Análisis de Lenguajes de Programación

Sintaxis

25 de Agosto de 2025

Sintaxis

- La sintaxis describe la forma que van a tener los programas válidos del lenguaje.

- Sintaxis concreta:
  - modela las **secuencias de caracteres** que son aceptadas como programas sintácticamente válidos
  - incluye información sobre la representación. Por ejemplo: cómo está parentizada una expresión, cómo se separan los elementos de una lista, si los operadores son prefijos o infijos, etc.

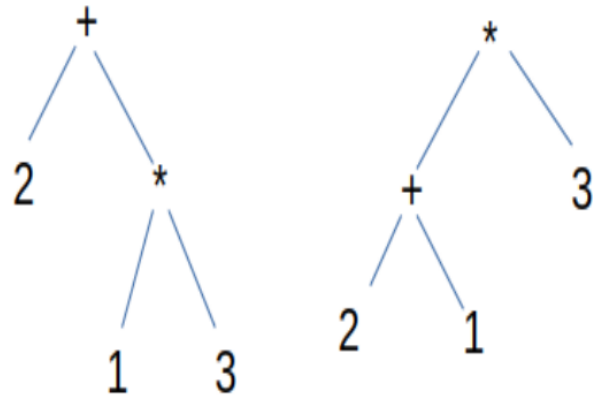
- Sintaxis abstracta:
  - modela la **estructural** esencial de los programas sintácticamente válidos
  - es independendiente de cualquier representación
  - son simples, ya que no se consideran detalles de la notación concreta

Sintaxis abstracta: Ejemplo

$exp \ ::= \ n \mid exp \ + \ exp \mid exp \ * \ exp$

donde $n \in \mathbb{N}$

Las siguientes son árboles sintácticos:



que corresponden a las expresiones $2 + (1 * 3)$ y $(2 + 1) * 3$

Sintaxis concreta: Ejemplo

$$exp ::= n \mid exp\ \text{'+'}\ exp \mid exp\ \text{'*'}\ exp \mid \text{'('}\ exp\ \text{')'}$$
$$n ::= d \mid dn$$
$$d ::= \text{'0'} \mid \text{'1'} \mid \ldots \mid \text{'9'}$$

Algunas derivaciones:

$$exp \rightarrow n \rightarrow dn \rightarrow dd \rightarrow 1d \rightarrow 12$$

$$exp \rightarrow exp\ \text{'+'}\ exp \rightarrow exp\ \text{'+'}\ exp\ \text{'*'}\ exp \rightarrow exp\ \text{'+'}\ exp\ \text{'*'}\ n \rightarrow \ldots \rightarrow 2\ \text{'+'}\ 1\ \text{'*'}\ 3$$

$$exp \rightarrow exp\ \text{'*'}\ exp \rightarrow exp\ \text{'+'}\ exp\ \text{'*'}\ exp \rightarrow exp\ \text{'+'}\ exp\ \text{'*'}\ n \rightarrow \ldots \rightarrow 2\ \text{'+'}\ 1\ \text{'*'}\ 3$$

Desambiguamos la gramática

1. Definimos convenciones sobre la precendencia y la asociatividad de los operadores.
2. Reflejamos las convenciones en la gramática.

- Reflejamos que el producto tiene más precedencia que la suma:

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

- Reflejamos además que ambos operadores asocian a izquierda:

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

Gramáticas Libres de contexto

- Una forma de definir la sintaxis de un lenguaje es mediante una gramática Libre de contexto (CFG).
- La mayoría de los lenguajes de programación definen su sintaxis mediante una CFG.
- Un **lenguaje es libre de contexto** si hay una CFG que lo genera.
- Capturan ciertas nociones que permiten desambiguar los lenguajes:
  - paréntesis balanceados
  - palabras claves emparejadas (como begin y end)

Gramáticas Libres de contexto

Una CFG puede ser definida por una 4-tupla (N, T, P, S), donde:

- N es un conjunto finito de **no terminales**.
- T es un conjunto finito de **terminales**, donde $N \cap T = \emptyset$.
- P es un conjunto de **producciones** de la forma $A \rightarrow \alpha$, donde $A \in N$ y $\alpha \in (N \cup T)^*$, siendo $*$ el operador estrella de Kleene.
- S es un **símbolo inicial** que pertenece a N.

Ejemplos

- $G = (S, \{a, b\}, P, S)$, donde P:

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

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

Relación de derivación

- Sea (N, T, P, S) una gramática definimos la relación binaria $\Rightarrow$ sobre $(N \cup T)^*$, como la menor relación tal que:

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

donde $A \rightarrow B$ es una producción de G.

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

  - Si $\alpha \Rightarrow \beta$, entonces $\alpha \Rightarrow^* \beta$
  - $\alpha \Rightarrow^* \alpha$
  - Si $\alpha \Rightarrow^* \beta$ y $\beta \Rightarrow^* \gamma$, entonces $\alpha \Rightarrow^* \gamma$

- Lenguaje generado por G:

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

Ejemplo

$S \Rightarrow aSb$ $\quad$ $S \Rightarrow \epsilon$

$aSb \Rightarrow aaSbb$ $\quad$ $aaS \Rightarrow aaaSb$

$S \Rightarrow^* aSb$ $\quad$ $S \Rightarrow^* \epsilon$

$aSb \Rightarrow^* aaSbb$ $\quad$ $aaSbb \Rightarrow^* aabb$

$S \Rightarrow^* aaabbb$

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

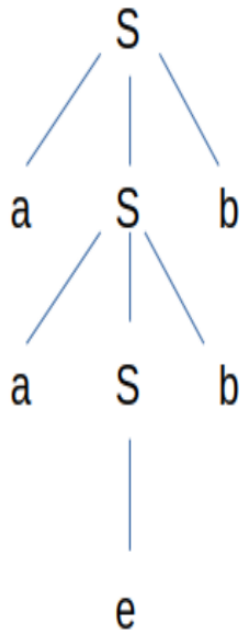
Propiedades

- Los lenguajes libre 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.

Árbol de parseo

Dada una gramática (N, T, P, S), $S \Rightarrow \alpha$ 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:



Árbol de parseo: definición

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

- cada nodo 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 $X_1, \ldots, X_k$, entonces $A \rightarrow X_1, \ldots, X_k$ en una regla en P.
- Si un nodo tiene etiqueta ϵ, entonces es una hoja y es único hijo.

Gramáticas ambiguas

- Una CFG G es **ambigua** si una palabra en $L(G)$ tiene más de un árbol de parseo.
- En general, los lenguajes CFG pueden ser generados por gramáticas ambiguas y no ambiguas. Pero existen algunos CFG que sólo pueden ser generados por CFG ambiguas, estos son llamado **lenguajes inherentemente ambiguos**.
- Determinar si una CFG es ambigua no es decidible.

Desambiguando gramáticas

- Resolver la ambiguedad es importante, por ejemplo dos árboles de parseos pueden tener distinta semántica.
- En el ejemplo introductorio vimos como desambiguar una gramática fijando reglas de precedencia y asociatividad entre los operadores.
- Veremos cómo solucionar de manera similar, agregando reglas y modificando la gramática, el problema de ambiguedad que se conoce como **"else colgado".**

Desambiguando gramática: else colgado

Supongamos una gramática con la siguiente producción:

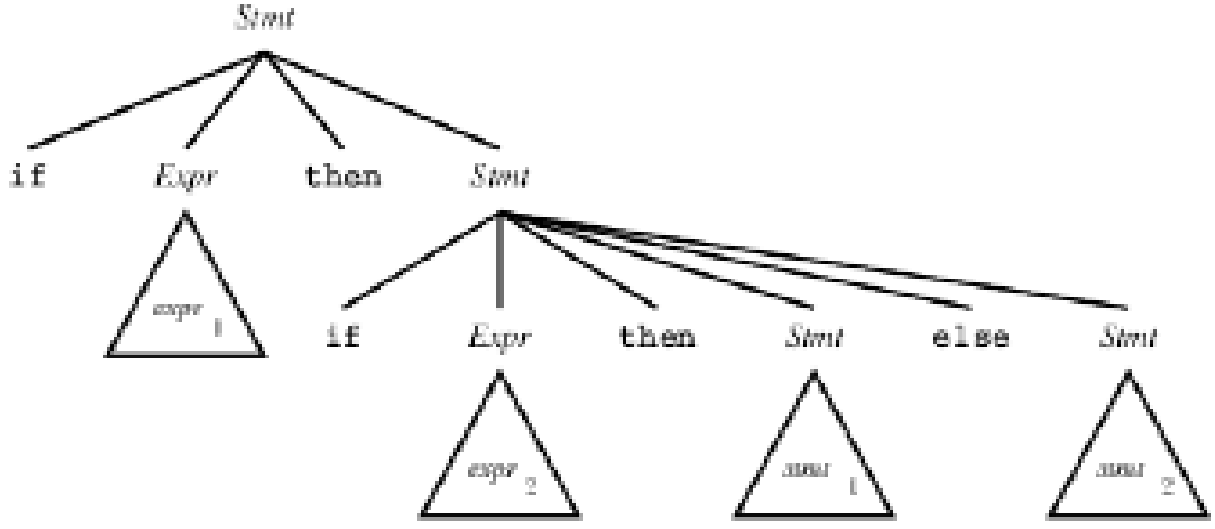
$$stm \quad \rightarrow \quad \textbf{if } exp \textbf{ then } stm \quad | \quad \textbf{if } exp \textbf{ then } stm \textbf{ else } stm$$

El siguiente programa tiene dos árboles de parseo:

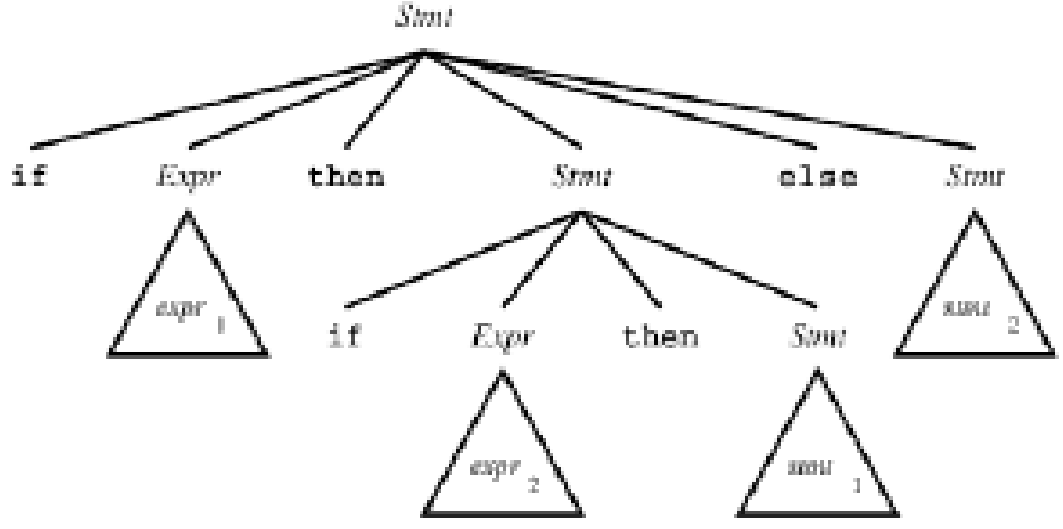
$$\textbf{if } exp_1 \textbf{ then if } exp_2 \textbf{ then } stm_1 \textbf{ else } stm_2$$

Desambiguando gramática: else colgado

Árbol 1:



Árbol 2:



Desambiguando gramática: else colgado

- En el primer árbol el **'else'** está asociado al **'then'** más cercano, mientras que en el segundo árbol está asociado al primer **'then'**.
- Agregamos la siguiente regla: "cada **'else'** está asociado al **'then'** más cercano que no está asociado a otro **'else'**"
- Transformamos la gramática a una equivalente pero **sin ambiguedades**.

Desambiguando gramática: else colgado

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

$$stm \rightarrow matched \mid unMatched$$

$$matched \rightarrow \textbf{if}\ exp\ \textbf{then}\ matched\ \textbf{else}\ matched \mid \ldots$$

$$unMatched \rightarrow \textbf{if}\ exp\ \textbf{then}\ stm \mid \textbf{if}\ exp\ \textbf{then}\ matched\ \textbf{else}\ unMatched$$

Ejercicio

Probar que la siguiete gramática es ambigua y desambiguarla.

$$exp ::= \; v \mid exp \text{ '+' } exp \mid exp \text{ '-' } exp \mid exp \text{ '*' } exp \mid \text{ '(' } exp \text{ ')'}$$
$$v ::= \; x \mid y \mid z$$

Sintaxis abstracta

- Vimos que al desambiguar una gramática concreta obtuvimos otra más difícil de interpretar.
- La sintaxis abstracta no tiene problemas de ambiguedad, ya que define árboles.
- Llamaremos árbol de sintaxis abstracta (AST), a la representación como árbol de la estructura sintáctica de un lenguaje.

  Por ejemplo, el AST en BNF del lenguaje del ejercicio es:

  $$exp \rightarrow n \mid exp + exp \mid exp - exp \mid exp * exp$$

- Las gramáticas son más simples.

Implementación de AST

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

```
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.

Lenguaje de booleanos y naturales

Definimos el AST del lenguaje como:

$$
\begin{array}{rcl}
t & \rightarrow & \textbf{true} \\
 & | & \textbf{false} \\
 & | & \textbf{if } t \textbf{ then } t \textbf{ else } t \\
 & | & 0 \\
 & | & \textbf{succ } t \\
 & | & \textbf{pred } t \\
 & | & \textbf{iszero } t
\end{array}
$$

- t es una **metavariable** (variable que pertenece al metalenguaje) y es usada para representar un término del lenguaje objeto (el cual se describe).
- usaremos también letras cercanas a t (u, v, t_1, t') en el lado derecho de las reglas.

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

Conjunto de términos

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:

$$\begin{array}{lcl} S_0 & = & \emptyset \\ S_{i+1} & = & \{\mathbf{true}, \mathbf{false}, \mathbf{0}\} \\ & & \cup \{\mathbf{suc}\ t,\ \mathbf{pred}\ t,\ \mathbf{iszero}\ t \mid t \in S_i\} \\ & & \cup \{\mathbf{if}\ t\ \mathbf{then}\ u\ \mathbf{else}\ v \mid t, u, v \in S_i\} \end{array}$$

- Luego definimos:

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

Prueba de que $T = S$

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)