

## 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. Sin embargo, 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 directa a todas sus estructuras.

### 2 Trabajo autónomo del alumno

Se recomienda *encarecidamente*, antes de empezar esta parte, leer el documento que se encuentra en esta misma carpeta llamado **“Acciones en ANTLR.html”**, en el cual se resumen las posibilidades a la hora de referirse a los atributos de los hijos en las reglas de ANTLR. Las soluciones aquí presentadas utilizarán conceptos explicados en dicho documento.

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 dicten qué elementos (nodos) debe tener un árbol y con qué otros deben conectarse. Pero, a diferencia de otras fases, éstos no se extraerán de la especificación del lenguaje que se halla en *Descripción del Lenguaje.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 es que sea:

- **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 en el Metalenguaje

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

```
program -> definitions:varDefinition* sentences:sentence*;

varDefinition -> type name:string;

intType:type -> ;
realType:type -> ;

print:sentence -> expression;
assignment:sentence -> left:expression right:expression;

arithmeticExpression:expression -> left:expression operator:string right:expression;
variable:expression -> name:string;
intConstant:expression -> value:string;
realConstant:expression -> value: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 que en el capítulo anterior. Aunque en este lenguaje el primer hijo será una *variable* con toda seguridad, 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 ejemplo, en el capítulo 5 (Comprobación de Tipos), permitirá mostrar un ejemplo más completo de inferencia de tipos.

---

<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.

## 3.2 Implementación

### 3.2.1 Implementación de los Nodos del AST

Ahora hay que codificar las clases cuyos objetos formarán los nodos del AST. Esto se puede hacer de dos maneras: de forma manual o usando la herramienta VGen.

#### 3.2.1.1 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.

#### 3.2.1.2 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
expression, sentence, type

NODES
program -> definitions:varDefinition* sentences:sentence*;

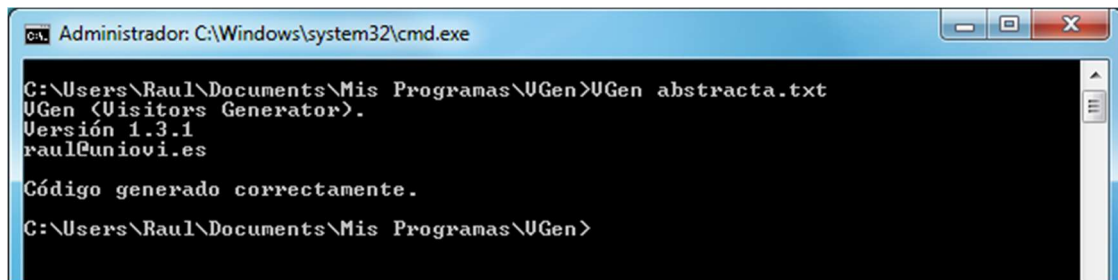
varDefinition -> type name:string;

intType:type -> ;
realType:type -> ;

print:sentence -> expression;
assignment:sentence -> left:expression right:expression;
```

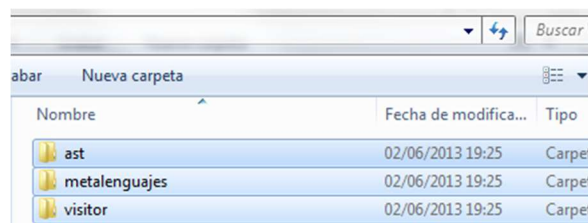
```
arithmeticExpression:expression -> left:expression operator:string right:expression;  
variable:expression -> name:string;  
intConstant:expression -> value:string;  
realConstant:expression -> value:string;
```

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, habrá aparecido una carpeta *output* dentro de la cual hay tres subcarpetas: *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 completas** al proyecto del tutorial. El proyecto ya tiene paquetes con dichos nombres para indicar dónde hay que copiar los ficheros generados. A la hora de arrastrar los ficheros, el editor puede mostrar 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 org.antlr.v4.runtime.*;
import visitor.*;

// print:sentence -> expression

public class Print extends AbstractSentence {

    private Expression expression;

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

        // Lo siguiente se puede borrar si no se quiere la posicion en el fichero.
        // Obtiene la línea/columna a partir de las de los hijos.
        setPositions(expression);
    }
}
```

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

    // Lo siguiente se puede borrar si no se quiere la posicion en el fichero.
    // Obtiene la línea/columna a partir de las de los hijos.
    setPositions(expression);
}

public Expression getExpression() {
    return expression;
}

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

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

public String toString() {
    return "{expression:" + getExpression() + "}";
}
}
```

En la implementación anterior puede observarse:

- El primer comentario es la regla de la gramática abstracta por la cual se ha generado esta clase.
- La clase *Print* deriva de *AbstractSentence*, ya que pertenece a la categoría sintáctica *sentence*.
- Se han definido los hijos como atributos (en este caso sólo un hijo llamado *expresión*) y se han añadido los métodos de acceso (*get* y *set*) correspondientes.
- Las llamadas a *setPositions* en los constructores 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 *ANTLR* (se verá en breve).
- 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.

### 3.2.2 Construcción del AST

Llegados a este punto, se tiene, de capítulos anteriores, la siguiente situación:

- Un fichero *Grammar.g4*, 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 en este capítulo 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 *Grammar.g4* las acciones que irán formando el árbol a medida que se van aplicando las reglas.

Para ello, se van a mostrar tres opciones a la hora de añadir las acciones (de más simples a más sofisticadas). El alumno podrá elegir cualquiera de ellas en función de su preferencia personal. Aunque cualquiera de ellas valdría para este tutorial, se ha decidido poner todas para que el alumno pueda practicar con distintas formas de uso de acciones en ANTLR.

Independientemente de qué solución se elija, todas ellas tienen en común que ahora devuelven un AST que representa a la entrada analizada. Por tanto, hay que actualizar el *main* para que recoja dicho árbol AST. Para ello, en el método *compile*, hay que comentar la llamada actual a *parser.start()* y sustituirla por la que hasta ahora estaba debajo comentada:

```
public class Main {  
  
    public static AST compile(...) throws Exception {  
        // ...  
  
        // IMPORTANTE: Cuando se genere el AST, INTERCAMBIAR las dos líneas siguientes:  
        // parser.start();  
        ast = parser.start().ast;  
  
        // ...  
    }  
}
```

### 3.2.2.1 Opción 1. Versión básica

Esta es la opción más directa en cuanto a uso de funcionalidades de ANTLR. Aunque quizás sea la más fácil de entender, tiene el problema de que no se guarda en el AST la información de línea y columna en los nodos.

```
grammar Grammar;  
  
import Lexicon;  
  
@parser::header {  
    import ast.*;  
}  
  
// -----  
// Solución 1. Versión básica original.  
// - No guarda posiciones de línea.  
// - Usa '$IDENT.text' y '$ctx.expr(0).ast'  
  
start returns[Program ast]  
    : 'DATA' variables 'CODE' sentences EOF { $ast = new Program($variables.list, $sentences.list); }  
    ;  
  
variables returns[List<VarDefinition> list = new ArrayList<VarDefinition>()]  
    : (variable { $list.add($variable.ast); })*  
    ;  
  
sentences returns[List<Sentence> list = new ArrayList<Sentence>()]  
    : (sentence { $list.add($sentence.ast); })*  
    ;
```

```
tipo returns[Type ast]
: 'float' { $ast = new RealType(); }
| 'int'   { $ast = new IntType(); }
;

variable returns[VarDefinition ast]
: tipo IDENT ';' { $ast = new VarDefinition($tipo.ast, $IDENT.text); }
;

sentence returns[Sentence ast]
: 'print' expr ';' { $ast = new Print($expr.ast); }
| expr '=' expr ';' { $ast = new Assignment($ctx.expr(0).ast, $ctx.expr(1).ast); }
;

expr returns[Expression ast]
: expr op=('*' | '/') expr
  { $ast = new ArithmeticExpression($ctx.expr(0).ast, $op.text, $ctx.expr(1).ast); }
| expr op=('+' | '-') expr
  { $ast = new ArithmeticExpression($ctx.expr(0).ast, $op.text, $ctx.expr(1).ast); }
| '(' expr ')' { $ast = $expr.ast; }
| IDENT       { $ast = new Variable($IDENT.text); }
| INT_CONSTANT { $ast = new IntConstant($INT_CONSTANT.text); }
| REAL_CONSTANT { $ast = new RealConstant($REAL_CONSTANT.text); }
;
```

Finalmente hay que ejecutar de nuevo *antlr.bat* para que *ANTLR* 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.

El contenido actual del fichero *source.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.

El programa se ha compilado correctamente.

ASTPrinter: Fichero 'AST.html' generado. Abra dicho fichero para validar el AST generado.

Sin embargo, aunque el mensaje que se recibe al ejecutar es el mismo que en el capítulo anterior, la diferencia es que ahora se ha creado un árbol en memoria. El fichero '*AST.html*' muestra la estructura del árbol que se ha creado para que pueda ser revisado:



[ASTPrinter] -----	line:col	line:col
Program →	<i>null</i>	<i>null</i>
definitions List<VarDefinition> =		
VarDefinition →	<i>null</i>	<i>null</i>
RealType →	<i>null</i>	<i>null</i>
"precio"		
VarDefinition →	<i>null</i>	<i>null</i>
IntType →	<i>null</i>	<i>null</i>
"ancho"		
sentences List<Sentence> =		
Assignment →	<i>null</i>	<i>null</i>
"ancho" Variable.name	<i>null</i>	<i>null</i>
ArithmeticExpression →	<i>null</i>	<i>null</i>
"25" IntConstant.value	<i>null</i>	<i>null</i>
"*"		
ArithmeticExpression →	<i>null</i>	<i>null</i>
"2" IntConstant.value	<i>null</i>	<i>null</i>
"+"		
"1" IntConstant.value	<i>null</i>	<i>null</i>
Print →	<i>null</i>	<i>null</i>
"ancho" Variable.name	<i>null</i>	<i>null</i>
Assignment →	<i>null</i>	<i>null</i>
"precio" Variable.name	<i>null</i>	<i>null</i>
"5.0" RealConstant.value	<i>null</i>	<i>null</i>
Print →	<i>null</i>	<i>null</i>
ArithmeticExpression →	<i>null</i>	<i>null</i>
"precio" Variable.name	<i>null</i>	<i>null</i>
"/"		
"2.0" RealConstant.value	<i>null</i>	<i>null</i>

[ASTPrinter] -----

Este fichero *html* ha sido creado por la *ASTPrinter* (clase creada por *VGen* en la carpeta *visitor*) mediante la invocación realizada desde el *main*. Si no se desea generar este HTML, bastará con borrar dicha línea y eliminar *ASTPrinter* del proyecto (ninguna otra clase tiene dependencias con ella).

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

    AST ast = compile(program, errorManager); // Poner args[0] en la versión final
    if (errorManager.errorsCount() == 0)
        System.out.println("El programa se ha compilado correctamente.");

    ASTPrinter.toHtml(program, ast, "AST"); // Utilidad generada por VGen (opcional)
}
```

### 3.2.2.2 Opción 2. Versión con información de línea y columna

Esta segunda solución aprovechará el código opcional que genera automáticamente *VGen*. Las ventajas de esta versión son:

- Añade automáticamente la información de fila/columna a los nodos. Ello se consigue pasando al constructor de los nodos *\$TOKEN* en vez de *\$TOKEN.text*. Es decir, en vez de pasar al constructor del nodo sólo el lexema, se le pasa todo el token para que pueda sacar la línea y la columna del mismo.
  - En la regla *variable* se puede ver que ahora se pasa *\$IDENT* en vez de *\$IDENT.text*.
  - En las tres últimas reglas de *expr* (*IDENT*, *INT\_CONSTANT* y *REAL\_CONSTANT*), se puede ver que también se ha quitado el *“.text”* en el constructor.

Esto no solo hace que los nodos terminales tengan la posición del fichero, sino que *VGen* hace que los nodos intermedios también obtengan sus posiciones a partir de sus hijos, quedando así todos los nodos con esta información.

- Simplifica las reglas. Cuando se accede por índice (por ejemplo '\$ctx.expr(0).ast'), no es necesario poner ".ast", ya que el constructor generado por *VGen* ya se encarga de sacar dicho atributo.
  - En la regla *sentence*, en el constructor de *Assignment*, se han quitado de los parámetros el ".ast".
  - Lo mismo se ha hecho en los constructores de *ArithmeticExpresión* de la regla *expr*.

Queda así un fichero *Grammar.g4* algo más compacto y fácil de entender:

```
start returns[Program ast]
: 'DATA' variables 'CODE' sentences EOF { $ast = new Program($variables.list, $sentences.list); }
;

variables returns[List<VarDefinition> list = new ArrayList<VarDefinition>()]
: (variable { $list.add($variable.ast); })*
;

sentences returns[List<Sentence> list = new ArrayList<Sentence>()]
: (sentence { $list.add($sentence.ast); })*
;

tipo returns[Type ast]
: 'float' { $ast = new RealType(); }
| 'int' { $ast = new IntType(); }
;

variable returns[VarDefinition ast]
: tipo IDENT ';' { $ast = new VarDefinition($tipo.ast, $IDENT); }
;

sentence returns[Sentence ast]
: 'print' expr ';' { $ast = new Print($expr.ast); }
| expr '=' expr ';' { $ast = new Assignment($ctx.expr(0), $ctx.expr(1)); }
;

expr returns[Expression ast]
: expr op=('*' | '/') expr { $ast = new ArithmeticExpression($ctx.expr(0), $op, $ctx.expr(1)); }
| expr op=('+' | '-') expr { $ast = new ArithmeticExpression($ctx.expr(0), $op, $ctx.expr(1)); }
| '(' expr ')' { $ast = $expr.ast; }
| IDENT { $ast = new Variable($IDENT); }
| INT_CONSTANT { $ast = new IntConstant($INT_CONSTANT); }
| REAL_CONSTANT { $ast = new RealConstant($REAL_CONSTANT); }
;
```

Además, se obtiene más información en *AST.html*. Si ahora se vuelve a usar *antlr.bat* y se ejecuta el programa, se obtiene la nueva versión de con información con posiciones en el fichero:

[ASTPrinter]	-----	line:col	line:col	
Program →		2:11	9:22	float <u>precio</u> ; ... <u>print</u>
<u>precio</u> / 2.0;				
. definitions List<VarDefinition> =				
. . VarDefinition →		2:11	2:16	float <u>precio</u> ;
. .   RealType →		2:11	2:16	<b>null null</b>
. .   "precio"				
. . VarDefinition →		3:9	3:13	int <u>ancho</u> ;
. .   IntType →		3:9	3:13	<b>null null</b>
. .   "ancho"				
. sentences List<Sentence> =				
. . Assignment →		5:5	5:23	<u>ancho</u> = 25 * (2 + 1);
. .   "ancho" Variable.name		5:5	5:9	<u>ancho</u> = 25 * (2 + 1);
. .   ArithmeticExpression →		5:13	5:23	ancho = <u>25</u> * (2 + 1);
. . . "25" IntConstant.value		5:13	5:14	ancho = <u>25</u> * (2 + 1);
. . . "★"				
. . . ArithmeticExpression →		5:19	5:23	ancho = 25 * (2 + 1);
. . . . "2" IntConstant.value		5:19	5:19	ancho = 25 * ( <u>2</u> + 1);
. . . . "+"				
. . . . "1" IntConstant.value		5:23	5:23	ancho = 25 * (2 + <u>1</u> );
. . Print →		6:11	6:15	print <u>ancho</u> ;
. .   "ancho" Variable.name		6:11	6:15	print <u>ancho</u> ;
. . Assignment →		8:5	8:16	<u>precio</u> = 5.0;
. .   "precio" Variable.name		8:5	8:10	<u>precio</u> = 5.0;
. .   "5.0" RealConstant.value		8:14	8:16	precio = <u>5.0</u> ;
. . Print →		9:11	9:22	print <u>precio</u> / 2.0;
. .   ArithmeticExpression →		9:11	9:22	print <u>precio</u> / 2.0;
. . . "precio" Variable.name		9:11	9:16	print <u>precio</u> / 2.0;
. . . "/"				
. . . "2.0" RealConstant.value		9:20	9:22	print precio / <u>2.0</u> ;
[ASTPrinter]	-----			

En la columna central puede verse la posición inicial y final de cada nodo en el fichero de entrada (teniendo cada posición su línea y columna). 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 del fichero.

Debido a que se consigue *más* funcionalidad y se escribe algo menos, *se recomienda esta versión* en vez de la anterior.

### 3.2.2.3 Opción 3. Versión mínima

Esta última solución se aprovechará de las opciones mostradas en el documento antes citado "**Acciones en ANTLR.html**" (concretamente del operador "+=") junto con el código que genera automáticamente VGen para sacarle partido.

Esta versión eliminará los no-terminales creados únicamente para listas. En este caso, eliminará las reglas *variables* y *sentences* al usar en la regla *start* el operador '+='. Esto hace que al constructor de la clase Programa, se le pase una lista de ParserRuleContext (el contexto de las reglas, ver "**Acciones en ANTLR.html**"). El código que genera VGen para dicho constructor se encargará de sacar los ast de los de dicha lista de contextos y añadirlos como hijos en el nodo.

```

start returns[Program ast]
: 'DATA' lv+=variable* 'CODE' ls+=sentence* EOF { $ast = new Program($lv, $ls); }
;

tipo returns[Type ast]
: 'float' { $ast = new RealType(); }
| 'int' { $ast = new IntType(); }
;

variable returns[VarDefinition ast]
: tipo IDENT ';' { $ast = new VarDefinition($tipo.ast, $IDENT); }
;

sentence returns[Sentence ast]
: 'print' expr ';' { $ast = new Print($expr.ast); }
| expr '=' expr ';' { $ast = new Assignment($ctx.expr(0), $ctx.expr(1)); }
;

expr returns[Expression ast]
: expr op=('*' | '/') expr { $ast = new ArithmeticExpression($ctx.expr(0), $op, $ctx.expr(1)); }
| expr op=('+' | '-') expr { $ast = new ArithmeticExpression($ctx.expr(0), $op, $ctx.expr(1)); }
| '(' expr ')' { $ast = $expr.ast; }
| IDENT { $ast = new Variable($IDENT); }
| INT_CONSTANT { $ast = new IntConstant($INT_CONSTANT); }
| REAL_CONSTANT { $ast = new RealConstant($REAL_CONSTANT); }
;

```

Entre esta versión y la anterior, se deja a la decisión del alumno cuál prefiere implementar. En cualquier caso, la que no se recomienda es la opción 1, ya que no dispone en el AST de las posiciones del fichero y, por tanto, cuando en fases posteriores se necesite esta información no estará disponible.

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

## 4 Resumen de Cambios

Fichero	Acción	Descripción
<b>Gramática 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
<b>metalenguajes\*.html</b>	Generado	Carpeta con las plantillas de Word que se utilizarán en futuros capítulos
<b>Grammar.g4</b>	Modificado	Se añaden las acciones que crean el AST
<b>GrammarParser.java</b>	Generado	Implementación del Analizador Sintáctico. Creado con <i>ANTLR</i> a partir de <i>Grammar.g4</i>
<b>AST.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 'AST.html', incluso en las opciones 2 y 3 en las que en el AST se incluye la posición en el fichero, 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"
```

Esto se debe a la forma en la que se obtiene las posiciones de cada nodo. Básicamente, el proceso consiste en:

- Los nodos terminales obtienen su posición de su Token (o Tokens).
- Los nodos intermedios obtienen su posición de sus hijos. La posición inicial se obtiene a partir de la de su primer hijo y la posición final a partir de la de su último hijo (si algún hijo no tiene la posición buscada se pasa al siguiente).

Y así, de abajo hacia arriba, todos los nodos obtienen de manera automática y sin ningún código adicional toda la gestión de posiciones.

El proceso anteriormente descrito es el que se hace en el método *setPositions* que *VGen* generó en cada constructor:

```
public ArithmeticExpression(Expression left, String operator, Expression right) {
    this.left = left;
    this.operator = operator;
    this.right = right;

    // Lo siguiente se puede borrar si no se quiere la posicion en el fichero.
    // Obtiene la linea/columna a partir de las de los hijos.
    setPositions(left, right);
}
```

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

```
tipo returns[Type ast]
: 'float' { $ast = new RealType(); }
| 'int' { $ast = new IntType(); }
;
```

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 a 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 en los nodos un método *setPositions* que permite pasarle uno o más símbolos (independientemente de que sean hijos suyos o no) para que extraiga de ellos sus posiciones.

La forma más sencilla es aprovechar el token que ha servido para reconocer el tipo (*'float'* o *'int'*) y, aunque no se vaya a añadir al árbol, extraer sus posiciones mediante el método *setPositions* para copiarlas en el nodo indicado:

```
tipo returns[Type ast]
  : t='float' { $ast = new RealType(); $ast.setPositions($t); }
  | t='int'   { $ast = new IntType(); $ast.setPositions($t); }
  ;
```

Otra forma de hacerlo, sin tener que usar etiquetas, es utilizar *`\$ctx.start`*. El atributo *'start'* lo guarda ANTLR en el contexto de cada regla apuntando siempre al primer token que reconozca dicha regla. En este caso, serán los tokens *float* e *int* (por lo que hace lo misma que la solución anterior).

```
tipo returns[Type ast]
  : 'float' { $ast = new RealType(); $ast.setPositions($ctx.start); }
  | 'int'   { $ast = new IntType(); $ast.setPositions($ctx.start); }
  ;
```

Si después de volver a pasar *Grammar.g4* por ANTLR se ejecuta el código resultante, se obtiene este otro árbol:

```
Program →                2:5 9:22  float precio; ... print precio / 2.0;
. definitions List<VarDefinition> =
. . VarDefinition →      2:5 2:16  float precio;
. . | RealType →        2:5 2:9   float precio;
. . | "precio"
. . VarDefinition →      3:5 3:13  int ancho;
. . | IntType →         3:5 3:7   int ancho;
. . | "ancho"
```

Nótese cómo no solo los tipos tienen ahora su posición inicial y final, sino que las posiciones de sus nodos padre, los dos *VarDefinition*, también han mejorado respecto al árbol anterior, ya que ahora pueden utilizar la información de su hijo *tipo* (obsérvese el texto que aparece subrayado después de las posiciones de las *VarDefinition* y compárese con el texto de la versión anterior).

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*.