TEMA 4

ANÁLISIS SEMÁNTICO Y TRADUCCIÓN DIRIGIDA POR LA SINTAXIS

Índice

EL análisis semántico es la fase del compilador encargada de realizar las comprobaciones sensibles al contexto del lenguaje analizado. En este tema se estudia una técnica conocida como *traducción dirigida por la sintaxis* que permite expresar, por cada regla de la gramática libre de contexto del lenguaje, una serie de acciones semánticas asociadas a la misma. De forma simultánea al análisis sintáctico se aplican dichas acciones cada vez que se reconoce la regla de producción correspondiente. Estas acciones permiten no sólo hacer las comprobaciones sensibles al contexto del lenguaje, sino también generar una representación intermedia o final de la entrada de forma simultánea al recorrido del árbol de análisis.

4.1. Definiciones dirigidas por la sintaxis

Para hacer un tratamiento sistemático de las operaciones correspondientes a la fase de análisis semántico, los textos sobre compiladores proponen el uso de la técnica conocida como *traducción dirigida por la sintaxis*. La técnica se basa en la estructura sintáctica del lenguaje para definir su semántica. Con este fin, el diseñador del lenguaje establece:

- por cada símbolo de la gramática, un *conjunto de atributos semánticos*.
- por cada regla de producción, un *conjunto de reglas semánticas* que determinan los valores de los atributos semánticos asociados a los símbolos que aparecen en esa producción.

A la combinación de la gramática y el conjunto de reglas semánticas se le denomina *definición dirigida por la sintaxis*.

Ejemplo 4.1. Para hacer una traducción de notación infija a notación postfija de expresiones aritméticas, podríamos tener, entre otras, la siguiente regla de producción de la gramática, con su correspondente regla semántica:

PRODUCCIÓN	REGLA SEMÁNTICA
$E \to E_1 + T$	$E.codigo = E_1.codigo \,\|\|\, T.codigo \,\|\|\, '+'$

donde $||$ se debe interpretar como un operador de concatenación de cadenas. En el ejemplo, los no terminales E y T tienen un atributo semántico, denominado $codigo$, que es una secuencia de caracteres con la representación postfija de la subcadena derivada del no terminal. La regla semántica describe qué valor toma el atributo del no terminal E a la izquierda de la producción. En este caso, es el resultado de concatenar el código del no terminal E_1 a la derecha (obsérvese el sufijo para distinguirlo del no terminal E de la cabeza de la regla), seguido del código del no terminal T, y seguido del carácter $+$.

Como vemos en el ejemplo anterior, si X es un símbolo y a uno de sus atributos, entonces escribimos $X.a$ para denotar el valor del atributo a en el nodo específico de un árbol de análisis sintáctico etiquetado como X. Se denomina *árbol de análisis sintáctico con anotaciones* a un árbol de análisis sintáctico que muestra los valores de los atributos de cada nodo.

El ejemplo anterior únicamente muestra el cálculo de un valor semántico. A veces es conveniente que las definiciones dirigidas por la sintaxis incluyan, en las reglas semánticas, *efectos adicionales* al cálculo de atributos, como puede ser imprimir un resultado o interactuar con la tabla de símbolos. En el caso de que las reglas semánticas de una definición dirigida por la sintaxis sólo evalúen atributos, es decir, cuando no tengan efectos adicionales, hablaremos de una *gramática atribuida*.

4.1.1. Atributos sintetizados y heredados

Un atributo puede representar cualquier valor (tipo de dato de un identificador, cadena de caracteres, posición de memoria, etc.). En una definición dirigida por la sintaxis podemos considerar como atributos de un símbolo terminal a su lexema, o bien a su entrada en la tabla de símbolos, proporcionados por el

analizador léxico. Por otra parte, podemos distinguir dos tipos de atributos para un símbolo no terminal: atributos sintetizados y atributos heredados.

4.1.1.1. Atributos sintetizados

Se denomina *atributo sintetizado* de un no terminal A en un nodo N de un árbol de análisis a un atributo cuyo valor se calcula sólo usando los valores de los atributos de los hijos de N y de otros atributos de N, mediante una regla semántica asociada con la producción usada para expandir N. La producción debe tener a A en su parte izquierda.

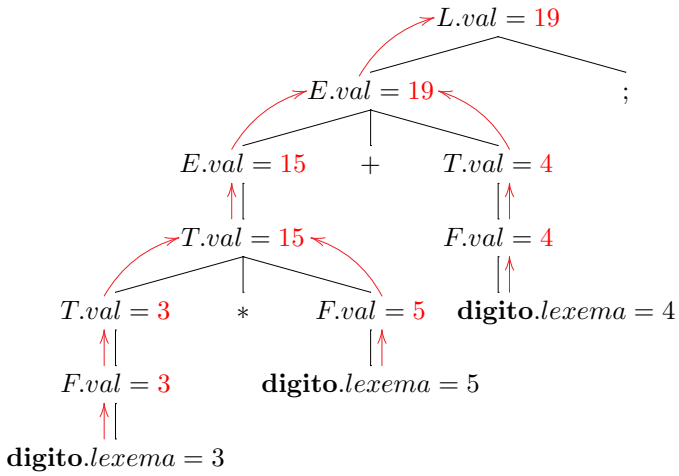
La característica más interesante de estos atributos es que pueden calcularse durante un solo recorrido en postorden del árbol de análisis sintáctico.

Ejemplo 4.2. Supongamos que queremos desarrollar una calculadora. El usuario introduce una cadena de texto con una expresión aritmética finalizada con **;** y la calculadora debe analizar la cadena, evaluar la expresión y calcular su resultado. Podemos plantear la siguiente definición dirigida por la sintaxis para hacer el cálculo necesario:

PRODUCCIÓN	REGLAS SEMÁNTICAS
$L \to E;$	$L.val = E.val$
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val \times F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to \mathbf{digito}$	$F.val = \mathbf{digito}.lexema$

Podemos ver que todos los no terminales tienen un único atributo sintetizado de tipo numérico, llamado *val*. En cuanto al terminal **digito**, se considera que tiene un atributo *lexema* que contiene el valor numérico de su lexema asociado.

El árbol de análisis sintáctico anotado para la expresión $3 * 5 + 4;$ se muestra a continuación:



Las flechas indican una dependencia entre dos atributos en el cálculo de los valores semánticos.

4.1.1.2. Atributos heredados

Se denomina *atributo heredado* de un no terminal B en un nodo N de un árbol de análisis a un atributo cuyo valor se calcula sólo usando los valores de los atributos del nodo padre de N, del mismo N, y de sus hermanos, mediante una regla semántica asociada con la producción usada para expandir el padre de N. La producción debe tener a B en su parte derecha.

Los atributos heredados son útiles cuando la gramática está más orientada al análisis sintáctico que a la traducción. Esto sucede cuando la gramática es el resultado de haber aplicado algoritmos de transformación para ajustarla a un análisis sintáctico que requiera ciertas características. Por ejemplo, si queremos aplicar un análisis LL a una gramática, puede ser necesario eliminar su recursividad izquierda y factorizarla previamente. La gramática transformada tiene una forma más acorde con el análisis sintáctico que con la traducción, y en consecuencia se puede requerir el uso de atributos heredados para hacer la traducción dirigida por la sintaxis.

En otros casos, se pueden emplear atributos heredados simplemente por claridad en la traducción. No obstante, siempre es posible escribir una definición dirigida por la sintaxis que sólo utilice atributos sintetizados.

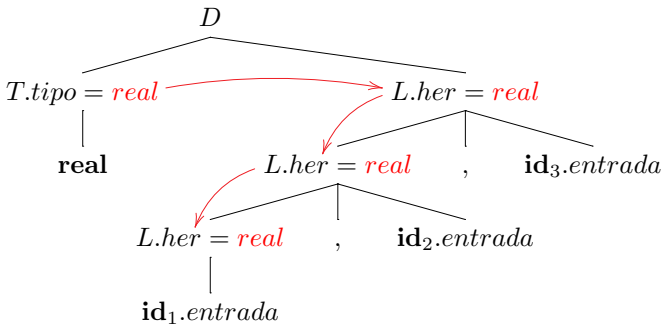
Ejemplo 4.3. La siguiente definición dirigida por la sintaxis permite actualizar la tabla de símbolos de un compilador con la información de las variables declaradas en un lenguaje similar a C. La sintaxis permite reconocer declaraciones como `real a,b,c`.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$D \rightarrow T\ L$	$L.her = T.tipo$
$T \rightarrow \mathbf{int}$	$T.tipo = integer$
$T \rightarrow \mathbf{real}$	$T.tipo = real$
$L \rightarrow L_1$,**id**	$L_1.her = L.her$
	$agregarTipo(\mathbf{id}.entrada, L.her)$
$L \rightarrow \mathbf{id}$	$agregarTipo(\mathbf{id}.entrada, L.her)$

El no terminal D representa una declaración formada por un tipo T, seguido de una lista de identificadores de variables L. El no terminal T tiene un atributo *tipo*, cuyo valor es un código que determina el tipo de la declaración D. El no terminal L tiene un atributo heredado llamado her. El propósito de este atributo es pasar el tipo declarado hacia abajo en el árbol de análisis, para actualizar las entradas de los identificadores en la tabla de símbolos en los nodos etiquetados con L.

Las dos últimas producciones tienen un efecto adicional, que es la llamada a la función $agregarTipo$. Esta función tiene dos argumentos: la entrada de la tabla de símbolos del identificador, y el tipo de la declaración del identificador. Su efecto es guardar en la tabla de símbolos el tipo correspondiente al identificador.

El árbol de análisis sintáctico con anotaciones para la entrada `real a,b,c` se muestra a continuación:



Como podemos ver en el árbol anotado, el atributo *her* del no terminal L recibe su valor bien de un nodo a su izquierda, o bien de un nodo padre.

4.1.2. Evaluación de los atributos

Una definición dirigida por la sintaxis no establece ningún orden específico para la evaluación de los atributos del árbol anotado. Como es obvio, antes de calcular el valor de un atributo en un nodo del árbol, se deben evaluar todos los atributos de los que éste depende, por lo que se pueden presentar situaciones en las que:

- es necesario evaluar los atributos de un nodo antes que los atributos de sus hijos.
- es necesario evaluar los atributos de un nodo después que los atributos de sus hijos.
- es necesario evaluar los atributos de un nodo entre la evaluación de los atributos de unos hijos y otros.

Para reflejar las dependencias en la evaluación de los atributos se emplean *grafos de dependencias*. Un grafo de este tipo describe el flujo de información entre los atributos de un árbol anotado concreto. Los nodos del grafo son los atributos semánticos, y las aristas son las dependencias. Cada arista asocia dos atributos: el atributo al que apunta se calcula en función del valor del atributo del que parte. Los ejemplos 4.2 y 4.3 muestran árboles anotados en los que se ha superpuesto el grafo de dependencias.

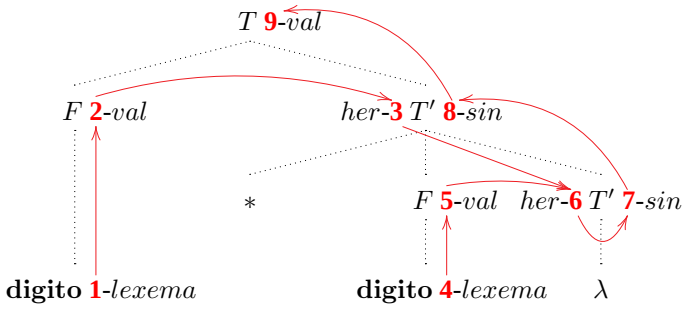
Ejemplo 4.4. El ejemplo 3.26 muestra la aplicación de los algoritmos de eliminación de la recursividad izquierda y factorización sobre la gramática no ambigua que describe las expresiones aritméticas. Estas transformaciones dejan la gramática preparada para el análisis $LL(1)$. La siguiente es una definición dirigida por la sintaxis para las reglas de producción relativas a la operación de multiplicación:

PRODUCCIÓN	REGLAS SEMÁNTICAS
$T \to FT'$	$T'.her = F.val$ $T.val = T'.sin$
$T' \to *FT'_1$	$T'_1.her = T'.her \times F.val$ $T'.sin = T'_1.sin$
$T' \to \lambda$	$T'.sin = T'.her$
$F \to \mathbf{digito}$	$F.val = \mathbf{digito.lexema}$

Los no terminales T y F tienen un atributo sintetizado val que almacena el resultado de la evaluación del término o del factor, respectivamente. El terminal **digito** tiene un atributo $lexema$ que almacena la representación numérica del dígito. El no terminal T', que se debe considerar como un símbolo auxiliar para el análisis descendente, tiene dos atributos: her, atributo heredado, y sin, atributo sintetizado.

Las reglas semánticas se basan en la idea de manejar el operando izquierdo de una multiplicación como un atributo heredado. El atributo her de T' cumple esta función. Dado el término $x * y * z$, la subexpresión $* y * z$ se representa con la aplicación de la regla $T' \to *FT'_1$, y el valor de x se almacena en la cabeza de la regla, es decir, en $T'.her$. Una vez evaluado $x * y$, se almacena el resultado en $T'_1.her$. Seguidamente, la subexpresión $* z$ se vuelve a representar con una nueva aplicación de la regla $T' \to *FT'_1$, de modo que la cabeza de la regla contiene $x * y$. Una vez que se ha completado el cálculo, el resultado se mueve hacia la raíz del árbol mediante los atributos sintetizados $T'.sin$, y finalmente se almacena en $T.val$.

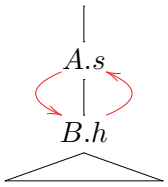
A continuación se muestra el árbol anotado para el producto de dos dígitos, usando la definición dirigida por la sintaxis anterior. Sobre el árbol de análisis se indica el grafo de dependencias para la evaluación del producto. Cada nodo del grafo de dependencias se representa con un número del 1 al 9, que indica el orden de evaluación. Junto a cada número, unido por un guión, aparece el nombre del atributo que se evalúa.



Se dice que una gramática atribuida es *no circular* cuando no es posible encontrar ningún grafo de dependencias con ciclos. En este caso, siempre es posible encontrar un *orden topológico* para la evaluación de los atributos. En caso contrario, no es posible. Por ejemplo, consideremos los no terminales A y B con un atributo sintetizado $A.s$ y uno heredado $B.h$, junto con las siguientes reglas semánticas:

PRODUCCIÓN	REGLAS SEMÁNTICAS
$A \to B$	$A.s = B.h;$ $B.h = A.s + 1$

Estas reglas son circulares, es decir, es imposible evaluar $A.s$ en un nodo N o $B.h$ en un nodo hijo de N. La dependencia circular de $A.s$ y $B.h$ se puede representar con el grafo:



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

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

Hay dos subclases de gramáticas atribuidas que garantizan la existencia de un orden de evaluación de los atributos. En concreto, los atributos se evalúan eficientemente de acuerdo con un orden predeterminado de recorrido del árbol de análisis con anotaciones. Estos dos tipos de gramáticas se estudian en las siguientes subsecciones.

4.1.2.1. Gramáticas S-atribuidas

Las gramáticas *S-atribuidas* son aquellas que sólo contienen atributos sintetizados. En este tipo de gramáticas, podemos evaluar sus atributos en cualquier orden de abajo hacia arriba de los nodos del árbol de análisis sintáctico. Por tanto, sería sencillo hacerlo con una recorrido en *postorden* del árbol, evaluando los atributos del nodo N cuando el recorrido sale de N por última vez:

Función de evaluación de gramáticas S-atribuidas.

```
1 postorden(N) {
2       for (cada hijo C de N, de izquierda a derecha)
3             postorden(C);
4       evaluar los atributos sintetizados de N;
5 }
```

En este caso puede realizarse una evaluación de los atributos al tiempo que se realiza un análisis LR. De hecho, el postorden corresponde exactamente al orden en que un analizador sintáctico LR hace las reducciones. El análisis LR, sin embargo, puede evaluar los atributos de un tipo de gramáticas que constituyen un superconjunto de las S-atribuidas: las gramáticas LR-atribuidas, en las que se basa Bison y que son, por otro lado, un subconjunto de las L-atribuidas, que se tratan en la siguiente sección.

El ejemplo 4.2 muestra un caso de gramática S-atribuida.

4.1.2.2. Gramáticas L-atribuidas

Las gramáticas *L-atribuidas* pueden tener atributos sintetizados y heredados. En una regla de producción $A \to X_1 \ldots X_n$ cada atributo heredado del símbolo 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.

Toda gramática S-atribuida es también L-atribuida, pues las restricciones 1, 2 y 3 se refieren sólo a los atributos heredados.

Ejemplo 4.5. Cualquier definición dirigida por la sintaxis que contenga reglas similares a las siguientes **no** puede ser L-atribuida.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$A \to B\ C$	$A.s = B.b$; $B.h = f(C.c, A.s)$

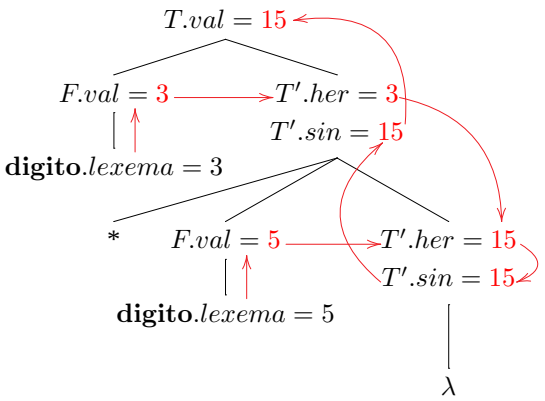
Estas condiciones permiten la evaluación de los atributos en una única pasada de izquierda a derecha, mediante un recorrido en *profundidad* (empieza en la raíz y visita recursivamente a los hijos de cada nodo en orden de izquierda a derecha):

Función de evaluación de gramáticas L-atribuidas.

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

Este recorrido es adecuado para el análisis descendente. De hecho, si la gramática subyacente es LL, este proceso puede llevarse a cabo al tiempo que se realiza el análisis sintáctico. Las definiciones con gramáticas L-atribuidas incluyen a todas las definiciones dirigidas por la sintaxis basadas en gramáticas $LL(1)$.

Ejemplo 4.6. El ejemplo 4.4 muestra una definición dirigida por la sintaxis con atributos heredados que cumple las condiciones para ser una gramática L-atribuida. El siguiente árbol anotado muestra el orden de evaluación de `3*5`:



4.1.3. Aplicaciones de las definiciones dirigidas por la sintaxis

Como se indicó en el primer tema, un compilador puede hacer uso de representaciones intermedias de los programas de entrada. En esta sección se muestran dos ejemplos de uso de las definiciones dirigidas

por la sintaxis para construir casos simples de representaciones intermedias. En el primero se muestra cómo realizar la traducción de expresiones aritméticas usando cadenas de caracteres, de forma que la notación infija de la entrada se transforma a notación postfija en la salida. En el segundo se muestra cómo construir árboles sintácticos que representen las expresiones aritméticas analizadas.

4.1.3.1. Definiciones simples dirigidas por la sintaxis para la traducción de expresiones

El tipo más simple de representación intermedia consiste en una cadena de caracteres cuyo contenido sea equivalente a la entrada en un nuevo lenguaje. Este lenguaje es más sencillo que el de la entrada, y más apto para la traducción final al lenguaje de salida.

En esta sección se describe un traductor de expresiones aritméticas que realiza una conversión de notación infija a notación postfija. La notación postfija es más sencilla, ya que no necesita paréntesis, porque la posición y el número de argumentos de los operadores sólo permiten una interpretación.

En general una *definición dirigida por la sintaxis simple* se rige por el siguiente principio: 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.

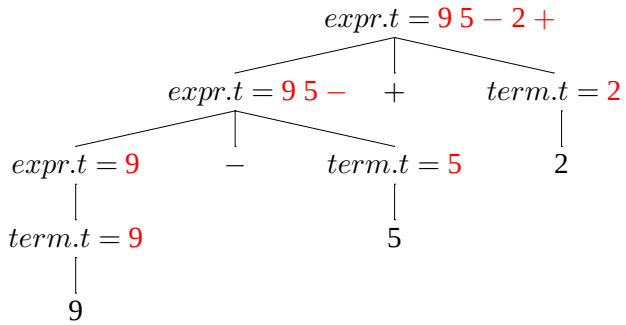
La notación postfija de una expresión E se puede definir de forma inductiva como sigue:

1. Si E es variable o constante, entonces su notación postfija es E.
2. Si E es la expresión $E_1\ op\ E_2$, donde op es un operador binario, entonces su notación postfija es $E_1'\ E_2'\ op$, 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 su notación postfija es la notación postfija de E_1.

Esta traducción queda reflejada en la siguiente definición dirigida por la sintaxis:

PRODUCCIÓN	REGLAS SEMÁNTICAS
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|\, '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|\, '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

donde || representa el operador de concatenación de cadenas de caracteres. Esta gramática es S-atribuida. El atributo t de los no terminales $expr$ y $term$ es de tipo cadena de caracteres[1]. A continuación se muestra el árbol anotado para la cadena de entrada `9-5+2`:



[1] La representación intermedia se puede crear incrementalmente sin utilizar almacenamiento para traducir subexpresiones. Como se indica más adelante, en este caso es necesario tener en cuenta el orden en el que se imprimen los caracteres de salida.

4.1.3.2. Definición dirigida por la sintaxis para la construcción de árboles sintácticos

Las definiciones dirigidas por la sintaxis pueden emplearse para construir representaciones intermedias de la entrada en forma de árbol sintáctico. En este contexto, un *árbol sintáctico* es una estructura de datos que resume la información más importante del propio árbol de análisis derivado por el analizador sintáctico durante el procesamiento de la entrada. El uso de árboles sintácticos como representación intermedia permite que la traducción se separe del análisis sintáctico. Por ejemplo, un analizador LL y un analizador LR pueden generar el mismo árbol sintáctico como representación intermedia a pesar de que sus árboles de análisis pueden diferir notablemente debido a las transformaciones que se aplican a las gramáticas para ser analizadas de forma descendente (eliminación de recursividad izquierda y factorización).

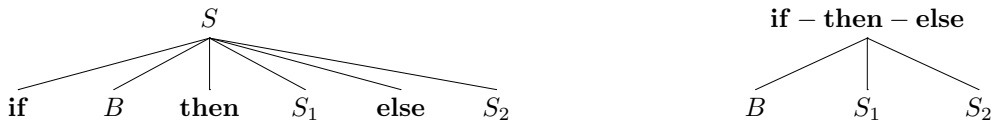
Una utilidad adicional del uso de árboles sintácticos como representación intermedia es que permiten no restringir el orden en que se consideran los nodos del árbol de análisis. Recordemos que un analizador descendente realiza un recorrido en profundidad, mientras que un analizador ascendente lo hace en postorden. Un lenguaje puede tener construcciones cuya traducción requiera un orden de tratamiento de los nodos distinto. En este caso, es necesario disponer de una estructura en forma de árbol que pueda ser inspeccionada después del análisis sintáctico en el orden que se desee.

Habitualmente el árbol sintáctico que se genera como representación intermedia es una *abstracción* del árbol de análisis formado por aplicación de las reglas de producción reconocidas por el analizador. La abstracción simplifica el árbol, eliminando información superflua para la traducción. Las simplificaciones más frecuentes son:

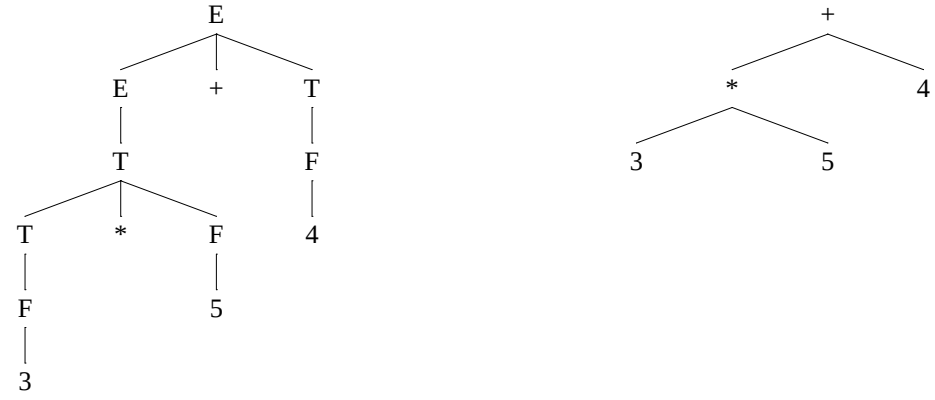
- Los operadores y palabras clave no aparecen como hojas, sino que se asocian al nodo padre de dichas hojas.
- Las producciones simples pueden eliminarse.

Una vez construido el árbol sintáctico abstracto, se puede aplicar sobre él la técnica de traducción dirigida por la sintaxis, de forma similar a como se ha visto aplicada hasta ahora sobre árboles de análisis sintáctico.

Ejemplo 4.7. Supongamos que el lenguaje analizado contiene la construcción `if-then-else`, descrita mediante la regla de producción $S \rightarrow \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$. A continuación se muestra el árbol de análisis a la izquierda y el árbol sintáctico abstracto a la derecha:



Ejemplo 4.8. Supongamos que disponemos de la gramática no ambigua para analizar expresiones aritmética. A continuación se muestra el árbol de análisis de la expresión `3*5+4` a la izquierda, y el árbol sintáctico abstracto a la derecha:



Se pueden construir los nodos de un árbol sintáctico abstracto mediante objetos (o estructuras) con campos adecuados a cada caso. Cada objeto puede tener un campo *op*, que contiene la etiqueta del nodo. Dependiendo de su posición en el árbol, los objetos tienen otros campos:

- Si el nodo es una hoja, un campo adicional almacena el valor léxico de la hoja.
- Si el nodo es interior, se requieren tantos campos adicionales como hijos tenga el nodo en el árbol sintáctico. Cada campo adicional es un puntero a cada uno de estos hijos.

La siguiente definición dirigida por la sintaxis permite la construcción de árboles sintácticos abstractos para la gramática no ambigua de las expresiones aritméticas:

PRODUCCIÓN	REGLAS SEMÁNTICAS
$E \to E_1 + T$	$E.nodo =$ **new** Nodo($'+'$, $E_1.nodo$, $T.nodo$)
$E \to E_1 - T$	$E.nodo =$ **new** Nodo($'-'$, $E_1.nodo$, $T.nodo$)
$E \to T$	$E.nodo = T.nodo$
$T \to (E)$	$T.nodo = E.nodo$
$T \to$ **id**	$T.nodo =$ **new** Hoja(**id** , **id**.$entrada$)
$T \to$ **num**	$T.nodo =$ **new** Hoja(**num** , **num**.$valor$)

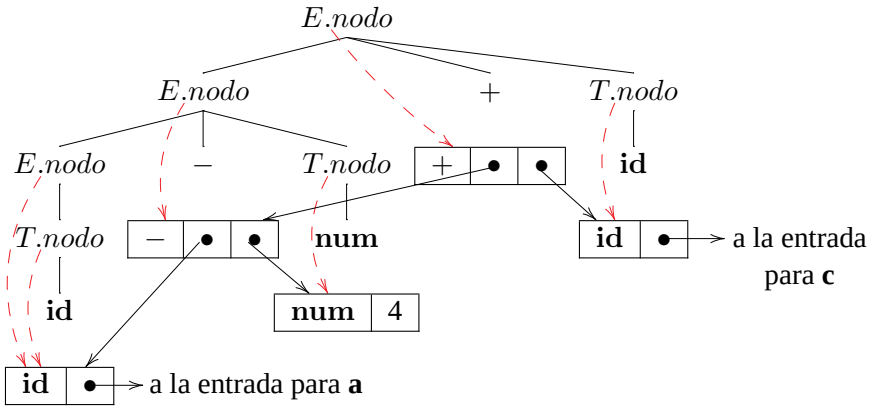
En la definición aparecen los siguientes constructores de objetos:

- Nodo(op,c_1,c_2,. . .,c_k): crea un objeto con el primer campo op y k campos para los hijos c_1, c_2, . . ., c_k.
- Hoja(op,$valor$): crea un objeto con el primer campo op (que en este caso puede ser o **id** o **num**) y un campo con el valor léxico (apuntador a la tabla de símbolos para identificadores o valor numérico para las constantes).

Los atributos usados en la definición son los siguientes:

- $nodo$: atributo sintetizado asociado a E y T que contiene un objeto devuelto por los constructores Hoja o Nodo.
- $entrada$: atributo sintetizado que almacena un apuntador a la tabla de símbolos.
- $valor$: atributo sintetizado que almacena el valor de una constante numérica.

A continuación se muestra el árbol de análisis con anotaciones para la entrada `a+4+c`:



Los atributos $nodo$ de los no terminales del árbol con anotaciones contienen referencias a los nodos del árbol sintáctico abstracto. En concreto, en el atributo $nodo$ de la raíz se almacena la referencia al nodo raíz del árbol sintáctico abstracto.

4.2. Esquemas de traducción orientados por la sintaxis

Los *esquemas de traducción orientados por la sintaxis* son una notación complementaria para las definiciones dirigidas por la sintaxis. Son complementarios en el sentido de que cualquier definición dirigida por la sintaxis puede implementarse con un esquema de traducción. Es decir, mientras las definiciones dirigidas por la sintaxis son declarativas, los esquemas de traducción son procedimentales.

Un esquema de traducción 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*. Por tanto, un esquema de traducción es como una definición dirigida por la sintaxis, con la excepción de que el orden de evaluación de las reglas semánticas se muestra explícitamente.

Se denominan *esquemas de traducción dirigidos por la sintaxis postfijos* a aquellos que tienen todas las acciones semánticas en los extremos derechos de los cuerpos de las producciones. Pueden implementarse durante el análisis sintáctico LR mediante la ejecución de las acciones cada vez que se reduce.

Existen también *esquemas de traducción dirigidos por la sintaxis con acciones en mitad de las reglas*. Se puede colocar una acción en cualquier lugar dentro de la parte derecha de una producción, ejecutándose de inmediato después de procesar todos los símbolos a su izquierda[2].

Supongamos la siguiente regla con una acción en mitad de la parte derecha:

$$B \rightarrow X \text{ \{ acción \} } Y$$

La acción se realiza después de que X sea reconocido. Por tanto:

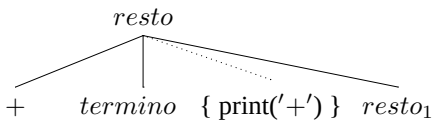
- Si el análisis sintáctico es *ascendente*, la acción se ejecuta cuando aparezca X en el tope de la pila de análisis sintáctico, independientemente de si X es un símbolo terminal o no terminal.
- Si el análisis sintáctico es *descendente*, la acción se ejecuta justo antes de expandir la ocurrencia de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

En el árbol sintáctico la acción se convierte en un hijo adicional conectado de forma especial. Por tanto, es posible implementar cualquier esquema de traducción orientado por la sintaxis, construyendo primero un árbol sintáctico y realizando después las acciones mediante un recorrido en profundidad.

Ejemplo 4.9. Supongamos un esquema de traducción con la siguiente regla:

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

La porción correspondiente a esta regla en un árbol sintáctico es:



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

No obstante, muchos traductores y compiladores ejecutan las acciones semánticas no a posteriori, tras construir un árbol sintáctico, sino durante el mismo proceso de análisis sintáctico. La razón es lograr una mayor eficiencia en la ejecución de las acciones semánticas. Existen dos clases importantes de definiciones dirigidas por la sintaxis que pueden implementarse con esquemas de traducción orientados a la sintaxis:

- La gramática subyacente es LR y la definición dirigida por la sintaxis es S-atribuida.

[2] Bison permite especificar este tipo de esquemas de traducción.

- La gramática subyacente es LL y la definición dirigida por la sintaxis es L-atribuida.

En ambos casos las reglas semánticas en la definición dirigida por la sintaxis pueden convertirse en un esquema de traducción orientado a la sintaxis con acciones que se ejecuten en el momento adecuado durante el análisis sintáctico. En concreto, 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.

Es importante tener en cuenta que no todos los esquemas de traducción orientados a la sintaxis pueden implementarse durante el análisis sintáctico.

En la sección 4.1.3.1 se trataron las definiciones simples dirigidas por la sintaxis, con las que se puede especificar una traducción que genere una cadena de caracteres como salida. En general, las definiciones simples dirigidas por la sintaxis se pueden implementar con esquemas de traducción en los que las acciones impriman las cadenas adicionales en el orden en que aparecen en la definición. Además, para implementar un esquema de traducción simple, se pueden ejecutar las acciones semánticas durante el análisis sintáctico, sin ser necesario construir el árbol de análisis sintáctico previamente.

Ejemplo 4.10. El siguiente esquema de traducción dirigida por la sintaxis permite traducir de notación infija a postija:

$$
\begin{array}{ll}
expr \rightarrow expr_1 \ + \ term & \{ \text{ print}('+'); \ \} \\
expr \rightarrow expr_1 \ - \ term & \{ \text{ print}('-'); \ \} \\
expr \rightarrow term & \\
expr \rightarrow 0 & \{ \text{ print}('0'); \ \} \\
expr \rightarrow 1 & \{ \text{ print}('1'); \ \} \\
\ldots & \ldots \\
expr \rightarrow 9 & \{ \text{ print}('9'); \ \}
\end{array}
$$

4.3. Tabla de símbolos y comprobaciones estáticas

En esta sección se estudian las comprobaciones estáticas llevadas a cabo por un compilador durante la fase de *análisis semántico*. Quedan excluidas de este estudio las comprobaciones dinámicas, es decir, aquellas que se realizan durante la ejecución del programa objeto.

Se denomina *comprobación estática* a cualquier verificación realizada por el compilador para garantizar que el programa fuente cumpla las especificaciones sintácticas y semánticas del lenguaje de entrada. Las comprobaciones estáticas no sólo aseguran que un programa pueda compilarse con éxito, sino que también tienen el potencial de atrapar errores de programación de forma anticipada, antes de ejecutar el programa. Las comprobaciones estáticas incluyen:

- *Comprobaciones sintácticas*. Generalmente, para comprobar la sintaxis de un lenguaje de programación, se usa una gramática libre de contexto, si bien la mayoría de estos lenguajes son generados por gramáticas dependientes del contexto. Esas restricciones sensibles al contexto deben también analizarse. Este grupo engloba, por ejemplo, las comprobaciones de unicidad como la declaración de identificadores dentro de un alcance, de flujo de control, como el que una instrucción `break` vaya dentro de un ciclo o `switch`, o las comprobaciones de la validez de las expresiones a ambos lados de las asignaciones, denominadas l-value y r-value.
- *Comprobaciones de tipos*. Las reglas sobre los tipos de un lenguaje verifican que un operador o función se aplique al número y tipo de operandos correctos. Por ejemplo, se comprueba que sólo se quite la referencia a un apuntador, que sólo se utilicen índices con las matrices, que el número y tipo de parámetros en una función sean correctos, etc. También se incluye en este grupo la realización de conversiones de tipos si son necesarias, por ejemplo, al sumar un entero con un punto flotante, insertando un operador en el árbol de análisis sintáctico. Estas comprobaciones pueden, por tanto, reunir información para la generación de código (sobrecarga de operadores, conversión de tipos y polimorfismo).

A veces se combina la comprobación estática y la generación de código intermedio con el análisis sintáctico (por ejemplo, en 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 (por ejemplo, en compiladores de Ada). La siguiente figura muestra esta segunda alternativa:

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 se encargarán de recolectar la información de forma incremental, mientras que las fases de síntesis la usan para generar código destino.

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 $factor \rightarrow \mathbf{id}$ obtiene información acerca de ese identificador consultando la tabla de símbolos.

Es usual que las tablas de símbolos deban 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.

4.4. Tipos y declaraciones

Las aplicaciones de los tipos incluyen:

- *Comprobación de tipos*. Se emplean reglas lógicas para razonar acerca del comportamiento de un programa en tiempo de ejecución. Más concretamente, se verifica 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*. 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.

4.4.1. 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.

Los conjuntos de tipos básicos y los constructores dependen del lenguaje que se va a comprobar. A continuación se definen las expresiones de tipo:

1. Un tipo básico es una expresión de tipo. Los tipos básicos suelen incluir `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:
   a) 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.
   b) Productos: si T_1 y T_2 son expresiones de tipos, entonces su producto cartesiano $T_1 \times T_2$ es una expresión de tipo.
   c) Registros: son constructores de tipo similares a 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.
   d) 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.
   e) 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.

Como representaciones de expresiones de tipos se pueden emplear grafos, árboles o grafos dirigidos acíclicos (cuyos nodos interiores son constructores de tipo y cuyas hojas son tipos básicos, nombres de tipo y variables de tipo). Veremos estos últimos en el próximo tema.

4.4.2. 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`

Por tanto, se necesita una definición precisa de *equivalencia* de expresiones de tipos. Los compiladores usan *representaciones* que permitan determinar rápidamente la equivalencia de tipos.

4.4.2.1. 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 si, y sólo si, son idénticas.

Ejemplo 4.11. A continuación se muestra un algoritmo para comprobar la equivalencia estructural de dos expresiones de tipos s y t:

```
1 function equiv_estr(s,t): boolean;
2 {
3       if (s y t son el mismo tipo basico) then
4             return true;
5       else if ( (s == array(s1,s2)) and (t == array(t1,t2) ) then
6             return ( equiv_estr(s1,t1) and equiv_estr(s2,t2) );
```

```
7       else if ( (s == s1 × s2) and (t == t1 × t2) then
8             return ( equiv_estr(s1,t1) and equiv_estr(s2,t2) );
9       else if ( (s == pointer(s1)) and (t == pointer(t1)) ) then
10            return equiv_estr(s1,t1);
11      else if ( (s == s1 → s2) and (t == t1 → t2) ) then
12            return ( equiv_estr(s1,t1) and equiv_estr(s2,t2 ) );
13      else return false;
14  }
```

Este algoritmo sólo tiene en cuenta cuatro constructores y no comprueba la existencia de ciclos.

4.4.2.2. Equivalencia de nombres

En aquellos lenguajes en los que se puede dar nombre a los tipos, se permitirá que las expresiones de tipos adquieran también un nombre y que estos puedan aparecer también en otras expresiones de tipos. En estos casos, con equivalencia estructural se sustituirán los nombres por las expresiones a las que representan y se dirá que dos expresiones de tipos son estructuralmente equivalentes si representan expresiones estructuralmente equivalentes cuando todos los nombres han sido sustituidos.

Sin embargo, la *equivalencia de nombres* 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.

Ejemplo 4.12. 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.

Supongamos el siguiente código Pascal:

```
1  type enlace = ^nodo;
2  var siguiente : enlace;
3  var ultimo : enlace;
4  var p : ^nodo;
5  var q,r : ^nodo;
```

¿Tienen tipos equivalentes `siguiente`, `ultimo`, `p`, `q` y `r`?

- De acuerdo con la *equivalencia de nombres*:
  - `siguiente` y `ultimo` tienen el mismo tipo.
  - `p`, `q` y `r` tienen el mismo tipo.
  - `siguiente` y `p` no tienen el mismo tipo.
- De acuerdo con la *equivalencia estructural*:
  - Todos los identificadores tienen el mismo tipo.

La mayoría de las implementaciones de Pascal emplean la equivalencia de nombres.

4.4.3. Ciclos en la representación de tipos

Una vez que se define una clase, su nombre se puede usar como el nombre de un tipo en C++ o Java; por ejemplo:

```
1  public class Nodo { ... }
2  ...
3  public Nodo n;
```

Pueden usarse nombres para definir tipos recursivos, necesarios para estructuras de datos como las listas enlazadas y los árboles. El pseudocódigo para el elemento de una lista:

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

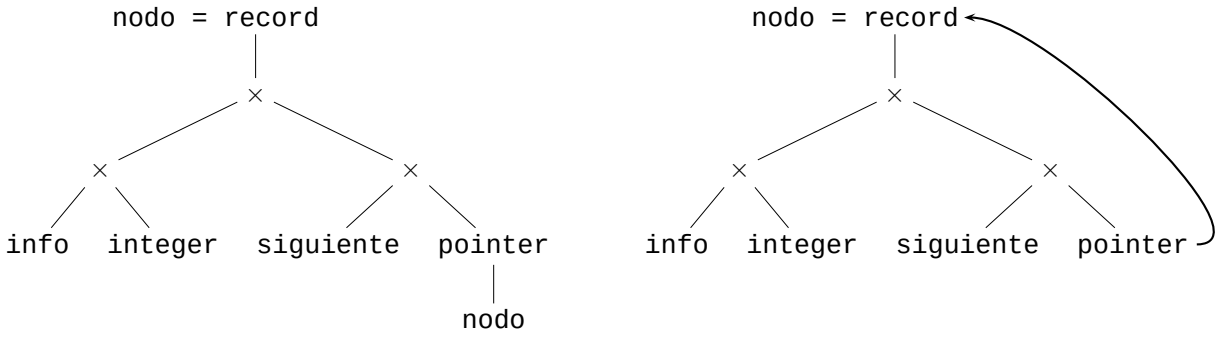
define el tipo recursivo `Celda` como una clase que contiene un campo `info` y un campo `siguiente` de tipo `Celda`.

En lenguajes como C y Pascal, no orientados a objetos, se implementan tipos de datos recursivos con registros que contienen apuntadores a registros similares, y los nombres de los tipos desempeñan un papel fundamental en la definición de tipos de dichos registros.

Ejemplo 4.13. A continuación se muestra una declaración en Pascal, un lenguaje que usa equivalencia de nombres:

```
type enlace = ^nodo;
type nodo = record
          info: integer;
          siguiente: enlace;
         end;
```

Se pueden usar las siguientes representaciones del tipo `nodo`, la primera acíclica y la segunda cíclica:



Ejemplo 4.14. El lenguaje C emplea equivalencia estructural salvo en los registros. En el siguiente ejemplo se declara una estructura en C similar a la declarada en Pascal en el ejemplo anterior:

```
struct nodo {
   int info;
   struct nodo *siguiente;
};
```

C evita los ciclos en los grafos usando la equivalencia estructural para todos los tipos, excepto los registros. C exige que los nombres de tipos se declaren antes de su uso, pero permite apuntadores a tipos de registros no declarados. Detiene la búsqueda de equivalencia estructural cuando encuentra un constructor de registros. Aquí aplica la equivalencia de nombre.

4.5. Comprobación de tipos

Un *sistema de tipos* está constituido por una serie de reglas que asignan expresiones de tipos a las distintas partes de un programa.

Un *comprobador de tipos* implementa un sistema de tipos y se basa en la información sobre las construcciones sintácticas del lenguaje, la noción de tipos y las reglas para asignar tipos a las construcciones del lenguaje. Por ejemplo:

- La documentación de Pascal indica: “Si los dos operandos de los operadores aritméticos de suma, sustracción y multiplicación son de tipo entero, entonces el resultado es de tipo entero”.
- La documentación de C indica: “El resultado del operador unario & es un apuntador hacia el objeto al que se refiere el operando. Si el tipo del operando es x, el tipo del resultado es un apuntador a x”.

Esta noción de tipos varía de unos lenguajes a otros en función de:

- si la comprobación de tipos se realiza de forma estática o dinámica.
- si se usa la equivalencia estructural o de nombre.
- si los tipos son *fuertes* o *débiles*.

Los lenguajes con tipos fuertes no permiten que se lleve a cabo una operación con argumentos con el tipo equivocado, de manera que el intento de hacerlo provocaría un error. El compilador garantiza, por tanto, que los programas se ejecutarán sin errores. 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 bien permiten que un valor de un tipo sea tratado como de otro, por ejemplo una cadena como un número. Esto puede ser útil, pero también puede conducir a errores. Ejemplos C y C++. A veces se les denomina lenguajes con *tipos inseguros*.

Cualquier comprobación puede realizarse dinámicamente, verificando en el código objeto la compatibilidad del tipo de un elemento con el valor que se le asigna. En la práctica, hay comprobaciones que tienen que hacerse dinámicamente.

En cualquier caso, un comprobador de tipos debe hacer *recuperación de errores*.

Diferentes compiladores o procesadores del mismo lenguaje podrían utilizar diferentes sistemas de tipos.

4.5.1. 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 (variables, funciones, tipos no básicos) antes de utilizarlos. Por 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 tiene la siguiente forma:

`if` f tiene el tipo $s \to t$ y x tiene el tipo s,
`then` la expresión $f(x)$ tiene el tipo t.

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`)`.

Por otro lado, la *inferencia de tipos* determina el tipo de una construcción del lenguaje a partir de la forma en que se utiliza. La inferencia de tipos es necesaria para lenguajes como ML, que comprueban los tipos pero no requieren la declaración de nombres.

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 se sepa el tipo de los elementos que contiene. Las variables que representan expresiones de tipos (α,β,...) permiten manejar tipos desconocidos.

Una regla común para la inferencia de tipos tiene la siguiente forma:

`if` $f(x)$ es una expresión,
`then` para ciertas α y β, f tiene el tipo $\alpha \to \beta$ `and` x tiene el tipo α.

Combinando varias reglas se puede deducir los tipos de las variables α, β, etc.

4.5.2. Conversiones de tipos

La definición de muchos lenguajes especifica conversiones automáticas de tipos, como la asignación de enteros a reales o viceversa (se convierte al tipo del lado izquierdo de la asignación), al emplear expresiones se suele convertir el entero en real, etc.

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

Ejemplo 4.15. Supongamos que manejamos la expresión $x + i$, dónde x es de tipo punto flotante e i de tipo entero. Necesitaremos que, en la representación intermedia, se inserte el operador de conversión:

```
1 t1 = (float)i;
2 t2 = x+t1;
```

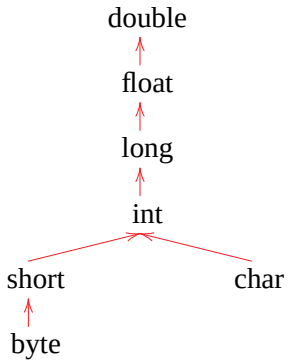
Si sólo tuviéramos estos dos tipos posibles, podría usarse el siguiente esquema:

`if` ($E_1.tipo$ == `integer and` $E_2.tipo$ == `integer`) $E.tipo$ = `integer`
`else if` ($E_1.tipo$ == `integer and` $E_2.tipo$ == `float`) $E.tipo$ = `float`
`else if` ($E_1.tipo$ == `float and` $E_2.tipo$ == `integer`) $E.tipo$ = `float`
...

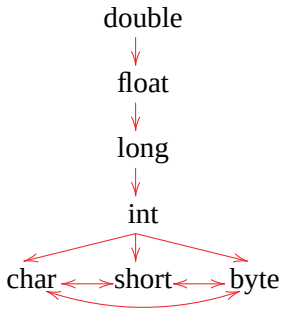
A medida que se incrementa el número de tipos sujetos a conversión, el número de casos aumenta con rapidez. Por tanto, con un número extenso de tipos, es importante una organización cuidadosa de las acciones semánticas.

Las reglas de conversión de tipos varían de un lenguaje a otro. Las reglas de Java, por ejemplo, distinguen entre *conversiones de ampliación*, que pretenden preservar la información, y las *conversiones de reducción*, en las que puede perderse información. A continuación se muestran los grafos para ambos casos:

Conversión de ampliación:



Conversión de reducción:



Las conversiones de tipo acompañan generalmente a la *sobrecarga de operadores* (un mismo operador tiene significados distintos según contexto en el que se use).

En el caso de la conversión explícita o *cast*, el programador debe escribir algo para motivarla. Para el comprobador de tipos es igual que una aplicación de función. Por ejemplo, en Pascal se emplea `ord` para transformar un carácter a entero, y `chr` para hacer la operación contraria.

La conversión implícita o *coerción* la realiza el compilador de forma automática. Suelen limitarse a conversiones de ampliación (entero a real) porque, en la práctica, las de reducción pueden provocar pérdida de información. Por ejemplo, una conversión implícita en C de caracteres ASCII a enteros entre 0 y 127 en expresiones aritméticas. La conversión implícita de constantes suele realizarse durante la compilación para mejorar el tiempo de ejecución del programa.

Ejemplo 4.16. Si se emplea un compilador de Pascal con las siguientes sentencias, donde X es una matriz de reales:

```
1 for I := 1 to N do X[I] := 1
2 for I := 1 to N do X[I] := 1.0
```

la ejecución del primer `for` es más lenta si el compilador no hace la conversión implícita de 1 a 1.0 en tiempo de compilación. Esto sucede porque el código generado para la asignación `X[I] := 1` incluye una llamada a la función de conversión a float de la constante 1, llamada que se debe ejecutar en cada iteración del bucle.