Análisis semántico y traducción dirigida por la sintaxis

Dpto. de Ingeniería de la Información y las Comunicaciones

Facultad de Informática
UNIVERSIDAD DE MURCIA

Índice: secciones principales

Definiciones Dirigidas por la Sintaxis (DDS)

Una **definición dirigida por la sintaxis** especifica la traducción de una construcción en función de atributos asociados con sus componentes sintácticos.

- A cada símbolo de la gramática le asocia un conjunto de **atributos**.
- A cada producción le asocia un conjunto de **reglas semánticas** que determinan los valores de los atributos asociados con los símbolos que aparecen en esa producción.

La gramática y el conjunto de reglas semánticas constituyen la **Definición Dirigida por la Sintaxis o DDS** .

Traducción de notación infija a postfija

Producción	Regla semántica
$E \rightarrow E_1 + T$	$E.codigo = E_1.codigo \mid\mid T.codigo \mid\mid \text{'+'}$

Definiciones Dirigidas por la Sintaxis (DDS)

A veces es conveniente permitir que las definiciones dirigidas por la sintaxis tengan **efectos adicionales** limitados, como imprimir un resultado o interactuar con la tabla de símbolos.

Se denomina **gramática atribuida** a una gramática *sin efectos adicionales* aumentada con los atributos y las acciones semánticas.

Si X es un símbolo y a uno de sus atributos, entonces escribimos $X.a$ para denotar el valor de a en el nodo específico de un árbol de análisis sintáctico, etiquetado como X.

Se llama **árbol de análisis sintáctico con anotaciones** a un árbol de análisis sintáctico que muestra los valores de los atributos en cada nodo.

Atributos sintetizados y heredados

Un **atributo** puede representar cualquier cantidad o valor (un tipo, una cadena, una posición de memoria, etc.). Son de dos tipos:

- Sintetizados.
- Heredados.

Atributos sintetizados

Un **atributo sintetizado** para un no terminal A en un nodo N de un árbol sintáctico, se define sólo en términos de los valores de los atributos en los hijos de N y en el mismo N, mediante una regla semántica asociada con la producción en N. La producción debe tener a A en su parte izquierda.

→ Tienen la propiedad de que pueden calcularse durante un sólo recorrido ascendente del árbol de análisis sintáctico.

◂ EjemploAtrSint

Atributos sintetizados y heredados

Atributos heredados

Un **atributo heredado** para un no terminal B en el nodo N de un árbol de análisis sintáctico se define sólo en términos de los valores de los atributos en el padre de N, en el mismo N y en sus hermanos, mediante una regla semántica asociada con la producción en el padre de N. La producción debe tener a B en su parte derecha.

- → Pueden ser útiles cuando, por ejemplo, la gramática está más orientada al análisis sintáctico que a la traducción (ver ejemplo de gramáticas L-atribuidas).
- → Siempre es posible escribir una definición dirigida por la sintaxis para que sólo se utilicen atributos sintetizados, pero a veces es más natural utilizar atributos heredados.

Atributos sintetizados y heredados

Ejemplo de DDS para declaración de tipos

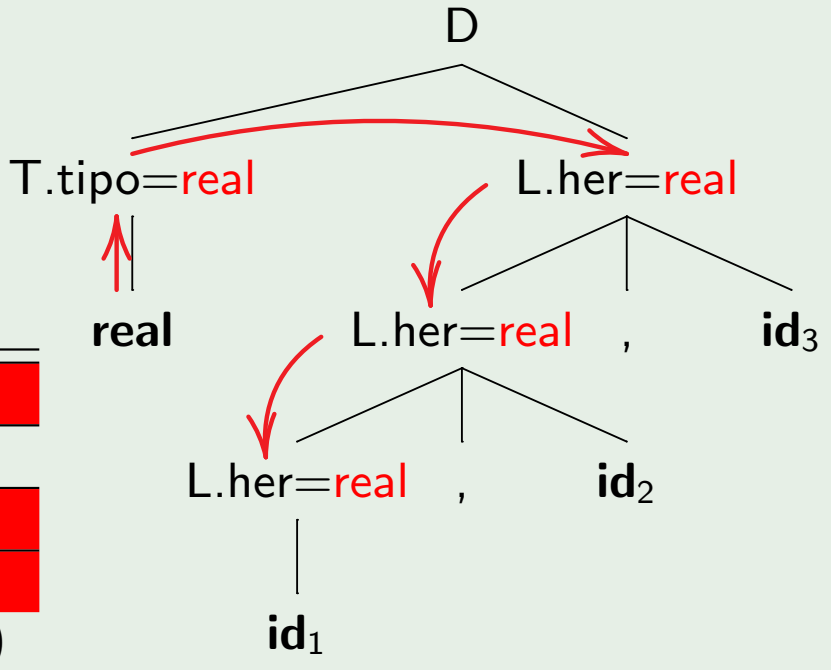
Las siguientes producciones generan las declaraciones de tipos en un lenguaje de programación:

$$
\begin{array}{lcl}
D & \rightarrow & T \quad L \\
T & \rightarrow & \textbf{int} \\
 & | & \textbf{real} \\
L & \rightarrow & L_1 \quad , \quad \textbf{id} \\
 & | & \textbf{id}
\end{array}
$$

Definición dirigida por la sintaxis:

Producción	Reglas semánticas
D → T L	L.her = T.tipo
T → **int**	T.tipo = integer
T → **real**	T.tipo = real
L → L_1 **, id**	L_1.her = L.her
	agregarTipo(**id**.entrada,L.her)
L → **id**	agregarTipo(**id**.entrada,L.her)

Árbol de AS con anotaciones
para la cadena: `real` $\mathtt{id}_1$`,` $\mathtt{id}_2$`,` $\mathtt{id}_3$



Evaluación de atributos

Una definición dirigida por la sintaxis no impone ningún orden específico a la evaluación de atributos en un árbol de análisis sintáctico.
La única **restricción** es que al calcular un atributo *a*, previamente han debido calcularse todos los atributos de los que depende *a*:

- → *A veces es necesario evaluar atributos del nodo antes que los de los hijos.*
- → *Otras veces después.*
- → *Y es posible que haya que hacerlo entre la evaluación de los atributos de unos hijos y otros.*

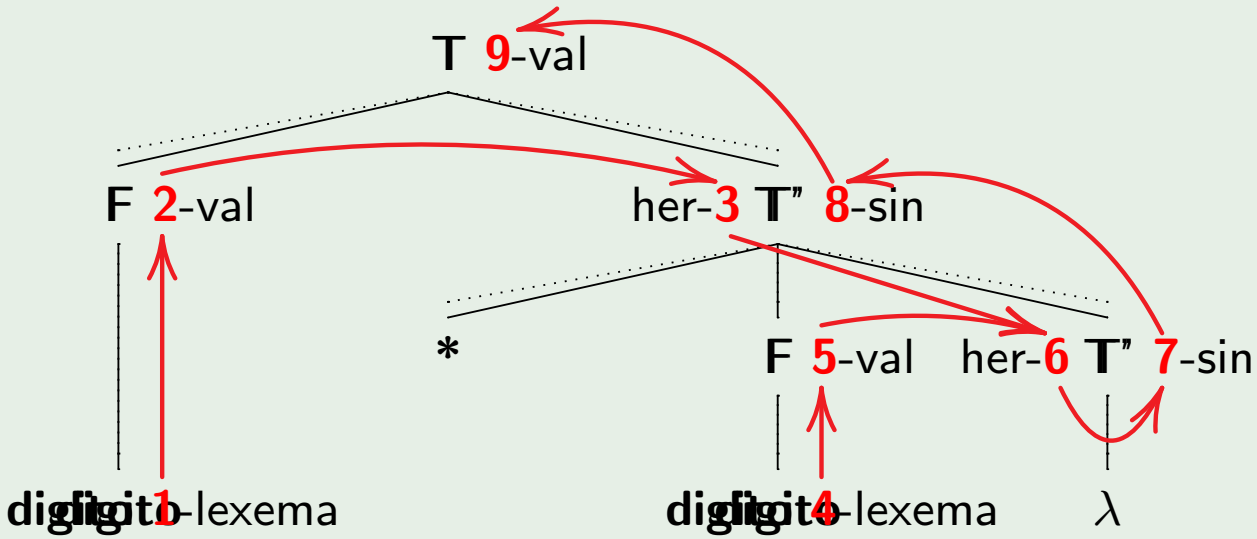
Grafo de dependencias

Se denomina **grafo de dependencias** en un árbol de análisis a un conjunto de nodos y aristas, en el que los nodos representan las ocurrencias de los atributos y las aristas las dependencias entre ellos.

Evaluación de atributos

Ejemplo de grafo de dependencias

Producción	Reglas semánticas
T → F T'	T'.her = F.val T.val = T'.sin
T' → * F T'$_1$	T'$_1$.her = T'.her x F.val T'.sin = T'$_1$.sin
T' → λ	T'.sin = T'.her
F → **digito**	F.val = **digito.lexema**



◂ EjemploLAtrib

Evaluación de atributos

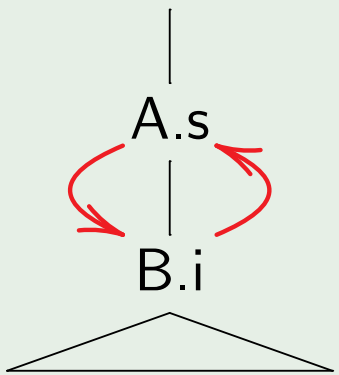
Gramática atribuida no circular

Se dice que una **gramática atribuida** es **no circular** cuando el grafo de dependencias no contiene ningún ciclo.

Entonces es posible encontrar un orden topológico para evaluar los atributos. En otro caso no.

Gramática circular

Producción	Reglas Semánticas
$A \to B$	$A.s = B.i;$ $B.i = A.s + 1$



Evaluación de atributos

Las gramáticas circulares son indeseables, puesto que están mal formadas y carecen de significado. Es imposible comenzar a calcular ninguno de los valores de los atributos en el ciclo.

- Existen **algoritmos para comprobar la no-circularidad de una gramática**, como el presentado por Knuth (1968, 1971), que es exponencial en el peor caso.
- De hecho, se ha demostrado (Jazayeri et al., 1975) que *no existe algoritmo que compruebe la circularidad de una gramática y que no sea exponencial en el peor de los casos*.
- Afortunadamente, la mayoría de las gramáticas no presentan este comportamiento del peor caso.

Gramáticas L-atribuidas y S-atribuidas

A veces las traducciones pueden hacerse evaluando las reglas semánticas de los atributos en un árbol de AS en un orden predeterminado.
En este caso la evaluación podría realizarse de forma eficiente.

Existen dos tipos de gramáticas atribuidas que permiten una evaluación eficiente de sus atributos:

- **Gramáticas S-atribuidas** ◂ GrSA Sólo contienen atributos sintetizados.
- **Gramáticas L-atribuidas** ◂ GrLA Pueden tener atributos sintetizados y heredados. En una regla de producción $A \rightarrow X_1 ... X_n$ cada atributo heredado de X_i sólo puede depender de:
  1. Atributos heredados de A.
  2. Atributos heredados o sintetizados asociados con las ocurrencias de los símbolos X_1 a X_{i-1}.
  3. Atributos heredados o sintetizados asociados con esa misma ocurrencia de X_i, pero siempre que no haya ciclos en un grafo de dependencias formado por los atributos de X_i.

Gramáticas S-atribuidas

Podemos evaluar los atributos de una gramática S-atribuida en cualquier orden de abajo hacia arriba de los nodos del árbol de AS. Sería sencillo hacerlo con una *recorrido en postorden* del árbol:

```
postorden(N)
{
 for (cada hijo C de N, de izquierda a derecha)
   postorden(C);
 evaluar los atributos asociados con N;
}
```

- *Puede realizarse una evaluación de los atributos al tiempo que se realiza un análisis LR*. El *postorden* corresponde exactamente al orden de reducciones de un analizador sintáctico LR.
- El análisis LR puede evaluar los atributos de un superconjunto de las gramáticas S-atribuidas: las **gramáticas LR-atribuidas** , en las que se basa Bison y que son un subconjunto de las L-atribuidas.

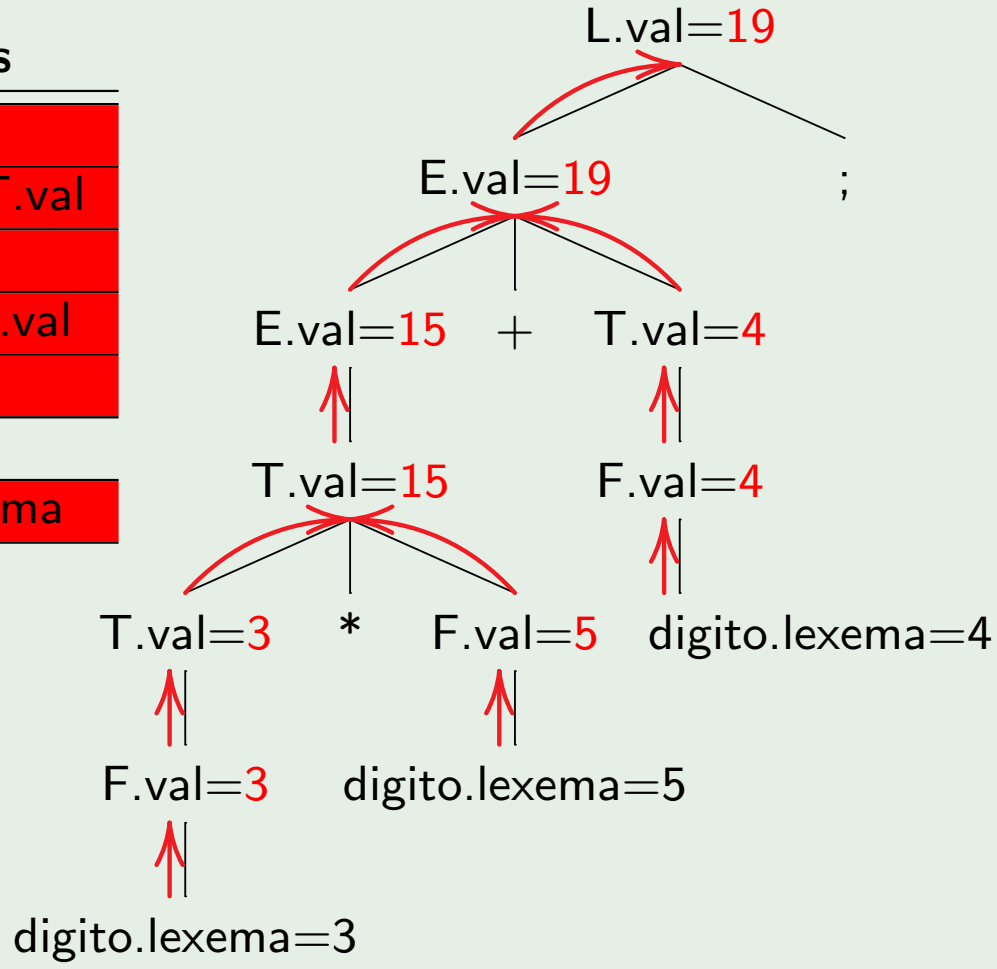
Gramáticas S-atribuidas

Ejemplo de DDS sólo con atributos sintetizados...

...para evaluación de expresiones aritméticas con +, * y ;

Producción	Reglas semánticas
$L \rightarrow E$;	L.val = E.val
$E \rightarrow E_1 + T$	E.val = E_1.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	T.val = T_1.val x F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
F → **digito**	F.val = **digito**.lexema

Árbol de análisis sintáctico anotado para 3*5+4;



◂ DefAtrSint

Gramáticas L-atribuidas

Toda gramática S-atribuida es también L-atribuida, pues las tres restricciones de ◂ GrLSA se refieren sólo a los atributos heredados.

Ejemplo: Cualquier definición dirigida por la sintaxis que contenga las siguientes reglas no puede ser L-atribuida.

PRODUCCION	REGLAS SEMANTICAS
$A \rightarrow B\ C$	$A.s = B.b;$ $B.i = f(C.c, A.s)$

Pueden evaluarse los atributos mediante un *recorrido en profundidad*:

```
visita(N)
{
 for (cada hijo C de N, de izquierda a derecha)
  {
   evaluar los atributos heredados de C;
   visita(C);
  }
 evalúa los atributos sintetizados de N;
}
```

Gramáticas L-atribuidas

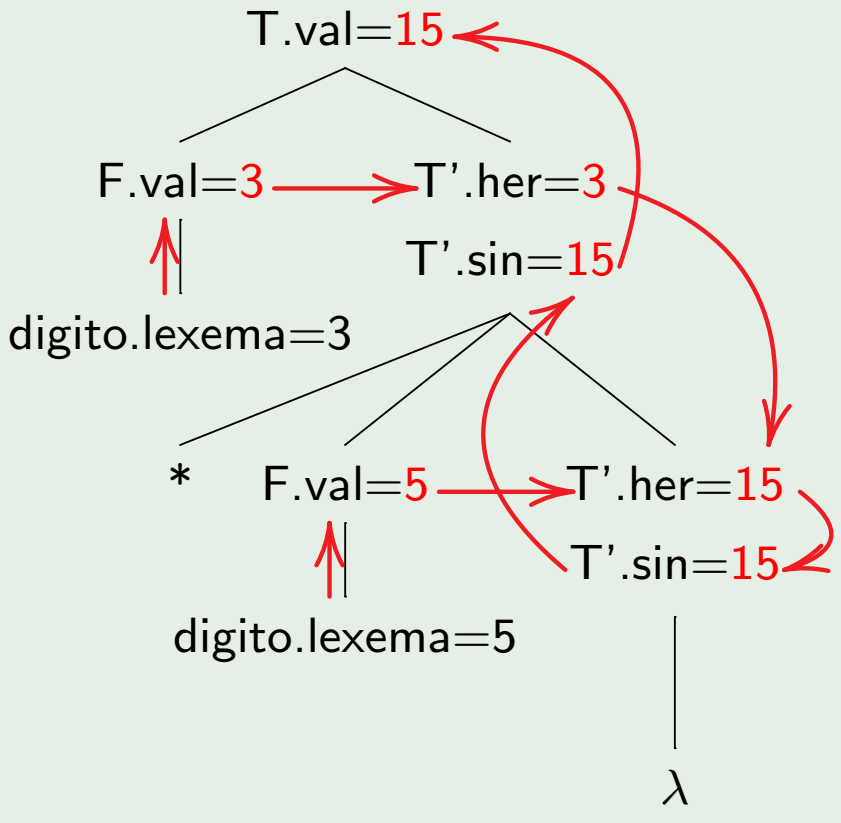
- Esto es adecuado para el análisis descendente.
- *Si la gramática subyacente es LL, este proceso puede llevarse a cabo al tiempo que se realiza el análisis sintáctico*.
- Las definiciones con atributos por la izquierda incluyen todas las definiciones dirigidas por la sintaxis basadas en gramáticas LL(1).

Gramáticas L-atribuidas

Ejemplo de DDS con atributos heredados por la izquierda

Volvemos al ejemplo que vimos en ◂ EjemploGD para ilustrar los grafos de dependencias.

Producción	Reglas semánticas
$T \rightarrow F\ T'$	$T'.her = F.val$
	$T.val = T'.sin$
$T' \rightarrow *\ F\ T'_1$	$T'_1.her = T'.her \times F.val$
	$T'.sin = T'_1.sin$
$T' \rightarrow \lambda$	$T'.sin = T'.her$
$F \rightarrow$ **digito**	$F.val =$ **digito.lexema**



DDS simples para traducción de expresiones

El orden en que se imprimen los caracteres es importante si la salida se crea incrementalmente sin utilizar almacenamiento.

Definición dirigida por la sintaxis simple

La cadena que representa la traducción del no terminal del lado izquierdo de cada producción es la concatenación de las traducciones de los no terminales de la derecha, en igual orden que en la producción, con algunas cadenas adicionales (tal vez ninguna) intercaladas.

Traducción de expresiones infijas a notación postfija

La **notación postfija** de una expresión E se puede definir así:

1. Si E es variable o constante, entonces E.
2. Si E es E_1 op E_2, entonces E'_1 E'_2 op, siendo E'_1 y E'_2 las notaciones postfijas de E_1 y E_2 respectivamente.
3. Si E es la expresión (E_1), entonces la notación postfija de E_1.

La notación postfija no necesita paréntesis. Por ejemplo:

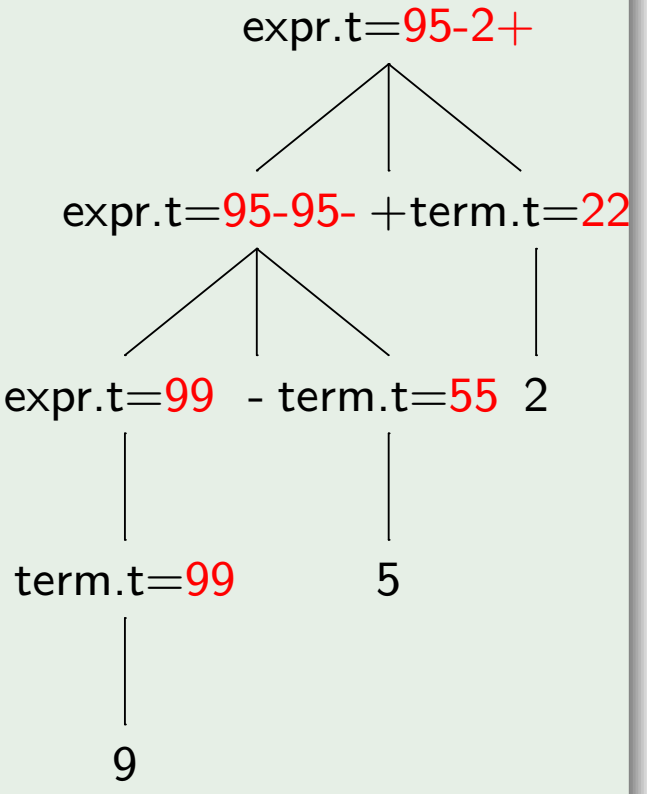
```
( 9 - 5 ) + 2    →    9 5 - 2 +
9 - ( 5 + 2 )    →    9 5 2 + -
```

DDS simples para traducción de expresiones

DDS para traducción de notación infija a postfija

◂ ET **Árbol de análisis sintáctico anotado** para la cadena de entrada 9−5+2 con un atributo *t* asociado con los no terminales *expr* y *term*. **El valor del atributo en la raíz es la notación postfija de la cadena de entrada**.

Producción	Reglas semánticas
expr → $expr_1$ + term	expr.t= $expr_1$.t || term.t || '+'
expr → $expr_1$ − term	expr.t= $expr_1$.t || term.t || '−'
expr → term	expr.t= term.t
term → 0	term.t= '0'
term → 1	term.t= '1'
term → 2	term.t= '2'
...	...
term → 5	term.t= '5'
...	...
term → 9	term.t= '9'



DDS para construcción de árboles de análisis sintáctico

→ Las definiciones dirigidas por la sintaxis pueden utilizarse para construir árboles sintácticos y otras representaciones gráficas de construcciones de lenguajes.

→ El uso de **árboles sintácticos como representación intermedia** permite que **la traducción se separe del análisis sintáctico**. Esto puede ser útil, por ejemplo, para no restringir el orden en que se consideran los nodos de un árbol de análisis sintáctico, pues puede no coincidir con el orden en que se va disponiendo de la información sobre una construcción.

Árboles sintácticos abstractos

- Son formas condensadas de árboles de análisis sintáctico, útiles para representar construcciones de lenguajes.

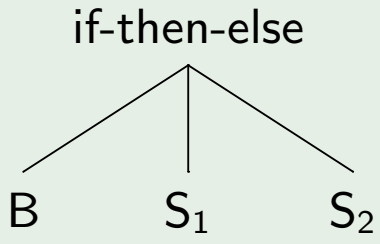
→ Los operadores y palabras clave no aparecen como hojas, sino que se asocian al nodo padre de dichas hojas.

→ Las producciones simples pueden desaparecer.

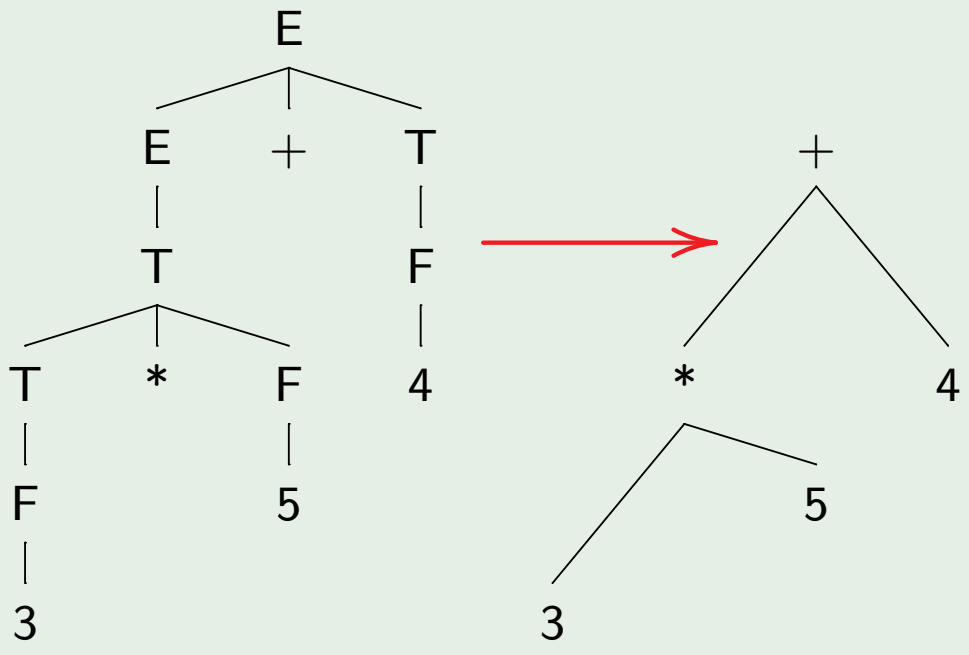
DDS para construcción de árboles de análisis sintáctico

Árbol de sintaxis abstracta...

...para la regla de producción
S → if B then S1 else S2:



...para la sentencia 3*5+4:

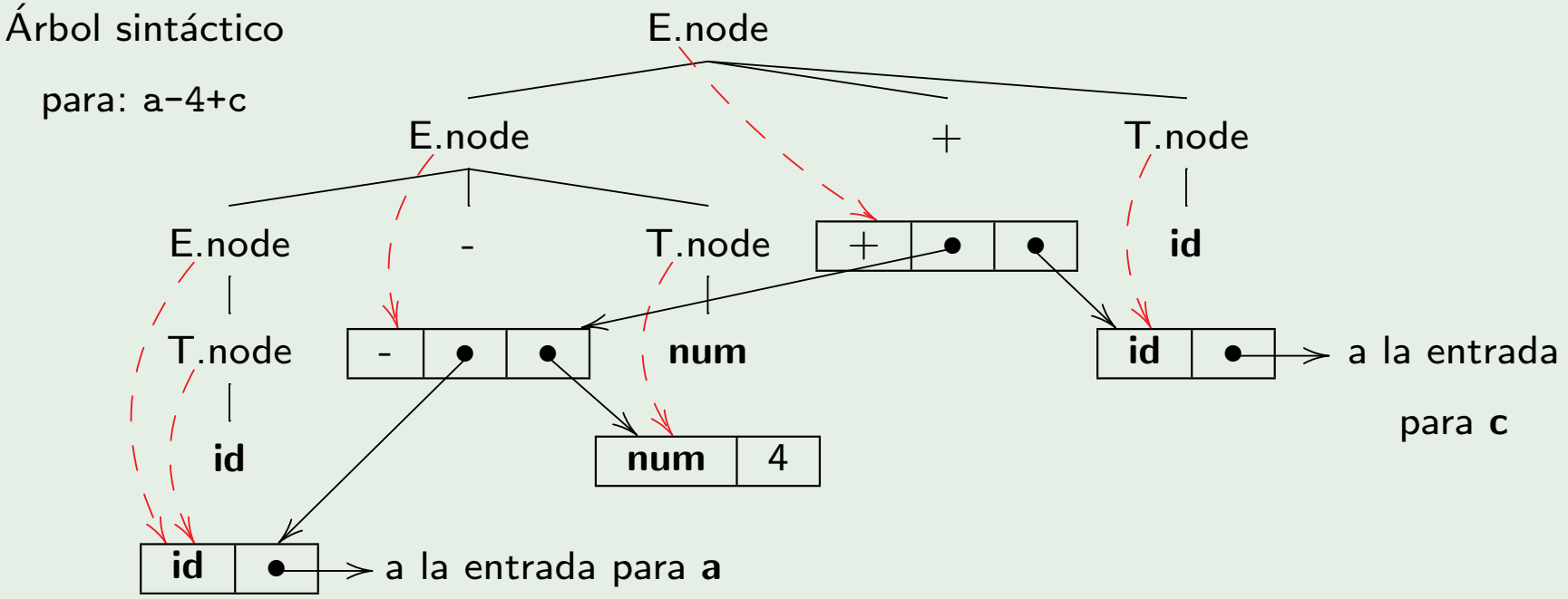


La traducción dirigida por la sintaxis puede basarse en este tipo de árboles usando el mismo enfoque que con árboles de análisis sintáctico.

DDS para construcción de árboles de análisis sintáctico

Definición con atributos sintetizados para árboles de AS para expresiones con + y -

Producción	Reglas semánticas
$E \rightarrow E_1 + T$	E.nodo = new Nodo('+',E_1.nodo,T.nodo)
$E \rightarrow E_1$ - T	E.nodo = new Nodo('-',E_1.nodo,T.nodo)
$E \rightarrow T$	E.nodo = T.nodo
$T \rightarrow (E)$	T.nodo = E.nodo
$T \rightarrow$ id	T.nodo = new Hoja(id,id.entrada)
$T \rightarrow$ num	T.nodo = new Hoja(num,num.valor)



Esquemas de traducción orientados por la sintaxis

Se trata de una notación complementaria a la DDS. Cualquier *DDS* puede implementarse con un *esquema de traducción*.

Un **esquema de traducción (ET)** es una gramática libre de contexto en la que se encuentran intercalados, en los lados derechos de las producciones, fragmentos de programa llamados **acciones semánticas**.

Especifican traducciones con una *notación más orientada a los procedimientos*.
A diferencia de la *DDS* el orden de evaluación de las reglas semánticas se muestra explícitamente.

ET dirigidos por la sintaxis postfijos

Tienen todas las *acciones semánticas* en los extremos derechos de los cuerpos de las producciones. Pueden implementarse durante el análisis LR mediante la ejecución de las acciones cada vez que se reduce.

Esquemas de traducción orientados por la sintaxis

ET dirigidos por la sintaxis con acciones en mitad de las reglas

Se colocan las acciones en cualquier lugar dentro de la parte derecha de una producción, ejecutándose de inmediato después de procesar todos los símbolos a su izquierda.

Ejemplo de ET con acción en mitad de la regla

$B \rightarrow X\{a\}Y$

La acción a se realiza después de que X sea reconocido (si es terminal) o todos los terminales derivados de él (si es no terminal). Por tanto:

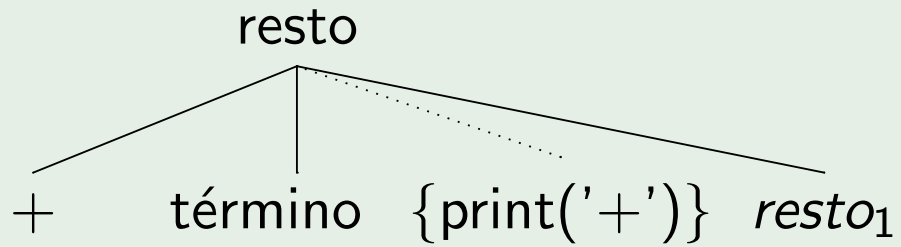
- Si el análisis sintáctico es ascendente, se ejecuta a cuando aparezca X en el tope de la pila de análisis sintáctico.
- Si el análisis sintáctico es descendente, se ejecuta a justo antes de expandir la ocurrencia de Y (si es no terminal) o encontrar un Y en la entrada (si es terminal).

Esquemas de traducción orientados por la sintaxis

Se puede implementar cualquier ET orientado por la sintaxis, **construyendo primero un árbol de AS** y realizando después las acciones en un orden de izquierda a derecha, con un recorrido en profundidad. En el árbol la acción se convierte en un hijo adicional conectado de forma especial.

Ejemplo

$resto \rightarrow + \ termino \ \{print('+')\} \ resto_1$



El símbolo '+' se imprime después de recorrer el subárbol de *termino* y antes de visitar a $resto_1$.

Esquemas de traducción orientados por la sintaxis

Generalmente se implementan durante el análisis sintáctico, sin construir el árbol de AS. Existen dos clases importantes de DDS que pueden implementarse con ET orientados a la sintaxis:

- La **gramática** subyacente es **LR** y la **DDS** es **S-atribuida**.
- La **gramática** subyacente es **LL** y la **DDS** es **L-atribuida**.

- En ambos casos las reglas semánticas en una *DDS* pueden convertirse en un *ET orientado a la sintaxis* con acciones que se ejecuten en el momento adecuado.
- Durante el análisis sintáctico, una acción en el cuerpo de una producción se ejecuta cuando todos los símbolos de la gramática a la izquierda de la acción han sido reconocidos.

No todos los ET orientados a la sintaxis pueden implementarse durante el análisis sintáctico.

Esquemas de traducción orientados por la sintaxis

Las **DDS simples** se pueden implementar con ET en los que las acciones impriman las cadenas adicionales en el orden en que aparecen en la definición.

ET dirigida por la sintaxis para traducir de notación infija a postija

En ◂ DDS vimos la DDS. El ET sería el siguiente:

PRODUCCION	ACCION
$expr \rightarrow expr_1 + term$	$\{print('+')\}$
$expr \rightarrow expr_1 - term$	$\{print('-')\}$
$expr \rightarrow term$	
$term \rightarrow 0$	$\{print('0')\}$
$term \rightarrow 1$	$\{print('1')\}$
...	...
$term \rightarrow 9$	$\{print('9')\}$

→ En general, para implementar un **ET simple**, se pueden ejecutar las *acciones semánticas* durante el análisis sintáctico: **No es necesario construir el árbol de análisis sintáctico**.

Tabla de símbolos y comprobaciones estáticas

- **Comprobación dinámica** Aquella que se realiza durante la ejecución del programa objeto.
- **Comprobación estática** La realizada por el compilador para garantizar que el programa fuente cumpla las especificaciones sintácticas y semánticas del lenguaje. Incluye:
  - **Comprobación sintáctica** *Para comprobar la sintaxis de un lenguaje de programación, se usa una* **gramática libre de contexto** *, pero la mayoría de estos lenguajes son generados por* **gramáticas dependientes del contexto** . Esas restricciones sensibles al contexto deben también chequearse. Por ejemplo: Comprobaciones de unicidad o de flujo de control.
  - **Comprobación de tipos** Las reglas sobre los tipos de un lenguaje aseguran que un operador o función se aplique al número y tipo de operandos correctos. Por ejemplo, que el número y tipo de parámetros en una función sean correctos. También realiza **conversión de tipos** si es necesaria, insertando un operador en el árbol de análisis sintáctico. Reúne, por tanto, información para la generación de código.

Tabla de símbolos y comprobaciones estáticas

- A veces se combina la comprobación estática y la generación de código intermedio con el análisis sintáctico (Ej.: Algunos compiladores de Pascal).
- Otras veces la comprobación de tipos se hace en una pasada independiente del análisis sintáctico y la generación de código intermedio (Ej.: Ada).

En muchas de las comprobaciones estáticas es necesario hacer uso de una **tabla de símbolos**.
Las **tablas de símbolos** son estructuras de datos que utilizan los compiladores para guardar información acerca de las construcciones de un programa fuente.

→ Las fases de análisis de un compilador **recolectarán la información de forma incremental**.

→ Las fases de síntesis **la usarán para generar código destino**.

Tabla de símbolos y comprobaciones estáticas

Cada entrada en una tabla de símbolos guardará información acerca de un identificador (su lexema, su tipo, su posición en el espacio de almacenamiento, etc.). De hecho, **la función de una tabla de símbolos es pasar información de las declaraciones a los usos**:

- → Una acción semántica "coloca" información acerca del identificador x en la tabla de símbolos cuando se analiza la declaración de x.
- → Una acción semántica asociada con una producción semejante a $\text{factor} \rightarrow \text{id}$ "obtiene" información acerca de ese identificador consultando la tabla de símbolos.

Generalmente las tablas de símbolos deben soportar varias declaraciones de un identificador en el mismo programa. De esta forma, hay que tener en cuenta el **alcance de una declaración** , o la **parte del programa a la cual se aplica dicha declaración**, a la hora de implementar la tabla de símbolos. Esto se resuelve asignando una tabla de símbolos a cada bloque de programa con declaraciones, a cada clase (con una entrada para cada campo y cada método), etc.

Tipos y declaraciones

Las aplicaciones de los tipos son:

- Comprobación de tipos **Usa reglas lógicas para razonar acerca del comportamiento de un programa en tiempo de ejecución**.
  - → Se asegura de que el tipo de una construcción coincida con el previsto en su contexto. En particular, que los tipos de los operandos coincidan con el tipo esperado por un operador. Por ejemplo, el operador && de Java espera operandos booleanos y el resultado también es booleano.
- Aplicaciones de traducción [1] **A partir del tipo de un nombre, un compilador puede determinar el almacenamiento necesario para ese nombre en tiempo de ejecución**. La información del tipo también se necesita para calcular la dirección que denota la referencia a un array, para insertar conversiones de tipo explícitas y para elegir la versión correcta de un operador aritmético, entre otras.

[1] Las veremos en el tema siguiente.

Expresiones de tipos

Una **expresión de tipo** es una estructura que se utiliza para denotar el tipo de una construcción de un lenguaje. Es, o bien un **tipo básico**, o se forma mediante la **aplicación de un operador llamado constructor de tipos a una expresión de tipos**. *(Ver siguiente transparencia)*

Los conjuntos de tipos básicos y los constructores dependen del lenguaje que se va a comprobar.

Las expresiones de tipos pueden representarse usando:

- Grafos
- Árboles
- GDAs[2]

En ellos...

→ ...los nodos interiores corresponden a los **constructores**

→ ...y las hojas a los **tipos básicos**, **nombres de tipo** y **variables de tipo**.

[2] Veremos los GDA's en el próximo tema.

1. Un **tipo básico** es una **expresión de tipo** . Tipos básicos: *boolean, char, integer, float y void*.
2. Un **nombre de un tipo** es una **expresión de tipo** .
3. Un **constructor de tipos aplicado a expresiones de tipos** es una **expresión de tipo** :
    1. Matrices: Si T es una expresión de tipo, entonces $array(I, T)$ es una expresión de tipo que indica el tipo de una matriz con elementos de tipo T y conjunto de índices I, que suele ser un rango de enteros.
    2. Productos: Si T_1 y T_2 son expresiones de tipos, entonces su producto cartesiano $T_1 x T_2$ es una expresión de tipo.
    3. Registros: Como los productos, pero con nombre en los campos. Se usa el constructor de tipos *record* aplicado a una tupla formada con nombres y tipos de campos.
    4. Apuntadores: Si T es una expresión de tipo, entonces $pointer(T)$ es una expresión de tipo que indica el tipo "apuntador a un objeto de tipo T".
    5. Funciones: Se consideran como transformaciones de un dominio de tipo D a un rango de tipo R. El tipo de una función viene indicado por la expresión de tipo $D \rightarrow R$.
4. Pueden contener **variables** cuyos valores son **expresiones de tipos** .

Equivalencia de tipos

Las reglas de comprobación de tipos suelen tener la forma:

`if` dos expresiones de tipo son iguales `then` devuelve un tipo

`else` devuelve error_tipo

→ **Se necesita una definición de equivalencia de expresiones de tipos .**

→ Los compiladores usan representaciones para determinarla rápidamente.

Nombres de tipos y tipos recursivos

El nombre de una clase, una vez definida, se puede usar como el nombre de un tipo en C++ o Java:

```
public class Nodo {...}
   ...
public Nodo n;
```

Pueden usarse nombres para definir **tipos recursivos**, para estructuras de datos como las listas:

```
class Celda { int info; Celda siguiente;...}
```

(Ver en apuntes ejemplos de uso de equivalencia de nombre en Pascal y C para evitar **ciclos en la representación de tipos***)*

Equivalencia de tipos

- **Equivalencia estructural** Dos expresiones son estructuralmente equivalentes si *son el mismo tipo básico* o *están formadas aplicando el mismo constructor a tipos estructuralmente equivalentes*, es decir, sii **son idénticas** cuando todos los nombres de tipos, si los hay, han sido sustituidos por las expresiones a las que representan. *(Ver en apuntes algoritmo para comprobar la equivalencia estructural de dos expresiones de tipos)*
- **Equivalencia de nombre** Con la **equivalencia de nombres** se considera cada nombre de un tipo como un tipo distinto, de forma que dos expresiones de tipo tienen equivalencia de nombre si, y sólo si, son idénticas sin sustituir los nombres.

Los conceptos de equivalencia estructural y de nombre son útiles para explicar las reglas que usan algunos lenguajes para asociar tipos con identificadores en las declaraciones. *(Ver en apuntes ejemplo de equivalencia de nombre en Pascal)*

Comprobación de tipos

Un **sistema de tipos** está constituido por reglas que asignan una **expresión de tipos** a cada parte de un programa.
Un **comprobador de tipos** implementa un **sistema de tipos**.

- Los lenguajes con **tipos fuertes** garantizan, en tiempo de compilación, que los programas se ejecutarán sin errores, no permitiendo que se lleve a cabo una operación con argumentos del tipo equivocado.
  Ejemplos: Haskell y Java.
  En ocasiones se les denomina lenguajes con **tipos seguros**.
- Los lenguajes con **tipos débiles** permiten que un lenguaje convierta implícitamente tipos cuando son usados, o que un valor de un tipo sea tratado como de otro. **Esto puede ser útil pero puede conducir a errores**.
  Ejemplos: C y C++.
  A veces se les denomina lenguajes con **tipos inseguros**.

Comprobación de tipos

La noción de tipos varía de unos lenguajes a otros según, por ejemplo, que se realice de forma **estática** o **dinámica**, que se considere la **equivalencia estructural** o **de nombre** o bien que los tipos sean **fuertes** o **débiles**.

→ Cualquier comprobación puede realizarse dinámicamente (tipo de un elemento + valor en el código objeto).

→ En la práctica, hay comprobaciones que tienen que hacerse dinámicamente.

Ver tabla con la clasificación de lenguajes según su sistema de tipos en pág. 197 de "Design and implementation of Compiler", R. Singh, V. Sharma y M. Varhney.

- Diferentes compiladores del mismo lenguaje podrían utilizar diferentes sistemas de tipos.
- Un comprobador de tipos debe hacer **recuperación de errores**.

Síntesis e inferencia

La comprobación de tipos puede tomar dos formas: **síntesis** e **inferencia**.

La **síntesis de tipos** construye el tipo de una expresión a partir de los tipos de sus subexpresiones. Requiere que se declaren los nombres antes de utilizarlos.

Ejemplo

El tipo de E_1+E_2 se define en términos de los tipos de E_1 y E_2.

Una **regla común para la síntesis de tipos** en funciones con un argumento[3] tiene la siguiente forma:

`if` f tiene el tipo $s \to t$ `and` x tiene el tipo s,

`then` la expresión $f(x)$ tiene el tipo t

[3] Podría generalizarse a más de un argumento y adaptarse a la expresión de la suma anterior si la consideramos como una aplicación de la función $sumar(E_1, E_2)$.

Síntesis e inferencia

La **inferencia de tipos** determina el tipo de una construcción del lenguaje a partir de la forma en que se utiliza.

Ejemplo

Supongamos que *null* es una función que evalúa si una lista está vacía. Entonces del uso de *null*(x) puede concluirse que x debe ser una lista, aunque no sabemos aún de qué.

Una **regla común para inferencia de tipos** tiene la siguiente forma[4]:

`if` $f(x)$ es una expresión,

`then` para cierta α y β, f tiene el tipo $\alpha \to \beta$ `and` x tiene el tipo α

La inferencia de tipos es necesaria para lenguajes como ML, que comprueban los tipos pero no requieren la declaración de nombres.

[4] Las variables que representan expresiones de tipos (α, β,...) permiten hablar de tipos desconocidos.

Conversiones de tipos

→ **La definición del lenguaje especificará las conversiones**:

- Asignación de enteros a reales o viceversa → se convierte al tipo del lado izquierdo.
- Expresiones → se suele convertir el entero en real.
- ...

→ **El comprobador de tipos puede utilizarse para insertar las operaciones de conversión**.

Ejemplo

Tenemos $x + i$, x de tipo punto flotante e i de tipo entero ⇒ el compilador debe convertir uno de los operandos antes de la suma.
La *representación intermedia* para la expresión $2 * 3{,}14$ sería:

$t_1 = (float)\ 2$
$t_2 = t_1 * 3{,}14$

Esquema si sólo tuviéramos estos dos tipos posibles:

```
if (E1.tipo = integer and E2.tipo = integer) E.tipo = integer
else if (E1.tipo = integer and E2.tipo = float) E.tipo = float
else if (E1.tipo = float and E2.tipo = integer) E.tipo = float
...
```

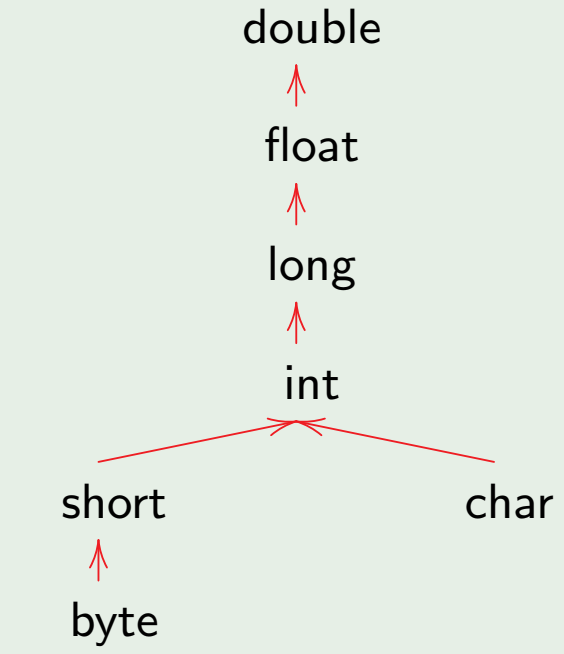
Conversiones de tipos

- → A medida que se incrementa el número de tipos sujetos a conversión, el número de casos aumenta con rapidez.
- → Por tanto, con número extensos de tipos, es importante una organización cuidadosa de las acciones semánticas.

Las reglas de conversión de tipos varían de un lenguaje a otro.

Conversiones de tipos

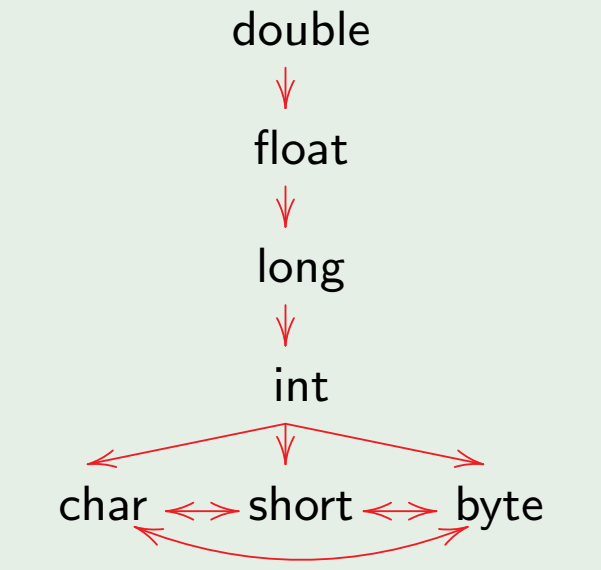
Las reglas de Java distinguen entre:



conversiones de ampliación

↓

preservan la información



conversiones de reducción

↓

puede perderse información

Conversiones de tipos

Las conversiones de tipo acompañan generalmente a la **sobrecarga de operadores** (significados distintos según contexto).

- **Conversión explícita o cast** El programador debe escribir algo para motivarla.
  - → Para el comprobador de tipos es igual que una aplicación de función.

  Ejemplos: Conversiones de Ada (todas). En Pascal: ord (carácter a entero) y chr (al revés).
- **Conversión implícita o coerción** El compilador la realiza de forma automática.
  - → Suelen limitarse a conversiones de ampliación (entero a real).
  - → En la práctica puede producirse pérdida de información.

  Ejemplo: Conversión implícita en C de caracteres ASCII a enteros entre 0 y 127 en expresiones aritméticas.

  En la pág. 389 de la edición del 2008 del *libro del dragón* aparece el seudocódigo para generar instrucciones para conversión (implementan reglas de ampliación de ◂ Ampl).

Conversiones de tipos

La conversión implícita de constantes suele realizarse durante la compilación para mejorar el tiempo de ejecución del programa.

Ejemplo: Uso de un compilador de Pascal con las siguientes sentencias, dónde X es una matriz de reales

```
for  I := 1  to  N  do  X[I] := 1
```
$\rightarrow$ Más lenta

```
for  I := 1  to  N  do  X[I] := 1.0
```
$\rightarrow$ Más rápida

El compilador debería hacer la conversión implícita.