Tema 2: Lenguajes Independientes del Contexto

Lección 2.3

Simplificación de gramáticas

Jordi Bernad

Índice

Árboles de derivación

- Las palabras generadas mediante una gramática se corresponden con derivaciones desde la variable inicial.

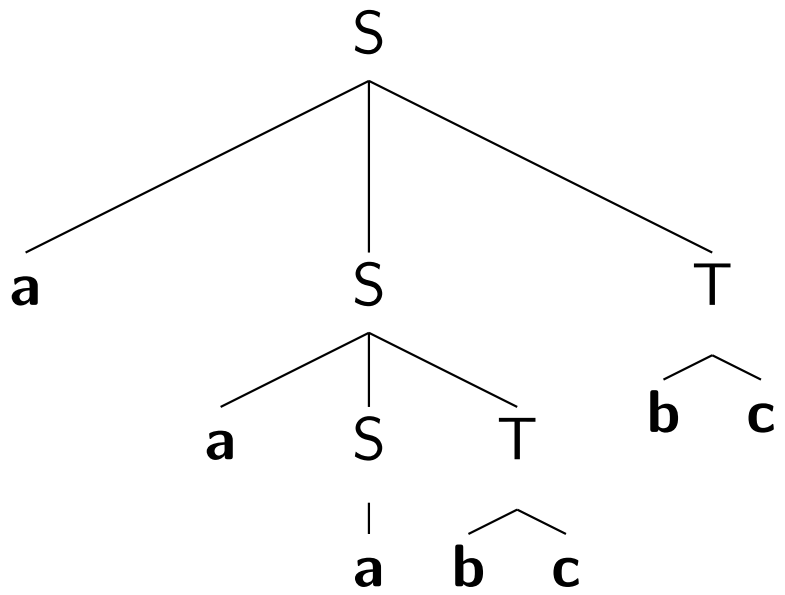
$$S \rightarrow aST|T|Sa|a$$
$$T \rightarrow bTc|bc$$

$$S \Rightarrow aST \Rightarrow aaSTT \Rightarrow aaSbcT \Rightarrow aaabcT \Rightarrow aaabcbc$$

Árboles de derivación

- Toda derivación se puede escribir en forma de **árbol de derivación**

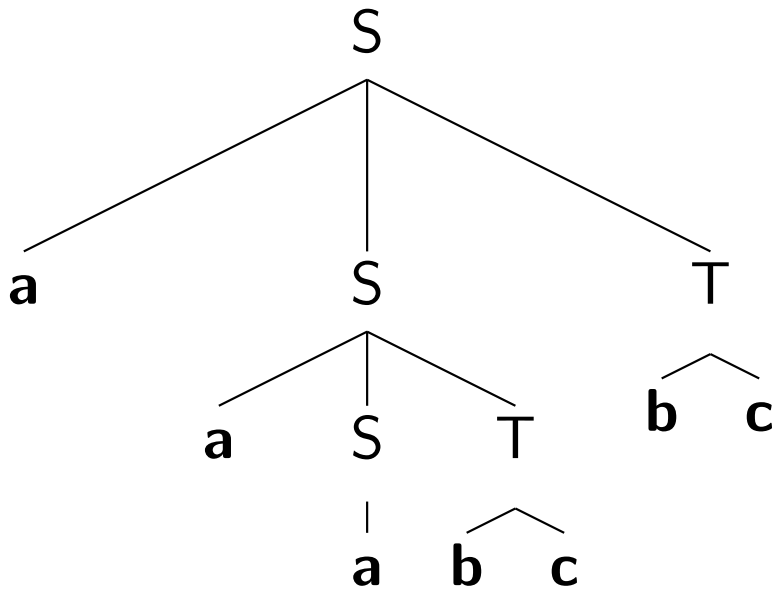
$$S \Rightarrow aST \Rightarrow aaSTT \Rightarrow aaSbcT \Rightarrow aaabcT \Rightarrow aaabcbc$$



Árboles de derivación

- Si cambiamos el orden en el que se producen las sustituciones, llegamos al mismo árbol de derivación

$$S \Rightarrow aST \Rightarrow aSbc \Rightarrow aaSTbc \Rightarrow aaSbcbc \Rightarrow aaabcbc$$



- La **derivación por la izquierda (derecha)** es aquella que se sustituye siempre la variable que está más la izquierda (derecha)

$$S \Rightarrow aST \Rightarrow aaSTT \Rightarrow aaaTT \Rightarrow aaabcT \Rightarrow aaabcbc$$

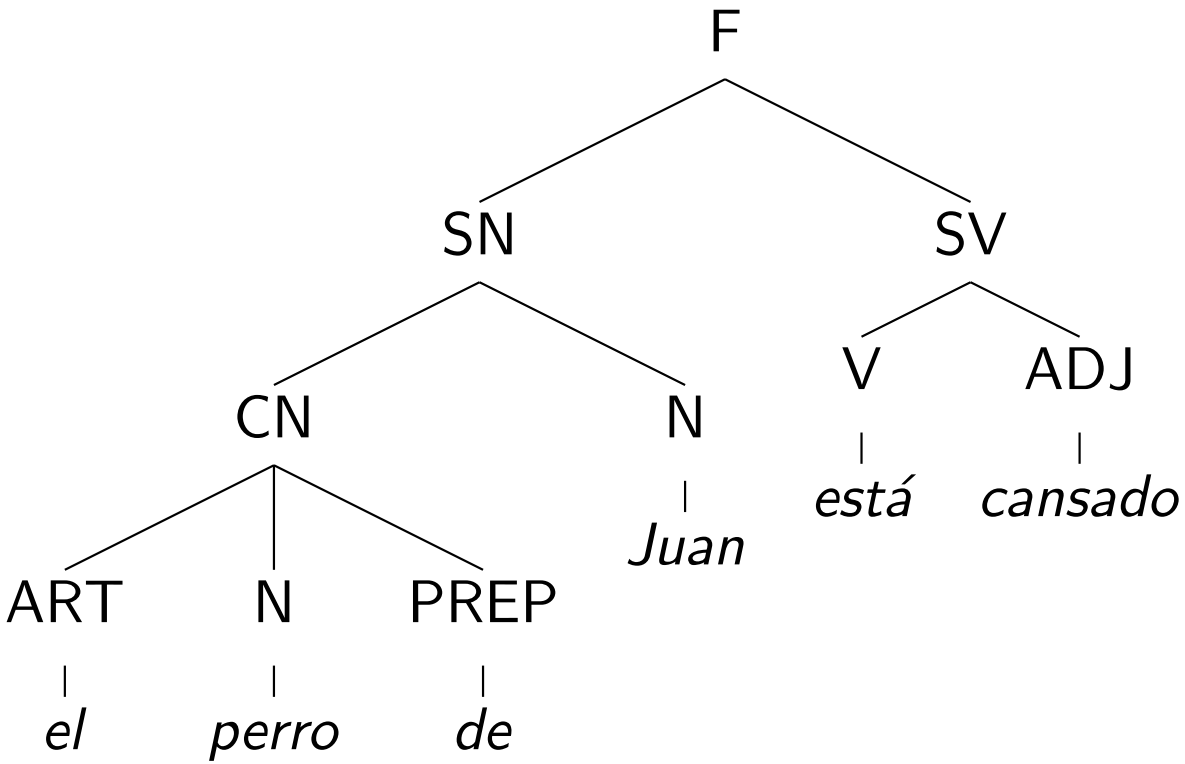
¿Por qué nos interesan los árboles de derivación

- Un **parser o analizador sintáctico** es un programa que construye el árbol de derivación para una cadena w, y decide si w es generado por la gramática.
- Todos los compiladores de los lenguajes de programación tienes su propio parser.
- Al crear el árbol de derivación no solo podemos hallar si el programa es sintácticamente correcto.
- Además, podemos saber cómo se genera, y crear el ejecutable
- Y si hay algún error, notificarlo.

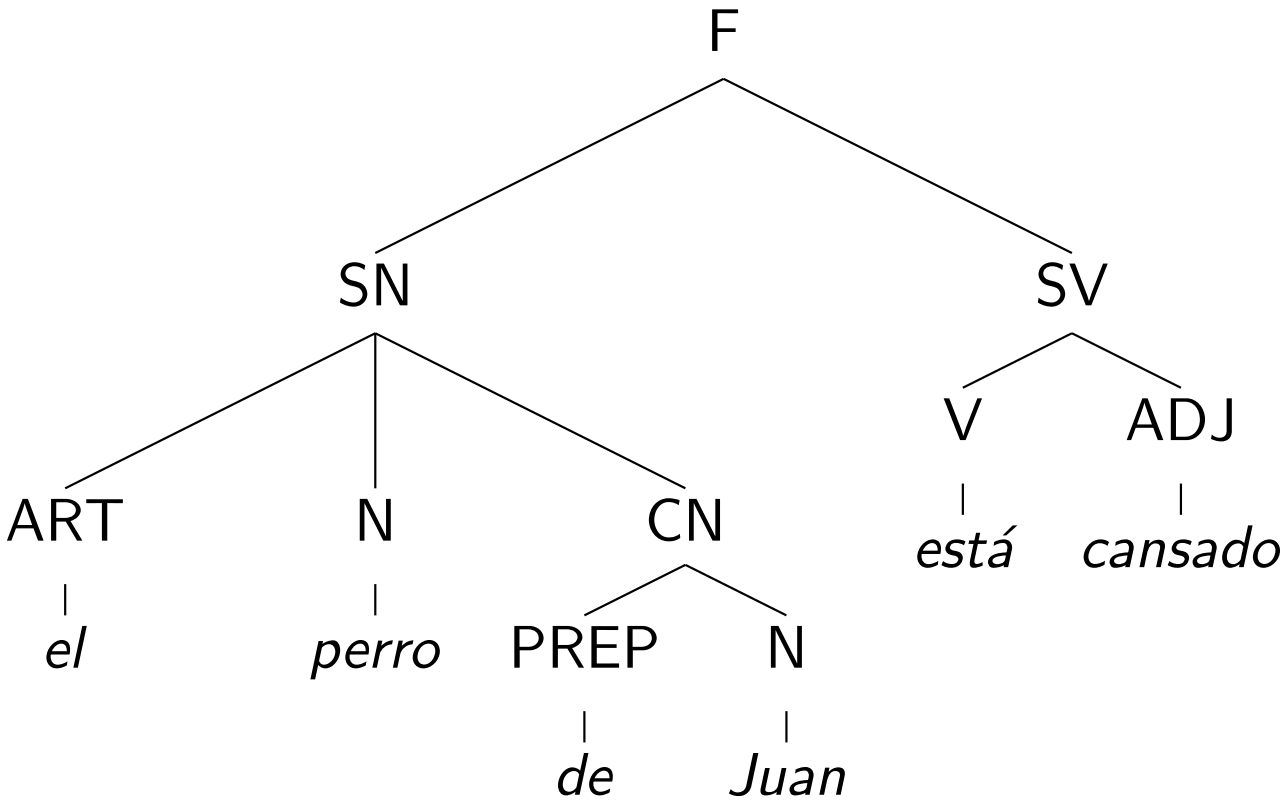
Frases ambiguas

- Consideremos la frase *El perro de Juan está cansado*
- Es una frase ambigua
- ¿Quién esta cansado Juan o su perro?
- Si analizamos sintácticamente la frase, tenemos dos árboles de derivación distintos.

Ambigüedad *El perro de Juan está cansado*



Ambigüedad *El perro de Juan está cansado*



Ambigüedad de gramáticas

- Un parser que analizase el lenguaje natural buscando el significado semántico (no solo si es sintácticamente correcto) de la frase

  El perro de Juan está cansado

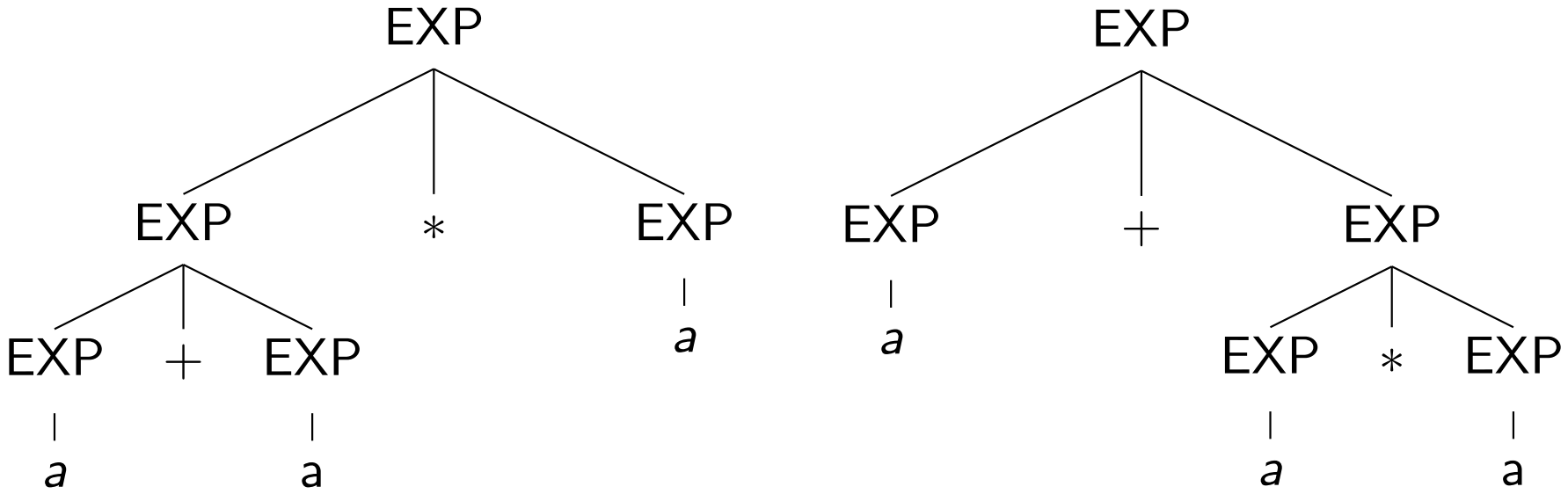
  se encontraría con dos posibles significados.
- ¿Y si una situación similar se produjese en las gramáticas de los lenguajes de programación?
- Un mismo programa tendría dos posibles significados, dos posibles ejecutables.
- No se debe producir tal situación.

Ejemplo gramática ambigua

- La siguiente gramática es ambigua sobre el alfabeto $\Sigma = \{a, +, *, (,)\}$

$$\text{EXP} \rightarrow \text{EXP} + \text{EXP} \mid \text{EXP} * \text{EXP} \mid (\ \text{EXP}\ ) \mid a$$

- $w = a + a * a$ tiene dos árboles de derivación distintos



Desambiguación

- La gramática anterior se puede expresar de forma distinta para que no sea ambigua.

$$\begin{aligned} \text{EXP} &\rightarrow \text{EXP} + \text{TERM} \mid \text{TERM} \\ \text{TERM} &\rightarrow \text{TERM} * \text{FACTOR} \mid \text{FACTOR} \\ \text{FACTOR} &\rightarrow (\ \text{EXP}\ ) \mid a \end{aligned}$$

- Se incluyen nuevas reglas para expresar que $*$ tiene mayor precedencia que $+$

Gramáticas ambiguas

Definición

Una gramática G se dice **ambigua** si existe una cadena generada por G, $w \in L(G)$, para la que existe más de una derivación por la izquierda. O equivalentemente, más de un árbol de derivación distinto para w.

- No siempre podremos dar una gramática equivalente no ambigua.
- $L = \{a^i b^j c^k \mid i = j \text{ ó } j = k\}$: cualquier gramática que genere L es ambigua.
- Se dice un lenguaje **inherentemente ambiguo**
- No nos interesan las gramáticas ambiguas
- Buscamos gramáticas lo más simple posibles.

Forma normal de Chomsky

Definición

Una gramática independiente del contexto se dice que está en **formal normal de Chomsky** si todas las reglas tienen la forma

$$A \rightarrow BC$$
$$A \rightarrow a$$
$$S \rightarrow \epsilon$$

donde: A es cualquier variable; B, C es cualquier variable que no sea la variable inicial S; y a es un símbolo terminal.

Forma normal de Chomsky

Teorema

Toda gramática independiente del contexto tiene una gramática equivalente en forma normal de Chomsky.

La demostración es constructiva. Se basa en transformar todas las reglas que no están en forma normal de Chomsky

1. Añadir una nueva variable S_0, que será la nueva variable inicial, y añadir la regla $S_0 \to S$
2. Eliminar las ϵ-producciones: eliminamos todas las reglas de la forma $A \to \epsilon$ (salvo si aparece $S_0 \to \epsilon$)
3. Eliminar las reglas unarias: eliminamos las reglas de la forma $A \to B$
4. Convertir las reglas que quedan a la forma

$$A \to BC$$

$$A \to a$$

Ejemplo hallar forma normal de Chomsky: paso 1

$$S \rightarrow ASA|aB$$
$$A \rightarrow B|S$$
$$B \rightarrow b|\epsilon$$

$$\mathbf{S_0 \rightarrow S}$$
$$S \rightarrow ASA|aB$$
$$A \rightarrow B|S$$
$$B \rightarrow b|\epsilon$$

Eliminación de ϵ-producciones

Repetir mientras quede alguna regla $A \to \epsilon$ que no sea $S_0 \to \epsilon$:

a. Quitar la regla $A \to \epsilon$
b. Añadir A al conjunto Quitadas

Eliminación de ϵ-producciones

Repetir mientras quede alguna regla $A \to \epsilon$ que no sea $S_0 \to \epsilon$:

a. Quitar la regla $A \to \epsilon$
b. Añadir A al conjunto Quitadas
c. Para cada regla que tenga A en la parte derecha

$$B \to BAb$$

añadir una regla igual pero sin A

$$B \to BAb \,|\, Bb$$

Eliminación de ϵ-producciones

Repetir mientras quede alguna regla $A \to \epsilon$ que no sea $S_0 \to \epsilon$:

a. Quitar la regla $A \to \epsilon$
b. Añadir A al conjunto Quitadas
c. Para cada regla que tenga A en la parte derecha

$$B \to BAb$$

añadir una regla igual pero sin A

$$B \to BAb \,|\, Bb$$

d. Si A aparece varias veces, añadir también el resultado de quitarla una vez, dos veces, etc

$$B \to BAbAa$$

$$B \to BAbAa \,|\, BbAa \,|\, BAba \,|\, Bba$$

Eliminación de ϵ-producciones

Repetir mientras quede alguna regla $A \to \epsilon$ que no sea $S_0 \to \epsilon$:

a. Quitar la regla $A \to \epsilon$
b. Añadir A al conjunto Quitadas
c. Para cada regla que tenga A en la parte derecha

$$B \to BAb$$

añadir una regla igual pero sin A

$$B \to BAb \,|\, Bb$$

d. Si A aparece varias veces, añadir también el resultado de quitarla una vez, dos veces, etc

$$B \to BAbAa$$

$$B \to BAbAa \,|\, BbAa \,|\, BAba \,|\, Bba$$

e. En los pasos c. y d., no añadir nunca una regla $A \to \epsilon$ para $A \in$ Quitadas

Paso 2: eliminar ϵ-producciones

Quitar $B \to \epsilon$

$$S_0 \to S$$
$$S \to ASA|aB$$
$$A \to B|S$$
$$B \to b|\epsilon$$

$$S_0 \to S$$
$$S \to ASA|aB|\boldsymbol{a}$$
$$A \to B|S|\boldsymbol{\epsilon}$$
$$B \to b$$

Quitar $A \to \epsilon$

$$S_0 \to S$$
$$S \to ASA|aB|a$$
$$A \to B|S|\epsilon$$
$$B \to b$$

$$S_0 \to S$$
$$S \to ASA|aB|a|\boldsymbol{SA}|\boldsymbol{AS}|\boldsymbol{S}$$
$$A \to B|S$$
$$B \to b$$

Eliminación de producciones unarias

Repetir mientras quede alguna regla $A \to B$:

a. Quitar la regla $A \to B$
b. Añadir $A \to B$ al conjunto Quitadas

Eliminación de producciones unarias

Repetir mientras quede alguna regla $A \to B$:

a. Quitar la regla $A \to B$
b. Añadir $A \to B$ al conjunto Quitadas
c. Para cada regla $B \to u$ añadir $A \to u$

Eliminación de producciones unarias

Repetir mientras quede alguna regla $A \to B$:

a. Quitar la regla $A \to B$
b. Añadir $A \to B$ al conjunto Quitadas
c. Para cada regla $B \to u$ añadir $A \to u$
d. En el paso c, no añadir nunca una regla $A \to B$ de Quitadas

Paso 3: eliminar reglas unarias

Quitar $A \to B$ y $S \to S$

$$
\begin{aligned}
S_0 &\to S \\
S &\to ASA|aB|a|SA|AS|S \\
A &\to B|S \\
B &\to b
\end{aligned}
\qquad
\begin{aligned}
S_0 &\to S \\
S &\to ASA|aB|a|SA|AS \\
A &\to S|\mathbf{b} \\
B &\to b
\end{aligned}
$$

Quitar $A \to S$ y $S_0 \to S$

$$
\begin{aligned}
S_0 &\to S \\
S &\to ASA|aB|a|SA|AS \\
A &\to S|b \\
B &\to b
\end{aligned}
\qquad
\begin{aligned}
S_0 &\to \mathbf{ASA|aB|a|SA|AS} \\
S &\to ASA|aB|a|SA|AS \\
A &\to b|\mathbf{ASA|aB|a|SA|AS} \\
B &\to b
\end{aligned}
$$

Conversión final

Para cada regla $A \to \alpha_1 \dots \alpha_k$ con $\alpha_1 \dots \alpha_k$ terminales y variables:

a. Quitar la regla $A \to \alpha_1 \dots \alpha_k$

Conversión final

Para cada regla $A \to \alpha_1 \dots \alpha_k$ con $\alpha_1 \dots \alpha_k$ terminales y variables:

a. Quitar la regla $A \to \alpha_1 \dots \alpha_k$
b. Añadir nuevas variables $A_1, \dots, A_{k-2}$
c. Añadir las reglas

$$\begin{aligned} A &\to \alpha_1 A_1 \\ A_1 &\to \alpha_2 A_2 \\ &\dots \\ A_{k-2} &\to \alpha_{k-1} A_{k-1} \\ A_{k-1} &\to \alpha_{k-1} \alpha_k \end{aligned}$$

Conversión final

Para cada regla $A \to \alpha_1 \dots \alpha_k$ con $\alpha_1 \dots \alpha_k$ terminales y variables:

a. Quitar la regla $A \to \alpha_1 \dots \alpha_k$
b. Añadir nuevas variables $A_1, \dots, A_{k-2}$
c. Añadir las reglas

$$\begin{aligned} A &\to \alpha_1 A_1 \\ A_1 &\to \alpha_2 A_2 \\ &\dots \\ A_{k-2} &\to \alpha_{k-1} A_{k-1} \\ A_{k-1} &\to \alpha_{k-1} \alpha_k \end{aligned}$$

d. Si α_i es un terminal, añadir una nueva variable U_i y sustituir $A_{i-1} \to \alpha_i A_i$ por las reglas

$$A_{i-1} \to U_i A_i, \qquad U_i \to \alpha_i$$

Paso 4: convertir

$$
\begin{aligned}
S_0 &\to ASA|aB|a|SA|AS \\
S &\to ASA|aB|a|SA|AS \\
A &\to b|ASA|aB|a|SA|AS \\
B &\to b
\end{aligned}
$$

$$
\begin{aligned}
S_0 &\to \mathbf{AA_1|UB}|a|SA|AS \\
S &\to \mathbf{AA_1|UB}|a|SA|AS \\
A &\to b|\mathbf{AA_1|UB}|a|SA|AS \\
\mathbf{A_1} &\to \mathbf{SA} \\
\mathbf{U} &\to \mathbf{a} \\
B &\to b
\end{aligned}
$$

Gramática en forma normal de Chomsky

- La gramática en forma normal de Chomsky

$$\begin{aligned} S &\to ASA|aB \\ A &\to B|S \\ B &\to b|\epsilon \end{aligned}$$

$$\begin{aligned} S_0 &\to AA_1|UB|a|SA|AS \\ S &\to AA_1|UB|a|SA|AS \\ A &\to b|AA_1|UB|a|SA|AS \\ A_1 &\to SA \\ U &\to a \\ B &\to b \end{aligned}$$

Implementación de un parser

- ¿Cómo implementar un analizador sintáctico para una gramática G?
- ¿Cómo hallar si $w \in L(G)$?
- Notar que si G está en forma normal de Chomsky, si $S \Rightarrow^* w$ y tenemos la regla $S \to AB$, entonces $w = uv$, con $A \Rightarrow^* u$ y $B \Rightarrow^* v$.
- El problema $S \Rightarrow^* w$ se divide en dos subproblemas $A \Rightarrow^* u$ y $B \Rightarrow^* v$ con longitud $|u|, |v| \leq |w|$

Algoritmo CYK

- CYK (Cocke-Younger-Kasami) es uno de los algoritmos más eficientes en tiempo para averiguar si una cadena w es generada por una gramática G.
- Es un parser bottom-up: genera el árbol de derivación de abajo a arriba.
- Partimos de una gramática G en forma normal de Chomsky y $w = a_1 a_2 \cdots a_n$.
- El algoritmo halla si $w \in L(G)$. También se puede utilizar para construir el árbol de derivación.

Algoritmo CYK

- Partimos de $G = (N, \Sigma, R, S)$ y $w = a_1 \cdots a_n$
- Construimos para $1 \leq i \leq j \leq n$

$$S_{ij} = \{A \in N \mid A \Rightarrow^* a_i \cdots a_j\}$$

- Si $S \in S_{1n}$, entonces $w \in L(G)$
- Tenemos que rellenar la siguiente tabla

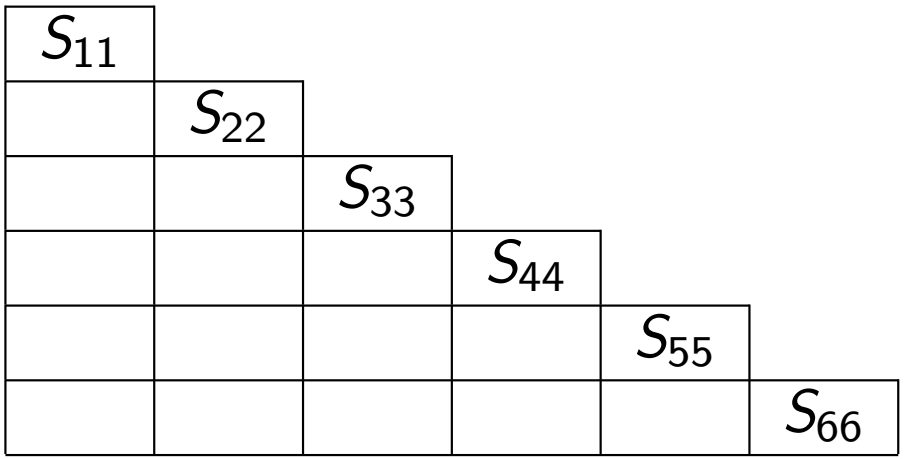
S_{11}						
S_{12}	S_{22}					
S_{13}	S_{23}	S_{33}				
			S_{44}			
S_{1n}						S_{nn}

Rellenar la tabla en CYK

- Primero se construye la diagonal principal de la tabla
- Si $i = j$: $S_{ii} = \{A \in N \mid$ existe la regla $A \to a_i \in R\}$

Rellenar la tabla en CYK

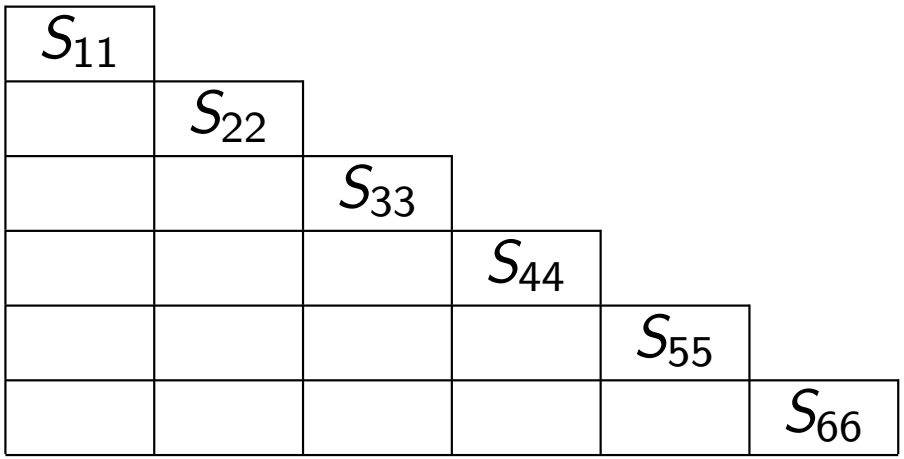
- Primero se construye la diagonal principal de la tabla
- Si $i = j$: $S_{ii} = \{A \in N \mid$ existe la regla $A \to a_i \in R\}$



Rellenar la tabla en CYK

- Primero se construye la diagonal principal de la tabla
- Si $i = j$: $S_{ii} = \{A \in N \mid$ existe la regla $A \to a_i \in R\}$
- Se construyen el resto de diagonales
- Si $j - i = d > 0$: $A \in S_{ij}$ si y solo si existe un regla $A \to BC$ y k tal que $i \leq k < j$

$$B \in S_{ik}, \ C \in S_{k+1j}$$



Rellenar la tabla en CYK

- Primero se construye la diagonal principal de la tabla
- Si $i = j$: $S_{ii} = \{A \in N \mid$ existe la regla $A \to a_i \in R\}$
- Se construyen el resto de diagonales
- Si $j - i = d > 0$: $A \in S_{ij}$ si y solo si existe un regla $A \to BC$ y k tal que $i \leq k < j$

$$B \in S_{ik}, \; C \in S_{k+1j}$$

$d = 1$

S_{11}					
S_{12}	S_{22}				
	S_{23}	S_{33}			
		S_{34}	S_{44}		
			S_{45}	S_{55}	
				S_{56}	S_{66}

Rellenar la tabla en CYK

- Primero se construye la diagonal principal de la tabla
- Si $i = j$: $S_{ii} = \{A \in N \mid$ existe la regla $A \to a_i \in R\}$
- Se construyen el resto de diagonales
- Si $j - i = d > 0$: $A \in S_{ij}$ si y solo si existe un regla $A \to BC$ y k tal que $i \leq k < j$

$$B \in S_{ik}, \ C \in S_{k+1j}$$

$d = 2$

S_{11}					
S_{12}	S_{22}				
S_{13}	S_{23}	S_{33}			
	S_{24}	S_{34}	S_{44}		
		S_{35}	S_{45}	S_{55}	
			S_{46}	S_{56}	S_{66}

Rellenar la tabla en CYK

- Primero se construye la diagonal principal de la tabla
- Si $i = j$: $S_{ii} = \{A \in N \mid$ existe la regla $A \to a_i \in R\}$
- Se construyen el resto de diagonales
- Si $j - i = d > 0$: $A \in S_{ij}$ si y solo si existe un regla $A \to BC$ y k tal que $i \leq k < j$

$$B \in S_{ik}, \; C \in S_{k+1j}$$

$d = 3$

S_{11}					
S_{12}	S_{22}				
S_{13}	S_{23}	S_{33}			
S_{14}	S_{24}	S_{34}	S_{44}		
	S_{25}	S_{35}	S_{45}	S_{55}	
		S_{36}	S_{46}	S_{56}	S_{66}

Rellenar la tabla en CYK

- Primero se construye la diagonal principal de la tabla
- Si $i = j$: $S_{ii} = \{A \in N \mid$ existe la regla $A \to a_i \in R\}$
- Se construyen el resto de diagonales
- Si $j - i = d > 0$: $A \in S_{ij}$ si y solo si existe un regla $A \to BC$ y k tal que $i \leq k < j$

$$B \in S_{ik}, \ C \in S_{k+1j}$$

$d = 4$

S_{11}					
S_{12}	S_{22}				
S_{13}	S_{23}	S_{33}			
S_{14}	S_{24}	S_{34}	S_{44}		
S_{15}	S_{25}	S_{35}	S_{45}	S_{55}	
	S_{26}	S_{36}	S_{46}	S_{56}	S_{66}

Rellenar la tabla en CYK

- Primero se construye la diagonal principal de la tabla
- Si $i = j$: $S_{ii} = \{A \in N \mid$ existe la regla $A \to a_i \in R\}$
- Se construyen el resto de diagonales
- Si $j - i = d > 0$: $A \in S_{ij}$ si y solo si existe un regla $A \to BC$ y k tal que $i \leq k < j$

$$B \in S_{ik}, \ C \in S_{k+1j}$$

$d = 5$

S_{11}					
S_{12}	S_{22}				
S_{13}	S_{23}	S_{33}			
S_{14}	S_{24}	S_{34}	S_{44}		
S_{15}	S_{25}	S_{35}	S_{45}	S_{55}	
S_{16}	S_{26}	S_{36}	S_{46}	S_{56}	S_{66}

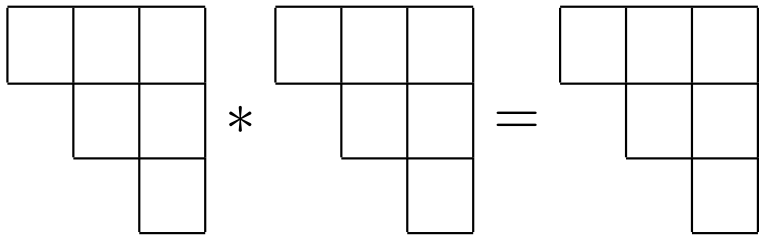
Algoritmo CYK

```
//construir diagonal principal de la tabla
for i=1 to n
    S_ii = {A | existe A → a_i}
//construir diagonales inferiores
for d=1 to n−1
  for i=1 to n−d
    S_{i,i+d} = ∅
    for k=i to i+d−1
        para cada regla A → BC: B ∈ S_ik, C ∈ S_{k+1 i+d}
            S_{i,i+d} = S_{i,i+d} ∪ {A}
```

- Observar que hay tres bucles anidados: complejidad $O(n^3)$

Curiosidad

- Suponer que tenemos que multiplicar dos matrices con ceros por debajo de la diagonal: matrices triangulares superiores
- El resultado es otra matriz triangular superior.



- El algoritmo para multiplicar este tipo de matrices sería

```
for d=0 to n−1
    for i=1 to n−d
        m[i][i + d] = 0
        for k=i to i+d
                m[i][i + d]+= m1[i][k] ∗ m2[k][i + d]
```

- Muy parecido al triple bucle del algoritmo CYK.

Curiosidad

- Se ha demostrado que se puede conseguir que CYK sea tan rápido como multiplicar matrices.
- ¿Cómo de rápido se pueden multiplicar matrices?
- Cualquiera de nosotros: $O(n^3)$
- Strassen(1969): $O(n^{2,807355})$
- Coppersmith-Winograd (1990): $O(n^{2,375477})$
- Stothers (2010): $O(n^{2,374})$
- Williams (2011): $O(n^{2,3728642})$
- Le Gall (2014): $O(n^{2,3728639})$
- Alman, William (2020): $O(n^{2,3728596})$
- Duan, Wu, Zhou (2020): $O(n^{2,371866})$
- Williams, Xu, Xu, Zhou (2024): $O(n^{2,371552})$
- Alman, Duan, Williams, Xu, Xu, and Zhou (2024): $O(n^{2,371339})$

Parser en la práctica

- El algortimo CYK consume demasiada memoria. No es útil para implementar parsers.
- Otra posibilidad: implementar el autómata de pila que reconoce el mismo lenguaje que la gramática.
- También inviable: ni es eficiente en tiempo, ni en memoria.
- Un autómata de pila es no determinista.
- Cada vez que el autómata tiene dos posibilidades: primero, se ha de investigar una, y si no se valida la entrada, se vuelve a la segunda (backtracking).

Otras clases de parsers

- Knuth en 1965 define un tipo parser de bottom-up llamado **LR**.
- Reconocen los lenguajes generados por gramáticas deterministas: autómatas de pila deterministas.
- La **L** de LR significa: se leen los símbolos de la entrada de izquierda a derecha
- La **R** significa: se genera una derivación por la derecha.
- No generan cualquier lenguaje independiente de contexto.
- Este tipo de parser puede mirar (sin leer) el siguiente elemento de la entrada para intentar adivinar qué hacer.
- Los parser LR(k) son parser LR que pueden mirar los siguientes k símbolos de la entrada.
- LR = LR(1)

Parser LALR

- Los parsers LR no utilizan backtracking. Complejidad $O(n)$ en el tamaño de la entrada.
- Siguen necesitando generar una tabla. Poco eficientes en memoria.
- En 1969 Frank DeRemer propone una versión simplificada de los parser LR.
- **LALR** (Look Ahead LR parser)
- Generan menos lenguajes que LR.
- Son lo más utilizados para definir lenguajes de programación.

Bison

- Bison (Robert Corbett 1988) es un programa generador de parsers que utilizaréis en prácticas.
- Flex es a las expresiones regulares como Bison a las gramáticas.
- Bison genera parsers LALR por defecto. También puede generar parsers GLR (generalized LR parser).
- Es compatible con Yacc (versión Unix): Richard Stallman
- Bison junto con un analizador lexicográfico se ha utilizado para generar compiladores para:
  - Ruby, PHP, GCC, Bash, PostgreSQL

Desplazamiento/reducción

- Todos los parsers que hemos comentado utilizan una técnica de análisis llamada **desplazamiento/reducción**.
- La entrada se analiza de izquierda a derecha. Se pueden realizar dos acciones.
- Desplazar: se introduce un símbolo de la entrada a la pila.
- Reducir: se aplica una regla de sustitución a los símbolos que hay en la pila. Se reduce la pila.
- Para desplazar o reducir se tienen en cuenta el siguiente símbolo de la entrada.
- Una cadena se acepta si la final en la pila solo tenemos el símbolo inicial.

Referencias

- Sipser: Capítulo 2, sección 2.1 (págs. 105-109)
- Sipser: Capítulo 7, teorema 7.16 (algoritmo CYK)