Autómatas y Lenguajes Formales 2019-II
Facultad de Ciencias UNAM
Nota de Clase 11, Ambigüedad, gramáticas en lenguajes de programación

Favio E. Miranda Perea A. Liliana Reyes Cabello Lourdes González Huesca

17 de mayo de 2019

1. Ambigüedad

Puede ser que dos derivaciones distintas tengan un mismo árbol de derivación y también puede suceder que haya más de un árbol de derivación para una cadena. Lo ideal es que cada cadena tenga sólo un árbol asociado. Si sucede lo anterior, implica que el lenguaje **no es ambiguo**. Desafortunadamente existen lenguajes ambiguos. Veamos una definición formal de este fenómeno:

Definición 1 *Una gramática se dice* ***ambigua*** *si existe una palabra w con dos o más árboles de derivación distintos. En general una palabra puede tener más de una derivación, pero* un sólo árbol *y en tal caso no hay ambigüedad.*

Algunas veces se puede suprimir la ambigüedad directamente. Sin embargo **no** hay un algoritmo para remover ambigüedad. Es decir el problema de eliminar la ambigüedad es indecidible.

Ejemplo: Considere la siguiente gramática:

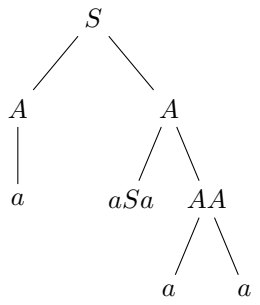
$$S \to AA \quad A \to aSa \mid a$$

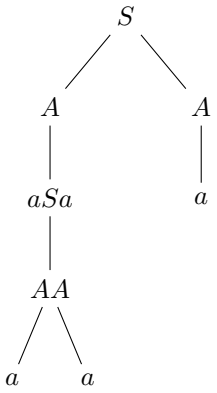
La palabra a^5 tiene las siguientes derivaciones diferentes y por la izquierda:

$$S \to AA \to aA \to aaSa \to aaAAa \to aaaAa \to aaaaa$$

$$S \to AA \to aSaA \to aAAaA \to aaAaA \to aaaaA \to aaaaa$$

Las dos derivaciones generan árboles distintos:





Definición 2 *Un lenguaje L es ambiguo si existe una gramática ambigua G que genera a L. Un lenguaje es* ***inherentemente*** *ambiguo si todas las gramáticas que lo generan son ambiguas.*

Ejemplo: El lenguaje $L = \{a^{2+3i} \mid i \geq 0\}$ es ambiguo por ser generado por la siguiente gramática ambigua:

$$S \to AA \qquad A \to aSa \mid a$$

Sin embargo este lenguaje también es generado por una gramática no ambigua:

$$S \to aa \mid aaU \qquad U \to aaaU \mid aaa$$

En este caso la derivación de a^5 es:

$$S \to aaU \to aaaaa$$

y por lo tanto L no es un lenguaje inherentemente ambiguo.

Ejemplo: El lenguaje

$$L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$$

es generado por la gramática:

$$\begin{array}{lll} S \to AB \mid C & A \to aAb \mid ab & B \to cBd \mid cd \\ C \to aCd \mid aDd & D \to bDc \mid bc & \end{array}$$

La cadena $aabbccdd$ tiene dos derivaciones por la izquierda:

$$S \to AB \to aAbB \to aabbB \to aabbcBd \to aabbccdd$$

$$S \to C \to aCd \to aaDdd \to aabDcdd \to aabbccdd$$

Como se comentó antes, la ambigüedad no puede ser eliminada con un método y en general, probar la ambigüedad inherente es complicado.

Veamos ahora algunos ejemplos de gramáticas ambigüas en relación a lenguajes de programación

Ejemplo 1 *Lenguaje de paréntesis balanceados:*

$$S \to \varepsilon \mid (S) \mid SS$$

- *Ambigüedad:* $()()()$ *tiene dos árboles de derivación*
- *Gramática no ambigua equivalente:*

$$S \to \varepsilon \mid (S)S$$

Ejemplo 2 *Expresiones aritméticas:*

$$S \to S + S \mid S * S \mid a$$

donde a *es un terminal que representa a los identificadores y constantes.*

- *Ambigüedad:* $a + a * a$
- *Gramática no ambigua equivalente: se obtiene modelando la precedencia de operadores como sigue:*

$$\begin{array}{lcl} S & \to & S + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (S) \mid a \end{array}$$

2. Gramáticas libres de contexto en lenguajes de programación

- El estudio formal de los lenguajes de programación se divide en sintaxis, pragmática y semántica.
- La semántica se encarga de definir el significado de las expresiones, enunciados y unidades de programa.
- La pragmática define la implementación del lenguaje basada en la metodología y estrategias de programación deseadas
- La sintaxis se encarga de definir la forma de las expresiones y enunciados de un lenguaje y se sirve fundamentalmente de los conceptos y herramientas de nuestro curso.
- Antes del proceso de evaluación, un compilador e intérprete necesita realizar los procesos de análisis léxico y sintáctico, describitos a continuación a grandes rasgos.

2.1. Análisis léxico y sintáctico

- Análisis léxico: se encarga de transformar el programa fuente en una lista de unidades sintácticas de bajo nivel llamadas lexemas, los cuales se clasifican en distintas categorías llamadas *tokens*, como pueden ser identificadores, constantes, separadores, etc.
- El análisis léxico se sirve fundamentalmente de expresiones regulares para su definición y reconocimiento.
- Análisis sintáctico: se encarga de transformar la lista de lexemas en un programa objeto, el cual es una expresión válida de la llamada sintaxis abstracta del lenguaje. Este programa es esencialmente un árbol de derivación dictado por una gramática libre de contexto que define al lenguaje de programación.
- Por lo tanto el análisis sintáctico es esencialmente una forma del problema de la pertenencia en gramáticas libres de contexto.

2.2. Forma de Backus-Naur

- Las gramáticas libres de contexto para lenguajes de programación suelen escribirse en la forma de Backus-Naur o BNF.
- Este método de definición de gramáticas fue introducido por John Backus para el lenguaje ALGOL 58 en 1959 y fue mejorado por Peter Naur para la definición de ALGOL 60.
- Este sistema notacional para definir lenguajes libres de contexto sigue las siguientes convenciones:
  - El símbolo de reescritura $\rightarrow$ se reemplaza con ::=.
  - El símbolo | significa *o* y abreviar la definición de producciones de una misma variable.
  - Las variables se escriben entre paréntesis triangulares y por lo general utilizan nombres largos que ayuden a la descripción de
    las categorías del lenguaje.

Ejemplo 3 *Algunos ejemplos de gramáticas en FBN son:*

- *Lenguaje de paréntesis balanceados*

$$\begin{array}{rcl} < parent\_balanc > & ::= & \varepsilon \mid \\ & & (< parent\_balanc >) \mid \\ & & < parent\_balanc >< parent\_balanc > \end{array}$$

- *Expresiones aritméticas:*

$$\begin{array}{rcl} < expr > & ::= & < expr >< op >< expr > \mid \\ & & (< expr >) \mid < id > \\ < op > & := & + \mid - \mid * \mid / \\ < id > & := & a \mid b \mid c \end{array}$$

- *Bloques de asignación de expresiones aritméticas:*

$$\begin{array}{rcl} < programa > & ::= & \texttt{begin} < sec\_enunc > \texttt{end} \\ < sec\_enunc > & ::= & < enunc > \mid < enunc > ; < sec\_enunc > \\ < enunc > & ::= & < id > := < expr > \\ < expr > & ::= & < expr >< op >< expr > \mid \\ & & (< expr >) \mid < id > \\ < op > & := & + \mid - \mid * \mid / \\ < id > & := & a \mid b \mid c \end{array}$$

Terminamos con un ejemplo importante en la historia de los lenguajes de programación:

- Expresiones condicionales:

$$\begin{array}{rcl} < enunc > & ::= & < condicional > \mid < otro > \\ < condicional > & := & \texttt{if } < expr > \texttt{ then } < enunc > \mid \\ & & \texttt{if } < expr > \texttt{ then } < enunc > \\ & & \texttt{ else } < enunc > \end{array}$$

- Problema del `else` colgante (*dangling else*):

```
if false then if false then 0 else 1
```

- Dos significados:

```
if false then (if false then 0) else 1
if false then (if false then 0 else 1)
```

- Una gramática equivalente no ambigua es:

$$\begin{array}{rcl} < enunc > & ::= & < condicional > \mid < otro > \\ < condicional > & := & < cond\_doble > \mid < cond\_final > \\ < cond\_doble > & := & \texttt{if } < expr > \texttt{ then } < cond\_doble > \texttt{ else } < cond\_doble > \\ < cond\_final > & := & \texttt{if } < expr > \texttt{ then } < enunc > \mid \\ & & \texttt{if } < expr > \texttt{ then } < cond\_simple > \texttt{ else } < cond\_final > \end{array}$$

- Idea: $C1$ genera condicionales dobles (if-then-else) balanceados; $C2$ representa condicionales simples (if-then) y condicionales dobles pero de forma que un if-then sólo figura colgando al final (en el else).