# Parseo y Generación de Código

**Análisis sintáctico descendente**
**Presentación del TP 1**
**(Otras técnicas de análisis sintáctico)**

Licenciatura en Informática con Orientación en Desarrollo de Software
Universidad Nacional de Quilmes

# Análisis sintáctico descendente

# Análisis sintáctico

**Objetivo:** dada una gramática independiente del contexto $G = (N, \Sigma, P, S)$, analizar sintácticamente una cadena $\alpha \in \Sigma^\star$. Es decir, decidir si $\alpha \in L(G)$ y en tal caso dar una derivación para $\alpha$.

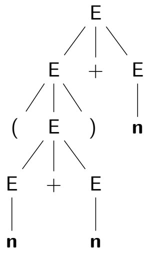
# Análisis sintáctico

**Acciones asociadas a las producciones.**
Generalmente es fácil modificar un método de análisis sintáctico para que cada producción dispare una acción. Esto puede servir, por ejemplo, para construir un AST.

**Ejemplo.** Si la gramática es $G = (\{E\}, \{\mathbf{n}, +, (, )\}, P, E)$:

$$E \to \mathbf{n} \mid E + E \mid (E)$$

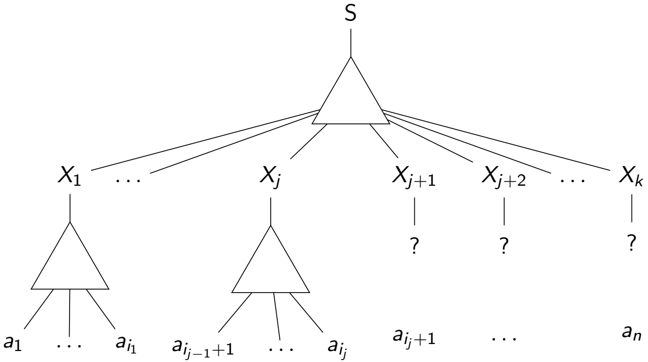
La cadena $\alpha = (\mathbf{n} + \mathbf{n}) + \mathbf{n}$ da lugar al siguiente árbol de derivación, del que se puede extraer recursivamente un AST, ejecutando una acción en el recorrido *post-order*.



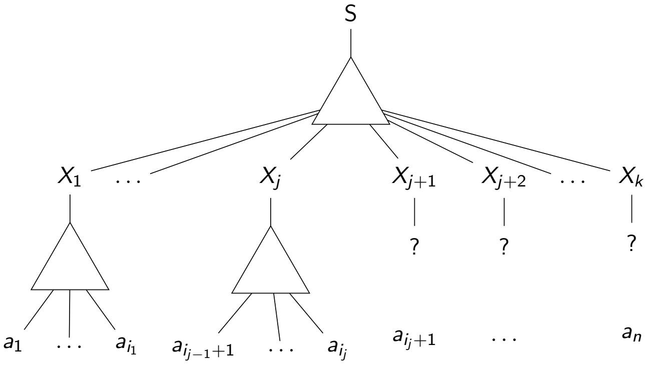
# Análisis sintáctico descendente

En el análisis sintáctico **descendente**, se empieza por la raíz $S$ del árbol, y una cadena $\alpha = a_1 \ldots a_n$ que se quiere analizar sintácticamente.

El análisis trata de construir un árbol de derivación para $S \Rightarrow^\star a_1 \ldots a_n$. En cada paso del análisis, el árbol de derivación está parcialmente construído, y la cadena $\alpha = a_1 \ldots a_n$ ya fue parcialmente consumida.

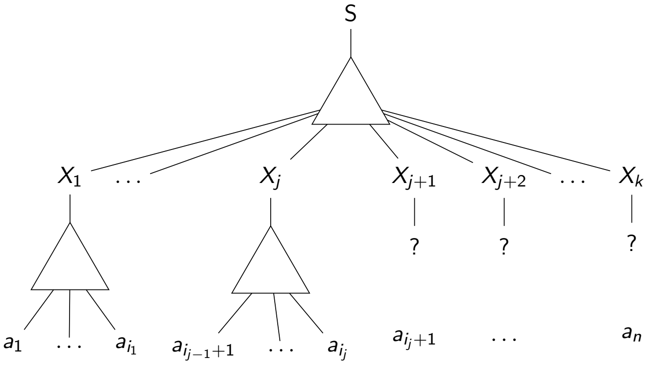


# Análisis sintáctico descendente



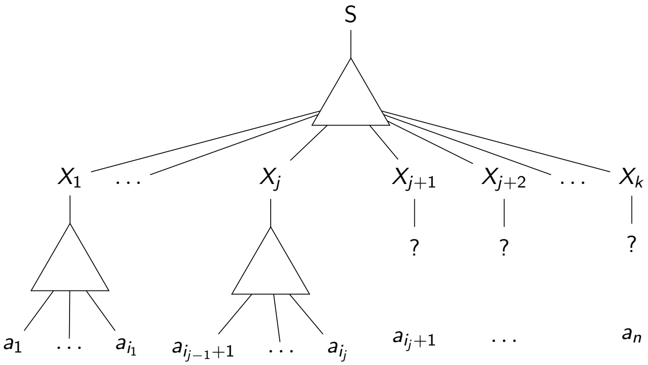

- **Caso 1:** si el símbolo $X_{j+1}$ coincide con el símbolo terminal $a_{i_j+1}$, el nodo $X_{j+1}$ pasa a ser una hoja del árbol de derivación. Se consume el símbolo $a_{i_j+1}$ de la entrada, y se avanza a analizar el símbolo no terminal $X_{j+2}$.

# Análisis sintáctico descendente




- **Caso 2:** si el símbolo $X_{j+1}$ es un símbolo terminal distinto de $a_{i_j+1}$, el árbol de derivación construido hasta el momento no sirve para la cadena $\alpha = a_1 \dots a_n$:
  - Puede ser culpa del usuario (la cadena no está en el lenguaje).
  - Puede ser culpa del parser (hay otra derivación para $\alpha$).

# Análisis sintáctico descendente




- **Caso 3:** si el símbolo $X_{j+1}$ es un símbolo no terminal, se debe expandir $X_{j+1}$ agregándole $r$ hijos, usando alguna de las producciones $X_{j+1} \to Y_1 \dots Y_r$, sin consumir ningún símbolo de la entrada. Se avanza a analizar el símbolo no terminal $Y_1$ (si $r \geq 1$) o el símbolo no terminal $X_{j+2}$ (si $r = 0$).

# Análisis sintáctico descendente predictivo

Como primera aproximación al análsis sintáctico descendente, supongamos que tenemos un “oráculo” que en todo momento puede predecir cuál es la producción correcta que se debe elegir para conseguir una derivación de la cadena de entrada.

# Análisis sintáctico descendente predictivo

```
Entrada: Una gramática G = (N, Σ, P, S) y una cadena α.
Salida:  Una derivación más a la izquierda S ⇒* α
         si es que existe.
  pila := [S]
  while pila no es vacía
     X = pila.pop()
     if X es un símbolo no terminal
        Elegir usando un oráculo una producción X → Y1 ... Yk.
        Agregar X → Y1 ... Yk a la derivación.
        Apilar Y1, ..., Yk, con Y1 en el tope.
     elseif X es un símbolo terminal y coincide
           con el próximo símbolo de la entrada
        Consumir un símbolo de la entrada.
     else
        return Falla: α no está en L(G).
     end
  end
  if toda la entrada fue consumida
     return Éxito: S ⇒* α con la derivación construida.
  else
     return Falla: α no está en L(G).
  end
```

# Análisis sintáctico descendente predictivo

**Observación.** ¿Qué pasa cuando se acaba la entrada? Por ejemplo, la cadena $\epsilon$ con la gramática $S \to a$.

Una práctica común para simplificar la presentación de los algoritmos es agregar un símbolo terminador al final de la entrada, es decir:

- Extender la gramática con un nuevo símbolo inicial $S'$, un nuevo símbolo terminal \$ (el terminador) y una producción $S' \to S\$$.
- Analizar sintácticamente la cadena $\alpha\$$ en lugar de la cadena $\alpha$.

# Análisis sintáctico descendente predictivo

**Ejercicio.** Dibujar la evolución del estado de la pila y de la entrada para un análisis sintáctico descendente de la cadena $(\mathbf{a} \Rightarrow \mathbf{a}) \Rightarrow \mathbf{a}$ con la gramática $G = (\{T, U\}, \{\mathbf{a}, \Rightarrow, (,)\}, P, T)$, suponiendo que se dispone de un oráculo que elige siempre la producción correcta. Las producciones son:

$$
\begin{array}{rcl}
T & \rightarrow & U \\
 & | & U \Rightarrow T \\
U & \rightarrow & \mathbf{a} \\
 & | & (T)
\end{array}
$$

# Análisis sintáctico predictivo con *backtracking*

El algoritmo se puede adaptar para que haga *backtracking*, es decir, para que en caso de falla vuelva hasta el último punto en el que se eligió una producción para elegir otra alternativa.

```
function analizar_bt(G, π, α)
Entrada: Una gramática G = (N, Σ, P, S), una pila de símbolos π = Z_1 ... Z_r, y una cadena α.
Salida:  Una derivación más a la izquierda Z_1 ... Z_r ⇒* α,
         en caso de que dicha derivación exista y el algoritmo termine.
  case π = ϵ
      if α = ϵ return Éxito: ϵ ⇒* ϵ.
      else return Falla.
  case π = Xπ'
      if X es un símbolo no terminal
          foreach producción X → Y_1 ... Y_k
              resultado := analizar_bt(G, Y_1 ... Y_k π', α)
              if resultado = Éxito: Y_1 ... Y_k π' ⇒* α
                  return Éxito: Xπ' ⇒ Y_1 ... Y_k π' ⇒* α
              end
          end
          return Falla.
      elseif α = Xα'
          resultado := analizar_bt(G, π', α')
          if resultado = Éxito: π' ⇒* α'
              return Éxito: Xπ' ⇒* Xα'.
          else
              return Falla.
          end
      else
          return Falla.
      end
  end
```

# Análisis sintáctico predictivo con *backtracking*

**Nota importante:** el algoritmo de análisis sintáctico predictivo con *backtracking* no funciona para cualquier gramática.

Por ejemplo, en general el algoritmo puede no terminar si la gramática tiene recursión a izquierda, es decir un símbolo $A$ tal que $A \Rightarrow^{+} A\alpha$.

Se pueden dar condiciones suficientes para que el algoritmo termine; por ejemplo, que la gramática sea $LL(k)$.

Además, aun si termina, puede tardar tiempo exponencial.

# Análisis sintáctico predictivo con *backtracking*

**Ejercicio.** Aplicar el algoritmo de análisis sintáctico predictivo con *backtracking* para las cadenas $a \bullet a$ y $a\bullet$ con las siguientes gramáticas.

1. $G_1 = (\{S\}, \{\mathbf{a}, \bullet\}, P, S)$ con las siguientes producciones:

$$S \rightarrow \mathbf{a} \mid S \bullet \mathbf{a}$$

2. $G_2 = (\{S, S'\}, \{\mathbf{a}, \bullet\}, P, S)$ con las siguientes producciones:

$$\begin{array}{rcl} S & \rightarrow & \mathbf{a}S' \\ S' & \rightarrow & \epsilon \mid \bullet\mathbf{a}S' \end{array}$$

# Análisis sintáctico predictivo con *backtracking*

**Ejercicio.** Aplicar el algoritmo de análisis sintáctico predictivo con *backtracking* para la cadena *aacc* con la siguiente gramática $G = (\{S\}, \{a, b, c\}, P, S)$:

$$S \rightarrow \epsilon \mid aSb \mid aSc$$

# Análisis sintáctico LL(1)

En las técnicas que vimos:

- El análisis con oráculo supone que siempre se predice la producción correcta.
- El análisis con *backtracking* prueba todas las alternativas posibles. En el peor caso no termina, y aun si termina tiene complejidad exponencial.

La técnica que veremos a continuación es el análisis LL(1):

- En general, el análisis sintáctico LL($k$) predice la producción correcta mirando los siguientes $k$ símbolos de la entrada.

# Análisis sintáctico LL(1)

**Ejercicio.** Hacer análisis sintáctico descendente de la expresión $(\mathbf{n} + \mathbf{n})/\mathbf{n}$ con la siguiente gramática:
$G = (\{E, E', T, T', F\}, \{\mathbf{n}, +, -, *, /, (, )\}, P, E)$:

$$
\begin{array}{lclclcl}
E & \rightarrow & TE' & & & & \\
E' & \rightarrow & \epsilon & \mid & +TE' & \mid & -TE' \\
T & \rightarrow & FT' & & & & \\
T' & \rightarrow & \epsilon & \mid & *FT' & \mid & /FT' \\
F & \rightarrow & \mathbf{n} & \mid & (E) & &
\end{array}
$$

Convencerse de que el oráculo podría elegir la producción siguiente usando únicamente un token de *lookahead*.

# Análisis sintáctico LL(1)

Empezaremos viendo el caso particular de los parsers LL(1).
Necesitamos dos definiciones previas:

- **Primeros (FIRST).**
  Dada una cadena de símbolos terminales y no terminales $\alpha \in (\Sigma \cup N)^\star$, el conjunto $\mathrm{FIRST}(\alpha) \subseteq \Sigma \cup \{\epsilon\}$ de los "primeros" de $\alpha$ es un conjunto de símbolos terminales (y posiblemente $\epsilon$) definido como sigue:

$$\begin{aligned}\mathrm{FIRST}(\alpha) = \ & \{a \in \Sigma \mid \alpha \Rightarrow^\star a\beta \text{ para } \beta \in (N \cup \Sigma)^\star\} \cup \\ & \textbf{if } \alpha \Rightarrow^\star \epsilon \textbf{ then } \{\epsilon\} \textbf{ else } \emptyset\end{aligned}$$

- **Siguientes (FOLLOW).**
  Dado un símbolo no terminal $A \in N$, el conjunto $\mathrm{FOLLOW}(A) \subseteq \Sigma \cup \{\$\}$ de los "siguientes" de $A$ es un conjunto de símbolos terminales (y posiblemente $) definido como sigue:

$$\begin{aligned}\mathrm{FOLLOW}(A) = \ & \{x \in \Sigma \mid S \Rightarrow^\star \alpha A x \beta \text{ para } \alpha, \beta \in (N \cup \Sigma)^\star\} \cup \\ & \textbf{if } S \Rightarrow^\star \alpha A \textbf{ then } \{\$\} \textbf{ else } \emptyset\end{aligned}$$

# Análisis sintáctico LL(1)

## Cómputo del conjunto FIRST para un símbolo.

*Entrada:* Una gramática $G = (N, \Sigma, P, S)$.
*Salida:* El conjunto $\mathrm{FIRST}(X)$
para cada símbolo $X \in N \cup \Sigma$.

```
𝓕 := un diccionario {X ↦ ∅ | X ∈ N ∪ Σ}
foreach símbolo terminal a ∈ Σ
    𝓕[a] := {a}
end
while hay algún cambio
    foreach producción (A → X_1 ... X_n) ∈ P
        for i = 1 to n
            if ε ∈ 𝓕[X_1] ∩ ... ∩ 𝓕[X_{i-1}]
                𝓕[A] := 𝓕[A] ∪ (𝓕[X_i] \ {ε})
            end
        end
        if ε ∈ 𝓕[X_1] ∩ ... ∩ 𝓕[X_n]
            𝓕[A] := 𝓕[A] ∪ {ε}
        end
    end
end
return 𝓕
```

# Análisis sintáctico LL(1)

**Observación.** Notar que el algoritmo para calcular el conjunto FIRST sirve en particular para calcular el conjunto de símbolos anulables: un símbolo $X$ es anulable si y sólo si $\epsilon \in \mathsf{FIRST}(X)$.

# Análisis sintáctico LL(1)

**Ejercicio.** Calcular el conjunto FIRST para todos los símbolos de la gramática

$$G = (\{S, V, D, T, T', U\}, \{\mathbf{id}, \mathbf{int}, \mathbf{bool}, \Rightarrow, ;, (, )\}, P, S)$$

$$\begin{array}{rcl}
S & \rightarrow & VS \\
 & | & \epsilon \\
V & \rightarrow & D\ \mathbf{id}; \\
D & \rightarrow & T \\
 & | & \epsilon \\
T & \rightarrow & UT' \\
T' & \rightarrow & \Rightarrow UT' \\
 & | & \epsilon \\
U & \rightarrow & \mathbf{int} \\
 & | & \mathbf{bool} \\
 & | & (T)
\end{array}$$

# Análisis sintáctico LL(1)

**Cómputo del conjunto FIRST para una cadena.**

```
Entrada: Una gramática G = (N, Σ, P, S)
         y una cadena X_1 ... X_n ∈ (N ∪ Σ)*.
Salida:  El conjunto FIRST(α).
  Computar el diccionario 𝓕
    que a cada símbolo X ∈ N ∪ Σ le asocia FIRST(X).
  𝓡 := ∅
  for i = 1 to n
      if ϵ ∈ 𝓕[X_1] ∩ ... ∩ 𝓕[X_{i-1}]
          𝓡 := 𝓡 ∪ (𝓕[X_i] \ {ϵ})
      end
  end
  if ϵ ∈ 𝓕[X_1] ∩ ... ∩ 𝓕[X_n]
      𝓡 := 𝓡 ∪ {ϵ}
  end
  return 𝓡
```

**Cómputo del conjunto FOLLOW.**

*Entrada:* Una gramática $G = (N, \Sigma, P, S)$.
*Salida:* El conjunto $\mathsf{FOLLOW}(A)$
para cada símbolo no terminal $A \in N$.

$\mathcal{W}$ := un diccionario $\{X \mapsto \emptyset \quad | \quad X \in N \cup \Sigma\}$
$\mathcal{W}[S]$ := $\{\$\}$
**foreach** producción $A \to \alpha B \beta$
    $\mathcal{W}[B]$ := $\mathcal{W}[B] \cup (\mathsf{FIRST}(\beta) \setminus \{\epsilon\})$
**end**
**while** hay algún cambio
    **foreach** producción $A \to \alpha B \beta$ tal que $\epsilon \in \mathsf{FIRST}(\beta)$
        $\mathcal{W}[B]$ := $\mathcal{W}[B] \cup \mathcal{W}[A]$
    **end**
**end**
**return** $\mathcal{W}$

# Análisis sintáctico LL(1)

**Ejercicio.** Calcular el conjunto FOLLOW para todos los símbolos no terminales de la gramática

$$G = (\{S, V, D, T, T', U\}, \{\textbf{id}, \textbf{int}, \textbf{bool}, \Rightarrow, ;, (, )\}, P, S)$$

$$
\begin{array}{rcl}
S & \rightarrow & VS \\
 & | & \epsilon \\
V & \rightarrow & D\ \textbf{id}; \\
D & \rightarrow & T \\
 & | & \epsilon \\
T & \rightarrow & UT' \\
T' & \rightarrow & \Rightarrow UT' \\
 & | & \epsilon \\
U & \rightarrow & \textbf{int} \\
 & | & \textbf{bool} \\
 & | & (T)
\end{array}
$$

# Análisis sintáctico LL(1)

Dada una gramática $G = (N, \Sigma, P, S)$, una **tabla de análisis sintáctico LL(1)** es un diccionario $\mathcal{T}$ que a cada par $(A, b) \in N \times (\Sigma \cup \{\$\})$ le asocia un conjunto de producciones $(C \to \alpha) \in P$. Se construye con el siguiente método:

*Entrada:* Una gramática $G = (N, \Sigma, P, S)$.
*Salida:* La tabla de análisis sintáctico LL(1) para $G$.
Poner $\mathcal{T}[A, b] := \emptyset$ para todo $(A, b) \in N \times \Sigma$.
**foreach** producción $(A \to \alpha) \in P$
    **foreach** símbolo terminal $x \in (\mathrm{FIRST}(\alpha) \setminus \{\epsilon\})$
        $\mathcal{T}[A, x]$ := $\mathcal{T}[A, x] \cup \{A \to \alpha\}$
    **end**
    **if** $\epsilon \in \mathrm{FIRST}(\alpha)$
        **foreach** símbolo $x \in \mathrm{FOLLOW}(A)$
            $\mathcal{T}[A, x]$ := $\mathcal{T}[A, x] \cup \{A \to \alpha\}$
        **end**
    **end**
**end**
**return** $\mathcal{T}$

# Análisis sintáctico LL(1)

Si la tabla de análisis sintáctico LL(1) verifica que todos los conjuntos $\mathcal{T}[A, b]$ tienen a lo sumo una entrada, se dice que la gramática es **LL(1)**.

En ese caso, la tabla de análisis sintáctico LL(1) funciona como oráculo para el análisis sintáctico predictivo.

# Análisis sintáctico LL(1)

**Ejercicio.** Calcular la tabla de análisis sintáctico LL(1) para la gramática

$$G = (\{S, V, D, T, T', U\}, \{\mathbf{id}, \mathbf{int}, \mathbf{bool}, \Rightarrow, ;, (, )\}, P, S)$$

$$\begin{array}{lcl} S & \to & VS \mid \epsilon \\ V & \to & D\ \mathbf{id}; \\ D & \to & T \mid \epsilon \end{array} \qquad \begin{array}{lcl} T & \to & UT' \\ T' & \to & \Rightarrow UT' \mid \epsilon \\ U & \to & \mathbf{int} \mid \mathbf{bool} \mid (T) \end{array}$$

Recordar que:

| | FIRST | FOLLOW |
|---|---|---|
| $S$ | $\epsilon$ **id int bool (** | \$ |
| $V$ | **id int bool (** | **id int bool (** \$ |
| $D$ | $\epsilon$ **int bool (** | **id** |
| $T$ | **int bool (** | **id )** |
| $T'$ | $\Rightarrow$ $\epsilon$ | **id )** |
| $U$ | **int bool (** | $\Rightarrow$ **id )** |

# Análisis sintáctico LL(1)

**Ejercicio.** Usando la tabla de análisis sintáctico LL(1) para la gramática anterior, analizar sintácticamente **int** $\Rightarrow$ **int id**;.

# Análisis sintáctico LL(1)

**Ejercicio.** Mostrar que la gramática
$G = (\{S, E\}, \{\textbf{if}, \textbf{then}, \textbf{else}, \textbf{cmd}, \textbf{exp}\}, P, S)$ no es LL(1):

$$
\begin{array}{rcl}
S & \rightarrow & \textbf{if } E \textbf{ then } S \\
  & | & \textbf{if } E \textbf{ then } S \textbf{ else } S \\
  & | & \textbf{cmd} \\
E & \rightarrow & \textbf{exp}
\end{array}
$$

# Análisis sintáctico LL(1)

- Un símbolo $X \in N \cup \Sigma$ es **inútil** si no hay ninguna derivación de la forma $S \Rightarrow^\star \alpha X \beta \Rightarrow^\star \gamma \in \Sigma^\star$, es decir, $X$ nunca puede aparecer en la derivación de una cadena $\gamma \in \Sigma^\star$.

  Una gramática recursiva a izquierda sin símbolos inútiles no puede ser LL(1). (¿Por qué?).

- Una gramática LL(1) no puede ser ambigua. (¿Por qué?).

- ¿Cuánto tiempo toma un parser descendente predictivo en analizar sintácticamente una cadena $\alpha$ si la gramática es LL(1) y se dispone de la tabla de análisis sintáctico LL(1)?

# Análisis sintáctico LL($k$)

El análisis sintáctico LL(1) se puede generalizar al análisis LL($k$) en el que el parser puede contemplar los siguientes $k$ símbolos de la entrada.

- La construcción es bastante más complicada.
- Se consideraba impracticable hasta mediados de los 1990. (Observar que el tamaño de la tabla es $|N| \cdot |\Sigma|^k$. Para números típicos como $|N| = 50$, $|\Sigma| = 128$, $k = 4$ *a priori* la tabla podría tener 13.421.772.800 entradas).
- La ventaja de los parsers LL($k$) es su expresividad: hay gramáticas que son LL($k + 1$) pero no son LL($k$).

# Técnicas surtidas de análisis sintáctico

# Algoritmo de Earley

# Algoritmo de Earley

El **algoritmo de Earley** es un algoritmo general de análisis sintáctico. Su interés es que funciona para gramáticas independientes del contexto **arbitrarias**.

# Algoritmo de Earley

- Si $A \to \alpha\beta$ es una producción, la notación $A \to \alpha \bullet \beta$ representa una situación en la que se consumió $\alpha$ y se espera leer $\beta$.
- Si la entrada que se quiere analizar es $a_1 \dots a_n$ hay $n+1$ posiciones en la entrada:

$$
\begin{array}{ccccccccccc}
 & a_1 & & a_2 & & a_3 & \dots & a_{n-1} & & a_n & \\
\uparrow & & \uparrow & & \uparrow & & & & \uparrow & & \uparrow \\
0 & & 1 & & 2 & & & & n-1 & & n
\end{array}
$$

- Para cada posición de la entrada $0 \leq k \leq n$, el parser genera un conjunto de estados $\mathcal{S}[k]$. Cada estado es de la forma $(A \to \alpha \bullet \beta,\ i)$ donde:
  - El número $0 \leq i \leq n$ representa la posición en la que comenzó la aplicación de la producción $A \to \alpha\beta$.
  - La cadena $\alpha$ es el fragmento de la producción $A \to \alpha\beta$ que ya fue consumido hasta la posición $k$.
  - La cadena $\beta$ es el fragmento aún no consumido.

# Algoritmo de Earley

```
Entrada: Una gramática G = (N, Σ, P, S),
         una cadena de entrada α = a_1 ... a_n.
Salida:  Un booleano indicando si α ∈ L(G).

  Crear un arreglo S[0..n] de conjuntos de estados.
  Poner S[k] = ∅ para cada 0 ≤ k ≤ n.
  S[0] := {(S' → •S, 0)}
  for k = 0 to n
      while hay algún estado de S[k] no visitado
          Sea (A → α•β, j) un estado no visitado de S[k].
          Marcar (A → α • β, j) como visitado.
          if β comienza con un símbolo no terminal
              predecir((A → α • β, j), k)
          elseif β comienza con un símbolo terminal
              avanzar((A → α • β, j), k)
          else /* β es vacía */
              completar((A → α • β, j), k)
          end
      end
  end
  return (S' → S•, 0) ∈ S[n]
```

# Algoritmo de Earley

```
// Si lo que sigue es un símbolo no terminal.
function predecir((A → α • Bβ, j), k)
    foreach producción B → γ
        Agregar (B → •γ, k) a S[k].
    end
end

// Si lo que sigue es un símbolo terminal.
function avanzar((A → α • bβ, j), k)
    if b = a_k
        Agregar (A → αb • β, j) a S[k + 1].
    end
end

// Si la producción termina.
function completar((A → α•, j), k)
    foreach estado (B → β • Aγ, i) ∈ S[j]
        Agregar (B → βA • γ, i) a S[k].
    end
end
```

# Algoritmo de Earley

**Ejercicio.** Analizar la cadena $(\mathbf{n} + \mathbf{n})$ con la gramática $G = (\{E, T\}, \{\mathbf{n}, +, (, )\}, P, E)$, con producciones:

$$\begin{array}{rcl} E & \rightarrow & T \\ & | & E + T \\ T & \rightarrow & \mathbf{n} \\ & | & (E) \end{array}$$

# Análisis sintáctico con combinadores

# Análisis sintáctico con combinadores

Un analizador sintáctico que procesa la entrada de izquierda a derecha se puede pensar como una función que:

- Recibe una entrada.
- Analiza sintácticamente algún prefijo de la entrada.
- Devuelve el resultado de hacer el análisis sintáctico y el sufijo de la entrada que todavía no fue analizado. (Alternativamente, puede fallar).

# Análisis sintáctico con combinadores

Por ejemplo, en Haskell:

```
data ErrorMsg = String
type Parser a = String -> Either ErrorMsg (a, String)

--
-- match1 c    Es un parser básico que acepta
--             el lenguaje {c}
--
match1 :: Char -> Parser Char
match1 c [] = Left ("Esperaba " ++ show c ++
                    " pero se encuentra EOF")
match1 c (c' : s')
  | c == c'    = Right (c, s')
  | otherwise = Left ("Esperaba " ++ show c ++
                      " pero se encuentra " ++ show c')
```

# Análisis sintáctico con combinadores

Se pueden construir **combinadores** de parsers: son funciones que reciben parsers y devuelven parsers.
Combinador de secuenciación binaria:

```
--
-- seqP p1 p2   Parser que acepta la concatenación
--              de los lenguajes aceptados por p1 y p2.
--
seqP :: Parser a -> Parser b -> Parser (a, b)
seqP p1 p2 s =
  case p1 s of
    Left errmsg -> Left errmsg
    Right (a, s') ->
      case p2 s' of
        Left errmsg    -> Left errmsg
        Right (b, s'') -> Right ((a, b), s'')
```

# Análisis sintáctico con combinadores

Combinador de alternativa binaria:

```
--
-- altP p1 p2   Parser que acepta la unión
--              de los lenguajes aceptados por p1 y p2.
--
-- Nota: hace backtracking.
--
altP :: Parser a -> Parser b -> Parser (Either a b)
altP p1 p2 s =
  case p1 s of
    Right (a, s') -> Right (Left a, s')
    Left errmsg ->
      case p2 s of
        Left errmsg -> Left ("La entrada no está en " ++
                             "ninguno de los lenguajes.")
        Right (b, s') -> Right (Right b, s')
```

# Análisis sintáctico con combinadores

Combinador de repetición:

```
--
-- repP p   Parser que acepta la "repetición"
--          del lenguaje aceptado por p.
--
-- Nota: no acepta exactamente la clausura de Kleene,
-- porque usa la regla de "maximal munch", es decir,
-- trata de consumir el prefijo de la entrada más
-- largo posible.
--
repP :: Parser a -> Parser [a]
repP p s =
  case p s of
    Left errmsg   -> Right []
    Right (a, s') ->
      case repP p s' of
        Left errmsg -> Left errmsg
        Right (as, s'') -> Right (a : as, s'')
```

# Análisis sintáctico con combinadores

Se pueden agregar acciones para construir distintos tipos de resultados:

```
mapP :: (a -> b) -> Parser a -> Parser b
mapP f p s =
  case p s of
    Left errmsg -> Left errmsg
    Right (a, s') -> Right (f a, s')
```

# Análisis sintáctico con combinadores

Por conveniencia se pueden crear combinadores de secuenciación o alternativa *n*-arios, por ejemplo:

```
seq4P :: Parser a -> Parser b -> Parser c -> Parser d ->
         Parser (a, b, c, d)
seq4P p1 p2 p3 p4 =
  mapP (\ (a, (b, (c, d))) -> (a, b, c, d))
       (seqP p1 (seqP p2 (seqP p3 p4)))
```

# Análisis sintáctico con combinadores

Se pueden crear combinadores de repetición con distintas restricciones sobre la cantidad de repeticiones, p.ej. repetición al menos una vez:

```
rep1P :: Parser a -> Parser [a]
rep1P p = mapP (\ (a, as) -> a : as) (seqP p (repP p))
```

# Análisis sintáctico con combinadores

Se pueden generalizar los combinadores de secuencia y alternativa para combinar listas de parsers. Por simplicidad supondremos que las listas no están vacías.

```
seqsP :: [Parser a] -> Parser [a]
seqsP [p]      = mapP (\ a -> [a]) p
seqsP (p : ps) = mapP (\ (a, as) -> a : as)
                      (seqP p (seqsP ps))

altsP :: [Parser a] -> Parser a
altsP [p]      = p
altsP (p : ps) = mapP f (altP p (altsP ps))
  where f (Left a)  = a
        f (Right a) = a
```

# Análisis sintáctico con combinadores

Ejemplo de un parser sencillo:

```
digitP = altsP (map match1 ['0','1','2','3','4',
                            '5','6','7','8','9'])
numP   = mapP (\ s -> read s :: Integer)
              (rep1P digitP)
spaceP  = altsP (map match1 [' ', '\t', '\r', '\n'])
spacesP = repP spaceP
exprP  = altsP [
  mapP (\ (a, _, _, _, b) -> a + b)
       (seq5P termP spacesP (match1 '+') spacesP exprP),
  termP ]
termP  = altsP [
  mapP (\ (a, _, _, _, b) -> a * b)
       (seq5P factorP spacesP (match1 '*') spacesP termP),
  factorP ]
factorP  = altsP [
   numP,
   mapP (\ (_, a, _) -> a)
        (seq3P (match1 '(') exprP (match1 ')')) ]
```

# Análisis sintáctico con combinadores

**Nota:** el análisis sintáctico basado en combinadores no es una técnica propia de los lenguajes funcionales. Por ejemplo los parsers y combinadores de parsers se pueden representar como objetos:

```
class Match1(object):

  def __init__(self, c):
      self.c = c

  def analizar(self, entrada):
      if entrada.peek() == self.c:
          entrada.read()
          return self.c
      else:
          raise Exception("Esperaba '%c'" % (self.c,))
```

# Análisis con tablas de precedencia

# Análisis con tablas de precedencia

Supongamos que tenemos una tabla de precedencia de operadores:

| Nivel | Operador | Asociatividad |
|---|---|---|
| 1 | $\theta_1$ | $assoc_1$ |
| $\vdots$ | $\vdots$ | |
| $M$ | $\theta_M$ | $assoc_M$ |

donde $\theta_1, \ldots, \theta_M \in \Sigma$ son símbolos terminales, $\theta_i$ tiene menor precedencia que $\theta_{i+1}$, es decir, se espera que $E_1\ \theta_i\ E_2\ \theta_{i+1}\ E_3$ se analice como $E_1\ \theta_i\ (E_2\ \theta_{i+1}\ E_3)$, y la asociatividad $assoc_i$ está en $\{\text{left}, \text{right}, \text{prefix}, \text{suffix}\}$.

# Análisis con tablas de precedencia

```
function analizar_expresión(nivel)
    if nivel = M
        if proximo_token() = "("
            consumir_token("(")
            e := analizar_expresión(1)
            consumir_token(")")
            return e
        else
            return analizar_expresión_atómica() // (TODO)
        end
    else
        case assoc_nivel = left
            return analizar_izquierdo(nivel)
        case assoc_nivel = right
            return analizar_derecho(nivel)
        case assoc_nivel = prefix
            return analizar_prefijo(nivel)
        case assoc_nivel = suffix
            return analizar_sufijo(nivel)
        end
    end
end
```

# Análisis con tablas de precedencia

```
function analizar_izquierdo(nivel)
    e := analizar_expresión(nivel + 1)
    while siguiente_token = θ_nivel
        consumir_token("θ_nivel")
        e := new AST("θ_nivel", e, analizar_op(nivel + 1))
    end
    return e
end

function analizar_derecho(nivel)
    lista := []
    lista.push(analizar_op(nivel + 1))
    while siguiente_token = θ_nivel
        consumir_token("θ_nivel")
        lista.push(analizar_op(nivel + 1))
    end
    e := lista[lista.size - 1];
    for i = lista.size - 2 downto 0
        e := new AST("θ_nivel", lista[i], e)
    end
    return e
end
```

# Análisis con tablas de precedencia

```
function analizar_prefijo(nivel)
    if siguiente_token = θ_nivel
        consumir_token(θ_nivel)
        e := analizar_expresión(nivel)
        return new AST("θ_nivel", e)
    else
        return analizar_expresión(nivel + 1)
    end
end

function analizar_sufijo(nivel)
    e := analizar_expresión(nivel + 1)
    while siguiente_token = θ_nivel
        consumir_token(θ_nivel)
        e := new AST("θ_nivel", e)
    end
    return e
end
```