

Compiladores e Intérpretes

3

Análisis Sintáctico



Paulo Félix Lamas

Área de Ciencias da Computación e Intelixencia Artificial

Departamento de Electrónica e Computación

[Índice]

Introducción

Objetivos

1. Nociones generales.
2. Diseño de una gramática.
3. Análisis sintáctico descendente.
4. Análisis sintáctico ascendente.

Bibliografía

[Introducción]

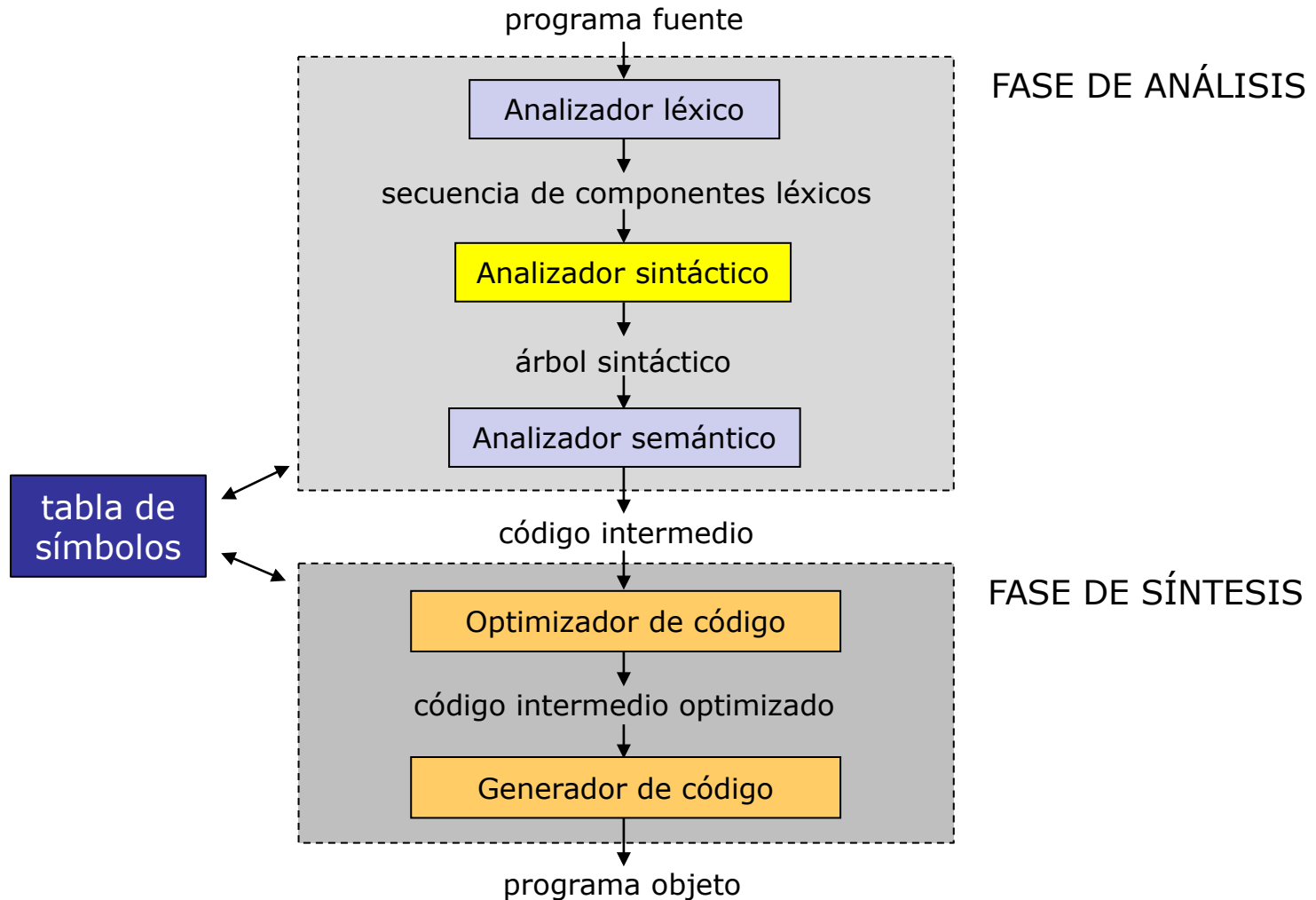
- En este tema estudiaremos la segunda fase del proceso de compilación: el **análisis sintáctico** del programa fuente.
- Una **gramática** es la herramienta formal que nos ayudará a comprobar la estructura de un programa.
- Dividimos el tema en **tres bloques**:
 - En el primer bloque daremos una visión general del analizador sintáctico, y algunas de las cuestiones fundamentales a la hora de diseñar gramáticas: **recursividad**, **ambigüedad**, **asociatividad** y **precedencia**.
 - En el segundo bloque estudiaremos el **análisis sintáctico descendente**.
 - En el tercer bloque estudiaremos el **análisis sintáctico ascendente**.

[Objetivos]

1. Saber definir una gramática utilizando la notación BNF.
2. Conocer y saber tratar los efectos de la recursividad y la ambigüedad en los lenguajes de programación.
3. Saber construir las producciones de gramáticas que incorporen asociatividad y precedencia en las operaciones.
4. Ser capaz de detectar y eliminar la recursividad izquierda de gramáticas independientes del contexto.
5. Ser capaz de diseñar gramáticas LL(1).
6. Ser capaz de diseñar gramáticas SLR(1).
7. Conocer las técnicas que se pueden utilizar para que el analizador sintáctico se recupere de un estado de error, y continúe el análisis del código fuente.

1. [Nociones generales.]

Estructura



1. [Nociones generales.]

Gramáticas independientes del contexto

- Una **gramática independiente del contexto** se define como una 4-tupla $G=(\Sigma,V,S,P)$, siendo:
 - Σ un alfabeto o conjunto de **símbolos terminales**.
 - V un conjunto de variables o **símbolos no terminales**.
 - S una variable denominada **símbolo inicial**.
 - P una colección de **reglas de sustitución**, llamadas reglas de producción, de la forma $A \rightarrow \alpha$, donde $A \in V$, y $\alpha \in (V \cup \Sigma)^*$.
- La máquina apropiada para el reconocimiento de lenguajes independientes del contexto es el **autómata a pila**.
- En este tema introduciremos el símbolo \$ para indicar el **fin de cadena**.

1. [Nociones generales.]

Tipos de analizadores sintácticos

- Un analizador sintáctico analiza el código fuente como una secuencia de componentes léxicos (símbolos terminales), y construye una representación interna en forma de árbol sintáctico. Podemos hablar de dos tipos de analizadores:
 1. **Descendentes**: parten de la raíz del árbol. Aplican las reglas de la gramática, sustituyendo la parte izquierda de una regla por su parte derecha, y generando el árbol sintáctico de arriba hacia abajo.
 2. **Ascendentes**: el árbol se construye de abajo hacia arriba, de las hojas a la raíz. Para ello, aplica las reglas de la gramática reduciendo la parte derecha de una producción por su parte izquierda, subiendo por los nodos hasta llegar a la raíz, el símbolo inicial S.

1. [Nociones generales.]

Tipos de analizadores sintácticos

Ejemplo: Sea la siguiente gramática:

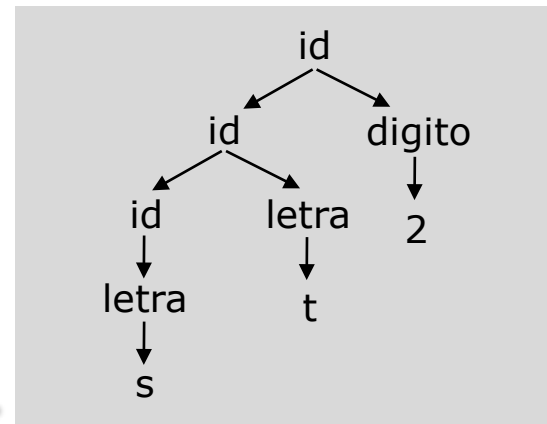
$id \rightarrow letra \mid id \ letra \mid id \ digito$

$letra \rightarrow a \mid b \mid \dots \mid z$

$digito \rightarrow 0 \mid 1 \mid \dots \mid 9$

- El análisis descendente aplicado a la cadena st2 resulta:

id \Rightarrow **id** digito \Rightarrow **id** letra digito \Rightarrow **letra** letra digito \Rightarrow
 \Rightarrow s **letra** digito \Rightarrow st **digito** \Rightarrow st2

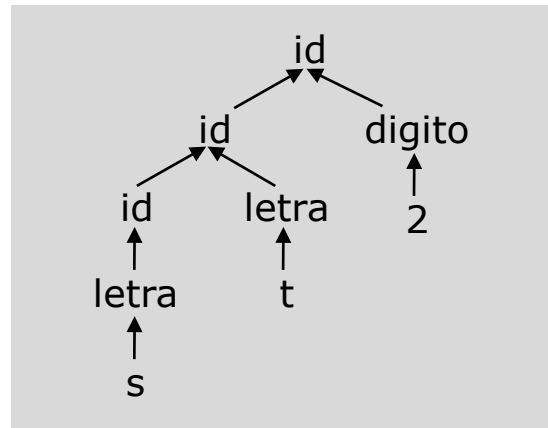


1. [Nociones generales.]

Tipos de analizadores sintácticos

- El análisis ascendente aplicado a la cadena st2 resulta:

st2 \Leftarrow **letra** **t2** \Leftarrow **id** **t2** \Leftarrow **id** **letra** **2** \Leftarrow **id** **2** \Leftarrow **id** **digito** \Leftarrow **id**



1. [Nociones generales.]

Forma de Backus-Naur (BNF) extendida

Es la notación más común para formalizar gramáticas. Los metasímbolos más comunes que utiliza son:

NOTACIÓN BNF BÁSICA

`::=`, utilizada en definiciones.

Ejemplo:

```
<condicional simple> ::= if <condicion> then <sentencia>
```

1. [Nociones generales.]

Forma de Backus-Naur (BNF) extendida

|, utilizada para representar disyunción.

Ejemplo:

```
<op_aritmetica_entera> ::= + | - | * | div | mod
```

<>, utilizados para representar símbolos no terminales.

Ejemplo:

```
<programa> ::= program
                <declaraciones>
            begin
                <sentencias>
            end.
```

1. [Nociones generales.]

Forma de Backus-Naur (BNF) extendida

NOTACIÓN BNF EXTENDIDA

[], utilizados para representar símbolos opcionales, es decir, pueden aparecer una vez o ninguna.

Ejemplo:

```
<sentencia_if> ::= if <expresion> then  
                    <sentencia>  
                    [ else  
                      <sentencia> ]  
                    endif;
```

1. [Nociones generales.]

Forma de Backus-Naur (BNF) extendida

{}, utilizados para representar repetición: cero, una o más veces.

Ejemplo:

```
<id> ::= <letra> { <letra> | <digito> }
```

es equivalente a la regla recursiva siguiente:

```
<id> ::= <letra> | <id> <letra> | <id> <digito>
```

", utilizados para distinguir metasímbolos de terminales.

Ejemplo:

```
<regla> ::= <id> " ::= " <expresion>
```

1. [Nociones generales.]

Forma de Backus-Naur (BNF) visual

NOTACIÓN BNF VISUAL

- Los símbolos terminales se marcan en negrita.
- Se eliminan los símbolos $\langle \rangle$ de los no terminales.
- Se utiliza el símbolo \rightarrow en lugar de $::=$

Ejemplo:

```
sentencia_if  $\rightarrow$  if expresion then  
                    sentencia  
                    [ else  
                      sentencia ]  
                    endif;
```

2. [Diseño de una gramática.]

- Cuando se desarrolla un nuevo lenguaje de programación, se debe diseñar una nueva gramática que describirá las características del lenguaje. Algunos aspectos que deben ser discutidos en el diseño de una nueva gramática son:
 - **Recursividad**. Lo que permite reducir el número de reglas sintácticas.
 - **Ambigüedad**. Se ha de evitar: proporciona una gramática a menudo más intuitiva, pero permite generar distintos códigos objeto para el mismo código fuente.
 - **Asociatividad**. Se deben proporcionar reglas de asociatividad para las distintas expresiones del lenguaje.
 - **Precedencia**. Determina el orden en el que se realizarán las distintas operaciones del lenguaje.

2. [Diseño de una gramática.]

Recursividad

- Los lenguajes de programación permiten generar innumerables programas, lo que obliga a utilizar un método de generación sin realizar una casuística interminable.
- La recursividad permite generar un **número infinito de programas** mediante una gramática finita.
- Una gramática es **recursiva** si al reescribir un no terminal, éste vuelve a aparecer en una o más derivaciones $A \Rightarrow^+ \alpha A \beta$.

Ojo esto examen

Ejemplo:

El + indica en algún momento de las sustituciones, nos encontramos cuando hay un símbolo terminal, la propia A o uno no terminal.

```
<sentencia> ::= begin <sentencia> {; <sentencia>} end
<sentencia> ::= <asignacion> | <llamada_funcion> |
                                     <sentencia_repetitiva> |
                                     <sentencia_condicional>
```


2. [Diseño de una gramática.]

Ambigüedad

- Una gramática es ambigua si genera al menos una cadena mediante dos o más árboles de derivación diferentes.
- Se debe evitar la ambigüedad, ya que pudiendo utilizar distintos árboles sintácticos para generar una misma sentencia, el **código generado podrá ser distinto** en distintas ejecuciones del compilador.

Ejemplo:

```
<expresion> ::= <expresion> + <expresion> |  
               <expresion> * <expresion> |  
               (<expresion>) | - <expresion> | id
```

Podemos generar la sentencia `id+id*id` (nota: `id` denota al identificador como un símbolo terminal) mediante dos árboles distintos:

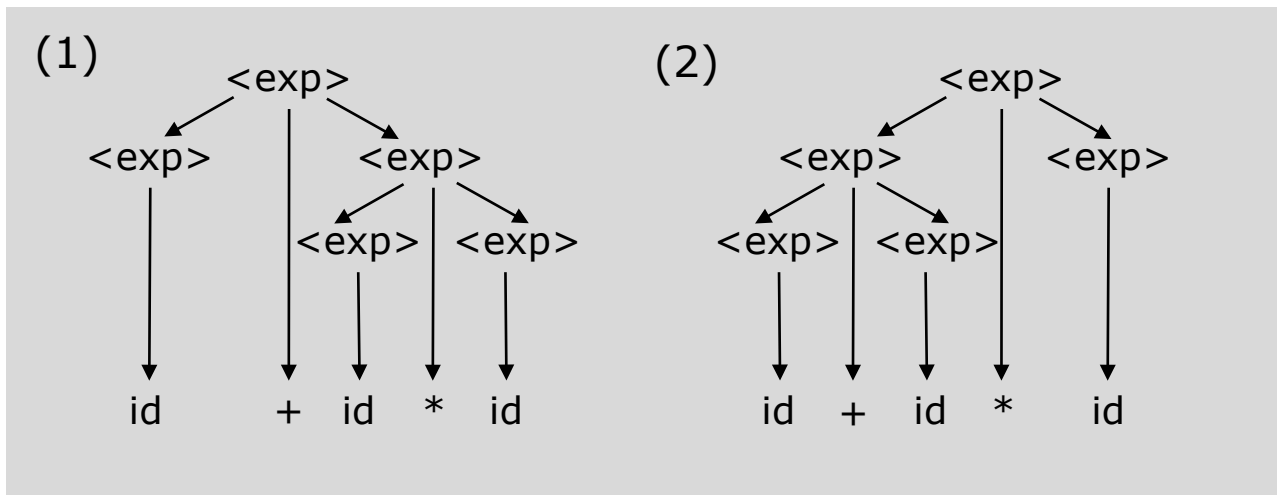
2. [Diseño de una gramática.]

Ambigüedad

(1) $\langle \text{expresion} \rangle \Rightarrow \langle \text{expresion} \rangle + \langle \text{expresion} \rangle \Rightarrow \text{id} + \langle \text{expresion} \rangle \Rightarrow$
 $\text{id} + \langle \text{expresion} \rangle * \langle \text{expresion} \rangle \Rightarrow \text{id} + \text{id} * \langle \text{expresion} \rangle \Rightarrow$
 $\text{id} + \text{id} * \text{id}$

El código objeto es distinto, porque el orden de las operaciones es distinto, y la segunda está mal.

(2) $\langle \text{expresion} \rangle \Rightarrow \langle \text{expresion} \rangle * \langle \text{expresion} \rangle \Rightarrow$
 $\langle \text{expresion} \rangle + \langle \text{expresion} \rangle * \langle \text{expresion} \rangle \Rightarrow \text{id} + \langle \text{expresion} \rangle * \langle \text{expresion} \rangle \Rightarrow$
 $\text{id} + \text{id} * \langle \text{expresion} \rangle \Rightarrow \text{id} + \text{id} * \text{id}$



2. [Diseño de una gramática.]

Ambigüedad

Ejemplo: Sea la siguiente gramática:

```
<sentencia> ::= if <expresion> then <sentencia> |  
                if <expresion> then <sentencia>  
                else <sentencia>
```

Obtener el árbol de derivación que proporciona esta gramática para el siguiente fragmento de código:

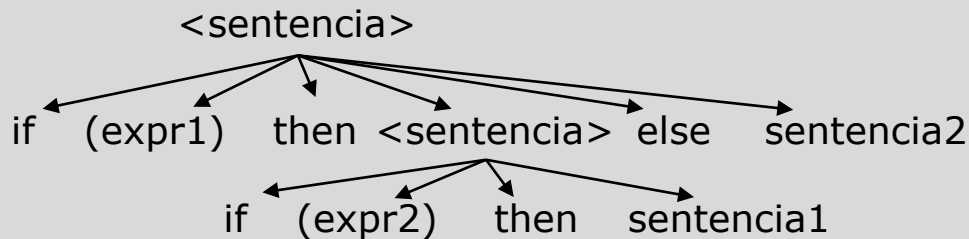
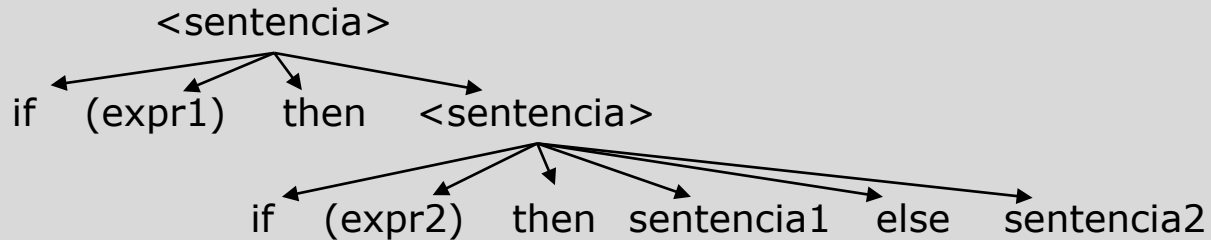
```
if (expr1) then  
    if (expr2) then  
        sentencia1;  
    else  
        sentencia2;
```

El else no se sabe si viene del primer if o del segundo, no depende de las tabulaciones.

2. [Diseño de una gramática.]

Ambigüedad

Esta gramática es ambigua, ya que da lugar a dos árboles de derivación distintos:



La primera de ellas se corresponde con su interpretación común: el 'else' se asocia al 'if' más cercano.

Ejercicio 6

- Sea la gramática siguiente:

```
<sentencia> ::= if <expresion> then <sentencia> <else> | <expresion>  
               <else> ::= else <sentencia> |  $\epsilon$   
               <expresion> ::= expresion
```

Se pide demostrar que es una gramática ambigua

2. [Diseño de una gramática.]

Factorización por la izquierda

- En ocasiones, durante el diseño de una gramática, aparecen dos producciones de un mismo no terminal que empiezan igual.
- Cuando durante el análisis se llegue a este no terminal, no se sabrá cuál de las reglas elegir.

Sacar factor común de lo que coincide en distintas reglas.

Ejemplo:

```
<sentencia> ::= if <expresion> then <sentencia> |  
                if <expresion> then <sentencia>  
                else <sentencia>
```

- Una solución es reescribir estas reglas, retrasando la decisión hasta haber visto lo suficiente. Llamamos a esta solución factorización por la izquierda.

2. [Diseño de una gramática.]

Factorización por la izquierda

- En el ejemplo anterior el resultado de la factorización sería el siguiente:

Ejemplo:

Este else es no terminal

```
<sentencia> ::= if <expresion> then <sentencia> <else>  
               <else> ::= else <sentencia> |  $\epsilon$ 
```

- En general, si nuestra gramática tiene una producción de la forma $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$.
- Podemos realizar la sustitución siguiente:

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

- Comprobamos si $\beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ también se pueden factorizar...

2. [Diseño de una gramática.]

Asociatividad

- Cuando diseñamos un lenguaje debemos decidir el tipo de asociatividad de las operaciones entre más de dos operandos:
 1. **Asociatividad por la derecha**. Las operaciones se hacen de derecha a izquierda. Por ejemplo, la asignación.

Asignación en C

$$a=b=c \equiv a=(b=c)$$

2. **Asociatividad por la izquierda**. Las operaciones se hacen de izquierda a derecha. Por ejemplo, las aritméticas.

$$a+b+c \equiv (a+b)+c$$

- La asociatividad se puede incorporar a una gramática mediante la recursividad: distinguimos entre operador asociativo por la izquierda y por la derecha.

2. [Diseño de una gramática.]

Asociatividad

1. **Operador asociativo por la derecha.** La regla sintáctica en la que aparece el operador se hace recursiva por la derecha.

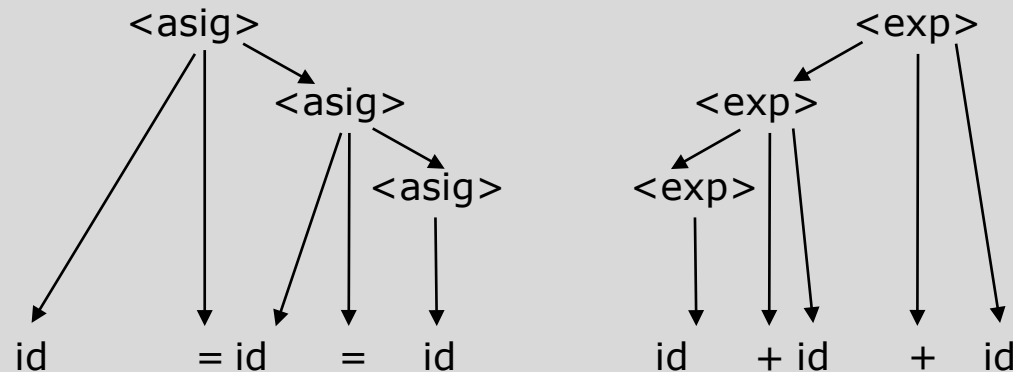
`<asignacion> ::= id = <asignacion> | id`

Asignación a la derecha para que sea asociativo por la derecha.

2. **Operador asociativo por la izquierda.** La regla sintáctica en la que aparece el operador se hace recursiva por la izquierda.

`<expresion> ::= <expresion> + id | id`

Ejemplo:



2. [Diseño de una gramática.]

Precedencia

- Como ya vimos, cuando en una expresión intervienen varios operadores se puede producir ambigüedad al construir el árbol de derivación.
- La precedencia permite especificar un **orden relativo de evaluación** de unos operadores respecto a otros.
- Se puede incorporar la precedencia utilizando:
 - a) Una variable sintáctica para cada nivel de precedencia.
 - b) Una producción para cada operador.
- Un operador tendrá menor precedencia cuanto más cerca esté su regla de derivación de la regla inicial de la gramática.

2. [Diseño de una gramática.]

Precedencia

Ejemplo:

Deseamos una gramática asociativa por la izquierda, con la suma y la resta con precedencia 1, la multiplicación y la división con precedencia 2, la potenciación con precedencia 3 asociativa por la derecha, y el paréntesis con precedencia máxima.

Lo más cerca del símbolo inicial es lo de menor nivel de precedencia. Es decir, la suma y la resta.

```
<expresion> ::= <expresion> + <expr_mult> |  
               <expresion> - <expr_mult> |  
               <expr_mult>  
<expr_mult> ::= <expr_mult> * <expr_exp> |  
               <expr_mult> / <expr_exp> |  
               <expr_exp>  
<expr_exp>  ::= <valor> ^ <expr_exp> | <valor>  
               <valor> ::= (<expresion>) | id
```

Ejercicio 7

- Sea la gramática mostrada en la página anterior.

Se pide construir el árbol de derivación para las siguientes expresiones:

a) $2 + (2 - 2) / 2 * (2 - 2) / (2 - 2)$

b) $2 + 2 - 2 / 2 * 2 - 2 / 2 - 2$

3. [Análisis sintáctico descendente.] (ASD)

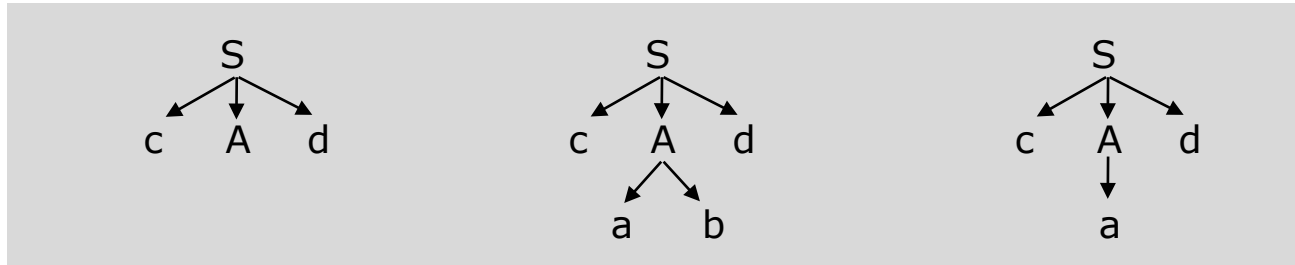
- Una vez definida una gramática, debemos comprobar que cada programa escrito en código fuente obedece a las reglas de la gramática. Este proceso es el **análisis sintáctico**.
Contruir un arbol de derivación del símbolo inicial al código fuente.
- El caso más simple de ASD es totalmente recursivo: Si es capaz de construirlo es correcto.
 1. **Avanzamos**: cuando sustituimos un no terminal por alguna de sus producciones.
 2. **Retrocedemos**: cuando se llega a un punto muerto y es necesario probar otra regla de sustitución.
- Para gestionar este proceso utilizamos una pila, donde almacenamos los símbolos de cada sustitución. Intentaremos emparejar el símbolo que hay en la cima de la pila con la entrada actual.

3. [Análisis sintáctico descendente.] (ASD)

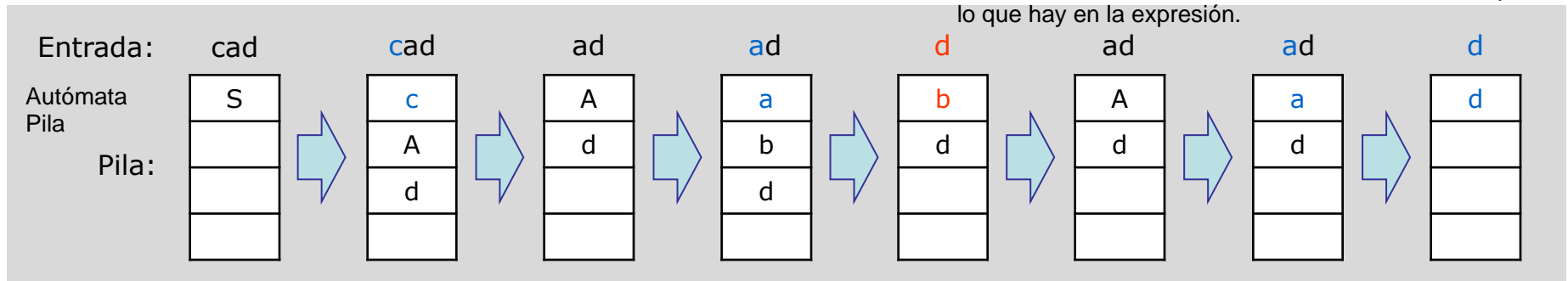
Ejemplo:

Queremos saber si la entrada $w = cad$ pertenece a la gramática siguiente: $S \rightarrow cAd$

$A \rightarrow ab \mid a$



Borro cuando coincide un terminal de la cima de la pila con lo que hay en la expresión.



- Si al final del proceso la pila y la entrada están vacías entonces la sentencia pertenece al lenguaje.

3. [Análisis sintáctico descendente.] (ASD)

- Una posible implementación del ASD se puede realizar desarrollando una función de análisis por cada no terminal.

Ejemplo:

Diseñar una función para la siguiente regla gramatical: $\langle \text{expr} \rangle ::= \langle \text{term} \rangle + \langle \text{expr} \rangle \mid \langle \text{term} \rangle$

```
boolean expresion()  
if (!termino())  
    return(false)  
if (TOKEN == + || TOKEN == -)  
    TOKEN=sig_comp_lexico()  
    if TOKEN=$ Fin de fichero  
        return(false)  
    if (!expresion())  
        return(false)  
    else  
        return(true)  
else  
    return(true)
```

Ojo

La Pila que se ha visto anteriormente es la pila del sistema para llamadas recursivas.

3. [Análisis sintáctico descendente.] (ASD)

- Estos métodos de ASD tienen algunos inconvenientes que no los hacen recomendables:
 1. Son muy lentos debido a los retrocesos.
 2. No se sabe si la entrada pertenece o no al lenguaje hasta analizar todas las posibilidades. Si no pertenece, en la pila tenemos el símbolo S. No podemos decir dónde está el error.
 3. Si se genera código objeto durante el análisis, en cada retroceso hay que eliminar parte del código generado.

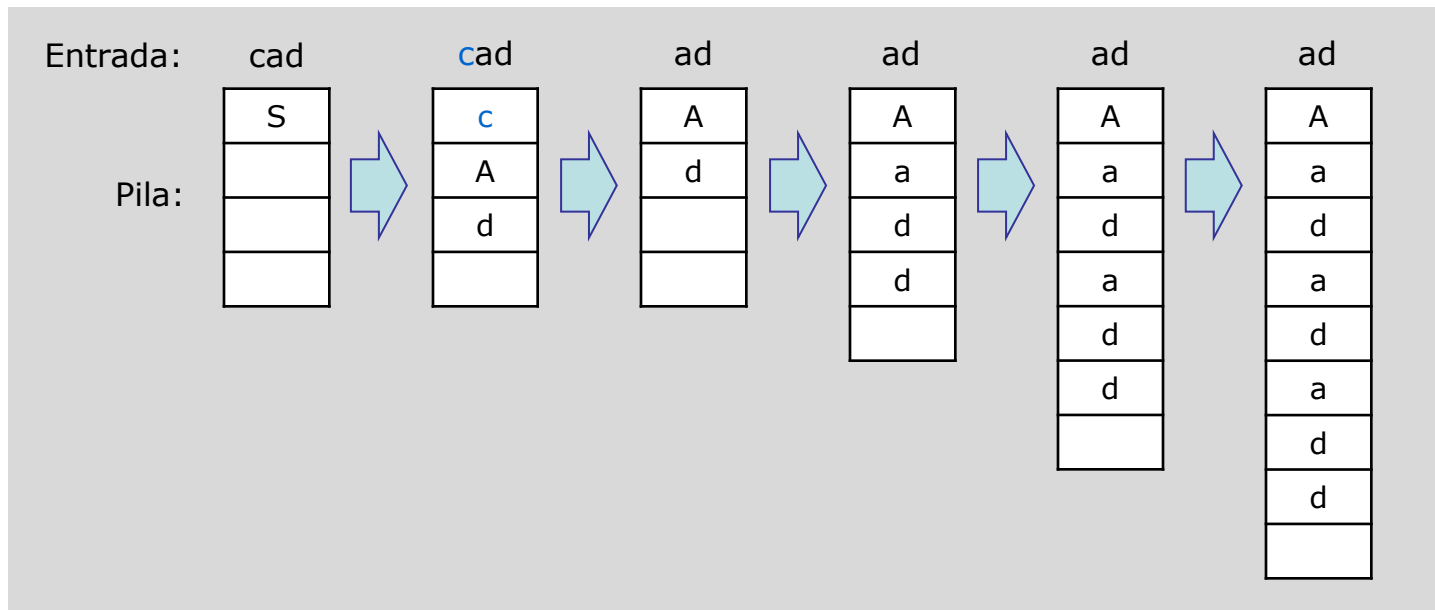
3. [Análisis sintáctico descendente.]

Rekursividad por la izquierda

- Las reglas recursivas por la izquierda pueden hacer entrar al ASD en un bucle infinito.

Ejemplo:

Queremos saber si la entrada $w = cad$ pertenece a la gramática siguiente:

$$S \rightarrow cAd$$
$$A \rightarrow Aad \mid a$$


3. [Análisis sintáctico descendente.]

Recursividad por la izquierda

- Debemos, por tanto, eliminar la recursividad por la izquierda.
- Sea una gramática recursiva:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$$

- Una gramática equivalente no recursiva por la izquierda será:

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \varepsilon \end{aligned}$$

Ejemplo:

Aplicamos este procedimiento a la gramática del ejemplo anterior:

$$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow Aad \mid a \end{aligned}$$



$$\alpha = ad, \beta = a$$



$$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow aA' \\ A' &\rightarrow adA' \mid \varepsilon \end{aligned}$$

Ejercicio 8

- Sea la gramática siguiente:

```
<expresion> ::= <expresion> + <termino> | <termino>  
<termino> ::= <termino> <factor> | <factor>  
<factor> ::= <factor> * | <valor>  
<valor> ::= a | b
```

Se pide eliminar la recursividad por la izquierda.

3. [Análisis sintáctico descendente.]

Rekursividad por la izquierda


- El método anterior elimina la recursividad inmediata, esto es, en la misma derivación. Si la recursividad aparece en derivaciones posteriores debemos encontrar el elemento conflictivo y sustituirlo por su definición.

Ejemplo:

$S \rightarrow Aa \mid b$		$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \varepsilon$		$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

Existe recursividad por la izquierda en la aplicación de las reglas de sustitución $S \rightarrow Aa$ y $A \rightarrow Sd$. Sustituimos $Sd \rightarrow Aad \mid bd$.

Y aplicamos el método anterior:

$\alpha = c \mid ad, \beta = bd \mid \varepsilon$		$S \rightarrow Aa \mid b$
		$A \rightarrow bdA' \mid A'$
		$A' \rightarrow cA' \mid adA' \mid \varepsilon$

3. [Análisis sintáctico descendente.]

Analizador descendente predictivo

- Deseamos evitar los retrocesos, y predecir en cada momento cuál de las reglas debemos aplicar para continuar el análisis correctamente.
- Supongamos que la cadena de entrada es $c_1 \dots c_i \dots c_n$.
- Supongamos que el componente léxico actual es c_i .
- Supongamos que el no terminal a sustituir es A .
- Las sustituciones posibles son $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$.
- Si cada una de las reglas de sustitución comienza por un componente léxico distinto debemos aplicar la regla de sustitución que comienza por c_i .

3. [Análisis sintáctico descendente.]

Analizador descendente predictivo

Ejemplo:

```
<sentencia> ::= if <expresion> then <sentencia> |  
                while <expresion> do <sentencia> |  
                begin <sentencia> end
```

- Puesto que todas las reglas de sustitución comienzan por un componente léxico distinto (if, while, begin) el análisis nunca retrocederá, salvo error.
- El tipo de gramáticas que se proponen son las LL(k):
 1. L (*left*): la entrada se lee de izquierda a derecha.
 2. L (*left*): se sustituye el no terminal situado a la izquierda.
 3. (k): se leen k componentes léxicos por anticipado.

3. [Análisis sintáctico descendente.]

Analizador descendente predictivo

- Nos centraremos en gramáticas LL(1), con un solo componente léxico analizado por anticipado. Una gramática LL(1) debe cumplir:
 - No puede ser recursiva por la izquierda.
 - Dos reglas de sustitución de un mismo no terminal no pueden dar lugar al mismo componente léxico inicial.
- Si encontramos una gramática no LL(1) la modificaremos:
 1. Eliminando la recursividad por la izquierda.
 2. Sacando factor común por la izquierda.

Ojo, estas son condiciones necesarias, no suficientes.

3. [Análisis sintáctico descendente.]

Analizador descendente predictivo

Ejemplo: Obtener la gramática LL(1) equivalente a:

$$A \rightarrow Aa \mid bB$$

$$B \rightarrow bc \mid bb \mid b$$

Eliminamos la recursividad en $A \rightarrow Aa \mid bB$:

$$A \rightarrow bBA'$$

$$A' \rightarrow aA' \mid \varepsilon$$

Sacamos factor común en $B \rightarrow bc \mid bb \mid b$:

$$B \rightarrow bB'$$

$$B' \rightarrow c \mid b \mid \varepsilon$$

Y ahora, ¿ya es LL(1)? No necesariamente.

3. [Análisis sintáctico descendente.] Conjuntos de predicción]

- Definimos dos conjuntos de predicción, que ayudarán en el proceso de análisis predictivo: el conjunto PRIMEROS y el conjunto SIGUIENTES.

CONJUNTO PRIMEROS

- Sea X un símbolo gramatical ($X \in (V \cup \Sigma)$), $\text{PRIMEROS}(X)$ es el conjunto de terminales (incluyendo ϵ) que aparecen al inicio de las cadenas derivables de X en ninguno o más pasos.

$$\text{PRIMEROS}(X) = \{v \mid X \Rightarrow^* v\beta, v \in \Sigma, \beta \in \Sigma^*\}$$

- Veremos un conjunto de reglas para obtener $\text{PRIMEROS}(X)$; posteriormente, describiremos la forma de obtención de $\text{PRIMEROS}(\alpha)$, donde $\alpha = X_1X_2\dots X_n$

3. [Análisis sintáctico descendente.] Conjuntos de predicción]

- Para calcular $\text{PRIMEROS}(X)$ hacemos:
 1. Si X es un símbolo terminal a , $\text{PRIMEROS}(X)=a$
 2. Si $(X \rightarrow \varepsilon) \in P$, añadimos ε a $\text{PRIMEROS}(X)$.
 3. Si X es un no terminal, y $X \rightarrow Y_1Y_2...Y_k$ es una regla de sustitución, añadimos $\text{PRIMEROS}(Y_j)$ a $\text{PRIMEROS}(X)$, si $\text{PRIMEROS}(Y_j)$ es un terminal, y para todo $i=1,2,...,j-1$, $Y_i \Rightarrow^* \varepsilon$.
 4. Si X es un no terminal, $X \rightarrow Y_1Y_2...Y_k$ es una regla de sustitución, y $Y_j \Rightarrow^* \varepsilon \forall j \in \{1,2,...,k\}$, añadimos ε a $\text{PRIMEROS}(X)$.

3. [Análisis sintáctico descendente.] Conjuntos de predicción]

- Para calcular $\text{PRIMEROS}(X_1X_2\dots X_n)$ hacemos:
 1. Añadimos todos los símbolos distintos de ε de $\text{PRIMEROS}(X_1)$.
 2. Si ε está en $\text{PRIMEROS}(X_1)$, añadir también los símbolos distintos de ε de $\text{PRIMEROS}(X_2)$.
 3. Si ε está en $\text{PRIMEROS}(X_2)$, añadir también los símbolos distintos de ε de $\text{PRIMEROS}(X_3)$. Y así sucesivamente.
 4. Por último, añadir ε a $\text{PRIMEROS}(X_1X_2\dots X_n)$ si $\text{PRIMEROS}(X_i)$ contiene a ε , $\forall i \in \{1, 2, \dots, n\}$.

3. [Análisis sintáctico descendente.]

Analizador descendente predictivo

Ejemplo:

Obtener los conjuntos PRIMEROS de la gramática siguiente:

```
<expresion> ::= <termino> <expresion'>
<expresion'> ::= + <termino> <expresion'> | ε
<termino> ::= <factor> <termino'>
<termino'> ::= * <factor> <termino'> | ε
<factor> ::= (<expresion>) | id
```

$\text{PRIMEROS}(\langle \text{expresion} \rangle) = \text{PRIMEROS}(\langle \text{termino} \rangle) =$
 $= \text{PRIMEROS}(\langle \text{factor} \rangle) = \{ (, id \}$

$\text{PRIMEROS}(\langle \text{expresion}' \rangle) = \{ +, \epsilon \}$

$\text{PRIMEROS}(\langle \text{termino}' \rangle) = \{ *, \epsilon \}$

3. [Análisis sintáctico descendente.] Conjuntos de predicción]

Ejemplo:

Para la gramática del ejemplo anterior, obtener el conjunto $\text{PRIMEROS}(<\text{termino}'> <\text{expresion}'> \text{id})$

- Sabemos que:

$$\text{PRIMEROS}(<\text{termino}'>) = \{*, \epsilon\}$$

$$\text{PRIMEROS}(<\text{expresion}'>) = \{+, \epsilon\}$$

- Por tanto,

$$\text{PRIMEROS}(<\text{termino}'> <\text{expresion}'> \text{id}) = \{*, +, \text{id}\}$$

3. [Análisis sintáctico descendente.] Conjuntos de predicción]

CONJUNTO SIGUIENTES

- Sea A un símbolo no terminal. $SIGUIENTES(A)$ es el conjunto de terminales que pueden aparecer inmediatamente a la derecha de A en alguna sustitución.

$$SIGUIENTES(A) = \{v \mid S \Rightarrow^+ \alpha A v \beta, v \in \Sigma, \alpha, \beta \in \Sigma^*\}$$

- Si A aparece al final de una cadena de símbolos, entonces diremos que el símbolo de fin de cadena $\$ \in SIGUIENTES(A)$.
- Para calcular $SIGUIENTES(A)$ hacemos:
 1. Si A es S , añadir $\$$ a $SIGUIENTES(S)$.
 2. Para cada regla $B \rightarrow \alpha A \beta$, añadir $PRIMEROS(\beta) - \epsilon$ a $SIGUIENTES(A)$.

3. [Análisis sintáctico descendente.] Conjuntos de predicción]

3. Para cada regla $B \rightarrow \alpha A \beta$, donde $\text{PRIMEROS}(\beta)$ contiene a ϵ ($\beta \Rightarrow^* \epsilon$), añadir todos los símbolos de $\text{SIGUIENTES}(B)$ a $\text{SIGUIENTES}(A)$.
4. Para cada regla $B \rightarrow \alpha A$ añadir $\text{SIGUIENTES}(B)$ a $\text{SIGUIENTES}(A)$.

Ejemplo:

Obtener los conjuntos SIGUIENTES de la gramática del ejemplo anterior.

$\text{SIGUIENTES}(\langle \text{expresion} \rangle) = \text{SIGUIENTES}(\langle \text{expresion}' \rangle) = \{), \$\}$

$\text{SIGUIENTES}(\langle \text{termino} \rangle) = \text{SIGUIENTES}(\langle \text{termino}' \rangle) = \{+,), \$\}$

$\text{SIGUIENTES}(\langle \text{factor} \rangle) = \{+, *,), \$\}$

3. [Análisis sintáctico descendente.]

Tabla de análisis sintáctico

- Mediante los conjuntos de predicción podemos construir una tabla que, a partir del no terminal a expandir y del componente léxico actual, indique qué regla se debe aplicar.
- En esta tabla las filas son los símbolos no terminales de la gramática, y sus columnas son los componentes léxicos, y el símbolo de fin de cadena \$. En las celdas de la tabla aparecen reglas de la gramática. Las celdas vacías se corresponden con un error en el análisis.
- Las celdas de la tabla $T[A,a]$ se rellenan siguiendo el siguiente procedimiento para cada regla de sustitución $A \rightarrow \alpha$:
 1. Para cada terminal $a \in \text{PRIMEROS}(\alpha)$, añadir $A \rightarrow \alpha$ en $T[A,a]$.
 2. Si $\epsilon \in \text{PRIMEROS}(\alpha)$, para cada terminal $b \in \text{SIGUIENTES}(A)$, añadir $A \rightarrow \alpha$ en $T[A,b]$.

3. [Análisis sintáctico descendente.]

Tabla de análisis sintáctico

- Una gramática es LL(1) si en la tabla de análisis sintáctico aparece como máximo una regla en cada celda.

Ejemplo:

Obtener la tabla de símbolos para la gramática anterior.

	id	+	*	()	\$
<expresion>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<expresion'>		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
<termino>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<termino'>		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
<factor>	$F \rightarrow id$			$F \rightarrow (E)$		

NOTA: $E = \langle \text{expresion} \rangle$, $E' = \langle \text{expresion}' \rangle$, $T = \langle \text{termino} \rangle$, $T' = \langle \text{termino}' \rangle$, $F = \langle \text{factor} \rangle$

Ejercicio 9

- Sea la gramática siguiente:

```
<expresion> ::= <izquierda> <derecha> | ε  
<izquierda> ::= <factor> <termino>  
  <derecha> ::= + <izquierda> <derecha> | ε  
  <termino> ::= * <factor> <termino> | ε  
  <factor> ::= (<expresion>) | id
```

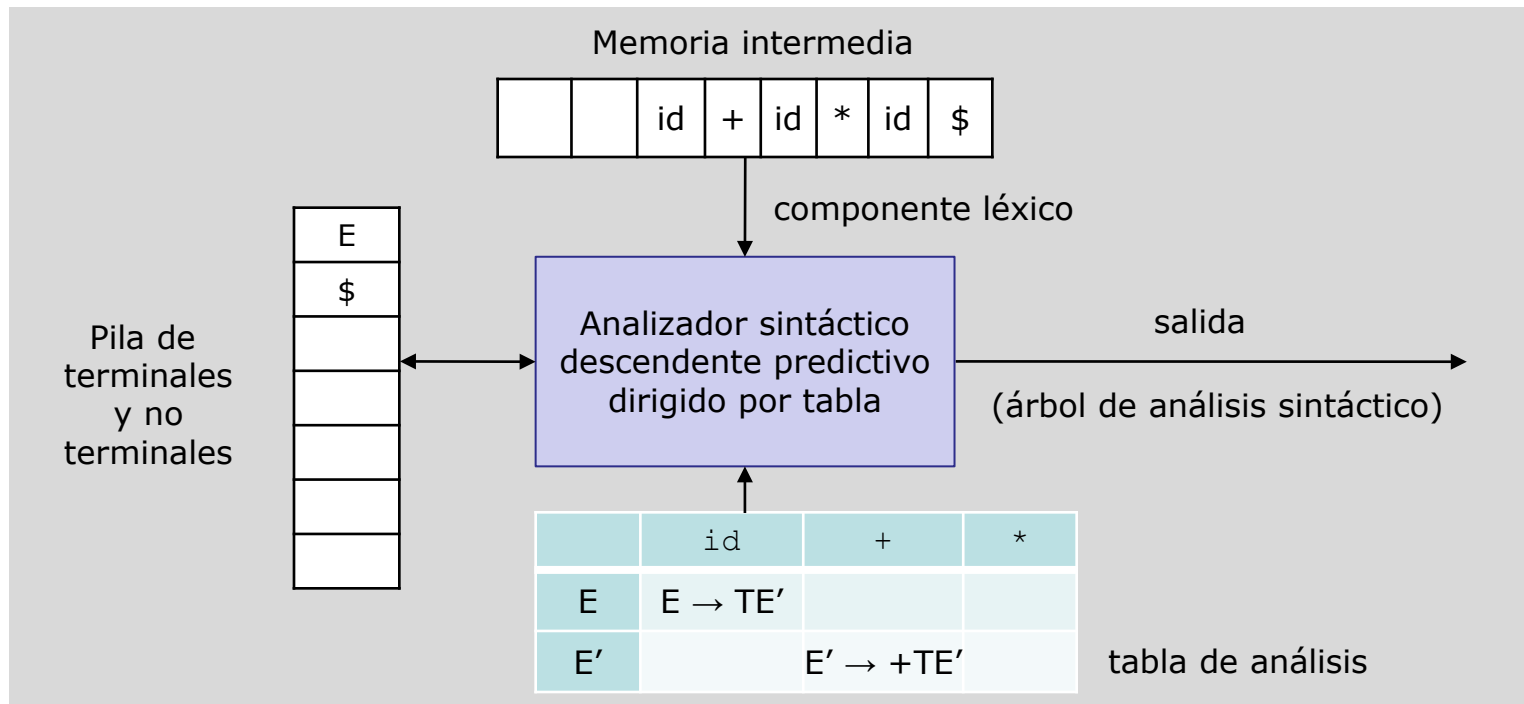
Se pide:

- Calcular los conjuntos PRIMEROS y SIGUIENTES.
- Construir la tabla de análisis sintáctico descendente. ¿Es una gramática LL(1)?
- Analizar la cadena $id * id + id$

3. [Análisis sintáctico descendente.]

Procedimiento de análisis

- Podemos proporcionar un procedimiento de análisis basado en el uso de una tabla de análisis sintáctico para seleccionar la siguiente regla de sustitución y una pila para almacenar los símbolos gramaticales



3. [Análisis sintáctico descendente.]

Procedimiento de análisis

- El algoritmo de análisis es el siguiente:
 1. Inicializar la pila con el símbolo S , y añadir $\$$ a continuación y al final de la cadena de entrada.
 2. Comparar el símbolo X de la cima de la pila y el siguiente símbolo a de la entrada.
 - i. Si $X = a = \$$ **aceptar** la cadena y salir.
 - ii. Si X y a son iguales, pero distintos a $\$$, extraer el elemento de la pila y avanzar una posición en la cadena de entrada.
 - iii. Si X es un terminal distinto de a , o X es un no terminal y $T(X,a)$ está vacía, entonces **error**.
 - iv. Si X es un no terminal y en $T(X,a)$ tenemos $X \rightarrow X_1X_2...X_n$ extraer X de la pila e insertar $X_1X_2...X_n$.
 - v. Volver a 2.

3. [Análisis sintáctico descendente.]

Procedimiento de análisis

Ejemplo: Analizar la cadena id+id

Pila	Entrada	Acción
\$E	id+id\$	Aplicar $E ::= TE'$
\$E'T	id+id\$	Aplicar $T ::= FT'$
\$E'T'F	id+id\$	Aplicar $F ::= id$
\$E'T'id	id+id\$	Avanzar
\$E'T'	+id\$	Aplicar $T' ::= \epsilon$
\$E'	+id\$	Aplicar $E' ::= +TE'$
\$E'T+	+id\$	Avanzar
\$E'T	id\$	Aplicar $T ::= FT'$
\$E'T'F	id\$	Aplicar $F ::= id$
\$E'T'id	id\$	Avanzar
\$E'T'	\$	Aplicar $T' ::= \epsilon$
\$E'	\$	Aplicar $E' ::= \epsilon$
\$	\$	Cadena aceptada

3. [Análisis sintáctico descendente.]

Recapitulemos

- Las gramáticas LL(1) admiten un proceso de análisis **cuya complejidad es $O(n)$** . No todas las gramáticas son LL(1), pero en algunos casos se pueden llevar a cabo ciertas transformaciones para convertirlas en LL(1):
 1. **Eliminar la recursividad por la izquierda**: Si la gramática es recursiva por la izquierda, durante el análisis entraremos en un bucle infinito. Hemos visto cómo eliminar esta recursividad.
 2. **Eliminar la ambigüedad**: Si la gramática es ambigua no es LL(1), ya que ante un símbolo de entrada puede generarse más de un árbol sintáctico. No hay reglas para eliminar la ambigüedad. Lo mejor es rediseñar la gramática.
 3. **Factorizar por la izquierda**: Si dos sustituciones empiezan por el mismo terminal no sabremos cuál elegir. La factorización por la izquierda permite retrasar la decisión hasta que la entrada nos permita tomar la correcta.

3. [Análisis sintáctico descendente.] Gestión de errores]

- Pueden darse dos tipos de errores en el ASD:
 1. En la parte superior de la pila hay un terminal que no se corresponde con el componente léxico actual. El compilador deberá señalar el error, indicando qué es lo que esperaba encontrarse.
 2. En la parte superior de la pila hay un no terminal, y la celda de la tabla correspondiente al componente léxico actual está vacía. El compilador deberá señalar un error, conforme se esperaba uno de los componentes léxicos del conjunto de celdas del no terminal que contienen algún dato.
- Al detectar el error el compilador debe continuar el análisis, para ello puede aplicar la técnica de **conjuntos de sincronización**.

3. [Análisis sintáctico descendente.] Conjuntos de sincronización]

- Esta técnica se basa en que, al detectar un error, se entra en un **estado de error** en el que se continúa analizando la entrada, descartando los componentes léxicos que aparecen hasta encontrar uno que pertenezca al **conjunto de sincronización**.
- En este momento se valora quitar el símbolo de encima de la pila, se abandona el estado de error y se prosigue el análisis.
- La efectividad de este método depende de la elección del conjunto de símbolos de sincronización. Normalmente se utilizan **técnicas empíricas**, basadas en el estudio de los errores más frecuentes.

3. [Análisis sintáctico descendente.] Conjuntos de sincronización]

Ejemplos:

- Para un no terminal A , poner en el conjunto de sincronización los símbolos de $SIGUIENTES(A)$. Es probable que el análisis pueda continuar si se recorren los terminales y así poder quitar A de la pila.
- En los lenguajes de programación suele haber niveles jerárquicos: expresión, sentencia, bloque, función, programa,... Se puede añadir al conjunto de sincronización de un no terminal de baja jerarquía los símbolos que inician un no terminal de nivel superior.
- Para un no terminal A , se pueden añadir los símbolos de $PRIMEROS(A)$ al conjunto de sincronización y continuar el análisis cuando aparezca uno de estos símbolos.

3. [Análisis sintáctico descendente.] Conjuntos de sincronización]

Ejemplos:

- Si se puede generar la cadena vacía, tomarla por omisión y dejar la decisión de la detección y recuperación del error para símbolos más importantes o de más jerarquía.
- Si el error se produce por no poder emparejar un terminal, se puede extraer de la pila como si hubiese aparecido en la entrada, dar un mensaje de error que indique que debiera haber aparecido, y proseguir el análisis.

3. [Análisis sintáctico descendente.] Conjuntos de sincronización

Ejemplo:

Añadir conjuntos de sincronización a la gramática anterior, haciendo uso del conjunto SIGUIENTES().

	id	+	*	()	\$
<expresion>	$E \rightarrow TE'$			$E \rightarrow TE'$	sinc	sinc
<expresion'>		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
<termino>	$T \rightarrow FT'$	sinc		$T \rightarrow FT'$	sinc	sinc
<termino'>		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
<factor>	$F \rightarrow \langle id \rangle$	sinc	sinc	$F \rightarrow (E)$	sinc	sinc

4. [Análisis sintáctico ascendente.] (ASA)

- Como ya hemos comentado, el análisis sintáctico ascendente parte de una cadena de entrada formada por una secuencia de componentes léxicos, y busca hacerla corresponder con las partes derechas de las reglas de la gramática.
- Cuando encuentra una, la sustituye (decimos reduce) por su parte izquierda, y crea un nuevo nodo del árbol sintáctico.
- Continúa así hasta llegar al símbolo inicial S, o encontrar un error.
- Genera, por tanto, el árbol sintáctico de abajo a arriba: empieza por las hojas, que se corresponden con componentes léxicos, y acaba en la raíz, aplicando las reglas de sustitución en sentido inverso.

4. [Análisis sintáctico ascendente.] (ASA)

Ejemplo:

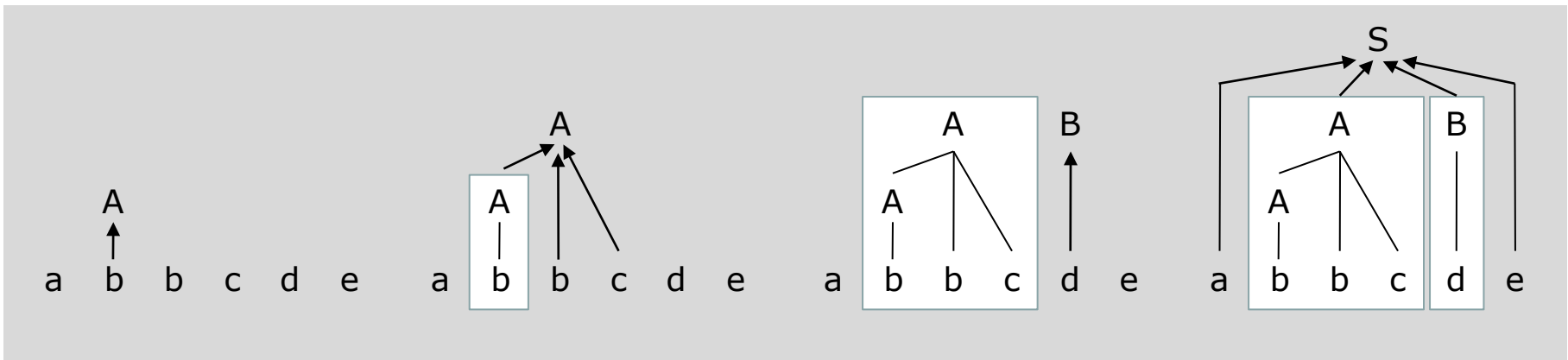
Analizar la cadena abbcde con la gramática:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

$abbcde \Leftarrow aAbcde \Leftarrow aAde \Leftarrow aABe \Leftarrow S$



- La cadena se procesa de izquierda a derecha, y en cada paso se sustituyen uno o más símbolos por una única variable.

4. [Análisis sintáctico ascendente.] (ASA)

- Puede haber distintas reglas de sustitución con alguna parte derecha común, y reglas que sustituyan por la cadena vacía. El problema es decidir cuándo lo que parece una parte derecha de una regla se puede sustituir por su parte izquierda.
- Los algoritmos de ASA utilizan una pila en su análisis, y realizan dos operaciones básicas:
 1. **Desplazamiento**: lleva el siguiente símbolo de entrada a la cima de la pila.
 2. **Reducción**: extrae tantos símbolos de la pila como símbolos tiene la parte derecha de la regla de sustitución, por la cual se quiere reducir, y los sustituye por el símbolo no terminal de la parte izquierda de la regla.

4. [Análisis sintáctico ascendente.] (ASA)

Ejemplo:

Analizar la cadena $\text{id}+\text{id}*\text{id}$ con la gramática:

$E \rightarrow E + E$
 $E \rightarrow E * E$
 $E \rightarrow (E)$
 $E \rightarrow \text{id}$

Pila	Entrada	Acción
	$\text{id}+\text{id}*\text{id}\$$	Desplazar
id	$+\text{id}*\text{id}\$$	Reducir con $E \rightarrow \text{id}$
E	$+\text{id}*\text{id}\$$	Desplazar
E+	$\text{id}*\text{id}\$$	Desplazar
E+id	$*\text{id}\$$	Reducir con $E \rightarrow \text{id}$
E+E	$*\text{id}\$$	Reducir con $E \rightarrow E+E$, o desplazar
E+E*	$\text{id}\$$	Desplazar
E+E*id	$\$$	Reducir con $E \rightarrow \text{id}$
E+E*E	$\$$	Reducir con $E \rightarrow E*E$
E+E	$\$$	Reducir con $E \rightarrow E+E$
E	$\$$	Aceptar

4. [Análisis sintáctico ascendente.]

Gramáticas LR(k)

- En el análisis ascendente por desplazamiento-reducción se utilizan gramáticas LR(k):
 1. L (*left*): la entrada se lee de izquierda a derecha.
 2. R (*right*): se construyen derivaciones por la derecha en orden inverso.
 3. (k): se leen k componentes léxicos por anticipado. Se puede demostrar que, para cada gramática LR(k) con $k > 1$, hay una gramática LR(1) equivalente. Por este motivo, normalmente se trabaja con $k=1$.
- Las gramáticas LR(1) son no ambiguas y **permiten describir más lenguajes que las gramáticas LL(1)**, englobando a éstas.

4. [Análisis sintáctico ascendente.]

Gramáticas LR(k)

- Hay tres tipos de gramáticas LR, con el mismo esquema de control, basado en la construcción y uso de una tabla de análisis. Por la manera de construir esta tabla distinguiremos:
 - Gramáticas SLR(1) (*Simple LR*): presentan la construcción más sencilla. Serán las que estudiaremos en este curso.
 - Gramáticas LR(1): permiten describir un rango mayor de gramáticas, pero la construcción es más compleja.
 - Gramáticas LALR(1) (*Look Ahead LR*): suponen un compromiso entre las dos anteriores.

4. [Análisis sintáctico ascendente.]

Procedimiento de análisis

- ¿Cómo sabe un analizador sintáctico ascendente cuándo desplazar y cuándo reducir?
- Utilizaremos un procedimiento basado en el diseño de un **autómata finito determinista**, que nos dirá si una entrada pertenece al lenguaje generado por la gramática.
- Sus **estados** son conjuntos de elementos que representan situaciones equivalentes en el análisis de una entrada.
- Una función a la que llamaremos Ir_a nos permitirá simular las **transiciones** del autómata para cada uno de los símbolos de la gramática.
- Para una gramática G creamos su **gramática aumentada**, resultado de añadir artificialmente la regla $S' \rightarrow S\$$. Cuando vamos a reducir mediante esta nueva regla aceptamos.

4. [Análisis sintáctico ascendente.] Gramáticas SLR(1). Elementos.]

- Un elemento LR(0), o elemento simplemente, es una regla simple que contiene una marca en algún lugar de la parte derecha. Esta marca separa los símbolos de la regla que han sido reconocidos de los pendientes de reconocer.

Ejemplo:

Analizar los elementos de la regla $A \rightarrow B-D$

Elementos posibles	Símbolos reconocidos	Símbolos esperados
$A \rightarrow \cdot B-D$	Ninguno	Primeros(B)
$A \rightarrow B \cdot -D$	B	-
$A \rightarrow B - \cdot D$	B-	Primeros(D)
$A \rightarrow B-D \cdot$	B-D	Siguientes(A)

- Para las reglas de sustitución $A \rightarrow \varepsilon$ sólo se obtiene el elemento $A \rightarrow \cdot$.

4. [Análisis sintáctico ascendente.] Gramáticas SLR(1). Clausura().

- La función `clausura()` se aplica a un conjunto de elementos I , y devuelve un nuevo conjunto de elementos que son equivalentes respecto a dónde ha llegado el análisis.
- El conjunto de elementos que devuelve `clausura(I)` se construye de la siguiente manera:
 1. Todo elemento de I se añade a `clausura(I)`.
 2. Si $A \rightarrow \alpha \cdot B \beta$ está en `clausura(I)` y $B \rightarrow \gamma$ es una regla de la gramática, entonces añadir $B \rightarrow \cdot \gamma$ a `clausura(I)`. Aplicar esta regla hasta que no se puedan añadir nuevos elementos a `clausura(I)`.

4. [Análisis sintáctico ascendente.] Gramáticas SLR(1). Clausura().

- Clausura(I) agrupa a todos los elementos que describen lo que se espera encontrar a partir del momento actual del análisis.
- Así, si $A \rightarrow \alpha \cdot B \beta \in \text{clausura}(I)$, acabamos de ver en la entrada una cadena derivable de α , y esperamos encontrar a continuación una cadena derivable de $B\beta$, es decir, una de aquellas cadenas resultado de sustituir B de todas las formas posibles.
- Por eso añadimos $B \rightarrow \cdot \gamma$ a $\text{clausura}(I)$. Si γ es una cadena que comienza con un no terminal repetimos este proceso.
- Terminamos cuando disponemos de todos los elementos que indiquen el principio de la cadena esperada a partir del momento actual del análisis.

4. [Análisis sintáctico ascendente.]

Gramáticas SLR(1). Clausura().

Ejemplo:

Obtener los sucesivos conjuntos de clausura(I), donde $I = \{E' \rightarrow \cdot E\$ \}$, para la siguiente gramática:

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

I	Clausura(I) primer paso	Clausura(I) segundo paso	Clausura(I) tercer paso	Clausura(I) cuarto paso
				$E' \rightarrow \cdot E\$$
			$E' \rightarrow \cdot E\$$	$E \rightarrow \cdot E+T$
		$E' \rightarrow \cdot E\$$	$E \rightarrow \cdot E+T$	$E \rightarrow \cdot T$
$E' \rightarrow \cdot E\$$	$E' \rightarrow \cdot E\$$	$E \rightarrow \cdot E+T$	$E \rightarrow \cdot T$	$T \rightarrow \cdot T * F$
		$E \rightarrow \cdot T$	$T \rightarrow \cdot T * F$	$T \rightarrow \cdot F$
			$T \rightarrow \cdot F$	$F \rightarrow \cdot (E)$
				$F \rightarrow \cdot id$

4. [Análisis sintáctico ascendente.]

Gramáticas SLR(1). $Ir_a(I, X)$.

- La función $Ir_a(I, X)$ se aplica a un conjunto de elementos I , y a un símbolo de la gramática $X \in (V \cup \Sigma)$, y devuelve un nuevo conjunto de elementos.
- $Ir_a(I, X)$ se define como la clausura del conjunto de todos los elementos $A \rightarrow \alpha X \cdot \beta$ tales que $A \rightarrow \alpha \cdot X \beta$ está en I .
- Intuitivamente, la función $Ir_a(I, X)$ adelanta el símbolo X en el análisis sintáctico ascendente.
- El conjunto que devuelve $Ir_a(I, X)$ está formado por aquellos elementos que después de I dan por reconocido el símbolo X .

4. [Análisis sintáctico ascendente.]

Gramáticas SLR(1). Ir_a(I,X).

Ejemplo:

Aplicar la función Ir_a(I₀,X) para la gramática del ejemplo anterior, donde I₀ es la clausura de I.

I ₀ =clausura(I)	I ₁ =Ir _a (I ₀ ,E)	I ₂ =Ir _a (I ₀ ,T)	I ₃ =Ir _a (I ₀ ,F)	I ₄ =Ir _a (I ₀ ,())	I ₅ =Ir _a (I ₀ ,id)
E' → ·E\$				F → (·E)	
E → ·E+T				E → ·E+T	
E → ·T	E' → E·\$	E → T·		E → ·T	
T → ·T*F	E → E·+T	T → T·*F	T → F·	T → ·T*F	F → id·
T → ·F				T → ·F	
F → ·(E)				F → ·(E)	
F → ·id				F → ·id	

4. [Análisis sintáctico ascendente.]

Colección canónica LR(0)

- Llamamos **colección canónica** LR(0) al conjunto de todos los posibles conjuntos de elementos de una gramática.
- Para generar la colección canónica de una gramática hacemos:
 1. Comenzamos creando su **gramática aumentada**, resultado de añadir artificialmente la regla $S' \rightarrow S\$$.
 2. El primer conjunto de elementos es $I_0 = \text{clausura}(S' \rightarrow \cdot S\$)$.
 3. Se calculan todos los posibles conjuntos resultado de aplicar $\text{Ir_a}(I_0, X)$ para todos los símbolos $X \in (V \cup \Sigma)$, y se descartan los conjuntos vacíos. Se obtienen I_1, I_2, \dots
 4. Aplicamos Ir_a de nuevo a I_1, I_2, \dots hasta no obtener nuevos conjuntos.

4. [Análisis sintáctico ascendente.]

Colección canónica LR(0)

Ejemplo:

Obtener la colección canónica LR(0) para la gramática del ejemplo anterior.

Antes obtuvimos I_0, \dots, I_5 . Si aplicamos sucesivamente Ir_a obtenemos:

$I_6 = Ir_a(I_1, +)$	$I_7 = Ir_a(I_2, *)$	$I_8 = Ir_a(I_4, E)$	$I_9 = Ir_a(I_6, T)$	$I_{10} = Ir_a(I_7, F)$	$I_{11} = Ir_a(I_8,))$
$E \rightarrow E + \cdot T$					
$T \rightarrow \cdot T * F$	$T \rightarrow T * \cdot F$	$F \rightarrow (E \cdot)$	$E \rightarrow E + T \cdot$	$T \rightarrow T * F \cdot$	$F \rightarrow (E) \cdot$
$T \rightarrow \cdot F$	$F \rightarrow \cdot (E)$	$E \rightarrow E \cdot + T$	$T \rightarrow T \cdot * F$		
$F \rightarrow \cdot (E)$	$F \rightarrow \cdot id$				
$F \rightarrow \cdot id$					

4. [Análisis sintáctico ascendente.] Colección canónica LR(0)

- Si consideramos cada conjunto I_j como un estado de un autómata finito, Ir_a nos proporciona su [tabla de transiciones](#).

Ejemplo:

Para la gramática del ejemplo anterior:

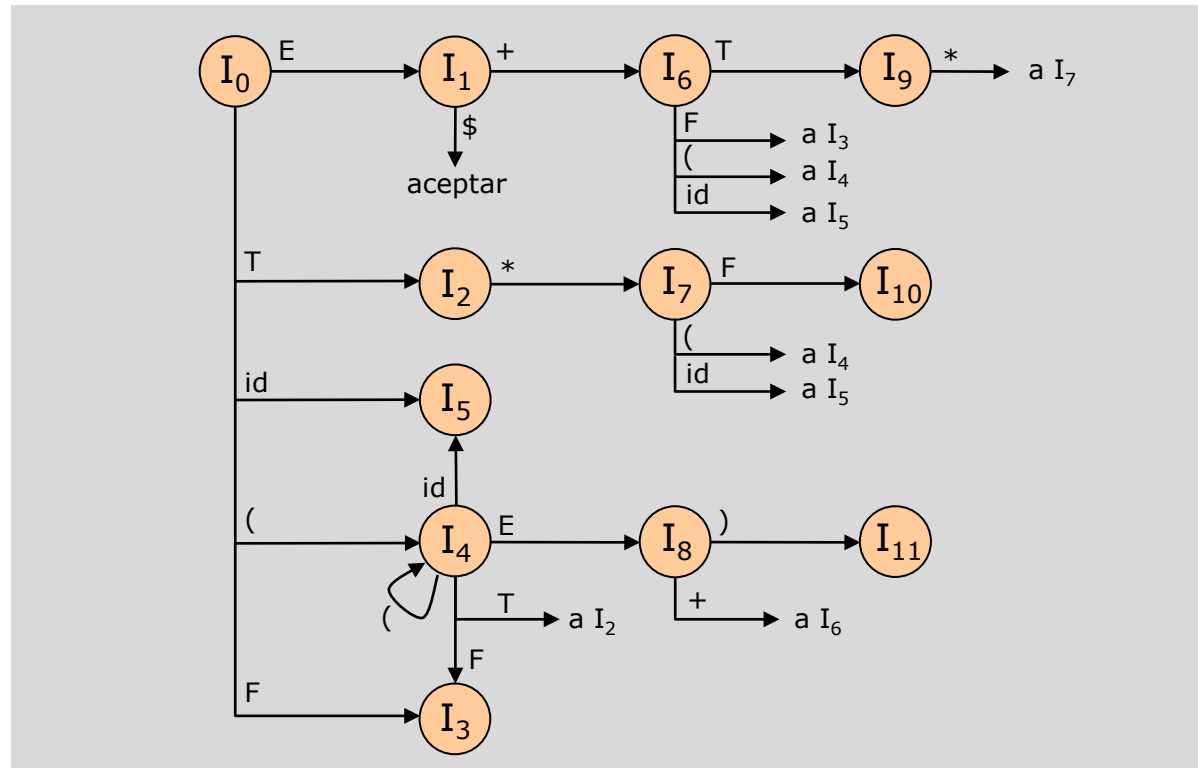
Estado	id	+	*	()	\$	E	T	F
0	5			4			1	2	3
1		6							
2			7						
3									
4	5			4			8	2	3
5									
6	5			4				9	3
7	5			4					10
8		6			11				
9			7						
10									
11									

4. [Análisis sintáctico ascendente.]

Colección canónica LR(0)

Ejemplo:

Obtener el autómata finito para la gramática del ejemplo anterior:



4. [Análisis sintáctico ascendente.]

Tabla de acciones SLR(1)

- Volvemos a la pregunta clave: ¿cómo sabe un analizador sintáctico ascendente cuándo desplazar y cuándo reducir?
- Podemos construir una **tabla de acciones** que nos permitirá tomar esta decisión. El procedimiento es el siguiente:
 1. Construir la colección canónica $LR(0)=\{I_0, I_1, \dots, I_n\}$.
 2. Para los elementos de I_j de la forma $A \rightarrow \alpha \cdot a \beta$, donde a es un terminal, e $Ir_a(I_j, a)=I_k$, entonces $acción[j, a]=$ **desplazar** la entrada e ir al estado correspondiente a I_k .
 3. Para los elementos de I_j de la forma $A \rightarrow \alpha \cdot$, entonces $acción[j, s]=$ **reducir** $A \rightarrow \alpha$, para cualquier $s \in SIGUIENTES(A)$.
 4. Si el elemento $S' \rightarrow S \cdot \$$ está en I_j , entonces $acción[j, \$]=$ **aceptar**.
 5. Las entradas vacías de la tabla de acciones representan estados de error.

4. [Análisis sintáctico ascendente.]

Tabla de acciones SLR(1)

Ejemplo:

Obtener la tabla de acciones para la gramática anterior:

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

Acciones							Ir_a		
Estado	id	+	*	()	\$	E	T	F
0	d5			d4			1	2	3
1		d6				OK			
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			

4. [Análisis sintáctico ascendente.]

Procedimiento de análisis

- En la tabla anterior **dn** significa desplazar la entrada e ir al estado n ; **rn** significa reducir mediante la regla (n) .
- Para entender mejor los desplazamientos y las reducciones observamos un ejemplo de la tabla:

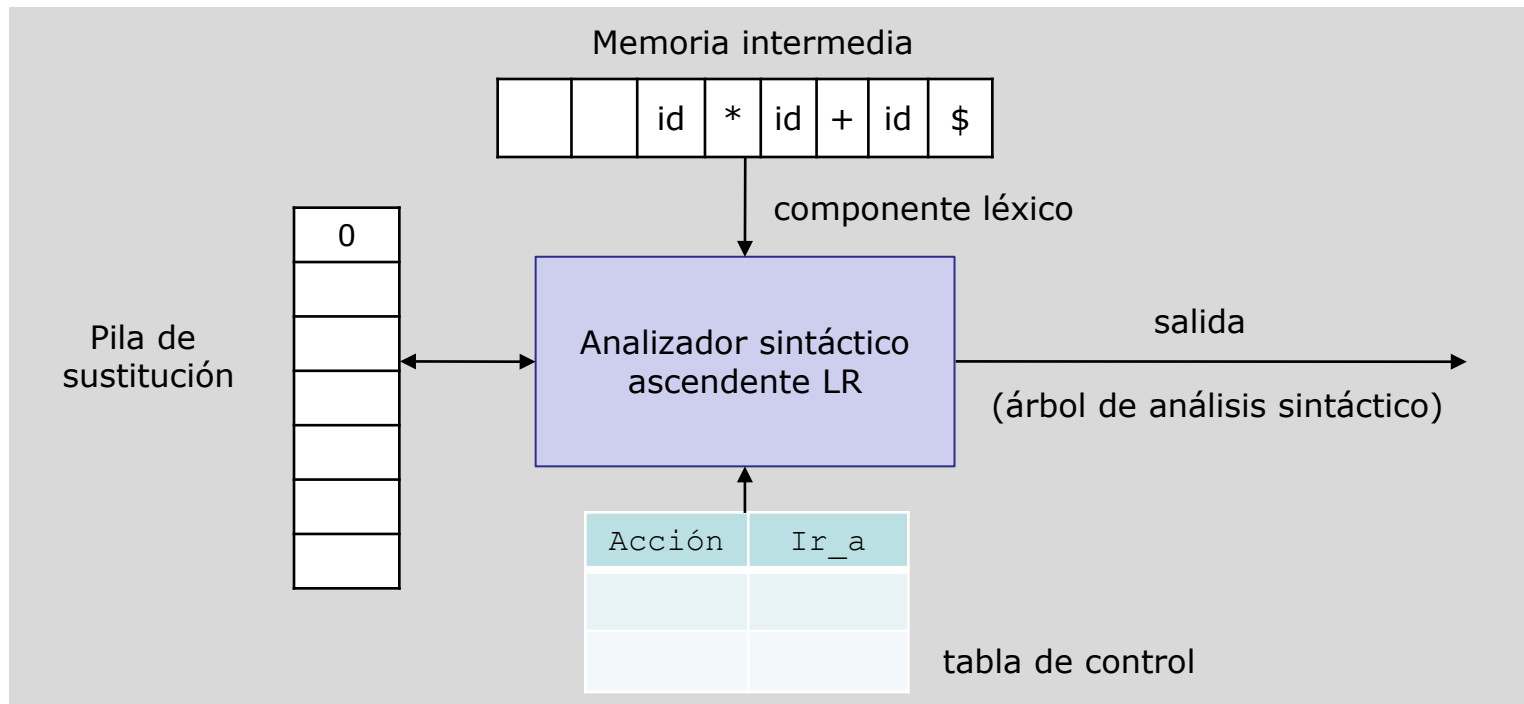
$$I_9 = Ir_a(I_6, T) = \{E \rightarrow E + T \cdot, T \rightarrow T \cdot * F\}$$

- Significa que en el estado 6, si analizamos T debemos ir al estado 9, caracterizado por la aplicación de dos reglas distintas:
 1. En el caso de $E \rightarrow E + T \cdot$, la acción será **reducir** por la regla (1) con los símbolos de $SIGUIENTES(E) = \{+,), \$\}$.
 2. En el caso de $T \rightarrow T \cdot * F$, la acción será **desplazar** la entrada e ir al estado 7, ya que $I_7 = Ir_a(I_9, *)$.

4. [Análisis sintáctico ascendente.]

Procedimiento de análisis

- Podemos proporcionar un procedimiento de análisis basado en el uso de una tabla de control para seleccionar la siguiente regla de sustitución y una pila para almacenar los símbolos gramaticales



4. [Análisis sintáctico ascendente.]

Procedimiento de análisis

- El algoritmo de análisis es el siguiente:
 1. Inicializar la pila con el estado 0.
 2. Para cada estado de la pila i y símbolo de la entrada a consultamos la celda acción[i,a].
 - i. Si acción[i,a]= dn , entonces **desplazamos** la entrada, vamos al estado n , y lo introducimos en la cima de la pila.
 - ii. Si acción[i,a]= rn , entonces **reducimos** mediante la regla (n) , de la forma $A \rightarrow \beta$; para ello, si m es la longitud de β , eliminamos m estados de la cima de la pila, y a continuación vamos al estado $Ir_a(h,A)$, donde h es el estado que queda en la pila al eliminar los m estados superiores.
 - iii. Si acción[i,a]=**aceptar**, completamos el análisis.
 - iv. Si acción[i,a]= \emptyset , **error**, e invocamos la gestión de errores.

4. [Análisis sintáctico ascendente.]

Tabla de acciones SLR(1)

Ejemplo: Desarrollar el análisis de $\text{id} * \text{id} + \text{id} \$$.

Estados	Símbolos	Entrada	Acción
0		$\text{id} * \text{id} + \text{id} \$$	Desplazar e ir al estado 5
0 5	id	$* \text{id} + \text{id} \$$	Reducir con (6) $F \rightarrow \text{id}$ (se quita 5 de la pila y se va a $\text{Ir_a}(0, F) = 3$)
0 3	F	$* \text{id} + \text{id} \$$	Reducir con (4) $T \rightarrow F$ (se quita 3 de la pila y se va a $\text{Ir_a}(0, T) = 2$)
0 2	T	$* \text{id} + \text{id} \$$	Desplazar e ir al estado 7
0 2 7	T^*	$\text{id} + \text{id} \$$	Desplazar e ir al estado 5
0 2 7 5	$T^* \text{id}$	$+ \text{id} \$$	Reducir con (6) $F \rightarrow \text{id}$ (se quita 5 y se va a $\text{Ir_a}(7, F) = 10$)
0 2 7 10	$T^* F$	$+ \text{id} \$$	Reducir con (3) $T \rightarrow T * F$ (se quitan 10, 7, 2 y se va a $\text{Ir_a}(0, T) = 2$)
0 2	T	$+ \text{id} \$$	Reducir con (2) $E \rightarrow T$ (se quita 2 y se va a $\text{Ir_a}(0, E) = 1$)
0 1	E	$+ \text{id} \$$	Desplazar e ir al estado 6
0 1 6	E^+	$\text{id} \$$	Desplazar e ir al estado 5
0 1 6 5	$E^+ \text{id}$	$\$$	Reducir con (6) $F \rightarrow \text{id}$ (se quita 5 y se va a $\text{Ir_a}(6, F) = 3$)
0 1 6 3	$E^+ F$	$\$$	Reducir con (4) $T \rightarrow F$ (se quita 3 y se va a $\text{Ir_a}(6, T) = 9$)
0 1 6 9	$E^+ T$	$\$$	Reducir con (1) $E \rightarrow E + T$ (se quitan 9, 6, 1 y se va a $\text{Ir_a}(0, E) = 1$)
0 1	E	$\$$	Aceptar

4. [Análisis sintáctico ascendente.]

Conflictos en las tablas SLR(1)

- Una gramática es SLR(1) si en la tabla de análisis **aparece como máximo una entrada en cada celda.**
- Cuando no es así se pueden señalar dos tipos de conflictos:
 1. **Desplazamiento-reducción.** Sucede cuando en la misma celda aparece una acción de desplazar y una de reducir. Normalmente, para poder continuar el análisis se elige una acción por defecto, generalmente la de desplazar.
 2. **Reducción-reducción.** Sucede cuando, en un estado de la tabla, la misma entrada se puede reducir por dos reglas distintas. La solución más razonable es modificar la gramática para que esto no ocurra.

Ejercicio 10

- Sea la gramática siguiente:

```
<sentencia> ::= = <valorizq> [ <corchete> ]  
<valorizq> ::= id1 | id2  
<corchete> ::= <corchete> + <valorizq> | <valorizq>
```

Se pide:

- Obtener la colección canónica de conjuntos de elementos LR(0).
- Generar la tabla SLR(1) ¿Es una gramática SLR(1)?
- Analizar la cadena =id1[id1+id2].

4. [Análisis sintáctico ascendente.]

Gestión de errores en analizadores SLR(1)

- Los errores sintácticos se producen cuando a partir del estado actual situado en la cima de la pila no hay ninguna transición definida para el símbolo actual de la entrada.
- En esta situación el estado de la pila representa el **contexto a la izquierda** del error, y el resto de la cadena de entrada, el **contexto a la derecha**.
- La recuperación del error se lleva a cabo modificando la pila y/o la cadena de entrada, hasta que el estado y la cadena le permiten al analizador continuar el análisis.
- Mostraremos dos métodos de gestión de errores:
 1. Métodos heurísticos.
 2. Método de análisis de transiciones.

4. [Análisis sintáctico ascendente.]

Gestión de errores en analizadores SLR(1)

- Los **métodos heurísticos** suponen la programación de rutinas específicas para cada celda en la que se produce un error.
- El método de **análisis de transiciones** propone explorar la pila hacia abajo hasta encontrar un estado d con un Ir_a para un no terminal A . Después se descartan cero o más símbolos en la entrada hasta encontrar uno que pueda seguir a A de acuerdo con la gramática. Eso permite meter el estado $Ir_a(d, A)$ en la pila y continuar con el análisis.

[Bibliografía]

- A.V. Aho, R. Sethi, J.D. Ullman. Compiladores. Principios, técnicas y herramientas. 1a edición. Addison Wesley, 1990.
- M. Alfonseca, M. de la Cruz, A. Ortega, E. Pulido. Compiladores e intérpretes: teoría y práctica. Pearson Educación, 2006.