

## Capítulo 3. Análisis Sintáctico: Creación del AST

### 1 Objetivo

En el capítulo anterior, el analizador sintáctico se limitaba a indicar si el programa de entrada estaba formado por estructuras válidas (definiciones de variables, asignaciones, etc.), pero dichas estructuras no se conservaban en memoria. El resto de las fases del compilador necesitan saber cuáles son estas estructuras para poder validarlas y generar código.

Lo que va a hacer este módulo del compilador es representar *el mismo programa* que se encuentra en el fichero de entrada, pero de una forma más fácil de acceder: mediante un *árbol sintáctico abstracto* (AST). Por tanto, el objetivo de esta fase es, dado un programa sintácticamente válido, crear en memoria el árbol correspondiente que lo represente y que permita al resto de las fases acceder de manera fácil y directa a todas sus estructuras.

### 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

Dado que en esta fase se va a crear un árbol, los requisitos que la corresponden son aquellos que dicen qué elementos (nodos) debe tener un árbol y cómo pueden conectarse. Pero a diferencia de otras fases, éstos no se extraerán de la especificación del lenguaje que se halla en *Introducción.pdf*. Dado que la misión de esta fase es crear un árbol que va a ser procesado por las demás fases, son éstas las que determinarán los requisitos del AST.

Los requisitos del AST de cualquier lenguaje son:

- **Completo.** Habrá que incluir toda información que necesiten el resto de las fases para realizar su tarea; debe preservarse la semántica del programa.
- **Mínimo.** Deberá incluir *sólo* la información anterior, eliminando toda información redundante (al contrario que el árbol sintáctico o concreto).

El diseño de los nodos de un AST no deja de ser un ejercicio de Diseño Orientado a Objetos clásico en el que habrá que identificar las clases (nodos) y las relaciones (hijos) que permitan un diseño más simple y eficaz del resto de las fases del compilador. Por tanto, lo normal es que a medida que se vayan implementado las fases posteriores del compilador, el diseño del

AST se vaya afinando para facilitar la implementación de dichas fases. No se debe pretender en este momento realizar un diseño de nodos que vaya a ser definitivo<sup>1</sup>.

### 3.1.2 Metalenguaje Elegido

Dado que el objetivo de esta fase es determinar cómo serán los AST que generará el compilador, se necesita un metalenguaje de descripción de árboles. El metalenguaje que se usará será las *Gramática Abstractas* (GAb).

Una *Gramática Abstracta*, descrita de manera informal, enumera todos los nodos que pueden usarse en la construcción de un árbol. Para cada uno de ellos define además a qué categoría sintáctica pertenece (expresión, sentencia, etc.) y cuántos hijos tiene y de qué tipo es cada uno.

### 3.1.3 Especificación

Los nodos mínimos para representar cualquier programa en el lenguaje del tutorial son los que se describen mediante la siguiente *gramática abstracta*:

```
programa → definiciones:defVariable*  sentencias:sentencia*

defVariable → tipo  nombre:string

intType:tipo →
realType:tipo →

print:sentencia → expresion
asigna:sentencia → left:expresion  right:expresion

exprAritmetica:expresion → left:expresion  operador:string  right:expresion
variable:expresion → nombre:string
literalInt:expresion → valor:string
literalReal:expresion → valor:string
```

En realidad, en este lenguaje tan sencillo, hubiera sido más correcto definir la asignación de la siguiente manera:

```
asigna:sentencia → variable  expresion
```

La razón de que se haya puesto como primer símbolo una *expresión* en lugar de una *variable*, es la misma por la que en el capítulo anterior, en la gramática, se puso un no-terminal *expr* a la izquierda de la asignación en lugar del token *IDENT*. Aunque en este lenguaje el primer hijo no sería una expresión, al hacerlo así se muestra cómo será la asignación en el lenguaje de la práctica del alumno (en el que, además de variables, a la izquierda de una asignación pueden aparecer accesos a arrays, estructuras, etc.) y permitirá en posteriores capítulos explicar aspectos adicionales de la asignación que, en caso contrario, no se presentarían. Por

---

<sup>1</sup> Al igual que ocurre con el diseño orientado a objetos, se tardará menos en hacer un buen diseño básico y modificarlo a medida que se necesiten más cosas que intentar prever en este momento todo lo que se va a necesitar en el AST.

ejemplo, en el capítulo 5 (Comprobación de Tipos), permitirá mostrar un ejemplo más completo de inferencia de tipos.

En resumen, aunque para este lenguaje el diseño elegido para la asignación no sería el adecuado, desde el punto de vista didáctico de este tutorial, la versión elegida permite mostrar más cosas (cómo se modelan las asignaciones en un lenguaje más completo y cómo se tratan en el semántico).

## 3.2 Implementación

### 3.2.1 Implementación manual de los nodos

El método para obtener la implementación de los nodos de un AST en Java a partir de la gramática abstracta es el siguiente:

- Cada categoría sintáctica se implementa como un interfaz.
- Cada nodo se implementa como una clase en Java. Dicha clase implementará los interfaces de las categorías sintácticas a las que pertenezca el nodo.
- Cada hijo es un atributo de la clase del padre. El tipo de dicho atributo está especificado en la gramática abstracta. Si el hijo es multievaluado (tiene un asterisco detrás del tipo) entonces el atributo será una *List* de Java.

Además, dado que el árbol se va a recorrer utilizando el patrón *Visitor*, habría que implementar también los requisitos que supone este patrón:

- Un interfaz *AST* que sea implementado por todos los nodos.
- Un interfaz *Visitor* con un método *visit* por cada nodo del árbol.
- Un método *accept* en cada nodo que invoque al método *visit* del *Visitor* que le corresponda.

Y, aunque no es obligatorio, también es conveniente que cada nodo del árbol guarde información sobre su posición original en el fichero de entrada (línea y columna) para poder ofrecer más información en los mensajes de error y en la generación de código.

### 3.2.2 Implementación mediante herramienta

En vez de hacer a mano todo el código anterior, otra opción más cómoda es utilizar la herramienta *VGen* incluida en este tutorial. El uso de esta herramienta es opcional, pero genera el mismo código que se obtendría de forma manual eliminando esta parte rutinaria de la construcción del compilador.

Para ello se escribe la gramática en el fichero *abstract.txt* siguiendo el formato de *VGen*:

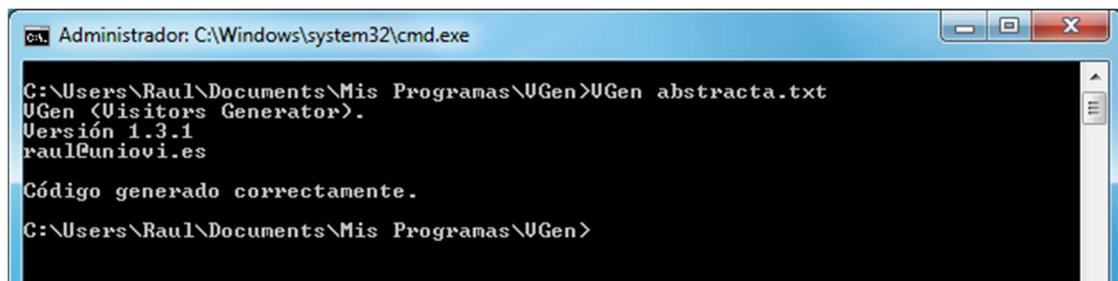
```
CATEGORIES
expresion, sentencia, tipo

NODES
programa -> definiciones: defVariable* sentencias: sentencia*;

defVariable -> tipo nombre: string;
```

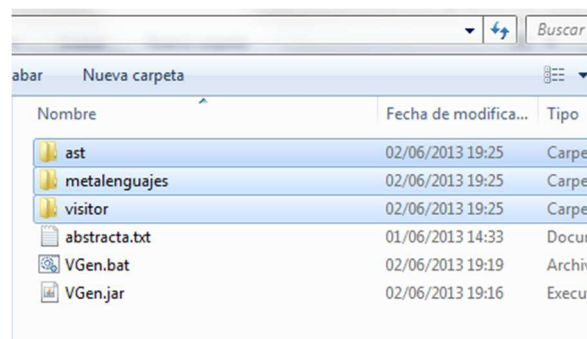
```
intType:tipo -> ;  
realType:tipo -> ;  
  
print:sentencia -> expresion;  
asigna:sentencia -> left:expresion right:expresion;  
  
exprAritmetica:expresion -> left:expresion operador:string right:expresion;  
variable:expresion -> nombre:string;  
literalInt:expresion -> valor:string;  
literalReal:expresion -> valor:string;
```

A continuación, se ejecuta *VGen* desde línea de comando. Aunque se puede invocar desde una ventana, se recomienda usar la línea de comandos para poder comprobar que no haya errores en el fichero:



```
C:\Users\Raul\Documents\Mis Programas\UGen>UGen abstracta.txt  
UGen (Visitors Generator).  
Versión 1.3.1  
raul@uniovi.es  
Código generado correctamente.  
C:\Users\Raul\Documents\Mis Programas\UGen>
```

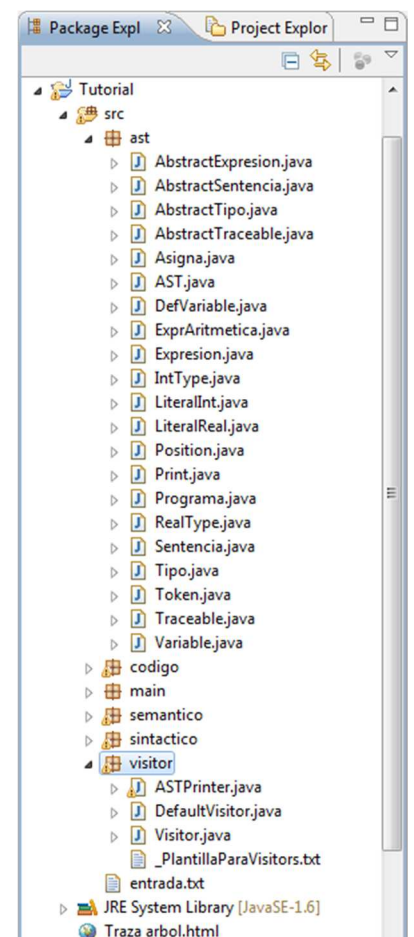
Una vez hecho lo anterior, habrán aparecido tres carpetas: *ast*, *metalenguajes* y *visitor*.



El contenido de la carpeta *metalenguajes* se utilizará en fases posteriores (semántico y generación de código), por lo que por ahora se puede apartar o bien borrar y volver a generar cuando llegue el momento.

Las carpetas *ast* y *visitor* deben copiarse a *eclipse*. El proyecto ya tiene paquetes con dichos nombres para indicar dónde hay que copiar los ficheros generados. A la hora de arrastrar los ficheros, *eclipse* mostrará un aviso indicando que ciertos ficheros van a ser sobrescritos. Esto debe ser así, ya que eran ficheros vacíos temporales hasta que éstos definitivos los sustituyeran.

El proyecto debería quedar tal y como aparece en la imagen lateral.



En la carpeta *ast* están la implementación de todos los nodos descritos en la gramática abstracta. Como muestra, se incluye el código del nodo *Print*:

```
package ast;

import visitor.*;

// print:sentencia -> expresion:expresion

public class Print extends AbstractSentencia {

    public Print(Expression expression) {
        this.expression = expression;

        searchForPositions(expression); // Obtener linea/columna a partir de los hijos
    }

    public Print(Object expression) {
        this.expression = (Expression) expression;

        searchForPositions(expression); // Obtener linea/columna a partir de los hijos
    }

    public Expression getExpression() {
        return expression;
    }

    public void setExpression(Expression expression) {
        this.expression = expression;
    }

    @Override
    public Object accept(Visitor v, Object param) {
        return v.visit(this, param);
    }

    private Expression expression;
}
```

En la implementación anterior puede observarse:

- El primer comentario es la regla de la gramática por la cual se ha generado esta clase.
- La clase *Print* deriva de *AbstractSentencia* ya que pertenece a dicha categoría sintáctica.
- Las llamadas a *searchForPosition* son las que se encargan de manera transparente de calcular la posición de este nodo en el fichero (línea y columna). No es necesario hacer nada adicional para realizar esta tarea.
- El constructor con parámetro *Object* permitirá eliminar los *cast* en las acciones de *BYaccJ* (se verá en breve).
- Se han definido los hijos como atributos y se han añadido los métodos de acceso (*get* y *set*) correspondientes.
- Se ha generado el método *accept*, necesario para el patrón *Visitor*.

Se recomienda detenerse en este momento para revisar el código generado en las carpetas *ast* y *visitor* y familiarizarse con él. Si se conoce el patrón *Visitor*, el código será el esperado.

### 3.2.3 Construcción del AST

Llegados a este punto, se tendría ya hecho:

- Un fichero *sintac.y*, creado en el capítulo anterior, que indica las estructuras que se esperan a la **entrada** del sintáctico.
- Varios ficheros Java, en la carpeta *ast*, con los nodos con los que deberá crear el AST que formará la **salida** del sintáctico.

Lo único que faltaría ahora es indicar qué nodo del AST se quiere crear cuando se encuentre cada una de las estructuras del programa. Para ello, habrá que añadir a la gramática del fichero *sintac.y* las acciones que irán formando el árbol a medida que se van produciendo reducciones.

```
programa: 'DATA' variables 'CODE' sentencias    { raiz = new Programa($2, $4); }

//-----
variables:          { $$ = new ArrayList(); }
  | variables variable { $$ = $1; ((List)$1).add($2); }

variable: tipo 'IDENT' ';' { $$ = new DefVariable($1, $2); }

tipo: 'INT'          { $$ = new IntType(); }
  | 'REAL'           { $$ = new RealType(); }

//-----
sentencias:          { $$ = new ArrayList(); }
  | sentencias sentencia { $$ = $1; ((List)$1).add($2); }

sentencia: 'PRINT' expr ';' { $$ = new Print($2); }
  | expr '=' expr ';'      { $$ = new Asigna($1, $3); }

//-----
expr: expr '+' expr    { $$ = new ExprAritmetica($1, "+", $3); }
  | expr '-' expr      { $$ = new ExprAritmetica($1, "-", $3); }
  | expr '*' expr      { $$ = new ExprAritmetica($1, "*", $3); }
  | expr '/' expr      { $$ = new ExprAritmetica($1, "/", $3); }
  | '(' expr ')'        { $$ = $2; }
  | 'IDENT'             { $$ = new Variable($1); }
  | 'LITERALINT'        { $$ = new LiteralInt($1); }
  | 'LITERALREAL'       { $$ = new LiteralReal($1); }
```

Finalmente hay que ejecutar de nuevo *genera.bat* para que *BYaccJ* genere la versión actualizada del analizador sintáctico.

**Nota.** Hay veces en que *eclipse* no detecta el cambio en un fichero generado por una herramienta y por tanto seguirá compilando la versión antigua del mismo. Si esto ocurre, debe seleccionarse el proyecto en la ventana *Package Explorer* y pulsar F5 para actualizar.

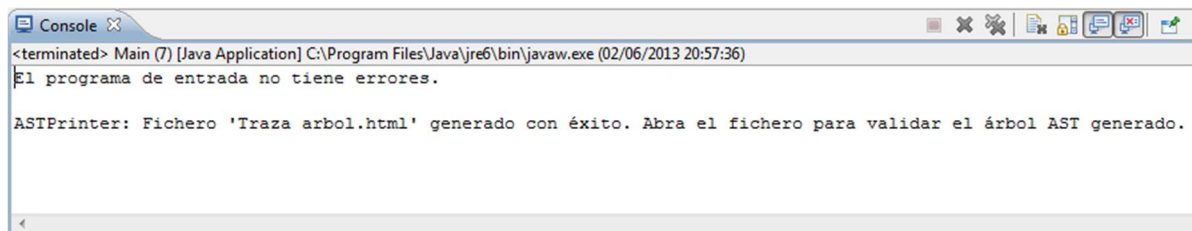
## 4 Ejecución

El contenido actual del fichero *entrada.txt* es:

```
DATA
    float precio;
    int ancho;
CODE
    ancho = 25 * (2 + 1);
    print ancho;

    precio = 5.0;
    print precio / 2.0;
```

Al ejecutar la clase *main.Main* del compilador, volverá a aparecer, como en el capítulo anterior, el mensaje de que el programa no tiene errores.



```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (02/06/2013 20:57:36)
El programa de entrada no tiene errores.

ASTPrinter: Fichero 'Traza arbol.html' generado con éxito. Abra el fichero para validar el árbol AST generado.
```

Sin embargo, aunque el mensaje que se recibe al compilar es el mismo, la diferencia es que ahora se ha creado un árbol en memoria. El fichero '*Traza arbol.html*' muestra la estructura del árbol que se ha creado:

```
[ASTPrinter] ----- line:col line:col
Programa →                               2:11  9:22  float precio; ... print
precio / 2.0;
. definiciones List<DefVariable> =
. . DefVariable →                          2:11  2:16  float precio;
. .   RealType →                          null null
. .   "precio"
. . DefVariable →                          3:9   3:13  int ancho;
. .   IntType →                          null null
. .   "ancho"
. sentencias List<Sentencia> =
. . Asigna →                               5:5   5:23  ancho = 25 * (2 + 1);
. .   "ancho" Variable.nombre             5:5   5:9   ancho = 25 * (2 + 1);
. .   ExprAritmetica →                     5:13  5:23  ancho = 25 * (2 + 1);
. .     "25" LiteralInt.valor              5:13  5:14  ancho = 25 * (2 + 1);
. .     "*"
. .     ExprAritmetica →                   5:19  5:23  ancho = 25 * (2 + 1);
. .       "2" LiteralInt.valor             5:19  5:19  ancho = 25 * (2 + 1);
. .       "+"
. .       "1" LiteralInt.valor             5:23  5:23  ancho = 25 * (2 + 1);
. . Print →                               6:11  6:15  print ancho;
. .   "ancho" Variable.nombre             6:11  6:15  print ancho;
. . Asigna →                               8:5   8:16  precio = 5.0;
. .   "precio" Variable.nombre            8:5   8:10  precio = 5.0;
. .   "5.0" LiteralReal.valor              8:14  8:16  precio = 5.0;

. . Print →                               9:11  9:22  print precio / 2.0;
. .   ExprAritmetica →                     9:11  9:22  print precio / 2.0;
. .     "precio" Variable.nombre           9:11  9:16  print precio / 2.0;
. .     "/"
. .     "2.0" LiteralReal.valor            9:20  9:22  print precio / 2.0;

[ASTPrinter] -----
```

En la columna central puede verse la posición inicial y final (línea y columna) de cada nodo en el fichero de entrada. El código generado por *VGen* en los nodos de AST realiza automáticamente esta tarea. A la derecha de las posiciones se muestra subrayado el texto contenido en dichas posiciones.

Este fichero *html* ha sido creado por la *ASTPrinter* (clase creada por *VGen* en la carpeta *visitor*) mediante la invocación realizada desde el *main*.

```
public static void main(String[] args) throws Exception {
    GestorErrores gestor = new GestorErrores();

    AST raiz = compile(programa, gestor);

    if (!gestor.hayErrores())
        System.out.println("El programa se ha compilado correctamente.");

    ASTPrinter.toHtml(programa, raiz, "Traza arbol"); // Generada por VGen (opcional)
}
```

Y con esto quedaría finalizado el Analizador Sintáctico de este tutorial.

## 5 Resumen de Cambios

Fichero	Acción	Descripción
<b>abstracta.txt</b>	Creado	Metalenguaje con la descripción de los nodos del árbol abstracto (AST)
<b>ast/*.java</b>	Generado	Clases cuyas instancias serán los nodos del AST
<b>visitor/*.java</b>	Generado	Interfaz <i>Visitor</i> y código auxiliar para facilitar su implementación (se verá posteriormente).
<b>sintac.y</b>	Modificado	Se añaden las acciones que crean el árbol
<b>Parser.java</b>	Generado	Implementación del Analizador Sintáctico. Creado con <i>BYaccJ</i> a partir de <i>sintac.y</i>
<b>Traza arbol.html</b>	Generado	Traza del AST generado por la clase <i>ASTPrinter</i> al ejecutar el compilador

## Apéndice I. Posiciones de los nodos en el fichero

Si se observa el fichero '*Traza arbol.html*' puede observarse que algunos nodos no tienen información de línea y columna y en su lugar aparece *null*:

```
Programa →                               2:11  9:22  float precio; ... print
· definiciones List<DefVariable> =
· · DefVariable →                         2:11  2:16  float precio;
· · · RealType →                          null  null
· · · "precio"
· · DefVariable →                         3:9   3:13  int ancho;
· · · IntType →                          null  null
· · · "ancho"
```



La forma en la que se obtiene las posiciones de cada nodo es la siguiente:

1. En el fichero *sintac.y*, cada vez que se obtiene un terminal del léxico, se guarda con él su posición en el fichero. De esta manera todos los símbolos terminales tienen esta información:

```
int yylex() {  
    try {  
        int token = lex.yylex();  
        yylval = new Token(token, lex.lexeme(), lex.line(), lex.column());  
        return token;  
    } catch (IOException e) {  
        return -1;  
    }  
}
```

2. Cada vez que se crea un nodo, el constructor creado por *VGen* asigna a dicho nodo la posición inicial de su primer hijo y la posición final de su último hijo (si algún hijo no tiene la posición buscada se pasa al siguiente).

```
public ExprAritmetica(Object left, Object operador, Object right) {  
    this.left = (Expresion) left;  
    this.operador = (operador instanceof Token) ? ((Token) operador).getLexeme() :  
    (String) operador;  
    this.right = (Expresion) right;  
  
    searchForPositions(left, operador, right); // Obtener linea/columna a partir  
    de los hijos  
}
```

De esta manera se obtiene de manera transparente toda la gestión de posiciones sin necesidad de saber de su existencia.

Por tanto, las dos situaciones en las que un nodo quedará sin información de posición son:

- Que el nodo no tenga hijos (y por tanto no pueda extraer de ellos posiciones).
- Que ninguno de los hijos tenga información de posición.

Estas dos situaciones (especialmente la segunda) son muy poco frecuentes.

En el caso de los *tipos*, el problema es el primero de los dos anteriores: no tienen ningún hijo (no se pasa ningún *\$n* en su constructor) y por ello es por lo que tienen sus posiciones a *null*:

```
tipo: 'INT'    { $$ = new IntType(); }  
      | 'REAL' { $$ = new RealType(); }
```

Esto no supone un problema en la práctica. No es necesario que todos los nodos tengan información de posición; basta con que lo tengan aquellos para los que se va a necesitar para dar mensajes de error o generar código. Y, en este caso, estos nodos no se van utilizar para ninguna de las dos funciones. Por tanto, sería correcto dejarlo como está.

De todas formas, si se quisiera información más precisa, VGen ha generado una serie de métodos auxiliares para introducir en los nodos esta información. Uno de ellos es *setPositions*:

```
tipo: 'INT'    { $$ = new IntType().setPositions($1); }  
      | 'REAL' { $$ = new RealType().setPositions($1); }
```

El método *setPositions* establece las posiciones de un nodo copiándolas del símbolo que se especifique como argumento (en este caso se sacan del lexema en \$1).

Si después de volver a pasar *sintac.y* por *BYaccJ* se ejecuta el código resultante, se obtiene este otro árbol:

Programa →	2:5	9:22	<u>float precio;</u> ... <u>print</u>
· definiciones List<DefVariable> =			
· · DefVariable →	2:5	2:16	<u>float precio;</u>
· · · <b>RealType</b> →	<b>2:5</b>	<b>2:9</b>	<u>float</u> precio;
· · · "precio"			
· · DefVariable →	3:5	3:13	<u>int ancho;</u>
· · · <b>IntType</b> →	<b>3:5</b>	<b>3:7</b>	<u>int</u> ancho;
· · · "ancho"			

Nótese cómo no solo los tipos tienen ahora posición, sino que las posiciones de las dos *DefVariable* también han mejorado respecto al árbol anterior, ya que ahora tiene la información de su primer hijo (el tipo).

Esto es un ejemplo de los métodos que incorpora VGen para controlar la información de las posiciones de los nodos. Si se desea más información sobre dichos métodos ésta puede encontrarse en el manual de VGen.