Análisis Semántico en Procesadores de Lenguaje

Cuaderno Nº 38

Francisco Ortín Soler
Juan Manuel Cueva Lovelle
Maria Cándida Luengo Díez
Aquilino Adolfo Juan Fuente
José Emilio Labra Gayo
Raúl Izquierdo Castanedo

Lenguajes y Sistemas Informáticos
Departamento de Informática
Universidad de Oviedo

Oviedo, Marzo 2004

Cuaderno Nº 38

ANÁLISIS SEMÁNTICO EN PROCESADORES DE LENGUAJE

Autorers:

Francisco Ortín Soler
Juan Manuel Cueva Lovelle
Maria Cándida Luengo Díez
Aquilino Adolfo Juan Fuente
José Emilio Labra Gayo
Raúl Izquierdo Castanedo

Universidad de Oviedo - España

Editorial:

SERVITEC

ISBN: 84-688-6208-8
Deposito Legal: AS-1358-04

PRÓLOGO

El objetivo de este libro es introducir los conceptos necesarios sobre la fase de análisis semántico en procesadores de lenguaje, para un curso universitario de traductores, compiladores e intérpretes: procesadores de lenguajes de programación.

Está principalmente dirigido a alumnos de cuarto curso de Ingeniería Informática, aunque cualquier persona con conocimientos básicos de teoría de lenguajes y gramáticas, así como el conocimiento de algún lenguaje de programación orientado a objetos –como Java o C++– está capacitado para seguir su contenido.

Para facilitar la labor docente del mismo, los conceptos introducidos han sido ilustrados con un conjunto importante de ejemplos. Asimismo, al final del libro se ha añadido un capítulo de cuestiones de revisión y otro de ejemplos propuestos. El objetivo principal de estos dos puntos es fijar los conocimientos adquiridos y enfatizar los puntos más importantes.

El libro se compone de los siguientes puntos:

- Inicialmente se definen los conceptos básicos a emplear a lo largo de todo el texto.
- El primer punto es una introducción somera a la especificación de la semántica de lenguajes de programación. Aunque la principal tarea de este texto es centrarnos en el análisis semántico de lenguajes y no en su semántica, introduciremos este concepto por la relación que posee con las gramáticas atribuidas.
- El segundo capítulo es el que muestra el contexto del análisis semántico dentro del marco de los procesadores de lenguaje. Detalla los objetivos principales de éste, así como la interacción de esta fase con el resto.
- El capítulo 3 introduce el mecanismo más empleado a la hora de definir analizadores semánticos de procesadores de lenguajes: las gramáticas atribuidas (definiciones dirigida por sintaxis).
- El siguiente capítulo profundiza en las características más importantes de las gramáticas atribuidas, empleadas para la implementación de un evaluador.
- El punto cuarto de este libro muestra cómo pueden evaluarse las gramáticas atribuidas. Se basa en los conceptos y clasificaciones expuestas en el capítulo anterior, ahondando en cómo, en función del tipo de gramática atribuida, podremos implementar ésta, empleando distintas técnicas.
- El capítulo 6 detalla la parte principal de prácticamente la mayoría de los analizadores semánticos: la comprobación de tipos. Define los conceptos necesarios e indica los objetivos y problemas que deberá solventar un procesador de lenguaje.
- Una vez concluidos los capítulos, cuestiones y ejercicios propuestos, se presenta un conjunto de apéndices en los que se detalla el código fuente empleado en los

ejemplos de implementación, presentados a lo largo del texto. Éstos también podrán ser descargados de la URL mostrada al final de este prólogo.

Finalmente se indica la lista de referencias bibliográficas principales empleadas para escribir este texto. Podrán servir al lector como un mecanismo para ampliar los contenidos aquí introducidos.

Por la amplia bibliografía existente en el idioma inglés –además de lo presente en Internet– se ha considerado oportuno hacer empleo de notas al pie de página para indicar, entre otras cosas, la traducción de los términos principales.

Para concluir este prólogo, el código fuente empleado en el texto se encuentra en mi página personal, así como mi dirección de correo electrónico para posibles mejoras o comentarios:

http://www.di.uniovi.es/~ortin

CONTENIDO

ANÁLISIS SEMÁNTICO EN PROCESADORES DE LENGUAJE

La fase de análisis semántico de un procesador de lenguaje es aquélla que computa la información adicional necesaria para el procesamiento de un lenguaje, una vez que la estructura sintáctica de un programa haya sido obtenida. Es por tanto la fase posterior a la de análisis sintáctico y la última dentro del proceso de síntesis de un lenguaje de programación [Aho90].

Sintaxis de un lenguaje de programación es el conjunto de reglas formales que especifican la estructura de los programas pertenecientes a dicho lenguaje. Semántica de un lenguaje de programación es el conjunto de reglas que especifican el significado de cualquier sentencia sintácticamente válida. Finalmente, el análisis semántico[1] de un procesador de lenguaje es la fase encargada de detectar la validez semántica de las sentencias aceptadas por el analizador sintáctico.

Ejemplo 1:Dado el siguiente ejemplo de código en C:

```
superficie = base * altura / 2;
```

La sintaxis del lenguaje C indica que las expresiones se pueden formar con un conjunto de operadores y un conjunto de elementos básicos. Entre los operadores, con sintaxis binaria infija, se encuentran la asignación, el producto y la división. Entre los elementos básicos de una expresión existen los identificadores y las constantes enteras sin signo (entre otros).

Su semántica identifica que en el registro asociado al identificador superficie se le va a asociar el valor resultante del producto de los valores asociados a `base` y `altura`, divididos por dos (la superficie de un triángulo).

Finalmente, el análisis semántico del procesador de lenguaje, tras haber analizado correctamente que la sintaxis es válida, deberá comprobar que se satisfacen las siguientes condiciones:

- Que todos los identificadores que aparecen en la expresión hayan sido declarados en el ámbito actual, o en alguno de sus ámbitos (bloques[2]) previos.
- Que la subexpresión de la izquierda sea semánticamente válida, es decir, que sea un *lvalue*[3].
- Que a los tipos de los identificadores `base` y `altura` se les pueda aplicar el operador de multiplicación. Un registro en C, por ejemplo, no sería válido.

---

[1] *Semantic Analysis* o *Contextual Analysis.*

[2] El lenguaje C es un lenguaje orientado a bloques. Los bloques se especifican mediante la pareja de caracteres `{` y `}`. Dentro de un bloque, es posible declarar variables que ocultan a las variables declaradas en bloques de un nivel menor de anidamiento.

[3] Una expresión es un *lvalue* (*left value*, valor a la izquierda) si puede estar a la izquierda en una expresión de asignación, es decir, si se puede obtener su dirección de memoria y modifica el contenido de ésta. Otros ejemplos de expresiones lvalue en C, son: `*puntero`, `array[n]`, `registro.campo`...

- Deberá inferirse el tipo resultante de la multiplicación anterior. Al tipo inferido se le deberá poder aplicar el operador de dividir, con el tipo entero como multiplicando.
- Deberá inferirse el tipo resultante de la división y comprobarse si éste es compatible con el tipo de `superficie` para llevar a cabo la asignación. Como ejemplo, si `superficie` fuese entera y `division` real, no podría llevarse a cabo la asignación.

□

La fase de análisis semántico obtiene su nombre por requerir información relativa al significado del lenguaje, que está fuera del alcance de la representatividad de las gramáticas libres de contexto y los principales algoritmos existentes de análisis; es por ello por lo que se dice que captura la parte de la fase de análisis considerada fuera del ámbito de la sintaxis. Dentro del la clasificación jerárquica que Chomsky dio de los lenguajes [Hopcroft02, Cueva03], la utilización de gramáticas sensibles al contexto (o de tipo 1) permitirían identificar sintácticamente características como que la utilización de una variable en el lenguaje Pascal ha de estar previamente declarada. Sin embargo, la implementación de un analizador sintáctico basado en una gramática de estas características sería computacionalmente más compleja que un autómata de pila [Louden97].

Así, la mayoría de los compiladores utilizan una gramática libre de contexto para describir la sintaxis del lenguaje y una fase de análisis semántico posterior para restringir las sentencias que "semánticamente" no pertenecen al lenguaje. En el caso que mencionábamos del empleo de una variable en Pascal que necesariamente haya tenido que ser declarada, el analizador sintáctico se limita a comprobar, mediante una gramática libre de contexto, que un identificador forma parte de una expresión. Una vez comprobado que la sentencia es sintácticamente correcta, el analizador semántico deberá verificar que el identificador empleado como parte de una expresión haya sido declarado previamente. Para llevar a cabo esta tarea, es típica la utilización de una estructura de datos adicional denominada **tabla de símbolos**. Ésta poseerá una entrada por cada identificador declarado en el contexto que se esté analizando. Con este tipo de estructuras de datos adicionales, los desarrolladores de compiladores acostumbran a suplir las carencias de las gramáticas libres de contexto.

Otro caso que se da en la implementación real de compiladores es ubicar determinadas comprobaciones en el analizador semántico, aun cuando puedan ser llevadas a cabo por el analizador sintáctico. Es factible describir una gramática libre de contexto capaz de representar que toda implementación de una función tenga al menos una sentencia `return`. Sin embargo, la gramática sería realmente compleja y su tratamiento en la fase de análisis sintáctico sería demasiado complicada. Así, es más sencillo transferir dicha responsabilidad al analizador semántico que sólo deberá contabilizar el número de sentencias `return` aparecidas en la implementación de una función.

El objetivo principal del analizador semántico de un procesador de lenguaje es asegurarse de que el programa analizado satisfaga las reglas requeridas por la especificación del lenguaje, para garantizar su correcta ejecución. El tipo y dimensión de análisis semántico requerido varía enormemente de un lenguaje a otro. En lenguajes interpretados como Lisp o Smalltalk casi no se lleva a cabo análisis semántico previo a su ejecución, mientras que en lenguajes como Ada, el analizador semántico deberá comprobar numerosas reglas que un programa fuente está obligado a satisfacer.

Vemos, pues, cómo el análisis semántico de un procesador de lenguaje no modela la semántica o comportamiento de los distintos programas construidos en el lenguaje de programación, sino que, haciendo uso de información parcial de su comportamiento, realiza todas las comprobaciones necesarias –no llevadas a cabo por el analizador sintáctico–

para asegurarse de que el programa pertenece al lenguaje. Otra fase del compilador donde se hace uso parcial de la semántica del lenguaje es en la optimización de código, en la que analizando el significado de los programas previamente a su ejecución, se pueden llevar a cabo transformaciones en los mismos para ganar en eficiencia.

1 Especificación Semántica de Lenguajes de Programación

Existen dos formas de describir la semántica de un lenguaje de programación: mediante especificación informal o natural y formal.

La descripción informal de un lenguaje de programación es llevada a cabo mediante el lenguaje natural. Esto hace que la especificación sea inteligible (en principio) para cualquier persona. La experiencia nos dice que es una tarea muy compleja, si no imposible, el describir todas las características de un lenguaje de programación de un modo preciso. Como caso particular, véase la especificación del lenguaje ISO/ANSI C++ [ANSIC++].

La descripción formal de la semántica de lenguajes de programación es la descripción rigurosa del significado o comportamiento de programas, lenguajes de programación, máquinas abstractas o incluso cualquier dispositivo hardware. La necesidad de hacer especificaciones formales de semántica surge para [Nielson92, Watt96, Labra03]:

- Revelar posibles ambigüedades existentes implementaciones de procesadores de lenguajes o en documentos descriptivos de lenguajes de programación.
- Ser utilizados como base para la implementación de procesadores de lenguaje.
- Verificar propiedades de programas en relación con pruebas de corrección o información relacionada con su ejecución.
- Diseñar nuevos lenguajes de programación, permitiendo registrar decisiones sobre construcciones particulares del lenguaje, así como permitir descubrir posibles irregularidades u omisiones.
- Facilitar la comprensión de los lenguajes por parte del programador y como mecanismo de comunicación entre diseñador del lenguaje, implementador y programador. La especificación semántica de un lenguaje, como documento de referencia, aclara el comportamiento del lenguaje y sus diversas construcciones.
- Estandarizar lenguajes mediante la publicación de su semántica de un modo no ambiguo. Los programas deben poder procesarse en otra implementación de procesador del mismo lenguaje exhibiendo el mismo comportamiento.

1.1. Especificación Formal de Semántica

Si bien la especificación formal de la sintaxis de un lenguaje se suele llevar a cabo mediante la descripción estándar de su gramática en notación BNF (*Backus-Naur Form*), en el caso de la especificación semántica la situación no está tan clara; no hay ningún método estándar globalmente extendido.

El comportamiento de las distintas construcciones de un lenguaje de programación, puede ser descrito desde distintos puntos de vista. Una clasificación de los principales

métodos formales de descripción semántica, así como una descripción muy breve de las ideas en las que se fundamentan, es [Nielson92, Labra01]:

- **Semántica operacional**[4]: El significado de cada construcción sintáctica es especificado mediante la computación que se lleva a cabo en su ejecución sobre una máquina abstracta. Lo que realmente se especifica es *cómo* se lleva a cabo dicha ejecución. Los significados del programa son descritos en términos de operaciones, utilizando un lenguaje basado en reglas de inferencia lógicas en las que se describen formalmente las secuencias de ejecución de las diferentes instrucciones sobre una máquina abstracta [Nielson92]. Es muy cercano a la implementación y se puede emplear para construir prototipos de procesadores de lenguajes como la descripción de PL/I en VDL [Lucas69].

Ejemplo 2. Lo siguiente es la especificación formal de la semántica de una asignación en un lenguaje de programación:

$$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

Lo que se encuentra en la parte superior es una "premisa" y en la parte inferior una "conclusión". La premisa indica que el resultado de evaluar una expresión e en un determinado almacén σ (estado de una máquina abstracta) produce un valor v. La conclusión indica que, dado un estado σ, la asignación de una expresión e a un identificador x produce un nuevo estado resultado de añadir a σ la asociación del valor de v al identificador x.

□

- **Semántica denotacional**[5]. La representación del comportamiento de cada sentencia o frase del lenguaje se lleva a cabo mediante entidades matemáticas (denotación) que representan el efecto de haber ejecutado las sentencia o frase asociada [Watt96]. Por tanto, se hace más hincapié en el *efecto* de la computación que en cómo se lleva a cabo. Se utiliza mayoritariamente en diseño de lenguajes de programación y se ha empleado para especificar la semántica completa de lenguajes como Ada, Algol-60 y Pascal [Bjorner82]

Ejemplo 3. La especificación de una asignación en semántica denotacional es:

$$[\![v := e]\!]_S(s) = s \oplus \{v \mapsto [\![e]\!]_E(s)\}$$

Los doble corchetes con un subíndice indican una función semántica. En nuestro ejemplo, se está definiendo parcialmente la que posee el subíndice S (sentencia[6]); la que posee el subíndice E (evaluación de una expresión) está definida en otra parte. Así, la especificación que tenemos arriba denota la semántica de la sentencia de asignación de cualquier identificador v a una expresión e. Ésta es aplicada a un estado s y devuelve el mismo estado ampliado con la asociación a v del resultado de evaluar la expresión e.

□

- **Semántica axiomática**[7]. Especifica las propiedades del efecto de ejecutar las sentencias sintácticamente correctas, expresadas mediante asertos, desoyendo así los aspectos de su ejecución. El sistema permite estudiar formalmente las propiedades del lenguaje y se requiere la utilización de sistemas consistentes y

[4] *Operational semantics.*
[5] *Denotational semantics*, inicialmente denominada *mathematical semantics.*
[6] La S viene de *statement* (sentencia).
[7] *Axiomatic semantics*.

completos [Hoare73]. Se utiliza mayoritariamente en verificación formal de corrección de programas.

Ejemplo 4. La siguiente especificación de la semántica de una sentencia de asignación ha sido descrita utilizando la semántica axiomática:

$$\{P[x \to e]\}x := e\{P\}$$

En la semántica axiomática, se antepone al fragmento sintáctico (asignación) la precondición que indica la estado que se satisface previamente a la asignación. Este estado, llamado genéricamente `P`, indica que todas las ocurrencias de *x* están textualmente sustituidas por la expresión `e`. Una vez llevada a cabo la asignación, la postcondición nos indica el estado que se satisface tras su evaluación. Ejemplos de satisfacción de predicados son:

$$\{2 = 2\}x := 2\{x = 2\}$$
$$\{n + 1 = 2\}x := n + 1\{x = 2\}$$
$$\{y \times 2 + 1 > 10\}x = y * 2 + 1\{x > 10\}$$

Las especificaciones axiomáticas también se denominan "tripletas de Hoare" en honor a su creador. Como se muestra en el ejemplo, la derivación de estas tripletas es llevada a cabo de la postcondición hacia la precondición siguiendo un razonamiento hacia atrás.

□

- **Semántica algebraica**[8]. Se basa en la especificación de tipos de datos abstractos mediante una colección de operaciones (incluyendo alguna constante). Puesto que un conjunto de valores al que se le añaden una colección de operaciones constituye un álgebra, este método de descripción formal de semántica se denomina semántica algebraica [Meinke92]. Este método está pues enfocado a especificar la semántica de los tipos y sus operaciones. La semántica algebraica constituye también la base de la semántica de acciones, empleada para especificar la semántica de lenguajes de programación al completo.

Ejemplo 5. La especificación del tipo lógico (booleano) en un lenguaje de programación puede llevarse a cabo del siguiente modo, siguiendo la semántica algebraica:

```
specification Truth-Values
        sort Truth-Value
        operations
        true   : Truth-Value
        false  : Truth-Value
        not_   : Truth-Value → Truth-Value
        _∧_    : Truth-Value, Truth-Value → Truth-Value
        _∨_    : Truth-Value, Truth-Value → Truth-Value
        variables t, u: Truth-Value
        equtations
        not true = false
        not false= true
        t ∧ true = t
        t ∧ false= false
        t ∧ u   = u ∧ t
        t ∨ true = true
        t ∨ false= t
        t ∨ u   = u ∨ t
end specification
```

□

[8] *Algegraic semantics*.

- **Semántica de acciones**[9]. Fue elaborado por Peter Mosses [Mosses91] para describir la semántica de lenguajes de un modo más inteligible. Las especificaciones semánticas de lenguajes siempre han sido consideradas como oscuras, complicadas y únicamente legibles por expertos, adquiriendo así una mala reputación por su uso intensivo de símbolos matemáticos [Watt96]. De este modo, esta semántica está basada en el concepto de *acciones* que reflejan las operaciones comunes en los lenguajes de programación, ofreciendo primitivas para la asignación y declaración de identificadores, así como la combinación de instrucciones mediante control de flujo secuencial, condicional e iterativo.

Otro modo de especificar formalmente lenguajes de programación es mediante el uso de **gramáticas atribuidas**[10]. Las gramáticas atribuidas asignan propiedades (atributos) a las distintas construcciones sintácticas del lenguaje. Estos atributos pueden describir información semántica para implementar un analizador semántico (como por ejemplo el tipo de una expresión), pero pueden emplearse también para representar cualquier otra propiedad como la evaluación de una expresión o incluso su traducción a una determinada plataforma. Al no estar directamente ligadas al comportamiento dinámico (en ejecución) de los programas, no suelen clasificarse como otro tipo de especificación formal de semántica de lenguajes. Sin embargo, su uso tan versátil hace que estén, de un modo directo o indirecto, en cualquier implementación de un procesador de lenguaje.

[9] *Action semantics*.
[10] *Attribute grammar*. Posiblemente, una mejor traducción podría ser "gramáticas con atributos", pero la amplia extensión del término "gramática atribuida" hace que utilicemos la segunda traducción.

2 Tareas y Objetivos del Análisis Semántico

Como comentábamos al comienzo de este libro, el análisis semántico[11] de un procesador de lenguaje es la fase encargada de detectar la validez semántica de las sentencias aceptadas por el analizador sintáctico. También comentábamos cómo, de un modo convencional y menos formal, se suele afirmar que la sintaxis de un lenguaje de programación es aquella parte del lenguaje que puede ser descrita mediante una gramática libre de contexto, mientras que el análisis semántico es la parte de su especificación que no puede ser descrita por la sintaxis [Scott00].

En el diagrama de fases de un compilador [Aho90] podemos destacar, a raíz de las dos definiciones previas, una mayor interconexión entre la fase de análisis semántico y las siguientes fases de un compilador:

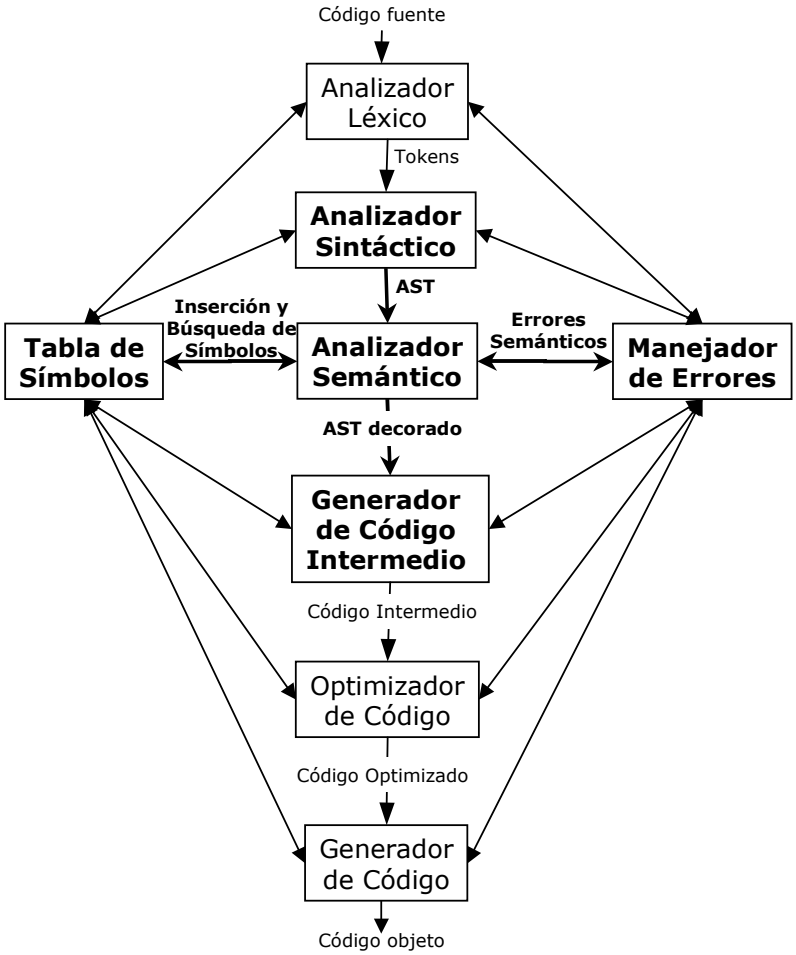

Figura 1: Fases de un compilador, destacando la interacción con el análisis semántico.

[11] *Semantic Analysis* o *Contextual Analysis.*

- Análisis sintáctico. Como se muestra en la Figura 1, la entrada del analizador semántico es la salida generada por el analizador sintáctico. La estructura empleada para intercambiar la información entre estas dos fases es lo que se conoce como árbol sintáctico –o una simplificación del mismo, denominada árbol sintáctico abstracto (§ 2.2). Una vez validada la sintaxis de un programa, el análisis semántico aplicará reglas semánticas para validar dicho árbol.
- Manejador de errores. Si la validación del árbol sintáctico descrita en el párrafo anterior no fuese satisfactoria, es decir, existiese un error semántico, la fase de análisis semántico debería notificar dicho error al manejador de errores para que éste se encargase de su gestión. El proceso de análisis podría seguir ejecutándose o no, en función de si el procesador de lenguaje implementa algún mecanismo de recuperación de errores [Louden97].
- Generación de código (intermedio). La salida del análisis semántico se suele emplear como entrada para la generación de código[12]. La estructura de datos empleada para intercambiar información entre las dos fases mencionadas es un árbol sintáctico *decorado* (§ 2.2). Este árbol posee información adicional al árbol generado por el analizador sintáctico, como por ejemplo la información relativa al tipo de cada una de las expresiones del programa. El empleo de dicha información es útil para llevar a cabo el proceso de generación de código (a bajo nivel, el tipo de una expresión es necesario, por ejemplo, para saber el número de bytes que ocupa su valor).
- Tabla de símbolos. Como hemos mencionado previamente, la utilización de gramáticas libres de contexto (de tipo 2) no permite expresar características representables con gramáticas sensibles al contexto –como la necesidad de que la utilización de una variable en el lenguaje Pascal requiera la declaración previa de la variable utilizada. Para poder implementar un procesador del lenguaje Pascal empleando gramáticas de tipo 2 e implementaciones de autómatas de pila, es necesario emplear una estructura de datos auxiliar denominada tabla de símbolos. Esta estructura de datos, a su nivel más básico, es un diccionario (memoria asociativa) que asocia identificadores a la información requerida por el compilador. Sus dos operaciones básicas son `insertar` y `buscar`. En nuestro ejemplo, la declaración de un identificador en Pascal requerirá una inserción del mismo en la tabla de símbolos; cada vez que se utilice un identificador en una sentencia, el analizador semántico buscará éste en la tabla de símbolos (llamando al manejador de errores si no existiere).

2.1. Ejemplos de Comprobaciones Realizadas por el Analizador Semántico

Existen multitud de ejemplos reales de comprobaciones llevadas a cabo por el analizador semántico de un procesador de lenguaje. Describiremos algunos de ellos a modo de ejemplo.

Declaración de identificadores y reglas de ámbitos

En el primer ejemplo de este libro, analizábamos la siguiente sentencia en C:

[12] Un compilador no suele generar el código destino directamente a una plataforma específica, sino que utiliza un mecanismo intermedio de representación de código [Cueva98].

```
superficie = base * altura / 2;
```

Para que la asignación previa fuese correcta, los tres identificadores deberían de estar declarados. Puede que estén declarados en el ámbito (bloque) actual o en uno menos anidado que el actual, en cuyo caso el analizador sintáctico tendría que aplicar reglas de ámbito como la ocultación de identificadores.

Ejemplo 6. Si enmarcamos la sentencia anterior en el siguiente programa C:

```
#include <stdio.h>
int main() {
       double base=2.5, altura=10;
       {
              double superficie, altura = 1;
              superficie = base * altura / 2;
              printf("%lf", superficie);
       }
       printf("%lf", superficie);
       return 0;
}
```

El primer `printf` es correcto, pero no el segundo. En el primer caso, todas los identificadores de la asignación están declarados: `superficie` y `altura` (con valor 1) están declarados en el ámbito actual (`altura` oculta al identificador con valor 10, puesto que está más anidado) y el valor mostrado es 1.25. Sin embargo, el segundo `printf` no es correcto puesto que la superficie del triángulo no ha sido declarada.

□

Comprobaciones de unicidad

Existen multitud de elementos en lenguajes de programación cuyas entidades han de existir de un modo único, es decir, no se permite que estén duplicadas. Ejemplos típicos son:

- Constantes de cada `case` en Pascal, C o Java. Cada uno de los elementos existentes en los condicionales múltiples de los lenguajes de programación mencionados, ha de ser único. En otro caso, el analizador semántico deberá generar un error de compilación.
- Los valores de un tipo *enumerado* de Pascal o C han de ser únicos.
- Las etiquetas de un lenguaje de programación, como un ensamblador, no pueden estar repetidas, puesto que los saltos a las mismas serían ambiguos.
- La declaración de un identificador en un ámbito ha de ser única en multitud de lenguajes de programación.

Comprobaciones de enlace[13]

En ocasiones, el empleo de un elemento de un lenguaje ha de estar ligado a una utilización previa del mismo:

- En un ensamblador, un salto a una etiqueta requiere que ésta haya sido referida como una posición de memoria.
- En el lenguaje de programación ANSI C [Kernighan91] y en ISO/ANSI C++ [ANSIC++] la invocación a una función o método requiere que éstos hayan sido declarados previamente.

[13] *Binding*.

- La especificación de ámbitos para determinadas estructuras de control, en determinados lenguajes como BASIC, requiere la utilización pareja de palabras reservadas –como `IF / END IF`, `FOR ID / NEXT ID` o `SUB / END SUB`.

Comprobaciones pospuestas por el analizador sintáctico

A la hora de implementar un procesador de un lenguaje de programación, es común encontrarse con situaciones en las que una gramática libre de contexto puede representar sintácticamente propiedades del lenguaje; sin embargo, la gramática resultante es compleja y difícil de procesar en la fase de análisis sintáctico. En estos casos es común ver cómo el desarrollador del compilador escribe una gramática más sencilla que no representa detalles del lenguaje, aceptándolos como válidos cuando realmente no pertenecen al lenguaje. En la posterior fase de análisis semántico será donde se comprueben aquellas propiedades del lenguaje que, por sencillez, no fueron verificadas por el analizador sintáctico.

Hay multitud de escenarios de ejemplo y están en función de la implementación de cada procesador. Sin embargo, los siguientes suelen ser comunes:

- Es posible escribir una gramática libre de contexto capaz de representar que toda implementación de una función en C tenga al menos una sentencia `return`. No obstante, si escribimos la gramática de cualquier función como una repetición de sentencias, siendo `return` es un tipo de sentencia, la gramática es más sencilla y fácil de procesar. El analizador semántico deberá comprobar, pues, dicha restricción.

- Cuando un lenguaje posee el operador de asignación como una expresión y no como una sentencia (C y Java frente a Pascal), hay que comprobar que la expresión de la parte izquierda de la asignación posee una dirección de memoria en la que se pueda escribir (*lvalue*). Esta restricción puede ser comprobada por el analizador semántico, permitiendo sintácticamente que cualquier expresión se encuentre en la parte izquierda del operador de asignación.

- Las sentencias `break` y `continue` de Java y C sólo pueden utilizarse en determinadas estructuras de control del lenguaje. Éste es otro escenario para que el analizador sintáctico posponga la comprobación hasta la fase análisis semántico.

Comprobaciones dinámicas

Todas las comprobaciones semánticas descritas en este punto suelen llevarse a cabo en fase de compilación y por ello reciben el nombre de "estáticas". Existen comprobaciones que, en su caso más general, sólo pueden ser llevadas a cabo en tiempo de ejecución y por ello se llaman "dinámicas". Éstas suelen ser comprobadas por un intérprete o por código de comprobación generado por el compilador –también puede darse el caso de que no se comprueben. Diversos ejemplos pueden ser acceso a un vector fuera de rango, utilización de un puntero nulo o división por cero.

Analizaremos más en detalle este tipo de comprobaciones en § 6.5.

Comprobaciones de tipo

Sin duda, este tipo de comprobaciones es el más exhaustivo y amplio en fase de análisis semántico. Ya bien sea de un modo estático (en tiempo de compilación), dinámico (en tiempo de ejecución) o en ambos, las comprobaciones de tipo son necesarias en todo lenguaje de alto nivel. De un modo somero, el analizador semántico deberá llevar a cabo las dos siguientes tareas relacionadas con los tipos:

1. Comprobar las operaciones que se pueden aplicar a cada construcción del lenguaje. Dado un elemento del lenguaje, su tipo identifica las operaciones que sobre él se pueden aplicar. Por ejemplo, en el lenguaje Java el operador de producto no es aplicable a una referencia a un objeto. De un modo contrario, el operador punto sí es válido.
2. Inferir el tipo de cada construcción del lenguaje. Para poder implementar la comprobación anterior, es necesario conocer el tipo de toda construcción sintácticamente válida del lenguaje. Así, el analizador semántico deberá aplicar las distintas reglas de inferencia de tipos descritas en la especificación del lenguaje de programación, para conocer el tipo de cada construcción del lenguaje.

La tarea de comprobar todas las restricciones de cada tipo y la inferencia de éstos será ampliamente descrita en § 0.

2.2. Análisis Semántico como Decoración del AST

Un procesador de lenguaje en el que todas las fases ocurren en un único recorrido del código fuente se denomina ***de una pasada***[14]. En este tipo de procesadores, el análisis semántico y la generación de código están intercaladas con el análisis sintáctico –y por tanto con el análisis léxico. Este tipo de compiladores es más eficiente y emplea menos memoria que los que requieren más de una pasada. Sin embargo, el código que genera acostumbra a ser menos eficiente que los compiladores que emplean más de una pasada. Pascal y C son ejemplos de lenguajes que pueden ser compilados con una sola pasada –por ello, es siempre necesario declarar una función antes de utilizarla (`forward` en Pascal).

Existen lenguajes como Modula-2 o Java cuyas estructuras requieren más de una pasada para ser procesados –se puede invocar a métodos o funciones definidas posteriormente en el archivo. Asimismo, la mayoría de los compiladores que optimizan el código generado (en su representación intermedia o final) son de más de una pasada, para poder analizar y optimizar el código[15]. Es importante resaltar que una pasada de un compilador es un concepto distinto al de una fase. En una pasada se pueden llevar a cabo todas las fases (si éste es de una pasada), o para una fase se pueden dar varias pasadas (como la optimización intensiva de código). Las configuraciones intermedias son también comunes.

El análisis semántico de un programa es más sencillo de implementar si se emplean para las fases de análisis sintáctico y semántico dos o más pasadas[16]. En este caso, la fase de análisis sintáctico creará un árbol sintáctico abstracto para que sea procesado por el analizador semántico.

Si el procesador es de una pasada, el analizador sintáctico irá llamando al analizador semántico de un modo recursivo y, si bien ningún árbol es creado de forma explícita, los ámbitos de las invocaciones recursivas (o los niveles de la pila del reconocedor) formarán implícitamente el árbol sintáctico.

Árbol de sintaxis abstracta

Como sabemos, un **árbol sintáctico**[17]es una representación de la estructura de una consecución de componentes léxicos (*tokens*), en la que éstos aparecen como nodos hoja y

[14] *One-pass compier*.
[15] Existen compiladores que optimizan el código de un modo intensivo, llegando a dar hasta 7 pasadas a la representación del programa [Louden97].
[16] En ocasiones, como en el caso de Java o Modula2, es de hecho en único modo de hacerlo.
[17] *Parse tree*.

los nodos internos representan los pasos en las derivaciones de la gramática asociada. Los árboles sintácticos poseen mucha más información de la necesaria para el resto de las fases de un compilador, una vez finalizada la fase de análisis sintáctico.

Una simplificación de árbol sintáctico que represente toda la información necesaria para el resto del procesamiento del programa de un modo más eficiente que el árbol sintáctico original, recibe el nombre de **árbol de sintaxis abstracta** (**AST**, *Abstract Syntax Tree*). Así, la salida generada por un analizador sintáctico de varias pasadas, será el AST representativo del programa de entrada.

Un AST puede ser visto como el árbol sintáctico de una gramática denominada abstracta, al igual que un árbol sintáctico es la representación de una gramática (en ocasiones denominada concreta). Por tanto, es común ver una gramática que representa una simplificación de un lenguaje de programación denominada **gramática abstracta** del lenguaje.

Ejemplo 7. Lo siguiente es una gramática (concreta) de una expresión, descrita en la notación propia de yacc/bison [Mason92]:

```
expresion: expresion '+' termino
         | expresion '-' termino
         | termino
         ;
termino: termino '*' factor
       | termino '/' factor
       | factor
       ;
factor: '-' factor
       | '(' expresion ')'
       | CTE_ENTERA
       ;
```

La sentencia `3*(21+-32)` pertenece sintácticamente al lenguaje y su árbol sintáctico es:

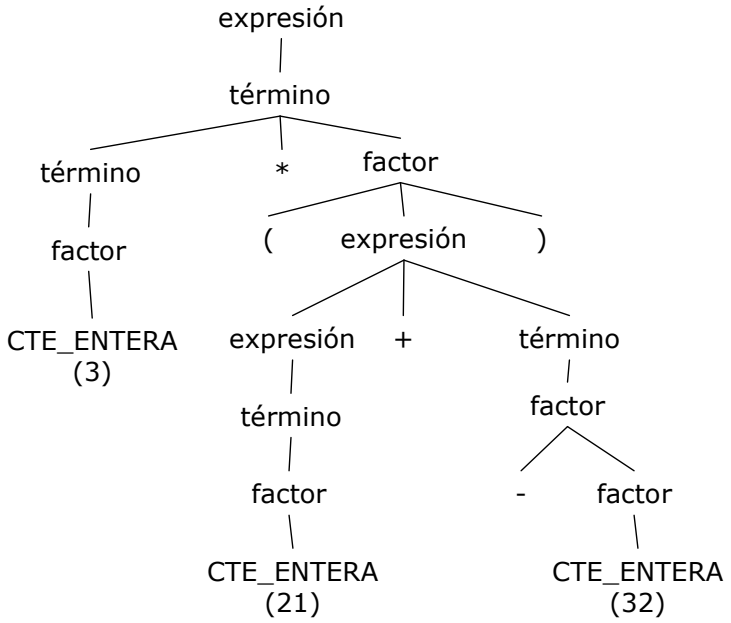

Figura 2: Árbol sintáctico generado para la sentencia 3*(21+-32).

En el árbol anterior hay información (como los paréntesis o los nodos intermedios factor y término) que no es necesaria para el resto de fases del compilador. La utilización de un AST simplificaría el árbol anterior. Una posible implementación sería siguiendo el siguiente diagrama de clases:

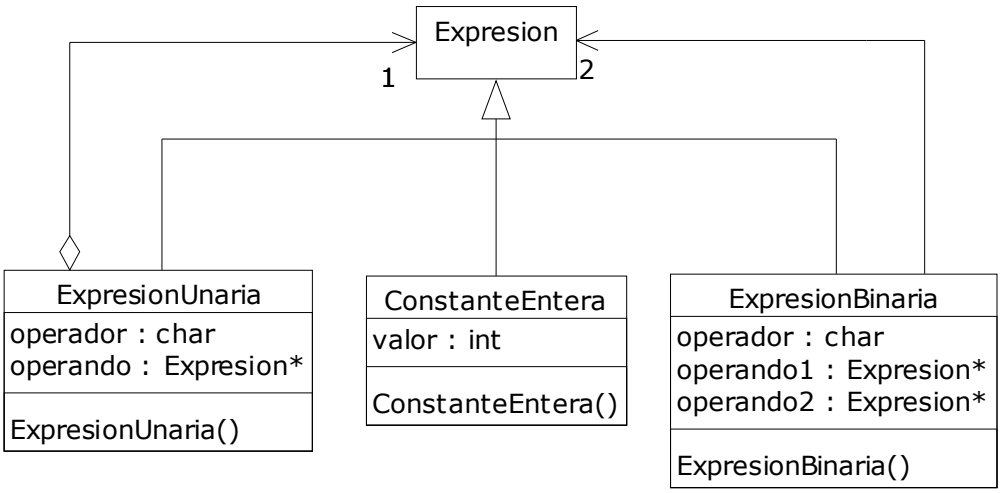

Figura 3: Diagrama de clases de los nodos de un AST.

Su implementación en C++ puede consultarse en el apéndice A.1. Todos los nodos son instancias derivadas de la clase abstracta expresión. De este modo, se podrá trabajar con cualquier expresión, independientemente de su aridad, mediante el uso de esta clase. Todas las expresiones binarias serán instancias de la clase `ExpresionBinaria`, teniendo un atributo que denote su operador. Del mismo modo, las expresiones con un operador unario (en nuestro ejemplo sólo existe el menos unario) son instancias de `ExpresionUnaria`. Finalmente, las constantes enteras son modeladas con la clase `ConstanteEntera`.

Una implementación yacc/bison que cree el AST a partir de un programa de entrada es:

```
%{
#include "ast.h"
int yyparse();
int yylex();
%}
%union {
  int entero;
  Expresion *expresion;
  }
%token <entero> CTE_ENTERA
%type <expresion> expresion termino factor
%%
expresion: expresion '+' termino {$$=new ExpresionBinaria('+',$1,$3); }
       | expresion '-' termino {$$=new ExpresionBinaria('-',$1,$3); }
       | termino {$$=$1;}
       ;
termino: termino '*' factor {$$=new ExpresionBinaria('*',$1,$3); }
       | termino '/' factor {$$=new ExpresionBinaria('/',$1,$3); }
       | factor { $$=$1; }
       ;
factor: '-' factor { $$=new ExpresionUnaria('-',$2); }
       | '(' expresion ')' { $$=$2; }
       | CTE_ENTERA { $$=new ConstanteEntera($1); }
       ;
%%
```

Así, ante la misma sentencia de entrada `3*(21+-32)`, el AST generado tendrá la siguiente estructura:

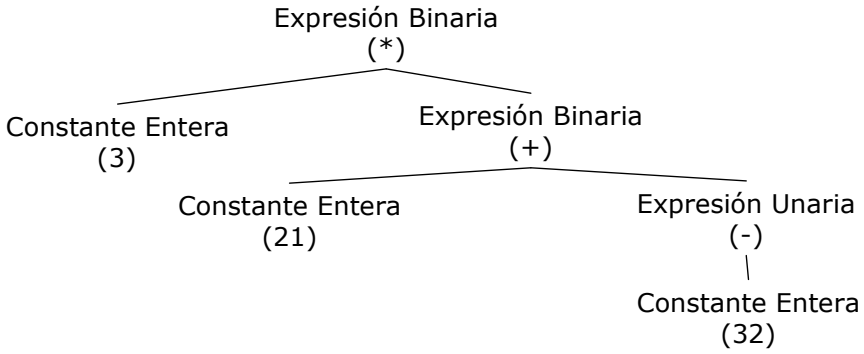

Figura 4: AST generado para la sentencia 3*(21+-32).

Nótese cómo ya no son necesarios los nodos de factor y término, puesto que el propio árbol ya se creará con una estructura que refleje la información relativa a la precedencia de operadores. De la misma forma, el procesamiento del AST en las siguientes fases es notablemente más sencillo que el del árbol sintáctico original.

La gramática abstracta de nuestro ejemplo es:

```
expresion: expresionBinaria
         | expresionUnaria
         | constanteEntera
         ;
expresioBinaria: expresion ('*'|'/'|'+'|'-') expresion
         ;
expresionUnaria: '-' expresion
         ;
constanteEntera: CTE_ENTERA
         ;
```

□

Ejemplo 8. Considérese la siguiente gramática libre de contexto, que representa una simplificación de la sintaxis de una sentencia condicional en un lenguaje de programación imperativo –los símbolos terminales se diferencian de los no terminales porque los primeros han sido escritos en negrita:

```
sentencia     →  sentenciaIf
              |  lectura
              |  escritura
              |  expresion
sentenciaIf   →  if ( expresion ) sentencia else
else          →  else sentencia
              |  λ
lectura       →  read expresion
escritura     →  write expresion
expresion     →  true
              |  false
              |  cte_entera
              |  id
              |  expresion + expresion
              |  expresion - expresion
              |  expresion * expresion
              |  expresion / expresion
              |  expresion = expresion
```

La gramática anterior acepta más sentencias que las pertenecientes al lenguaje, como suele suceder en la mayor parte de los casos. El analizador semántico debería restringir la validez semántica de las sentencias teniendo en cuenta que la expresión de la estructura condicional debe ser lógica (o entera en el caso de C), y que tanto la expresión a la izquierda del operador de asignación como la expresión de la sentencia de lectura sean *lvalues*.

La siguiente sentencia es sintácticamente válida:

```
if (true)
      read a
else
      write a=2+a*b
```

La implementación de un procesador del lenguaje presentado en la que se emplee más de una pasada, un AST apropiado generado por el analizador sintáctico podría ser el mostrado en la Figura 5.

Nótese cómo el AST representa, de un modo más sencillo que el árbol sintáctico, la estructura de la sentencia condicional con tres nodos hijos: la expresión de la condición, la sentencia a ejecutar si la condición es válida, y la asociada al valor falso de la condición. Del mismo modo, las expresiones creadas poseen la estructura adecuada para ser evaluadas con un recorrido en profundidad infijo, de un modo acorde a las precedencias.

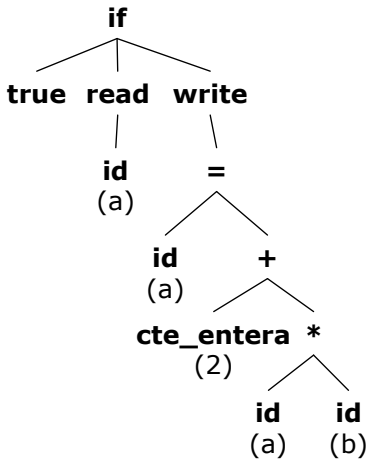

Figura 5: AST generado para la gramática y lenguaje de entrada presentados.

□

Decoración del AST

Una vez que el analizador semántico obtenga el AST del analizador sintáctico, éste deberá utilizar el AST para llevar a cabo todas las comprobaciones necesarias para verificar la validez semántica del programa de entrada. Este proceso es realizado mediante lo que se conoce como decoración o anotación del AST: asignación de información adicional a los nodos del AST representando propiedades de las construcciones sintácticas del lenguaje, tales como el tipo de una expresión. Un **AST decorado** o **anotado**[18] es una ampliación del AST, en el que a cada nodo del mismo se le añaden atributos indicando las propiedades necesarias de la construcción sintáctica que representan.

Ejemplo 9. Dada la gramática libre de contexto:

```
S            →   S declaracion ;
              |   S expresion ;
              |   λ
declaracion  →   int id
              |   float id
expresion    →   id
              |   cte_entera
              |   cte_real
              |   ( expresion )
              |   expresion + expresion
              |   expresion - expresion
```

[18] *Annotated AST* o *decorated AST*.

```
|   expresion * expresion
|   expresion / expresion
|   expresion = expresion
```

Para implementar un analizador semántico deberemos decorar el AST con la siguiente información:

- Las expresiones tendrán asociadas un atributo `tipo` que indique si son reales o enteras. Esto es necesario porque se podrá asignar un valor entero a una expresión real, pero no al revés.
- Las expresiones deberán tener un atributo lógico que indique si son o no *lvalues*. De este modo, se podrá comprobar si lo que está a la izquierda de la asignación es o no semánticamente correcto.
- En una declaración se deberá insertar el identificador en una tabla de símbolos con su tipo declarado, para poder conocer posteriormente el tipo de cualquier identificador en una expresión. Es, por tanto, necesario asignar un atributo `nombre` (cadena de caracteres) a un identificador.
- Finalmente –aunque más enfocado a la fase de generación de código o interpretación que al análisis semántico– se le asigna un valor entero o real a las constantes del lenguaje.

Para el siguiente programa, un posible AST decorado es el mostrado en la Figura 6.

```
int a;
float b;
b=(a+1)*(b-8.3);
```

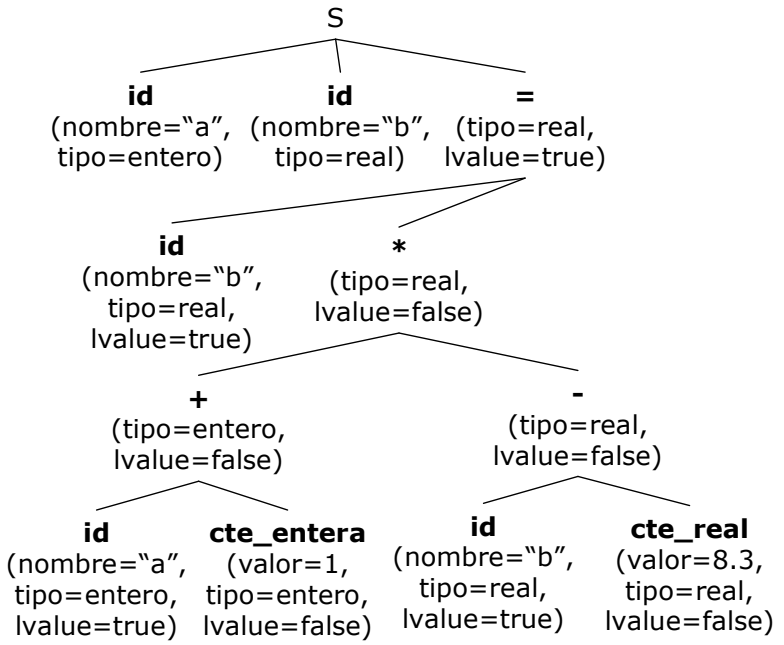

Figura 6: AST decorado para llevar a cabo el análisis semántico del programa analizado.

Nótese cómo el analizador semántico será el encargado de decorar el AST como se muestra en la Figura 6, además de comprobar las restricciones mencionadas con anterioridad –por las que, precisamente, se ha decorado el árbol.

□

De este modo es más sencillo llevar a cabo la tarea de análisis semántico puesto que toda la información sintáctica está explícita en el AST, y el análisis semántico tiene que limitarse a decorar el árbol y comprobar las reglas semánticas del lenguaje de programación. En el ejemplo anterior, utilizando la estructura del árbol se va infiriendo el tipo de cada una de las expresiones. La información anotada a cada nodo del árbol será empleada también por las siguientes fases del procesador de lenguajes como la generación de código. Siguiendo con el ejemplo, el conocer los tipos de cada una de las construcciones del AST facilita al generador de código saber el número de bytes que tiene que reservar, la instrucción de bajo nivel que tiene que emplear, o incluso si es necesario o no convertir los operandos antes de realizar la operación.

El principal formalismo que existe para decorar árboles sintácticos es el concepto de gramática atribuida. Mediante gramáticas atribuidas se implementan analizadores semánticos a partir del AST o del árbol sintáctico. También, partiendo de una gramática atribuida, existen métodos de traducción de éstas a código. Estos conceptos y técnicas serán lo que estudiaremos en los siguientes puntos.

3 Gramáticas Atribuidas

Las gramáticas libres de contexto son las elegidas comúnmente para representar la sintaxis de los lenguajes de programación. Éstas representan cómo debe ser la estructura de cualquier programa perteneciente al lenguaje que describen. Sin embargo, un procesador de lenguaje necesita conocimiento adicional del significado de las construcciones para llevar a cabo acciones en función de la fase en la que se encuentre.

A modo de ejemplo, consideremos la gramática inicial de expresiones mostrada en el Ejemplo 7. La gramática representa la estructura de un tipo de expresiones aritméticas de constantes enteras, teniendo en cuenta la precedencia común de los operadores empleados. En la fase de análisis sintáctico el reconocimiento de dicha estructura es suficiente, pero en las fases posteriores hay que llevar a cabo el cálculo de distintas parte de su semántica:

- En la fase de análisis semántico, para una ampliación del ejemplo con diversos tipos, habrá que comprobar que la expresión satisface las reglas propias de los tipos del lenguaje, calculando los tipos de cada subexpresión y analizando si las operaciones aplicadas son válidas para los tipos inferidos.
- En la generación de código habrá que ir traduciendo el programa de entrada a otro programa con igual semántica, pero expresado en otro lenguaje de salida.
- Si nos encontramos desarrollando un intérprete, en la fase de ejecución necesitaremos conocer el valor de cada una de las expresiones. Esto mismo puede darse si se está implementando un compilador y nos encontramos en la fase de optimización de código, en la que podemos reemplazar el cálculo de una expresión con todos sus operandos constantes por su valor –esta optimización recibe el nombre de *calculo previo de constantes*[19].

Para llevar a cabo las tareas previas, es común emplear en el diseño e implementación de un procesador de lenguaje gramáticas atribuidas. Las gramáticas atribuidas (o con atributos) son ampliaciones de las gramáticas libres de contexto que permiten especificar semántica dirigida por sintaxis: la semántica de una sentencia de un lenguaje de programación está directamente relacionada con su estructura sintáctica y, por tanto, se suele representar mediante anotaciones de su árbol sintáctico [Louden97]. Puesto que el uso de las gramáticas atribuidas es posterior a la fase de análisis sintáctico, es común ver éstas como ampliación de gramáticas que especifiquen sintaxis abstracta y no concreta [Saraiva99] (§ 2.2).

3.1. Atributos

Un **atributo** es una propiedad de una construcción sintáctica de un lenguaje. Los atributos varían considerablemente en función de la información que contienen, de la fase de procesamiento en la que se hallen, e incluso en si están siendo calculados en fase de traducción (empleados por los traductores y compiladores) o de ejecución (empleados por

[19] *Constant folding*.

los intérpretes). Típicos ejemplos de atributos de una gramática son: el tipo de una variable, el valor de una expresión, la dirección de memoria de una variable o el código objeto (destino) de una función.

El tiempo en el que los atributos mencionados son calculados es variable en función del lenguaje y procesador empleado. Pueden estáticos si son calculados de un modo previo a la ejecución de la aplicación, o dinámicos si su valor se obtiene en tiempo de ejecución del programa. En el primer ejemplo mencionado –el cálculo del tipo de una variable–, para lenguajes como C y Pascal, la inferencia de tipos es resuelta de un modo estático por el analizador semántico; en el caso de Lisp, la comprobación relativa a los tipos es siempre calculada en tiempo de ejecución.

Si `a` es un atributo de un símbolo gramatical `X`, escribiremos `X.a` para referirnos al valor del atributo `a` asociado al símbolo gramatical `X`. Para cada atributo, se debe especificar su dominio: el conjunto de posibles valores que puede tomar[20]. Si en una producción de la gramática libre de contexto aparece más de una vez un símbolo gramatical X, entonces se debe añadir un subíndice a cada una de las apariciones para podernos referir a los valores de los atributos de un modo no ambiguo: $X_1.a, X_2.a, X_3.a...$

Ejemplo 10. Dado el símbolo no terminal `expresion`, podremos tener en un compilador los siguientes atributos definidos sobre él:

Atributo	Dominio (posibles valores)	Descripción
`expresion.tipo`	Entero, carácter, real, booleano, array, registro, puntero o función.	El tipo inferido para la expresión (análisis semántico).
`expresion.lvalue`	Un valor lógico	Si la expresión puede estar o no a la izquierda de la asignación (análisis semántico).
`expresion.codigo`	Cadena de caracteres	El código objeto (generación de código).
`expresion.valor`	Una unión de valores de tipo entero, real o lógico	Cuando sea posible, calcular el valor de una expresión de tipo entera, real o booleana (optimización de código).

□

3.2. Reglas Semánticas

Las relaciones entre los valores de los atributos definidos sobre una gramática atribuida se especifican mediante **reglas semánticas** o **ecuaciones de atributos**[21] [Louden97]. Dada una gramática libre de contexto G={S,P,VN,VT}, donde VN es el vocabulario no terminal, VT el vocabulario terminal (*tokens*), S el símbolo no terminal inicial y P el conjunto de producciones de la gramática, donde cada p ∈ P es de la forma:

$$X_0 \rightarrow X_1 X_2 ... X_n, X_0 \in VN, X_i \in VT \cup VN, 1 \leq i \leq n$$

[20] El concepto de dominio viene de la especificación semántica de lenguajes, en especial de la denotacional. En programación, el concepto de dominio es traducido al concepto de tipo.
[21] *Attribute equation* o *semantic rule*.

Una regla semántica asociada a la producción p es una función matemática que especifica las relaciones entre los valores de los atributos del siguiente modo [Aho90]:

$$a := f(a_1, a_2, a_3 ... a_k)$$

Donde $a_i, 1 \le i \le k$ son atributos de los símbolos gramaticales de la producción ($X_j \in VT \cup VN, 0 \le j \le n$) y a es:

- O bien un atributo **sintetizado** del símbolo gramatical no terminal X_0 situado a en la parte izquierda de la producción p.
- O bien un atributo **heredado** de uno de los símbolos gramaticales $X_i \in VT \cup VN, 1 \le i \le n$ situados en la parte derecha de la producción p.

En cualquier caso, se dice que el atributo a depende de los atributos $a_i, 1 \le i \le k$. El conjunto de atributos sintetizados de un símbolo gramatical X se suele representar con el conjunto AS(X). Del mismo modo, los atributos heredados de X se representan con el conjunto AH(X)[22].

Los atributos de los símbolos terminales de la gramática se consideran atributos sintetizados –puesto que su valor ha sido asignado por el analizador léxico.

3.3. Gramáticas Atribuidas

Las gramáticas atribuidas fueron definidas originalmente por Knuth [Knuth68] como un método para describir la semántica de un lenguaje de programación. Una gramática atribuida (GA) es una tripleta GA={G,A,R}, donde:

- G es una gramática libre de contexto definida por la cuádrupla G={S,P,VN,VT}
- $A = \left\{ \bigcup_{X \in VN \cup VT} A(X) \right\}$ es un conjunto finito de atributos, siendo X un símbolo gramatical (X ∈ VN ∪ VT) y A(X) el conjunto de atributos definidos para X.
- $R = \left\{ \bigcup_{p \in P} R(p) \right\}$ es un conjunto finito de reglas semánticas o ecuaciones de atributos, siendo R(p) el conjunto de reglas semánticas asociadas a p ∈ P y P el conjunto de producciones de la gramática libre de contexto G.

Como hemos definido en el punto § 3.2, las reglas semánticas establecen relaciones entre los valores de los atributos de una gramática atribuida, expresadas mediante una función matemática. Desde el punto de vista más estricto de definición de gramática atribuida, las reglas semánticas únicamente pueden recibir como parámetros otros atributos de la gramática (no están permitidas las constantes, variables o llamadas a funciones que generen efectos colaterales[23]) y devolver, sin generar un efecto colateral, un único valor que será asignado a otro atributo de la producción actual.

[22] En los textos escritos en inglés, este conjunto es AI(X) ya que el atributo es *inherited*.

[23] Una función que no posea efectos colaterales es aquélla que devuelve y genera siembre el mismo resultado al ser llamada con el mismo conjunto de argumentos. Una función que incremente una variable global, por ejemplo, genera efectos laterales. Las funciones que no tienen efectos colaterales son meras expresiones de cálculo computacional sobre los argumentos pasados.

A efectos prácticos, la reglas semánticas son fragmentos de código expresadas en una notación bien definida, como pueda ser un lenguaje de programación, una notación matemática o un lenguaje de especificación semántica como la denotacional o axiomática (§ 1.1). El lenguaje empleado para especificar las acciones semánticas recibe el nombre de **metalenguaje** [Louden97]. Realmente, ni la notación de especificación de las reglas semánticas, ni la especificación de los tipos (dominios) de los atributos, son intrínsecos al concepto de gramática atribuida[24] [Scott00].

En ocasiones, las gramáticas atribuidas que emplean un metalenguaje que permite la utilización de funciones que tengan efectos colaterales son denominadas ***definiciones dirigidas por sintaxis***[25] o *definiciones atribuidas* [Aho90].

Ejemplo 11. Dada la gramática libre de contexto G={S,P,VN,VT}, donde S es `expresion`, VN={`expresión`, `termino` y `factor`}, VT={`+`, `-`, `*`, `/`, `(`, `)`, `CTE_ENTERA`} y P es[26]:

```
(1)  expresion1   →  expresion2  +  termino
(2)               |  expresion2  -  termino
(3)               |  termino
(4)  termino1     →  termino2  *  factor
(5)               |  termino2  /  factor
(6)               |  factor
(7)  factor1      →  -  factor2
(8)               |  (  expresion  )
(9)               |  CTE_ENTERA
```

Nótese como, en caso de existir una repetición de un símbolo gramatical en una misma producción, se ha roto la posible ambigüedad con la adición de subíndices. Una gramática atribuida que sea capaz de evaluar el valor de las expresiones del lenguaje, necesaria en un intérprete o para optimizar código (véase Ejemplo 10), estará formada por GA={G, A, R}, donde A será:

Atributo	Dominio
`expresion.valor`	Números enteros
`termino.valor`	Números enteros
`factor.valor`	Números enteros
`CTE_ENTERA.valor`	Números enteros

Finalmente, R será:

P	R
(1)	$expresion_1$.valor = $expresion_2$.valor + termino.valor
(2)	$expresion_1$.valor = $expresion_2$.valor – termino.valor
(3)	expresion.valor = termino.valor
(4)	$termino_1$.valor = $termino_2$.valor * factor.valor
(5)	$termino_1$.valor = $termino_2$.valor / factor.valor
(6)	termino.valor = factor.valor
(7)	$factor_1$.valor = $factor_2$.valor

[24] De hecho, desde la primera definición de gramática atribuida dada por Knuth [Knuth68] existe múltiple bibliografía que define y emplea gramáticas atribuidas variando estos dos parámetros en función del objetivo buscado.

[25] *Syntax-directed definition*.

[26] Las producciones de las gramáticas han sido numeradas para poder hacer referencia a ellas de un modo más sencillo.

P	R
(8)	`factor.valor = expresion.valor`
(9)	`factor.valor = CTE_ENTERA.valor`

Hemos utilizado como metalenguaje el lenguaje de programación C. En todos los casos los atributos son sintetizados, puesto que el atributo que se calcula en toda regla semántica está en la parte izquierda de su producción asociada –el atributo `CTE_ENTERA.valor` es sintetizado por el analizador léxico.

La gramática atribuida se ha construido para que la evaluación de los cuatro operadores sea asociativa a izquierdas, de modo que la sentencia `1-1-2` poseerá el valor -2 (y no 2).

□

Del mismo modo que las gramáticas libres de contexto no especifican cómo deben ser procesadas por el analizador sintáctico (ascendente o descendentemente, en sus múltiples alternativas), una gramática atribuida no especifica el orden en el que los atributos tienen que ser calculados. Las gramáticas atribuidas son, pues, declarativas y no imperativas [Wilhelm95]. Veremos cómo, al igual que existen clasificaciones de gramáticas libres de contexto en función de los algoritmos que las procesan (LL, SLL, LL(K), LR, SLR o LALR) también existen gramáticas atribuidas en función del orden de evaluación de sus atributos (L-atribuidas, S-atribuidas y bien definidas).

Las principales clasificaciones de gramáticas atribuidas tienen en cuenta si los atributos calculados en las producciones son heredados o sintetizados. Es importante comprender la noción de cada uno de ellos y cuándo y cómo es necesario emplear uno u otro. Supongamos una producción p ∈ P de una gramática libre de contexto:

$$X_0 \rightarrow X_1 X_2 ... X_n, X_0 \in VN, X_i \in VT \cup VN, 1 \leq i \leq n$$

Los atributos sintetizados se calculan "ascendentemente" en el árbol sintáctico: en función de los atributos de los nodos hijos y asignándole un valor a un atributo sintetizado del nodo padre (Figura 7). Por este motivo, se dice que en esa producción el atributo se sintetiza (podrá ser empleado en producciones en las que el símbolo gramatical X_0 se encuentre en la parte derecha de la producción).

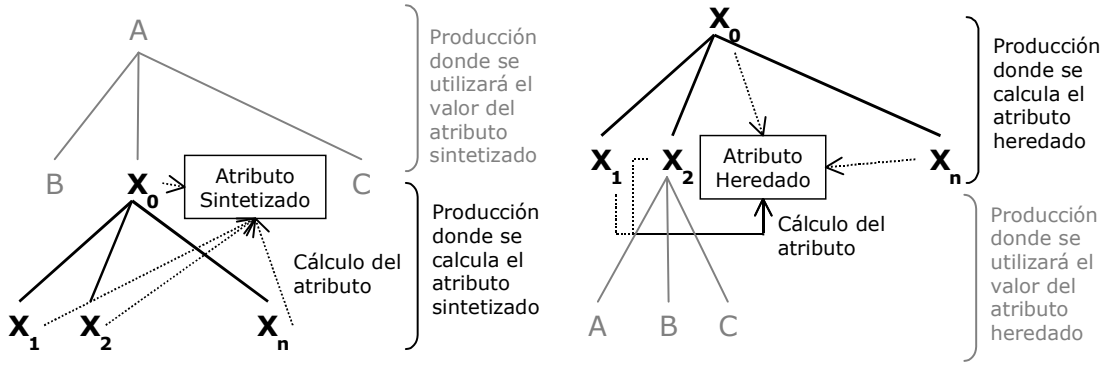

Figura 7: Evaluación de atributos heredados y sintetizados.

De un modo contrario, los atributos heredados se calculan "descendentemente" en el árbol sintáctico. Como se muestra en la Figura 7, se asigna un valor a un atributo del nodo hijo X_2 para que, en aquellas reglas en las que éste aparezca en la parte izquierda de la producción, herede el valor asignado.

Siguiendo con este mismo criterio, en cualquier producción se podrá utilizar los atributos de los símbolos terminales de la gramática –que, por definición, aparecerán en la

parte derecha de la producción. Es por ello por lo que éstos se consideran sintetizados por el analizador léxico.

Ejemplo 12: Dada la siguiente gramática libre de contexto:

```
(1)  declaracion    →  tipo variables ;
(2)  tipo           →  int
(3)                 |  float
(4)  variables1     →  id , variables2
(5)                 |  id
```

Supóngase que tenemos un objeto `ts` (tabla de símbolos) que nos ofrece el método `insertar` para añadir a una estructura de datos el valor de un tipo asociado a una cadena. El objetivo es, mediante una definición dirigida por sintaxis, insertar los identificadores de la gramática en la tabla de símbolos junto a su tipo apropiado. En un procesador de lenguaje real, esta información se necesitará para conocer el tipo de un identificador cuando forme parte de una expresión.

Nótese cómo las producciones en las que realmente conocemos el identificador a insertar en la tabla de símbolos son la cuarta y quinta. Sin embargo, el tipo que puedan tener asociado es conocido en las reglas 2 y 3. De este modo, una solución es transmitir la información relativa al tipo hasta las reglas 4 y 5 para que éstas inserten el identificador en la tabla de símbolos:

P	R
(1)	variables.tipo = tipo.tipo
(2)	tipo.tipo = 'I'
(3)	tipo.tipo = 'F'
(4)	ts.insertar(id.valor,variables$_1$.tipo) variables$_2$.tipo=variables$_1$.tipo
(5)	ts.insertar(id.valor,variables.tipo)

El valor del tipo declarado se sintetiza en las reglas 2 o 3 y es pasado como heredado, por medio de la regla 1, al no terminal `variables`. Éste lo empleará para, junto al valor del identificador (su nombre), insertarlo en la tabla de símbolos.

Es importante darse cuenta de que el valor que `variables` ha heredado en la regla 4, deberá asignárselo a la segunda aparición del mismo no terminal, para que este proceso sea llevado a cabo de un modo recursivo –heredando siempre su valor. En la Figura 8 se muestra el árbol propio del siguiente programa de entrada: `int a,b;`

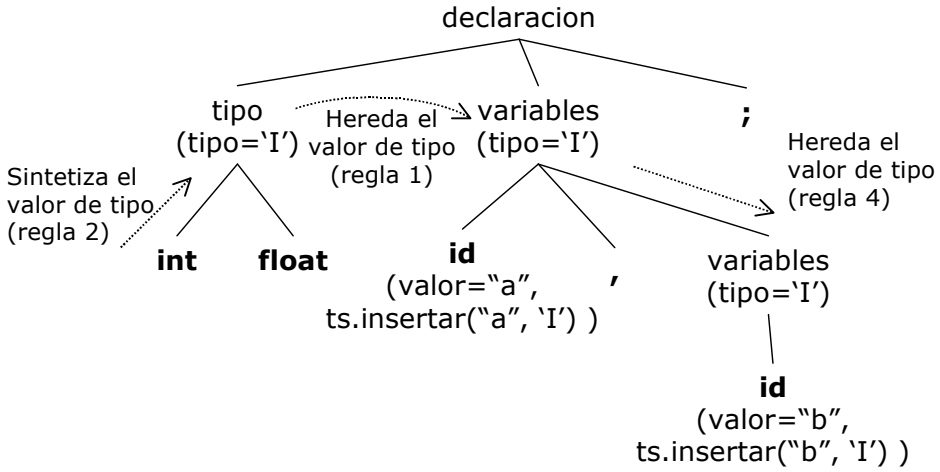

Figura 8: Árbol sintáctico generado para el programa de entrada `int a,b;`.

Si el valor del atributo `tipo` no es asignado al nodo `variables` situado en la parte inferior derecha por medio de la asignación de la producción cuarta, este atributo no tendría el valor `I`. El resultado sería que no se insertaría correctamente el identificador `b` en la tabla de símbolos mediante la última regla semántica.

Si se trata de resolver el problema propuesto únicamente con atributos sintetizados, la solución sería mucho más compleja. Tendríamos que ir guardando en un atributo sintetizado de `variables` la lista de todos los identificadores declarados. En la primera producción se tendría que poner un bucle en el que cada uno de los identificadores de la lista de variables sería insertado en la tabla de símbolos con el tipo sintetizado.

Si en lugar de una definición dirigida por sintaxis tuviésemos que escribir una gramática atribuida, no se podría utilizar una estructura de datos auxiliar por los efectos colaterales que se producen al insertar elementos. Debería definirse un atributo que fuese una lista de los símbolos e ir pasando éste a lo largo de todos los no terminales del programa. En el desarrollo real de un procesador de lenguaje esto produciría un elevado acoplamiento y, por lo tanto, se suele abordar con una definición dirigida por sintaxis.

□

Ejemplo 13: Dada la siguiente gramática libre de contexto:

```
(1)  expresion      → termino masTerminos
(2)  masTerminos1   → + termino masTerminos2
(3)                 | - termino masTerminos2
(4)                 | λ
(5)  termino        → factor masFactores
(6)  masFactores1   → * factor masFactores2
(7)                 | / factor masFactores2
(8)                 | λ
(9)  factor         → CTE_ENTERA
```

Se plantea, en la fase de interpretación u optimización de código, la cuestión de calcular el valor de la expresión en un atributo `expresion.valor`. Hay que tener en cuenta que todos los operadores poseen una asociatividad a izquierdas y los factores poseen mayor precedencia que los términos. La siguiente tabla muestra ejemplos de las evaluaciones que se han de conseguir:

Sentencia	Valor de `expresion.valor`
`1-3-5`	`-7`
`1+2*3`	`7`
`16/4/4`	`1`

La complejidad del problema radica en que los dos operandos de todas las expresiones binarias se encuentran en producciones distintas. En el caso de la suma, el primer operando aparece en la primera producción, pero el segundo se encuentra en la producción 2. El cálculo del valor de la subexpresión ha de llevarse a cabo en la segunda producción, como suma del término de la primera producción y el término de ésta. Pero, ¿cómo podemos acceder en la segunda producción al valor del primer operando? Mediante el empleo de un atributo heredado:

```
(1)  masTerminos.operando1 = termino.valor
```

El atributo heredado `operando1` hace referencia al valor ya calculado, sintetizado, del término empleando como primer operando. El no terminal `masTerminos` ya conoce el

valor del primer operando y, en la segunda producción, estará en condiciones de poder calcular el valor de la subexpresión. Este valor lo devolverá con un atributo sintetizado (`masTerminos.valor`). Por tanto, la segunda regla semántica a ubicar en la primera producción será asignar este atributo sintetizado a la expresión, teniendo:

```
(1)  masTerminos.operando1 = termino.valor
     expresion.valor = masTerminos.valor
```

Este cálculo de subexpresiones, mediante la utilización de un atributo heredado que posee el valor del primer operando y un atributo sintetizado con el valor de la subexpresión, será aplicado de un modo recursivo a las producciones 5 y 9:

```
(5)  masFactores.operando1 = factor.valor
     termino.valor = masFactores.valor
(9)  factor.valor = CTE_ENTERA.valor
```

Al ser todos los operadores asociativos a izquierdas, la evaluación de toda subexpresión será el cálculo del primer operando y el segundo. Recursivamente, el resultado de la subexpresión calculada podrá ser el primer operando de otra subexpresión de la misma precedencia (por ejemplo en las sentencias `1+2+3` y `23+14-8`).

```
(2)  masTerminos2.operando1=masTerminos1.operando1+termino.valor
     masTerminos1.valor = masTerminos2.valor
```

La primera regla semántica calcula la subexpresión con sus dos términos y la convierte en el primer operando de la siguiente subexpresión. Una vez que la siguiente subexpresión haya evaluado su valor, el valor que retornamos (atributo $masTerminos_1.valor$) es el valor de la siguiente subexpresión ($masTerminos_2.valor$). En el caso de que una subexpresión no posea segundo operando (y por tanto produzca el vacío), su valor es el valor de su primer operando:

```
(4)  masTerminos1.valor = masTerminos1.operando1
```

La totalidad de las reglas semánticas de la gramática atribuida es:

```
(1)  masTerminos.operando1 = termino.valor
     expresion.valor = masTerminos.valor
(2)  masTerminos2.operando1=masTerminos1.operando1+termino.valor
     masTerminos1.valor = masTerminos2.valor
(3)  masTerminos2.operando1=masTerminos1.operando1-termino.valor
     masTerminos1.valor = masTerminos2.valor
(4)  masTerminos1.valor = masTerminos1.operando1
(5)  masFactores.operando1 = factor.valor
     termino.valor = masFactores.valor
(6)  masFactores2.operando1=masFactores1.operando1*factor.valor
     masFactores1.valor = masFactores2.valor
(7)  masFactores2.operando1=masFactores1.operando1/factor.valor
     masFactores1.valor = masFactores2.valor
(8)  masFactores1.valor = masFactores1.operando1
(9)  factor.valor = CTE_ENTERA.valor
```

□

3.4. Gramáticas Atribuidas en Análisis Semántico

Existe una ampliación de la definición de gramática atribuida presentada en § 3.3 enfocada a especificar el análisis semántico de un lenguaje de programación, es decir, a verificar la validez semántica de las sentencias aceptadas por el analizador sintáctico [Wai-

te84]. Así, la tripleta definida en § 3.3 es aumentada a la siguiente cuádrupla: GA={G,A,R,B} donde G, A y R poseen el mismo valor que en el caso anterior y

- $B = \left\{ \bigcup_{p \in P} B(p) \right\}$ es un conjunto finito de condiciones (predicados), siendo `B(p)` la conjunción de predicados que deberán satisfacer los atributos de `p` ∈ `P`, y `P` el conjunto de producciones de la gramática libre de contexto `G`.

Así, para que una sentencia pertenezca al lenguaje definido por la gramática atribuida deberá ser sintácticamente correcta (reconocida por la gramática libre de contexto), y los valores de los atributos deberán, tras crearse el árbol y evaluarse (decorarse) éste, satisfacer todas las restricciones especificadas en `B`.

Ejemplo 14: Considérese la siguiente gramática que reconoce números en base octal y decimal, posponiendo para ello el carácter `o` o `d` a cada número, respectivamente[27].

```
numero      →  digitos base
base        →  o | d
digitos     →  digitos digito
            |  digito
digito      →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Para calcular el valor real del número, debemos añadir a los no terminales `base`, `digitos` y `digito` un atributo `base` que pueda tener los valores 8 y 10. El atributo `valor` poseerá el valor del no terminal asociado, en base decimal. Añadimos, pues, las reglas semánticas para calcular este atributo:

P				**R**
(1)	numero	→	digitos base	numero.valor = digitos.valor digitos.base = base.base
(2)	base	→	**o**	base.base = 8
(3)		|	**d**	base.base = 10
(4)	$digitos_1$	→	$digitos_2$ digito	$digitos_1$.valor = digito.valor + $digitos_2$.valor * $digitos_1$.base $digitos_2$.base = $digitos_1$.base digito.base = $digitos_1$.base
(5)	digitos	→	digito	digitos.valor = digito.valor digito.base = digitos.base
(6)	digito	→	**0**	digito.valor = 0
(7)		|	**1**	digito.valor = 1
(8)		|	**2**	digito.valor = 2
(9)		|	**3**	digito.valor = 3
(10)		|	**4**	digito.valor = 4
(11)		|	**5**	digito.valor = 5
(12)		|	**6**	digito.valor = 6
(13)		|	**7**	digito.valor = 7
(14)		|	**8**	digito.valor = 8
(15)		|	**9**	digito.valor = 9

Con las reglas semánticas previas, la gramática atribuida consigue que el atributo `numero.valor` posea el valor del número en base decimal. Sin embargo, existen senten-

[27] Podría tratarse de una gramática atribuida para la fase de análisis léxico. Existen herramientas que utilizan estos tipos de gramáticas para identificar los componentes léxicos del lenguaje a procesar [ANTLR, JavaCC].

cias de este lenguaje que, aunque sean sintácticamente correctas, no lo son semánticamente. Esta condición se produce cuando un número octal posea algún dígito `8` o `9`. Esta restricción hace que las sentencias "`185o`" y "`109o`" no deban ser semánticamente correctas. Así, ampliaremos la gramática atribuida a una cuádrupla que posea, además de gramática, atributos y reglas, un conjunto de predicados:

P	B
(14) digito → **8**	digito.base > 8
(15) digito → **9**	digito.base >= 10

Nótese cómo las únicas producciones que poseen restricciones semánticas son las dos últimas, ya que es donde aparecen los dos dígitos no permitidos en los números octales. Además se ha escrito el conjunto de rutinas semánticas de modo que el no terminal `digito` posea el atributo heredado `base`, precisamente para poder comprobar que ésta sea superior a ocho.

□

Ejemplo 15: En el Ejemplo 9 se especificaba mediante una gramática libre de contexto (`G`) el siguiente lenguaje:

```
(1)  S            →  sentencias
(2)  sentencias1  →  declaracion ; sentencias2
(3)  sentencias1  →  expresion ; sentencias2
(4)  sentencias   →  λ
(5)  declaracion  →  int id
(6)  declaracion  →  float id
(7)  expresion    →  id
(8)  expresion    →  cte_entera
(9)  expresion    →  cte_real
(10) expresion1   →  ( expresion2 )
(11) expresion1   →  expresion2 + expresion3
(12) expresion1   →  expresion2 - expresion3
(13) expresion1   →  expresion2 * expresion3
(14) expresion1   →  expresion2 / expresion3
(15) expresion1   →  expresion2 = expresion3
```

Para implementar un compilador, las restricciones semánticas identificadas en el lenguaje son: comprobar que lo que está a la izquierda de una asignación es correcto (un *lvalue*); comprobar que un identificador empleado en una expresión haya sido declarado previamente; inferir el tipo de las expresiones, ratificando que las asignaciones sean correctas (nunca asignar a un entero un real); asegurar que un identificador no esté previamente declarado.

Para poder implementar la gramática atribuida, se utilizarán los siguientes atributos (`A`):

- Del terminal `id`, el atributo `id.valor` es la cadena de caracteres que representa el identificador.
- El atributo `tipo` es un carácter indicando el tipo del no terminal asociado: '`I`' para el tipo entero y '`F`' para el real.
- El atributo `declaracion.id` es un par (producto cartesiano) en el que el primer valor es una cadena de caracteres (el valor del identificador) y el segundo será un carácter (el tipo asociado a dicho identificador).

- El atributo `ids` es un contenedor asociativo o diccionario (*map*) que ofrece la gestión de una clave de tipo cadena de caracteres (identificador) y un valor carácter asociado a cada una de las claves (su tipo). El contenedor vacío es `nil`. La inserción de pares se lleva a cabo con la operación +. El acceso a un valor a partir de una clave, se obtiene con el operador `[]` –si no existe un valor asociado a la clave solicitada, devuelve `nil`.
- El atributo `expresion.lvalue` es un valor lógico indicando si la expresión puede estar a la izquierda de la asignación.

Con estos atributos, se codifican las siguientes reglas semánticas (R) para la evaluación de los mismos:

P	R
(1)	sentencias.ids = nil
(2)	declaracion.ids = $sentencias_1$.ids $sentencias_2$.ids = $sentencias_1$.ids+declaracion.id
(3)	expresion.ids = $sentencias_1$.ids $sentencias_2$.ids = $sentencias_1$.ids
(4)	
(5)	declaracion.id = (id.valor, 'I')
(6)	declaracion.id = (id.valor, 'F')
(7)	expresion.lvalue = true expresion.tipo = expresion.ids[id.valor]
(8)	expresion.lvalue = false expresion.tipo = 'I'
(9)	expresion.lvalue = false expresion.tipo = 'F'
(10)	$expresion_2$.ids = $expresion_1$.ids $expresion_1$.lvalue = $expresion_2$.lvalue $expresion_1$.tipo = $expresion_2$.tipo
(11)	$expresion_2$.ids = $expresion_1$.ids $expresion_3$.ids = $expresion_1$.ids $expresion_1$.lvalue = false $expresion_1$.tipo=mayorTipo($expresion_2$.tipo,$expresion_3$.tipo)
(12)	$expresion_2$.ids = $expresion_1$.ids $expresion_3$.ids = $expresion_1$.ids $expresion_1$.lvalue = false $expresion_1$.tipo=mayorTipo($expresion_2$.tipo,$expresion_3$.tipo)
(13)	$expresion_2$.ids = $expresion_1$.ids $expresion_3$.ids = $expresion_1$.ids $expresion_1$.lvalue = false $expresion_1$.tipo=mayorTipo($expresion_2$.tipo,$expresion_3$.tipo)
(14)	$expresion_2$.ids = $expresion_1$.ids $expresion_3$.ids = $expresion_1$.ids $expresion_1$.lvalue = false $expresion_1$.tipo=mayorTipo($expresion_2$.tipo,$expresion_3$.tipo)
(15)	$expresion_2$.ids = $expresion_1$.ids $expresion_3$.ids = $expresion_1$.ids $expresion_1$.lvalue = $expresion_2$.lvalue $expresion_1$.tipo=$expresion_2$.tipo

```
char mayorTipo(char t1,char t2) {
 if ( t1 == 'F' || t2 == 'F' ) return 'F';
 return 'I';
}
```

Una vez especificado en modo en el que se deben calcular cada uno de los atributos, limitaremos la sintaxis del lenguaje mediante el último elemento de la gramática atribuida: un conjunto de predicados asociados a las producciones de la gramática (B), que establece las restricciones semánticas del lenguaje.

P	B
(5)	`declaracion.ids[id.valor] == nil`
(6)	`declaracion.ids[id.valor] == nil`
(7)	`expresion.ids[id.valor] != nil`
(15)	$expresion_2$`.lvalue` $expresion_2$`.tipo!='I' || expresion`$_3$`.tipo!='F'`

En este ejemplo se aprecia la principal diferencia entre gramática atribuida y definición dirigidas por sintaxis. Las primeras, al no poder generar efectos colaterales, han de representar la tabla de símbolos como un atributo (`ids`) de varios símbolos no terminales. En las definiciones dirigidas por sintaxis, la tabla de símbolos es una estructura de datos global a la que se accede desde las reglas semánticas. Adicionalmente, el mecanismo de detección de errores se desarrolla con código adicional dentro de las reglas semánticas, en lugar de especificarlo aparte en un conjunto de predicados (B). Las definiciones dirigidas por sintaxis son menos formales, pero más pragmáticas.

□

4 Tipos de Gramáticas Atribuidas

Como hemos visto, las gramáticas atribuidas ofrecen un modo de decorar o anotar un árbol sintáctico (concreto o abstracto) de un modo declarativo, sin identificar explícitamente el modo en el que deban ejecutarse las reglas semánticas –en el caso de que realmente se puedan ejecutar. En función de las propiedades que cumpla una gramática atribuida, podremos decir si se puede evaluar cualquier árbol asociado a un programa de entrada e incluso podremos determinar un orden específico de ejecución de las reglas semánticas.

Existen multitud de trabajos, bibliografía e investigación relativa a gramáticas atribuidas. De este modo, los distintos tipos de gramáticas aquí presentados son lo más representativos, existiendo otros tipos que no mencionamos.

4.1. Atributos Calculados en una Producción

Los atributos calculados en una producción p de la gramática libre de contexto asociada a una gramática atribuida son los que cumplan la siguiente condición:

$$AC(p) = \{a / a := f(a_1, a_2, a_3 ... a_k) \in R(p); a, a_k \in A(X), 1 \le i \le k; X \in (VT \cup VN); p \in P\}$$

Los atributos calculados asociados a una producción son, pues, aquéllos cuyo valor es calculado en una regla semántica asociada a dicha producción.

4.2. Gramática Atribuida Completa

Una gramática atribuida se dice que es completa si satisface las siguientes condiciones [Waite84]:

- $\forall X \in VN(G), AS(X) \bigcap AH(X) = \varnothing$
- $\forall X \in VN(G), AS(X) \bigcup AH(X) = A(X)$
- $\forall p \in P, p : X \to \alpha, AS(X) \subseteq AC(p)$
- $\forall p \in P, p : Y \to \alpha X \beta, AH(X) \subseteq AC(p)$

La primera condición obliga a que un mismo atributo de la gramática no pueda ser sintetizado al mismo tiempo que heredado. En el segundo caso, la restricción impuesta es que todo atributo ha de sintetizarse o heredarse, es decir, no podrá existir un atributo al que nunca se le asigne valor alguno.

Las dos últimas condiciones buscan un mismo objetivo: que un a atributo sintetizado o heredado siempre se le asigne un valor, en toda producción en la aparezca en la parte izquierda o derecha de la misma, respectivamente. Si un atributo es sintetizado en una producción, en el resto de producciones en el que el no terminal asociado aparezca en la parte izquierda, deberá ser calculado. La misma condición, aplicada a las partes derechas de las producciones, deberá satisfacerse en el caso de los atributos heredados.

El significado real de que una gramática atribuida sea completa es que haya sido escrita de un modo correcto. Pongamos un ejemplo con el caso de gramáticas libres de contexto. Este tipo de gramáticas se utiliza para describir lenguajes sintácticamente. Sin embargo, podemos escribir una gramática *sucia*, en la que existan símbolos no terminales que no tengan una producción asociada (símbolos muertos) o producciones cuyo símbolo no terminal de la parte izquierda nunca se emplee en otra producción (símbolos inaccesibles). Así, aunque una gramática libre de contexto es un mecanismo para expresar la sintaxis de un lenguaje de programación, si escribimos una gramática *sucia* no estaremos describiendo ningún lenguaje.

La misma problemática puede surgir cuando escribimos gramáticas atribuidas. Podría darse el caso de que, siguiendo una notación específica, tratásemos de describir atributos de un programa mediante una gramática atribuida. Si dicha gramática no es completa, no estaremos describiendo realmente evaluaciones de los atributos, puesto que cu cálculo no se podría llevar a cabo ante todos los programas de entrada. Por ello, la comprobación de que una gramática atribuida sea completa es una verificación de que se esté expresando correctamente la asignación de valores a los atributos.

Ejemplo 16: Dada la gramática atribuida del Ejemplo 13, en la que G era:

```
(1) expresion        → termino masTerminos
(2) masTerminos1     → + termino masTerminos2
(3)                  | - termino masTerminos2
(4)                  | λ
(5) termino          → factor masFactores
(6) masFactores1     → * factor masFactores2
(7)                  | / factor masFactores2
(8)                  | λ
(9) factor           → CTE_ENTERA
```

y sus reglas semánticas:

```
(1) masTerminos.operando1 = termino.valor
    expresion.valor = masTerminos.valor
(2) masTerminos2.operando1=masTerminos1.operando1+termino.valor
    masTerminos1.valor = masTerminos2.valor
(3) masTerminos2.operando1=masTerminos1.operando1-termino.valor
    masTerminos1.valor = masTerminos2.valor
(4) masTerminos1.valor = masTerminos1.operando1
(5) masFactores.operando1 = factor.valor
    termino.valor = masFactores.valor
(6) masFactores2.operando1=masFactores1.operando1*factor.valor
    masFactores1.valor = masFactores2.valor
(7) masFactores2.operando1=masFactores1.operando1/factor.valor
    masFactores1.valor = masFactores2.valor
(8) masFactores1.valor = masFactores1.operando1
(9) factor.valor = CTE_ENTERA.valor
```

Los atributos calculados y, para cada caso, si son heredados o sintetizados en una producción se muestra en la siguiente tabla:

Producción	Atributos Calculados	Atributo Heredado o Sintetizado
(1)	masTerminos.operando1	Heredado
	expresion.valor	Sintetizado

Producción	Atributos Calculados	Atributo Heredado o Sintetizado
(2)	masTerminos.operando1	Heredado
	masTerminos.valor	Sintetizado
(3)	masTerminos.operando1	Heredado
	masTerminos.valor	Sintetizado
(4)	masTerminos.valor	Sintetizado
(5)	masFactores.operando1	Heredado
	termino.valor	Sintetizado
(6)	masFactores.operando1	Heredado
	masFactores.valor	Sintetizado
(7)	masFactores.operando1	Heredado
	masFactores.valor	Sintetizado
(8)	masFactores.valor	Sintetizado
(9)	factor.valor	Sintetizado

Vemos en la tabla anterior que los atributos sintetizados (`expresion.valor`, `masTerminos.valor`, `termino.valor`, `masFactores.valor`, `factor.valor` y `CTE_ENTERA.valor`) no son heredados para ninguna producción. Del mismo modo, los heredados (`masTerminos.operando1` y `masFactores.operando1`) nunca se sintetizan. Por ello, se satisface la primera condición de gramática completa.

La comprobación de la segunda condición para que la gramática sea completa, es buscar algún atributo que no se haya calculado y se emplee, es decir, algún atributo que no pertenezca a la tabla anterior. Si existiere uno sin ser sintetizado ni heredado, la gramática no sería completa. El único atributo que no se calcula en ninguna producción es `CTE_ENTERA.valor` que, por definición, es sintetizado –por el analizador léxico.

La tercera condición obliga a que, para todas las producciones, los atributos sintetizados del no terminal de la izquierda se calculen. Esta comprobación se puede llevar a cabo mediante una tabla:

Producción	No Terminal Izquierda	Atributos Sintetizados	¿Calculados?
(1)	expresion	expresion.valor	Sí
(2)	masTerminos	masTerminos.valor	Sí
(3)	masTerminos	masTerminos.valor	Sí
(4)	masTerminos	masTerminos.valor	Sí
(5)	termino	termino.valor	Sí
(6)	masFactores	masFactores.valor	Sí
(7)	masFactores	masFactores.valor	Sí
(8)	masFactores	masFactores.valor	Sí
(9)	factor	factor.valor	Sí

Finalmente, la última restricción indica que, para todas las producciones, los atributos heredados de los símbolos gramaticales de la parte derecha deberán ser calculados:

Producción	No Terminal Derecha	Atributos Heredados	¿Calculados?
(1)	termino		
	masTerminos	masTerminos.operando1	Sí
(2)	termino		
	masTerminos	masTerminos.operando1	Sí
(3)	termino		
	masTerminos	masTerminos.operando1	Sí

Producción	No Terminal Derecha	Atributos Heredados	¿Calculados?
(4)			
(5)	factor		
	masFactores	masFactores.operando1	Sí
(6)	factor		
	masFactores	masFactores.operando1	Sí
(7)	factor		
	masFactores	masFactores.operando1	Sí
(8)			
(9)			

Se puede concluir, pues, que la gramática atribuida anterior es completa.

□

Ejemplo 17: Supóngase que para la misma gramática que hemos presentado en el ejemplo anterior, las reglas semánticas son:

```
(1) masTerminos.operando1 = termino.valor
    expresion.valor = masTerminos.valor
(2) masTerminos2.operando1=masTerminos1.operando1+termino.valor
    masTerminos1.valor = masTerminos2.valor
(3) masTerminos2.operando1=masTerminos1.operando1-termino.valor
    masTerminos1.valor = masTerminos2.valor
(4)
(5) masFactores.operando1 = factor.valor
    termino.valor = masFactores.valor
(6) masFactores2.operando1=masFactores1.operando1*factor.valor
    masFactores1.valor = masFactores2.valor
(7) masFactores2.operando1=masFactores1.operando1/factor.valor
    masFactores1.valor = masFactores2.valor
(8) masFactores1.valor = masFactores1.operando1
(9) factor.valor = CTE_ENTERA.valor
```

La gramática anterior no es completa, ya que en la producción 4 el no terminal de la izquierda posee un atributo sintetizado (`masTerminos.valor`) que no se calcula. No se cumple, por tanto, la tercera condición de gramática completa.

Nótese cómo el resultado de que la gramática no sea completa implica que ante determinados programas de entrada, no se puedan calcular los atributos del árbol. Así, por ejemplo, el programa de entrada 33, generaría el siguiente árbol anotado:

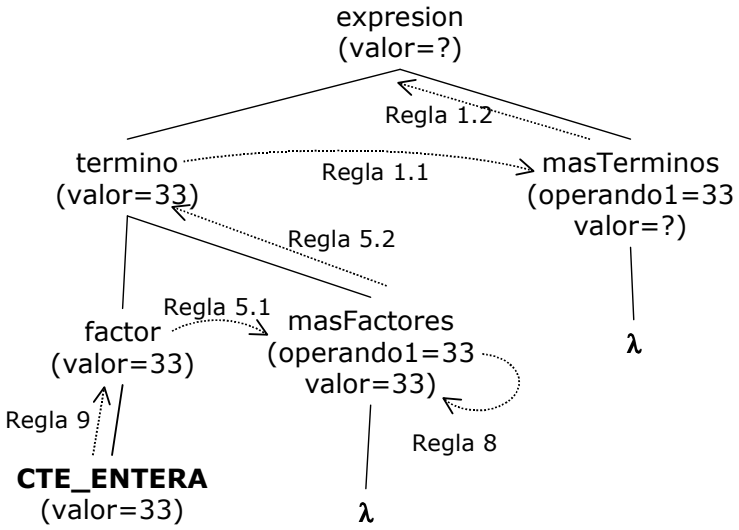

Figura 9: Árbol con anotaciones para el programa de entrada 33.

El hecho de que un atributo sea sintetizado, hace que su valor se calcule cuando el no terminal se encuentra en su parte izquierda. Al no haberse calculado en la cuarta producción, hace que este valor no esté disponible en la segunda regla de la primera producción. Así, el valor que obtendría la expresión sería desconocido. Para evitar estos errores surge el concepto de gramática completa, obligando, entre otras cosas, a que todo atributo sintetizado sea calculado cuando su no terminal aparezca a la izquierda de una producción.

□

Ejemplo 18: Dada la siguiente gramática atribuida:

(1)	S	→	A G	S.s = A.s + G.s
(2)	A	→	F G	G.h = F.s
				A.s = G.s
(3)	F	→	**TOKEN_A**	F.s = **TOKEN_A**.s
(4)		|	**TOKEN_B**	F.s = **TOKEN_B**.s
(5)	G_1	→	**TOKEN_C**	G_1.s = G_1.h + **TOKEN_C**.s
(6)		|	G_2 **TOKEN_D**	G_2.h = G_1.h
				G_1.s = G_1.h + **TOKEN_D**.s
				G_2.s = G_1.h

No es completa por dos motivos:

- El atributo G.s es sintetizado en la producción 5 y heredado en la producción 6.
- El atributo heredado G.h no se calcula en la primera producción, apareciendo el símbolo gramatical G en la parte derecha de la misma.

□

4.3. Gramática Atribuida Bien Definida

Una gramática es bien definida[28] (también denominada *no circular*) si para todas las sentencias del lenguaje generado por su gramática libre de contexto, es posible calcular los valores de los atributos de todos sus símbolos gramaticales [Waite84]. Este proceso de de-

[28] *Well-defined aatribute grammar.*

coración o anotación del árbol sintáctico ante un programa de entrada se denomina **evaluación de la gramática** [Aho90].

Para que una gramática sea bien definida (no circular), deberá ser posible encontrar un modo de calcular la gramática ante cualquier programa de entrada. Esto implica saber en qué orden se han de ejecutar las reglas semánticas ante cualquier programa, para poder evaluar la gramática[29]. Así, una gramática es completa si asigna correctamente valores a sus atributos y será bien definida si, además, es posible encontrar un orden de ejecución de sus reglas semánticas ante cualquier programa de entrada. Por lo tanto, toda gramática atribuida bien definida es completa.

Existe un algoritmo para calcular si una gramática está bien definida [Jazayeri75]. Su principal inconveniente es que posee una complejidad computacional exponencial.

Ejemplo 19: La siguiente gramática atribuida:

```
<S>     →  <A> <B>     A.a = B.b
                       B.b = A.a + 1
<A>     →  A
<B>     →  B
```

es una gramática atribuida completa, puesto que los dos únicos atributos `A.a` y `B.b` son heredados y en todas las producciones en las que `A` y `B` aparecen en la parte derecha, éstos son calculados. Sin embargo no está bien definida puesto que, dado el único programa de entrada válido `AB`, es imposible evaluar la gramática –calcular el valor de los dos atributos. □

4.4. Gramáticas S-Atribuidas

Una gramática es S-atribuida si todos sus atributos son sintetizados. Esta característica específica de determinadas gramáticas atribuidas es empleado para preestablecer un orden de ejecución de sus rutinas semánticas –como veremos en § 5.2.

La gramática del Ejemplo 11 es una gramática S-atribuidas puesto que únicamente posee atributos sintetizados.

4.5. Gramáticas L-Atribuidas

Una gramática es L-atribuida si para cada producción de la forma:

$$X_0 \rightarrow X_1 X_2 ... X_n, X_0 \in VN, X_i \in VT \cup VN, 1 \leq i \leq n$$

todos los atributos heredados de X_j, $1 \leq j \leq n$, son calculados en función de:

- Los atributos de los símbolos $X_1, X_2, ..., X_{j-1}$
- Los atributos heredados de X_0

La definición anterior indica que, para que una gramática sea L-atribuida, los atributos heredados de la parte derecha de toda producción han de calcularse en función de los heredados del no terminal de la izquierda y/o en función de los atributos de los símbolos gramaticales de la parte derecha de la producción, ubicados a su izquierda. En el caso de

[29] Estudiaremos en mayor profundidad el cálculo de orden de evaluación de las reglas semánticas de una gramática atribuida en § 1.

que la condición se satisfaga para una definición dirigida por sintaxis, se dice que ésta es **con atributos por la izquierda** [Aho90].

Si nos fijamos en la definición de gramática L-atribuida, nos daremos cuenta que especifica una restricción de los atributos heredados de la gramática. Si una gramática es S-atribuida, no posee atributo heredado alguno y, por tanto, es también L-atribuida: toda gramática S-atribuida es también L-atribuida. Vemos pues cómo la expresividad de las segundas es superior a las de las primeras –ya que se trata de un subconjunto.

Ejemplo 20: La gramática atribuida completa presentada en el Ejemplo 13 y en el Ejemplo 16, es una gramática L-atribuida puesto que sus atributos heredados satisfacen la restricción indicada:

P	Atributo Heredado Calculado	Depende de	Restricción
(1)	masTerminos.operando1	termino.valor	1ª
(2)	$masTerminos_2$.operando1	$masTerminos_1$.operando1	2ª
		termino.valor	1ª
(3)	$masTerminos_2$.operando1	$masTerminos_1$.operando1	2ª
		termino.valor	1ª
(5)	masFactores.operando1	factor.valor	1ª
(6)	masFactores.operando1	$masFactores_1$.operando1	2ª
		factor.valor	1ª
(7)	masFactores.operando1	$masFactores_1$.operando1	2ª
		factor.valor	1ª

En la tabla anterior se indican los atributos heredados calculados en cada una de las producciones. La tercera columna indica los valores empleados para realizar dicho cálculo. Por último, la cuarta columna muestra si el atributo empleado para el cálculo (el de la tercera columna) es un heredado del no terminal de la izquierda (1ª restricción) o un atributo de un símbolo gramatical de la parte derecha, situado a la izquierda del atributo calculado (2ª restricción).

Obviamente, al tener la gramática atributos heredados, no es S-atribuida.

□

Ejemplo 21: La gramática atribuida del Ejemplo 14:

P				**R**
(1)	numero	→	digitos base	numero.valor = digitos.valor digitos.base = base.base
(2)	base	→	**o**	base.base = 8
(3)		|	**d**	base.base = 10
(4)	$digitos_1$	→	$digitos_2$ digito	$digitos_1$.valor = digito.valor + $digitos_2$.valor * $digitos_1$.base $digitos_2$.base = $digitos_1$.base digito.base = $digitos_1$.base
(5)	digitos	→	digito	digitos.valor = digito.valor digito.base = digitos.base
(6)	digito	→	**0**	digito.valor = 0
(7)		|	**1**	digito.valor = 1
(8)		|	**2**	digito.valor = 2
(9)		|	**3**	digito.valor = 3
(10)		|	**4**	digito.valor = 4
(11)		|	**5**	digito.valor = 5

P			R
(12)	|	**6**	digito.valor = 6
(13)	|	**7**	digito.valor = 7
(14)	|	**8**	digito.valor = 8
(15)	|	**9**	digito.valor = 9

- No es una gramática atribuida S-atribuida porque los atributos `digitos.base` y `digito.base` son atributos heredados.
- No es una gramática L-atribuida puesto que en la primera producción `digitos.base` se calcula en función de `base.base`, y el símbolo gramatical del segundo atributo se encuentra a la derecha del primero, en la producción.
- Si analizados las condiciones que se han de satisfacer para que la gramática sea completa, nos daremos cuenta de que los cumple (§ 4).

□

4.6. Traducción de Gramáticas S-Atribuidas a L-Atribuidas

Donald E. Knuth demostró cómo toda gramática L-atribuida podía ser traducida a otra gramática S-atribuida que reconozca el mismo lenguaje, mediante un esquema de traducción de atributos heredados a atributos sintetizados.

Ejemplo 22: En el Ejemplo 12 empleamos la siguiente definición dirigida por sintaxis con atributos por la izquierda, para insertar en una tabla de símbolos los identificadores declarados en un determinado lenguaje de programación:

P			R
declaracion	→	tipo variables ;	variables.tipo = tipo.tipo
tipo	→	**int**	tipo.tipo = 'I'
tipo	→	**float**	tipo.tipo = 'F'
$variables_1$	→	**id ,** $variables_2$	ts.insertar(id.valor,$variables_1$.tipo) $variables_2$.tipo = $variables_1$.tipo
variables	→	**id**	ts.insertar(id.valor,variables.tipo)

Esta definición dirigida por sintaxis se puede traducir a otra equivalente (que reconozca el mismo lenguaje e inserte la misma información en la tabla de símbolos) tan solo con el empleo de atributos sintetizados:

P			R
declaracion	→	variables **id ;**	ts.insertar(id.valor,variables.tipo)
$variables_1$	→	$variables_2$ **id ,**	$variables_1$.tipo = $variables_2$.tipo ts.insertar(id.valor,$variables_2$.tipo)
	|	tipo	$variables_1$.tipo = tipo.tipo
tipo	→	**int**	tipo.tipo = 'I'
	|	**float**	tipo.tipo = 'F'

□

Como hemos visto en el ejemplo anterior, el resultado de traducir una gramática de L a S-atribuida es que las producciones y las reglas semánticas de la nueva gramática serán más complejas y difíciles de entender por el desarrollador del compilador. No es por tanto aconsejable esta traducción, a no ser que sea estrictamente necesario. Este caso se podría

dar si, por ejemplo, no tuviésemos un evaluador de gramáticas L-atribuidas, sino uno que únicamente reconociese gramáticas S-atribuidas –como es el caso de yacc/bison [Mason92].

En ocasiones sucede que la estructura y evaluación de una gramática atribuida parece demasiado compleja. Es común que se esté dando el caso que no se haya escrito la gramática de un modo adecuado, haciendo demasiado uso de atributos sintetizados. La reescritura de la misma empleando más atributos heredados suele reducir la complejidad de la gramática, así como su evaluación.

5 Evaluación de Gramáticas Atribuidas

Como hemos mencionado en la definición de gramática atribuida (§ 3.3), el orden de evaluación de sus atributos ante una sentencia de entrada no es llevada a cabo de un modo imperativo, sino declarativo. En una gramática atribuida (o definición dirigida por sintaxis) no se especifica el modo en el que se han de calcular los atributos, sino los valores que éstos deben tomar en función de otros. La evaluación de los mismos se llevará a cabo posteriormente mediante una herramienta evaluadora de gramáticas atribuidas, un determinado recorrido del AST o empleando un mecanismo más dirigido hacia la sintaxis del lenguaje, conocido como esquema de traducción.

En este punto analizaremos cómo se debe llevar a cabo la evaluación de los atributos de una gramática atribuida, ya bien sea de un modo automático gracias a la utilización de una herramienta, o mediante alguna traducción de ésta a código por parte del programador.

5.1. Grafo de Dependencias

El tercer valor que constituía una gramática atribuida era un conjunto de ecuaciones de atributos, también llamadas reglas semánticas (§ 3.2). En ellas se establecen las relaciones entre los valores de los atributos definidos en la gramática atribuida asociada, mediante ecuaciones del modo: $a := f(a_1, a_2, a_3 ... a_k)$, siendo a y $a_i, 1 \leq i \leq k$ atributos de los símbolos gramaticales de una producción asociada.

De la ecuación general anterior, se deduce que el valor del atributo a **depende de** los valores de los atributos $a_i, 1 \leq i \leq k$, implicando que sea necesario que se evalúe la regla semántica para a en esa producción después de haberse evaluado las reglas semánticas que definan los valores de $a_i, 1 \leq i \leq k$ [Aho90].

Dada una gramática atribuida, cada producción tendrá asociada un **grafo de dependencias locales**[30] en el que se establecen, mediante un grafo dirigido, las dependencias existentes entre todos los atributos que aparecen en la producción [Wilhelm95]. Dicho grafo tendrá un nodo por cada uno de los atributos que aparezcan en las reglas semánticas asociadas a la producción. Aparecerá una arista desde un nodo $a_i, 1 \leq i \leq k$ hasta otro a, siempre que a dependa de $a_i, 1 \leq i \leq k$; es decir, siempre que exista una regla semántica $a := f(a_1, a_2, a_3 ... a_k)$ asociada a dicha producción, indicando que es necesario calcular $a_i, 1 \leq i \leq k$ previamente a la regla semántica.

En la representación de un grafo de dependencias locales de una producción, cada nodo que representa un atributo de un símbolo gramatical X estará agrupado con el resto de atributos (nodos) asociados al mismo símbolo gramatical. Así, el grafo estará estructura-

[30] *Production-local dependency graph.*

do en base al árbol sintáctico y, por tanto, se suele representar superpuesto al subárbol sintáctico correspondiente a la producción.

Ejemplo 23. En el Ejemplo 15 presentamos una gramática atribuida que comprobaba la validez semántica de las expresiones de un pequeño lenguaje de programación. Para la última producción teníamos las siguientes reglas semánticas asociadas:

P	**R**
$expresion_1 \rightarrow$ $expresion_2$ **=** $expresion_3$	$expresion_2.ids = expresion_1.ids$ $expresion_3.ids = expresion_1.ids$ $expresion_1.lvalue = expresion_2.lvalue$ $expresion_1.tipo = expresion_2.tipo$

Para la producción anterior, el grafo de dependencias locales es el mostrado en la Figura 10:

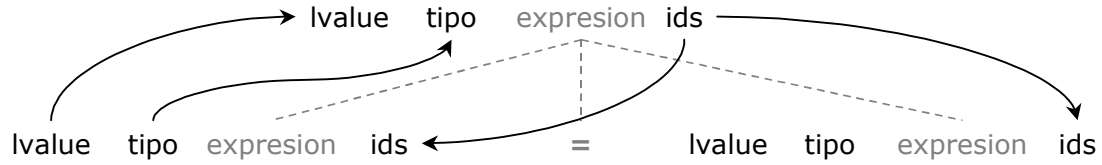

Figura 10: Grafo de dependencias locales para la producción que define las asignaciones.

Nótese como las dependencias de los atributos, explicitadas en las reglas semánticas, están representadas mediante aristas dirigidas. Además, cada atributo de un símbolo gramatical está ubicado al lado de éste, superponiendo el grafo de dependencias al subárbol sintáctico (en tono gris) correspondiente a la producción asociada.

□

Dado un programa de entrada perteneciente al lenguaje descrito por la gramática atribuida, su **grafo de dependencias** es la unión de los grafos de dependencias locales de las producciones empleadas para reconocer el programa, representando cada nodo del árbol sintáctico creado [Louden97].

Ejemplo 24. Empleando la misma gramática atribuida del Ejemplo 15, el siguiente programa:

```
int i;
float r;
r=i*0.5;
```

Satisface las restricciones sintácticas y semánticas del lenguaje definido por dicha gramática atribuida. Para ello, se definieron unos atributos y un conjunto de reglas semánticas que definía la dependencia entre ellos.

Haciendo uso de la gramática libre de contexto y de las reglas semánticas definidas sobre ésta, podremos obtener el siguiente grafo de dependencias:

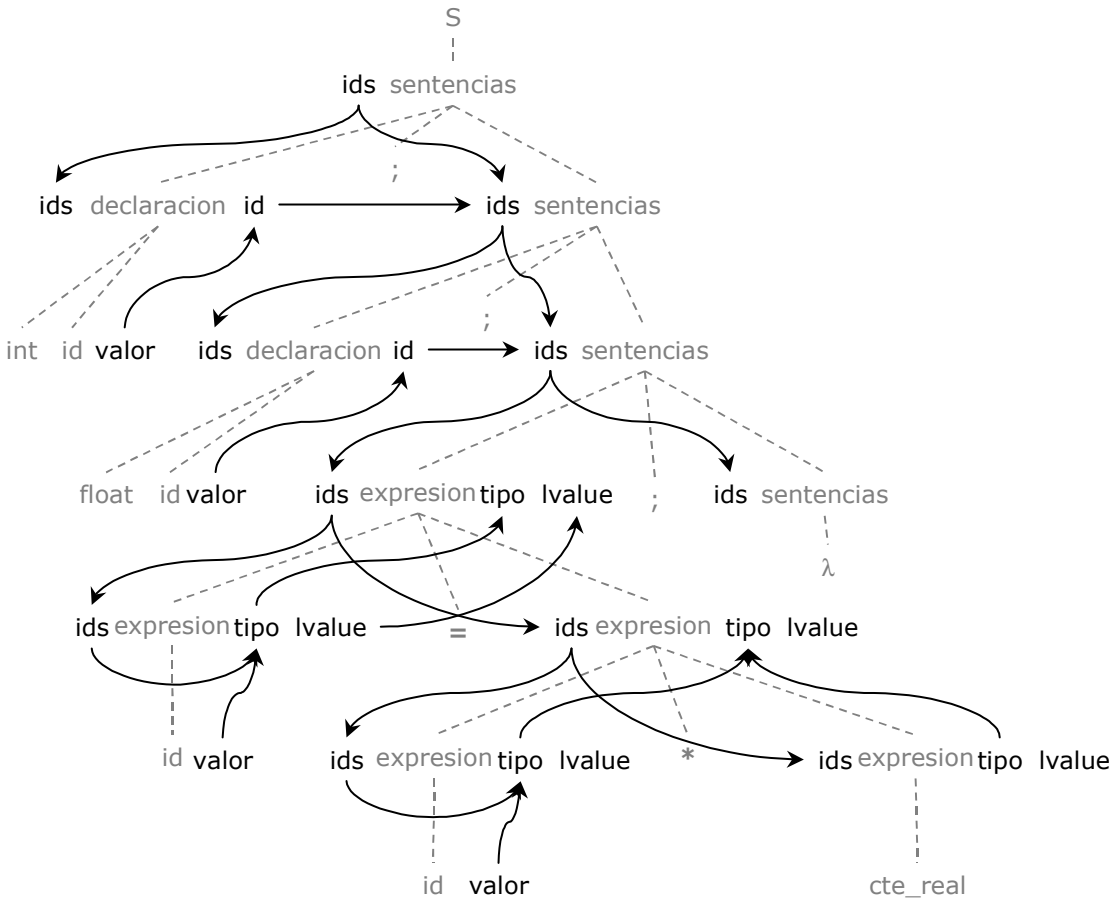

Figura 11: Grafo de dependencias para el cálculo de atributos, ante el programa especificado.

Sobre el árbol sintáctico se ha ubicado todo atributo asociado a los símbolos gramaticales. En cada una de las derivaciones del árbol (agrupación de un nodo y sus hijos) generada por la correspondiente producción de la gramática, se representa el grafo de dependencias locales. El grafo resultante es el grafo de dependencias obtenido para el programa de entrada.

A modo de ejemplo, centrémonos en la producción séptima (`expresion → id`) y sus reglas semánticas asociadas. Esta producción tiene, en el programa de ejemplo, dos derivaciones y, por tanto, dos subárboles asociados –los dos en los que aparece `id` como nodo hoja y `expresion` como su nodo padre. Las dos reglas semánticas asociadas a esta producción son:

```
expresion.lvalue = true
expresion.tipo = expresion.ids[id.valor]
```

La primera no denota ninguna dependencia, puesto que puede ejecutarse sin requerir la ejecución de ninguna otra regla. Sin embargo, la segunda revela una dependencia del atributo `expresion.tipo` respecto a los dos atributos `expresion.ids` e `id.valor`. Por este motivo, se aprecia en el grafo de dependencia –para los dos subárboles que representan las dos derivaciones– una arista dirigida que representa las dependencias indicadas.

□

Si nos encontramos describiendo una definición dirigida por sintaxis, puede darse el caso de que una regla semántica sea la invocación a un procedimiento o a algún método que no devuelva ningún valor y que, por tanto, dicha regla no sea empleada para asignar un valor a un atributo. En este caso, para calcular el grafo de dependencias debe inventarse un

atributo ficticio para el no terminal del lado izquierdo de la producción, haciendo que dependa de los parámetros pasados al procedimiento.

Ejemplo 25. Recordemos la definición dirigida por sintaxis presentada en el Ejemplo 12:

P			**R**
`declaracion`	→	`tipo variables ;`	`variables.tipo = tipo.tipo`
`tipo`	→	**`int`**	`tipo.tipo = 'I'`
`tipo`	→	**`float`**	`tipo.tipo = 'F'`
$variables_1$	→	**`id ,`** $variables_2$	`ts.insertar(id.valor,`$variables_1$`.tipo)` $variables_2$`.tipo =` $variables_1$`.tipo`
`variables`	→	**`id`**	`ts.insertar(id.valor,variables.tipo)`

La problemática comentada surge en las reglas semánticas en las que se inserta un identificador en la tabla de símbolos. Al permitir los efectos colaterales de inserción de datos en una estructura global en lugar de representar ésta mediante atributos de la gramática, nos encontramos con la problemática de que, para ejecutar la rutina de la última producción:

```
ts.insertar(id.valor,variables.tipo)
```

es necesario que previamente se hayan hallado los valores de `id.valor` y `variables.tipo`. Sin embargo, al no asignar éstos a ningún otro atributo, no sabemos a quién otorgar esta dependencia. Alfred Aho resuelve esta problemática estableciendo la dependencia con un atributo ficticio del no terminal de la parte izquierda (nodo padre). Así, podremos representar el grafo de dependencias, para la entrada `float a,b;`, del siguiente modo:

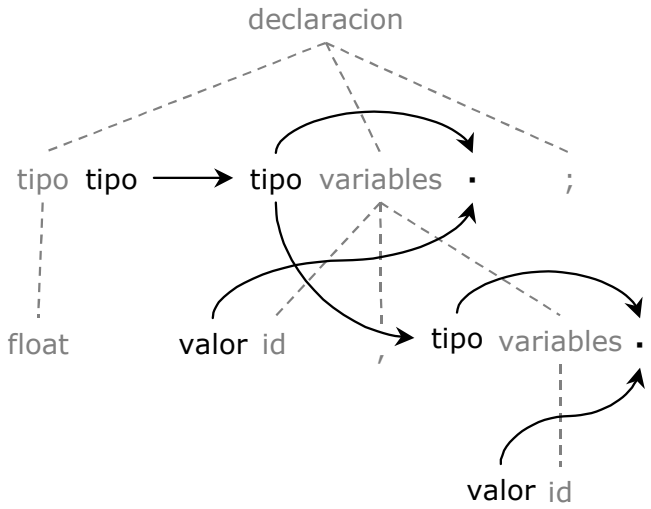

Figura 12: Grafo de dependencias para la entrada "float a,b;".

El atributo inventado, ha sido representado con un punto.

□

Ordenamiento topológico del grafo de dependencias

El grafo de dependencias de un programa de entrada para una gramática atribuida dada (o definición dirigida por sintaxis) representa el conjunto de restricciones relativas al orden que un algoritmo de evaluación de la gramática ha de satisfacer para calcular todos los atributos. De este modo, cualquier algoritmo de evaluación de la gramática deberá cal-

cular un nodo (atributo) del grafo de dependencias antes de evaluar los atributos dependientes de éste. Una ordenación de los nodos de un grafo de dependencias que cumpla dicha restricción se denomina ordenamiento topológico del grafo.

Un **ordenamiento topológico** de un grafo de dependencias es todo ordenamiento $m_1, m_2, ..., m_k$ de los nodos del grafo tal que las aristas vayan desde los nodos que aparecen primero en el orden a los que parecen más tarde; es decir, si $m_i \rightarrow m_j$ es una arista desde m_i a m_j, entonces m_i aparece antes que m_j en el orden topológico [Aho90]. Todo ordenamiento topológico de un grafo de dependencias establece un orden válido en el que se pueden evaluar las reglas semánticas asociadas a los nodos del árbol sintáctico.

La condición necesaria y suficiente para que exista al menos un ordenamiento topológico para un grafo de dependencias es que el grafo sea acíclico. Este tipo de grafos recibe el nombre de *grafos acíclicos dirigidos* (DAG, *Directed Acyclic Graphs*). Una gramática atribuida no circular (bien definida) es aquella para la que cualquier posible grafo de dependencias es acíclico –de ahí la importancia de crear gramáticas atribuidas no circulares, para que pueda evaluarse ésta ante cualquier programa de entrada.

Ejemplo 26. A raíz del grafo de dependencias del Ejemplo 25, podemos establecer un ordenamiento topológico de los nodos del mismo. Para ello, numeramos los atributos conforme a la dependencia que puedan tener entre sí:

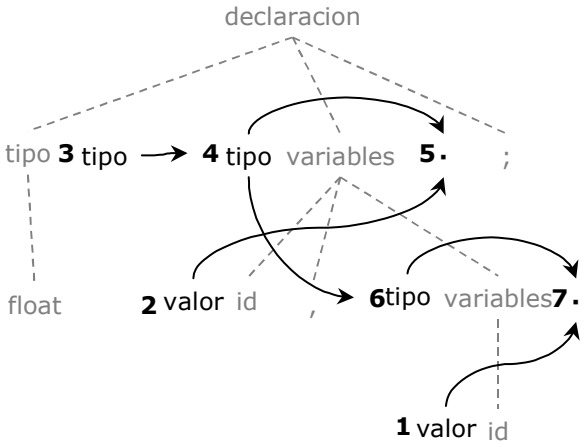

Figura 13: Numeración de los nodos del grafo de dependencias, estableciendo un ordenamiento topológico sobre el mismo.

Los números iniciales son asignados a los nodos que no poseen ninguna dependencia de otro nodo. Una vez hecho esto, se numeran los nodos consecutivamente conforme dependan de otros nodos que posean un valor menor. Pueden existir distintas numeraciones.

Un ordenamiento topológico de los nodos del grafo establece un orden válido en el que se pueden evaluar las reglas semánticas asociadas a los nodos del árbol. De este modo, el orden de ejecución de las reglas semánticas será el indicado por los nodos del grafo. Si un nodo del grafo corresponde a un símbolo terminal, no habrá que calcular su atributo puesto que éste ya fue sintetizado por el analizador léxico.

Número de nodo	Atributo	Regla semántica donde se calcula	Producción
1	`id.valor`	Analizador Léxico	Léxico

Número de nodo	Atributo	Regla semántica donde se calcula	Producción
2	id.valor	Analizador Léxico	Léxico
3	tipo.tipo	tipo.tipo = 'F'	3
4	variables.tipo	variables.tipo = tipo.tipo	1
5	.	ts.insertar(id.valor, $variables_1$.tipo)	4
6	varirables.tipo	$variables_2$.tipo = $variables_1$.tipo	4
7	·	ts.insertar(id.valor, variables.tipo)	5

Puede, pues, convertirse el programa declarativo especificado con la definición dirigida por sintaxis en un programa imperativo que, empleando el árbol sintáctico como principal estructura de datos, ejecute la siguiente secuencia de sentencias[31]:

```
tipo.tipo3 = 'F'
variables.tipo4 = tipo.tipo3
ts.insertar(id.valor2,variables.tipo4)
variables.tipo6 = variables.tipo4
ts.insertar(id.valor1,variables.tipo6)
```

□

Ejemplo 27. En el Ejemplo 14 se presentó la siguiente gramática atribuida que evaluaba el valor y la corrección semántica de números decimales y octales:

P				R
(1)	numero	→	digitos base	numero.valor = digitos.valor digitos.base = base.base
(2)	base	→	**o**	base.base = 8
(3)		|	**d**	base.base = 10
(4)	$digitos_1$	→	$digitos_2$ digito	$digitos_1$.valor = digito.valor + $digitos_2$.valor * $digitos_1$.base $digitos_2$.base = $digitos_1$.base digito.base = $digitos_1$.base
(5)	digitos	→	digito	digitos.valor = digito.valor digito.base = digitos.base
(6)	digito	→	**0**	digito.valor = 0
(7)		|	**1**	digito.valor = 1
(8)		|	**2**	digito.valor = 2
(9)		|	**3**	digito.valor = 3
(10)		|	**4**	digito.valor = 4
(11)		|	**5**	digito.valor = 5
(12)		|	**6**	digito.valor = 6
(13)		|	**7**	digito.valor = 7
(14)		|	**8**	digito.valor = 8
(15)		|	**9**	digito.valor = 9

Cabría preguntarnos cuál sería un orden de evaluación de los atributos ante la entrada de la sentencia `71o`. Su grafo de dependencias con un ordenamiento topológico válido del mismo es:

[31] A los atributos de los nodos se les ha añadido el subíndice de su ordenación topológica, para evitar ambigüedades entres las distintas ocurrencias (instancias) de cada atributo.

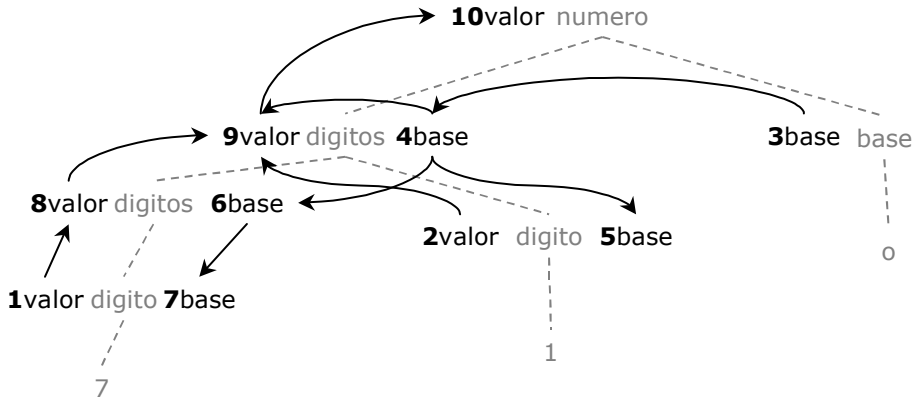

Figura 14: Grafo de dependencias para la entrada "71o".

Tras el ordenamiento identificado con la numeración de los nodos del grafo, podemos concluir que una secuencia válida de ejecución de las reglas semánticas es la siguiente:

```
digito.valor1 = 7
digito.valor2 = 1
base.base3 = 8
base.base4 = base.base3
digito.base5 = base.base4
digitos.base6 = base.base4
digito.base7 = digitos.base6
digitos.valor8 = digito.valor1
digitos.valor9 = digito.valor2+digitos.valor8*digitos.base6
numero.valor10 = digitos.valor9
```

Nótese cómo, tras la ejecución de las sentencias previas, el valor del atributo `digito.valor` es igual a 57 –valor de 71 en octal.

□

Evaluación de una gramática atribuida

Los pasos necesarios para evaluar una gramática atribuida se pueden precisar como sigue. Se utiliza la gramática libre de contexto subyacente para construir, a partir del programa de entrada, su árbol sintáctico (o su AST en función de si la gramática es concreta o abstracta). Se construye un grafo de dependencias para el programa de entrada. Se establece un orden topológico de evaluación de las reglas semánticas de la gramática atribuida. Se ejecutan las reglas semánticas siguiendo el ordenamiento calculado, traduciendo así la gramática atribuida a un programa imperativo que trabaja sobre una estructura de datos: el árbol sintáctico. Recordemos que para que esto pueda efectuarse ante cualquier programa de entrada, la condición necesaria y suficiente es que la gramática sea no circular –y, por tanto, todo grafo de dependencias sea acíclico.

Para llevar a cabo el proceso de evaluación de una gramática atribuida existen principalmente dos métodos [Louden97, Aho90]:

- **Métodos de árbol sintáctico**[32]. En el momento de procesamiento del lenguaje, estos métodos obtienen un orden de evaluación a partir de un ordenamiento topológico del grafo de dependencias, construido para cada entrada. Para poder llevarlo a cabo, se tiene que comprobar la no-circularidad de la gramática atri-

[32] *Parse tree methods.*

buida, cuya ejecución es de complejidad exponencial [Jazayeri75][33]. Adicionalmente, la creación del grafo de dependencias y la obtención de un ordenamiento topológico cada vez que se procesa un programa, supone una complejidad adicional.
Existen herramientas que implementan este método generando, tras escribir la gramática con una sintaxis determinada, un evaluador de la misma si ésta no es circular; ejemplos de estas herramientas son FNC-2 [FNC2], Ox [Bischoff92], Elegant [Jansen93] o lrc [LRC].

– **Métodos basados en reglas[34]**. La alternativa al método anterior, adoptado por prácticamente la totalidad de los desarrollos, se basa en analizan las reglas de la gramática atribuida en el momento de la construcción del compilador, fijando a priori el orden de evaluación de las mismas. La gramática atribuida se clasifica (§ 4) y se establece para ella el mecanismo de evaluación más propicio. Aunque este método es menos general que el anterior, en la práctica es posible encontrar una gramática atribuida que cumpla estas propiedades.
Los siguientes puntos dentro de esta sección estarán enfocados a analizar cómo, en función del tipo de gramática, se puede evaluar ésta empleando un método basado en reglas.

5.2. Evaluación de Atributos en una Pasada

Cuando el procesador de lenguaje es de una pasada, la evaluación de los atributos de una gramática atribuida (y por tanto la ejecución de cada una de sus reglas semánticas) es llevada a cabo al mismo tiempo que se produce el análisis sintáctico (§ 2.2). Históricamente, la posibilidad de que un compilador pudiese llevar a cabo todas sus fases durante el análisis sintáctico en una única pasada era de especial interés por el ahorro computacional y de memoria que representaba. En la actualidad, al existir mayor capacidad de cómputo y memoria de los computadores, dicho característica no cobra tanta importancia. Al mismo tiempo, la complejidad de los lenguajes actuales hace que sea obligatorio su procesamiento en varias pasadas.

En el punto anterior señalábamos cómo los métodos basados en reglas son capaces de identificar una forma de ejecutar las reglas semánticas de la gramática atribuida en función de propiedades que han de satisfacer dichas reglas. Así, cuando deseemos evaluar los atributos de una gramática atribuida en una sola pasada, en función su tipo de reglas se podrá llevar a cabo este proceso con un análisis sintáctico u otro. Es decir, el modo en el que escribamos las reglas de la gramática atribuida determinará el tipo de análisis sintáctico que podremos utilizar –ascendente o descendente.

Las principales propiedades de una gramática atribuida que indican el modo en el que éstas deban ser evaluadas son las características de sus atributos (sintetizados y heredados) y, por tanto, la clasificación de gramáticas S y L-atribuidas (§ 4).

Un factor importante es que la mayoría de los algoritmos de análisis sintáctico procesan el programa de entrada de izquierda a derecha (por esta razón los analizadores sintácticos ascendentes o LR, y descendentes o LL, comienzan con una L[35]). Este orden implica

[33] Puesto que ésta es una condición de la gramática atribuida, la computación sólo debería llevarse a cabo una única vez –tras la escritura de la gramática atribuida– y no cada vez que se procese un programa de entrada.
[34] *Rule-based methods.*
[35] La "L" (*left*) indica que el programa de entrada es analizado de izquierda a derecha. Dada una sentencia de entrada, éste es el orden en el que el analizador léxico le pasa los *tokens* al analizador sintáctico.

una restricción en la evaluación, puesto que la utilización de los atributos de los elementos terminales de la gramática debe seguir este mismo orden.

Evaluación ascendente de gramáticas S-atribuidas

Toda gramática S-atribuida se puede evaluar ascendentemente, calculando los atributos –todos ellos sintetizados– conforme el programa de entrada es analizado [Aho90]. El analizador sintáctico deberá almacenar en su pila los valores de los atributos sintetizados asociados a cada uno de los símbolos gramaticales. En el momento en el que se lleve a cabo una reducción, se calcularán los valores de los nuevos atributos sintetizados (del no terminal de la izquierda de la producción) en función de los atributos que estén en la pila (de los no terminales de la parte derecha, entre otros). Este mecanismo es el llevado a cabo por la herramienta yacc/bison.

El poder clasificar una gramática atribuida como S-atribuida ofrece, por tanto, dos beneficios frente al concepto general de gramática atribuida:

1. Sin necesidad de calcular el grafo de dependencias, se conoce a priori el orden de ejecución de las reglas semánticas. No es necesario ordenar topológicamente el grafo (ni siquiera crearlo) para conocer su orden de evaluación.
2. Sólo es necesario visitar una vez cada nodo del árbol sintáctico creado ante la entrada de un programa[36]. Una vez ejecutada la regla semántica asociada al nodo del árbol, no necesitará volver a procesar éste. Esta propiedad conlleva una clara mejora de eficiencia en tiempo de compilación.

Evaluación descendente de gramáticas L-atribuidas

Dada una gramática atribuida que cumpla las condiciones de gramática L-atribuida[37], podrá ser evaluada mediante un analizador sintáctico descendente recursivo:

- La gramática libre de contexto deberá ser LL[38]
- Cada símbolo no terminal será traducido a una función (método) que reciba sus atributos heredados como parámetros y devuelva sus atributos sintetizados en cada invocación. Cada método asociado a un no terminal ha de realizar, además del análisis sintáctico, la evaluación de sus reglas semánticas asociadas.
- Los atributos heredados de un no terminal `A` deberán ser calculados antes de la invocación a su método asociado, y éstos deberán ser pasados como parámetros a la misma.
- Los atributos sintetizados de un no terminal `A` deberán ser calculados en la implementación de su función (método) y devueltos tras su invocación.

La principal ventaja de este algoritmo, adicionalmente a conocer a priori el recorrido de evaluación de las reglas semánticas, es que únicamente tiene que invocarse una vez a cada método representante de cada nodo del árbol sintáctico.

Ejemplo 28. En el Ejemplo 13 se definía la siguiente gramática L-atribuida para evaluar el valor de una expresión:

[36] Aunque en los procesadores de una pasada el árbol no se crea explícitamente, éste sí existe, representándose cada uno de sus nodos como un contexto en la pila del reconocedor.
[37] O una definición dirigida por sintaxis con atributos por la izquierda.
[38] La gramática deberá ser descendente y acorde con el algoritmo de análisis sintáctico, es decir, gramática LL1 si el algoritmo tiene un *lookahead* 1 o LL(k) si el algoritmo permite un k determinado.

P

(1)	expresion	→	termino masTerminos
(2)	$masTerminos_1$	→	**+** termino $masTerminos_2$
(3)		|	**-** termino $masTerminos_2$
(4)		|	**λ**
(5)	termino	→	factor masFactores
(6)	$masFactores_1$	→	***** factor $masFactores_2$
(7)		|	**/** factor $masFactores_2$
(8)		|	**λ**
(9)	factor	→	**CTE_ENTERA**

R

(1) $masTerminos.operando1 = termino.valor$
$expresion.valor = masTerminos.valor$
(2) $masTerminos_2.operando1 = masTerminos_1.operando1 + termino.valor$
$masTerminos_1.valor = masTerminos_2.valor$
(3) $masTerminos_2.operando1 = masTerminos_1.operando1 - termino.valor$
$masTerminos_1.valor = masTerminos_2.valor$
(4) $masTerminos_1.valor = masTerminos_1.operando1$
(5) $masFactores.operando1 = factor.valor$
$termino.valor = masFactores.valor$
(6) $masFactores_2.operando1 = masFactores_1.operando1 * factor.valor$
$masFactores_1.valor = masFactores_2.valor$
(7) $masFactores_2.operando1 = masFactores_1.operando1 / factor.valor$
$masFactores_1.valor = masFactores_2.valor$
(8) $masFactores_1.valor = masFactores_1.operando1$
(9) $factor.valor =$ **CTE_ENTERA**$.valor$

Es una gramática LL1, por lo que implementar un analizador sintáctico descendente recursivo predictivo con un *lookahead* de 1 es relativamente sencillo: traducimos cada no terminal en un método recursivo de una clase `Sintactico` y cada terminal a una comprobación de que el *token* esperado es el actual –denominada comúnmente `match` [Watt00].

Al mismo tiempo que se reconoce la estructura sintáctica del programa de entrada, es posible evaluar los atributos de la gramática atribuida. Los métodos que representan los no terminales de la gramática recibirán tantos parámetros como atributos heredados posean, y devolverán los valores de sus atributos sintetizados[39]. En el cuerpo de los métodos se traducirán las reglas semánticas a código, calculando los atributos heredados de un no terminal previamente a su invocación para, en su llamada, pasárselos como parámetros. Los atributos sintetizados de los símbolos de la parte derecha se obtendrán como retorno de su invocación. Finalmente, el método deberá devolver el cálculo de los atributos sintetizados del no terminal de la parte izquierda de la producción.

Siguiendo este esquema de traducción, podremos codificar en Java la primera producción y sus reglas semánticas del siguiente modo:

```
/** Método que reconoce sintácticamente el
 * no terminal "expresion".<br/>
 * Producción: expresion -> terminos masTerminos<br/>
 * Además evalúa los atributos calculados en las reglas semánticas
 * de esta producción. Recibe los atributos heredados (ninguno) y
 * devuelve los sintetizados (expresion.valor)<br/>
 */
public int expresion() {
```

[39] El modo de devolver múltiples valores por un método varía en función del leguaje de programación elegido: con registros, direcciones de memoria, vectores de elementos o incluso empleando herencia.

```
    // * Regla: masTerminos.operando1 = termino.valor
    int masTerminosOperando1=termino();
    // expresion.valor = masTerminos.valor
    return masTerminos(masTerminosOperando1);
  }
```

La traducción de la parte derecha de la producción, en lo referente al análisis sintáctico, es trivial: se traducen los no terminales a invocaciones. En el caso de la traducción de las reglas semánticas, se obtiene el atributo sintetizado `termino.valor` tras la invocación a `termino`. Éste es utilizado para asignárselo al atributo heredado `masTerminos.operando1` –primera regla. Se le pasa como parámetro a su invocación y el atributo sintetizado que nos devuelve es precisamente el que devolverá `expresion` –segunda regla.

Siguiendo con este esquema, podremos traducir las producciones 2, 3 y 4, así como sus reglas semánticas:

```
  /** Método que reconoce sintácticamente el no
   * terminal "masTerminos".<br/>
   * Produccion: masTerminos1 -> '+' termino masTerminos2<br/>
   * Produccion: masTerminos1 -> '-' termino masTerminos2<br/>
   * Produccion: masTerminos1 ->  lambda<br/>
   * Además evalúa los atributos calculados en las reglas semánticas
   * de esta producción. Recibe los atributos heredados
   * (masTerminos.operando1) y devuelve los sintetizados
   * (masTerminos.valor)<br/>
   */
  private int masTerminos(int operando1) {
  int token=lexico.getToken();
  switch (token) {
    case '+': lexico.match('+');
// * Regla: masTerminos2.operando1=masTerminos1.operando1+termino.valor
      int masTerminos2Operando1=masTerminos(operando1+termino());
// * Regla: masTerminos1.valor = masTerminos2.valor
      return masTerminos2Operando1;
    case '-': lexico.match('-');
// * Regla: masTerminos2.operando1=masTerminos1.operando1-termino.valor
      masTerminos2Operando1=masTerminos(operando1-termino());
// * Regla: masTerminos1.valor = masTerminos2.valor
      return masTerminos2Operando1;
    default: // * lambda
// * Regla: masTerminos1.valor = masTerminos1.operando1
      return operando1;
    }
  }
```

Al tratarse de un analizador descendente recursivo predictivo con un *lookahead* igual a 1, lo que se hace para distinguir por qué parte derecha producir es consultar, precisamente, el valor de dicho *lookahead*. Sabiendo el componente léxico actual, podremos conocer por cuál de las tres producciones derivar. Nótese cómo la aparición de un elemento terminal en la parte derecha es traducida a una invocación al método `match` del analizador léxico. La traducción del resto de reglas semánticas es igual que el comentado previamente.

Siguiendo este esquema es posible evaluar la gramática L-atribuida ante cualquier programa de entrada, invocando una única una vez a cada método asociado a cada símbolo gramatical del árbol. Para consultar cualquier otra faceta de la implementación en Java de este evaluador, consúltese el apéndice B.

□

El reconocimiento sintáctico del programa de entrada y su evaluación (cálculo de los atributos) se lleva a cabo tras la invocación del método asociado al no terminal inicial. El valor devuelto será el conjunto de atributos sintetizados del no terminal inicial.

Cuando un método es invocado, recibirá como parámetros los valores de sus atributos heredados, deberá invocar a los no terminales de la parte derecha, calculará sus atributos sintetizados y, finalmente, los retornará. Este proceso se hace de un modo recursivo. Fíjese como el orden de invocación de los no terminales de la parte derecha tiene que ser de izquierda a derecha, puesto que ésta la restricción impuesta en la definición de gramática L-atribuida (§ 4):

- Si un atributo heredado de un símbolo de la parte derecha depende de un atributo heredado del no terminal de la izquierda, este valor ya ha sido calculado puesto que se ha recibido como parámetro en el método que se está ejecutando.
- Si un atributo heredado de un símbolo de la parte derecha depende de un atributo de otro símbolo de la parte derecha, el segundo ha de estar la izquierda del primero (si no, no sería L-atribuida). Puesto que la invocación de métodos se realiza siguiendo un orden de izquierda a derecha, los atributos necesarios estarán siempre disponibles ya que sus métodos asociados fueron invocados previamente.

Esta técnica de evaluación descendente de gramáticas atribuidas es la llevada a cabo por las herramientas descendentes AntLR [ANTLR] y JavaCC [JavaCC].

Evaluación ascendente de atributos heredados

Hemos comentado cómo una gramática S-atribuida puede ser fácilmente evaluada por un analizador sintáctico ascendente. Para ello, la pila del analizador deberá aumentarse para albergar los valores de los atributos de los símbolos gramaticales. En cada reducción se evaluarán los atributos sintetizados del no terminal de la izquierda, empleando pare ello los valores de los atributos de la parte derecha de la producción.

En la clasificación de gramáticas presentadas en § 4 veíamos cómo las gramáticas S-atribuidas constituían un subconjunto de las L-atribuidas y, por tanto, poseían una expresividad menor. Así, el poder calcular atributos heredados mediante un analizador sintáctico ascendente siempre representará una característica positiva del mismo.

Dada la siguiente gramática atribuida:

```
<S>     →  <A> <B>      B.e = f(A.s)
<A>     →  a            A.s = a.s
<B>     →  b            B.s = g(B.e)
```

Existe la necesidad de un atributo heredado `B.e`. Este caso es el mismo que se daba en el Ejemplo 12. El atributo heredado, en un esquema de evaluación ascendente, no se podía pasar a la última producción para que pueda calcularse `B.s`. Sin embargo, la gramática anterior se puede rescribir añadiendo un no terminal vacío justo antes del símbolo que necesita el atributo heredado.

```
(1)  <S>     →  <A> <C> <B>
(2)  <A>     →  a              A.s = a.s
(3)  <B>     →  b              B.s = g(topepila-1.e)
(4)  <C>     →  λ              C.e = f(topepila.s)
```

El lenguaje reconocido por la nueva gramática no varía puesto que el no terminal introducido produce el vacío. Sin embargo, la modificación de la misma permite simular el cálculo del atributo heredado `B.e`. Puesto que el no terminal `C` se ha puesto en la primera producción justo antes del no terminal `B`, cuando `C` sea reducido por el vacío en la cuarta producción podrá acceder a todos los atributos de los que depende, al estar éstos en la pila.

En nuestro ejemplo depende únicamente de `A.s` y podrá acceder a él puesto que siempre se encontrará en el tope de la pila –ya que está justo a su izquierda en la primera producción[40].

Una vez calculado el valor de `C.e`, éste se posicionará en el tope de la pila al haberse reducido la cuarta producción. Puesto que `B` es el siguiente símbolo de `C`, la siguiente reducción será aquélla en la que `B` aparezca en la parte izquierda –producción 3. ¿Cómo podrá la regla semántica de la tercera producción acceder al atributo `B.e`? Como su parte derecha está en el tope de la pila, y `B` justo debajo, deberá restar al tope de la pila tantos elementos como símbolos aparezcan en su parte derecha. En nuestro caso sólo está el símbolo terminal `b`, por lo que el acceso a `B.e` será acceder al tope de la pila menos uno.

Hemos visto cómo es posible, siempre que se tenga acceso a los valores de la pila de un reconocedor ascendente, simular atributos heredados en un analizador sintáctico ascendente. Esta traducción tiene un conjunto de limitaciones que están en función del modo en el que se ha escrito la gramática. Dichas limitaciones pueden ser consultadas en [Scott00] y [Aho90].

El generador de analizadores sintácticos ascendentes yacc/bison ofrece, de un modo implícito, esta traducción de definiciones dirigidas por sintaxis. Esta herramienta permite ubicar la asignación de atributos heredados como código C dentro de una producción. La siguiente especificación yacc/bison, produce el mismo resultado que nuestra gramática atribuida:

```
S: A {$<e>$=f($<s>1);} B
 ;
A: a {$<s>$=$<s>1;}
 ;
B: b {$<s>$=g($<e>-1);}
 ;
```

En yacc/bison, la regla semántica ubicada en el medio de la primera producción es reemplazada por un nuevo símbolo no terminal inventado. Adicionalmente se añade una regla en la que este nuevo no terminal produce el vacío, y su regla semántica asociada es una traducción del cuerpo descrito en el medio de la primera producción. Por este motivo, al acceder a `$$` en el medio de una producción no se hace referencia al no terminal de la izquierda, sino al atributo del no terminal inventado. Es decir, yacc/bison traduce el código anterior a:

```
S: A $$1 B
 ;
A: a {$<s>$=$<s>1;}
 ;
B: b {$<s>$=g($<e>-1);}
 ;
$$1: {$<e>$=f($<s>0);}
 ;
```

Este nuevo código posee precisamente la traducción hecha previamente por nosotros, empleando la sintaxis adecuada de la herramienta yacc/bison.

Hemos visto cómo las gramáticas S y L-atribuidas pueden evaluarse en una única pasada al mismo tiempo que se lleva a cabo el análisis sintáctico. Además, para ambas se puede identificar un modo de evaluarlas a priori, sin necesidad de establecer un grafo de dependencias y un ordenamiento topológico. Las gramáticas S-atribuidas son un subconjunto de las L-atribuidas y se pueden evaluar ascendente y descendentemente, respectiva-

[40] Si dependiese de más símbolos ubicados a su izquierda (situados todos en la parte derecha de la producción) podría obtener sus valores mediante el acceso a los registros anteriores al tope de la pila.

mente. Surge la ironía de que, de un modo opuesto, las gramáticas ascendentes (LR) poseen más expresividad que las descendentes (LL) [Scott00]. Así, las gramáticas ascendentes permiten representar sintácticamente un mayor número de lenguajes, pero éstas poseen una menor capacidad a la hora de representar un cálculo de propiedades (atributos) para las distintas construcciones del lenguaje. Por este motivo surgen dos soluciones prácticas:

- Las herramientas ascendentes permiten acceder directamente a la pila del reconocedor, ofreciendo al desarrollador del compilador la posibilidad de simular atributos heredados mediante traducciones (explícitas o implícitas) de la gramática.
- Las herramientas descendentes amplían su capacidad de reconocimiento sintáctico mediante técnicas como el *backtracking* selectivo, modificación del *lookahead*, notación extendidas (EBNF) o el empleo de predicados semánticos.

5.3. Evaluación de Atributos en Varias Pasadas

En el punto anterior (§ 5.2) hemos analizado cómo pueden evaluarse las gramáticas atribuidas en el desarrollo de procesadores de una sola pasada, empleando para ello métodos basados en reglas (§ 5.1): establecimiento estático de un orden de ejecución de las reglas semánticas, en función de las características de la gramática atribuida.

Los procesadores de lenguaje de una pasada poseen los beneficios de ser más eficientes (menor tiempo de compilación) y requerir menos memoria. Sin embargo, es más sencillo realizar el análisis semántico como un recorrido del AST puesto que refleja de un modo sencillo la estructura del programa de entrada, se puede elegir el orden de evaluación en función de la propiedad analizada y, finalmente, se pueden dar tantas pasadas al AST como sea necesario. A modo de ejemplo, David Watt [Watt00] identifica que las dos tareas principales a llevar a cabo por un lenguaje de programación típico (con tipos y ámbitos estáticos) son la comprobación de ámbitos y tipos[41]. Para ello, el analizador semántico puede desarrollarse en dos pasadas:

- Identificación: Aplicando las reglas de ámbitos que define el lenguaje, cada utilización de un identificador en una expresión será resuelta conociendo su entidad concreta dentro del programa de entrada. El resultado de este recorrido del AST es que cada nodo de tipo identificador tendrá asociado una referencia a su símbolo y tipo apropiados.
- Comprobación de tipos: Posteriormente a la pasada de identificación, se va recorriendo en AST para, aplicando las reglas semánticas de inferencia de tipos del lenguaje, poder asignar un tipo a las distintas construcciones del programa. Conforme se van infiriendo los tipos, se realizarán las comprobaciones de que las operaciones llevadas a cabo con los mismos sean las correctas[42].

El número de pasadas al AST que son necesarias para llevar a cabo el análisis semántico de un lenguaje está función de sus reglas semánticas.

[41] La comprobación de ámbitos (*scope rules*) valida la correcta utilización de los identificadores en cada ámbito, y aplica las reglas de ocultación en ámbitos anidados. La comprobación de tipos (*type rules*) conlleva aplicar las reglas semánticas para inferir los tipos y restringir las operaciones que se pueden aplicar a cada expresión, en función de su tipo.

[42] Por ejemplo, al resultado de la invocación de una función (o método), sólo se le podrá aplicar el operador `[]` si el tipo devuelto es un vector –si acepta la operación `[]`.

Recorrido del AST

Hemos visto cómo en el desarrollo de compiladores reales es necesario procesar el programa de entrada en múltiples pasadas [Scott00]. En ocasiones, el propio análisis semántico ya requiere más de una pasada (identificación y comprobación de tipos) y el resto las fases siguientes realizan su cometido con pasadas aparte (generación de código intermedio, generación y optimización de código, e interpretación). El AST es comúnmente la estructura de datos empleada para cada una de las fases[43], y cada una de ellas es implementada con distintos recorridos sobre el AST.

Para realizar un compilador de varias pasadas, será necesario un diseño del mismo que nos permita realizar cada una de las pasadas sin cambiar la estructura del AST y separando claramente el código asociado a cada una de ellas. Una solución ampliamente utilizada[44] para este tipo de problemas es el patrón de diseño *Visitor* [GOF02].

Un patrón de diseño es una descripción de un conjunto de clases y objetos que se comunican entre sí, empleados para resolver un problema general de diseño dentro un contexto particular [GOF02]. Un patrón de diseño nombra, abstrae e identifica los aspectos clave de una estructura común de diseño útil para ser reutilizada en la resolución de una serie de problemas similares.

En el caso que nos atañe, el patrón de diseño *Visitor* es capaz de modelar distintas operaciones a llevar a cabo sobre una misma estructura de datos, permitiendo definir nuevas operaciones sin modificar la estructura del diseño. La estructura estática de este patrón es la mostrada en la Figura 15.

[43] En la fase de optimización de código, es común traducir este estructura de datos en un grafo dirigido acíclico [Muchnick97].

[44] Existen herramientas de desarrollo de compiladores como SableCC [SableCC] que hacen uso intensivo de este patrón de diseño.

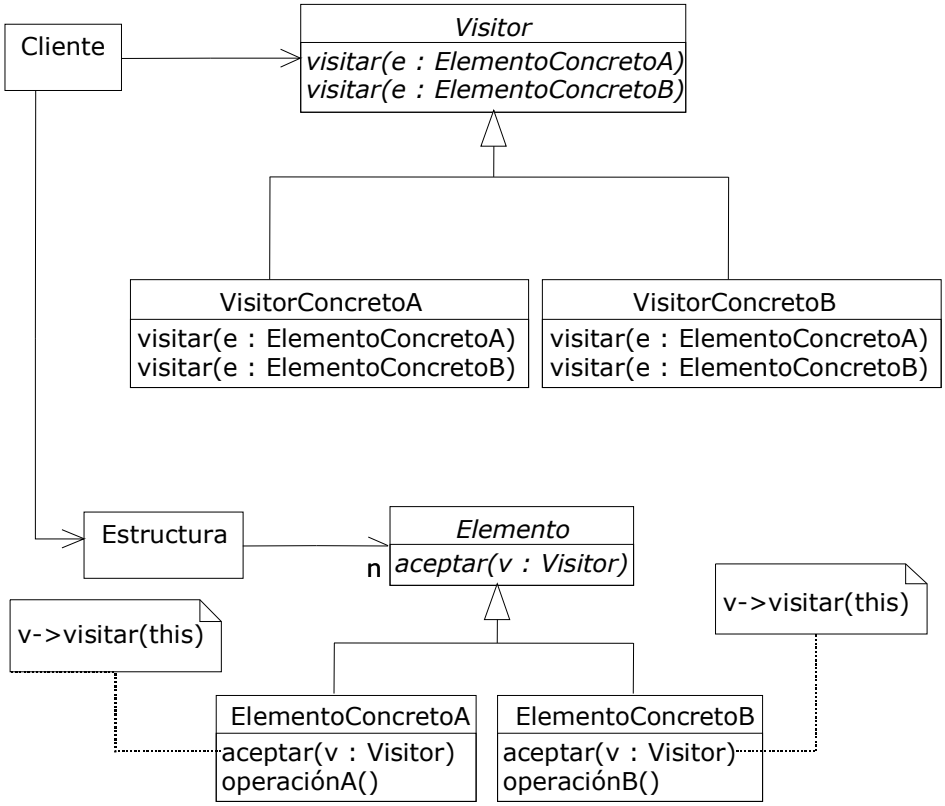

Figura 15: Diagrama de clases del patrón de diseño *Visitor*.

Las distintas abstracciones del diseño son las que siguen:

- Elemento. Clase comúnmente abstracta que define el método polimórfico `aceptar`, recibiendo un *Visitor* como parámetro. En nuestro problema es la clase base de todo nodo del AST.
- Elementos Concretos. Cada una de las estructuras concretas a recorrer. Adicionalmente a implementar el método `aceptar` como una visita concreta, podrán tener una estructura y comportamiento propio. Son cada una de las estructuras sintácticas del lenguaje, representadas como nodos del AST.
- Estructura. Es una colección de elementos. En el caso de un AST, cada nodo poseerá la colección de sus elementos mediante referencias a la clase `Elemento` y, por tanto, esta abstracción estará dentro de los nodos[45].
- Visitor. Clase comúnmente abstracta que define una operación de visita por cada elemento concreto de la estructura a recorrer. Cada uno de sus métodos se deberá redefinir (en las clases derivadas) implementando el recorrido específico de cada nodo.
- Visitor Concreto. Cada uno de los recorridos de la estructura será definido por un *Visitor* concreto, separando su código en clases distintas. La especificación de los recorridos vendrá dada por la implementación de cada uno de sus métodos de visita, que definirán el modo en el que se recorre cada nodo del AST. Éstos podrán acceder a los nodos de la estructura mediante el parámetro recibido.

[45] Realmente, esta colección de nodos por parte de cada nodo es diseñada mediante otro patrón de diseño denominado *Composite*.

Siguiendo este patrón, el analizador sintáctico creará el AST con la estructura descrita por `Elemento` y sus clases derivadas. Después, para cada una de las distintas pasadas, se creará un *Visitor* concreto y se le pasará el mensaje `aceptar` al nodo raíz del AST con el *Visitor* como parámetro. El *Visitor* concreto definirá el orden y modo de evaluar un conjunto de atributos del AST. Como el orden de ejecución de cada una de las pasadas es secuencial, los atributos calculados que sean necesarios de una pasada a otra podrán asignarse a atributos de cada uno de los objetos del AST. Por ejemplo, el tipo de cada una de las expresiones inferido en fase de análisis semántico es necesario para llevar a cabo tareas de generación de código.

Ejemplo 29. Dada la sintaxis del siguiente lenguaje en notación yacc/bison:

```
expresion: expresion '+' expresion
         | expresion '-' expresion
         | expresion '*' expresion
         | expresion '/' expresion
         | expresion '=' expresion
         | '-' expresion
         | '(' expresion ')'
         | CTE_ENTERA
         | CTE_REAL
         ;
```

Se puede crear un AST en la fase de análisis sintáctico, al igual que se hizo en el Ejemplo 7, con la siguiente estructura:

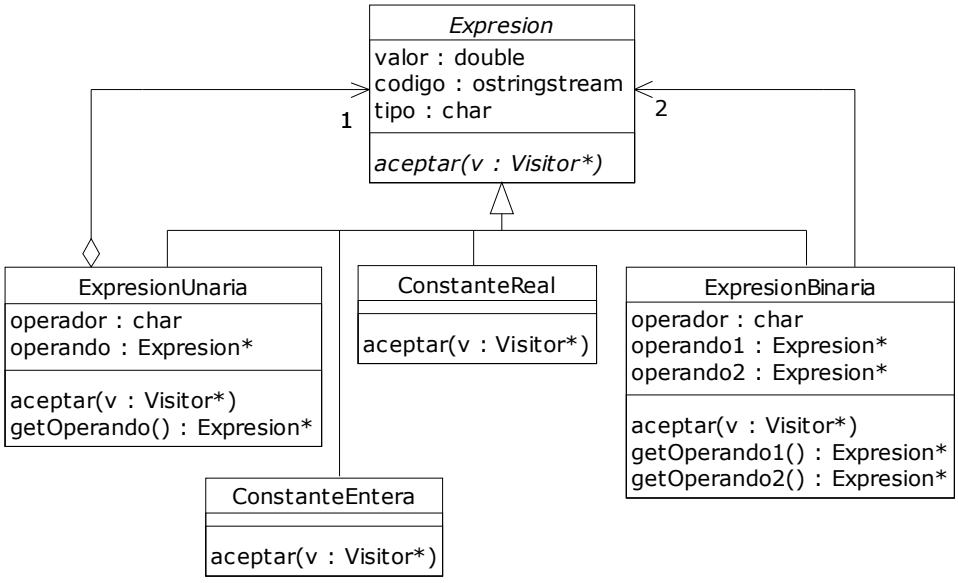

Figura 16: Estructura estática de los AST empleados para las expresiones del lenguaje presentado.

Para implementar un compilador e intérprete del lenguaje, identificamos las siguientes pasadas al AST:

- La pasada de análisis semántico que infiere el tipo de toda expresión y comprueba que se cumplen todas las reglas semánticas. El único caso semánticamente incorrecto del lenguaje –debido a su sencillez– es comprobar que no se asigne un entero a un real[46]. Añadimos el atributo `tipo` al nodo raíz del AST, puesto que nodos los nodos deben tener asociado un tipo.

[46] Realmente, todas las asignaciones son incorrectas porque tratamos únicamente con constantes. Sin embargo, lo especificaremos así para, en un futuro, poder añadir identificadores a las expresiones.

- Una pasada sería la generación de código. Añadimos, pues, un atributo al nodo raíz del AST para albergar el código de cada construcción sintáctica del lenguaje.
- Otra pasada sería el cálculo del valor de la expresión. Este escenario podría darse tanto en la optimización de código de un compilador, como en la implementación de un intérprete.
- Finalmente, para poder hacer trazas del procesador de lenguaje en tiempo de desarrollo, identificaremos una pasada que muestre el contenido del AST.

A modo de ejemplo, la siguiente sería la implementación en C++ de los métodos del *Visitor* que calcula el valor de la expresión:

```
#include "visitorcalculo.h"
#include <cassert>

void VisitorCalculo::visitar(ExpresionUnaria *e) {
       assert(e->getOperador()=='-');
       // * Calculo el valor del operando
       e->getOperando()->aceptar(this);
       e->valor= - e->getOperando()->valor;
}

void VisitorCalculo::visitar(ExpresionBinaria *e) {
       // * Calculo el valor de los operandos
       e->getOperando1()->aceptar(this);
       e->getOperando2()->aceptar(this);
       // * Calculo el valor del nodo
       switch (e->getOperador()){
       case '+': e->valor=e->getOperando1()->valor +
              e->getOperando2()->valor; break;
       case '-': e->valor=e->getOperando1()->valor -
              e->getOperando2()->valor; break;
       case '*': e->valor=e->getOperando1()->valor *
              e->getOperando2()->valor; break;
       case '/': e->valor=e->getOperando1()->valor /
              e->getOperando2()->valor; break;
       case '=': e->valor=e->getOperando2()->valor; break;
       default: assert(0);
       }
       if (e->getOperando1()->tipo=='I' && e->getOperando2()->tipo=='I')
              e->valor=(int)e->valor;
}

void VisitorCalculo::visitar(ConstanteEntera *c)  {}

void VisitorCalculo::visitar(ConstanteReal *c)  {}
```

El cálculo de un nodo que sea una expresión unaria requiere primero el cálculo de su único operando. Para ello, invoca recursivamente al método `aceptar` de la expresión almacenada como único operando. Una vez calculada ésta, asigna al atributo del nodo `ExpresionUnaria` el valor calculado, cambiado de signo.

Respecto al cálculo de las expresiones binarias el proceso es similar. Primero se calcula las dos subexpresiones mediante visitas recursivas, para posteriormente asignar el valor adecuado en función del operador. Finalmente, la visita de las dos constantes no requieren el cálculo de las mismas puesto que los nodos del AST que modelan estos terminales ya poseen dicho valor –se le pasa en su construcción.

La generación de código es lleva a cabo por el *Visitor* `VisitorGC`. Una implementación en C++ es:

```
#include "visitorgc.h"
#include <cassert>
```

```
void VisitorGC::visitar(ExpresionUnaria *e) {
	assert(e->getOperador()=='-');
	// * Genero el código del operando
	e->getOperando()->aceptar(this);
	e->codigo<<e->getOperando()->codigo.str();
	// * Genero el código de negación
	e->codigo<<"\t"<<e->tipo<<"NEG\n";  // * Instrucción INEG o FNEG
}

void VisitorGC::visitar(ExpresionBinaria *e) {
	// * Código para apilar el primer operando
	e->getOperando1()->aceptar(this);
	e->codigo<<e->getOperando1()->codigo.str();
	// * ¿Es necesario convertir el op1 de entero a real?
	if (e->getOperando1()->tipo=='I' && e->tipo=='F' )
		e->codigo<<"\tITOF\n"; // * Integer to Float
	// * Código para apilar el segundo operando
	e->getOperando2()->aceptar(this);
	e->codigo<<e->getOperando2()->codigo.str();
	// * ¿Es necesario convertir el op2 de entero a real?
	if (e->getOperando2()->tipo=='I' && e->tipo=='F' )
		e->codigo<<"\tITOF\n"; // * Integer to Float
	// * Tipo del operador
	e->codigo<<'\t'<<e->tipo; // I o F
	// * Operador
	switch (e->getOperador()){
		case '+': e->codigo<<"ADD\n"; break;
		case '-': e->codigo<<"SUB\n"; break;
		case '*': e->codigo<<"MUL\n"; break;
		case '/': e->codigo<<"DIV\n"; break;
		case '=': e->codigo<<"STORE\n"; break;
		default: assert(0);
	}
	if (e->getOperando1()->tipo=='I' && e->getOperando2()->tipo=='I')
		e->valor=(int)e->valor;
}

void VisitorGC::visitar(ConstanteEntera *c)  {
	// * Apila un entero
	c->codigo<<"\tPUSHI\t"<<(int)(c->valor)<<"\n";
}

void VisitorGC::visitar(ConstanteReal *c)  {
	// * Apila un real
	c->codigo<<"\tPUSHF\t"<<c->valor<<"\n";
}
```

Como vimos en la estructura del AST, cada nodo tiene un buffer de memoria (de tipo `ostringstream`) con su código destino de pila asociado. Podría haberse hecho con un único buffer pasado como parámetro en cada visita, necesitando así menos memoria. Sin embargo, la utilización de un atributo con el código generado para cada no terminal, es un mecanismo típico cuando se utilizan gramáticas atribuidas en la generación de código [Louden97].

La visita de una constante implica su generación de código. La traducción es utilizar la instrucción de bajo nivel `PUSH` anteponiendo el tipo de la constante (`I` o `F`) y el valor a apilar. Nótese cómo se emplea el atributo `tipo` del nodo para anteponer el tipo a la sentencia. Esto es posible porque previamente el *Visitor* de análisis semántico infirió el tipo de cada subexpresión.

Para las expresiones unarias, se genera el código de su operando (que apilará el valor de la subexpresión) y posteriormente se genera una instrucción `NEG` con su tipo adecuado. En las expresiones binarias se genera el código de las dos subexpresiones y después el de la operación. Posteriormente a la generación de cada subexpresión, como el análisis semántico calculó previamente el tipo de ambas subexpresiones, se generan instrucciones de coerción (promoción) a bajo nivel. La instrucción `ITOF` convierte el valor entero del tope de la

pila a un valor real; ésta es generada cuando el operando es entero y la expresión binaria real.

Podemos ver el código principal, dejando el análisis sintáctico y la construcción del AST a la herramienta yacc/bison:

```
#include "visitorsemantico.h"
#include "visitorgc.h"
#include "visitorcalculo.h"
#include "visitormostrar.h"
#include <iostream>
using namespace std;

int main() {
  yyparse();
  VisitorSemantico semantico;
  ast->aceptar(&semantico);
  if (ast->tipo=='E')
    cerr<<"El programa no es semánticamente válido."<<endl;
  else {
    cout<<"\nAnálisis semántico finalizado correctamente.\n";
    VisitorCalculo calculo;
    ast->aceptar(&calculo);
    cout<<"\nValor de la expresión: "<<ast->valor<<endl;
    VisitorGC gc;
    ast->aceptar(&gc);
    cout<<"\nCódigo generado:\n"<<ast->codigo.str()<<endl;
    }
  VisitorMostrar traza(cout);
  cout<<"\nAST generado:\n";
  ast->aceptar(&traza);
  cout<<endl;
  delete ast;
  }
```

Dada la siguiente entrada: `3*(2+-1)/3.2`

El procesador implementado generará la siguiente salida:

```
Análisis semántico finalizado correctamente.

Valor de la expresión: 0.9375

Código generado:
       PUSHI  3
       PUSHI  2
       PUSHI  1
       INEG
       IADD
       IMUL
       ITOF
       PUSHF  3.2
       FDIV

AST generado:
( / ( * 3 ( + 2 ( - 1 ) ) ) 3.2 )
```

Para consultar la implementación completa del AST, los cuatro *Visitor* implementados y el resto del procesador consúltese el apéndice A.

□

Un ejemplo más amplio de una implementación completa de un procesador de lenguaje en Java, empleando distintos recorridos del AST mediante el patrón de diseño *Visitor*, puede consultarse en [Watt00].

Evaluación de gramáticas S-atribuidas

Como en el caso de la evaluación de atributos en una sola pasada (§ 5.2), este tipo de gramática posibilita su evaluación mediante una única visita de cada uno de los nodos del árbol, con el consecuente beneficio de eficiencia.

En el caso de las gramáticas S-atribuidas, la evaluación de la gramática se obtiene mediante un recorrido del árbol en profundidad postorden. Si se emplea el patrón de diseño *Visitor* para recorrer un árbol sintáctico a partir de una gramática S-atribuida, su recorrido postorden en profundidad implicará la visita de todos los hijos de cada nodo, para posteriormente computar su regla semántica asociada. Dependiendo de la fase en la que nos encontremos, las visitas de los nodos producirán una determinada acción: identificación de símbolos, inferencia y comprobación de tipos, generación y optimización de código, o incluso interpretación.

Ejemplo 30. En el Ejemplo 29, la pasada de cálculo del valor de la expresión representada por el AST supone una gramática S-atribuida. La gramática abstracta (sobre el AST) es:

```
(1)  Expresion Binaria:   Expresion1  →  Expresion2 Operador Expresion3
(2)  Expresion Unaria:    Expresion1  →  Operador Expresion2
(3)  Constante Entera:    Expresion   →  CTE_ENTERA
(4)  Constante Real:      Expresion   →  CTE_REAL
```

La notación empleada para representar la gramática del AST es la identificada por Michael L. Scott [Scott00]. La sintaxis `A:B` en la parte izquierda de una producción de una gramática abstracta significa que `A` es un nodo del AST que se crea al reconocer la producción asociada al no terminal `B`. Además, A es un tipo de `B` y, por tanto, `A` puede aparecer siempre que `B` aparezca en la parte derecha de una producción. De este modo, la relación entre `A` y `B` puede modelarse en orientación a objetos como una relación de generalización (herencia): `A` es una clase derivada (más específica) que `B` (clase abstracta del AST).

Para la gramática abstracta anterior, la fase de cálculo de la expresión se puede representar con las siguientes reglas semánticas:

```
(1)  switch(opearador) {
       case '+': Expresion1.valor = Expresion2.valor +
                                    Expresion3.valor; break;
       case '-': Expresion1.valor = Expresion2.valor -
                                    Expresion3.valor; break;
       case '*': Expresion1.valor = Expresion2.valor *
                                    Expresion3.valor; break;
       case '/': Expresion1.valor = Expresion2.valor /
                                    Expresion3.valor; break;
       case '=': Expresion1.valor = Expresion3.valor; break;
     }
(2)  Expresion1.valor = - Expresion2.valor;
(3)  Expresion.valor = CTE_ENTERA.valor
(4)  Expresion.valor = CTE_REAL.valor
```

Nótese cómo la gramática es S-atribuida puesto que todos los atributos son sintetizados. La evaluación del árbol puede hacerse mediante un recorrido en profundidad postorden, evaluando el atributo valor de los nodos hijos y posteriormente calculando el valor de la subexpresión padre.

□

En los dos ejemplos anteriores se puede ver la traducción llevada a cabo entre una gramática atribuida (o definición dirigida por sintaxis) sobre una sintaxis abstracta de un lenguaje, y su representación mediante un AST. Los nodos concretos y abstractos de cada nodo, así como las relaciones de herencia, se obtienen a partir de los no terminales de la gramática y las etiquetas añadidas a cada producción. Los atributos de cada símbolo gramatical se traducen a atributos de cada una de las clases que modelan el árbol –véase el Ejemplo 29 y Ejemplo 30.

Adicionalmente a la traducción previa, válida para cualquier tipo de gramática, una gramática S-atribuida indicará un modo de codificar el método `visitar` asociado a cada uno de los nodos, pudiéndose así realizar una traducción al patrón *Visitor*. Cada método `visitar` representa la ejecución de la regla semántica asociada a la producción, cuya parte izquierda es el nodo pasado como parámetro. Puesto que, para calcular (sintetizar) los atributos del nodo que se está visitando es necesario calcular previamente los atributos (sintetizados) de sus hijos, se invocará inicialmente a los métodos `aceptar` (y por tanto al método `visitar`) para todos sus hijos y, posteriormente, se ejecutará la regla semántica asociada a la producción. Siguiendo este esquema de traducción de reglas semánticas a métodos de visita, una gramática atribuida podrá ser evaluada mediante un recorrido en profundidad postorden de un árbol sintáctico.

Evaluación de gramáticas L-atribuidas

Las gramáticas L-atribuidas, al igual que las S-atribuidas, pueden evaluarse en una única visita de todos sus nodos, con un criterio de recorrido establecido a priori. Tras haberse creado una estructura de árbol sintáctico a partir de una gramática concreta o abstracta, el problema que nos queda por abordar es cómo codificar los métodos de visita, es decir, en qué orden se ha de evaluar la gramática.

Recordemos que las gramáticas S-atribuidas son un subconjunto de las L-atribuidas. El modo de calcular los atributos sintetizados de ambas gramáticas fue analizado en el punto anterior. Para estos atributos debemos seguir el esquema de evaluación mediante un recorrido postorden de árbol. El estudio que ahora nos atañe es relativo a los atributos heredados de la gramática. Aquí, si recordamos la definición de gramática L-atribuida (§ 4), comentábamos cómo los atributos heredados podían calcularse en función de:

- Los atributos heredados del nodo padre. En este caso, el atributo del nodo padre (nodo actual, sobre el que se está ejecutando el método `visitar`) debe utilizarse para calcular un atributo heredado, antes de invocar al método `aceptar` con el nodo poseedor de dicho atributo. Por tanto, este recorrido en profundidad emplea una evaluación preorden.
- Los atributos de la parte derecha, situados a la izquierda del símbolo gramatical. Para poder asignar a un nodo `A` un atributo de un nodo `B`, situado a su izquierda, se visitará `B` (pasándole el mensaje `aceptar`) y posteriormente se evaluará el atributo de `A`. Éste es un recorrido inorden.

Vemos, pues, cómo el recorrido de una gramática L-atribuida se lleva a cabo con una única visita de cada nodo mediante un recorrido en profundidad. Se evaluarán los atributos sintetizados con un recorrido postorden, y los heredados combinando inorden y preorden, en función de sus dependencias con otros atributos.

Ejemplo 31: Dada la sintaxis dirigida por sintaxis del Ejemplo 12:

```
declaracion → tipo          variables.tipo = tipo.tipo
              variables ;
```

```
tipo          → int            tipo.tipo = 'I'
              | float          tipo.tipo = 'F'
variables1    → id ,           ts.insertar(id.valor,variables.tipo)
                variables2     variables2.tipo=variables1.tipo
              | id             ts.insertar(id.valor,variables.tipo)
```

El método visitar asociado al nodo declaración de su árbol sintáctico será:

```
void Visitor::visitar(Declaracion *e) {
      // * Se visita el no terminal "tipo"
      e->getTipo()->aceptar(this);
      // * Reglas semántica de la primera producción:
      //     variables.tipo = tipo. tipo
      e->getVariables()->tipo=e->getTipo()->tipo;
      // * Se visita el no terminal "variables"
      e->getVariables()->aceptar(this);
}
```

Se ve cómo la ejecución de la regla (asignación de un atributo heredado a partir de un atributo de un nodo hermano a su izquierda) es evaluada mediante un recorrido inorden. El otro cálculo del atributo heredado `tipo` asociado al no terminal `Variables` se codifica con el siguiente método de visita:

```
void Visitor::visitar(Variables *e) {
      // * Se ejecuta la regla que depende del heredado del padre
      ts.insertar(e->getID(),e->tipo);
      if (e->getVariables()) {
      // * Reglas semántica que depende de un hermano de la izquierda
        e->getVariables()->tipo=e->tipo;
      // * Se visita al no terminal "variables" de la parte derecha
        e->getVariables()->aceptar(this);
      }
}
```

En este caso, el recorrido es preorden: primero se ejecuta la inserción en la tabla de símbolos y la asignación de tipos, para posteriormente llevarse a cabo la visita. Aunque no se muestre, la evaluación del atributo `tipo` para segunda y tercera producción se realiza postorden –ya que `tipo` es un atributo sintetizado.

□

Otras evaluaciones con una única visita

En la evaluación de gramáticas con una única pasada, indicábamos cómo un inconveniente era que, al realizarse todas las fases en la misma pasada, la obtención de componentes léxicos (*tokens*) seguía siempre una ordenación de izquierda a derecha respecto al archivo fuente (§ 5.2). Ésta no es una restricción en el caso de realizar un procesador de lenguaje en varias pasadas, puesto que el árbol sintáctico ya posee en su estructura todos los componentes léxicos, pudiendo acceder a éstos sin necesidad de seguir un orden preestablecido.

Por la diferencia existente entre los compiladores de una y varias pasadas, descrita en el párrafo anterior, es posible que una gramática, sin ser ni S ni L-atribuida, pueda ser evaluada con una única visita de sus nodos ante cualquier programa de entrada. Además, el orden de evaluación podrá ser descrito de un modo estático –siguiendo un método basado en reglas definido en § 5.1. La principal consideración para evaluar los atributos durante el recorrido del árbol es que los atributos heredados en un nodo se calculen antes de que el nodo sea visitado, y que los atributos sintetizados se calculen antes de abandonar el nodo [Aho90].

Un primer ejemplo de este tipo de gramáticas es aquél que, sin ser L-atribuida, cumple que la evaluación de todos sus atributos heredados puede llevarse a cabo con un orden predeterminado, mediante una única visita de los no terminales de la parte derecha.

Ejemplo 32. Sea la siguiente gramática atribuida:

```
(1) A    → L M        L.h = 1
                      M.h = h(L.s)
                      A.s = i(M.s)
(2)      |  Q R       R.h = 2
                      Q.h = k(R.s)
                      A.s = l(Q.s)
(3) L    → TOKEN_L    L.s = f1(TOKEN_L.s,L.h)
(4) M    → TOKEN_M    M.s = f2(TOKEN_M.s,M.h)
(5) Q    → TOKEN_Q    Q.s = f3(TOKEN_Q.s,R.h)
(6) R    → TOKEN_R    R.s = f4(TOKEN_R.s,Q.h)
```

Se puede demostrar que no es ni S ni L-atribuida y que sí es una gramática completa y bien definida. Adicionalmente, se puede evaluar su árbol sintáctico con una única visita de cada uno de sus nodos, ante cualquier programa de entrada.

La primera producción posee un atributo sintetizado (`A.s`) y los heredados (`M.h` y `L.h`) dependen de los símbolos de su izquierda o de los heredados de su nodo padre. Así, el orden de visita única es: asignación de `L.h`, visita de `L`, asignación de `M.h`, visita de `M` y asignación de `A.s`.

En la segunda producción, la condición de gramática L-atribuida no se satisface; el orden anterior de evaluación no es válido. Sin embargo, se aprecia cómo los atributos heredados tienen una dependencia de sus hermanos por la derecha, pudiéndose establecer el siguiente ordenamiento de evaluación con una única visita de cada nodo: asignación de `R.h`, visita de `R`, asignación de `Q.h`, visita de `Q` y asignación de `A.s`.

El resto de producciones se puede evaluar con un recorrido postorden.

□

Otro tipo de gramáticas atribuidas que pueden evaluarse con una única visita de un nodo de su árbol sintáctico es aquél cuyos atributos heredados no dependan de ningún atributo sintetizado, tan solo de otros heredados [Louden97]. Así, cada método `visitar` asociado a un nodo del árbol tendrá la siguiente forma:

```
void Visitor::visitar(Nodo *n) {
      for (cada nodo hijo de n) {
            asignar los atributos heredados del hijo;
            pasar el mensaje aceptar(this) al hijo;
      }
      asignar los atributos sintetizados de n;
}
```

Evaluación de gramáticas atribuidas bien definidas

En este punto analizaremos la evaluación de gramáticas atribuidas bien definidas (no circulares) que no estén dentro de las clasificaciones previas, es decir, que requieran necesariamente más de una visita a alguno de los nodos del árbol. Si bien existen herramientas y algoritmos de análisis y evaluación que procesan este tipo de gramáticas, en el caso pragmático de desarrollo de procesadores de lenguajes se suelen evitar por la ineficiencia y complejidad que implican [Scott00].

En la práctica, si en una fase de un compilador es necesario describir una gramática que requiera varias pasadas, se suele descomponer ésta en distintas gramáticas atribuidas

que decoran el árbol, evaluando secuencialmente una tras otra. Un caso típico es el separar, dentro de la fase de análisis semántico, la gramática que lleva a cabo el proceso de identificación de la que infiere y comprueba los tipos del lenguaje [Watt00]. Con la primera gramática se decora el árbol con los símbolos y tipos de todos los identificadores declarados en el programa (clases, métodos, atributos, objetos...). En la segunda visita se decoran las distintas construcciones del lenguaje ante el programa de entrada con el tipo inferido, para procesar todas las comprobaciones de tipos necesarias.

Ejemplo 33. Supóngase la siguiente gramática libre de contexto:

```
(1)   S              →  expresion
(2)   expresion1     →  expresion2 / expresion3
(3)   expresion      →  CTE_ENTERA
(4)   expresion      →  CTE_REAL
```

Se quiere ampliar con una gramática atribuida para que el no terminal inicial (`S`) posea un atributo `S.valor` de tipo real que posea el valor de la expresión. La expresión puede ser entera o real[47]. En el caso de que la expresión sea real, todas las divisiones de la misma deben ser reales (ninguna será entera, aunque ambos operandos sean enteros). Como ejemplo, la sentencia `5/2/2.0` tendrá deberá evaluarse con valor `1.25`[48]. En el caso de que la expresión sea entera, todas sus divisiones se realizarán despreciando la parte decimal del resultado. A modo de ejemplo, `5/2/2` se evaluará con el valor de `1`.

Del problema anterior se deduce que habrá que calcular primero el tipo de la expresión y después llevar a cabo su procesamiento. El atributo que se asignará a toda subexpresión para conocer su tipo será `tiposub`, con los posibles valores `entero` y `real`. Una vez se conozca el atributo de todas las subexpresiones de una expresión, se podrá conocer el tipo global de la expresión: este valor se guardará en el atributo `tipoexp`.

Empleando los atributos definidos, la siguiente gramática atribuida resuelve el problema:

```
(1)   expresion.tipoexp = expresion.tiposub
      S.valor = expresion.valor
(2)   if (expresion2.tiposub==real || expresion3.tiposub==real)
           expresion1.tiposub = real
      else expresion1.tiposub = entero
      expresion2.tipoexp = expresion1.tipoexp
      expresion3.tipoexp = expresion1.tipoexp
      if (expresion1.tipoexp==entero)
           expresion1.valor = (int)(expresion2.valor) /
                              (int)(expresion3.valor)
      else expresion1.valor = expresion2.valor/expresion3.valor
(3)   expresion.tiposub = entero
      expresion.valor = CTE_ENTERA.valor
(4)   expresion.tiposub = real
      expresion.valor = CTE_REAL.valor
```

Esta gramática no es ni S ni L-atribuida, pero si es completa y está bien definida. Posee atributos heredados y sintetizados. El atributo `expresion.tipoexp` es heredado y `expresion.tiposub` es sintetizado. El primero depende del segundo (primera producción), así que es necesario sintetizar `expresion.tiposub` previamente al cálculo de `expresion.tipoexp`. Del mismo modo, `expresion.valor` es un atributo sintetizado dependiente del heredado `expresion.tipoexp` –segunda producción.

[47] Si la expresión es entera, el atributo `S.valor` tendrá el valor entero de la expresión.

[48] Nótese que en los lenguajes como C o Java, la evaluación de dicha expresión es 1.0 en lugar de 1.25.

En función de la dependencia de los atributos, será imposible evaluar la gramática con una única visita de los nodos del árbol. Una evaluación viable será llevar a cabo una visita en recorrido postorden para evaluar el atributo sintetizado `expresion.tiposub`. Mediante una segunda visita del árbol, se podrá ir calculando el valor `expresion.tipoexp` con un recorrido preorden; al mismo tiempo, pero con evaluación postorden, se podrá evaluar el atributo sintetizado `valor`.

Mediante la separación del objetivo final de una fase de un procesador de lenguaje en subobjetivos, se puede recorrer el árbol con una única visita de cada nodo para cada uno de los subobjetivos –empleando para ello, por ejemplo, el patrón de diseño *Visitor* (§ 5.3). Será el desarrollador del compilador el encargado de dividir en varias pasadas las distintas fases del desarrollo del procesador, en función de los requisitos del lenguaje.

□

El enfoque más práctico de evaluación de gramáticas atribuidas que requieren más de una visita de los nodos del árbol es el presentado anteriormente. Sin embargo, pueden emplearse otros algoritmos de evaluación o herramientas como FNC-2 [FNC2], Ox [Bischoff92], Elegant [Jansen93] o lrc [LRC] –amén de obtener una menor eficiencia.

Existe un algoritmo empleado para evaluar en múltiples pasadas una gramática atribuida no circular denominado algoritmo de **evaluación bajo demanda**[49] [Engelfriet84]. Éste puede aplicarse como una modificación del patrón de diseño *Visitor* [GOF02]. Consiste en reemplazar cada atributo de los nodos del árbol con un método que ejecute la parte de la regla semántica asociada a su evaluación, devolviendo su valor. Para evaluar un atributo, su método asociado deberá ser invocado y éste, a su vez, llamará a todos los métodos asociados a los atributos de los que dependa. El algoritmo recursivo finaliza si la gramática atribuida no es circular.

La técnica de evaluación bajo demanda no lleva a cabo una ordenación topológica del grafo de dependencias (§ 5.1), sino que hace uso de que las rutinas semánticas definen dicho grafo de forma implícita. El mayor inconveniente de este algoritmo es su baja eficiencia, ya que la misma regla semántica es ejecutada en múltiples ocasiones. En el peor de los casos, la complejidad computacional es exponencial en relación con el número de atributos de la gramática [Engelfriet84].

5.4. Rutinas Semánticas y Esquemas de Traducción

Al igual que existen herramientas que construyen analizadores sintácticos a partir de gramáticas libres de contexto, también existen herramientas automáticas que generan evaluadores de gramáticas atribuidas –o, en la mayor parte de los casos, definiciones dirigidas por sintaxis. En muchas ocasiones, las herramientas de desarrollo de procesadores de lenguaje ofrecen la posibilidad de especificar, de un modo imperativo en lugar de declarativo, las reglas semánticas de las definiciones dirigidas por sintaxis (gramáticas atribuidas). Esta notación es la que se conoce como **esquema de traducción (dirigida por sintaxis)**: una gramática libre de contexto en la que se asocian atributos con los símbolos gramaticales y se insertan rutinas semánticas dentro de las partes derecha de las producciones [Aho90]. Las **rutinas semánticas** son, a su vez, fragmentos de código que el desarrollador del compilador escribe –normalmente entre llaves `{}`– dejando explícito el momento en el que la herramienta ha de ejecutar la misma, durante su proceso de análisis.

La principal diferencia entre las herramientas que emplean gramáticas atribuidas y aquéllas que ofrecen esquemas de traducción es que en las segundas el desarrollador especi-

[49] *Demand-driven algorithm.*

fica el momento en el que se ha de ejecutar el código. Sin embargo, en las gramáticas atribuidas y definiciones dirigidas por sintaxis, el proceso de evaluación de los atributos debe ser resuelto por la propia herramienta. La evaluación de una gramática atribuida, conlleva procesos como la creación y ordenamiento topológico de un grafo de dependencias, o la limitación a priori de las características de la gramática –véase § 1.

La mayoría de las herramientas que generan analizadores sintácticos ofrecen la posibilidad de añadir rutinas semánticas, definiendo así un esquema de traducción. Herramientas como yacc/bison [Johnson75], ANTLR [ANTLR] o JavaCC [JavaCC] permiten entremezclar rutinas semánticas con las producciones de las gramáticas libres de contexto.

Puesto que los esquemas de traducción ejecutan las rutinas semánticas de un modo imperativo, el modo en el que se deriven las distintas producciones de la gramática variará el orden de ejecución de las rutinas. De este modo, el diferenciar si un esquema de traducción emplea un análisis descendente o ascendente es fundamental para la ubicación de sus rutinas semánticas.

En los generadores de analizadores sintácticos descendentes que incorporan esquemas de traducción (por ejemplo JavaCC o ANTLR), las rutinas semánticas pueden aparecer en cualquier parte de la parte derecha de la producción. Una rutina semántica al comienzo de la parte derecha de una producción será ejecutada cuando el analizador tome la decisión de derivar por dicha producción. Una rutina situada en el medio de la parte derecha de una producción se ejecutará una vez haya derivado todos los símbolos de la parte derecha, ubicados a su izquierda.

Como se aprecia en el párrafo anterior, las limitaciones de los esquemas de traducción basados en analizadores descendentes son los mismos que los identificados para la evaluación descendente de gramáticas L-atribuidas en una única pasada (§ 5.2). Adicionalmente a estas limitaciones, hay que añadir que la ubicación de las rutinas semánticas, dentro de la parte derecha de cada producción, sea en los sitios oportunos. Las restricciones para ubicar las rutinas son [Aho90]:

- Un atributo heredado para un símbolo en el lado derecho de una producción se debe calcular antes que dicho símbolo.
- Una rutina semántica no debe utilizar atributos sintetizados de un símbolo gramatical que esté a la derecha de ella.
- Un atributo sintetizado para el no terminal de la izquierda sólo se puede calcular posteriormente a los atributos de los que depende. La rutina semántica que calcula estos atributos se suele colocar al final del lado derecho de la producción.

Ejemplo 34. En el Ejemplo 13 se presentó una gramática L-atribuida para evaluar los valores de expresiones, mediante una gramática descendente (LL). Al haber empleado una gramática atribuida, las reglas semánticas únicamente indican la producción en a la que están asociadas. Este modo declarativo de representar las reglas semánticas hace que un evaluador de gramáticas –más o menos complejo– sea el encargado de calcular el orden de ejecución de las mismas.

En el caso de un esquema de traducción, la ubicación de las reglas (rutinas) semánticas, juega un papel importante: es la persona que escribe la gramática la encargada de indicar cuándo se va a ejecutar la rutina semántica, en función de la posición seleccionada. El esquema de traducción se limitará a ejecuta ésta con un orden preestablecido, sin necesidad de establecer previamente un ordenamiento topológico o una clasificación de la gramática.

La gramática atribuida del Ejemplo 13 podrá ser traducida, por tanto, al siguiente esquema de traducción descendente:

```
expresion       → termino
                     {masTerminos.operando1 = termino.valor}
                  masTerminos
                     {expresion.valor = masTerminos.valor}
masTerminos1    → + termino
                     {masTerminos2.operando1 =
                         masTerminos1.operando1+termino.valor}
                  masTerminos2
                     {masTerminos1.valor = masTerminos2.valor}
                | - termino
                     {masTerminos2.operando1 =
                         masTerminos1.operando1-termino.valor}
                  masTerminos2
                     {masTerminos1.valor = masTerminos2.valor}
                | λ
                   {masTerminos1.valor=masTerminos1.operando1}
termino         → factor
                     {masFactores.operando1 = factor.valor}
                  masFactores
                     {termino.valor = masFactores.valor}
masFactores1    → * factor
                     { masFactores2.operando1 =
                          masFactores1.operando1*factor.valor}
                  masFactores2
                     { masFactores1.valor = masFactores2.valor
                  }
                | / factor
                     { masFactores2.operando1 =
                          masFactores1.operando1/factor.valor}
                  masFactores2
                     {masFactores1.valor = masFactores2.valor}
                | λ
                   {masFactores1.valor=masFactores1.operando1}
factor          → CTE_ENTERA
                     {factor.valor = CTE_ENTERA.valor}
```

La ejecución de las reglas comienza por producir el no terminal `expresion`. Éste producirá el símbolo `termino`, obteniendo su atributo sintetizado `termino.valor` y asignándoselo al heredado `masTerminos.operando1` antes de su producción. Tras haber calculado el único atributo heredado de `masTerminos`, se podrá producir dicho no terminal. El esquema general es asignar los heredados antes de producir y calcular los sintetizados en las rutinas ubicadas en una producción.

El momento en el que el esquema de traducción ejecuta una rutina semántica es exactamente el mismo en el que se derivaría por un no terminal que estuviese en lugar de la rutina.

□

En el caso de los analizadores ascendentes, no es posible ubicar rutinas semánticas en cualquier zona de la parte derecha de una producción, puesto que el reconocedor no sabe en todo momento en que producción se halla –reduce únicamente cuando comprueba que el tope de la pila es igual a la parte derecha de una producción. Si se trabaja con gramáticas S-atribuidas, los esquemas de traducción asociados deberán poseer sus rutinas semán-

ticas al final de todas las producciones. Éstas se ejecutarán cuando se reduzca su producción asociada.

Si la herramienta ascendente que nos ofrece un esquema de traducción permite simular atributos heredados –como es el caso de yacc/bison– será necesario conocer las traducciones empleadas para simular dichos atributos, tales como la ubicación de rutinas semánticas en el medio de una producción, o el acceso a cualquier elemento de la pila (vistos en el punto § 5.2, dentro del epígrafe "Evaluación ascendente de atributos heredados").

Otra característica añadida a los esquemas de traducción más avanzados es la posibilidad de procesar tanto gramáticas de análisis sintáctico (gramáticas concretas), como gramáticas del resto de fases de un procesador de lenguaje (gramáticas abstractas). Son capaces de validar la estructura sintáctica, no sólo de componentes léxicos (*tokens*), sino también de nodos de un AST. Empleando los mismos esquemas de traducción se puede implementar cualquier fase de un compilador. Estos tipos de esquemas de traducción, en los que los elementos terminales de su gramática son nodos de un AST en lugar de *tokens*, se denominan *tree walkers* –implementados por ANTLR y JavaCC, entre otros.

Ejemplo 35. El siguiente esquema de traducción, empleando la sintaxis de AST de la herramienta ANTLR [ANTLR], reconoce árboles de expresiones sencillas de un lenguaje de programación:

```
expresion    returns [int r=0] {int op1,op2;}
             :      #(MAS op1=expresion op2=expresion) { r=op1+op2; }
             |      #(MENOS op1=expresion op2=expresion) {r=op1-op2;}
             |      #(POR op1=expresion op2=expresion) {r=op1*op2;}
             |      #(ENTRE op1=expresion op2=expresion) {r=op1/op2;}
             |      #(MODULO op1=expresion op2=expresion) {r=op1%op2;}
             |      cte:CTE_ENTERA {r=Integer.parseInt(cte.getText());}
             ;
```

Para comprender el ejemplo sin conocer la herramienta, se han indicado los elementos de la gramática en negrita. La sintaxis `(#Padre Hijo1 Hijo2 Hijo3)` representa un nodo en el que `Padre` es el nodo raíz y sus descendientes son `Hijo1`, `Hijo2` e `Hijo3`. Así, el esquema de traducción previo reconoce el AST de una expresión, al mismo tiempo que evalúa su valor –éste es el objetivo del código Java incluido como rutinas semánticas.

□

6 Comprobación de Tipos

En el punto § 2.1 de este libro mostrábamos un conjunto de comprobaciones que el analizador semántico de un procesador de lenguaje debería llevar a cabo. El análisis semántico acepta programas cuya estructura es válida. El analizador semántico, mediante la comprobación de cumplimiento de las reglas semánticas del lenguaje, reconoce si las estructuras aceptadas por el analizado sintáctico pertenecen al lenguaje. De este modo, el analizador semántico realiza todas las comprobaciones necesarias no llevadas a cabo por el analizador sintáctico, asegurándose de que un programa de entrada pertenezca al lenguaje.

Si bien hemos mostrado un conjunto variado de ejemplos de comprobación semántica llevada a cabo por esta fase del procesador de lenguaje (§ 2.1), la tarea más amplia y compleja de esta fase es la encomendada al **comprobador de tipos**[50]: es la parte del análisis semántico encargada de asegurar que el tipo de toda construcción del lenguaje coincida con el previsto en su contexto, dentro de las reglas semánticas del lenguaje. Si dichas reglas no se cumplen, se produce un **conflicto** u **error de tipo**[51].

Ejemplo 36. La siguiente sentencia en el lenguaje ANSI C:

```
f(*v[i+j]);
```

Es sintácticamente correcta, pero el comprobador de tipos de la fase de análisis semántico deberá comprobar un conjunto de restricciones que se han de cumplir en el contexto de la expresión:

- Las variables `i` y `j` han de estar declaradas, tener definas en su tipo la operación suma, y calcular (inferir) el tipo de dicha operación.
- Respecto al tipo calculado, deberá ser entero o convertible implícitamente (promocionable) a entero, puesto que en C los índices de un *array* han de ser enteros.
- La variable `v` deberá estar declarada y poseer como tipo un *array* o puntero (en C el operador `[]` se puede aplicar a punteros) de punteros a otro tipo `T`. De este modo, se podrán aplicar los dos operadores `[]` y `*` a dicho identificador.
- Finalmente, la función (o procedimiento) `f` tendrá que haber sido declarada previamente y poseer un único parámetro de tipo `T` o promocionable a un tipo `T`.

Todas estas tareas deberán ser llevadas a cabo por una parte del analizador semántico: el comprobador de tipos.

□

También es común que los compiladores necesiten el resultado de la inferencia o cálculo de tipos llevada a cabo por el analizador semántico, para la fase de generación de código. A modo de ejemplo, el operador + del lenguaje de programación Java, genera distinto código en el caso de que los dos operandos sean números enteros, número reales o, incluso, cadenas de caracteres. En Java, el operador + posee una semántica de suma para

[50] *Type checker*.
[51] *Type clash*.

los números enteros y reales, y una semántica de concatenación en el caso de cadenas de caracteres; para cada uno de los tres casos, el generador de código producirá distintas instrucciones.

La comprobación de tipos puede llevarse a cabo en tiempo de compilación (estática), en tiempo de ejecución (dinámica) o en ambos casos. Cada implementación de un comprobador de tipos (en tiempo de compilación o ejecución) tiene en cuenta un conjunto elevado de reglas semánticas asociadas a los tipos del lenguaje, haciendo variar de un modo significativo esta parte del análisis de un lenguaje a otro. De este modo, habrá que tener en cuenta conceptos como sistemas, expresiones y equivalencia de tipos, polimorfismo, sobrecarga, coerción y conversión de tipos. Todos estos conceptos, así como métodos para diseñar e implementar comprobadores de tipos, serán los abordados en este punto.

6.1. Beneficios del Empleo de Tipos

Incluso antes introducir las distintas definiciones de tipo de un lenguaje de programación, será interesante preguntarnos por qué existen los tipos y qué tiene de positivo su empleo [Pierce02]:

- **Fiabilidad**. La comprobación estática de tipos reduce el número de errores que un programa puede generar en tiempo de ejecución. Éste es el beneficio más obvio ya que, gracias a la detección temprana de los errores, el programador podrá reparar éstos de un modo casi inmediato –y no cuando se esté ejecutando la aplicación, pudiendo incluso haber sido implantada.
- **Abstracción**: Otra ventaja de emplear tipos en los lenguajes de programación es que su uso fuerza al programador a dividir el problema en diversos tipos de módulos, de un modo disciplinado. Los tipos identifican la interfaz de los módulos (funciones, clases, paquetes o componentes) proporcionando una simplificación de los servicios que ofrece cada módulo; un tipo de contrato parcial entre los desarrolladores del módulo y sus clientes.

  El estructurar sistemas complejos en distintos módulos con interfaces claras hace que los diseños puedan poseer una mayor abstracción, de modo que las interfaces puedan ser diseñados y debatidos de un modo independiente a su posterior implementación.
- **Legibilidad**: Un tipo de una entidad (variable, objeto o función) transmite información acerca de lo que se intenta hacer con ella, constituyendo así un modo de documentación del código.
- **Eficiencia**: Como hemos comentado en la propiedad anterior, una entidad de un programa declarada con un tipo específico indica información relativa a lo que se intenta hacer con ella. De este modo, al conocer el tipo de las construcciones del lenguaje, se podrá generar código de carácter más específico y eficiente que si no tuviésemos esa información. En el caso de no poseer tipos en tiempo de compilación, debería descubrirse la ejecución específica dinámicamente con la consecuente pérdida de eficiencia.

Ejemplo 37. En el lenguaje de programación Smalltalk [Goldberg83], el paso de un mensaje a un objeto desencadena la búsqueda, en tiempo de ejecución, de un método con el mismo nombre en el árbol de herencia. Sin embargo, en el caso de C++ la invocación a un método no virtual es resuelta por el compilador en tiempo de compilación[52], pudiendo ejecutarla

[52] Si el método es virtual, es necesario una indirección mediante una tabla de métodos virtuales.

dinámicamente en un tiempo constante, muy inferior al necesitado por Smalltalk. Posteriores optimizaciones del lenguaje Smalltalk-80 emplearon sistemas de tipos para mejorar su rendimiento en tiempo de ejecución [Atkinson86].

□

6.2. Definición de Tipo

Existen tres modos de definir el concepto de tipo, en función de distintos puntos de vista, todos ellos compatibles entre sí [Scott00]:

- **Denotacional**: Desde este punto de vista, un tipo es un conjunto de valores. Un valor es de un tipo determinado si pertenece al conjunto denotado por dicho tipo. Una variable u objeto es de un tipo dado si puede garantizarse que sus posibles valores estén siempre dentro del conjunto descrito por su tipo.

  Este punto de vista del concepto de tipo es definido por uno de los métodos más comunes de especificar la semántica de un lenguaje: la semántica denotacional (§ 1.1), en la que un conjunto de valores suele definirse como un dominio. Los tipos de un lenguaje de programación son dominios en la formalización denotacional. Este concepto de tipo está enfocado a la representación de sus posibles valores, es decir, a su significado o valor. De este modo, la definición denotacional de tipo está muy ligada a la fase de generación de código, ya que en esta fase se genera un programa de salida con igual semántica que el programa de entrada, pero expresada en otro lenguaje de programación [Watt00].

- **Basado en la abstracción**: Desde este punto de vista, un tipo es una interfaz consistente en un conjunto de operaciones que se pueden realizar sobre éste. Estas operaciones serán aplicables a una variable u objeto de dicho tipo, y poseerán una semántica bien definida.

  Este modo de ver los tipos está especialmente dirigido a la fase de análisis semántico de un procesador de lenguaje. Esta fase ha de comprobar, a partir del tipo de una expresión, que las operaciones aplicadas sobre ésta sean correctas. La semántica de cada una de las operaciones, está relacionada con la fase de generación de código.

- **Constructivo**: Un tipo es definido como un conjunto de tipos simples[53] (también llamados predefinidos, básicos o primitivos), o bien un tipo compuesto o construido formado a partir de otros tipos.

  Un **tipo simple** es un tipo atómico cuya estructura interna no puede ser modificada por el programador (`integer`, `float`, `boolean`...). Un **tipo construido** o compuesto es un tipo construido por el programador a partir de un **constructor de tipos** (`record`, `array`, `set`...) aplicado a otros tipos –básicos o construidos.

  Este modo de ver los tipos posee principalmente un carácter interno al compilador, indicando un modo de representarlos para implementar las distintas fases del procesador de lenguaje. Como veremos en breve, el punto de vista constructivo de los tipos es representado por las expresiones de tipo de un compilador.

[53] *Built-in type.*

6.3. Expresión de Tipo

Una expresión de tipo es un modo de expresar el tipo de cualquier construcción de un lenguaje, es decir, es la forma en el que un procesador de lenguaje representa cada tipo del lenguaje que procesa. Las expresiones de tipo se centran en la definición constructiva de un tipo. Así, una expresión de tipo es, o bien un tipo básico, o el resultado de aplicar un constructor de tipos a otras expresiones de tipos.

Cada compilador representará internamente sus expresiones de tipo de un modo distinto, haciendo uso de la expresividad que le ofrezca el lenguaje de programación empleado en su implementación. Inicialmente nos centraremos en una representación independiente de la implementación, para posteriormente mostrar una representación basada en un diseño orientado a objetos.

Ejemplo 38. El lenguaje de programación Java, posee el tipo primitivo entero (`int`). Desde el punto de vista denotacional, este tipo de datos denota el conjunto de número enteros comprendido entre –2.147.483.648 y 2.147.483.647 [Gosling00]. Centrándonos en el punto de vista basado en la abstracción, las operaciones permitidas son de comparación, igualdad, aritméticas, incremento, decremento, desplazamiento y operaciones a nivel de bit. Desde el punto de vista constructivo, el tipo entero es un tipo básico.

Una expresión de tipo de un tipo básico es el propio tipo básico, en nuestro caso: *`int`*. Distintas implementaciones la expresión de tipo `int` pueden ser un entero, carácter, cadena de caracteres o incluso un objeto.

En el caso del lenguaje de programación C, el mismo tipo posee una semántica denotacional no necesariamente igual. Este lenguaje deja el rango de posibles valores como dependiente de la implementación, sin establecer sus posibles valores de un modo general. Sin embargo, sí limita el conjunto de operaciones que se pueden aplicar a este tipo; éstas son similares, pero no exactamente las mismas que las de Java. Nótese cómo en ambos casos la expresión de tipo y su implementación pueden ser iguales que las del lenguaje Java.

□

Ejemplo 39. El lenguaje de programación Pascal posee los siguientes tipos simples: `boolean`, `char`, `integer` y `real`. Sus expresiones de tipo pueden ser respectivamente *`boolean`*, *`char`*, *`integer`* y *`real`*. Sus posibles valores y operaciones están definidos en la especificación del lenguaje [Pascal82]. La siguiente gramática libre de contexto representa un subconjunto de las posibles expresiones de Pascal.

```
(1)  expresion    → false
(2)  expresion    → true
(3)  expresion    → cte_caracter
(4)  expresion    → cte_entera
(5)  expresion    → cte_real
(6)  expresion1   → expresion2 AND expresion3
(7)  expresion1   → expresion2 OR expresion3
(8)  expresion1   → NOT expresion2
(9)  expresion1   → ( expresion2 )
(10) expresion1   → expresion2 + expresion3
(11) expresion1   → expresion2 - expresion3
(12) expresion1   → expresion2 * expresion3
(13) expresion1   → expresion2 / expresion3
(14) expresion1   → expresion2 div expresion3
(15) expresion1   → expresion2 > expresion3
(16) expresion1   → expresion2 < expresion3
```

(17) $expresion_1 \rightarrow expresion_2$ **=** $expresion_3$

A partir de la siguiente gramática libre de contexto, debemos definir una gramática atribuida que nos diga si la expresión posee errores de tipo.

P	R
(1)	expresion.et = boolean
(2)	expresion.et = boolean
(3)	expresion.et = char
(4)	expresion.et = integer
(5)	expresion.et = real
(6)	$expresion_1$.et = boolean
(7)	$expresion_1$.et = boolean
(8)	$expresion_1$.et = boolean
(9)	$expresion_1$.et = $expresion_2$.et
(10)	$expresion_1$.et = MayorTipo($expresion_2$.et,$expresion_3$.et)
(11)	$expresion_1$.et = MayorTipo($expresion_2$.et,$expresion_3$.et)
(12)	$expresion_1$.et = MayorTipo($expresion_2$.et,$expresion_3$.et)
(13)	$expresion_1$.et = MayorTipo($expresion_2$.et,$expresion_3$.et)
(14)	$expresion_1$.et = integer
(15)	$expresion_1$.et = boolean
(16)	$expresion_1$.et = boolean
(17)	$expresion_1$.et = boolean

	`ExpTipo mayorTipo(ExpTipo t1,ExpTipo t2) {`
	`  if ( t1==real || t2==real ) return real;`
	`  return integer;`
	`}`

Una vez calculado el atributo sintetizado `expresion.et` que posee la expresión de tipo en el caso de que la operación sea correcta, podemos identificar los casos de error como un conjunto de predicados:

P	B
(6)	$expresion_2$.et==boolean && $expresion_3$.et==boolean
(7)	$expresion_2$.et==boolean && $expresion_3$.et==boolean
(8)	$expresion_2$.et==boolean
(10)	realOentero($expresion_2$.et,$expresion_3$.et)
(11)	realOentero($expresion_2$.et,$expresion_3$.et)
(12)	realOentero($expresion_2$.et,$expresion_3$.et)
(13)	realOentero($expresion_2$.et,$expresion_3$.et)
(14)	$expresion_2$.et==integer && $expresion_3$.et==integer
(15)	realOentero($expresion_2$.et,$expresion_3$.et) || ($expresion_2$.et==char && $expresion_3$.et==char)
(16)	realOentero($expresion_2$.et,$expresion_3$.et) || ($expresion_2$.et==char && $expresion_3$.et==char)
(17)	realOentero($expresion_2$.et,$expresion_3$.et) || ($expresion_2$.et==char && $expresion_3$.et==char)

	`boolean realOentero(ExpTipo t1,ExpTtipo t2) {`
	`  return (t1==integer || t2==real) &&`
	`  (t1==integer || t2==real );`
	`}`

Otro modo de obtener los posibles errores es definiendo una nueva expresión de tipo, *error*, que denote un error de tipo al tratar de aplicar una operación no definida sobre un

tipo determinado. En este caso, la expresión deberá tener un tipo distinto a *error* para ser semánticamente correcta.

□

Existen lenguajes en los que es posible crear nuevos tipos simples. Ejemplos típicos son los tipos enumerados y subrango de lenguajes como Pascal o Ada. Estos tipos no son construidos, puesto que no utilizan expresiones de tipo para construir los nuevos tipos.

Ejemplo 40. En el lenguaje Pascal, un tipo subrango de valores enteros comprendidos entre 0 y 9 puede definirse del siguiente modo:

```
type Digito = 0..9;
```

Del mismo modo, un tipo enumerado de tres enteros distintos puede definirse, en el lenguaje C, del siguiente modo:

```
typedef enum { rojo, verde, azul } Color;
```

Tanto los valores enumerados como los subrangos son un subconjunto del tipo entero. Expresiones de tipo de los dos ejemplos pueden ser *0..9* y *[rojo,verde,azul]*. Nótese cómo los valores de los que están compuestos las expresiones tipo no son a su vez expresiones de tipo. Por este hecho, los tipos enumerados y subrango no son compuestos, sino simples.

□

Dado un conjunto de tipos simples, los lenguajes de programación permiten al usuario crear nuevos tipos, empleando constructores de tipos tales como `struct`, `array`, `union` o `class`. Estos constructores pueden ser vistos como funciones que, recibiendo un conjunto de tipos como parámetros, devuelven un nuevo tipo compuesto, cuya estructura depende del constructor empleado. El conjunto de valores que pueden albergar las entidades de estos tipos construidos, al igual que las operaciones que sobre ellos se pueden aplicar, dependerán del constructor y tipos empleados para componer el nuevo tipo.

Ejemplo 41. En este ejemplo mostraremos una representación de las expresiones de tipo, para un subconjunto de los constructores de tipo del lenguaje Pascal.

Vectores (*arrays*). Un vector denota una agrupación o colección de elementos homogéneos. Su semántica suele representarse mediante la asociación de un índice (comúnmente de tipo entero) a un elemento del tipo empleado para construir el *array*. La operación más común será, pues, el acceso a un elemento a partir de un índice –típicamente representado con el operador `[]` o `()`.

Si `T` es una expresión de tipo e `I` es una expresión de tipo subrango, entonces la expresión de tipo *array(I,T)* denota el tipo vector (*array*) de elementos de tipo `T` e índices `I`. Así, la siguiente declaración:

```
var a: array [1..10] of integer;
```

haría que un compilador de Pascal asociase la expresión de tipo *array(1..10,integer)* al identificador `a`.

Punteros. Desde el punto de vista semántico, un puntero es un modo indirecto para referenciar un elemento del tipo empleado en su construcción. Por esta característica son muy empleados para definir estructuras recursivas. Algunas implementaciones hacen que un puntero denote una dirección de memoria; en otras, simplemente albergan el identificador único de un objeto o variable. Desde el punto de vista basado en la abstracción, las operaciones más comunes son: desreferenciar para acceder al elemento al que apuntan, asignación entre punteros, y obtención y liberación de memoria.

Si T es una expresión de tipo, entonces *pointer(T)* es una expresión de tipo que representa el tipo puntero a un elemento de tipo T. A modo de ejemplo, la siguiente declaración en Pascal:

```
var b: array [-2..2] of ↑real;
```

declara la variable b con tipo *array(-2..2, pointer(real))*

Productos. Un producto o tupla de dos tipos denota el producto cartesiano de los distintos valores que cada uno de los tipos puede tomar. Una variable de este tipo poseerá una combinación de valores cada uno de los tipos empleados en la construcción de la tupla. La operación básica aplicada a este tipo es el acceso a un valor concreto de los poseídos por la tupla.

El lenguaje ML [Milner84] posee el concepto de tupla de un modo explícito, mediante el constructor de tipos *. Por ejemplo, la siguiente sentencia en ML declara una variable persona como una tripleta, dentro del producto cartesiano que representa todas las posibles combinaciones de los tres tipos string, string e int:

```
val persona = ("Suarez", "Ricardo", 9234194) : string * string * int
```

Si T_1 y T_2 son expresiones de tipo, entonces la expresión de tipo $T_1 x T_2$ denota el tipo producto (cartesiano) de elementos de tipo T_1 y T_2. Así, la anterior declaración deberá asociar a persona la expresión de tipo *string x string x int*

Si, en ML, se desea acceder al nombre de la persona, el operador que nos ofrece dicha funcionalidad es #. De este modo, la expresión #2(persona) nos devuelve "Ricardo".

Aunque el lenguaje Pascal no posee explícitamente la construcción del tipo tupla (producto), sí puede utilizarse éste para representar elementos como la lista de parámetros de una función o los tipos de un registro o unión.

Registros. Un registro es un tipo de producto (tupla) en el que los distintos elementos del mismo (campos) poseen un identificador único. Su denotación es la misma que la de los productos, pero desde el punto de vista de la abstracción, el acceso a cada uno de los elementos se hace mediante el identificador único del campo –en lugar de emplear la posición del mismo. Es común asociar el operador punto a esta operación.

Dadas dos expresiones de tipo T_1 y T_2, y dos identificadores de tipo N_1 y N_2, la expresión de tipo *Record((N1 x T1) x (N2 x T2))* denota el producto cartesiano de los dos tipos T_1 y T_2, donde N_1 y N_2 serán los nombres empleados para identificar los dos valores de T_1 y T_2 respectivamente. La siguiente declaración en Pascal:

```
TYPE cadena = array [1..255] of char;
     puntero = ↑ real;
     registro = record
                direccion: cadena;
                importe: puntero;
     end;
VAR  r:registro;
```

asocia a la variable r la expresión de tipo:

```
record( (direccion x array(1..255,char)) x
        (importe x pointer(real)) )
```

Uniones. Determinados lenguajes, como C, poseen el concepto de unión. Otros, como Pascal o Ada, añaden a los registros la posibilidad de tener campos *variantes*: campos que hacen variar la composición del registro, en función del valor que tome un campo especial llamado *selector*. Tanto el concepto de unión, como el de campo variante de un registro,

denotan la unión disjunta de cada uno de sus campos –se pueden dar cualquiera de ellos, pero únicamente uno al mismo tiempo.

En el lenguaje ANSI C, la siguiente declaración de la variable `entero_o_real` hace que ésta pueda tener la unión disjunta de un valor entero o real:

```
union miunion {
       int entero;
       float real;
} entero_o_real;
```

El lenguaje de programación ANSI C define las operaciones válidas a aplicar sobre una unión como las mismas existentes para los registros [Kernighan91]: asignación, obtención de dirección y acceso al miembro. Tampoco existe comprobación adicional de acceso a un campo incorrecto[54]. Para la fase de análisis semántico, no sería necesario, pues, crear un nuevo constructor de tipo para las uniones. Sin embargo, la fase de generación de código sí lo requiere, ya que las direcciones de memoria de cada campo no serán las mimas que en el caso de los registros. Mediante un constructor de tipo *union*, se podría asociar a la variable `entero_o_real` la expresión de tipo:

union((entero x int) x (real x float))

Funciones. Una variable de tipo función denota una transformación de elementos de un tipo, a elementos de otro tipo. En los lenguajes funcionales, las funciones son elementos de primer categoría[55], entendiendo que pueden ser pasados y devueltos como parámetros de otras funciones, o aparecer incluso ubicados en cualquier estructura de datos. En determinados lenguajes imperativos, esta facilidad aparece gracias a los punteros a funciones. De este modo, es factible poseer variables de tipo (puntero a) función. Será por tanto necesario representar dicha expresión de tipo al procesar estos lenguajes. La operación principal a aplicar sobre una función es requerir la transformación que denota, es decir, invocarla.

Si T_1 y T_2 son dos expresiones de tipo, la expresión de tipo $T_1 \rightarrow T_2$ representa el conjunto de funciones que lleva a cabo transformaciones de un dominio T_1 a un dominio T_2. A continuación mostramos ejemplos de funciones en Pascal y sus expresiones de tipos:

Funciones	**Expresiones de Tipo**
`Function f(a:char):↑integer;`	*char → pointer(integer)*
`Procedure p(i:integer);`	*integer → void*
La función estándar `mod`	*(integer x integer) → integer*

Nótese cómo, sin que el lenguaje de programación Pascal posea el tipo `void` –como los lenguajes basados en C– se puede emplear esta expresión de tipo para representar los procedimientos, como casos especiales de funciones. Podría crearse un constructor de tipos distinto para los procedimientos, y no sería necesaria la utilización de la expresión de tipo `void`. Sin embargo, se perdería el tratamiento común que se le da a funciones y procedimientos, desde la fase de análisis semántico y generación de código.

Otro elemento a destacar es cómo las funciones que reciben más de un parámetro emplean un producto o tupla de los tipos de cada uno de sus parámetros, como tipo base para llevar a cabo la transformación.

Clases. Las clases denotan un tipo de objetos que posee una estructura y comportamiento común. La mayoría de los lenguajes orientados a objetos poseen el concepto de clase para indicar un tipo de objeto. A la hora de representar este tipo por parte de un procesador de

[54] En otros lenguajes como Ada, sí se lleva a cabo esta comprobación dinámicamente.
[55] *Fist-class*.

lenguaje, es necesario tener en cuenta un conjunto de características propias de los modelos computacionales orientados a objetos. Centrándonos en el análisis semántico, las características principales a tener en cuenta a la hora de representar las clases son:

- Encapsulamiento (encapsulación). Las clases poseen, al igual que los registros, un conjunto de campos (atributos) que son identificados mediante un nombre. Éstos representan la estructura de cada uno de los objetos. Adicionalmente poseen comportamiento descrito por un conjunto de métodos o funciones miembro. Éstas también son accesibles mediante un identificador único.

- Ocultación de la información. Cada uno de los campos (atributos o métodos) puede poseer un grado de ocultación de información. Ejemplos de distintos niveles de ocultación son los grados público, privado, protegido y *package* definidos en el lenguaje de programación Java. Un compilador ha de conocer qué nivel de ocultación ha sido empleado para cada miembro, para así poder comprobar la validez semántica de los posibles accesos.

- Herencia y polimorfismo. La herencia es una relación de generalización establecida entre clases que denota una jerarquía de tipos: una clase que deriva de otra será un subtipo de la misma. De este modo, un analizador semántico deberá permitir la aparición de un subtipo donde se requiere una expresión de un tipo determinado. Las expresiones de tipo deberán representar de algún modo las relaciones de generalización entre las clases, para poder llevar a acabo el análisis semántico.

- Enlace dinámico. Esta faceta está relacionada con la fase de generación de código. El paso de un mensaje a un objeto puede desencadenar ejecuciones de distintos métodos en función del tipo de éste. La resolución del método a invocar se produce en tiempo de ejecución, y este mecanismo se conoce con el nombre de enlace dinámico. El generador de código ha de tener en cuenta esta propiedad y deberá generar el código de paso de un mensaje como el acceso a una tabla de métodos virtuales [Louden97]. Cada objeto tendrá una tabla con las direcciones de los métodos a ejecutar ante la recepción de cada mensaje. Estas direcciones son dinámicas y su valor indicará qué método ha de ser ejecutado para ese objeto concreto.

Existen otras características como métodos de clase y métodos abstractos que poseen determinados lenguajes, pero no todos.

□

Implementación

Hemos visto cómo las expresiones de tipo representan el tipo de las construcciones sintácticas de un lenguaje. Hemos visto también una notación ideada por Alfred Aho, para representar las mismas [Aho90]. El caso que ahora nos atañe es cómo representar las expresiones de tipo de un lenguaje, a la hora de implementar un procesador del mismo.

Para un lenguaje que únicamente posee tipos simples, podremos utilizar una representación mediante enteros, enumerados, caracteres o cadena de caracteres, por ejemplo. Sin embargo, no todos estos tipos serán válidos cuando el lenguaje a procesar posea algún constructor de tipo. Por ejemplo, si adicionalmente a los tipos simples existe el constructor de tipo puntero, no será posible emplear enteros, enumerados y caracteres, puesto que, mediante el constructor de tipos `pointer`, se puede construir infinitos tipos: `pointer(char)`, `pointer( pointer(char) )`, `pointer( pointer( pointer (char)))`...

Empleado cadenas de caracteres para representar las expresiones de tipos no poseemos la restricción previa. Se puede representar cada expresión de tipo con la notación descrita por Alfred Aho. No obstante, cada vez que queramos extraer un tipo que forma parte de otro tipo compuesto, deberíamos procesar la cadena de caracteres con la complejidad que ello conlleva. En el caso de los punteros no sería excesivamente complejo, pero aparecería mayor dificultad conforme surgiesen nuevos constructores de tipos. Se podría representar de un modo más sencillo con estructuras de datos recursivas. La representación de las expresiones de tipo mencionadas podría pasar a ser, en el lenguaje C, la siguiente:

```
typedef enum enum_tipos {
        entero, caracter, logico, real, puntero
} tipo_t;

typedef struct struct_puntero {
        tipo_t tipo;
        struct struct_puntero *puntero;
} expresion_tipo;
```

Esta estructura de tipos sería más fácil de manipular que una cadena de caracteres, cuando existiesen más constructores de tipos. La obtención de los tipos empleados para componer cada uno de los tipos construidos será a través del acceso a un campo del registro.

La principal limitación de este tipo de estructura de datos es que, precisamente, sólo modela datos. En la fase de análisis semántico y generación de código será necesario asociar a estas estructuras un conjunto de rutinas, tales como comprobaciones de validez semántica o generación de código intermedio. Existirán rutinas específicas para cada expresión de tipo, y otras comunes a todas ellas. Haciendo uso de técnicas ofrecidas por lenguajes orientados a objetos –tales como encapsulamiento, herencia y polimorfismo– podremos resolver esta problemática de un modo más sencillo.

El problema de representar estructuras compuestas recursivamente de un modo jerárquico, aparece en diversos contextos dentro del campo de la computación. El patrón de diseño *Composite* [GOF02] ha sido utilizado para modelar y resolver este tipo de problemas. Permite crear estructuras compuestas recursivamente, tratando tanto los objetos simples como los compuestos de un modo uniforme. Podremos representar su modelo estático mediante el siguiente diagrama de clases:

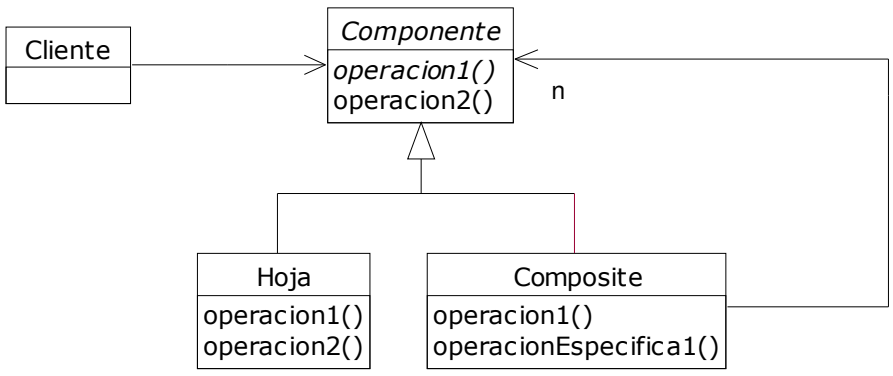

Figura 17: Diagrama de clases del patrón de diseño *Composite*.

Los distintos elementos del modelo son:

– Componente. Clase normalmente abstracta que declara la interfaz de todos los elementos, independientemente de que sean simples o compuestos. Puede im-

plementar en sus métodos un comportamiento por omisión, o declararlo como abstracto para que cada uno de los subtipos lo implementen.

- Hoja. Representa cada nodo hoja de la estructura jerárquica. Un nodo hoja no tienen ningún tipo "hijo". Definirá en sus métodos las operaciones concretas para ese nodo específico.
- Compuesto (*Composite*). Modela aquellos nodos que se construyen como composición de otros nodos, almacenando referencias a sus nodos "hijo". Implementa las operaciones en función de los hijos que posee, e incluso en función de los resultados de cada una de las operaciones de sus hijos.

Siguiendo este patrón, cada expresión de tipo primitivo será una clase hoja, y los tipos compuestos con cada constructor de tipo serán clases *Composite*. Las operaciones comunes a todos los tipos, propias de la fase de análisis semántico y de generación de código, serán ubicadas en la clase componente. Si existe un comportamiento por omisión, será implementado a este nivel. Cada clase derivada redefinirá la operación general definida en el componente, para su caso específico –u obtendrá, en el caso de existir, su comportamiento por defecto. Adicionalmente, cualquier clase derivada podrá definir métodos específicos propios de su expresión de tipo, sin necesidad de que éstos estén declarados en su clase base.

Ejemplo 42. Mostraremos en este ejemplo cómo, empleando el patrón de diseño *Composite*, las expresiones de tipo introducidas en el Ejemplo 41 pueden ser modeladas mediante un lenguaje orientado a objetos. Posteriormente (en el Ejemplo 44) se implementará, con este diseño de expresiones de tipo, un comprobador de tipos de un subconjunto del lenguaje Pascal, mediante una definición dirigida por sintaxis.

Las expresiones de tipo del lenguaje serán los tipos simples `char`, `integer` y subrango –este último, sólo válido para construir *arrays*. Adicionalmente se va a introducir un tipo `void` para los procedimientos, y `error` para indicar la existencia de un error de tipo. En el caso de que una expresión posea un error de tipo, el comprobador de tipos debería dar un mensaje al usuario y, o bien finalizar el proceso, o bien recuperarse ante el error y seguir procesando el programa.

Respecto a los constructores de tipo, tendremos `pointer`, `array` (únicamente con subrangos de números enteros), → (función) y `record`. El diagrama de clases que modela todas las posibles expresiones de tipo, es el siguiente:

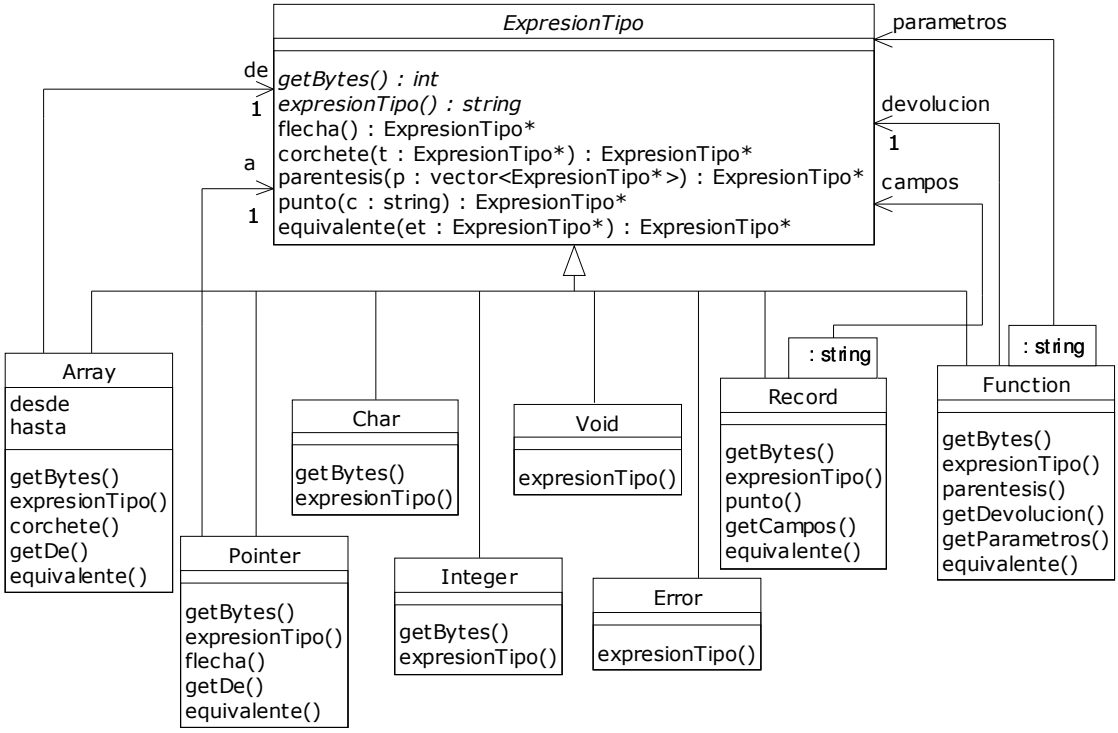

Figura 18: Diagrama de clases empleado para modelar expresiones de tipo.

La clase que juega el papel de componente en el patrón *Composite* es la clase abstracta `ExpresionTipo`. Ésta posee el comportamiento general de todas las expresiones de tipos, alguno de ellos predefinido por omisión. Los métodos de ejemplo definidos son:

- Métodos `flecha`, `corchete`, `parentesis` y `punto`: Estos métodos calculan (infieren) el tipo resultante tras aplicar un operador del lenguaje, a partir de una expresión de tipo (el objeto implícito) y, en algún caso, otras expresiones de tipo pasadas como parámetro. La implementación por omisión es devolver siempre el tipo `Error`, indicando así que dicha operación no está semánticamente definida para el tipo. Cada tipo que implemente este operador deberá redefinir el método relacionado. Por ejemplo, el tipo `Pointer` implementa el método `flecha` puesto que esta operación está permitida par este tipo; la expresión de tipo que devuelve es el tipo al que apunta.

- Método `expresionTipo`: Devuelve una cadena de caracteres representativa de su expresión de tipo, siguiendo la notación descrita por Aho (Ejemplo 41). Su objetivo principal es facilitar las tareas de depuración.

- Método `getBytes`: Es un mero ejemplo de cómo los tipos del lenguaje poseen funcionalidad propia de la fase de generación de código. Este método devuelve el tamaño en bytes necesario para albergar una variable de ese tipo.

- Método `equivalente`: Indica si dos expresiones de tipo son o no equivalentes entre sí. Posee una implementación por omisión enfocada a los tipos simples: dos tipos son equivalentes si son instancias de la misma clase. Existen diversos modos de definir la equivalencia entre tipos; profundizaremos en esta cuestión en § 6.6.

Las clases derivadas poseen métodos adicionales propios de su comportamiento específico, tales como `getCampos` (registro), `getDevolucion` y `getParametros` (función), `getA` (puntero) y `getDe` (*array*).

Nótese cómo los tipos compuestos poseen asociaciones al tipo base: un puntero requiere un tipo al que apunta (`a`); un *array* al tipo que colecciona (`de`); un registro requiere una co-

lección a sus campos (`campos`), cualificada por el nombre del campo (`string`); una función requiere una asociación al tipo que devuelve (`devolucion`), así como una colección cualificada por el nombre de cada uno de sus parámetros (`parametros`).

El hecho de que todas las asociaciones estén dirigidas hacia la clase base de la jerarquía, hace que cada tipo compuesto pueda formarse con cualquier otro tipo –incluyendo él mismo– gracias al polimorfismo.

A modo de ejemplo, la siguiente expresión de tipo:

```
record( (c x array(1..10,pointer(char))) x
        (f x (integer,pointer(char))->pointer(integer)) x
        (p x (char,integer)->void) )
```

generaría dinámicamente el siguiente diagrama de objetos:

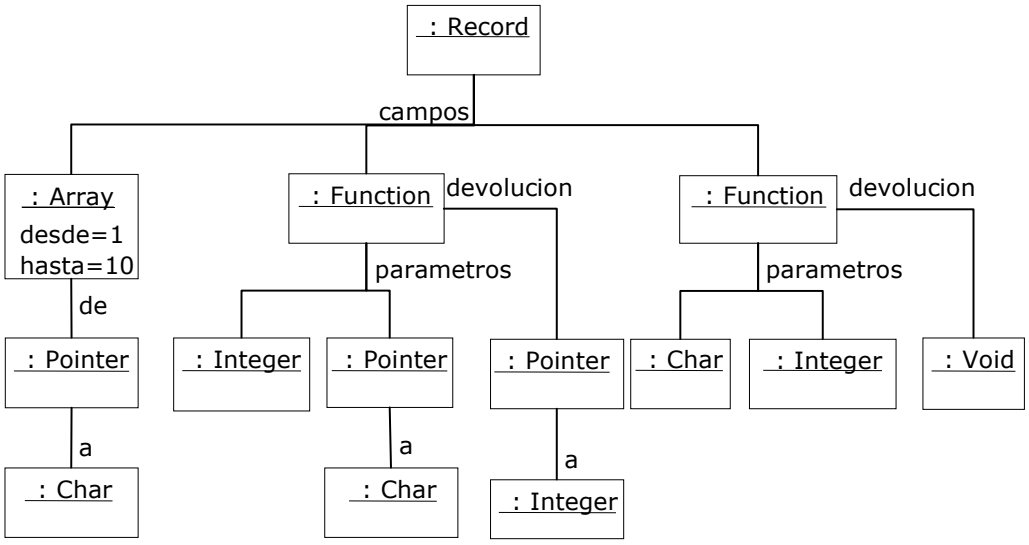

Figura 19: Diagrama de objetos creado a partir de la expresión de tipos especificada.

El diagrama de objetos posee una estructura jerárquica de árbol. Esta estructura, empleando el mismo diagrama de clases, podría haberse creado en memoria mediante un grafo acíclico dirigido (DAG), con el consecuente ahorro de memoria. La instanciación de objetos mediante esta estructura puede llevarse a cabo implementando una tabla de tipos que no cree expresiones de tipo ya existentes.

Una implementación de ejemplo es la mostrada en el apéndice C.1.

□

6.4. Sistema de Tipos

Un sistema de tipos es un conjunto de reglas para asignar expresiones de tipos a las distintas construcciones de un lenguaje [Aho90]. Para ello, un sistema de tipos deberá definir sus expresiones de tipos, asignar éstas a las distintas construcciones sintácticas del lenguaje, y comprobar que las reglas semánticas de los tipos del lenguaje se cumplan ante cualquier programa de entrada. Si no fuere así, generará un error de tipo (*type clash*), continuando el procesamiento del programa o finalizando, en función del tipo del mecanismo de manejo de errores que implemente [Louden97].

Un comprobador de tipos de un lenguaje de programación deberá implementar un sistema de tipos. En las reglas definidas por un sistema de tipos se deberá tener en cuenta conceptos como equivalencia, compatibilidad, conversión e inferencia de tipos.

Ejemplo 43. Como ejemplo de un sistema de tipos, representaremos un subconjunto de expresiones del lenguaje Pascal. Mediante una definición dirigida por sintaxis, asignaremos las expresiones de tipos del Ejemplo 41 a las distintas construcciones del lenguaje.

Recordemos que las expresiones de tipo del lenguaje serán los tipos simples *char*, *integer*, subrango, void y error. Los constructores de tipo que tenemos son *pointer*, *array* (únicamente con subrangos de números enteros), → (función) y *record*.

Un programa válido es el siguiente:

```
VAR
        vector:array[1..10] of integer;
        puntero:^integer;
        pDoble:^^integer;
        v:^array[1..10] of ^char;
        w:array[1..10] of ^char;
        f:function(integer,^char):^integer;
        p:procedure(^integer);
        r:record
                dia:integer;
                mes:integer;
                anio:integer;
        end;
BEGIN
        45;
        'A';
        f;
        vector[3];
        puntero^;
        pDoble^^;
        pDoble^;
        vector[puntero^];
        v^[puntero^]^;
        w[f(3,w[1])^]^;
        p(f(r.dia,w[2]));
END.
```

A continuación se muestra una gramática libre de contexto capaz de reconocer sintácticamente el lenguaje:

(1)	S	→	**VAR** declaraciones **BEGIN** expresiones **END** .
(2)	$declaraciones_1$	→	$declaraciones_2$ declaracion ;
(3)		|	**λ**
(4)	declaracion	→	**ID** : tipo
(5)	$tipo_1$	→	**INTEGER**
(6)		|	**CHAR**
(7)		|	**^** $tipo_2$
(8)		|	**ARRAY [** $\textbf{CTE\_ENTERA}_1$ **..** $\textbf{CTE\_ENTERA}_2$ **]** **OF** $tipo_2$
(9)		|	**FUNCTION (** listatipos **) :** $tipo_2$
(10)		|	**PROCEDURE (** listatipos **)**
(11)		|	**RECORD** listacampos **END**
(12)	listatipos	→	tipos
(13)		|	**λ**
(14)	$tipos_1$	→	$tipos_2$, tipo
(15)		|	tipo

```
(16)  listacampos1     →  listacampos2 ID : tipo ;
(17)                  |   λ
(18)  expresiones1     →  expresiones2 expresion ;
(19)                  |   λ
(20)  expresion1       →  ( expresion2 )
(21)                  |   CTE_ENTERA
(22)                  |   CTE_CARÁCTER
(23)                  |   ID
(24)                  |   expresion2 ^
(25)                  |   expresion2 [ expresion3 ]
(26)                  |   expresion2 . ID
(27)                  |   ID ( listaexps )
(28)  listaexps        →  exps
(29)                  |   λ
(30)  exps1            →  exps2 , expresion
(31)                  |   expresion
```

Desarrollaremos una definición dirigida por sintaxis que implemente un sistema de tipos, asignando expresiones de tipos a cada construcción del lenguaje. Puesto que tenemos declaraciones de identificadores, necesitaremos una tabla de símbolos (objeto `ts`) que ofrezca las operaciones `insertar` y `buscar`.

En la declaración de una variable, el símbolo no terminal `tipo` podrá definir un atributo sintetizado que albergue su expresión de tipo:

```
(5)  tipo1.et = integer
(6)  tipo1.et = char
(7)  tipo1.et = pointer(tipo2.et)
(8)  tipo1.et = array( CTE_ENTERA1.valor..CTE_ENTERA2.valor,
     tipo2.et)
```

La declaración de los tipos función, procedimiento y registro requieren una lista de tipos para poder coleccionar parámetros y campos. Un modo de representar esta colección es mediante el constructor de tipo producto (cartesiano):

```
(12)  listatipos.et = tipos.et
(13)  listatipos.et = ()
(14)  tipos1.et = ( tipos2.et x tipo.et )
(15)  tipos1.et = ( tipo.et )
(16)  listacampos1.et = ( listacampos2.et x
                          ( ID.valor x tipo.et) )
(17)  listacampos1 = ()
```

Una vez asignadas las expresiones de tipo producto, tanto a los parámetros de funciones y procedimientos como a los campos de los registros, podremos acabar de sintetizar el atributo `tipo.et`:

```
(9)   tipo1.et = listatipos.et → tipo1.et
(10)  tipo1.et = listatipos.et → void
(11)  tipo1.et = record(listacampos.et)
```

Tras llevar a cabo el cálculo del atributo `tipo.et`, podremos insertar los identificadores con su tipo adecuado en la tabla de símbolos. De este modo, cuando posteriormente aparezca un identificador en una expresión, podremos conocer su tipo.

```
(4)  ts.insertar(ID.valor, tipo.et)
```

El resto de la definición dirigida por sintaxis tendrá por objetivo implementar la parte del sistema de tipos en la que se asignará expresiones de tipo a cada una de las expresiones del lenguaje. El sistema de tipos deberá, pues, asignar al no terminal `expresion` un atributo que represente su tipo. En el caso de haberse producido algún error, el tipo a asignar será *error*.

```
(20) expresion1.et = expresion2.et
(21) expresion1.et = integer
(22) expresion1.et = char
(23) expresion1.et = ts.buscar(ID.valor)
(24) if (expresion2.et == pointer(t))
          expresion1.et = t
     else expresion1.et = error
(25) if (expresion2.et==array(i,t) && expresion3.et==integer)
          expresion1.et = t
     else expresion1.et = error

(26) if (expresion2.et == record(et1 x (ID.valor x t) x et2)
          expresion1.et = t
     else expresion1.et = error
```

La primera regla semántica es necesaria para que la gramática atribuida sea completa. Las reglas 21 y 22 se limitan a asignar el tipo de la expresión al ser ésta una constante. En la producción en la que aparece un identificador en la parte derecha, se accede a la tabla de símbolos para conocer la expresión de tipo del identificador. Si éste no se ha declarado previamente en el programa –si no está en la tabla–, el método buscar devolverá el tipo *error*.

En el caso del operador de desreferenciar, la comprobación es que la expresión a la que se aplica éste ha de ser de tipo puntero. La expresión de tipo inferida es el tipo apuntado por el puntero. Para los *arrays* la inferencia del tipo es similar. Sin embargo, es necesario comprobar que la expresión utilizada como índice sea de tipo entero. Nótese cómo, en el caso más general, no podremos saber en tiempo de compilación si la expresión está comprendida en el subrango del *array*, ya que este valor será evaluado en tiempo de ejecución[56].

Para los registros, la comprobación del operador punto es, por un lado, ratificar que el tipo de la expresión al que se le aplica dicho operador es de tipo registro. Por otra parte, debemos encontrar un campo con igual nombre que el identificador ubicado en la parte derecha del punto. Si no fuere así, el tipo inferido será *error*. En la notación empleada en el ejemplo, la expresión de tipo *record(et1 x (ID.valor x t) x et2)* indica que `et1` y `et2` pueden ser cualquier expresión de tipo –incluyendo ninguna. De este modo, `t` hace referencia a la expresión de tipo asociada al campo que coincide con el valor del identificador.

En la invocación a funciones y procedimientos, vuelve a aparecer el producto de expresiones de tipos:

```
(28)  listaexps.et = exps.et
(29)  listaexps.et = ()
(30)  exps1.et = ( exps2.et x expresion.et )
(31)  exps1.et = ( expresion.et )
```

[56] Puede haber casos en los que un compilador pueda dar un error. Por ejemplo, el acceso `v[23]` si el *array* v fue declarado con el subrango 1..10. Sin embargo, esto no es siempre factible: `v[i]` depende del valor que posea la variable `i` en la ejecución del programa.

Una vez inferido el tipo de los parámetros, podremos comprobar la validez semántica de la invocación:

```
(27) if ( ts.buscar(ID.valor)==(tp → td) && tp==listaexps.et )
         expresion1.et = td
     else expresion1.et = error
```

La regla anterior comprueba que el identificador empleado en la invocación haya sido declarado como función o procedimiento, y que el tipo de todos los parámetros reales coincida con los tipos de los parámetros formales. Si la condición es verdadera, el tipo inferido resultante de la invocación será el tipo que devuelva la función –*void* en el caso de un procedimiento. En cualquier otro caso, el tipo resultante será *error*.

Una vez implementado el sistema de tipos, es posible conocer el tipo de toda construcción sintáctica. El comprobador de tipos deberá verificar que todas las reglas semánticas relativas a los tipos se cumplen. En nuestro caso, podemos identificar los siguientes puntos en los que el comprobador podría dar un mensaje de error:

P	B
(4)	!ts.existe(**ID**.valor)
(18)	expresiones1.et != error

El comprobador de tipos deberá verificar que la expresión no haya sintetizado una expresión de tipo *error*, y que un identificador no sea declarado más de una vez.

□

Ejemplo 44. En este ejemplo se mostrará cómo, a partir de la representación de las expresiones de tipo diseñadas en el Ejemplo 42, la implementación del sistema de tipos del ejercicio anterior es realmente evidente. El procesamiento del lenguaje fuente será llevado a cabo en una única pasada, gracias a la sencillez del mismo y a que no se va a desarrollar la fase de generación de código. El siguiente fragmento de la especificación yacc/bison realiza las fases de análisis sintáctico y semántico:

```
programa: VAR declaraciones BEGINPR expresiones END '.'
        ;
declaraciones: declaraciones declaracion ';'
             | /* vacio */
             ;
declaracion: ID ':' tipo { if (ts.existe($1)) {
                             cerr<<"Error en linea "<<yylineno;
                             cerr<<". "<<$1<<" ya declarado.\n";
                             }
                           ts.insertar($1,$3);
                           }
           ;
tipo: INTEGER                          { $$=new Integer; }
    | CHAR                             { $$=new Char; }
    | '^' tipo                         { $$=new Pointer($2); }
    | ARRAY '[' CTE_ENTERA DOS_PUNTOS CTE_ENTERA ']' OF tipo
                         { $$=new Array($3,$5,$8); }
    | FUNCTION '(' listatipos ')' ':' tipo
                         { $$=new Function($6,*$3); delete $3; }
    | PROCEDURE '(' listatipos ')'
                         { $$=new Function(new Void,*$3); delete $3; }
    | RECORD listacampos END
                         { $$=new Record(*$2); delete $2; }
    ;
listatipos: tipos       { $$=$1; }
      | /* vacio */     { $$=new vector<ExpresionTipo*>; }
      ;
tipos: tipos ',' tipo   { $$=$1; $$->push_back($3); }
      | tipo     { $$=new vector<ExpresionTipo*>; $$->push_back($1); }
```

```
        ;
listacampos: listacampos ID ':' tipo ';'  { $$=$1; (*$$)[$2]=$4; }
       |     /* vacio */    { $$=new map<string,ExpresionTipo*>; }
        ;
expresiones: expresiones expresion ';'
             { Error *e=dynamic_cast<Error*>($2);
               if (e) cerr<<*e<<endl;
               else {
                 cout<<"Linea "<<yylineno<<", tipo "<<
                 $2->expresionTipo()<<", "<<
                 $2->getBytes()<<" bytes.\n"; }
             }
             | /* vacio */
             ;
expresion: '(' expresion ')'    { $$=$2; }
         | CTE_ENTERA           { $$=new Integer; }
         | CTE_CARÁCTER         { $$=new Char; }
         | ID                   { $$=ts.buscar($1); }
         | expresion '^'        { $$=$1->flecha(); }
         | expresion '[' expresion ']'    { $$=$1->corchete($3); }
         | expresion '.' ID               { $$=$1->punto($3); }
         | ID '(' listaexpresiones ')'
                  { $$=ts.buscar($1)->parentesis(*$3); delete $3; }
        ;
listaexpresiones: comaexpresiones    { $$=$1; }
       | /* vacio */                 { $$=new vector<ExpresionTipo*>; }
        ;
comaexpresiones: comaexpresiones ',' expresion
                 { $$=$1; $$->push_back($3); }
      | expresion { $$=new vector<ExpresionTipo*>; $$->push_back($1); }
      ;
```

La primera rutina semántica propia del comprobador de tipos es la ubicada en la producción `declaracion`: si el identificador que se está declarando ya existe en la tabla de símbolos, se muestra un error semántico al programador.

La construcción de las expresiones de tipo a la hora de declarar un identificador ha sido implementada mediante el empleo de constructores. Se ha aumentado el símbolo no terminal `tipo` con un atributo de tipo `ExpresionTipo*`. Este no terminal tendrá siempre la expresión de tipo asociada. En el caso de los parámetros de una función, el producto de expresiones de tipo se ha representado con la clase `vector` de la librería estándar de ISO/ANSI C++. Del mismo modo, para el producto de los campos de un registro se ha utilizado la clase `map` [Stroustrup93].

Para las distintas alternativas de producciones en las que `expresion` aparece en la parte izquierda, el sistema de tipos se limita a inferir el tipo de cada una de las expresiones. En el caso de que la expresión sea una constante, su tipo es el de la constante. Si es un identificador, el tipo se obtiene de la tabla de símbolos –si el identificador no pertenece a la tabla, la expresión de tipo devuelta será *`error`*.

En el resto de alternativas, el sistema de tipos se limita a llamar a un método de la clase `ExpresionTipo`, raíz de la jerarquía del patrón de diseño *Composite*, que modela el operador oportuno: `flecha`, `corchete`, `punto` y `parentesis`. Estos métodos tienen un comportamiento por omisión, el implementado en la clase `ExpresionTipo`, que se limita a devolver el tipo `error`:

```
ExpresionTipo *ExpresionTipo::flecha() {
             return new Error("Operación ^ no permitida.");
}

ExpresionTipo *ExpresionTipo::corchete(const ExpresionTipo*) {
             return new Error("Operación [] no permitida.");
}

ExpresionTipo *ExpresionTipo::punto(const string&) {
```

```
                return new Error("Operación . no permitida.");
}

ExpresionTipo *ExpresionTipo::parentesis(const vector<ExpresionTipo*>&)
{
                return new Error("Operación [] no permitida.");
}
```

Los objetos `Error` (que modelan la expresión de tipo *error*) poseen un atributo con el mensaje de error, y otro con el número de línea en el que éste se produjo. Por omisión, los cuatro operadores devuelven un error semántico. Cada tipo que defina el operador como válido deberá redefinir (derogar) este funcionamiento por omisión:

```
ExpresionTipo *Pointer::flecha() { return a; }

ExpresionTipo *Array::corchete(const ExpresionTipo *e) {
  if (!dynamic_cast<const Integer*>(e))
    return new Error("El índice ha de ser de tipo entero.");
  return de;
}

ExpresionTipo *Record::punto(const string &campo) {
  if (campos.find(campo)==campos.end()) {
    ostringstream o;
    o<<"El campo \""<<campo<<"\" no esta declarado en el registro.";
    return new Error(o.str());
  }
  return campos[campo];
}

ExpresionTipo *Function::parentesis(const vector<ExpresionTipo*> &v) {
  if (v.size()!=parametros.size()) {
    ostringstream o;
    o<<"La función posee "<<parametros.size()<<
       " parámetros y se le están "<<"pasando "<<v.size()<<".";
    return new Error(o.str());
  }
  // * Comparamos la igualdad de los tipos
  int equivalente=1;
  unsigned i=0;
  for (;i<v.size()&&equivalente;i++)
    equivalente=parametros[i]->equivalente(v[i]);
  if (!equivalente) {
    ostringstream o;
    o<<"La funcion está declarada con el parámetro número "<<i<<
       " de tipo "<<parametros[i-1]->expresionTipo()<<
       " y se le está pasando una expresión de tipo "<<
       v[i-1]->expresionTipo()<<".";
    return new Error(o.str());
  }
  return devolucion;
}
```

Nótese cómo cada uno de estos métodos implementa la parte del sistema de tipos asociada a cada uno de los operadores oportunos. En el caso de que el tipo no coincida, se ejecutará el funcionamiento por omisión devolviendo el tipo *error*. Los casos correctos y su semántica son:

- Operador flecha sobre el tipo *pointer*. Simplemente devuelve el tipo al que apunta.
- Operador corchete sobre el tipo *array*. Si el tipo del índice es entero, devuelve el tipo que contiene. En caso contrario devuelve el tipo *error* con el mensaje oportuno.
- Operador punto sobre el tipo *record*. Si el valor del identificador ubicado a la derecha del punto coincide con uno de los campos del registro, se devuelve la expresión de tipo asociada; si no, *error*.

– Operador paréntesis sobre el tipo →. Si el número de parámetros reales y cada un de sus tipos coincide con el número y tipos de los parámetros formales, el tipo inferido es el tipo que devuelva la función –`void` para procedimientos. En caso contrario, *`error`* con un mensaje apropiado.

Una vez el sistema de tipos infiera el tipo de cada expresión, el comprobador de tipos, en la producción `expresiones`, verificará si el tipo inferido es *`error`*. En ese caso, mostrará por la salida estándar de error el número de línea y el mensaje de error, y seguirá llevando a cabo el análisis del código fuente. Uno de los beneficios de emplear esta expresión de tipo especial (*`error`*) es que, de un modo sencillo, se puede implementar un procesador de lenguaje capaz de recuperarse ante los errores de tipo.

En el caso de que el tipo inferido en cada expresión sea correcto, a modo de ejemplo se muestra el número de línea, la expresión de tipo de tipo inferida, y el número de bytes que el generador de código emplearía para albergar en memoria el valor de la expresión. Los métodos `expresionTipo` y `getBytes` son implementados por cada tipo específico de un modo recursivo, haciendo uso de la estructura del patrón *Composite*.

El siguiente programa de entrada genera la salida mostrada:

Archivo de entrada	**Salida del procesador**

`VAR`
`  vector:array[1..10] of`
`          integer;`
`  puntero:^integer;`
`  pDoble:^^integer;`
`  v:^array[1..10] of`
`          ^char;`
`  w:array[1..10] of ^char;`
`  f:function(integer,^char`
`           ):^integer;`
`  p:procedure(^integer);`
`  r:record`
`    dia:integer;`
`    mes:integer;`
`    anio:integer;`
`    end;`	

`BEGIN`
`  45;`
`  'A';`
`  f;`
`  vector[3];`
`  puntero^;`
`  pDoble^^;`
`  pDoble^;`
`  vector[puntero^];`
`  v^[puntero^]^;`
`  w[f(3,w[1])^]^;`
`  p(f(r.dia,w[2]));`
`END.`
	`Linea 15, tipo integer, 4 bytes.`
	`Linea 16, tipo char, 1 bytes.`
	`Linea 17, tipo (integer,pointer(char))`
	`       ->pointer(integer), 4 bytes.`
	`Linea 18, tipo integer, 4 bytes.`
	`Linea 19, tipo integer, 4 bytes.`
	`Linea 20, tipo integer, 4 bytes.`
	`Linea 21, tipo pointer(integer), 4 bytes.`
	`Linea 22, tipo integer, 4 bytes.`
	`Linea 23, tipo char, 1 bytes.`
	`Linea 24, tipo char, 1 bytes.`
	`Linea 25, tipo void, 0 bytes.`

Como caso contrario, se muestra un programa en la que cada expresión del mismo posee un error semántico. Los mensajes son mostrados por el procesador en la salida estándar de error:

Archivo de entrada	**Salida del procesador**

`VAR`
`  vector:array[1..10]`
`    of integer;`
	`Error en la línea 15. El identificador "i"`
	`    no se ha declarado.`
	`Error en la línea 16. El índice ha de ser`

Archivo de entrada	Salida del procesador
`  puntero:^integer;`	`    de tipo entero.`
`  pDoble:^^integer;`	`Error en la línea 17. Operación [] no`
`  v:^array[1..10]`	`    permitida.`
`    of ^char;`	`Error en la línea 18. El índice ha de`
`  w:array[1..10]`	`    ser de tipo entero.`
`    of ^char;`	`Error en la línea 19. Operación . no`
`  f:function(integer,`	`    permitida.`
`    ^char):^integer;`	`Error en la línea 20. Operación ^ no`
`  p:procedure(^integer);`	`    permitida.`
`  r:record`	`Error en la línea 21. Operación ^ no`
`    dia:integer;`	`    permitida.`
`    mes:integer;`	`Error en la línea 22. La función posee 2`
`    anio:integer;`	`    parámetros y se le están pasando 1.`
`    end;`	`Error en la línea 23. La funcion está`
`BEGIN`	`    declarada con el parámetro número 2`
`  i;`	`    de tipo pointer(char) y se le está`
`  vector['a'];`	`    pasando una expresión de tipo integer.`
`  puntero[3];`	`Error en la línea 24. La funcion está`
`  vector[puntero];`	`    declarada con el parámetro número 1`
`  puntero.campo;`	`    de tipo integer y se le está pasando`
`  puntero^^;`	`    una expresión de tipo void.`
`  vector^;`	`Error en la línea 25. El campo "campo"`
`  f(3);`	`    no esta declarado en el registro.`
`  f(3,3);`	
`  f(p(puntero),w[3]);`	
`  r.campo;`	
`END.`	

Para consultar cualquier detalle de implementación, refiérase al apéndice C.

□

6.5. Comprobación Estática y Dinámica de Tipos

Las comprobaciones de tipos son **estáticas** si éstas son llevadas a cabo antes de que el programa se ejecute, es decir, en tiempo de compilación. Para poder implementar un comprobador de tipos estático, es necesario que toda construcción del lenguaje tenga asociada un tipo mediante el sistema de tipos.

Ejemplo 45. El siguiente programa en Pascal:

```
PROGRAM Tipos;
VAR i,j:integer;
    r:real;
BEGIN
  j:=i*r; {* Error en la comprobación estática de tipos *}
END.
```

Posee un error de tipo puesto que no es posible asignar un número real a un número entero. La parte derecha de la asignación posee un tipo real. El sistema de tipos del lenguaje Pascal identifica en una de sus reglas que el producto de un entero y un real devuelve un tipo real. No es necesario ejecutar la aplicación para que el comprobador de tipos (estático) dé un mensaje de error en la sentencia de asignación.

□

Aquellas comprobaciones de tipos que sean llevadas a cabo en tiempo de ejecución por de un procesador de lenguaje se definen como **dinámicas**. Cualquier verificación relativa a los tipos puede llevarse a cabo dinámicamente, ya que es en tiempo de ejecución

cuando se puede conocer el valor exacto que toma una expresión. Sin embargo, aunque estas comprobaciones son más efectivas, poseen dos inconvenientes:

- Las comprobaciones estáticas detectan los errores antes de que se ejecute la aplicación y, por ello, el programador podrá reparar éstos de un modo casi inmediato –y no cuando se esté ejecutando la aplicación.
- Las comprobaciones dinámicas ralentizan la ejecución de la aplicación.

Muchos lenguajes no hacen una declaración explícita de los tipos de sus variables (Lisp, Smalltalk, Python o Perl), teniendo el sistema de tipos que inferir éstos en tiempo de ejecución para poder llevar a cabo las comprobaciones de tipos.

Ejemplo 46. La siguiente función en Lisp:

```
(defun division(a,b)
       (/ a b)  )
```

Devuelve la división de dos valores pasados como parámetros. En función de los valores de ambos parámetros, puede devolver un valor entero (si ambos son enteros y la división posee resto cero) real o racional (en función de la implementación), e incluso podrá generar un error en tiempo de ejecución si uno de los parámetros es, por ejemplo, una lista. La siguiente tabla muestra invocaciones de ejemplo, la salida del intérprete y, si lo hubiere, el mensaje dinámico de error generado por el intérprete Allegro Common Lisp 6.2:

Entrada	**Salida**	**Tipo Inferido**
`(division 2 1)`	`2`	entero
`(division 2 3)`	`2/3`	racional
`(division 2 3.0)`	`0.6666667`	real
`(division 8/6 2/3)`	`2`	entero
`(division 3/2 1.5)`	`1.0`	real
`(division () 1)`	Type-Error: 'nil' is not the expected type 'number'	

Vemos como el intérprete evaluado posee un sistema y comprobador de tipos dinámico, ya que el tipo inferido y las comprobaciones de tipo varían en función del valor dinámico de cada una de las expresiones.

□

Existen lenguajes (Ada, Java o C#) que poseen comprobación estática y dinámica de tipos. En general, implementan todas las comprobaciones de tipo que puedan llevarse a cabo en tiempo de compilación, demorando aquéllas que únicamente puedan realizarse dinámicamente. En este tipo de lenguajes es común que las comprobaciones de tipo dinámicas, en el caso de que no cumplirse, lancen excepciones en tiempo de ejecución.

Ejemplo 47. El siguiente programa en Java:

```
public class Tipos {
  public static void main(String args[]) throws Exception {
    int array[]=new int[10];
    for (int i=0;i<=10;i++)
        array[i]=i;
    }
  }
```

es procesado por el compilador llevando a cabo las comprobaciones de tipos estáticas, resultando todas ellas correctas. En la ejecución, sin embargo, se produce un error dinámico, lanzando la máquina virtual de Java la excepción `IndexOutOfBoundsException`.

□

Los sistemas más sofisticados en la inferencia de tipos se dan en determinados lenguajes funcionales, notablemente en ML, Miranda y Haskell. En estos lenguajes, los programadores tienen la posibilidad de declarar explícitamente los tipos de las variables, en cuyo caso los compiladores se comportan como el resto de lenguajes con comprobación estática de tipos. No obstante, los programadores también pueden rehusar dichas declaraciones dejando al compilador la tarea de inferir los tipos, basándose en los tipos de las constantes y la estructura sintáctica del lenguaje. Lo realmente interesente es que la inferencia y comprobación de tipos la realiza estáticamente el compilador, no siendo necesaria la ejecución de la aplicación[57].

Ejemplo 48. El siguiente código es la implementación de una función recursiva del cálculo de la serie de Fibonacci, en el lenguaje ML:

```
fun fib(n) =
      let fun fib_helper(f1, f2, i) =
            if i = n then f2
            else fib_helper( f2, f1+f2, i+1)
      in
            fib_helper(0, 1, 0)
end;
```

La función anterior recibe el número de la serie como parámetro (`n`) y nos devuelve su valor. El cuerpo de la función es una invocación a otra función anidada definida dentro de ella: `fib_helper`. Ésta finaliza en el caso de haber sido invocada `n` veces. En caso contrario, calcula el siguiente valor de la serie.

El compilador de ML asocia el tipo entero al parámetro `i`, porque se le suma a éste el valor de la constante `1` en la cuarta línea. De modo similar, asigna al parámetro `n` el tipo entero, ya que esta variable se compara con `i` en la línea anterior. La única invocación posible a la función `fib_helper` recibe tres constantes enteras como parámetros; `f1`, `f2` e `i` serán, pues, enteros. Puesto que la función anidada devuelve en la tercera línea `f2`, ésta poseerá el tipo entero. Por el mismo motivo, la función principal también devolverá un entero.

Una vez llevado a cabo todo este proceso de comprobación e inferencia de tipos estática, el compilador nos devuelve un mensaje indicándonos la expresión de tipo de la función `fib`: `int → int`.

□

Un lenguaje posee un **sistema de tipos fuerte**[58] si es capaz de asegurar que no se vaya a aplicar una operación a un elemento del lenguaje que no soporte dicha operación (que no sea de dicho tipo), detectándose siempre los errores de tipo –ya bien sea estática o dinámicamente [Scott00]. Determinados errores de tipo sólo pueden ser detectados dinámicamente.

Ejemplo 49. El siguiente código Java ha de implementar comprobaciones dinámicas y estáticas para hacer que el programa no posea errores de tipo:

```
import java.io.*;
```

[57] Existen multitud de lenguajes que no requieren la declaración de los tipos de las variables u objetos: Smalltalk, Python, Perl y la mayoría de lo lenguajes de *Scripting* (TCL, PHP o {J,Java,ECMA}Script). Sin embargo, la inferencia de tipos de todos ellos se realiza dinámicamente.

[58] *Strongly typed*. En ocasiones traducido como fuertemente tipificado.

```
public class AccesoArray {
  public static void main(String args[]) throws Exception {
    int array[]=new int[10];
    int i=Integer.parseInt( (new BufferedReader(
              new InputStreamReader(System.in))).readLine() );
    System.out.println(array[i]);
    }
  }
```

El valor de la variable `i` es introducido por el usuario en tiempo de ejecución. El único modo de verificar que no se salga de rango es haciendo la comprobación dinámicamente.

□

Se dice que un lenguaje es **seguro**[59] respecto al tipo, cuando protege la integridad de sus propias abstracciones y la de las abstracciones creadas por el programador [Pierce02]. Los errores de tipo pueden ser detectados estática y dinámicamente. Cuando se detecte un error de tipo dinámicamente, un lenguaje seguro debe implementar un mecanismo para o bien manejar el error (comúnmente con excepciones), o bien para finalizar la aplicación sin que ésta rompa la integridad de las abstracciones del programa.

Aunque determinados autores, como Luca Cardelli [Cardelli97], afirman que los lenguajes con comprobación fuerte de tipos son un subconjunto de los lenguajes con sistemas de tipos seguros, en general la mayoría de autores suele emplear ambos términos como sinónimos.

Ejemplo 50. En Ejemplo 49 se mostró cómo Java implementaba comprobación dinámica de tipos, generando una excepción cuando ésta se produce. La seguridad del lenguaje permite, además, que el programador puede detectar estas situaciones y tomar la determinación de "manejar" el error del modo que estime oportuno:

```
import java.io.*;
public class AccesoArray {
  public static void main(String args[]) throws Exception {
    int array[]=new int[10];
    try {
    int i=Integer.parseInt( (new BufferedReader(
              new InputStreamReader(System.in))).readLine() );
    System.out.println(array[i]);
    } catch(IndexOutOfBoundsException e) {
      System.err.println("Fuera de rango");
    }
  }
}
```

□

Ejemplo 51. El lenguaje de programación ISO/ANSI C++ no es seguro. Características como el uso directo de punteros a memoria y el operador de ahormado (*cast*) hacen que el lenguaje no asegure la integridad de sus abstracciones. Un programa de ejemplo:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
       char s[]="C++ no es un lenguaje seguro";
       ((vector<int>*)s)->push_back(3);
       return 0;
}
```

El anterior es un programa válido para un compilador de C++. La segunda línea del programa principal ahorma la cadena de caracteres a un puntero a vector de enteros, e "inten-

[59] *Safe*.

ta" insertar un 3 al final. El resultado de la ejecución del programa anterior es imprevisible y el comité ISO/ANSI la deja como "dependiente de la implementación" [ANSIC++].

□

La mayor parte de los lenguajes que poseen comprobación estática de tipos requieren adicionalmente comprobación dinámica, para poder ofrecer un sistema de tipos fuerte. Ada es un ejemplo de este caso. Los campos variantes de los registros de Ada son compilados del mismo modo que en Pascal. Sin embargo, en Ada se genera adicionalmente código para, en tiempo de ejecución, comprobar que el acceso a un campo variante sea acorde con el valor del campo selector. Pascal no lleva a cabo esta comprobación.

La siguiente tabla muestra la relación existente entre la "seguridad"[60] respecto al tipo y el momento en el que se lleva a cabo las comprobaciones, refiriéndonos a casos reales de lenguajes de programación:

	Sin comprobación de tipos	**Sólo Comprobación Estática**	**Comprobación Estática y Dinámica**	**Sólo Comprobación Dinámica**
Safe			C#, Miranda, Java, Haskell, ML.	Lisp, Perl, Smalltalk, Scheme
Unsafe	Ensamblador, BCPL	C, Fortran, Pascal	C++	

De la tabla anterior se pueden extraer un conjunto de conclusiones:

- Los lenguajes que no poseen comprobación de tipos (como ensamblador o BCPL) no son seguros.
- Los lenguajes que únicamente poseen comprobación estática de tipos no pueden ser seguros, ya que existen abstracciones de ellos que requieren comprobaciones en tiempo de ejecución –por ejemplo los *arrays*.
- La mayor parte de los lenguajes que implementan comprobación dinámica de tipos ofrecen seguridad respecto al tipo. Una vez implementadas comprobaciones dinámicas, es común que se implementen todas. Una excepción es C++. Este lenguaje añadió una pequeña información de tipos en tiempo de ejecución (*RunTime Type Information*, RTTI) para poder recuperar el tipo de un objeto, al emplear éste en una jerarquía polimórfica [Eckel00]. No se añadieron otro tipo de comprobaciones dinámicas por motivos de eficiencia.
- Un lenguaje seguro no tiene por qué tener comprobación estática de tipos.

6.6. Equivalencia de Expresiones de Tipo

En la definición dirigida por sintaxis mostrada en el Ejemplo 43, aparecían reglas semánticas como:

```
if (expresion2.et == pointer(t))
```

En dicha regla, y en cualquier sistema de tipos, es necesario tener una definición precisa de cuándo dos expresiones de tipo son equivalentes; es decir, en la sentencia ante-

[60] *Safety.*

rior, qué significa exactamente la igualdad de expresiones de tipo. La equivalencia de tipos es utilizada intensivamente en todos los sistemas de tipos[61].

Existen distintas formas en las que un lenguaje define semánticamente la equivalencia entre sus tipos. Surgen además posibles ambigüedades en los lenguajes que permiten asignar nombres a las expresiones de tipo –mediante `type` en Pascal y Haskell, o con `typedef` en C++.

Una primera equivalencia es la **equivalencia estructural** de expresiones de tipo. Un sistema de tipos define que dos tipos son estructuralmente equivalentes, únicamente si poseen la misma estructura, es decir, o son el mismo tipo básico, o bien están formadas aplicando el mismo constructor a tipos estructuralmente equivalentes. Lenguajes tales como ML, Algol-68 y Modula-3 emplean un sistema de tipos basado en equivalencia estructural. C++ implementa equivalencia estructural, excepto para clases, registros y uniones. El lenguaje Pascal no tiene una equivalencia de tipos definida; depende de la implementación del compilador.

Ejemplo 52. El siguiente código ISO/ANSI C++:

```
#include <iostream>
using namespace std;

typedef int entero;
typedef int *punteroEntero;

void fi(int n,int *p) { cout<<n<<' '<<p<<endl; }
void fe(entero n,punteroEntero p) { cout<<n<<' '<<p<<endl; }

int main() {
       int n,*pn=&n;
       entero e=n;
       punteroEntero pe=pn;

       fi(e,pe);
       fe(n,pn);

       return 0;
}
```

es semánticamente correcto. Un compilador que implemente el estándar ISO/ANSI aceptará el programa como válido, generando un código objeto. Tanto en las dos últimas asignaciones del programa principal, como en las dos invocaciones a funciones, aparece la equivalencia estructural de tipos que dicho lenguaje implementa en su sistema de tipos.

□

En § 6.3 indicábamos cómo el modo más común de modelar las expresiones de tipo en un procesador de lenguaje era mediante estructuras compuestas recursivamente. Éstas se crean de un modo jerárquico, pudiendo dar lugar a estructuras de árbol o –más eficientes– estructuras de grafo acíclicos dirigidos (DAGs). En el caso de que se estén representando las expresiones de tipo con un DAG, cada expresión de tipo tendrá una única representación en memoria. Por tanto, la comprobación de equivalencia, en este caso, se reduce a verificar la identidad del nodo: si los dos nodos del DAG son el mismo, entonces las expresiones de tipo son estructuralmente equivalentes.

En el cado de implementar las expresiones de tipo con una estructura de árbol, hace que éstas puedan estar repetidas y que, por tanto, la equivalencia estructural de tipos

[61] Nótese que no se está abordando el problema de la conversión, implícita o explícita, de tipos que se da en muchos lenguajes de programación –esto será tratado en el siguiente punto. Lo que se está acometiendo aquí es la equivalencia (correspondencia o igualdad) entre tipos.

no esté ligada a la identidad de los nodos del árbol. El proceso que deberá llevarse a cabo será comprobar recursivamente que las estructuras de los dos nodos sean equivalentes.

Ejemplo 53. En el Ejemplo 43 se desarrolló una definición dirigida por sintaxis que implementaba un sistema de tipos de un subconjunto del Pascal. Los tipos simples definidos eran *char*, *integer*, subrango, void y error. Los constructores de tipo que teníamos eran *pointer*, *array*, → (función) y *record*.

Para poder implementar un sistema de tipos con equivalencia estructural, el sistema de tipos se centrará en una función equivale que recibirá dos expresiones de tipo, representadas mediante un árbol, y devolverá si ambas son o no equivalentes.

```
boolean equivale(ExpTipo t1,ExpTipo t2) {
      if (t1==char && t2==char) return true;
      if (t1==integer && t2==integer) return true;
      if (t1==void && t2==void) return true;
      if (t1==a1..b1 && t2==a2..b2) return a1==a2&&b1==b2;
      if (t1==pointer(tp1) && t2==pointer(tp2))
            return equivale(tp1,tp2);
      if (t1==array(a1,b1) && t2==array(a2,b2))
            return equivale(a1,a2) && equivale(b1,b2);
      if (t1==a1xb1 && t2==a2xb2)
            return equivale(a1,a2) && equivale(b1,b2);
      if (t1==a1→b1 && t2==a2→b2)
            return equivale(a1,a2) && equivale(b1,b2);
      if (t1==record(a) && record(b))
            return equivale(a,b);
      if (t1==t2) // Nombres de los campos de un registro
            return true;
      return false; // En el resto de casos
}
```

□

Ejemplo 54. Si la implementación de equivalencia estructural de tipos fuese reutilizada mediante el patrón de diseño *Composite* (§ 6.3), el algoritmo anterior estaría disperso en la implementación de un único método de las distintas clases de la jerarquía. Éste es el método equivalente que mostrábamos en el diagrama de clases del Ejemplo 42. Su comportamiento por omisión está enfocado a la equivalencia de los tipos simples: dos tipos básicos con equivalentes si son el mismo tipo:

```
bool ExpresionTipo::equivalente(const ExpresionTipo *et) const {
      return typeid(*this)==typeid(*et); // * RTTI
}
```

La comparación realizada es respecto a las dos clases de los dos objetos C++. Esta funcionalidad se puede obtener aplicando el operador de igualdad a la devolución del operador typeid[62]. La comparación estructural de los tipos construidos se obtiene mediante un proceso recursivo de comparación de equivalencia. La implementación es muy sencilla:

```
bool Pointer::equivalente(const ExpresionTipo *et) const {
  const Pointer *puntero=dynamic_cast<const Pointer*>(et);
  if (!puntero) return false;
  return a->equivalente(puntero->a);
}
```

[62] Esta funcionalidad del ISO/ANSI C++ se denomina RTTI y será explicada con más detenimiento en § 6.7.

```
bool Array::equivalente(const ExpresionTipo *et) const {
  const Array *array=dynamic_cast<const Array*>(et);
  if (!array) return false;
  return desde==array->desde && hasta==array->hasta &&
    array->equivalente(array->de);
}

bool Function::equivalente(const ExpresionTipo *et) const {
  const Function *fun=dynamic_cast<const Function*>(et);
  if (!fun) return false;
  if (parametros.size()!=fun->parametros.size())
    return false;
  for (unsigned i=0;i<parametros.size();i++)
    if (!parametros[i]->equivalente(fun->parametros[i]))
      return false;
  return devolucion->equivalente(fun->devolucion);
}

bool Record::equivalente(const ExpresionTipo *et) const {
  const Record *record=dynamic_cast<const Record*>(et);
  if (!record) return false;
  if (campos.size()!=record->campos.size())
    return false;
  map<string,ExpresionTipo*>::const_iterator it1,it2;
  for (it1=campos.begin(),it2=record->campos.begin();
                      it1!=campos.end();++it1,++it2) {
    if (it1->first!=it2->first)
      return false;
    if (!it1->second->equivalente(it2->second))
      return false;
  }
  return true;
}
```

Como hemos mencionado previamente, esta comparación recursiva de la estructura de las expresiones de tipo es necesaria por existir la duplicidad de objetos que representan una misma expresión de tipo –estructura de árbol. Si se implementa mediante un DAG, la equivalencia estructural se reduce a una comparación de identidad de los nodos del DAG. □

En la implementación del algoritmo anterior hay que tener cuidado a la hora de tratar definiciones de tipos mutuamente recursivas, ya se pueden producir bucles infinitos [Louden97]. Este es el caso de la siguiente declaración en Pascal:

```
Type enlace = ^nodo;
       nodo = record
              dato : integer;
              siguiente: enlace
       end;
```

Supóngase, el siguiente programa en Pascal:

```
PROGRAM EquivalenciaTipos;
TYPE Estudiante = Record
       nombre,direccion: array[1.255] of char;
       edad: integer;
       end;
Colegio = Record
       nombre,direccion: array[1.255] of char;
       edad: integer;
       end;
VAR e:Estudiante;
    c:Colegio;
BEGIN
  e:=c;
END.
```

Un programador probablemente deseará que la anterior asignación sea catalogada por el analizador semántico como un error de compilación, ya que se ha asignado accidentalmente un registro colegio a un estudiante. Sin embargo, un sistema de tipos con equivalencia estructural de tipos (Algol-68 y Módula-3) no produciría un error semántico. Así, la **equivalencia de nombres** establece que todo tipo ha de tener un nombre único, considerando dos tipos equivalentes sólo si tienen el mismo nombre. Lenguajes con equivalencia de nombres son Java y Ada.

Ejemplo 55. El siguiente programa en Java demuestra que su sistema de tipos posee equivalencia de nombres:

```
interface IA {
  public void mA();
}

interface IB {
  public void mB();
}

class A implements IA, IB {
  public void mA() { System.out.println("A.mA"); }
  public void mB() { System.out.println("A.mB"); }
};

class B implements IA, IB {
  public void mA() { System.out.println("B.mA"); }
  public void mB() { System.out.println("B.mB"); }
};

public class EquivalenciaNombres {
  public static void main(String[] args) {
       A a=new B();
  }
}
```

La única sentencia del programa principal resulta un error semántico. Aunque la estructura de las dos clases `A` y `B` coincida, y ambas implementen los dos tipos `IA` e `IB`, no existe equivalencia al ser los nombres de los tipos distintos.

□

Existe un caso particular que se puede dar en aquellos lenguajes que permitan definir nuevos identificadores de tipo –como `type` en Pascal, Ada y Haskell, o `typedef` en C++. Dicho caso se da cuando un tipo se define igual que el nombre de otro tipo, estableciéndose así como un *alias* del segundo. De este modo, si un lenguaje con equivalencia de tipos basada en nombres identifica un alias de un tipo como equivalente al tipo al que hace referencia, se dice que posee **equivalencia de declaración**.

Ejemplo 56. En el siguiente programa en Modula-2:

```
MODULE EquivalenciaDeclaracion.
TYPE   celsius = REAL;
       fahrenheit = REAL;
       centigrados = celsius;
VAR
       c: celsius;
       f: fahrenheit;
       m: centigrados;
BEGIN
       f := c; (* Error *)
       m := c; (*  OK  *)
END    EquivalenciaDeclaracion.
```

Define tres tipos: `celsius` y `fahrenheit` (alias de real) y `centigrados` (alias de `celsius`). Modula-2 posee un sistema de tipos con equivalencia por declaración. Así, la

primera asignación no es correcta puesto `celsius` y `fahrenheit` no poseen igual nombre de tipo y, además, tampoco son *alias*. De forma contraria, la segunda asignación es correcta ya que el tipo de `m` (`centigrados`) es un alias del tipo de `c` (`celsius`). En el caso de que tuviese equivalencia de nombres estricta, la última asignación también sería descartada por el analizador semántico.

□

El lenguaje C++ (así como muchas implementaciones de Pascal) emplean equivalencia de declaración para clases, registros y uniones. ML posee equivalencia estructural, pero ofrece al programador la posibilidad de utilizar equivalencia por nombre. Si deseamos definir un tipo autor, como producto cartesiano de una cadena de caracteres (nombre) y un entero (código), podemos hacerlo de la siguiente forma:

```
type autor = string * int;
```

Este tipo es equivalente con cualquier producto de una cadena y un entero. Si lo que el programador desea es evitar dicha compatibilidad, podrá crear el nuevo tipo con equivalencia de nombres, mediante la siguiente sintaxis:

```
datatype autor = au of string * int;
```

6.7. Conversión y Coerción de Tipos

Considérese la expresión `x+i` donde `x` es una variable de tipo real e `i` es de tipo entero. Como un ordenador representa los reales y los enteros de forma distinta, la operación de suma es distinta en función de si ésta es de números reales o enteros. En el ejemplo expuesto, deberá convertirse el valor de `i` a su correspondiente valor real, previamente a llevar a cabo la operación suma.

La conversión de tipos entre valores puede ser llevada a cabo implícitamente si el compilador la realiza de un modo automático. Este tipo de conversión implícita recibe el nombre de **coerción**[63]. La mayoría de los lenguajes ofrecen coerción de tipos en aquellos contextos donde la conversión no supone pérdida alguna de información. La conversión implícita de números enteros donde se requiera un número real se da, por ejemplo, en C++, Pascal, Java y C#. En Ada[64] y Modula-2 no existe coerción alguna.

La coerción de tipos suele requerir que el generador de código produzca rutinas de conversión (e incluso de validación semántica) que serán ejecutadas por el programa en tiempo de ejecución. Esto hará que la ejecución del programa se vea ralentizada por la conversión dinámica de tipos. Si el lenguaje no posee un comprobador de tipos estático, la coerción supondrá un mayor coste computacional puesto que, además de la conversión, será necesario inferir el tipo en tiempo de ejecución. Por este motivo, la selección correcta de los tipos de las expresiones en un programa, puede mejorar sustancialmente el rendimiento de éste.

Ejemplo 57. La generación del siguiente programa en Microsoft Visual C++.NET 2003, sin optimización de código:

```
#include <iostream>
#include <ctime>
using namespace std;

int main() {
```

[63] En los lenguajes basados en C, es común denominar a la coerción de tipos *promoción*.
[64] A excepción de los subtipos que se puedan definir en el lenguaje.

```
  const unsigned max=30000;
  short in;
  cin>>in;
  long double id=in;
  long double v[max];
  long antes,despues;
  antes=clock();
  for (unsigned j=0;j<max;j++)
    for (unsigned i=0;i<max;i++) v[i]=in;
  despues=clock();
  cout<<(despues-antes)/(double)CLOCKS_PER_SEC*1000<<endl;
  antes=clock();
  for (unsigned j=0;j<max;j++)
    for (unsigned i=0;i<max;i++) v[i]=id;
  despues=clock();
  cout<<(despues-antes)/(double)CLOCKS_PER_SEC*1000<<endl;
  return 0;
}
```

Hace que el segundo bucle sea un 4,33% más rápido que el primero, gracias a que no tiene que efectuar conversiones (implícitas) de tipo en tiempo de ejecución.

□

La mayor parte de los lenguajes compilados tienen en la actualidad a reducir la coerción de tipos. Un ejemplo es la eliminación, por parte del lenguaje Java, de conversiones implícitas del C tales como la interpretación de enteros como valores lógicos. En Java existe el tipo `boolean` (`bool` en C++) que es incompatible (implícita y explícitamente) con cualquier otro tipo.

Por otro lado, existen diseñadores de lenguajes que opinan que la coerción de tipos ofrece un modo natural de ofrecer extensibilidad de las abstracciones, haciendo más sencilla la utilización de nuevos tipos en conjunción con los que ofrece el lenguaje. C++, en particular, ofrece mecanismos extremadamente ricos para establecer coerción de tipos. Al procurar un mecanismo tan rico de coerción, las reglas del sistema de tipos de este lenguaje hacen que dichos mecanismos sean complejos de entender y emplear correctamente.

Ejemplo 58. El siguiente programa en C++ define una clase `Entero`, con conversión implícita a dos tipos básicos del lenguaje: `int` y `double`.

```
#include <iostream>
using namespace std;

class Entero {
    int entero;
  public:
    Entero(int n) { entero=n; }
    operator double() const { return entero; }
    int getEntero() const { return entero; }
    Entero operator+(const Entero &e) const {
           return Entero(entero+e.entero);
    }
};

void fe(Entero n) { cout<<n.getEntero()<<endl; }
void fd(double n) { cout<<n<<endl; }
int main() {
    fe(3);
    Entero e(3);
    fd(e);
    return 0;
}
```

La primera conversión es por vía de su constructor. Al poseer éste un único parámetro, C++ reconoce una coerción del tipo de su parámetro (`int`) a los objetos de la clase. De este modo, la invocación `fe(3)` es totalmente correcta, ya que el `int` 3 se convierte, por

medio del constructor, a un objeto `Entero`. La segunda conversión se especifica con la redefinición del operador de ahormado (*cast*) a `double`. La invocación `fd(e)` produce una conversión implícita empleando el método que define la conversión.

Este mecanismo tan sofisticado puede, no obstante, dar lugar a ambigüedades. Si se evalúa la expresión `e+1`, ésta poseerá un error de compilación por ambigüedad:

- Podría tratarse de una conversión por medio del constructor: `e+Entero(1)`
- Podría significar una conversión a real: `(double)e+1`

□

En los lenguajes orientados a objetos la coerción por excelencia se da mediante el empleo de la herencia. Puesto que el conjunto de mensajes que se le puede lanzar a una superclase es también válido para cualquier subclase, se establece un mecanismo de conversión implícita en el sentido ascendente de la jerarquía. La herencia supone, así, un mecanismo de coerción de tipos.

Ejemplo 59. El lenguaje de programación Java, como todo lenguaje orientado a objetos, ofrece conversión implícita de tipos en sentido ascendente de una jerarquía de herencia. No posee herencia múltiple, pero identifica los tipos con el concepto de *interface*: una colección de mensajes que un objeto acepta. Nótese como el concepto de *interface* de Java es precisamente el punto de vista "basado en la abstracción" de tipo que introdujimos en § 6.2: el conjunto de operaciones que se puede realizar sobre el objeto. Una clase puede implementar (y por tanto existir coerción a) múltiples *interfaces* (tipos).

```
class Persona implements Cloneable {
  protected String nombre;
  public String getNombre() { return nombre; }
  public Persona(String n) { nombre=n; }
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
}

class Empleado extends Persona implements Comparable {
  private String departamento;
  public String getDepartamento() { return departamento; }
  public Empleado(String n,String d) {
    super (n);
    departamento=d;
    }
  public int compareTo(Object o) {
    if (o instanceof Empleado)
      return ((Empleado)o).nombre.compareTo(nombre);
    return -1;
    }
}

public class Coercion {
  public static void main(String[] args) throws Exception {
    Empleado e=new Empleado("Juan","Informática");
    System.out.println(e.getDepartamento());
    Persona p=e;
    System.out.println(e.getNombre());
    Cloneable c=e;
    Object o=e.clone();
    Comparable comp=e;
    System.out.println(comp.compareTo(c));
  }
}
```

En el programa anterior, el objeto accesible desde la referencia `e`, posee –además de `Object`[65]– la unión de los tipos `Empleado` (es una instancia de esta clase), `Persona` (hereda de ella), `Comparable` y `Clonable` (implementa ambos tipos). De este modo, las conversiones llevadas a cabo en el programa principal son todas implícitas.

□

En los casos en los que la conversión de una expresión a otro tipo distinto al que posee conlleve una posible pérdida de información, la conversión suele requerirse de un modo explícito. Una **conversión explícita** es aquélla que requiere que el programador escriba algo para motivar la conversión. En Modula-2 todas las conversiones son explícitas. La asignación:

```
r=i;
```

siendo `r` una variable real e `i` entera, genera en Java un error de compilación indicando que puede haber una pérdida de precisión. Para asumir la posible pérdida, el programador deberá realizar la conversión de un modo explícito, mediante el operador de ahormado (*cast*):

```
i=(int)r;
```

Ahora el programador asume el riesgo de perder la parte decimal del real y el compilador aceptará la conversión.

Las conversiones explícitas (al igual que las implícitas) suelen generar código adicional que será ejecutado en tiempo de ejecución. En algunos lenguajes con sistemas de tipos fuertes suelen ejecutar, adicionalmente, un código de validación de la conversión, verificando que no se produzca desbordamiento. Este tipo de conversiones explícitas siempre llevan a cabo una modificación de la representación interna de la información.

Sin embargo, existe en ocasiones la necesidad de convertir el tipo de una expresión, sin que se produzca una modificación de la representación interna de la información. Lo que se busca es poder aplicar operaciones de un tipo distinto al real, sin que se cambie la representación interna de la información. Esta posibilidad hace que un lenguaje, por definición, no sea seguro. Un posible escenario en el que puede interesar hacer conversiones explícitas, sin convertir la representación de la información, es en la implementación de un nuevo algoritmo de gestión de memoria.

Ejemplo 60. En el lenguaje C, el modo de hacer conversiones explícitas sin modificar la representación de los datos, es mediante los ahormados a punteros. Puesto que se permite la conversión entre punteros de distintos tipos, lo buscado se puede obtener del siguiente modo:

```
#include <iostream>

template<typename Tipo>
Tipo *nuevo() {
        const unsigned numeroBytes=100;
        static unsigned ultimo=0;
        static char memoria[numeroBytes];
        if (ultimo+sizeof(Tipo)<=numeroBytes) {
                ultimo+=sizeof(Tipo);
                return (Tipo*)(memoria+ultimo-sizeof(Tipo));
        }
        return 0;
}

int main() {
```

[65] En Java toda clase deriva de `Object`.

```
        unsigned max=0;
        int *p=nuevo<int>();
        *p=3;
        long double*q=nuevo<long double>();
        *q=34.65;
        std::cout<<*p<<'\t'<<*q<<std::endl;
        return 0;
}
```

El programa anterior implementa una función `nuevo` capaz de ofrecer memoria para cualquier tipo, con un tamaño limitado a 100 *bytes*[66]. Primero se aplica, en la función `nuevo`, la conversión de un puntero de caracteres a un puntero a cualquier tipo. Posteriormente, en el programa principal, se aplica el operador `*` para convertir el tipo sin modificar la representación interna de la información –en este caso, un *array* de caracteres.

□

Ejemplo 61. El lenguaje C++ sigue empleando el operador *cast* heredado del C. Sin embargo, para distinguir las distintas semánticas de dicho operador, ha creado distintas versiones del mismo. En este ejemplo compararemos su operador `static_cast` que efectúa una conversión de la representación interna del dato a convertir, con el `reinterpret_cast` que no realiza tal modificación.

```
#include <iostream>
using namespace std;

int main() {
        double d=33.44;
        int i1,i2;
        i1=static_cast<int>(d);
        i2=reinterpret_cast<int&>(d);
        cout<<i1<<'\t'<<i2<<endl;
        return 0;
}
```

El programa anterior hace que el entero `i1` posea el valor 33, ejecutándose el código de conversión en tiempo de ejecución. De forma contraria, el valor `i2` depende de la plataforma empleada, puesto que no modifica la representación interna del `double`, mostrando su valor como si de un entero se tratase.

□

Como hemos mencionado, en los lenguajes orientados a objetos es común tener un tipo general para cualquier objeto. En C++ es `void*`; en Modula-2, `address`; en Modula-3 `refany`; en Eiffel y Clu, `any`; en Java y C#, `Object`. Este tipo genérico es utilizado a la hora de implementar clases contenedoras de cualquier objeto (vectores, listas, pilas, colas...). La conversión de cualquier objeto es implícita, puesto que estas referencias, o bien no poseen ninguna operación (`void*`), o bien su conjunto de operaciones es tan reducido que lo posee todo objeto. De este modo, facilitan la implementación de colecciones de cualquier tipo.

Una vez empleemos una colección de referencias genéricas, surgirá la necesidad de recuperar el tipo del objeto, a la hora de extraer el elemento del contenedor. El problema es que el contenedor nos devolverá una referencia con el tipo general, en lugar de ser del tipo específico que habíamos insertado. Las operaciones que podremos solicitar a dicha referencia serán siempre menores a las que deseemos: necesitamos obtener su tipo original.

La cuestión de cómo recuperar el tipo de un objeto a partir de una referencia genérica, pasa por intentar ampliar el número de operaciones que sobre éste se puedan aplicar.

[66] En ISO/ANSI C++, el tamaño de un `char` es un *byte*. Los caracteres extendidos se obtienen con el tipo `wchar_t`.

Así, puesto que estamos tratando de ampliar el conjunto de operaciones permitidas, debemos efectuar la conversión de un modo explícito. Esta operación puede dar lugar a un error en tiempo de ejecución: al convertir una referencia genérica a una referencia concreta, puede suceder que ésta apunte a un objeto de otro tipo distinto al que se quiere convertir – ya que, al ser genérica, puede estar apuntando a cualquier objeto.

Ejemplo 62. En el lenguaje de programación Java, la sintaxis para llevar a cabo la recuperación de un tipo, de un modo descendente en la jerarquía, es mediante el operador *cast*.

```
import java.util.ArrayList;
public class DownCast {
  public static ArrayList crearVector() {
    ArrayList v=new ArrayList();
    for (int i=-10;i<=10;i++)
      v.add( new Integer(i) );
    return v;
    }
  public static void main(String[] a) {
    ArrayList v=crearVector();
    System.out.println(((Integer)v.get(0)).intValue());
    System.out.println(((Character)v.get(1)).charValue());
    }
  }
```

En el programa anterior la primera recuperación del tipo es correcta, mostrándose el valor del primer entero metido en el contenedor: -1. La segunda genera una excepción (`ClassCastException`) en tiempo de ejecución, finalizando el programa. No es posible convertir el tipo del segundo objeto a un tipo `Character`. Nótese cómo el método `charValue` no es una operación válida del objeto en cuestión.

Un programador de Java tiene múltiples mecanismos para conocer el tipo de un objeto en tiempo de ejecución. Distintos ejemplos son el operador `instanceof`, el método `getClass` o manejar la excepción `ClassCastException` en el ahormado al tipo deseado.

□

El modo en el que los lenguajes orientados a objetos permiten realizar esta conversión[67] de supertipo a subtipo (de general a específico) es haciendo que los objetos posean información de su tipo en tiempo de ejecución. De este modo, se podrá comprobar dinámicamente si el tipo a ahormar es correcto y, por tanto, si se pueden aplicar al objeto las operaciones propias de ese tipo.

Ejemplo 63. El lenguaje ISO/ANSI C++ introdujo información dinámica de tipos (RTTI, *RunTime Type Information*), para poder llevar a cabo la recuperación segura del tipo de un objeto.

```
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
      ~Base() {}
      virtual const char *quienSoy() {return "Base";}
};
class Derivada: public Base {
public:
      virtual const char *quienSoy() {return "Derivada";}
};
```

[67] Esta conversión suele denominarse *downcast*, por el sentido descendente que sigue en la jerarquía de herencia.

```
int main() {
      Base *base=new Derivada;
      Derivada *derivada=dynamic_cast<Derivada*>(base);
      if (derivada)
        cout<<"Conversión correcta.\n"<<
              "Quien es: "<<derivada->quienSoy()<<".\n"<<
              "Tipo: "<<typeid(*derivada).name()<<'.'<<endl;
      else
        cout<<"Conversión errónea.\n";
      return 0;
}
```

El operador `dynamic_cast` lleva a cabo la comprobación dinámica del tipo del objeto apuntado. Si éste posee un tipo compatible con el solicitado, devuelve un puntero al tipo demandado apuntando al objeto. En caso contrario devuelve 0. Otra característica de RTTI es el operador `typeid` que devuelve un objeto `type_info` con información respecto a su tipo. En el ejemplo anterior, la conversión es correcta y se muestra que el tipo es `Derivada`[68].

□

6.8. Sobrecarga y Polimorfismo

Un símbolo está **sobrecargado** cuando tiene distintos significados dependiendo de su contexto. Esta definición suele aplicarse a operadores y funciones. Un ejemplo común es el operador +, ya que puede representar suma de enteros, de reales o incluso de cadena de caracteres. Aunque para un humano la suma de enteros y reales puede tener el mismo significado, para una máquina no es así, ya que ejecuta operaciones distintas en cada caso. En los lenguajes Ada y Basic, el operador paréntesis está sobrecargado; `A(i)` puede significar tanto el acceso al elemento i-ésimo de un *array*, como la invocación a la función `A` con el parámetro `i`.

El proceso de conocer, de entre todos los posibles significados, el significado concreto de un símbolo sobrecargado se denomina **resolución** de la sobrecarga. La sobrecarga se resuelve en función del contexto en el que aparezca el símbolo sobrecargado, basándose en las reglas que especifica el sistema de tipos.

Ejemplo 64. La siguiente gramática libre de contexto representa un subconjunto de expresiones en la que el operador suma representa la adición de números enteros y reales:

(1) `expresion` → **`cte_entera`**
(2) `expresion` → **`cte_real`**
(3) $\texttt{expresion}_1$ → $\texttt{expresion}_2$ **`+`** $\texttt{expresion}_3$

Se define una gramática atribuida para calcular el código a generar, en la que se infiere el tipo de cada expresión:

P	R
(1)	`expresion.tipo = 'I'`
	`expresion.codigo = "PUSH_INT " +` **`cte_entera`**`.valor`
(2)	`expresion.tipo = 'F'`
	`expresion.codigo = "PUSH_FLT " +` **`cte_real`**`.valor`

[68] El modo de representar el tipo de un objeto en RTTI es dependiente de la implementación. Lo único que se ha de cumplir es que si dos objetos poseen el mismo tipo, la comparación de ambas devoluciones del operador `typeid` ha de ser cierta.

P	R
(3)	$expresion_1$.tipo=mayorTipo($expresion_2$.tipo,$expresion_3$.tipo) $expresion_1$.codigo=$expresion_2$.codigo + coercion($expresion_1$.tipo,$expresion_2$.tipo) + $expresion_3$.codigo + coercion($expresion_1$.tipo,$expresion_3$.tipo) + suma($expresion_1$.tipo)

```
char mayorTipo(char t1,char t2) {
 if ( t1 == 'F' || t2 == 'F' ) return 'F';
 return 'I';
}

String coercion(char t1,char t2) {
 if ( t1 == 'F' && t2 == 'I' ) return "INT2FLT";
 return "";
}

String suma(char tipo) {
 if ( t1 == 'F') return "ADD_FLT";
 return "ADD_INT";
}
```

Para la expresión 3+4+3.4, el código a generar (el valor del atributo `expresion.codigo`) es la consecución de las sentencias `PUSH_INT 3` (apilar un entero), `PUSH_INT 4`, `ADD_INT` (apilar los dos enteros en el tope de la pila), `INT2FLT` (convertir el entero en la pila a un real), `PUSH_FLT 3.4` (apilar un real), `ADD_FLT` (sumar los dos reales de la pila). Vemos cómo la gramática atribuida infiere el tipo y, basándose en éste, resuelve la sobrecarga: la primera suma se resuelve como una operación entera, cuando la segunda es real.

En este ejemplo se muestra la relación entre el concepto de sobrecarga y el de coerción de tipos, existente en la mayoría de lenguajes. Si un operador está sobrecargado es porque se define para distintos tipos de operandos. Si el operador es binario (como el caso de la suma), puede ser que los dos operandos posean el mismo tipo, pero también es común que no sea así. Es en ese caso (en nuestro ejemplo, la suma de un entero y un real y viceversa) es cuando, además de resolver la sobrecarga, es necesario que el compilador realice una conversión implícita de tipos.

□

El concepto de sobrecarga se emplea también para funciones y métodos, donde el mismo identificador se puede utilizar para implementaciones distintas. Comúnmente, el criterio necesario para sobrecargar un método o función implementado previamente es modificar el número de parámetros, o el tipo de alguno de ellos. En este contexto, la resolución de la sobrecarga supone conocer a qué método o función invocar. La resolución se llevará cabo a partir del número y tipo de los parámetros reales. El comprobador de tipos deberá tomar todos los tipos asociados a un identificador y resolver cuál es el correcto ante una invocación dada.

La sobrecarga añade complejidad a un sistema de tipos de un compilador. Si unimos a ésta otras características ya mencionadas, como la coerción de tipos, pueden darse contextos ambiguos en los que la sobrecarga no pueda ser resuelta, y se tenga que generar un error de compilación.

Ejemplo 65. El siguiente programa en C++:

```
#include <iostream>
using namespace std;

void f(double,float) { cout<<"f(double,float)\n"; }
void f(float,double) { cout<<"f(float,double)\n"; }
int main() {
      f(1.0,2.0);  // Error
      f(1.0f,2.0);  // f(float,double)
      f(1.0,2.0f);  // f(double,float)
      f(1.0f,2.0f); // Error
      return 0;
}
```

Posee dos líneas en las que las invocaciones a una de las funciones `f` son ambiguas: la primera y la última. Puesto que los parámetros han de ser `double` y `float`, o viceversa, la invocación con dos `double` o dos `float` es ambigua[69].

□

Un símbolo (operador o función) es **polimórfico** respecto al tipo si define una semántica independiente de los tipos a los que se aplique. Para que un símbolo sea polimórfico, deberá definir un único comportamiento para todos los tipos del sistema.

Un operador polimórfico del lenguaje C es el operador de dirección `&`. Este operador se puede aplicar a cualquier *lvalue* de un tipo T, devolviendo un puntero a T. La semántica no varía y se puede aplicar a determinados elementos, indistintamente del tipo que posean. En el lenguaje de programación ML, el operador de comparación de igualdad (=) es polimórfico, puesto que acepta dos expresiones del mismo tipo y devuelve un valor lógico. Sin embargo, los operadores de comparación <, <=, >= y > no son polimórficos puesto que, de forma contraria al operador de igualdad, no se pueden aplicar a cualquier tipo.

El modo de representar las expresiones de tipos polimórficas mediante la notación introducida por Alfred Aho (§ 6.3) es empleando **variables de tipo**: variables que representan, dentro de una expresión de tipo, cualquier aparición de otra expresión de tipo. Una variable de tipo sirve para representar cualquier tipo dentro de otra expresión de tipo. Para indicar que una variable de tipo podrá ser sustituida por cualquier tipo, se utiliza el cuantificador universal ∀. Así, la expresión de tipo del operador `&` del lenguaje C tendrá la siguiente expresión de tipo:

$$\forall\ \alpha.\ \alpha \rightarrow pointer(\alpha)$$

La expresión de tipos anterior indica que el operador `&` se puede aplicar sobre cualquier tipo del lenguaje y devuelve un puntero al tipo al que haya sido aplicado.

Ejemplo 66. En los entornos interactivos del lenguaje de programación ML[70], dado un símbolo el entorno nos muestra su expresión de tipo asociada. Así, podemos preguntarle a sistema por la expresión de tipo del operador =. El entorno mostrará la siguiente expresión de tipo:

```
'a * 'a -> bool
```

En ML, la expresión de tipo `'a` indica que `a` es una variable de tipo al que se le aplica el cuantificador universal –el apóstrofo. De este modo, el operador de igualdad en ML es polimórfico, y su tipo asociado es una función que recibe un producto de cualquier par de tipos, iguales entre sí, devolviendo un valor lógico. El tipo de datos al que se le puede aplicar el operador binario = puede variar en cada invocación; sin embargo, ambos operandos han de ser del mismo tipo –ya que en la expresión de tipo aparece la misma variable de tipo, `a`.

[69] En el lenguaje de programación C++, las constantes 1.0 y 2.0 son ambas `double`. Para que sean `float` hay que añadirles el sufijo `f` (o `F`); para que sean `long double`, el sufijo es `l` (o `L`).
[70] Como el sistema interactivo *Standard ML of New Jersey* (SML/NJ).

□

El polimorfismo de tipos también se aplica al concepto de función y método. Una función o método polimórfico es aquélla que es capaz de recibir y devolver parámetros de cualquier tipo. En muchos lenguajes de programación –como C++, Eiffel o Ada– esta característica es definida como genericidad[71]. El polimorfismo también es implementado por lenguajes funcionales como ML, Haskell o Miranda.

El principal beneficio de la utilización de funciones polimórficas es que permiten implementar algoritmos que manipulan estructuras de datos, independientemente del tipo de elemento que contengan. De este modo, las implementaciones pueden desarrollarse de un modo independiente al tipo, dejando al procesador de lenguaje la tarea de sustituir, en cada invocación, las variables de tipo por los tipos concretos.

Ejemplo 67. Supóngase que, en el lenguaje de programación C, se desea crear una función que nos devuelva la longitud de una lista enlazada de enteros. Una implementación sería la siguiente:

```
typedef struct s_l {
        int informacion;
        struct s_l *siguiente;
} lista;

unsigned longitud(lista *l) {
        unsigned lon=0;
        while (l) {
                lon++;
                l=l->siguiente;
        }
        return lon;
}
```

El principal problema de la función `longitud` es que depende del tipo a contener. En el caso de que quisiésemos calcular la longitud de una lista de reales, deberíamos implementar una nueva función puesto que el parámetro de la función posee por tipo una lista de enteros. Es fácil apreciar cómo esta limitación es demasiado "fuerte", ya que el algoritmo en ningún caso hace uso del tipo contenido por la lista; sin embargo, si variamos éste, el algoritmo deja de ser válido.

Una alternativa para el problema planteado es utilizar el tipo más general que posee C: `void*`. Recordemos que podemos convertir la dirección de cualquier tipo a `void*`. Sin embargo, puesto que este tipo es tan genérico que no posee ninguna operación, la recuperación del tipo es una tarea no segura (§ 6.7). Esto, además del código de conversión a añadir, hace que los programas puedan tener errores en tiempo de ejecución.

El lenguaje C++ añade la posibilidad de implementar estructuras y funciones polimórficas (genéricas), independientes del tipo. Así, la siguiente versión del programa en C++ es segura respecto al tipo:

```
template<typename Tipo>
struct Lista {
        int informacion;
        Lista<Tipo> *siguiente;
};

template<typename Tipo>
unsigned longitud(Lista<Tipo> *l) {
        unsigned lon=0;
        while (l) {
```

[71] El término polimorfismo en estos lenguajes, como veremos, es aplicado a un polimorfismo restringido basado en la herencia.

```
                lon++;
                l=l->siguiente;
            }
            return lon;
        }
```

El cuantificador universal en C++ se define mediante identificadores dentro de una plantilla (`template`). Nótese cómo, en esta versión, la función `longitud` es válida para cualquier lista, independientemente del tipo que contenga.

□

Un procesador de un lenguaje que ofrezca polimorfismo, deberá implementar un mecanismo de **unificación**: proceso de encontrar una sustitución para cada variable de tipo con otra expresión de tipo, acorde al empleo del operador o función polimórfico. Puesto que el operador `&` del lenguaje C posee la expresión de tipo:

$$\forall\ \alpha.\ \ \alpha \rightarrow \mathit{pointer}(\alpha)$$

Al utilizar este operador en la expresión `&i`, donde `i` es una variable entera, el algoritmo de unificación encontrará la sustitución de α por *integer* como válida y, por tanto, se determinará el tipo de la expresión `&i` como *pointer(integer)*. Las variables de tipo que reciben un valor tras aplicar una sustitución, se dice que son *instanciadas*.

Un mecanismo de unificación es una técnica potente. Además de su utilización en la inferencia de tipos polimórficos [Milner78], desarrolla un papel fundamental en el modelo computacional del lenguaje Prolog. Los algoritmos de unificación también se emplean para técnicas de reconocimiento o emparejamiento de patrones[72], ampliamente utilizados en lenguajes como Perl o awk.

Cabe mencionar que en los lenguajes orientados a objetos, el término polimorfismo suele emplearse para lo que realmente es polimorfismo de subtipos (herencia): poder referirse, mediante un supertipo, a cualquier tipo derivado. Puesto que todos los tipos derivados ofrecen las operaciones de un tipo base, el compilador acepta el tratamiento polimórfico de objetos derivados –mediante referencias a su supertipo. Nótese cómo esto es un subconjunto de la amplitud del concepto de polimorfismo, que implica un tratamiento genérico para cualquier tipo –no sólo para las clases que heredan de una dada.

6.9. Inferencia de Tipos

Aunque ya hemos utilizado el concepto de inferencia de tipos a lo largo de todo este libro, este término puede definirse como el problema de determinar el tipo de una construcción del lenguaje. Ésta es la tarea fundamental de un sistema de tipos. Hemos visto cómo existen lenguajes con inferencia estática de tipos (tiempo de compilación) e inferencia dinámica (tiempo de ejecución).

Como hemos descrito a lo largo del punto § 1 de este libro, la inferencia de tipos es un problema de naturaleza compleja en el que se ha de tener en cuenta:

- La representación interna de las expresiones de tipo del lenguaje.
- La de equivalencia de tipos que posea el lenguaje de programación –nominal, declarativa, estructural, o una combinación de éstas.
- Las distintas coerciones de tipos existentes en el lenguaje.

[72] *Pattern matching.*

- Las reglas que definen las conversiones explícitas –no factibles[73], de modificación de representación interna, o de reinterpretación de la información.
- La especificación de las distintas semánticas de los operadores sobrecargados, así como el modo de resolver la sobrecarga de funciones y métodos.
- El modo en el que se pueden construir estructuras de datos y funciones polimórficas, así como la especificación del algoritmo de unificación capaz de inferir los tipos.

Adicionalmente a estas características, existen otras más específicas de determinados lenguajes, como las listas variables de argumentos o los parámetros por omisión[74].

[73] Por ejemplo, en el lenguaje de programación Java, aunque se ahorme una expresión entera a otra lógica, esta conversión no se permite.
[74] Ambos presentes en el lenguaje C++.

CUESTIONES DE REVISIÓN

1. ¿Qué determina si una regla de un lenguaje es parte del análisis semántico o análisis sintáctico? Ponga un par de ejemplos para cada caso.
2. Defina semántica de un lenguaje de programación. Enumere los distintos tipos de lenguajes de especificación semántica que conozca.
3. ¿Por qué es imposible detectar ciertos errores en tiempo de compilación? Ponga tres ejemplos.
4. ¿Qué significa anotar o decorar un árbol sintáctico?
5. Defina y explique los conceptos de árbol sintáctico, árbol sintáctico abstracto, sintaxis concreta y sintaxis abstracta de un lenguaje de programación.
6. ¿Qué es un atributo de un símbolo gramatical? ¿Qué es una regla semántica?
7. ¿Qué es una gramática atribuida?
8. Indique las diferencias entre una gramática atribuida y una definición dirigida por sintaxis.
9. ¿Qué un atributo calculado en una producción?
10. ¿Cuál es la diferencia entre un atributo heredado y uno sintetizado? ¿Cómo se representa dicha diferencia en un árbol sintáctico?
11. ¿Qué es un compilador de una pasada? ¿Qué ventajas e inconvenientes ofrece respecto a los compiladores de varias pasadas?
12. Indique qué es un grafo de dependencias de una gramática atribuida.
13. ¿Qué es un ordenamiento topológico de un grafo de dependencias de una gramática atribuida? ¿Cuál es su principal utilidad?
14. ¿Con qué tipo de recorrido de un AST puede evaluarse una gramática S-atribuida? ¿Y una L-atribuida?
15. Indique las diferencias entre una definición dirigida por sintaxis y un esquema de traducción.
16. ¿Qué significa que una gramática sea S-atribuida y L-atribuida? ¿Qué implica y por qué es una faceta importante?
17. ¿Qué es una gramática atribuida completa? ¿Qué relación existe con el concepto de gramática atribuida bien definida? ¿Por qué es un concepto importante?
18. ¿Cuál es el significado de evaluar una gramática?
19. Indique, además las gramáticas atribuidas S y L-atribuidas, cuáles pueden ser evaluadas, a partir de su árbol sintáctico, con una única visita de cada uno de sus nodos.
20. Indique qué tipo de recorrido ha de efectuarse sobre un AST para poder evaluar una gramática L-atribuida visitando una sola vez cada nodo del árbol.
21. ¿Es posible convertir una gramática L-atribuida a S-atribuida? ¿Y en el sentido contrario? Razone ambas respuestas.

22. Indique cuales son los dos métodos principales empleados para evaluar una gramática atribuida. Explique ambos.
23. ¿Cuál es la diferencia entre una regla semántica y una rutina semántica?
24. Algunos compiladores realizan el análisis semántico al mismo tiempo que el sintáctico. Otros, en fase de análisis sintáctico, crean un AST y posteriormente realizan las comprobaciones semánticas. Indique las ventajas e inconvenientes de ambas alternativas.
25. ¿Bajo qué circunstancias puede calcularse un atributo heredado en una herramienta de análisis sintáctico ascendente?
26. ¿Cuáles son los principales objetivos de la existencia de tipos en los lenguajes de programación?
27. ¿De cuántas formas distintas podría definirse el concepto de tipo?
28. Defina comprobación de tipos estática y dinámica. Indique dos lenguajes que posea cada una de ellas.
29. Ponga un ejemplo de dos comprobaciones dinámicas de tipo que realice un lenguaje que usted conozca.
30. ¿Cuál es la diferencia entre equivalencia de tipos y compatibilidad de tipos? Ponga dos ejemplos.
31. Defina y relacione expresión de tipo, sistema de tipos y comprobador de tipos.
32. Explique las diferencias surgidas a la hora de representar expresiones de tipo mediante estructuras de árboles o grafos acíclicos dirigidos.
33. ¿Qué significa que un lenguaje sea fuerte respecto al tipo? ¿Y que sea seguro? Cite características del lenguaje C que hacen que no sea seguro.
34. ¿Qué relación existe entre el concepto de lenguaje seguro, comprobación estática de tipos y comprobación dinámica de tipos?
35. ¿Cómo suelen gestionar en tiempo de ejecución los errores de tipo los lenguajes seguros?
36. ¿Qué es la coerción y conversión de tipos?
37. ¿Qué significa que un símbolo esté sobrecargado?
38. Defina polimorfismo de tipos. Dónde se suele dar en los lenguajes de programación. Ponga dos ejemplos de cada caso.
39. Defina y explique los distintos tipos de equivalencia de tipos que conoce.
40. ¿Qué relación existe entre la sobrecarga de operadores y las coerciones de un lenguaje de programación?
41. Indique las diferencias existentes entre un operador polimórfico y un sobrecargado. Ejemplifíquelo con dos casos.
42. ¿Cite complejidades del problema de la inferencia de tipos en un lenguaje como el C++?
43. ¿Qué es un algoritmo de unificación? ¿En qué lenguajes se emplea?
44. Cite las características polimórficas del lenguaje C++.

EJERCICIOS PROPUESTOS

1. Dada la sentencia `3*(61+-3)`, válida para la gramática atribuida del Ejemplo 11, muestre el árbol sintáctico decorado y un AST alternativo. ¿Cuál sería su sintaxis abstracta?
2. Amplíe la gramática atribuida del Ejemplo 14 para que sea capaz de utilizar todas las bases de 2 a 16. La base se pospondrá al número entre llaves. Por defecto será base 10. Ejemplos `0110{2}`, `123{10}`, `af45{16}`, `981`.
3. Respecto a la gramática atribuida del Ejemplo 14. ¿Puede evaluarse con una única pasada, visitando una única vez cada nodo del árbol sintáctico? Si optamos por hacer un compilador de varias pasadas, ¿se podría decorar su AST con una sola visita de sus nodos?
4. Impleméntese, mediante el patrón de diseño *Visitor*, un evaluador de la gramática del Ejemplo 14.
5. Implemente una gramática atribuida capaz de traducir expresiones infijas a expresiones prefijas. Por ejemplo, traducir de "`a/2+10*c`" a "`+ / a 2 * 10 c`".
6. Escriba una gramática atribuida para calcula el valor real del número descrito por el no terminal `real` de la siguiente gramática:

```
real  → num . num
num → num digito
    | digito
digito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

7. Dada la siguiente gramática:

```
arbolbin → ( cte_entera arbolbin arbolbin )
         | nil
```

   Escriba una gramática atribuida para comprobar si el árbol binario es de búsqueda. Como ejemplo, `(2 (1 nil nil) (3 nil nil))` es un árbol de búsqueda, pero no lo es `(2 (0 nil nil) (1 nil nil))`.

8. Dada la siguiente gramática atribuida GA={G,A,R,B} siendo G y R:

P	G	
(1)	S →	Logico MasLogico
(2)	$MasLogico_1$ →	**AND** Logico $MasLogico_2$
(3)	$MasLogico_1$ →	**OR** Logico $MasLogico_2$
(4)	MasLogico →	**λ**
(5)	Logico →	$Termino_1$ **+** $Termino_2$
(6)	Logico →	$Termino_1$ **-** $Termino_2$
(7)	Logico →	Termino
(8)	Termino →	**CTE_ENTERA**
(9)	Termino →	**CTE_REAL**

P	R
(1)	S.tipo = MasLogico.tipo MasLogico.inicial = Logico.tipo
(2)	$MasLogico_2$.inicial = Logico.tipo $MasLogico_1$.tipo = ENTERO

P	R
(3)	MasLogico$_2$.inicial = Logico.tipo MasLogico$_1$.tipo = ENTERO
(4)	MasLogico.tipo = MasLogico.inicial
(5)	Logico.tipo = MayorTipo(Termino$_1$.tipo,Termino$_2$.tipo)
(6)	Logico.tipo = MayorTipo(Termino$_1$.tipo,Termino$_1$.tipo)
(7)	
(8)	Termino.tipo = ENTERO
(9)	Termino.tipo = REAL

	Tipo MayorTipo(Tipo t1,Tipo t2){
	  if (t1==REAL) || (t2===REAL) return REAL;
	  return ENTERO;
	}

y siendo B:

P	B
(2)	MasLogico$_1$.inicial==ENTERO Logico.tipo==ENTERO
(3)	MasLogico$_1$.inicial==ENTERO Logico.tipo==ENTERO

¿Es la gramática GA una gramática S-atribuida? ¿Es la gramática GA una gramática L-atribuida? ¿Pertenece la sentencia `3+4.5` al lenguaje definido por la gramática GA? ¿Pertenece la sentencia `3-2 OR 3+2.1` al lenguaje definido por la gramática GA? ¿Es la gramática GA una gramática atribuida completa? ¿Está GA bien definida?

9. Muestre el grafo de dependencias del ejercicio anterior.

10. Dada la siguiente gramática G libre de contexto:

P	G	
(1)	S →	logico masLogico
(2)	logico →	implica masImplica
(3)	implica →	**true**
(4)	implica →	**false**
(5)	masLogico$_1$ →	**AND** logico masLogico$_2$
(6)	masLogico$_1$ →	**OR** logico masLogico$_2$
(7)	masLogico →	**λ**
(8)	masImplica$_1$ →	**=>** implica masImplica$_2$
(9)	masImplica →	**λ**

Siendo la asociatividad de los tres operadores a izquierdas, defina una gramática atribuida que reconozca como sentencias pertenecientes a su lenguaje únicamente aquellas expresiones que sean evaluadas como ciertas. Evalúe la gramática ante la entrada:

`false OR true AND false OR true => false`

¿Pertenece el programa anterior al lenguaje definido por la gramática atribuida?

11. Indique un ordenamiento topológico del grafo de dependencias del Ejemplo 15 y un orden de ejecución de sus reglas semánticas.

12. Considere la siguiente gramática

[1] <L> → <M> <L> **b**
[2] |**a**
[3] <M> |**λ**

¿En qué orden se podrían evaluar sus reglas semánticas asociadas –en el caso de tenerlas– en una única pasada con un analizador sintáctico ascendente?

13. Dada la siguiente gramática libre de contexto:

```
[1] <S>          → <Elemento> <Elementos>
[2]               | λ
[3] <Elemento>   → A
[4]               | B
[5] <Elementos>  → <Elemento> <Elementos>
[6]               | λ
```

Especifíquese sobre ella una gramática atribuida bien definida que restrinja el lenguaje reconocido a aquellos programas que posean igual número de *tokens* A y B. Demuéstrese su condición de gramática completa.

14. Indique una representación de las expresiones de tipo de las clases del Pascal para los cuatro puntos que hemos identificado en el Ejemplo 41.

15. Dado el siguiente código C:

```
typedef struct {
      int a, b;
      } nodo, *ptr_nodo;
nodo aa[100];
ptr_nodo bb(int x, nodo y);
```

Escriba las expresiones de tipo para `aa` y `bb`.

16. Muestre la expresión de tipo de una función que tome como argumento un *array* indexado por enteros, con elementos de cualquier tipo, y devuelva un *array* cuyos elementos sean los elementos apuntados por los elementos del *array* dado.

17. Indique cómo los métodos de clase (`static`) y los métodos abstractos deberían ser tenidos en cuenta en la fase de análisis semántico de un procesador de lenguaje.

18. Identifique una expresión de tipo necesaria para llevar las comprobaciones de tipo del identificador `miFuncion`, para el siguiente código C++:

```
template <class T> T *miFuncion(T *t) { return *t; }
```

19. Implemente el Ejemplo 41, creando las expresiones de tipo mediante un DAG en lugar de empleando un árbol.

20. Identifique las expresiones de tipo necesarias y escriba una definición dirigida por sintaxis para realizar un comprobador de tipos de comparaciones de igualdad de un pseudopascal con las siguientes características:

- Tipos básicos `char` e `integer` y sus correspondientes constantes.
- Constructores de tipo `array` y ↑.
- Se permiten comparaciones de punteros si son de igual tipo.
- Se permiten comparaciones de arrays siempre que tengan igual rango y tipo.
- Los subrangos de los arrays serán números enteros (no caracteres).

Utilice para ello un objeto `tablaSimbolos` con los métodos `insertar` y `buscar`. El siguiente es un programa correcto:

```
VAR
        vector:array[1..10] of integer;
        puntero:^integer;
        pDoble:^^integer;
        v:^array[1..10] of ^char;
```

```
        w:array[1..10] of ^char;
BEGIN
        45=vector[3];
        pDoble^^=puntero^;
        puntero=pDoble^;
        puntero^=vector[puntero^];
        v^[puntero^]^='3';
        v^=w;
END.
```

Los siguientes son programas semánticamente incorrectos:

```
VAR
BEGIN
  '3'=3;
END.
```

```
VAR
  pChar:^char;
  pInt:^integer;
BEGIN
  pChar=pInt;
END.
```

```
VAR
  puntero:^integer;
  pDoble:^^integer;
BEGIN
  puntero=pDoble;
END.
```

```
VAR
  v:ARRAY[1..10] OF char;
BEGIN
  v['a']='a';
END.
```

```
VAR
  v:ARRAY[1..10] OF char;
BEGIN
  v[3]=3;
END.
```

```
VAR
  v:ARRAY[1..10] OF char;
  w:ARRAY[1..10] OF integer;
BEGIN
  v=w;
END.
```

```
VAR
  v:ARRAY[1..10] OF char;
  w:ARRAY[1..11] OF char;
BEGIN
  v=w;
END.
```

21. Implemente, mediante el patrón de diseño *Composite* y comprobador de tipos equivalente a la definición dirigida por sintaxis del ejercicio anterior.

22. Dada la siguiente gramática:

```
<S>        → <var> = <exp>
<var>      → ID
<exp>      → <term> <masTerm>
<term>     → <fact> <masFact>
<masTerm>  → + <term> <masTerm>
           | - <term> <masTerm>
           | λ
<fact>     → CTE_ENTERA
           | CTE_REAL
           | ( <exp> )
<masFact>  → * <fact> <masFact>
           | / <fact> <masFact>
           | λ
```

Diseñe sobre ella una definición dirigida por sintaxis para que el atributo `var.tipo` infiera el tipo adecuado, entero o real, en la asignación. Además, el atributo `var.valor` ha de poseer la evaluación de la expresión asignada, siguiendo las precedencias típicas en los lenguajes de programación.

23. Supóngase las declaraciones generadas por la siguiente gramática:

```
(1) D  → id L
(2) L  → , id L
(3) L  → : T
(4) T  → integer
(5) T  → real
```

Construya un esquema de traducción para introducir el tipo de cada identificador en una tabla de símbolos.

24. La siguiente expresión en el lenguaje C:

```
(t)-x
```

Puede significar dos cosas. Si `t` es un tipo (definido mediante `typedef`), se está ahormando el valor de la expresión `-x` al tipo. También puede tratarse de que `t` sea otra variable y, simplemente, tengamos una resta.

Describa cómo un analizador sintáctico puede resolver estas dos posibles interpretaciones. ¿Tiene sentido que el analizador semántico emplee en analizador léxico para resolver este problema?

A Evaluación de un AST

En este apéndice se muestra el código fuente del Ejemplo 7, en el que se creaba un AST de una expresión aritmética. El árbol se construía por medio de una especificación yacc/bison. Puesto que en el Ejemplo 29 se ampliaba el AST para implementar un procesador de lenguaje de múltiples pasadas, nos limitaremos a mostrar la segunda versión –algo mayor que la presentada en el Ejemplo 7.

El lenguaje de programación es C++.

A.1 Implementación del AST

ast.h

```
#ifndef _ast_h
#define _ast_h

#include "visitor.h"
#include <iostream>
#include <sstream>
using namespace std;

class Expresion {
public:
   virtual ~Expresion() {}
   virtual void aceptar(Visitor*) = 0;
   // * Atributos
   double valor;
   char tipo;
   ostringstream codigo;
};

class ExpresionUnaria: public Expresion {
   Expresion *operando;
   char operador;
   ExpresionUnaria(ExpresionUnaria&) {}
   ExpresionUnaria &operator=(ExpresionUnaria&) {return *this;}
public:
   ExpresionUnaria(char operador,Expresion *operando) {
      this->operador=operador;
      this->operando=operando;
   }
   ~ExpresionUnaria() { delete operando; }
   virtual void aceptar(Visitor *v) {v->visitar(this);}
   char getOperador() const { return operador; }
   Expresion *getOperando() { return operando; }
};


class ExpresionBinaria: public Expresion {
   Expresion *operando1,*operando2;
   char operador;
   ExpresionBinaria(ExpresionBinaria&) {}
   ExpresionBinaria &operator=(ExpresionBinaria&) {return *this;}
public:
   ExpresionBinaria(char operador,Expresion *operando1,Expresion *operando2) {
      this->operador=operador;
      this->operando1=operando1;
      this->operando2=operando2;
   }
   ~ExpresionBinaria() { delete operando1; delete operando2; }
   virtual void aceptar(Visitor *v) {v->visitar(this);}
   char getOperador() const { return operador; }
   Expresion *getOperando1() { return operando1; }
   Expresion *getOperando2() { return operando2; }
};
```

```
class ConstanteEntera: public Expresion {
public:
   ConstanteEntera(int v) {valor=v;}
   virtual void aceptar(Visitor *v) {v->visitar(this);}
};

class ConstanteReal: public Expresion {
public:
   ConstanteReal(double v) {valor=v;}
   virtual void aceptar(Visitor *v) {v->visitar(this);}
};

#endif
```

A.2 Visitas del AST

En este punto mostraremos cómo, mediante la utilización del patrón de diseño *Visitor*, es posible separar cada una de las visitas del AST de la representación del árbol. Las distintas visitas serán presentadas siguiendo el siguiente orden: semántico, generación de código, cálculo y mostrar (traza). Para cada una de ellas se mostrará la declaración e implementación de la clase. Comenzaremos con la clase abstracta `Visitor`.

visitor.h

```
#ifndef _visitor_h
#define _visitor_h

class ExpresionUnaria;
class ExpresionBinaria;
class ConstanteEntera;
class ConstanteReal;
class Visitor {
public:
   virtual void visitar(ExpresionUnaria *) = 0;
   virtual void visitar(ExpresionBinaria *) = 0;
   virtual void visitar(ConstanteEntera *) = 0;
   virtual void visitar(ConstanteReal *) = 0;
};

#endif
```

visitorsemantico.h

```
#ifndef _visitorsemantico_h
#define _visitorsemantico_h

#include "ast.h"

class VisitorSemantico: public Visitor{
    static char tipoMayor(char tipo1,char tipo2);
public:
   void visitar(ExpresionUnaria *) ;
   void visitar(ExpresionBinaria *);
   void visitar(ConstanteEntera *) ;
   void visitar(ConstanteReal *) ;
};

#endif
```

visitorsemantico.cpp

```
#include "visitorsemantico.h"

char VisitorSemantico::tipoMayor(char tipo1,char tipo2) {
   // 'E' -> error
   // 'I' -> entero
   // 'F' -> real
   if (tipo1=='E'||tipo2=='E') return 'E';
   if (tipo1=='F'||tipo2=='F') return 'F';
   return 'I';
}

void VisitorSemantico::visitar(ExpresionUnaria *e) {
   // * Calculo el tipo del operando
   e->getOperando()->aceptar(this);
   // * Calculo el tipo del nodo
   e->tipo=e->getOperando()->tipo;
}
```

```
void VisitorSemantico::visitar(ExpresionBinaria *e) {
   // * Calculo el tipo de los operandos
   e->getOperando1()->aceptar(this);
   e->getOperando2()->aceptar(this);
   // * Calculo el tipo del nodo
   e->tipo=tipoMayor(e->getOperando1()->tipo,e->getOperando2()->tipo);
   // * Error semántico
   if (e->getOperador()=='=' &&
      e->getOperando1()->tipo=='I' && e->getOperando2()->tipo=='F')
      e->tipo='E';
}

void VisitorSemantico::visitar(ConstanteEntera *c)  {
   c->tipo='I';
}

void VisitorSemantico::visitar(ConstanteReal *c)  {
   c->tipo='F';
}
```

visitorgc.h

```
#ifndef _visitorgc_h
#define _visitorgc_h

#include "ast.h"

class VisitorGC: public Visitor{
public:
   void visitar(ExpresionUnaria *) ;
   void visitar(ExpresionBinaria *);
   void visitar(ConstanteEntera *) ;
   void visitar(ConstanteReal *) ;
};

#endif
```

visitorgc.cpp

```
#include "visitorgc.h"
#include <cassert>

void VisitorGC::visitar(ExpresionUnaria *e) {
   assert(e->getOperador()=='-');
   // * Genero el código del operando
   e->getOperando()->aceptar(this);
   e->codigo<<e->getOperando()->codigo.str();
   // * Genero el código de negación
   e->codigo<<"\t"<<e->tipo<<"NEG\n";  // * Instrucción INEG o FNEG
}

void VisitorGC::visitar(ExpresionBinaria *e) {
   // * Código para apilar el primer operando
   e->getOperando1()->aceptar(this);
   e->codigo<<e->getOperando1()->codigo.str();
   // * ¿Es necesario convertir el op1 de entero a real?
   if (e->getOperando1()->tipo=='I' && e->tipo=='F' )
      e->codigo<<"\tITOF\n"; // * Integer to Float
   // * Código para apilar el segundo operando
   e->getOperando2()->aceptar(this);
   e->codigo<<e->getOperando2()->codigo.str();
   // * ¿Es necesario convertir el op2 de entero a real?
   if (e->getOperando2()->tipo=='I' && e->tipo=='F' )
      e->codigo<<"\tITOF\n"; // * Integer to Float
   // * Tipo del operador
   e->codigo<<'\t'<<e->tipo; // I o F
   // * Operador
   switch (e->getOperador()){
      case '+': e->codigo<<"ADD\n"; break;
      case '-': e->codigo<<"SUB\n"; break;
      case '*': e->codigo<<"MUL\n"; break;
      case '/': e->codigo<<"DIV\n"; break;
      case '=': e->codigo<<"STORE\n"; break;
      default: assert(0);
   }
   if (e->getOperando1()->tipo=='I' && e->getOperando2()->tipo=='I')
      e->valor=(int)e->valor;
}

void VisitorGC::visitar(ConstanteEntera *c)  {
   // * Apila un entero
   c->codigo<<"\tPUSHI\t"<<(int)(c->valor)<<"\n";
}

void VisitorGC::visitar(ConstanteReal *c)  {
```

```
   // * Apila un real
   c->codigo<<"\tPUSHF\t"<<c->valor<<"\n";
}
```

visitorcalculo.h

```
#ifndef _visitorcalculo_h
#define _visitorcalculo_h

#include "ast.h"

class VisitorCalculo: public Visitor{
public:
   void visitar(ExpresionUnaria *) ;
   void visitar(ExpresionBinaria *);
   void visitar(ConstanteEntera *) ;
   void visitar(ConstanteReal *) ;
};

#endif
```

visitorcalculo.cpp

```
#include "visitorcalculo.h"
#include <cassert>

void VisitorCalculo::visitar(ExpresionUnaria *e) {
   assert(e->getOperador()=='-');
   // * Calculo el valor del operando
   e->getOperando()->aceptar(this);
   e->valor= - e->getOperando()->valor;
}

void VisitorCalculo::visitar(ExpresionBinaria *e) {
   // * Calculo el valor de los operandos
   e->getOperando1()->aceptar(this);
   e->getOperando2()->aceptar(this);
   // * Calculo el valor del nodo
   switch (e->getOperador()){
      case '+': e->valor=e->getOperando1()->valor+e->getOperando2()->valor; break;
      case '-': e->valor=e->getOperando1()->valor-e->getOperando2()->valor; break;
      case '*': e->valor=e->getOperando1()->valor*e->getOperando2()->valor; break;
      case '/': e->valor=e->getOperando1()->valor/e->getOperando2()->valor; break;
      case '=': e->valor=e->getOperando2()->valor; break;
      default: assert(0);
   }
   if (e->getOperando1()->tipo=='I' && e->getOperando2()->tipo=='I')
      e->valor=(int)e->valor;
}

void VisitorCalculo::visitar(ConstanteEntera *c)  {}

void VisitorCalculo::visitar(ConstanteReal *c)  {}
```

visitormostrar.h

```
#ifndef _visitormostrar_h
#define _visitormostrar_h

#include "ast.h"
#include <iostream>

class VisitorMostrar: public Visitor{
   std::ostream &flujo;
public:
   VisitorMostrar(std::ostream &f):flujo(f) {}
   void visitar(ExpresionUnaria *) ;
   void visitar(ExpresionBinaria *);
   void visitar(ConstanteEntera *) ;
   void visitar(ConstanteReal *) ;
};

#endif
```

visitormostrar.cpp

```
#include "visitormostrar.h"
#include <iostream>

using namespace std;

void VisitorMostrar::visitar(ExpresionUnaria *e) {
   flujo<<"( "<<e->getOperador()<<' ';
   e->getOperando()->aceptar(this);
```

```
   flujo<<" )";
}

void VisitorMostrar::visitar(ExpresionBinaria *e){
   flujo<<"( "<<e->getOperador()<<' ';
    e->getOperando1()->aceptar(this);
   flujo<<' ';
   e->getOperando2()->aceptar(this);
   flujo<<" )";
}

void VisitorMostrar::visitar(ConstanteEntera *c) {
   flujo<<(int)(c->valor);
}

void VisitorMostrar::visitar(ConstanteReal *c) {
   flujo<<c->valor;
}
```

A.3 Especificación Léxica y Sintáctica del Lenguaje

Para finalizar el ejemplo, se muestra la especificación léxica y sintáctica realizada con las herramientas `plclex` y `pcyacc`.

sintac.y

```
%{
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include "ast.h"
int yyparse();
int yylex();
Expresion *ast;
%}
%union {
  int entero;
  double real;
  Expresion *expresion;
  }
%right '='
%left '+' '-'
%left '*' '/'
%right MENOS_UNARIO
%nonassoc '(' ')'

%token <entero> CTE_ENTERA
%token <real> CTE_REAL
%token MENOS_UNARIO
%type <expresion> expresion
%%
programa: expresion { ast=$1;}
   ;
expresion: expresion '+' expresion {$$=new ExpresionBinaria('+',$1,$3); }
         | expresion '-' expresion {$$=new ExpresionBinaria('-',$1,$3); }
         | expresion '*' expresion {$$=new ExpresionBinaria('*',$1,$3); }
         | expresion '/' expresion {$$=new ExpresionBinaria('/',$1,$3); }
         | expresion '=' expresion {$$=new ExpresionBinaria('=',$1,$3); }
         | '-' expresion %prec MENOS_UNARIO {$$=new ExpresionUnaria('-',$2); }
         | '(' expresion ')' { $$=$2; }
         | CTE_ENTERA { $$=new ConstanteEntera($1); }
         | CTE_REAL { $$=new ConstanteReal($1); }
         ;
%%
void yyerror(char *s) {
  printf(s);
  }

#include "visitorsemantico.h"
#include "visitorgc.h"
#include "visitorcalculo.h"
#include "visitormostrar.h"
#include <iostream>
using namespace std;

int main() {
  yyparse();
  VisitorSemantico semantico;
  ast->aceptar(&semantico);
  if (ast->tipo=='E')
     cerr<<"El programa no es semánticamente válido."<<endl;
  else {
```

```
    cout<<"\nAnálisis semántico finalizado correctamente.\n";
    VisitorCalculo calculo;
    ast->aceptar(&calculo);
    cout<<"\nValor de la expresión: "<<ast->valor<<endl;
    VisitorGC gc;
    ast->aceptar(&gc);
    cout<<"\nCódigo generado:\n"<<ast->codigo.str()<<endl;
    }
  VisitorMostrar traza(cout);
  cout<<"\nAST generado:\n";
  ast->aceptar(&traza);
  cout<<endl;
  delete ast;
  }
```

lexico.l

```
%{
#include <string.h>
#include <stdlib.h>
#include "ast.h"
#include "yytab.h"

unsigned yylineno=1;
void errorLexico(char);
void inicializarLexico(char *,char *);
%}
comentario \/\/.*\n
linea      \n
letra      [a-zA-Z]
numero     [0-9]
real       {numero}+(\.{numero}+)?
alfa       [{numero}{letra}]
read       [Rr][Ee][Aa][Dd]
write      [Ww][Rr][Ii][Tt][Ee]
main       [Mm][Aa][Ii][Nn]
integer    [Ii][Nn][Tt][Ee][Gg][Ee][Rr]
%%
[ \t]+         ; // * Se ignoran los espacions y tabuladores
{comentario}   { yylineno++; }
{linea}        { yylineno++; }
{numero}+      { yylval.entero=atoi(yytext); return CTE_ENTERA; }
{real}([Ee][\+\-]?{numero}+)? { yylval.real=atof(yytext); return CTE_REAL; }
\+             |
\-             |
\*             |
\/             |
\(             |
\)             |
%              |
=              |
\}             |
\{             |
,              |
;              { return yytext[0]; } // Token=Caracter ASCII
.              { errorLexico(yytext[0]); }  // Caracter no perteneciente al lenguaje
%%

void errorLexico(char c) {
  printf("Error lexico en linea %d. Caracter %c no perteneciente al lenguaje.\n",
      yylineno,c);
  exit(-1);
  }

void inicializarLexico(char *fOrigen,char *fDestino) {
  yyin=fopen(fOrigen,"r");
  if (yyin==NULL) {
    printf("Error abriendo el fichero %s.\n",fOrigen);
    exit(-1);
    }
  yyout=fopen(fDestino,"w");
  if (yyout==NULL) {
    printf("Error abriendo el fichero %s.\n",fDestino);
    exit(-1);
    }
  }
```

B Evaluación de una Gramática L-Atribuida mediante un Analizador Descendente Recursivo

En el Ejemplo 28 se presentó un mecanismo para poder evaluar una gramática L-atribuida en una única pasada. Para ello, se establecía un esquema de traducción de las reglas semánticas a código. La ejecución de las reglas se procesaba al mismo tiempo que la tarea de reconocer sintácticamente el lenguaje, mediante un analizador descendente recursivo predictivo. La gramática del ejemplo es LL1, resultado muy sencillo el desarrollo del analizador sintáctico.

La implementación se ha realizado en el lenguaje de programación Java, distribuyéndose en tres paquetes: sintáctico, léxico y errores. Éste será el orden en el que se presentará el código fuente de cada uno de los módulos.

B.1 Módulo Sintáctico

Sintactico.java

```
package latribuida.sintactico;

/**
 * <p>Title: Implementanción LAtribuida</p>
 * <p>Description: Ejemplo de Implementacion de un Evaluador Descendente Recursivo
 *  LL1, sobre una gramática LAtribuida</p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Universidad de Oviedo</p>
 * @author Francisco Ortín
 * @version 1.0
 */

import latribuida.lexico.*;

/** Clase que ofrece toda la funcionalidad del analizador sintáctico */
public class Sintactico {
  /** Analizador léxico */
  private Lexico lexico;

  /** Constructor con el archivo de entrada */
  public Sintactico(String nombreFichero) {
    lexico=new Lexico(nombreFichero);
  }

  /** Constructor con entrada estándar */
  public Sintactico() { lexico=new Lexico(); }

  /** Método que reconoce sintácticamente el no terminal "expresion".<br/>
   * Producción: expresion -> terminos masTerminos<br/>
   * Además evalúa los atributos calculados en las reglas semánticas
   * de esta producción. Recibe los atributos heredados (ninguno) y
   * devuelve los sintetizados (expresion.valor)<br/>
   */
  public int expresion() {
    // * Regla: masTerminos.operando1 = termino.valor
    int masTerminosOperando1=termino();
    // expresion.valor = masTerminos.valor
    return masTerminos(masTerminosOperando1);
  }

  /** Método que reconoce sintácticamente el no terminal "masTerminos".<br/>
   * Produccion: masTerminos1 -> '+' termino masTerminos2<br/>
   * Produccion: masTerminos1 -> '-' termino masTerminos2<br/>
   * Produccion: masTerminos1 ->  lambda<br/>
   * Además evalúa los atributos calculados en las reglas semánticas
   * de esta producción. Recibe los atributos heredados (masTerminos.operando1)
```

```
   * y devuelve los sintetizados (masTerminos.valor)<br/>
   */
  private int masTerminos(int operando1) {
  int token=lexico.getToken();
  switch (token) {
    case '+': lexico.match('+');
      // * Regla: masTerminos2.operando1=masTerminos1.operando1+termino.valor
      int masTerminos2Operando1=masTerminos(operando1+termino());
      // * Regla: masTerminos1.valor = masTerminos2.valor
      return masTerminos2Operando1;
    case '-': lexico.match('-');
      // * Regla: masTerminos2.operando1=masTerminos1.operando1-termino.valor
      masTerminos2Operando1=masTerminos(operando1-termino());
      // * Regla: masTerminos1.valor = masTerminos2.valor
      return masTerminos2Operando1;
    default: // * lambda
      // * Regla: masTerminos1.valor = masTerminos1.operando1
      return operando1;
    }
  }

  /** Método que reconoce sintácticamente el no terminal "termino".<br/>
   * Produccion: termino -> factor masFactores<br/>
   * Además evalúa los atributos calculados en las reglas semánticas
   * de esta producción. Recibe los atributos heredados (ninguno)
   * y devuelve los sintetizados (termino.valor)<br/>
   */
  private int termino() {
    // * Regla: masFactores.operando1 = factor.valor
    int masFactoresOperando1=factor();
    // * Regla: termino.valor = masFactores.valor
    return masFactores(masFactoresOperando1);
  }

  /** Método que reconoce sintácticamente el no terminal "masFactores".<br/>
   * Produccion: masFactores1 -> '*' factor masFactores2<br/>
   * Produccion: masFactores1 -> '/' factor masFactores2<br/>
   * Produccion: masFactores1 ->  lambda<br/>
   * Además evalúa los atributos calculados en las reglas semánticas
   * de esta producción. Recibe los atributos heredados (masFactores.operando1)
   * y devuelve los sintetizados (masFactores.valor)<br/>
   */
  private int masFactores(int operando1) {
  int token=lexico.getToken();
  switch (token) {
    case '*': lexico.match('*');
      // * Regla: masFactores2.operando1=masmasFactores1.operando1+factor.valor
      int masFactores2Operando1=masFactores(operando1*factor());
      // * Regla: masFactores1.valor = masFactores2.valor
      return masFactores2Operando1;
    case '/': lexico.match('/');
      // * Regla: masFactores2.operando1=masFactores1.operando1-factor.valor
      masFactores2Operando1=masFactores(operando1/factor());
      // * Regla: masFactores1.valor = masFactores2.valor
      return masFactores2Operando1;
    default: // * lambda
      // * Regla: masFactores1.valor = masFactores1.operando1
      return operando1;
    }
  }

  /** Método que reconoce sintácticamente el no terminal "factor".<br/>
   * Produccion: factor -> CTE_ENTERA<br/>
   * Además evalúa los atributos calculados en las reglas semánticas
   * de esta producción. Recibe los atributos heredados (ninguno)
   * y devuelve los sintetizados (factor.valor)<br/>
   */
  private int factor() {
    int cteEnteraValor=lexico.getYYlval().entero;
    lexico.match(lexico.CTE_ENTERA);
    // * Regla: factor.valor = CTE_ENTERA.valor
    return cteEnteraValor;
  }

  /** Entrada del programa.<br/>
   * Sin argumentos: recibe la sentencia por la entrada estándar.
   * Un argumento: archivo de entrada del evaluador.
   */

  public static void main(String[] args) {
    Sintactico sintactico=null;
    if (args.length>=1) sintactico=new Sintactico(args[0]);
    else sintactico=new Sintactico();
    // * Nos limitamos a llamar al símbolo inical
    System.out.println("Valor de la expresión: "+
                    sintactico.expresion() );
```

```
    }
}
```

B.2 Módulo Léxico

Atributo.java

```
package latribuida.lexico;

/**
 * <p>Title: Implementanción LAtribuida</p>
 * <p>Description: Ejemplo de Implementacion de un Evaluador Descendente Recursivo LL1,
sobre una gramática LAtribuida</p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Universidad de Oviedo</p>
 * @author Francisco Ortín
 * @version 1.0
 */

/** Distintos atributos de los símbolos terminales */
public class Atributo {
  public final int entero;
  public final char carácter;
  public Atributo(int entero) { this.entero=entero; carácter='?'; }
  public Atributo(char carácter) { this.carácter=carácter; entero=-1; }
}
```

Lexico.java

```
package latribuida.lexico;

/**
 * <p>Title: Implementanción LAtribuida</p>
 * <p>Description: Ejemplo de Implementacion de un Evaluador Descendente Recursivo LL1,
sobre una gramática LAtribuida</p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Universidad de Oviedo</p>
 * @author Francisco Ortín
 * @version 1.0
 */

import java.io.*;

/** Clase que ofrece toda la funcionalidad del analizador léxico */
public class Lexico {
  /** Flujo de entrada */
  private PushbackReader in;
  /** Número de línea */
  private int yylineno=1;
  /** Número de línea */
  public int getYYlineno() {return yylineno; }
  /** Tokens */
  public static final int CTE_ENTERA=256;
  /** Valor semántico del token */
  private Atributo yylval;
  /** Atributo del token */
  public Atributo getYYlval() { return yylval; }

  /** Atributo que indica el token actual */
  private int token;
  /** Método que devuelve el token actual */
  public int getToken() { return token; }
  /** Método que comprueba que el siguiente token es el pasado. Si así
   * es, avanza el token actual al siguiente. */
  public void match(int token) {
    if (this.token==token) {
      // * Avanzamos
      this.token=yylex();
      return;
    }
    latribuida.errores.Error.esperaba(yylineno,token,this.token);
  }

  /** Construye un analizador léxico sobre la entrada estándar */
  public Lexico() {
    in=new PushbackReader(new InputStreamReader(System.in));
    token=yylex(); // * Leemos el primer token
  }

  /** Construye un analizador léxico sobre el nombre del fichero pasado */
  public Lexico(String fichero) {
    try {
```

```
      this.in=new PushbackReader(new FileReader(fichero));
      token=yylex(); // * Leemos el primer token
    } catch(Exception e) {
      System.err.println("El fichero "+fichero+" no se encuentra.");
      e.printStackTrace();
      System.exit(-1);
    }
  }

  private int getCar() {
        int ch=0;
        try { ch=in.read(); }
        catch (Exception e) {
                System.err.println("No se puede leer de disco.\n");
                e.printStackTrace();
                System.exit(-2);
        }
        if (ch=='\n') yylineno++;
        return ch;
  }

  private void putCar(int c) {
        try {  in.unread(c); }
        catch (Exception e) {
                System.err.println("Buffer lleno.\n");
                e.printStackTrace();
                System.exit(-3);
        }
        if (c=='\n') yylineno--;
        }

 /** Método que devuelve el siguiente token.<br/>
  * Este método es privado. El analizador sintáctico deberá utilizar los
  * métodos getToken y match. */
  private int yylex() {
        int ch=0;
        ch=getCar();
        // * Basura
        while (ch==' '||ch=='\t'||ch=='\n'||ch=='\r') ch=getCar();
        // * Número
        if (ch>='0'&&ch<='9') {
                StringBuffer s=new StringBuffer();
                while (ch>='0'&&ch<='9') {
                        s.append((char)ch);
                        ch=getCar();
                        }
                putCar(ch);
                yylval=new Atributo( Integer.parseInt(s.toString()) );
                return CTE_ENTERA;
                }
        // * Cualquier otro carácter
        yylval=new Atributo((char)ch);
        return ch;
    }


    /** Método de prueba de la clase */
    public static void main(String[] args) {
      Lexico lexico=null;
      if (args.length>=1) lexico = new Lexico();
      else lexico=new Lexico();
      int token;
      while ((token=lexico.yylex()) != -1 ) {
         System.out.println("Número de línea: "+lexico.getYYlineno());
         System.out.println("\tNúmero de token: "+token);
         System.out.print("\tAtributo:");
         if (token==CTE_ENTERA)
           System.out.println(lexico.getYYlval().entero);
         else System.out.println(lexico.getYYlval().carácter);
      }
    }
}
```

B.3 Módulo Errores

Error.java

```
package latribuida.errores;

/**
 * <p>Title: Implementanción LAtribuida</p>
 * <p>Description: Ejemplo de Implementacion de un Evaluador Descendente Recursivo LL1,
sobre una gramática LAtribuida</p>
```

```
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Universidad de Oviedo</p>
 * @author Francisco Ortín
 * @version 1.0
 */

import latribuida.lexico.Lexico;
/** Clase que gestiona el módulo gestor de errores */
public class Error {

  /** Mensaje que muestra que esperaba otro terminal en lugar de otro recibido */
  static public void esperaba(int nLínea, int tokenEsperado, int tokenRecibido) {
    System.err.println("Error en línea:" +nLínea);
    System.err.println("\tEsperaba un "+aCadena(tokenEsperado)+
                       ", y encontré un "+aCadena(tokenRecibido));
    System.exit(-2);
  }

  /** Traduce el código numérico de un token a una cadena para que
   * el usuario la pueda entender.
   */
  private static String aCadena(int token) {
    if (token==Lexico.CTE_ENTERA) return "constante entera";
    // * Es un carácter
    else return "\'"+(char)token+"\'";
  }
}
```

C Comprobador de Tipos

En el Ejemplo 42 se introdujo cómo, empleando el patrón de diseño *Composite*, pueden ser modeladas las expresiones de tipo de un lenguaje de programación. La primera parte de este apéndice muestra la implementación de éstas en el lenguaje C++.

Las expresiones de tipo eran, posteriormente, empleadas en el Ejemplo 44 para implementar un sistema y comprobador de tipos de un subconjunto del lenguaje Pascal. El segundo punto de este apéndice muestra la implementación del sistema de tipos implementado en una única pasada, mediante la utilización de `pclex` y `pcyacc`.

C.1 Expresiones de Tipo

tipos.h

```
#ifndef _tipos_h
#define _tipos_h

#include <string>
#include <map>
#include <vector>
#include <iostream>
#include <cassert>
using namespace std;

// * Clase componente del patrón Composite
class ExpresionTipo {
public:
   virtual ~ExpresionTipo() {}
   // * Para el generador de código
   virtual unsigned getBytes() const = 0;
   // * Por depuración
   virtual string expresionTipo() const = 0;
   // * Métodos de análisis semántico, con comportamiento predefinido
   virtual ExpresionTipo *flecha();
   virtual ExpresionTipo *corchete(const ExpresionTipo*);
   virtual ExpresionTipo *parentesis(const vector<ExpresionTipo*>&);
   virtual ExpresionTipo *punto(const string&);
   // * Equivalencia de expresiones de tipo
   virtual bool equivalente(const ExpresionTipo *) const;
};

// * Clase hoja del patrón Composite: tipo simple Integer
class Integer: public ExpresionTipo {
   Integer(Integer&) {}
   Integer &operator=(Integer&) { return *this; }
public:
   Integer() {}
   // * Para el generador de código
   unsigned getBytes() const { return 4; }
   // * Por depuración
   string expresionTipo() const { return "integer"; }
};


// * Clase hoja del patrón Composite: tipo simple Char
class Char: public ExpresionTipo {
   Char(Char&) {}
   Char&operator=(Char&) { return *this; }
public:
   Char() {}
   // * Para el generador de código
   unsigned getBytes() const { return 1; }
   // * Por depuración
   string expresionTipo() const { return "char"; }
};
```

```
// * Clase hoja del patrón Composite: tipo simple Void
class Void: public ExpresionTipo {
   Void(Void&) {}
   Void&operator=(Void&) { return *this; }
public:
   Void() {}
   // * Para el generador de código
   unsigned getBytes() const { return 0; }
   // * Por depuración
   string expresionTipo() const { return "void"; }
};

// * Clase hoja del patrón Composite: tipo simple Error
class Error: public ExpresionTipo {
   Error(Error&) {}
   Error&operator=(Error&) { return *this; }
   string mensaje;
   unsigned numeroLinea;
public:
   Error(const string &s) {
      mensaje=s; extern unsigned yylineno; numeroLinea=yylineno; }
   // * Para el generador de código
   unsigned getBytes() const { assert(0); return 0; }
   // * Por depuración
   string expresionTipo() const { return "error"; }
   friend ostream &operator<<(ostream &,const Error &);
};

inline ostream &operator<<(ostream &o,const Error &e) {
   return o<<"Error en la línea "<<e.numeroLinea<<". "<<e.mensaje;
}

// * Clase compuesta del patrón Composite: tipo compuesto Pointer
class Pointer: public ExpresionTipo {
   Pointer(Pointer&) {}
   Pointer &operator=(Pointer&) { return *this; }
   ExpresionTipo *a;
public:
   Pointer(ExpresionTipo *a) { this->a=a; }
   ~Pointer() { delete a; }
   // * Para el generador de código
   unsigned getBytes() const { return 4; }
   // * Por depuración
   string expresionTipo() const;
   // * Comprobaciones semánticas
   ExpresionTipo *flecha() { return a; }
   // * Equivalencia de expresiones de tipo
   virtual bool equivalente(const ExpresionTipo *) const;
   // * Métodos específicos
   const ExpresionTipo *getA() const { return a; }
};

// * Clase compuesta del patrón Composite: tipo compuesto Array
class Array: public ExpresionTipo {
   Array(Array&) {}
   Array&operator=(Array&) { return *this; }
   int desde,hasta;
   ExpresionTipo *de;
public:
   Array(int desde,int hasta,ExpresionTipo *de) {
      this->desde=desde; this->hasta=hasta; this->de=de; }
   ~Array() { delete de; }
   // * Para el generador de código
   unsigned getBytes() const { return (hasta-desde+1)*de->getBytes(); }
   // * Por depuración
   string expresionTipo() const;
   // * Comprobaciones semánticas
   ExpresionTipo *corchete(const ExpresionTipo *e) {
      if (!dynamic_cast<const Integer*>(e))
         return new Error("El índice ha de ser de tipo entero.");
      return de;
   }
   // * Equivalencia de expresiones de tipo
   virtual bool equivalente(const ExpresionTipo *) const;
   // * Métodos específicos
   const ExpresionTipo *getDe() const { return de; }
};

// * Clase compuesta del patrón Composite: tipo compuesto función ->
class Function: public ExpresionTipo {
   Function(Function&) {}
   Function&operator=(Function&) { return *this; }
   ExpresionTipo *devolucion;
   vector<ExpresionTipo*> parametros;
public:
```

```
    Function(ExpresionTipo *d,const vector<ExpresionTipo*> &v):
      devolucion(d),parametros(v) {}
     ~Function();
    // * Para el generador de código
    unsigned getBytes() const { return 4; }
    // * Por depuración
    string expresionTipo() const;
    // * Comprobaciones semánticas
    ExpresionTipo *parentesis(const vector<ExpresionTipo*>&) ;
    // * Equivalencia de expresiones de tipo
    virtual bool equivalente(const ExpresionTipo *) const;
    // * Métodos específicos
    const ExpresionTipo *getDevolucion() const { return devolucion; }
    const vector<ExpresionTipo*> &getParametros() const { return parametros; }

};

// * Clase compuesta del patrón Composite: tipo compuesto Record
class Record: public ExpresionTipo {
    Record(Record&) {}
    Record &operator=(Record&) { return *this; }
    map<string,ExpresionTipo*> campos;
public:
    Record(const map<string,ExpresionTipo*> &c): campos(c) {}
    ~Record();
    // * Para el generador de código
    unsigned getBytes() const;
    // * Por depuración
    string expresionTipo() const;
    // * Comprobaciones semánticas
    ExpresionTipo *punto(const string&);
    // * Equivalencia de expresiones de tipo
    virtual bool equivalente(const ExpresionTipo *) const;
    // * Métodos específicos
    const map<string,ExpresionTipo*> &getCampos() const { return campos; }
};

#endif
```

tipos.cpp

```
#include "tipos.h"
#include <map>
#include <string>
#include <vector>
#include <sstream>
#include <cstring>
using namespace std;

/******** Expresion Tipo ******************************************/

ExpresionTipo *ExpresionTipo::flecha() {
      return new Error("Operación ^ no permitida.");
}

ExpresionTipo *ExpresionTipo::corchete(const ExpresionTipo*) {
      return new Error("Operación [] no permitida.");
}

ExpresionTipo *ExpresionTipo::punto(const string&) {
      return new Error("Operación . no permitida.");
}

ExpresionTipo *ExpresionTipo::parentesis(const vector<ExpresionTipo*>&) {
      return new Error("Operación [] no permitida.");
}

// * Funcionamiento por omisión para los tipos simples
bool ExpresionTipo::equivalente(const ExpresionTipo *et) const {
   return typeid(*this)==typeid(*et); // * RTTI
}


/******** Pointer *************************************************/

string Pointer::expresionTipo() const {
   ostringstream o;
   o<<"pointer("<<a->expresionTipo()<<")";
   return o.str();
}

bool Pointer::equivalente(const ExpresionTipo *et) const {
   const Pointer *puntero=dynamic_cast<const Pointer*>(et);
   if (!puntero) return false;
   return a->equivalente(puntero->a);
}
```

```
/******** Array *****************************************/

string Array::expresionTipo() const {
   ostringstream o;
   o<<"array("<<desde<<".."<<hasta<<","<<de->expresionTipo()<<")";
   return o.str();
}

bool Array::equivalente(const ExpresionTipo *et) const {
   const Array *array=dynamic_cast<const Array*>(et);
   if (!array) return false;
   return desde==array->desde && hasta==array->hasta &&
      array->equivalente(array->de);
}
/******** Function ****************************************/

Function::~Function(){
   delete devolucion;
   for (unsigned i=0;i<parametros.size();i++)
      delete parametros[i];
}

ExpresionTipo *Function::parentesis(const vector<ExpresionTipo*> &v) {
   if (v.size()!=parametros.size()) {
      ostringstream o;
      o<<"La función posee "<<parametros.size()<<" parámetros y se le están "<<
         "pasando "<<v.size()<<".";
      return new Error(o.str());
   }
   // * Comparamos la igualdad de los tipos
   int equivalente=1;
   unsigned i=0;
   for (;i<v.size()&&equivalente;i++)
      equivalente=parametros[i]->equivalente(v[i]);
   if (!equivalente) {
      ostringstream o;
      o<<"La funcion está declarada con el parámetro número "<<i<<
         " de tipo "<<parametros[i-1]->expresionTipo()<<
         " y se le está pasando una expresión de tipo "<<
         v[i-1]->expresionTipo()<<".";
      return new Error(o.str());
   }
   return devolucion;
}

string Function::expresionTipo() const {
   ostringstream o;
   if (parametros.size())
        o<<"(";
   unsigned i=0;
   for (;i<parametros.size()-1;i++)
      o<<parametros[i]->expresionTipo()<<',';
   if (parametros.size())
      o<<parametros[i]->expresionTipo()<<')';
   o<<"->"<<devolucion->expresionTipo();
   return o.str();
}

bool Function::equivalente(const ExpresionTipo *et) const {
   const Function *fun=dynamic_cast<const Function*>(et);
   if (!fun) return false;
   if (parametros.size()!=fun->parametros.size())
      return false;
   for (unsigned i=0;i<parametros.size();i++)
      if (!parametros[i]->equivalente(fun->parametros[i]))
         return false;
   return devolucion->equivalente(fun->devolucion);
}

/******** Record ****************************************/

Record::~Record() {
   map<string,ExpresionTipo*>::const_iterator it;
   for (it=campos.begin();it!=campos.end();++it)
      delete it->second;
}

ExpresionTipo *Record::punto(const string &campo) {
   if (campos.find(campo)==campos.end()) {
      ostringstream o;
      o<<"El campo \""<<campo<<"\" no esta declarado en el registro.";
      return new Error(o.str());
   }
   return campos[campo];
}
```

```
unsigned Record::getBytes() const {
   unsigned suma=0;
   map<string,ExpresionTipo*>::const_iterator it;
   for (it=campos.begin();it!=campos.end();++it)
      suma+=it->second->getBytes();
   return suma;
}

string Record::expresionTipo() const {
   ostringstream o;
   o<<"record(";
   map<string,ExpresionTipo*>::const_iterator it;
   for (it=campos.begin();it!=campos.end();) {
      o<<" ("<<it->first<<" x "<<it->second->expresionTipo()<<") ";
      ++it;
      if (it!=campos.end()) o<<"x ";
   }
   o<<")";
   return o.str();
}

bool Record::equivalente(const ExpresionTipo *et) const {
   const Record *record=dynamic_cast<const Record*>(et);
   if (!record) return false;
   if (campos.size()!=record->campos.size())
      return false;
   map<string,ExpresionTipo*>::const_iterator it1,it2;
   for (it1=campos.begin(),it2=record->campos.begin();it1!=campos.end();++it1,++it2) {
      if (it1->first!=it2->first)
         return false;
      if (!it1->second->equivalente(it2->second))
         return false;
   }
   return true;
}
```

ts.h

```
#ifndef _ts_h
#define _ts_h

#include "tipos.h"
#include <map>
#include <string>
#include <sstream>

using namespace std;

class TablaSimbolos {
   map<string,ExpresionTipo*> tabla;
public:
   ~TablaSimbolos() {
      map<string,ExpresionTipo*>::iterator it;
      for (it=tabla.begin();it!=tabla.end();++it)
         delete it->second;
   }

   bool existe(const string &id) {  return tabla.find(id)!=tabla.end();}

   ExpresionTipo *buscar(const string &id) {
      if (!existe(id)) {
         ostringstream o;
         o<<"El identificador \""<<id<<"\" no se ha declarado.";
         return new Error(o.str());
      }
      return tabla[id];
   }

   void insertar(const string &id,ExpresionTipo *et) {
      tabla[id]=et;
   }

};

#endif
```

C.2 Sistema y Comprobador de Tipos

sintac.y

```
%{
```

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include "tipos.h"
#include "ts.h"
#include <map>
#include <vector>
#include <iostream>
using namespace std;
TablaSimbolos ts;
extern unsigned yylineno;
int yyparse();
int yylex();
%}
%union {
  int entero;
  char caracter;
  char cadena[255];
  ExpresionTipo *tipo;
  vector<ExpresionTipo*>* Vector;
  map<string,ExpresionTipo*>* Map;
  }
%right MENOS_UNARIO
%right '^'
%nonassoc '(' ')'
%token <entero> CTE_ENTERA
%token <caracter> CTE_CARACTER
%token <cadena>   ID
%token FUNCTION RECORD PROCEDURE VAR BEGINPR END DOS_PUNTOS ARRAY OF INTEGER CHAR
%type <tipo> expresion tipo
%type <Vector> tipos listatipos listaexpresiones comaexpresiones
%type <Map> listacampos
%%
programa: VAR declaraciones BEGINPR expresiones END '.'
   ;
declaraciones: declaraciones declaracion ';'
      | /* vacio */
   ;
declaracion: ID ':' tipo { if (ts.existe($1)) { cerr<<"Error en linea "<<yylineno;
                     cerr<<". "<<$1<<" ya declarado.\n"; }
                     ts.insertar($1,$3); }
   ;
tipo: INTEGER  { $$=new Integer; }
    | CHAR     { $$=new Char; }
    | '^' tipo  { $$=new Pointer($2); }
    | ARRAY '[' CTE_ENTERA DOS_PUNTOS CTE_ENTERA ']' OF tipo
           { $$=new Array($3,$5,$8); }
    | FUNCTION '(' listatipos ')' ':' tipo
           { $$=new Function($6,*$3); delete $3; }
    | PROCEDURE '(' listatipos ')'
           { $$=new Function(new Void,*$3); delete $3; }
    | RECORD listacampos END
           { $$=new Record(*$2); delete $2; }
   ;
listatipos: tipos { $$=$1; }
   | /* vacio */  { $$=new vector<ExpresionTipo*>; }
   ;
tipos: tipos ',' tipo   { $$=$1; $$->push_back($3); }
   | tipo           { $$=new vector<ExpresionTipo*>; $$->push_back($1); }
   ;
listacampos: listacampos ID ':' tipo ';'  { $$=$1; (*$$)[$2]=$4; }
   |  /* vacio */    { $$=new map<string,ExpresionTipo*>; }
   ;
expresiones: expresiones expresion ';'{ Error *e=dynamic_cast<Error*>($2);
                                if (e) cerr<<*e<<endl;
                                else {
                                  cout<<"Linea "<<yylineno<<", tipo "<<
                                  $2->expresionTipo()<<", "<<
                                  $2->getBytes()<<" bytes.\n"; }
                                }
      | /* vacio */
   ;
expresion: '(' expresion ')'       { $$=$2; }
         | CTE_ENTERA              { $$=new Integer; }
         | CTE_CARACTER            { $$=new Char; }
         | ID                   { $$=ts.buscar($1); }
         | expresion '^'           { $$=$1->flecha(); }
         | expresion '[' expresion ']'{ $$=$1->corchete($3); }
         | expresion '.' ID          { $$=$1->punto($3); }
         | ID '(' listaexpresiones ')'{ $$=ts.buscar($1)->parentesis(*$3); delete $3; }
         ;
listaexpresiones: comaexpresiones   { $$=$1; }
   | /* vacio */             { $$=new vector<ExpresionTipo*>; }
   ;
comaexpresiones: comaexpresiones ',' expresion  { $$=$1; $$->push_back($3); }
   | expresion    { $$=new vector<ExpresionTipo*>; $$->push_back($1); }
```

```
   ;
%%
void yyerror(char *s) {
  extern unsigned yylineno;
  cerr<<"Error sintactico en la linea "<<yylineno;
  }

int main() {
  yyparse();
  }
```

lexico.l

```
%{
#include <string.h>
#include <stdlib.h>
#include "tipos.h"
#include <map>
#include <vector>
using namespace std;
#include "yytab.h"

unsigned yylineno=1;
void errorLexico(char);
void inicializarLexico(char *,char *);
%}
comentario \/\/.*\n
linea      \n
letra      [a-zA-Z]
numero     [0-9]
real       {numero}+(\.{numero}+)?
alfa       ({numero}|{letra})
var        [Vv][Aa][Rr]
begin    [Bb][Ee][Gg][Ii][Nn]
end        [Ee][Nn][Dd]
function [Ff][Uu][Nn][Cc][Tt][Ii][Oo][Nn]
procedure   [Pp][Rr][Oo][Cc][Ee][Dd][Uu][Rr][Ee]
record     [Rr][Ee][Cc][Oo][Rr][Dd]
array    [Aa][Rr][Rr][Aa][Yy]
of       [Oo][Ff]
integer     [Ii][Nn][Tt][Ee][Gg][Ee][Rr]
char     [Cc][Hh][Aa][Rr]
%%
[ \t]+         ; // * Se ignoran los espacions y tabuladores
{comentario}   { yylineno++; }
{linea}        { yylineno++; }
{numero}+      { yylval.entero=atoi(yytext); return CTE_ENTERA; }
\'{alfa}\'       { yylval.caracter=yytext[1]; return CTE_CARACTER; }
{var}       { return VAR; }
{begin}        { return BEGINPR;}
{end}       { return END;}
{function}     { return FUNCTION;}
{procedure}    { return PROCEDURE;}
{record}    { return RECORD;}
{array}        { return ARRAY;}
{of}        { return OF; }
{integer}      { return INTEGER; }
{char}         { return CHAR; }
{letra}{alfa}*  { strcpy(yylval.cadena,yytext); return ID; }
".."           { return DOS_PUNTOS; }
\.             |
\:             |
\,             |
\[             |
\]             |
\(             |
\)             |
\^             |
\;             |
;              { return yytext[0]; } // Token=Caracter ASCII
.              { errorLexico(yytext[0]); }  // Caracter no perteneciente al lenguaje
%%

void errorLexico(char c) {
  printf("Error lexico en linea %d. Caracter %c no perteneciente al lenguaje.\n",
      yylineno,c);
  exit(-1);
  }

void inicializarLexico(char *fOrigen,char *fDestino) {
  yyin=fopen(fOrigen,"r");
  if (yyin==NULL) {
    printf("Error abriendo el fichero %s.\n",fOrigen);
    exit(-1);
    }
  yyout=fopen(fDestino,"w");
```

```
  if (yyout==NULL) {
    printf("Error abriendo el fichero %s.\n",fDestino);
    exit(-1);
    }
  }
```

REFERENCIAS BIBLIOGRÁFICAS

[Aho90] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Compiladores: Principios, Técnicas y Herramientas. Addison-Wesley Iberoamericana. 1990.

[ANSIC++] Programming Language C++. Doc No: X3J16/96-0225. ANSI (American National Stardards Institute).1996.

[ANTLR] ANTLR, Another Tool for Language Recognition. http://www.antlr.org/

[Atkinson86] R.G. Atkinson. Hurricane: An Optimizing Compiler for Smalltalk. ACM SIGPLAN Notices, vol. 21, no. 11. 1986.

[Bischoff92] Kurt M. Bischoff. User Manual for Ox: an Attribute Grammar Compiling System based on Yacc, Lex and C. Technical Report 92-30. Department of Computer Science, Iowa State University. 1992.

[Bjorner82] D. Bjorner, C.B. Jones. Formal Specification and Software Development. Prentice Hall. 1982.

[Cardelli97] Luca Cardelli. Type Systems. The Computer Science and Engineering Handbook, CRC Press. 1997.

[Cueva03] J.M. Cueva, R. Izquierdo, A.A. Juan, M.C. Luengo, F. Ortín, J.E. Labra. Lenguajes, Gramáticas y Autómatas en Procesadores de Lenguaje. Servitec. 2003.

[Cueva98] Juan Manuel Cueva Lovelle. Conceptos básicos de Procesadores de Lenguaje. Servitec. 2003.

[Eckel00] Bruce Eckel. Thinking in C++. Volume 1: Introduction to Standard C++. 2nd Edition. Prentice Hall. 2000.

[Engelfriet84] J. Engelfriet. Attribute grammars: Attribute evaluation methods. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pp 103-137. Cambridge University Press, 1984.

[FNC-2] The FNC-2 attribute grammar system. http://www-rocq.inria.fr/oscar/www/fnc2/

[GOF02] Erich Gamma, Richard Elm, Ralph Johnson, John Vlissides. Patrones de diseño: elementos de software orientado a objetos reutilizable. Pearson Educación. 2002.

[Goldberg83] Adele Goldberg, David Robson. Smalltalk-80: the language and its implementation. Addison-Wesley. 1983.

[Gosling00] J. Gosling, B. Joy, G. Steele, G. Bracha. The Java Language Specification. Addison Wesley. 2000.

[Hoare73] C.A.R. Hoare, N. Wirth. An axiomatic definition of the programming language Pascal. Acta Informatica 2. 1973.

[Hopcroft02] J.E. Hopcroft, R. Motwani, J.D. Ulman. Introducción a la Teoría de autómatas, lenguajes y computación. Pearson Educación. 2002.

[Jansen93] Paul Jansen, Lex Augusteijn y Harm Munk. An introduction to Elegant. Philips Research Laboratories. 1993.

[JavaCC] JavaCC, Java Compiler Compiler. https://javacc.dev.java.net/

[Jazayeri75] M Jazayeri, W.F. Ogden y W.C. Rounds. The intrinsic exponential complexity of the circularity problem for attribute grammars. Communications of the ACM 18 (12). 1975.

[Johnson75] S. C. Johnson. YACC –yet another compiler compiler. Technical Report Computing Science TR32, AT&T Bell Laboratories, Murray Hill. 1975.

[Kernighan91] Brian W. Kernighan, Dennis M. Ritchie. El lenguaje de programación C. Segunda Edición. Pearson Educación. 1991.

[Kernighan91] Brian Kernighan, Dennis M. Ritchie. El lenguaje de programación C. 2ª Edición. Prentice Hall 1991.

[Knuth68] Donald E. Knuth. Semantics of context-free languages. Mathematical Systems Theory 2(2). 1968.

[Labra01] José Emilio Labra Gayo. Desarrollo Modular de Procesadores de Lenguajes a partir de Especificaciones Semánticas Reutilizables. Tesis Doctoral. Departamento de Informática de la Universidad de Oviedo. 2001.

[Labra03] J.E. Labra, J.M. Cueva, R. Izquierdo, A.A. Juan, M.C. Luengo, F. Ortín. Intérpretes y Diseño de Lenguajes de Programación. Servitec. 2003.

[Louden97] Kenneth C. Louden. Compiler Construction: Principles and Practice. Brooks Cole. 1997.

[LRC] System for generating efficient incremental attribute evaluators. http://www.cs.uu.nl/groups/ST/Software/LRC/

[Lucas69] Peter Lucas, Kurt Walk. On the Formal Description of PL/I. Annual Review of Automatic Programming. Oxford Pergamon Press. 1969.

[Mason92] Tony Mason, John Levine, Doug Brown. Lex & Yacc. 2nd Edition. O'Reilly & Associates. 1992.

[Meinke92] K. Meinke y J. V. Tucker. Universal algebra. En S. Abramsky, D. M. Gabbay, y T. S. E. Maibaum, editores, Handbook of Logic in Computer Science, tomo I. Oxford University Press, 1992.

[Milner78] Robin Milner. A theory of type polymorphism in programming. Journal of Computer and Systems Sciences 17. 1978.

[Milner84] Robin Milner. A proposal for standard ML. CDM Symposium on Lisp and Functional Programming Languages. 1984.

[Mosses91] P.D. Mosses. Action Semantics. Cambridge University Press. 1991.

[Muchnick97] Steven Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann. 1997.

[Nielson92] H. R. Nielson, F. Nielson. Semantics with Applications. Wiley. 1992.

[Pascal82] British Standards Institute. Specification for Computer Programming Language Pascal. Publication BS6192:1982. British Standards Institute. 1982.

[Pierce02] Benjamin C. Pierce. Types & Programming Languages. MIT Press. 2002.

[SableCC] SableCC: Java object-oriented framework to generate compilers and interpreters. http://www.sablecc.org/

[Saraiva99] Joao Saraiva. Implementing a Functional Implementation of Attribute Grammars. Tesis Doctoral. Universiteit Utrecht. 1999.

[Scott00] Michael Scott. Programming Language Pragmatics. Morgan Kaufmann. 2000.

[Stroustrup93] Bjarne Stroustrup. El lenguaje de programación C++. 3ª edición. Addison Wesley – Diaz de Santos. 1993.

[Waite84] William M. Waite y Gerhard Goos. Compiler Construction. Springer-Verlag. 1984.

[Watt00] David A. Watt, Deryck Brown.Programming Language Processors in Java: Compilers and Interpreters. Prentice Hall. 2000.

[Watt96] David A. Watt, Muffy Thomas. Programming Language Syntax and Semantics. Prentice Hall. 1996.

[Wilhelm95] Renhard Wilhelm, Dieter Maurer. Compiler Design. Addison Wesley. 1995.