PARSING

Regex convertido en robot

Ing. Max Cerna

Agenda

1. Parsers
2. Gramáticas libres de contexto (CFG)
3. Derivaciones
4. Ambigüedad

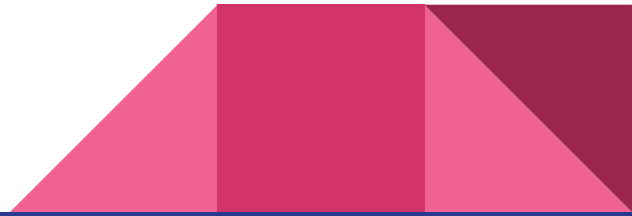
Parsers

PARSING

Lenguajes regulares

- Los lenguajes formales más débiles que se pueden utilizar
  - Son capaces de describir patrones simples como palabras clave, identificadores, y ciertas estructuras repetitivas en el código fuente

- Muchas aplicaciones
  - Validación de Patrones Simples
  - Filtros y Procesadores de Texto



PARSING

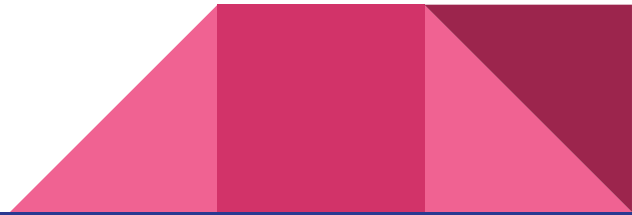
Muchos lenguajes no son regulares

Las cadenas de paréntesis balanceados no son regulares

$$\left(^{i}\right)^{i} \mid i \geq 0$$

¿Qué pueden expresar los lenguajes regulares?

- Expresan propiedades que dependen de contar hasta cierto punto
- Tienen una capacidad limitada de "memoria"
- Debido a esta limitación, no pueden manejar lenguajes que requieran un conteo exacto de elementos



PARSING

INPUT

Secuencia de tokens del lexer

OUTPUT

Árbol de parsing del programa

PARSING


Cadena de caracteres
LEXER
Cadena de tokens
PARSER
Árbol de parsing

PARSING

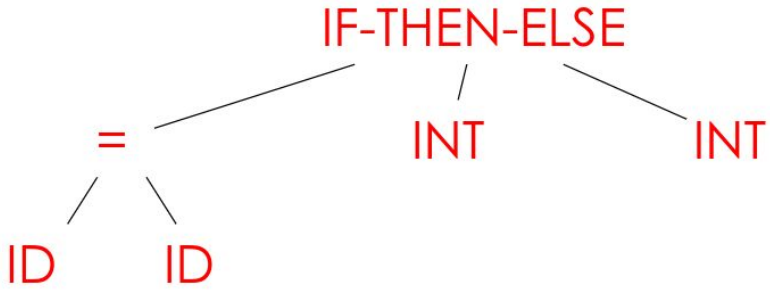
Cool (pseudo lenguaje)

```
if x = y then 1 else 2 fi
```

Entrada al parser (cadena de tokens)

```
IF ID = ID THEN INT ELSE INT FI
```

Salida del parser



Gramáticas libres de contexto (CFG)

CFG

- No todas las cadenas de tokens son programas. . .
- . . . el analizador debe distinguir entre cadenas de tokens válidas e inválidas
- Nosotros necesitamos
  - Un lenguaje para describir cadenas válidas de tokens
  - Un método para distinguir cadenas de tokens válidas de las no válidas

CFG

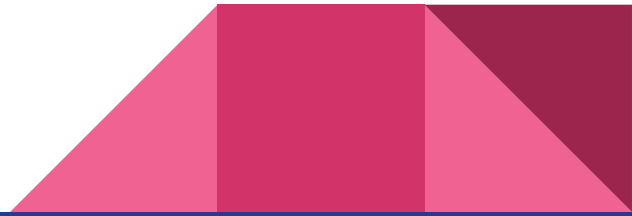
- Los lenguajes de programación tienen estructura recursiva
- Una EXPR es

  IF EXPR then EXPR else EXPR fi

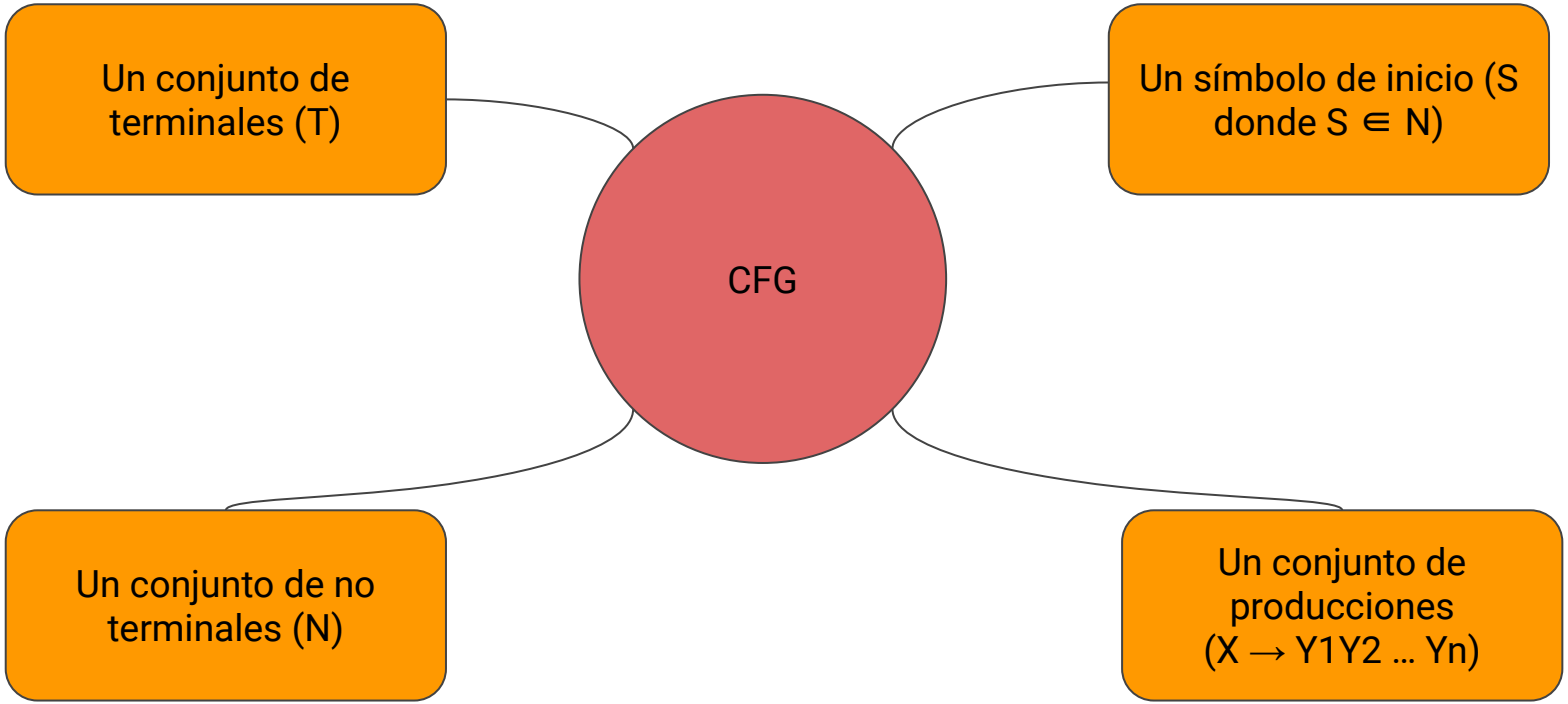
  while EXPR loop EXPR pool

  ...

- Las gramáticas libres de contexto son una notación natural para esta estructura recursiva



CFG consta de..

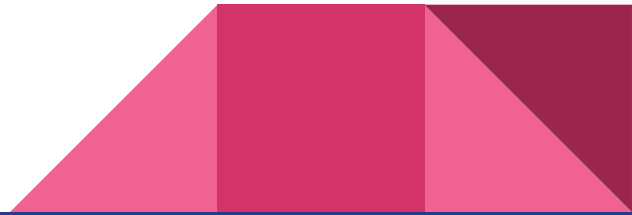


CFG

Las producciones se pueden leer como reglas

CFG - Notación

- Los no terminales se escriben en mayúsculas.
- Los terminales se escriben en minúsculas.
- El símbolo de inicio es el lado izquierdo de la primera producción.

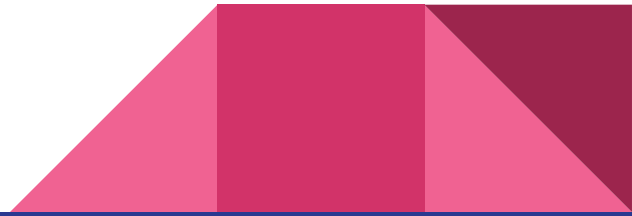


CFG - Algoritmo

1) Comience con una cadena con solo el símbolo de inicio S

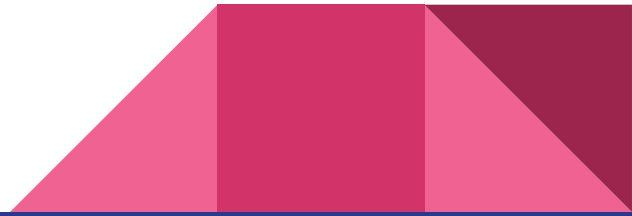
2) Reemplace cualquier X no terminal en la cadena por el lado derecho de alguna producción ej: X → Y1Yn

3) Repita (2) hasta que no haya no terminales



CFG

- Los terminales se llaman así porque no hay reglas para reemplazarlos
- Una vez generados, los terminales son permanentes
- Los terminales deben ser tokens del idioma.



CFG - EXPRESIONES ARITMÉTICAS SIMPLES

$$
\begin{aligned}
E &\rightarrow E * E \\
&\mid E + E \\
&\mid (E) \\
&\mid id
\end{aligned}
$$

CFG - Ejemplo

Cual de las siguientes cadenas están en el lenguaje dado por la CFG

- abcba
- acca
- aba
- abcbcba

$$
\begin{aligned}
S &\rightarrow aXa \\
X &\rightarrow \varepsilon \\
&\mid bY \\
Y &\rightarrow \varepsilon \\
&\mid cXc
\end{aligned}
$$

CFG

- Permite determinar si una cadena pertenece al lenguaje definido por la gramática.
- Además de verificar si una cadena pertenece al lenguaje, es esencial generar un árbol de análisis sintáctico (parse tree).
- Debe manejar los errores con gracia.
- Necesita una implementación de CFG (p. ej., Bison, CUP).
- La forma de la gramática es importante
  - Muchas gramáticas generan el mismo lenguaje
  - Las herramientas son sensibles a la gramática.

Derivaciones

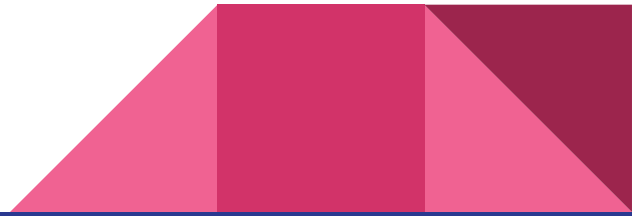
Derivaciones

Una derivación es una secuencia de producciones:

$S \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots$

Una derivación puede dibujarse como un árbol

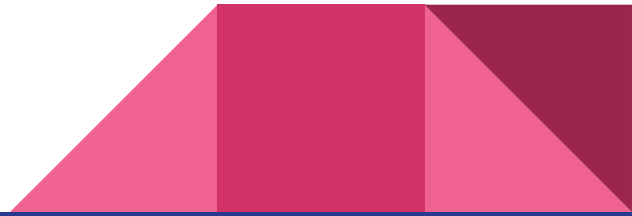
- El símbolo de inicio es la raíz del árbol
- Para una producción $X \rightarrow Y1 \ldots Yn$ agregar los hijos $Y1 \ldots Yn$ al nodo



Derivaciones

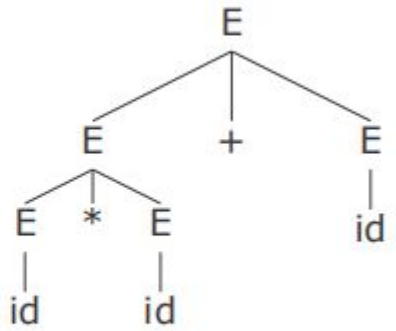
Gramática `E → E + E  |E * E  |(E)  |id`

Cadena `id * id + id`



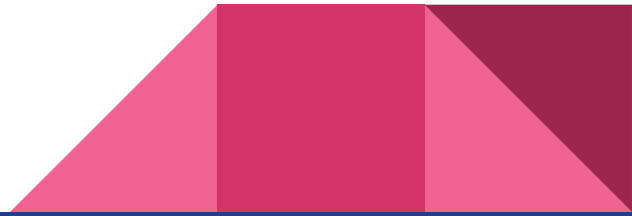
Derivaciones

$$
\begin{aligned}
& E \\
\longrightarrow\ & E + E \\
\longrightarrow\ & E * E + E \\
\longrightarrow\ & id * E + E \\
\longrightarrow\ & id * id + E \\
\longrightarrow\ & id * id + id
\end{aligned}
$$



Derivaciones

- Un árbol de análisis tiene
  - Terminales en las hojas
  - No terminales en los nodos interiores
- Un recorrido en orden de las hojas es la entrada original
- El árbol de análisis muestra la asociación de operaciones, la cadena de entrada no



Derivaciones

El ejemplo es una derivación por la izquierda (left-most derivation).

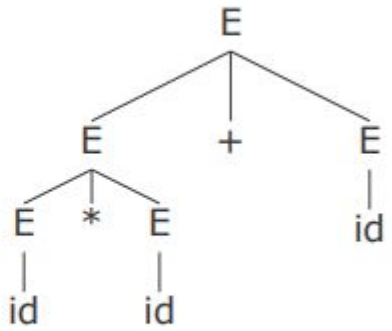
En cada paso, reemplaza el no terminal más a la izquierda.

Existe una noción equivalente de una derivación por la derecha (right-most derivation).

$$E$$
$$\longrightarrow E + E$$
$$\longrightarrow E + id$$
$$\longrightarrow E * E + id$$
$$\longrightarrow E * id + id$$
$$\longrightarrow id * id + id$$

Derivaciones

$$
\begin{aligned}
& E \\
\longrightarrow\ & E + E \\
\longrightarrow\ & E + id \\
\longrightarrow\ & E * E + id \\
\longrightarrow\ & E * id + id \\
\longrightarrow\ & id * id + id
\end{aligned}
$$



Derivaciones

Tenga en cuenta que las derivaciones por la la derecha y por la izquierda tienen el mismo árbol de análisis

Derivaciones

¿Cuál de las siguientes es una derivación válida de la gramática dada?

```
S → aXa

X → ε | bY

Y → ε | cXc | d
```

1)
S
aXa
abYa
acXca
acca

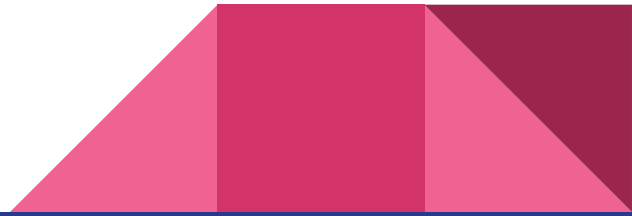
2)
S
aa

3)
S
aXa
abYa
abcXca
abcbYca
abcbdca

4)
S
aXa
abYa
abcXcda
abccda

Derivaciones

- No solo estamos interesados en si $s \in L(G)$. Necesitamos un árbol de análisis para s
- Una derivación define un árbol de análisis. Pero un árbol de análisis puede tener muchas derivaciones
- Las derivaciones más a la izquierda y más a la derecha son importantes en la implementación del analizador

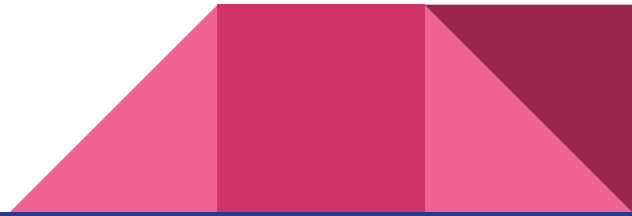


Ambigüedad

Ambigüedad

Gramática `E → E + E  |E * E  |(E)  |id`

Cadena `id * id + id`

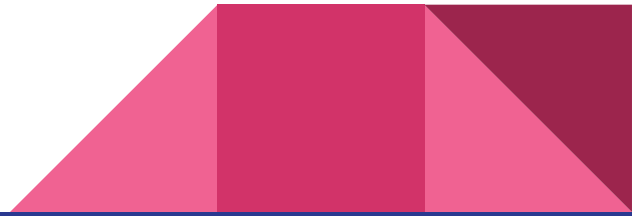


Ambigüedad

La cadena tiene dos árboles diferentes

Ambigüedad

- Una gramática es ambigua si tiene más de un árbol de análisis para alguna cadena.
  - De manera equivalente, existe más de una derivación más a la derecha o más a la izquierda para alguna cadena.
- La ambigüedad es MALA
  - Deja indefinido el significado de algunos programas.



Ambigüedad

¿Cuáles de las siguientes gramáticas son ambiguas?

- S → SS | a | b
- E → E + E | id
- S → Sa | Sb
- E → E |E + E
- E → −E | id

Ambigüedad

- Hay varias formas de manejar la ambigüedad
- El método más directo es reescribir la gramática inequívocamente, es decir sin ambigüedad

$E \rightarrow E + E | E$

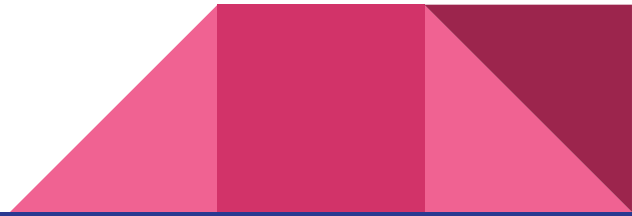
$E \rightarrow id * E | id | (E) * E | (E)$

Precedencia de $*$ sobre $+$

Ambigüedad

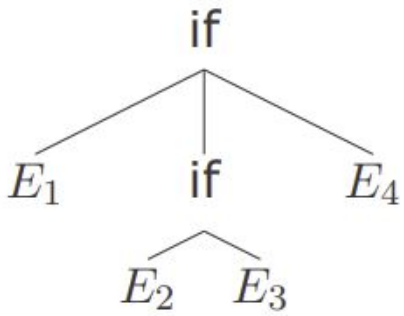
Considere la gramática

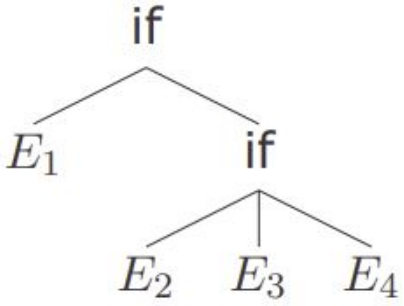
```
E →  if E then E
   | if E then E else E
   | OTHER
```



Ambigüedad

La expresión `if` E_1 `then if` E_2 `then` E_3 `else` E_4 tiene dos árboles



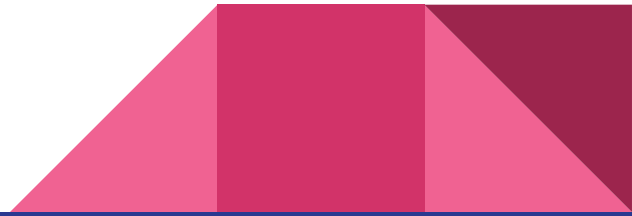


Ambigüedad

else coincide con el *then* aún sin coincidir más cercano:

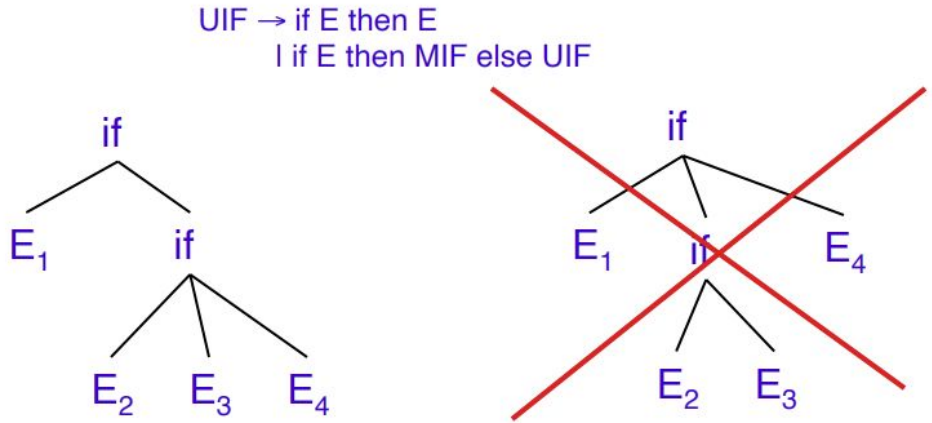
```
E →  MIF
   | UIF
MIF →    if E then MIF else MIF
       | OTHER

UIF →    if E then E
       | if E then MIF else UIF
```



Ambigüedad

Entonces expresión `if E1 then if E2 then E3 else E4`



Ambigüedad

- No hay técnicas generales para manejar la ambigüedad
- Es imposible convertir automáticamente una gramática ambigua en una no ambigua
- Usada con cuidado, la ambigüedad puede simplificar la gramática
    - A veces permite definiciones más naturales
    - Necesitamos mecanismos de desambiguación

Ambigüedad

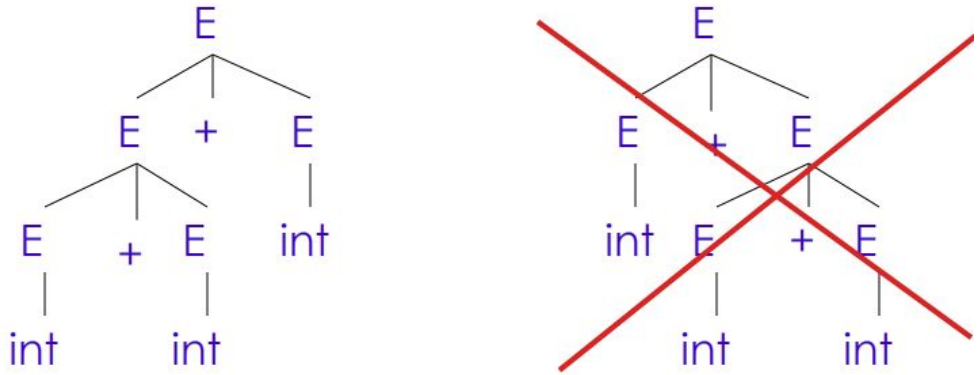
- En lugar de reescribir la gramática:
  - Utiliza la gramática más natural (ambigua)
  - Junto con declaraciones de desambiguación

- La mayoría de las herramientas permiten declaraciones de precedencia y asociatividad para desambiguar las gramáticas

Ambigüedad

Considere la gramática `E → E + E|int`

Dos árboles ambiguos para `int + int + int:`

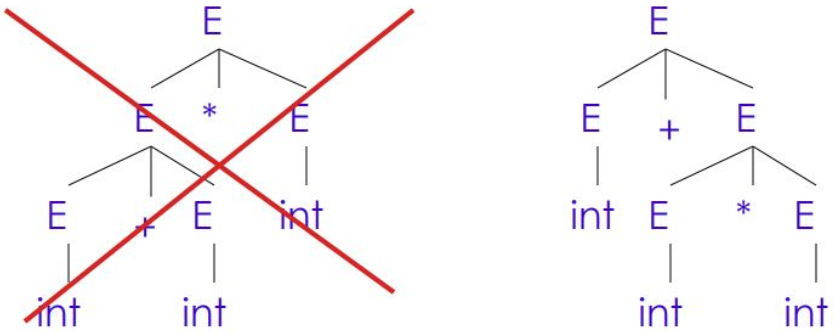


Asociación y declaración por la izquierda: `%left +`

Ambigüedad

Considere la gramática `E → E + E|E * E | int`

Dos árboles ambiguos para `int + int * int:`



Asociación de precedencia: `%left +`

`%left *`