# Análisis Semántico: Gramáticas con Atributos
# Compiladores 2020-1, Nota de clase 7

Lourdes Del Carmen González Huesca

18 de octubre de 2019
Facultad de Ciencias UNAM

El analizador sémático corresponde a la última fase del compilador en la parte de front-end. Este analizador verificará la corrección del programa más alla de la forma, es decir a nivel semántico, respecto a la definición del lenguaje de programación en particular.

## Introducción

Recordemos que un lenguaje de programación se define mediante:

1. Reglas sintácticas, mediante gramáticas libres de contexto.
2. Reglas semánticas, que definen características estáticas y dinámicas.

Estas definiciones siguen un estilo declarativo al sólo establecer reglas que definen las especificaciones del lenguaje. Una gramática libre de contexto no especifica cómo debe ser *parseado* un programa sino que define las reglas de construcción y esta parte sintáctica es el fundamento para los análisis léxico y sintáctico. De las reglas semánticas, las estáticas establecen propiedades de los programas que serán verificadas en tiempo de compilación para obtener código que preservará estas propiedades. Y las reglas dinámicas permitirán describir cómo se ejecuta un programa. Ambas reglas semánticas son de gran utilidad en esta fase de compilación ya que cada compilador tiene diferentes reglas semánticas particulares que deben asegurar que las reglas semánticas definidas del lenguaje se cumplan. Las reglas semánticas del compilador corresponden a un tipo especial de gramáticas, que también siguen un estilo declarativo ya que no especifican el orden en que deben ser aplicadas ni mucho menos indican el tipo de información que es sensible para estudiar la corrección del programa más alla de su forma.

Después del análisis sintáctico se debe considerar información importante que permita indagar si el programa tiene sentido, esto es al analizar en alto nivel su "significado". El significado real de un programa se estudia a través la **semántica denotacional** que permite asociar un objeto matemático a cada construcción sintáctica del lenguaje a través de funciones semánticas que describen el efecto de ejecutar dicha construcción. Este análisis no se estudiará en este curso.

La información importante son detalles de las partes de un programa que no pueden obtenerse del análisis sintáctico, por ejemplo verificar que los tipos de argumentos de una función coincidan con los esperados por su defición. Esta información ha sido parcialmente recuperada del programa por el lexer y el parser pero es necesario que sea incluida en el árbol sintáctico [1].

[1] En esta fase nos referimos al árbol sintáctico al resultado del análisis sintáctico ya sea como un árbol de sintaxis concreta o una simplificación del mismo mediante un árbol de sintaxis abstracta.

Decimos que el análisis semántico puede ser descrito en téminos de anotaciones o **decoraciones** en el árbol sintáctico. Las anotaciones son llamadas **atributos**. Los atributos y su relación con los tokens fueron obtenidos por el analizador léxico. Recordemos que un token es una palabra significativa que consta de dos partes, el nombre y su atributo: la primera es la representación del tipo de unidad léxica y la segunda es el valor de dicha unidad.

En el diseño de un compilador es importante definir cuáles atributos **no** son libres de contexto ya que éstos serán los valores que aparecen en el programa, que no están generados por una gramática libre de contexto y que por lo tanto no son resultado de la fase anterior del compilador es decir del analizador sintáctico. Estos atributos deben ser incluidos en el árbol sintáctico para poder complementar una representación intermedia que sea fiel al programa original.

Los resultados de la fase del análisis dependiente del contexto constan de

- anotar el árbol sintáctico con atributos, por ejemplo al usar apuntadores a identificadores en la tabla de símbolos;
- recorrer el árbol para generar una representación intermedia alternativa que describa el control de flujo del programa.

En esta parte estudiaremos cómo obtener la información sensible al contexto que aclare definiciones (por ejemplo, los tipos de argumentos, variables globales, etc.) además de algunos requerimientos para la ejecución. Por otra parte, si se desea estudiar más a fondo las propiedades de programas que sean dependientes de la ejecución se pueden abordar los métodos formales. Éstos son herramientas y formalismos que aseguran que se cumplen propiedades y requerimientos en tiempo de ejecución al relacionar el diseño y la implementación de especificaciones, por ejemplo las pruebas unitarias, la Lógica de Hoare, el análisis sintáctico, los sistemas de tipos, lenguajes con tipos dependientes, etc. De ellos, sólo nos ocuparemos de los sistemas de tipos para describir propiedades de seguridad de programas en tiempo de compilación. Estos sistemas los veremos en una nota posterior.

## Gramáticas con atributos

Una gramática con atributos es un complemento a una gramática libre de contexto al anotar las producciones con atributos. Estas gramáticas no especifican el significado del programa sólo ayudan a asociarlo con valores que explican su significado, es así que para cada símbolo de la gramática $A$, se le asocia una función que describe su valor $A.\texttt{val}$ o algún otro atributo:

- para los símbolos no-terminales, el valor es generado por la parte derecha de las producciones al dar significado o valor a la cadena de tokens derivada del símbolo
- los símbolos terminales que tengan atributo se le asocia el valor que depende del programa

El valor que se genere para cada regla de producción de una gramática libre de contexto clasifican a las reglas en:

**Reglas de copiado**

El atributo es copia de algún otro atributo, por ejemplo en las producciones unitarias de la forma $A \to B$:

$$A \to B \quad \triangleright \quad A.\texttt{val} := B.\texttt{val}$$

**Reglas con función semántica**

El atributo es calculado con funciones específicas dirigidas por el diseño del lenguaje y cuyos argumentos son atributos de la parte derecha de la producción:

$$A \to \alpha \quad \triangleright \quad A.\texttt{val} := \mathcal{F}(\alpha)$$

Obsérvese que no se puede hacer referencia a valores o atributos fuera de una producción y que cada símbolo de la gramática puede tener cero o más atributos. La forma en que se calculan los atributos permite clasificarlos en:

**Atributo sintético** obtiene su valor de un enunciado hacia la izquierda en una producción. Los símbolos terminales tienen propiedades intrínsecas y es por esto que son atributos sintéticos, obtienen su valor del programa original a través de la información recabada en la tabla de símbolos desde el lexer.

**Atributo heredado** obtiene su valor cuando el mismo símbolo no-terminal está a la derecha de la producción o al usar valores o atributos de otros símbolos. Es decir que la información contextual en el árbol sintáctico debe fluir de un símbolo en un nodo superior o en el mismo nivel. De esta forma, las reglas de producción de la gramática de atributos heredados pueden ser usadas muchas veces para obtener diferentes valores dependiendo del contexto y que se va heredando desde la información que está almacenada en la tabla de símbolos.

Una gramática con atributos está bien definida si las reglas determinan un conjunto único de valores para cada árbol sintáctico derivado de la gramática del lenguaje. Y es no-circular si nunca genera ciclos en el árbol sintáctico al calcular el flujo de los atributos.

Existen tres **esquemas de traducción** para aplicar las reglas de la gramática con atributos y obtener el valor de los atributos:

1. Esquema Inadvertido (*oblivious*): no toma en cuenta un orden de las reglas de producción e ignora el árbol sintáctico generado por el análisis anterior. Se escoge un orden conveniente para calcular los atributos y se repite este orden para calcular todos los atributos.

2. Esquema Dinámico: este método toma en cuenta la forma del parse tree al calcular una gráfica de dependencias. El orden para calcular los atributos de los nodos está determinado por el orden topológico en el árbol.

3. Esquema Estático: se realiza un análisis de las reglas semánticas en la construcción del compilador y se establece un orden en cada regla de producción para obtener los atributos en ella. Así se calculan todos los atributos dependiendo de la regla.

Una gráfica de dependencias en este caso establece el flujo de la información entre las instancias de los atributos en el parse tree. Las aristas son las restricciones de las reglas semánticas y los nodos son los diferentes atributos asociados a cada símbolo. Esta gráfica es particular e independiente a cada parse tree.

### Tipos de gramáticas con atributos

Decimos que una gramática es **S-atribuida** si todos sus atributos son sintéticos y los argumentos de las funciones semánticas usan únicamente símbolos en la parte derecha de la producción. El resultado de un atributo es la parte izquierda de la producción. Estas gramáticas están asociadas a los parsers LR, es decir que el cálculo de los atributos se realiza desde las hojas y hacia la raíz que es la misma forma en que se obtiene el parse tree. Este tipo de gramática favorece que los atributos puedan ser calculados al vuelo en la fase de análisis sintáctico.

Por otro lado, decimos que una gramática es **L-atribuida** si los atributos de los nodos son evaluados al visitar los nodos del parse tree de una sola vez de izquierda a derecha (**L**eft-to-right) y en un recorrido a profundidad:

- cada atributo sintético a la izquierda depende de los heredados o de la parte derecha
- cada atributo heredado a la derecha depende de los atributos heredados de la izquierda o de los atributos de la parte izquierda

Estas gramáticas están asociadas a los parsers LL ya que los atributos pueden ser calculados desde la raíz y hacia las hojas.

Toda gramática S-atribuida es L-atribuida pero no al revés. El tipo gramática S-atribuida es la más general y es la que en la práctica se implementa junto con un parser LR. Si el parser es LL entonces se implementará una L-atribuida.

Los atributos decoran el árbol sintáctico y se almacenan en los nodos; esto puede realizarse de dos formas:

1. Después del análisis sintáctico y como resultado del análisis semántico; es claro que las fases del compilador están separadas y se implementa una gramática con atributos en especial junto con un esquema de traducción apropiado.
2. Al mismo tiempo que se realiza el análisis sintáctico; es decir que la gramática tenga intercaladas funciones que permitan decorar el parse tree y generar una representación intermedia al mismo tiempo, entonces el tipo de compilador es uno *one-pass*.

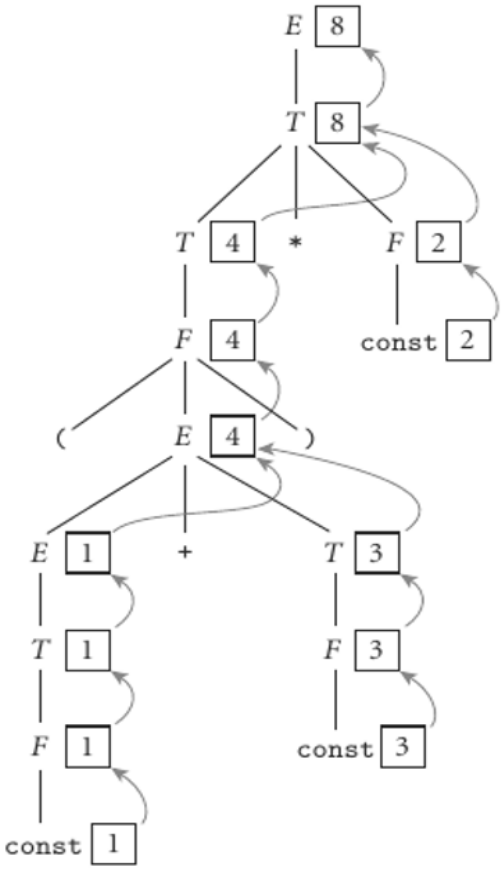
### Evaluación de atributos

Calcular los atributos se realiza ya sea desde las hojas o desde la raíz dependiendo del tipo de gramática S-atribuida o L-atribuida. A continuación veremos dos ejemplos, además en el archivo *AttributeGrammars-ParseTrees.pdf* se encuentran los ejemplos correspondientes a las mismas gramáticas pero cuyas funciones semánticas construyen árboles de sintaxis abstracta.

**Ejemplo 1** (Gramática S-atribuida)**.** La siguente gramática de expresiones aritméticas calcula los atributos desde las hojas

| | | |
|---|---|---|
| 1. | $E_1 \longrightarrow E_2$ + $T$ | ✂ $E_1$.val := sum($E_2$.val, T.val) |
| 2. | $E_1 \longrightarrow E_2$ - $T$ | ✂ $E_1$.val := difference($E_2$.val, T.val) |
| 3. | $E \longrightarrow T$ | ✂ E.val := T.val |
| 4. | $T_1 \longrightarrow T_2$ * $F$ | ✂ $T_1$.val := product($T_2$.val, F.val) |
| 5. | $T_1 \longrightarrow T_2$ / $F$ | ✂ $T_1$.val := quotient($T_2$.val, F.val) |
| 6. | $T \longrightarrow F$ | ✂ T.val := F.val |
| 7. | $F_1 \longrightarrow$ - $F_2$ | ✂ $F_1$.val := additive_inverse($F_2$.val) |
| 8. | $F \longrightarrow$ ( $E$ ) | ✂ F.val := E.val |
| 9. | $F \longrightarrow$ const | ✂ F.val := const.val |

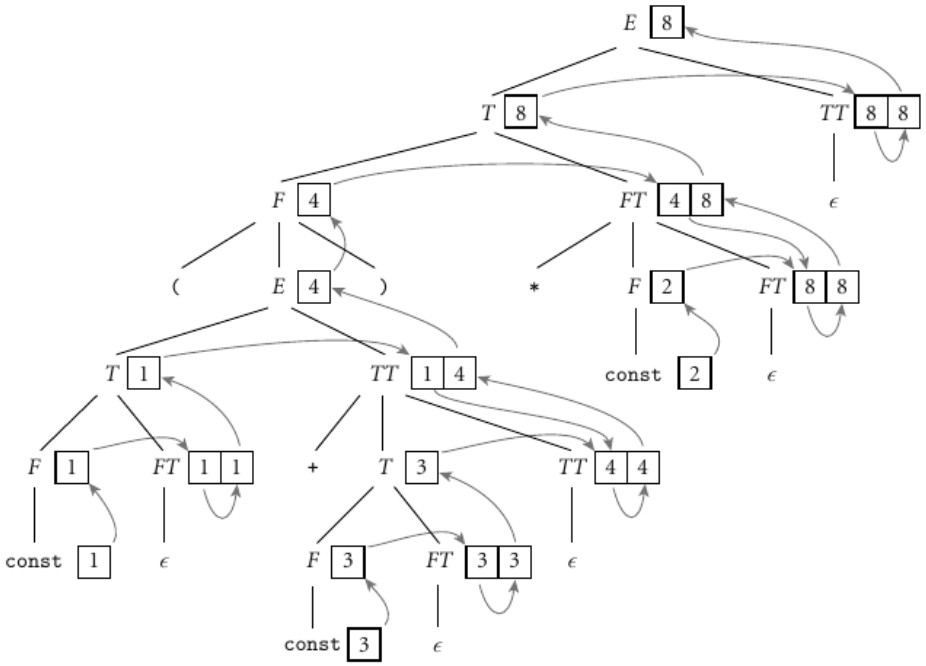
El siguiente árbol corresponde a la expresión $(1 + 3) * 2$, las flechas curvas indican la forma en que se obtuvieron los atributos.



**Ejemplo 2** (Gramática L-atribuida)**.** La siguente gramática de expresiones aritméticas calcula los atributos desde la raíz siguiendo una estrategia a profundidad y de izquierda a derecha

1. $E \longrightarrow T\ TT$
   ✂ TT.st := T.val ✂ E.val := TT.val
2. $TT_1 \longrightarrow$ `+` $T\ TT_2$
   ✂ $TT_2$.st := $TT_1$.st + T.val ✂ $TT_1$.val := $TT_2$.val
3. $TT_1 \longrightarrow$ `-` $T\ TT_2$
   ✂ $TT_2$.st := $TT_1$.st − T.val ✂ $TT_1$.val := $TT_2$.val
4. $TT \longrightarrow \epsilon$
   ✂ TT.val := TT.st
5. $T \longrightarrow F\ FT$
   ✂ FT.st := F.val ✂ T.val := FT.val
6. $FT_1 \longrightarrow$ `*` $F\ FT_2$
   ✂ $FT_2$.st := $FT_1$.st × F.val ✂ $FT_1$.val := $FT_2$.val
7. $FT_1 \longrightarrow$ `/` $F\ FT_2$
   ✂ $FT_2$.st := $FT_1$.st ÷ F.val ✂ $FT_1$.val := $FT_2$.val
8. $FT \longrightarrow \epsilon$
   ✂ FT.val := FT.st
9. $F_1 \longrightarrow$ `-` $F_2$
   ✂ $F_1$.val := − $F_2$.val
10. $F \longrightarrow$ `(` $E$ `)`
    ✂ F.val := E.val
11. $F \longrightarrow$ `const`
    ✂ F.val := const.val

El siguiente árbol corresponde a la expresión $(1 + 3) * 2$, las flechas curvas indican la forma en que se obtuvieron los atributos.




## Referencias


[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Pearson Education Inc., Second edition, 2007.

[2] Jean-Christophe Filliâtre. Curso Compilation (inf564) école Polytechnique, Palaiseau, Francia. `http://www.enseignement.polytechnique.fr/informatique/INF564/`, 2018. Material en francés.

[3] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag, Berlin, Heidelberg, 2007.

[4] Frank Pfenning. Notas del curso (15-411) Compiler Design, 2014. `https://www.cs.cmu.edu/~fp/courses/15411-f14/`.

[5] Michael Lee Scott. *Programming Language Pragmatics*. Morgan-Kauffman Publishers, Third edition, 2009.

[6] Yunlin Su and Song Y. Yan. *Principles of Compilers, A New Approach to Compilers Including the Algebraic Method*. Springer-Verlag, Berlin Heidelberg, 2011.

[7] Steve Zdancewic. Notas del curso (CIS 341) - Compilers, Universidad de Pennsylvania, Estados Unidos, 2018. `https://www.cis.upenn.edu/~cis341/current/`.