Análisis de Lenguajes de Programación

Ignacio Rimini

August 2025

Índice

1. Unidad 1 - Sintaxis.

1.1. Definición. Sintaxis.

La **sintaxis** describe la forma que van a tener los programas válidos del lenguaje. Permite distinguir entre secuencias de símbolos que pertenecen al lenguaje y las que no.

Existen dos tipos de sintaxis:

1. **Sintaxis concreta.**
   - Modela las **secuencias de caracteres** que son aceptadas como programas sintácticamente válidos.
   - Incluye información sobre la **representación textual:**
     - cómo se parentiza una expresión,
     - cómo se separan los elementos de una lista,
     - si los operadores son prefijos, infijos o postfijos,
     - reglas de precedencia y asociatividad.
   - Es la sintaxis que se especifica típicamente mediante **gramáticas formales** (por ejemplo, BNF o EBNF).
2. **Sintaxis abstracta.**
   - Modela la **estructura esencial** de los programas válidos.
   - Es independiente de la representación textual.
   - Se centra en los componentes importantes para la semántica del programa, dejando de lado detalles como paréntesis redundantes o la notación exacta de los operadores.
   - Suele representarse mediante **árboles de sintaxis abstracta (ASTs).**

   En resumen, la **sintaxis concreta** es para los humanos (escribir programas correctamente, con reglas claras de formato y notación) y la **sintaxis abstracta** es para la máquina/compilador (trabajar con la estructura esencial sin el *ruido* textual).

1.2. Ejemplos de sintaxis.

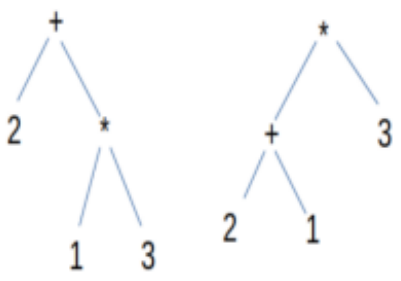
1.2.1. Ejemplo de sintaxis abstracta.

```
exp ::= n | exp + exp | exp * exp
```

donde $n \in \mathbb{N}$. Los siguientes son árboles sintácticos que corresponden a las expresiones:

$$2 + (1 * 3)$$

$$(2 + 1) * 3$$



1.2.2. Ejemplo de sintaxis concreta.

```
exp ::= n | exp '+' exp | exp '*' exp | '(' exp ')'
n ::= d | dn
d ::= '0' | '1' | ... | '9'
```

Y a continuación se describen algunas derivaciones:

- `exp` $\rightarrow$ `n` $\rightarrow$ `dn` $\rightarrow$ `dd` $\rightarrow$ `1d` $\rightarrow$ `12`
- `exp` $\rightarrow$ `exp '+' exp` $\rightarrow$ `exp '+' exp '*' exp` $\rightarrow$ `exp '+' exp '*' n` $\rightarrow \cdots \rightarrow$ `2 '+' 1 '*' 3`
- `exp` $\rightarrow$ `exp '*' exp` $\rightarrow$ `exp '+' exp '*' exp` $\rightarrow$ `exp '+' exp '*' n` $\rightarrow \cdots \rightarrow$ `2 '+' 1 '*' 3`

1.3. Gramáticas Libres de Contexto (CFG).

1.3.1. Introducción.

Una forma de definir la sintaxis de un lenguaje es mediante una **gramática libre de contexto (CFG).** La mayoría de lenguajes de programación definen su sintaxis mediante una CFG.

Un **lenguaje** resulta ser **libre de contexto** si hay una gramática libre de contexto que lo genera.

Las CFGs permiten capturar nociones esenciales de la sintaxis, como:

- Paréntesis balanceados.
- Palabras clave emparejadas (por ejemplo `begin` y `end`).

1.3.2. Definición. Grámatica Libre de Contexto.

Una **gramática libre de contexto (CFG)** puede ser definida por una 4-upla `(N, T, P, S)` donde:

- `N` es un conjunto finito de **no terminales** (categorías sintácticas o símbolos que no aparecen en el lenguaje: `exp, term, atom`, etc).
- `T` es un conjunto finito de **terminales** (símbolos básicos que si aparecen en el lenguaje: +, *, (,), números. identificadores, etc). Se cumple que `N` $\cap$ `T` $= \emptyset$.
- `P` es un conjunto de **producciones** de la forma $A \rightarrow \alpha$, donde $A \in N$ (es un no terminal) y $\alpha \in (N \cup T)^*$.

  Donde $*$ es el operador estrella de Kleene y el conjunto definido denota cualquier secuencia (incluyendo la secuencia vacía) de símbolos que sean terminales o no terminales.
- `S` es un **símbolo inicial** que pertenece a N.

1.3.3. Ejemplo de gramática libre de contexto.

Sea la gramática libre de contexto $G = (S, \{a, b\}, P, S)$ donde P tiene las siguientes reglas:

- $S \rightarrow aSb$
- $S \rightarrow \epsilon$

Esta gramática genera el lenguaje $\{a^n b^n : n \geq 0\}$.

Observaciones.

- Las producciones con el mismo lado izquierdo se pueden agrupar (notación de Backus-Naur o BNF): $S \rightarrow aSb \mid \epsilon$

- También se utiliza ::= en lugar de $\rightarrow$ en las producciones.
- El símbolo ϵ representa la cadena vacía.

1.3.4. Definición. Relación de derivación ($\Rightarrow$).

Sea $G = (N, T, P, S)$ una gramática libre de contexto, definimos la relación binaria:

$$\Rightarrow \subseteq (N \cup T)^* \times (N \cup T)^*$$

sobre secuencias de símbolos (terminales y no terminales), llamada **derivación** y donde $A \rightarrow B$ es una producción de G. Formalmente, $\Rightarrow$ es la menor relación tal que:

$$\alpha A \gamma \Rightarrow \alpha B \gamma$$

donde α y γ son secuencias de símbolos (terminales o no terminales), A es un no terminal que estamos reemplazando y B es un RHS (Right-Hand Side) de la producción.

Es decir, si tenemos una cadena donde aparece un no terminal A, podemos reemplazarlo por lo que dice la regla $A \rightarrow B$ manteniendo el contexto. Esta relación **derivación** es más abstracta que una regla de producción $\rightarrow$ de P, pues se puede definir para cualquier contexto.

1.3.5. Definición. Relación de derivación reflexiva-transitiva ($\Rightarrow^*$).

Dada la relación derivación $\Rightarrow$, se define la relación $\Rightarrow^*$ como la **clausura reflexiva-transitiva** de $\Rightarrow$. Es decir, es la menor relación sobre $(N \cup T)^*$ tal que:

- Si $\alpha \Rightarrow \beta$, entonces $\alpha \Rightarrow^* \beta$ (α deriva en muchos pasos a β).
- $\alpha \Rightarrow^* \alpha$
- Si $\alpha \Rightarrow^* \beta$ y $\beta \Rightarrow^* \gamma$, entonces $\alpha \Rightarrow^* \gamma$

1.3.6. Definición. Lenguaje generado por una gramática.

Dada una gramática G tenemos que un lenguaje L es generado por G si:

$$L(G) = \{w \mid w \in T^* \wedge S \Rightarrow^* w\}$$

Es decir, el lenguaje generado por G son todas las cadenas **formadas únicamente por terminales** que se pueden derivar desde el símbolo inicial S.

1.3.7. Ejemplo de derivación a partir de gramática.

Sea la gramática libre de contexto $G = (S, \{a, b\}, P, S)$ donde P tiene las siguientes reglas:

- $S \rightarrow aSb$
- $S \rightarrow \epsilon$

Tenemos que son válidas las siguientes derivaciones:

- $S \Rightarrow aSb$
- $S \Rightarrow \epsilon$
- $aSb \Rightarrow aaSbb$
- $aaS \Rightarrow aaaSb$ (aunque es imposible obtener el lado izquierdo con la gramática, la expresión vale)

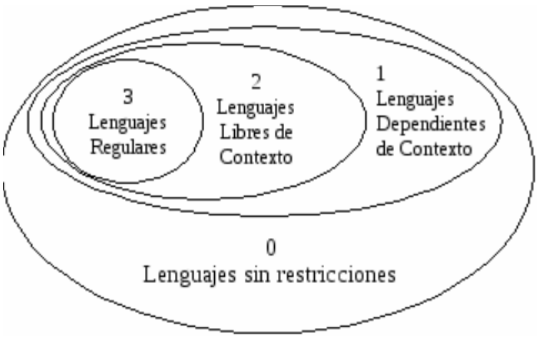
- $S \Rightarrow^* aSb$
- $S \Rightarrow^* \epsilon$
- $aSb \Rightarrow^* aaSbb$
- $aaSbb \Rightarrow^* aabb$
- $S \Rightarrow^* aaabbb$

Y luego, $L(G) = \{a^n b^n : n \geq 0\}$.

1.3.8. Propiedades de lenguajes libres de contexto.

- Los lenguajes libres de contexto pueden definirse también a través de autómatas no deterministas.
- La unión de dos lenguajes libres de contexto es también libre de contexto (la intersección no necesariamente).
- Determinar si dos CFGs generan el mismo lenguaje no es decidible.

A modo de observación, a continuación se inserta un gráfico sobre la **Jerarquía de Chomsky.**



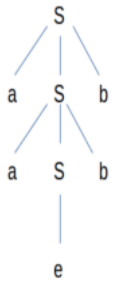
1.3.9. Definición. Árbol de parseo.

Un **árbol de parseo** es una derivación de una CFG $G = (N, T, P, S)$ si:

- Cada nodo del árbol tiene una etiqueta en $N \cup T \cup \{\epsilon\}$.
- La raíz tiene etiqueta S.
- Las etiquetas de los nodos interiores están en N.
- Si un nodo con etiqueta A tiene k hijos con etiquetas $X_1, ...X_k$, entonces $A \to X_1, ...X_k$ es una regla en P.
- Si un nodo tiene etiqueta ϵ, entonces es una hoja y es único hijo.

Luego, dada una gramática (N, T, P, S), decimos que $S \Rightarrow \alpha$ sii α es el **resultado** de un árbol de parseo, siendo éste la cadena que se forma al unir las etiquetas de las hojas del árbol (de izquierda a derecha).

Para la CFG del ejemplo, la cadena `aabb` tiene el siguiente árbol:



1.4. Gramáticas ambiguas.

1.4.1. Definición. Gramática ambigua.

Una gramática CFG G es **ambigua** si una palabra en `L(G)` tiene más de un árbol de parseo.

Observaciones.

- En general, los lenguajes CFG pueden ser generados por gramáticas ambiguas y no ambiguas. Pero existen algunos lenguajes que sólo pueden ser generados por CFG ambiguas. Estos son llamados **lenguajes inherentemente ambiguos.**
- Determinar si una CFG es ambigua no es decidible.

1.4.2. Desambiguar gramáticas.

Resolver la ambiguedad es importante, pues por ejemplo, dos árboles de parseo pueden tener distinta semántica.

Desambiguar fijando reglas de precedencia y asociatividad.

A continuación veremos un ejemplo de como desambiguar una gramática fijando reglas de precedencia y asociatividad entre los operadores. Tenemos la siguiente gramática (vista al inicio):

```
exp ::= n | exp '+' exp | exp '*' exp | '(' exp ')'
n ::= d | dn
d ::= '0' | '1' | ... | '9'
```

Podemos ver que hay dos derivaciones distintas para un mismo resultado. El resultado es $2 + 1 * 3$ y la primer derivación arranca con `exp '+' exp` y la segunda con `exp '*' exp`.

- `exp` $\to$ `exp '+' exp` $\to$ `exp '+' exp '*' exp` $\to$ `exp '+' exp '*' n` $\to \cdots \to$ `2 '+' 1 '*' 3`
- `exp` $\to$ `exp '*' exp` $\to$ `exp '+' exp '*' exp` $\to$ `exp '+' exp '*' n` $\to \cdots \to$ `2 '+' 1 '*' 3`

Para desambiguar la gramática, seguimos estos puntos que consisten en separar la gramática en niveles de precedencia. Así los no terminales sirven como *capas*: `exp` (suma y resta, nivel más bajo de precedencia), `term` (multiplicación y división, nivel intermedio), `atom` (paréntesis o números, nivel más alto).

- Definimos convenciones sobre la precedencia y asociatividad de los operadores. En este caso, reflejamos que el producto tiene más precedencia que la suma:

  ```
  exp ::= exp '+' exp | term
  term ::= term '*' term | atom
  atom ::= '(' exp ')' | n
  ```

Esto fuerza que siempre se reduzcan primero las multiplicaciones antes de llegar a las sumas: el producto tiene mayor precedencia que la suma.

- Reflejamos además que ambos operadores asocian a izquierda:

```
exp ::= exp '+' term | term
term ::= term '*' atom | atom
atom ::= '(' exp ')' | n
```

¿Por qué cambia? Antes: `exp ::= exp '+' exp` permitía que a la derecha hubiera otra `exp` completa, lo que da lugar a asociaciones a derecha.

Ahora: `exp ::= exp '+' term` fuerza a que lo que quede a la derecha sea un `term` (no una nueva suma completa), asegurando que las sumas se agrupan a izquierda.

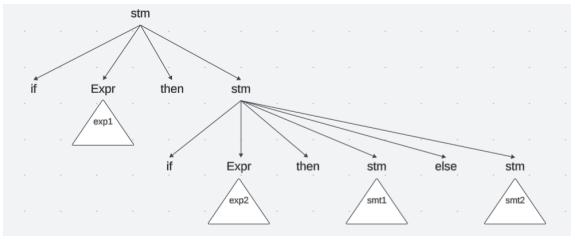
Problema de ambiguedad: *else colgado.*

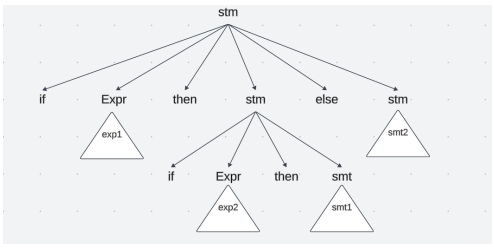
Supongamos una gramática con la siguiente regla de producción:

`stm ::= if exp then stm | if exp then stm else stm`

El siguiente programa tiene dos árboles de parseo:

`if` exp_1 `then if` exp_2 `then` stm_1 `else` stm_2





En el primer árbol, el `else` está asociado al `then` más cercano (el más a la derecha), mientras que en el segundo árbol está asociado al primer `then`. Es decir: primer árbol `if` exp_1 `then (if` exp_2 `then` stm_1 `else` stm_2`)` y segundo árbol `if` exp_1 `then (if` exp_2 `then` stm_1`) else` stm_2

Para transformar la gramática a una equivalente pero sin ambiguedades, agregamos la siguiente regla:

'Cada `else` está asociado al `then` más cercano que no está asociado a otro `else`'

Así, la sentencia que está entre `then` y `else` (matched) no puede contener un `if_then`, ya que esto violaría la regla.

```
stm -> matched | unMatched
matched -> if exp then matched else unMatched | otraSentencia
unMatched -> if exp then stm | if exp then matched else unMatched
```

Con esta reformulación todo `else` queda asociado de manera obligatoria al `then` más cercano. La derivación que asociaba un `else` a un `if` más externo deja de ser posible, eliminando la ambigüedad.

1.5. Árboles de sintaxis abstracta.

1.5.1. Introducción.

Hemos visto que al desambiguar una gramática concreta obtuvimos otra más díficil de interpretar. La **sintaxis abstracta** no tiene problemas de ambiguedad, porque se enfoca en la estructura jerárquica de los programas en lugar de su representación textual exacta: la sintaxis abstracta define árboles.

Un **Árbol de Sintaxis Abstracta (AST)** es una representación en forma de árbol de la estructura sintáctica de un lenguaje.

Las gramáticas abstractas son más simples que las concretas, porque eliminan detalles de notación (como paréntesis innecesarios o reglas de precedencia).

Ejemplo. La sintaxis de la siguiente gramática es concreta. Además, la gramática es ambigua:

```
exp ::= v | exp '+' exp | exp '-' exp | exp '*' exp | '(' exp ')'
v ::= x | y | z
```

Luego, el AST en notación BNF (Backus-Naur) de esta gramática es:

```
exp -> n | exp + exp | exp - exp | exp * exp
```

1.5.2. Implementación de AST.

Podemos implementar el AST de un lenguaje en Haskell con un tipo de datos algebraico. Por ejemplo:

```haskell
data Exp = Num Int | Sum Exp Exp | Prod Exp Exp | Minus Exp Exp
```

En la implementación notamos que los elementos de `Exp` son árboles y los nombres de los constructores son arbitrarios.

1.5.3. Lenguaje de booleanos y naturales.

Definimos el AST del lenguaje de booleanos y naturales como:

```
t -> true | false | if t then t else t | 0 | succ t | pred t | iszero t
```

Aquí `t` es una **metavariable**, que es una variable que pertenece al **metalenguaje** (lenguaje utilizado para analizar otro lenguaje) y es usada para representar un término del lenguaje objeto (el cual se describe).

1.5.4. Conjunto de términos.

Los AST definen el **conjunto de términos** del lenguaje. Una forma alternativa de definir el conjunto de términos de un lenguaje es mediante una definición inductiva.

Por ejemplo, definimos T como el menor conjunto tal que:

- $\{$`true, false, 0`$\} \subseteq T$
- Si $t \in T$ entonces $\{$`succ t, pred t, iszero t`$\} \subseteq T$
- Si $t, u, v \in T$ entonces `if t then u else v` $\in T$

Otra forma de definir el conjunto de términos (más concreta), es mediante un procedimiento que genera los elementos del conjunto. Por ejemplo, daremos una definición alternativa de T.

Primero damos una definición de S_i:

$$S_0 = \emptyset$$
$$S_{i+1} = \{\texttt{true, false, 0}\} \cup \{\texttt{succ t, pred t, iszero t} \mid \texttt{t} \in S_i\} \cup \{\texttt{if t then u else v} \mid \texttt{t, u, v} \in S_i\}$$

Y luego definimos:

$$S = \bigcup_{i \in N} S_i$$

Dado que T fue definido como el menor conjunto que satisface ciertas condiciones, para probar $T = S$, basta con probar:

1. S satisface las condiciones de T,
2. cualquier conjunto que satisfaga las condiciones contiene a S (S es el menor conjunto que satisface las condiciones)