

## Capítulo 4. Análisis semántico: Fase de Identificación

### 1 Objetivo

#### 1.1 Objetivo del Análisis semántico

El tipo de los hijos de cada nodo (*expresión*, *sentencia*, etc.) ya supone una restricción sobre qué nodos se pueden conectar con qué otros en el árbol (por ejemplo, no se puede poner un objeto *VarDefinition* como hijo de un nodo *Print* por no ser una expresión). Sin embargo, quedan aún situaciones en las que, aunque los tipos de los hijos sean válidos, no deben ser aceptadas en un programa. Un ejemplo de esto sería la asignación de un real a una variable de tipo entero. Ambos nodos son de tipo *expresión* (tal y como se requiere a los hijos del nodo *asignación*) pero, sin embargo, el subárbol que forman esos tres nodos no debe aceptarse como válido en el AST (en este lenguaje solo se permiten asignaciones del mismo tipo).

Por tanto, una manera [demasiado] informal de explicar la misión del semántico es la de *validar todos los subárboles* del AST (que el padre y los hijos cumplan unas determinadas reglas). Si todos ellos son válidos, el AST será válido.

#### 1.2 Fases del análisis semántico

El análisis semántico debe hacer distintas validaciones. Aunque se podría hacer todo el análisis semántico en una sola fase (de hecho, en varios textos se presenta así), es más fácil de diseñar e implementar si se hace por partes. La mayor parte (no todas) de las validaciones que debe hacer el semántico se corresponden con las reglas de ámbito y las reglas de tipo. Por tanto, el análisis semántico se hará en dos fases que se corresponden con la validación de cada uno de estos tipos reglas: las reglas de ámbito se comprueban en la Fase de Identificación y las reglas de tipo se comprobarán en el próximo capítulo en la fase de comprobación de tipos<sup>1</sup>.

#### 1.3 Objetivo de la Fase de Identificación

La Fase de Identificación es la primera de las dos subfases en las que se va a dividir el análisis semántico y es a lo único que se va a dedicar este capítulo. Esta fase tiene que realizar dos tareas: validar y enlazar.

##### 1.3.1 Tarea 1. Validar

La primera tarea de esta fase es comprobar que:

- Que no haya definiciones inválidas.
- Todo símbolo que se usa en el programa ha sido definido y es accesible desde el lugar en el que se utiliza.

---

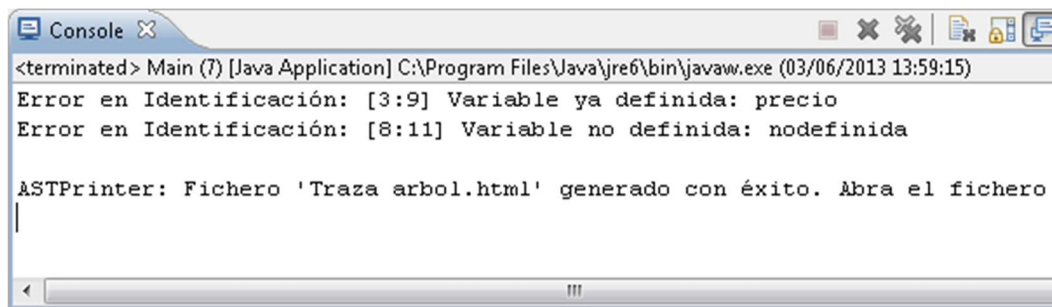
<sup>1</sup> Cabe preguntarse qué, si no todas las validaciones se corresponden con estas reglas, en qué fase se tratan las validaciones restantes que no son de ninguno de estos dos tipos. Aunque quizás fuera más coherente realizarlas en una tercera fase con “el resto” de las validaciones, en la práctica se suelen incluir en la segunda fase, la comprobación de tipos, debido a que suelen ser muy pocas y por tanto no suele merecer la pena hacerlas aparte.

Por ejemplo, ante una entrada como:

```
DATA
    float precio;
    int precio;    // Error: variable ya definida
    int ancho;

CODE
    print precio;
    print nodefinida;    // Error: variable no definida
```

Debería notificar los siguientes errores:



### 1.3.2 Tarea 2. Enlazar

Aunque, estrictamente, la tarea anterior sería la única misión de esta fase, también debe realizar, como tarea adicional, el enlazado de las variables con su definición (que los objetos *Variable* apunten a su *VarDefinition*). La razón de esta última tarea es que las fases posteriores del compilador van a necesitar de nuevo información sobre la definición de una variable (su tipo, su dirección, etc.). Si quedan ya enlazadas no habrá que repetir en fases posteriores la búsqueda de definiciones que ya se hace aquí.

## 2 Trabajo autónomo del alumno

Diseñar e implementar esta fase *antes de seguir leyendo* el resto del documento. Una vez hecho, comparar la solución del alumno con la planteada a continuación.

## 3 Solución

### 3.1 Diseño. Creación de la Especificación

#### 3.1.1 Extracción de Requisitos

Los requisitos de esta fase son las *Reglas de Ámbito* del lenguaje. Sin embargo, en *Descripción del lenguaje.pdf*, no se menciona ninguna regla de este tipo. Por tanto, es de suponer que las reglas de ámbito sean las habituales:

1. Cada variable es accesible desde toda la sección CODE.
2. No se puede definir una variable con el mismo nombre que otra previamente definida (independientemente del tipo de ambas).

Es razonable suponer que si hubiera reglas de ámbito excepcionales se hubieran mencionado en la especificación (por ejemplo, que se pudiera cambiar el tipo de una variable volviéndola a definir de nuevo).

### 3.1.2 Metalenguaje elegido

Los requisitos semánticos anteriores se expresarán mediante una *Gramáticas Atribuida* (GAt). Este metalenguaje es una forma sencilla de expresar:

- Qué información se añade al árbol y en qué nodos (*atributos*).
- Qué condiciones deben cumplir los nodos de un árbol para que este sea válido (*predicados*).

Concretando para este caso:

- La información que se añade al árbol es un enlace entre las variables y su definición (un atributo *definición*)
- Las condiciones que debe cumplir el árbol es que no se defina dos veces una variable o que no se use una que no se haya definido (predicados).

### 3.1.3 Especificación en el Metalenguaje

Una alternativa a comenzar la Gramática Atribuida desde cero es utilizar el esqueleto generado por *VGen*<sup>2</sup>. En la carpeta *metalenguajes* éste generó un fichero '*Attribute Grammar.html*' en el cual ya aparece la estructura de una gramática atribuida. En la primera tabla aparece una primera columna con la gramática abstracta del lenguaje. Aparecen además una segunda y tercera columna vacías para que se pongan los predicados y las reglas semánticas respectivamente. Finalmente, aparece una segunda tabla en el que se indicarán los atributos que se añadirán al árbol y en qué nodos.

Una vez abierto dicho esqueleto en *Word* (o editor equivalente), se traducen los requisitos a la notación del metalenguaje rellenando las dos columnas de la derecha y la tabla de atributos. El resultado de este proceso está en "*Cambios\metalenguajes\Identificación. Gramática Atribuida.pdf*"

---

<sup>2</sup> Si no se guardó la carpeta *metalenguajes* del capítulo 3, basta con volver a ejecutar *VGen* con el fichero *gramática.txt* que se creó en dicho capítulo.

Nodo	Predicados	Reglas Semánticas
<b>program</b> → <i>definitions</i> :varDefinition* <i>sentences</i> :sentence*		
<b>varDefinition</b> → <i>type</i> :type <i>name</i> :String	variables[name] == ∅	variables[name] = varDefinition
<b>intType</b> :type → λ		
<b>realType</b> :type → λ		
<b>print</b> :sentence → <i>expression</i> :expression		
<b>assignment</b> :sentence → <i>left</i> :expression <i>right</i> :expression		
<b>arithmeticExpression</b> :expression → <i>left</i> :expresión <i>operator</i> :String <i>right</i> :expression		
<b>variable</b> :expression → <i>name</i> :String	variables[name] ≠ ∅	variable.definition = variables[name]
<b>intConstant</b> :expression → <i>valor</i> :String		
<b>realConstant</b> :expression → <i>valor</i> :String		

Tabla de Atributos:

Categoría Sintáctica	Nombre	Tipo Java	H/S	Descripción
variable	definition	VarDefinition	Sintetizado	Nodo donde está definida la variable

Estructuras de datos auxiliares:

Nombre	Tipo Java	Descripción
variables	Map<String, VarDefinition>	Hash donde se guardan las definiciones a medida que se vayan encontrando

## 3.2 Implementación

### 3.2.1 Enlace con las definiciones

Tal y como se indica en la tabla de atributos de la gramática atribuida anterior, hay que añadir una propiedad *definición* a los objetos (nodos) de tipo *Variable*:

```
public class Variable extends AbstractExpression {

    private VarDefinition definition;

    public VarDefinition getDefinicion() {
        return definition;
    }

    public void setDefinicion(VarDefinition definition) {
        this.definition = definition;
    }

    // El resto igual que estaba...
}
```

### 3.2.2 Fase de Identificación

Para implementar esta fase, hay que aplicar a la gramática anterior la técnica de implementación de Gramáticas Atribuidas vista en clase de teoría. Esta generará un visitor en el que por cada nodo del AST habrá un método *visit* con la siguiente estructura:

#### Recorrido de Gramáticas S-Atribuidas

##### Características de una Gramática S-Atribuida

- Puede implementarse con un solo recorrido del árbol
- La implementación de *todo* método *visit* debe seguir el patrón:

```
visit(Node nodoPadre) {  
  Para cada hijoi de nodoPadre { // Da igual el orden de visita de los hijos  
    Visitar(hijoi) // Al volver, el hijo tendrá ya asignados sus atributos  
  }  
  B(p): Comprobar predicados  
  R(p): Asignar valor a los atributos de nodoPadre  
}
```

Para implementar este *visitor*, en el proyecto de eclipse ya existe un fichero "*semantico/Identification.java*" que indica donde colocar esta nueva clase. En dicho fichero se tiene un esqueleto al que solo falta añadirle los métodos *visit* para los nodos de nuestra gramática.

El añadir los métodos *visit* se puede hacer a mano, aunque es más rápido utilizar el fichero '*visitor/\_PlantillaParaVisitors.txt*' que fue generado por *VGen*. Este fichero ya contiene todos los métodos *visit* para los nodos de nuestra gramática junto con el código que recorre los hijos. Basta con copiar y pegar los métodos *visit* de dicha plantilla en *Identification.java* de tal manera que el trabajo se podrá centrar en decidir qué se quiere hacer y en qué nodos.

En este caso, tal y como muestra la gramática atribuida, solo hay que actuar sobre dos nodos: *Variable* (para buscar la definición y enlazarla) y *VarDefinition* (para comprobar que no esté repetida). Por tanto, todos los demás métodos *visit* se pueden borrar<sup>3</sup>.

La clase *Identification.java*, aplicando la técnica sobre la Gramática Atribuida anterior, quedaría así:

<sup>3</sup> Dado que al final solo se han utilizado dos métodos *visit* de *\_PlantillaParaVisitor.txt*, podría plantearse el por qué copiar todos en vez de escribir esos dos directamente. En general, se recomienda copiar todo y determinar método a método si hace falta actuar sobre dicho nodo o no para no olvidar ninguno. Además, es más fácil borrar lo que sobra que comenzar con un *visitor* vacío.

```
public class Identification extends DefaultVisitor {

    public Identification(ErrorManager errorManager) {
        this.errorManager = errorManager;
    }

    // class VarDefinition { Type type; String name; }
    public Object visit(VarDefinition node, Object param) {
        node.getType().accept(this, param); // No es necesario realmente

        VarDefinition definicion = variables.get(node.getName());
        predicado(definicion == null, "Variable ya definida: " + node.getName(), node);
        variables.put(node.getName(), node);
        return null;
    }

    // class Variable { String name; }
    public Object visit(Variable node, Object param) {
        VarDefinition definicion = variables.get(node.getName());
        predicado(definicion != null, "Variable no definida: " + node.getName(), node);
        node.setDefinicion(definicion); // Enlazar referencia con definición
        return null;
    }

    // # -----
    // Métodos auxiliares recomendados (opcionales) -----

    private void error(String msg, Position position) {
        errorManager.notify("Identification", msg, position);
    }

    private void predicado(boolean condition, String errorMessage, AST node) {
        if (!condition)
            error(errorMessage, node.getStart());
    }

    private ErrorManager errorManager;
    private Map<String, VarDefinition> variables = new HashMap<String, VarDefinition>();
}
;
```

Siguiendo la técnica para pasar de gramática s-atribuida a código, lo primero sería recorrer los hijos y luego implementar los predicados y las reglas. Sin embargo, en este caso, el recorrido del hijo *tipo* de *VarDefinition* (destacado en rojo) no es necesario, ya que, en esta gramática atribuida, no hay que realizar ninguna acción sobre los tipos. Sin embargo, se ha añadido para que se vean las tres partes de la estructura (lo normal hubiera sido no poner dicha línea).

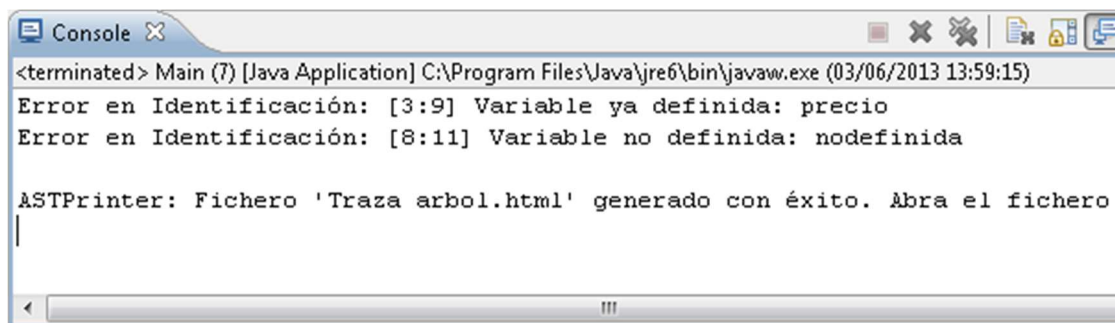
## 4 Ejecución

Se modifica el fichero *source.txt* para comprobar que el compilador detecta adecuadamente los errores de esta fase:

```
DATA
    float precio;
    int precio;    // Error: variable ya definida
    int ancho;

CODE
    print precio;
    print nodefinida;    // Error: variable no definida
```

Al ejecutar la clase *Main* se obtiene la siguiente salida tal y como se esperaba:



```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (03/06/2013 13:59:15)
Error en Identificación: [3:9] Variable ya definida: precio
Error en Identificación: [8:11] Variable no definida: nodefinida

ASTPrinter: Fichero 'Traza arbol.html' generado con éxito. Abra el fichero
```

## 5 Resumen de Cambios

Fichero	Acción	Descripción
<b>Identificación. Gramática Atribuida.pdf</b>	Creado	Metalenguaje en el que se han descrito los requisitos de esta fase
<b>Variable.java</b>	Modificado	Se añade una referencia a su definición ( <i>VarDefinition</i> )
<b>Identification.java</b>	Modificado	Comprueba que toda variable ha sido definida y solo lo ha hecho una vez. Además, enlaza las variables con sus definiciones
<b>source.txt</b>	Modificado	Contienen errores que deben detectarse en esta fase