\ \forall A \in V_N, a \in V_T.$

- $\to$ La tabla indica la producción que debemos usar en un paso de derivación, donde tiene que expandirse la variable A y el token actual es a.
- $\to$ Si ninguna casilla tiene más de una regla de producción, se dice que la **gramática** es **LL(1)** .

Construcción de la tabla de análisis en el ADPNR

Gramática LL(1)

Una **g.l.c.** es **LL(1)** sii $\forall A \in V_N$, si existen producciones $A \to \alpha$ y $A \to \gamma$, entonces $predict(A \to \alpha) \cap predict(A \to \gamma) = \emptyset$.

Hay ocasiones en que **eliminando la recursividad por la izquierda**, **la ambigüedad** y **factorizando**, no podemos conseguir una gramática LL(1).

Ejemplo

$$\begin{array}{lcl} S & \to & i\ C\ t\ S \mid i\ C\ t\ S\ e\ S \mid a \mid b \\ C & \to & p \mid q \end{array}$$

Construcción de la tabla de análisis en el ADPNR

Algoritmo **Construcción de una tabla de análisis sintáctico predictivo**

Entrada Una gramática G.

Salida La tabla de análisis sintáctico M.

Método

Para cada producción $A \to \alpha$ de la gramática, dense los pasos:

1. Calcular el conjunto $predict(A \to \alpha)$.
2. Para cada terminal $a \in predict(A \to \alpha)$, añádase $A \to \alpha$ a $M[A, a]$.

Hágase que cada entrada no definida de M sea *error*.

Ejemplo de construcción de la tabla

Cálculo de PRIMERO y SIGUIENTE

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \lambda \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \lambda \\
F & \rightarrow & (\ E\ ) \mid id
\end{array}
$$

$PRIMERO(E) = PRIMERO(T) = PRIMERO(F) = \{(, id\}$
$PRIMERO(E') = \{+, \lambda\}$
$PRIMERO(T') = \{*, \lambda\}$
$SIGUIENTE(E) = SIGUIENTE(E') = \{), \$\}$
$SIGUIENTE(T) = SIGUIENTE(T') = \{+,), \$\}$
$SIGUIENTE(F) = \{+, *,), \$\}$

Construcción de la tabla de análisis en el ADPNR

Ejemplo de construcción de una tabla de análisis

Continuamos con la gramática LL(1) siguiente:

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \lambda \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \lambda \\
F & \rightarrow & (\ E\ ) \mid id
\end{array}
$$

La tabla de análisis sintáctico M para esta gramática sería:

NO TERM	**TERMINAL**					
	id	$+$	$*$	$($	$)$	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E′		$E' \rightarrow +TE'$			$E' \rightarrow \lambda$	$E' \rightarrow \lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T′		$T' \rightarrow \lambda$	$T' \rightarrow *FT'$		$T' \rightarrow \lambda$	$T' \rightarrow \lambda$
F	$F \rightarrow id$			$F \rightarrow (E)$		

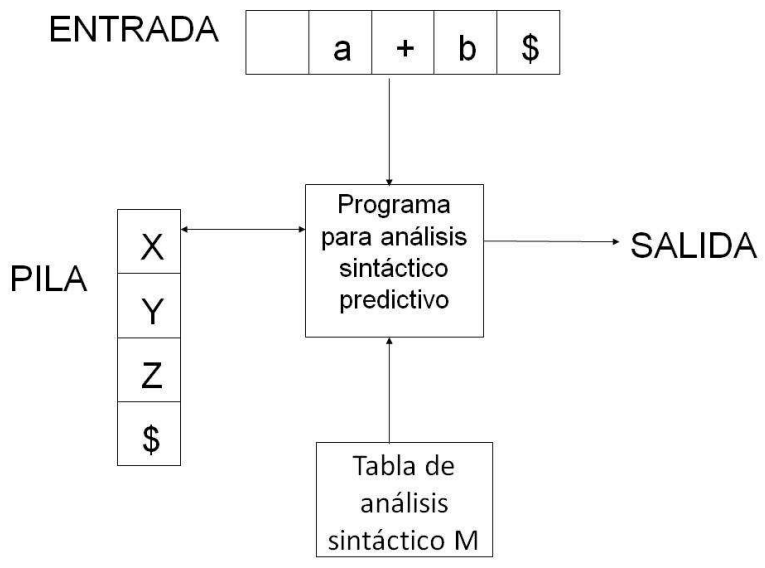
◂ EjSimul

Algoritmo para el ADPNR

Algoritmo Análisis sintáctico predictivo no recursivo
Entrada Una cadena w y una tabla de análisis sintáctico M para $G = (V_N, V_T, S, P)$.
Salida Si $w \in L(G)$, una *derivación por la izquierda* de w; si no, una indicación de *error*.
Método Inicialmente, el AS está en una configuración en la que tiene a $\$S$ en la pila con S en el tope, y $w\$$ en el buffer de entrada.

◂ ADPNRE

```
Hacer que ae apunte al primer símbolo de w$
        y asignar a a;
Asignar a X el símbolo del tope de la pila;
while (X≠$){ /* pila no vacía */
    if (X==a) sacar tope de pila y avanzar ae
    else if (X es terminal)  error1();
    else if (M[X,a] está vacía)  error2(X);
    else if (M[X,a]=X→Y1Y2...Yk) {
                Extraer tope de pila;
                Meter Yk,Yk−1,...,Y1 en la
                    pila, con Y1 en la cima;
                Emitir la producción
                    X→Y1Y2...Yk
    }
    Asignar a X el símbolo del tope de la
    pila y a a el símbolo apuntado por ae;
}
```



Algoritmo para el ADPNR

Orden de complejidad

Teorema Es $\bigcirc(n)$ en tiempo y en espacio, donde n es la longitud de la cadena de entrada.

Ejemplo: Aplicación del algoritmo con la sentencia $id + id * id$

Continuamos con el ejemplo ◂ EjTabla para la gramática:

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \lambda \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \lambda \\
F & \rightarrow & (\ E\ ) \mid id
\end{array}
$$

Los movimientos realizados por el analizador LL serían los siguientes:

PILA	ENTRADA	SALIDA
$\$E$	$id + id * id\$$	
$\$E'T$	$id + id * id\$$	$E \rightarrow TE'$
$\$E'T'F$	$id + id * id\$$	$T \rightarrow FT'$
$\$E'T'id$	$id + id * id\$$	$F \rightarrow id$
$\$E'T'$	$+id * id\$$	
$\$E'$	$+id * id\$$	$T' \rightarrow \lambda$
$\$E'T+$	$+id * id\$$	$E' \rightarrow +TE'$
$\$E'T$	$id * id\$$	
$\$E'T'F$	$id * id\$$	$T \rightarrow FT'$
$\$E'T'id$	$id * id\$$	$F \rightarrow id$
$\$E'T'$	$*id\$$	
$\$E'T'F*$	$*id\$$	$T' \rightarrow *FT'$
$\$E'T'F$	$id\$$	
$\$E'T'id$	$id\$$	$F \rightarrow id$
$\$E'T'$	$\$$	
$\$E'$	$\$$	$T' \rightarrow \lambda$
$\$$	$\$$	$E' \rightarrow \lambda$

Ambigüedad en el ADPNR

Ejemplo

$$
\begin{array}{lcl}
S & \rightarrow & i\ C\ t\ S \mid i\ C\ t\ S\ e\ S \mid a \mid b \\
C & \rightarrow & p \mid q
\end{array}
$$

Eliminando la ambigüedad:

$$
\begin{array}{lcl}
S & \rightarrow & S_1 \mid S_2 \\
S_1 & \rightarrow & i\ C\ t\ S_1\ e\ S_1 \mid a \mid b \\
S_2 & \rightarrow & i\ C\ t\ S \mid i\ C\ t\ S_1\ e\ S_2 \\
C & \rightarrow & p \mid q
\end{array}
$$

Factorizando:

$$
\begin{array}{lcl}
S & \rightarrow & S_1 \mid S_2 \\
S_1 & \rightarrow & i\ C\ t\ S_1\ e\ S_1 \mid a \mid b \\
S_2 & \rightarrow & i\ C\ t\ S_2' \\
S_2' & \rightarrow & S \mid S_1\ e\ S_2 \\
C & \rightarrow & p \mid q
\end{array}
$$

La gramática sigue sin ser LL(1)!!!!

Tabla de análisis:

NT	TERMINAL							
	i	**t**	**e**	**a**	**b**	**p**	**q**	**$**
S	$S \rightarrow S_1$ $S \rightarrow S_2$			$S \rightarrow S_1$	$S \rightarrow S_1$			
S_1	$S_1 \rightarrow i\ C\ t\ S_1\ e\ S_1$			$S_1 \rightarrow a$	$S_1 \rightarrow b$			
S_2	$S_2 \rightarrow i\ C\ t\ S_2'$							
S_2'	$S_2' \rightarrow S$ $S_2' \rightarrow S_1\ e\ S_2$			$S_2' \rightarrow S$ $S_2' \rightarrow S_1\ e\ S_2$	$S_2' \rightarrow S$ $S_2' \rightarrow S_1\ e\ S_2$			
C						$C \rightarrow p$	$C \rightarrow q$	

Ambigüedad en el ADPNR

En casos como el anterior puede utilizarse una tabla no predictiva con ciertos convenios.

Ejemplo

Factorizamos la gramática original anterior sin quitarle la ambigüedad:

$$S \to i\ C\ t\ S\ S' \mid a \mid b$$
$$S' \to e\ S \mid \lambda$$
$$C \to p \mid q$$

Esta gramática sigue siendo ambigua (comprobarlo con la sentencia `i p t i q t a e b`).
Aún así construimos la **tabla de análisis**: $SIGUIENTE(S') = SIGUIENTE(S) = \{\$, e\}$

NT	TERMINAL							
	i	**t**	**e**	**a**	**b**	**p**	**q**	**$**
S	$S \to i\ C\ t\ S\ S'$			$S \to a$	$S \to b$			
S′			$S' \to e\ S$ $S' \to \lambda$					$S' \to \lambda$
C						$C \to p$	$C \to q$	

Por convenio, en $T[S', e]$ elegimos $S' \to eS$ (*else* asociado al *if* más próximo).

En general, no hay forma sistemática de elegir una única regla sin alterar el lenguaje.

Análisis descendente predictivo recursivo

Método recursivo

Consiste en hacer el análisis usando un procedimiento recursivo para cada variable de la gramática.
Este procedimiento se encargará de reconocer cualquier secuencia de caracteres que pueda ser derivada a partir de esa variable.

Dentro del procedimiento se podrán leer o saltar símbolos de la entrada como en el no recursivo, y se llamará a otros procedimientos.
Puede generalizarse así:

◂ ADPRE

```
void coincide (terminal t){
    if (preanalisis==t) preanalisis=sigTerminal;
    else error1()
}
void A() {
    Sea A→ α1 | α2 |...| αn;
    if (preanalisis∈predict(αi)) para algún i tal que αi ≡X1X2...Xk
        for (i=1 to k){
            if (Xi ∈VN)  Xi();
            else if (Xi ∈VT) coincide(Xi);
            }
    else error2(A);
}
```

Análisis descendente predictivo recursivo

Ejemplo

$$tipo \rightarrow simple \mid \uparrow\ id \mid array\ [\ simple\ ]\ of\ tipo$$
$$simple \rightarrow integer \mid char \mid num\ puntopunto\ num$$

```
void tipo(){
    if (preanalisis∈{‘integer’,‘char’,‘num’}) simple();
    else if (preanalisis==‘↑’) {
            coincide(‘↑’); coincide(‘id’);
        }
    else if (preanalisis==‘array’) {
            coincide(‘array’);coincide(‘[’);simple();coincide(‘]’);
            coincide(‘of’);tipo();
        }
    else error2(‘tipo’);
}
void simple(){
    if (preanalisis==‘integer’) coincide(‘integer’);
    else if (preanalisis==‘char’) coincide(‘char’);
    else if (preanalisis==‘num’) {
            coincide(‘num’); coincide(‘puntopunto’);coincide(‘num’);
            }
    else error2(‘simple’);
}
```

Manejo de errores en el análisis predictivo

Los analizadores LL(1) son capaces de detectar un error sintáctico en cuanto se produce, y pueden repararlo sin hacer retroceso en la entrada.

Recuperación de errores en el ADP no recursivo

◂ ADPNR

- **Los errores** en el ADP **se detectan** en uno de estos casos:
  - El terminal del tope de la pila no coincide con el token de la entrada (`error1`).
  - En la pila hay un NT a expandir, y en la tabla no aparece ninguna producción aplicable para esa variable y el símbolo de entrada (`error2`).
- **Para intentar reparar los errores** se suelen aplicar dos **técnicas**:
  1. **Tratamiento de errores a nivel de frase** : Se rellenan las casillas de error de la tabla con llamadas a rutinas de error, que indican el tipo de error e intentan repararlo. Puede ser complicado.
  2. **Tratamiento de errores en modo "pánico"** : Es más sencillo y sistemático. Se saltan símbolos de la entrada y/o de la pila, hasta que el análisis pueda proseguir. Pueden ocurrir distintos casos:
     * $T[A, t] = \emptyset$ Descartamos símbolos en la entrada hasta encontrar un a tal que, o bien
       $\rightarrow a \in PRIMERO(A) \Rightarrow$ continuamos, o bien
       $\rightarrow a \in SIGUIENTE(A) \Rightarrow$ sacamos A de la pila y continuamos.
     * Que no coincida el terminal de la pila con el de entrada:
       $\rightarrow$Extraemos el terminal de la pila (inserción en la entrada).

Manejo de errores en el análisis predictivo

Recuperación de errores en el ADP recursivo

La **recuperación de errores en modo pánico** , equivalente a la del análisis no recursivo, podría realizarse implementando los procedimientos de error `error1` y `error2` de ◂ ADPR:

- `error1()` Se informa sin hacer nada, y esto equivaldría a sacar el terminal de la pila (o insertarlo en la entrada).
- `error2(`$X \in V_N$`)` Saltamos símbolos de la entrada hasta encontrar uno que pertenezca:
  - al conjunto PRIMERO de X: volvemos a realizar la llamada a X().
    o bien
  - al conjunto SIGUIENTE de X: salimos del procedimiento sin hacer nada.

Métodos generales de análisis sintáctico

→ Son métodos de AS que permiten analizar cualquier GLC.

→ A veces es necesario transformar previamente la gramática.

→ Los métodos generales son poco usados por su falta de eficiencia.

→ Algunos de ellos son:

- **Método de Cocke-Younger-Kasami (CYK)** Requiere que G esté en *Forma Normal de Chomsky*. Tiene $\bigcirc(n^3)$ en tiempo y $\bigcirc(n^2)$ en espacio.
- **Método de Earley** Para uno de sus algoritmos requiere que G *no tenga ciclos*. Calcula $w \in L(G)$ en $\bigcirc(n^3)$ en tiempo ($\bigcirc(n^2)$ si G no es ambigua) y $\bigcirc(n^2)$ en espacio.
- **Análisis general descendente con retroceso** Requiere que G sea *no recursiva por la izquierda*. Tiene orden de complejidad lineal en espacio y exponencial en el tiempo, en función de $|w|$.
- **Análisis general ascendente con retroceso** Requiere que G sea λ-*libre y libre de ciclos*. Tiene orden de complejidad lineal en espacio y exponencial en tiempo, en función de $|w|$.