

## VGen. Manual de usuario

### Contenido

Objetivo .....	2
Ejecución de VGen .....	2
Contenidos generados por VGen .....	4
Carpeta <i>ast</i> .....	4
Carpeta <i>Visitor</i> .....	5
Interfaz <i>Visitor</i> .....	5
Clase <i>DefaultVisitor</i> .....	5
Fichero <i>_PlantillaParaVisitors.txt</i> .....	6
Clase <i>ASTPrinter</i> .....	7
Carpeta <i>metalenguajes</i> .....	7
Funcionalidades adicionales de VGen .....	9
Clase <i>Token</i> .....	9
Funcionalidad 1. Conversiones automáticas.....	9
Funcionalidad 2. Posiciones de los nodos en el fichero .....	11
Utilidad de las posiciones.....	11
Cálculo automático de las posiciones con VGen .....	13
Nodos sin posiciones .....	14
Necesidad de la posición en los nodos.....	15
Asignación manual de posiciones .....	15
Apéndice I. Opciones en línea de comando de VGen .....	20
Apéndice II. Especificación de VGen .....	20
Léxico .....	20
Sintaxis .....	21
Validaciones semánticas. Gramática Atribuida.....	21

Raúl Izquierdo Castanedo  
[raul@uniovi.es](mailto:raul@uniovi.es)

Área de Lenguajes y Sistemas Informáticos  
Departamento de Informática

Escuela de Ingeniería Informática de Oviedo  
Universidad de Oviedo



## Objetivo

*VGen* es una herramienta creada como soporte de la asignatura *Diseño de Lenguajes de Programación* de la *Escuela de Ingeniería Informática* de la Universidad de Oviedo. Su objetivo es facilitar el diseño de compiladores de lenguajes de programación permitiendo centrarse en las partes conceptuales del proceso (la especificación de cada fase mediante su metalenguaje) dejando a la herramienta la implementación del código que no aporta conocimiento propio de esta materia.

## Ejecución de *VGen*

La entrada de *VGen* es un fichero de texto con la *Gramática Abstracta* (GAb) del lenguaje para el cual se quiere generar el código. En la carpeta de *VGen* se encuentra un fichero *abstracta.txt* con la siguiente gramática de ejemplo:

```
CATEGORIES
expresion, sentencia, tipo

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

defVariable -> tipo nombre:string;

intType:tipo -> ;
realType:tipo -> ;

print:sentencia -> expresion;
asignacion:sentencia -> left:expresion right:expresion;
return:sentencia -> expresion;

exprAritmetica:expresion -> left:expresion operador:string right:expresion;
variable:expresion -> string;
literalInt:expresion -> valor:string;
```

Como corresponde a una gramática abstracta, ésta define los nodos con los que se forman los AST de dicho lenguaje. Para cada uno de los nodos se indica el nombre y el tipo de cada uno de sus hijos<sup>1</sup> así como las categorías sintácticas a las que pertenece<sup>2</sup> (en este caso, se han declarado las categorías *expresión*, *sentencia* y *tipo*)<sup>3</sup>.

Para generar el código a partir de una gramática abstracta hay que invocar el fichero *VGen.bat*:

```
C:\Users\Raul\>VGen abstracta.txt
VGen (Visitors Generator).
Versión 1.3.2
raul@uniovi.es
Escuela de Ingeniería informática de Oviedo. Universidad de Oviedo

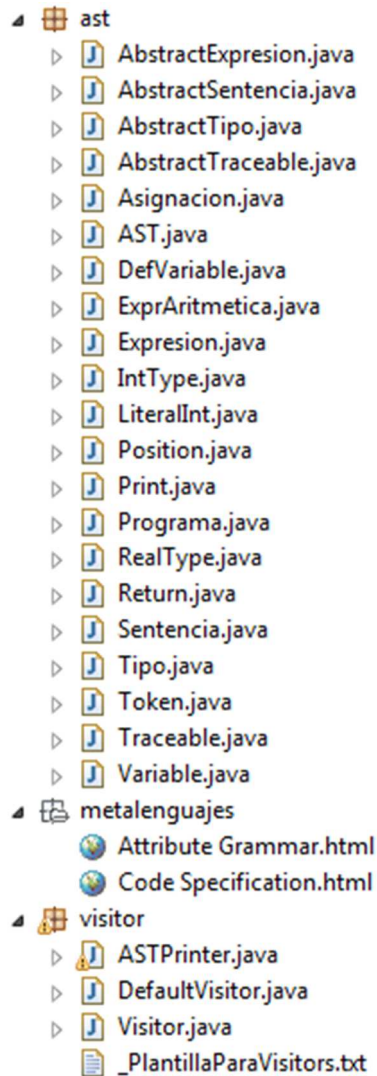
El fichero 'abstracta.txt' no contiene errores. Código generado con éxito.
```

<sup>1</sup> Solo es obligatorio el tipo de los hijos. Si un hijo no tiene nombre, *VGen* le asigna uno convirtiendo en minúscula la primera letra de su tipo. Por ejemplo, en *DefVariable* el primer hijo se llamará *tipo*. De esta manera, se evita tener que escribir los nombres habituales como *tipo*, *expresión*, *sentencia*, etc.

<sup>2</sup> Aunque en este ejemplo no se muestra, un nodo puede pertenecer a más de una categoría sintáctica. Se escribirían todas ellas separadas por coma.

<sup>3</sup> Tanto las categorías como los nodos se pueden poner en cualquier combinación de minúsculas y mayúsculas. *VGen* convertirá la primera letra a mayúscula para generar las clases Java correspondientes.

Una vez hecho lo anterior aparecerán varios ficheros repartidos en tres directorios llamados *ast*, *metalenguajes* y *visitor*.

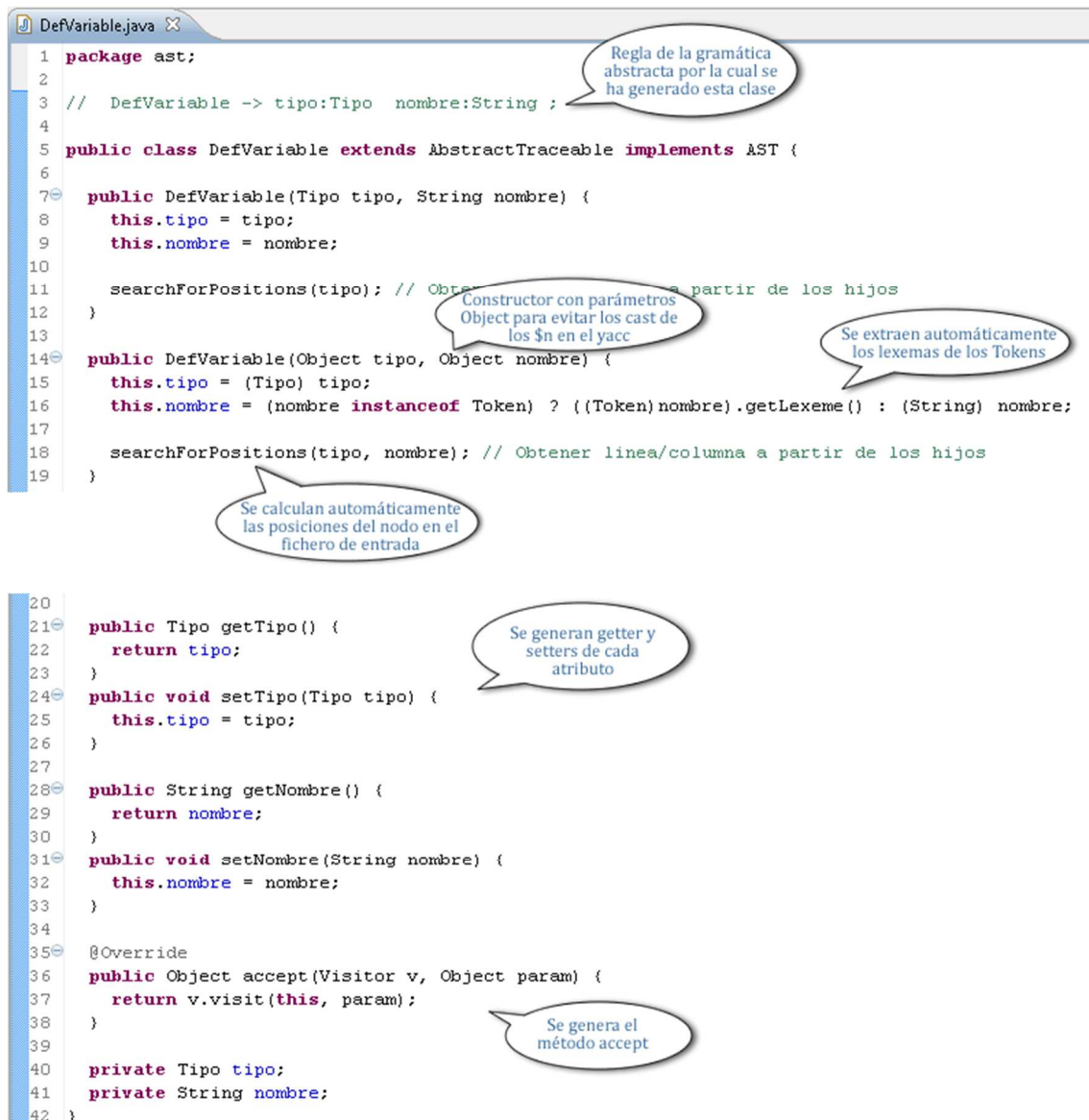


## Contenidos generados por VGen

En este documento se describen algunos de los ficheros generados por VGen. Para ver el uso de cada uno de ellos dentro del proceso de construcción de un compilador, se recomienda descargar "*Tutorial. Creación de un compilador básico*". En dicho tutorial se utiliza VGen y se puede ver en más detalle dónde y cómo se ha utilizado cada uno de los ficheros generados.

### Carpeta ast

En esta carpeta se encuentran las clases que representan los nodos del AST.



```
1 package ast;
2
3 // DefVariable -> tipo:Tipo nombre:String ;
4
5 public class DefVariable extends AbstractTraceable implements AST {
6
7     public DefVariable(Tipo tipo, String nombre) {
8         this.tipo = tipo;
9         this.nombre = nombre;
10
11         searchForPositions(tipo); // Obtener linea/columna a partir de los hijos
12     }
13
14     public DefVariable(Object tipo, Object nombre) {
15         this.tipo = (Tipo) tipo;
16         this.nombre = (nombre instanceof Token) ? ((Token) nombre).getLexeme() : (String) nombre;
17
18         searchForPositions(tipo, nombre); // Obtener linea/columna a partir de los hijos
19     }
20
21     public Tipo getTipo() {
22         return tipo;
23     }
24
25     public void setTipo(Tipo tipo) {
26         this.tipo = tipo;
27     }
28
29     public String getNombre() {
30         return nombre;
31     }
32
33     public void setNombre(String nombre) {
34         this.nombre = nombre;
35     }
36
37     @Override
38     public Object accept(Visitor v, Object param) {
39         return v.visit(this, param);
40     }
41
42     private Tipo tipo;
43     private String nombre;
44 }
```

Regla de la gramática abstracta por la cual se ha generado esta clase

Constructor con parámetros Object para evitar los cast de los \$n en el yacc

Se extraen automáticamente los lexemas de los Tokens

Se calculan automáticamente las posiciones del nodo en el fichero de entrada

Se generan getter y setters de cada atributo

Se genera el método accept

También se genera una clase con el prefijo *Abstract* por cada una de las categorías sintácticas. Estas clases son útiles para añadir atributos a todos los nodos de una categoría en las fases de análisis semántico y de generación de código:

```
1 package ast;
2
3 public abstract class AbstractExpresion extends AbstractTraceable implements Expresion {
4
5 }
```

Las clases abstractas son para añadir atributos a todos los nodos de una categoría cómodamente

Por ejemplo aquí se añadiría el Tipo a las expresiones

## Carpeta Visitor

En esta carpeta se generan las clases que se encargan de implementar la infraestructura del patrón *Visitor*.

### Interfaz Visitor

Por un lado, aparece el propio interfaz *Visitor*:

```
Visitor.java
1 package visitor;
2
3 import ast.*;
4
5 public interface Visitor {
6     public Object visit(Programa node, Object param);
7     public Object visit(DefVariable node, Object param);
8     public Object visit(IntType node, Object param);
9     public Object visit(RealType node, Object param);
10    public Object visit(Print node, Object param);
11    public Object visit(Asigna node, Object param);
12    public Object visit(Return node, Object param);
13    public Object visit(ExprAritmetica node, Object param);
14    public Object visit(Variable node, Object param);
15    public Object visit(LiteralInt node, Object param);
16 }
```

### Clase DefaultVisitor

También se genera una clase *DefaultVisitor*, la cual es una implementación del interfaz que incluye en un método *visit* por cada nodo con el código que recorre sus hijos.

```
1 public class DefaultVisitor implements Visitor {
2
3     // class Programa { String nombre; List<DefVariable> definiciones; List<Sentencia> sentencias; }
4     public Object visit(Programa node, Object param) {
5
6         if (node.getDefiniciones() != null)
7             for (DefVariable child : node.getDefiniciones())
8                 child.accept(this, param);
9
10        if (node.getSentencias() != null)
11            for (Sentencia child : node.getSentencias())
12                child.accept(this, param);
13
14        return null;
15    }
16
17    // Resto de los visit...
18 }
```

Implementación del interfaz Visitor para ser derivada por el resto de los Visitor

Implementación que recorre los hijos del nodo

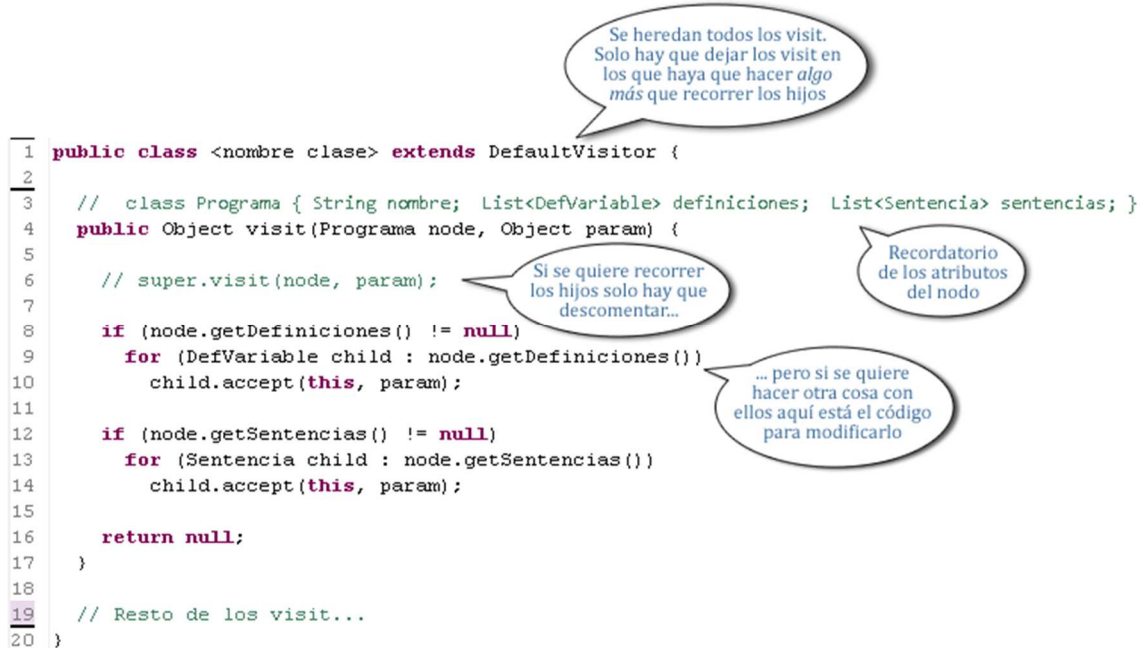
Se pueden desactivar las comprobaciones de null

No es necesario modificar esta clase. Para crear nuevos Visitor usar PlantillaParaVisitors.txt

## Fichero `_PlantillaParaVisitors.txt`

Para crear un nuevo *Visitor* bastaría con crear una clase que derivara de *DefaultVisitor*. Este nuevo *Visitor* ya realizaría todo el recorrido por los nodos del árbol (aunque no haría nada en cada nodo aún).

Sin embargo, para crear un nuevo *Visitor* es más cómodo aún usar el fichero `_PlantillaParaVisitors.txt` que genera *VGen*:



```
1 public class <nombre clase> extends DefaultVisitor {
2
3 // class Programa { String nombre; List<DefVariable> definiciones; List<Sentencia> sentencias; }
4 public Object visit(Programa node, Object param) {
5
6 // super.visit(node, param);
7
8 if (node.getDefiniciones() != null)
9     for (DefVariable child : node.getDefiniciones())
10         child.accept(this, param);
11
12 if (node.getSentencias() != null)
13     for (Sentencia child : node.getSentencias())
14         child.accept(this, param);
15
16 return null;
17 }
18
19 // Resto de los visit...
20 }
```

Se heredan todos los visit. Solo hay que dejar los visit en los que haya que hacer algo más que recorrer los hijos

Recordatorio de los atributos del nodo

Si se quiere recorrer los hijos solo hay que descomentar...

... pero si se quiere hacer otra cosa con ellos aquí está el código para modificarlo

Este fichero contiene implementados todos los métodos *visit* con el código de recorrido de sus hijos (es igual que *DefaultVisitor*). Para crear un nuevo *Visitor*, basta con cortar y pegar este código y ya se tendrá un *Visitor* que compila y que al ejecutarlo recorrerá todo el árbol (sin hacer nada aún en él).

Si en algún método *visit* no se quiere hacer nada más que recorrer los hijos, entonces se puede borrar dicho método completamente (se heredará de *DefaultVisitor* el mismo código que se acaba de borrar).

En cambio, cuando se quiera realizar alguna tarea sobre algún nodo, la plantilla ofrece dos opciones:

- Lo habitual será borrar todo el código del método dejando solo la llamada a *super.visit* (descomentando dicha llamada). De esta manera, cada método *visit* se puede centrar en la tarea que tiene que realizar sobre su nodo del AST olvidándose del recorrido de los hijos.
- Para los casos en los que se necesite realizar alguna tarea durante el recorrido de los hijos y, por tanto, no valga la implementación heredada de *DefaultVisitor* (por ejemplo, para ir comprobando el tipo de cada hijo), se incluye entonces todo el código de recorrido de los hijos para que sea modificado de la forma conveniente.

La plantilla se ofrece porque es más rápido y cómodo partir de un *Visitor* completo e ir borrando métodos (o adaptando el código de recorrido) que encontrarse con un *Visitor* vacío y tener que recordar cuántos nodos *visit* hay que implementar y cuál es código de recorrido adecuado de los hijos de cada uno.

## Clase *ASTPrinter*

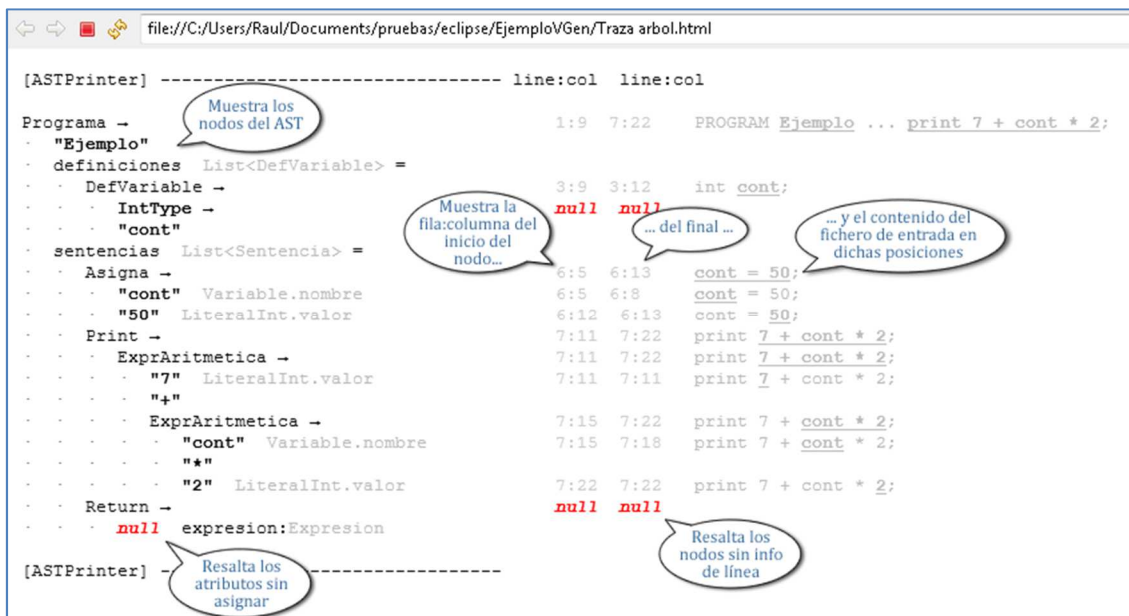
Por último, se genera una clase *ASTPrinter* que permite validar el AST generado de un vistazo. Esta clase genera una traza del AST en HTML (*Traza arbol.html*) que permite comprobar su estructura, detectar rápidamente hijos a *null* y comprobar las posiciones de los nodos (línea y columna) en el fichero fuente.

Supóngase que el compilador crea un AST para el siguiente programa de entrada:

```
PROGRAM Ejemplo
DATA
  int cont;

CODE
  cont = 50;
  print 7 + cont * 2;
  return;
```

*ASTPrinter* generaría el siguiente HTML para poder ver el AST creado:



[ASTPrinter]	-----	line:col	line:col	
Programa -		1:9	7:22	PROGRAM Ejemplo ... print 7 + cont * 2;
- "Ejemplo"				
- definiciones List<DefVariable> =				
- DefVariable -		3:9	3:12	int cont;
- IntType -				
- "cont"				
- sentencias List<Sentencia> =				
- Asigna -		6:5	6:13	cont = 50;
- "cont" Variable.nombre		6:5	6:8	cont = 50;
- "50" LiteralInt.valor		6:12	6:13	cont = 50;
- Print -		7:11	7:22	print 7 + cont * 2;
- ExprAritmetica -		7:11	7:22	print 7 + cont * 2;
- "7" LiteralInt.valor		7:11	7:11	print 7 + cont * 2;
- "+"				
- ExprAritmetica -		7:15	7:22	print 7 + cont * 2;
- "cont" Variable.nombre		7:15	7:18	print 7 + cont * 2;
- "*"				
- "2" LiteralInt.valor		7:22	7:22	print 7 + cont * 2;
- Return -				
- null expresion:Expresion				
[ASTPrinter]	-----			

## Carpeta metalenguajes

A la hora de crear en *Word* tanto el documento para la *Gramática Atribuida* (semántico) como el de la *Especificación de Código*, hay que partir de la gramática abstracta y formatearla en distintas tablas. Para evitar este trabajo, *VGen* ya genera la estructura de estos dos documentos en HTML<sup>4</sup>.

<sup>4</sup> Están preparados para ser abiertos en *Word* directamente



## Attribute Grammar

Nodo	Predicados	Reglas Semánticas
<b>programa</b> → <i>nombre:String definiciones: defVariable* sentencias: sentencia*</i>		
<b>defVariable</b> → <i>tipo: tipo nombre: String</i>		Centrarse en qué hay que validar
<b>intType: tipo</b> → λ		
<b>realType: tipo</b> → λ		
<b>print: sentencia</b> → <i>expresion: expresion</i>		
<b>asigna: sentencia</b> → <i>left: expresion right: expresion</i>		
<b>return: sentencia</b> → <i>expresion: expresion</i>		Recordatorio del nombre y tipo de los hijos (para los predicados)
<b>exprAritmetica: expresion</b> → <i>left: expresion operator: String right: expresion</i>		
<b>variable: expresion</b> → <i>string: String</i>		
<b>literalInt: expresion</b> → <i>valor: String</i>		

Recordatorio de operadores (para cortar y pegar):  $\Rightarrow \neq \emptyset \in \cup \cap \subset \varnothing \sum \exists \forall$

## Atributos

Categoría Sintáctica	Nombre del atributo	Tipo Java	Heredado/Sintetizado

Y la especificación de código tendría esta forma:

## Code Specification

Función	Plantillas de Código
$f_1[[\text{programa}]]$	$f_1[[\text{programa} \rightarrow \text{nombre:String definiciones: defVariable* sentencias: sentencia*}]] =$
$f_2[[\text{defVariable}]]$	$f_2[[\text{defVariable} \rightarrow \text{tipo: tipo nombre: String}]] =$
$f_3[[\text{tipo}]]$	$f_3[[\text{intType} \rightarrow \lambda]] =$
	$f_3[[\text{realType} \rightarrow \lambda]] =$
$f_4[[\text{sentencia}]]$	$f_4[[\text{print} \rightarrow \text{expresion: expresion}]] =$
	$f_4[[\text{asigna} \rightarrow \text{left: expresion right: expresion}]] =$
	$f_4[[\text{return} \rightarrow \text{expresion: expresion}]] =$
$f_5[[\text{expresion}]]$	$f_5[[\text{exprAritmetica} \rightarrow \text{left: expresion operator: String right: expresion}]] =$
	$f_5[[\text{variable} \rightarrow \text{string: String}]] =$
	$f_5[[\text{literalInt} \rightarrow \text{valor: String}]] =$

Se crea una función de código por cada categoría sintáctica...

... y ya aparecen las cabeceras de todas sus plantillas de código

Solo hay que determinar el código que hay que generar en cada una

Recordatorio del nombre y tipo de los hijos (para invocar sus funciones)



## Funcionalidades adicionales de VGen

Las dos funcionalidades adicionales de VGen son las *conversiones automáticas* y el *cálculo de posiciones* de los nodos. Ambas funcionalidades requieren el uso de la clase *Token* (generada por VGen) en Yacc. No es obligatorio el uso de dicha clase para usar todo lo visto anteriormente de VGen. Si VGen detecta (de manera transparente) que la clase *Token* no se está usando, simplemente no se realizarán las dos funciones adicionales que se describirán a continuación.

### Clase Token

La clase *Token* se usa para dejar en *yylval* los símbolos terminales con toda su información:

```
// En Yacc
int yylex() {
    int token = lex.yylex();
    yylval = new Token(token, lex.yytext(), lex.yyline(), lex.yycolumn());
    return token;
}
```

Tanto la clase *Token* como la clase *Position* (usada por *Token*) son generadas por VGen:

```
public class Token implements Traceable {

    public Token(int tokenType, String lexeme, int line, int column) {
        this.tokenType = tokenType;
        this.lexeme = lexeme;
        this.line = line;
        this.column = column;
    }

    public Position getStart() {
        if (start == null)
            start = new Position(line, column);
        return start;
    }

    public Position getEnd() {
        if (end == null)
            end = new Position(line, column + lexeme.length() - 1);
        return end;
    }

    // getters y setters omitidos...

    private int tokenType;
    private String lexeme;
    private int line;
    private int column;
    private Position start, end;
}
```

```
public class Position {
    public Position(int line, int column) {
        this.line = line; this.column = column;
    }
    // getters y setters omitidos...
    private int line, column;
}
```

## Funcionalidad 1. Conversiones automáticas

Como se ha visto anteriormente, *VGen* genera un constructor que admite argumentos de tipo *Object* y que se encarga de realizar los *cast* al tipo correspondiente de los atributos. *VGen* añade además a dicho constructor el código que realiza las conversiones de objetos *Token* a todos los tipos primitivos y a *String*.

Supóngase la siguiente definición de un nodo en la gramática abstracta:

```
NodoEjemplo -> expr:Expr texto:String nombres:String* i:int ;
```

Al crear con *VGen* la clase Java para dicho nodo, se obtiene la siguiente implementación de uno de sus constructores:

```
public NodoEjemplo(Object expr, Object texto, Object nombres, Object i) {  
    this.expr = (Expr) expr;  
  
    this.texto = (texto instanceof Token) ? ((Token)texto).getLexeme()  
        : (String) texto;  
  
    this.nombres = tokensToStrings(nombres);  
  
    this.i = (i instanceof Token) ? Integer.parseInt(((Token)i).getLexeme())  
        : (Integer) i;  
  
    // resto del constructor omitido...  
}
```

El argumento *expr*, al no ser un terminal y por tanto no haber posibilidad de que sea un objeto de tipo *Token*, recibe la conversión normal a su tipo.

Sin embargo, *VGen* detecta que los otros tres argumentos son terminales y realiza un tratamiento especial con ellos:

- En el caso de *texto*, se extrae el lexema si es un *Token*.
- En el caso de *nombres*, mediante el método *tokensToStrings*, se extraen todos los lexemas de la *List<Token>* recibida y se dejan en una *List<String>*. Es decir, se realiza el paso anterior para cada uno de los elementos de la lista, creando así otra lista con el resultado.
- En el caso del argumento *i*, además de extraer el lexema, se convierte a entero antes de asignarlo<sup>5</sup>. Si no es un *Token*, entonces realiza el *unboxing* usando la clase envoltorio (*wrapper*) adecuada para dicho tipo primitivo.

Todo el código anterior permite crear el nodo desde *Yacc* de la siguiente manera ahorrando todas las conversiones que supone usar la clase *Token* en *yylval* (conversiones que también hubieran sido necesarias aunque se hubiera usado *String* en su lugar):

```
NodoEjemplo: expr IDENT idents CTEINT { $$ =new NodoEjemplo($1, $2, $3, $4); }  
  
idents: { $$ = new ArrayList<Token>(); }  
| idents IDENT { ((List<Token>)$1).add($2); $$ = $1; }
```

En definitiva, se trata de definir en la gramática abstracta los tipos que realmente se necesiten para los atributos de los nodos y dejar que *VGen* haga las conversiones necesarias.

<sup>5</sup> Se usaría el método adecuado de conversión de *String* en función del tipo primitivo (*float*, *char*, *boolean*, etc.)

## Funcionalidad 2. Posiciones de los nodos en el fichero

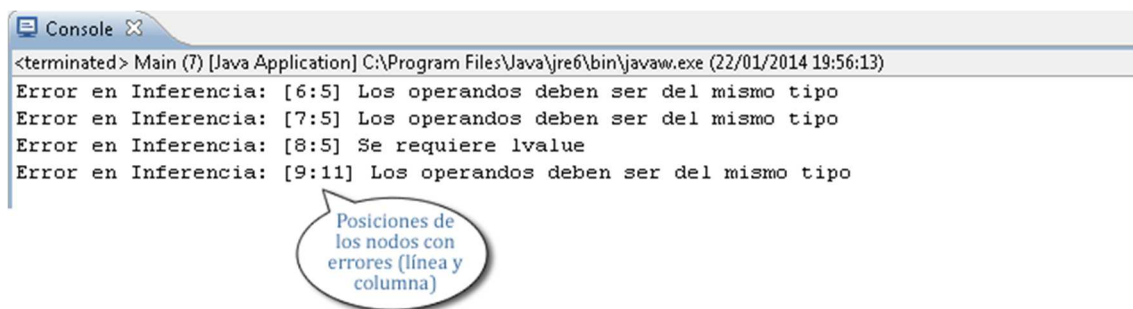
### Utilidad de las posiciones

El análisis léxico es el único que trata con el fichero de entrada y sabe en qué posición del mismo se encontraba cada símbolo del programa. Aunque no es obligatorio, es muy conveniente almacenar esta información en el AST. Las dos principales ventajas de hacer esto son:

- Mejorar los mensajes de error
- Facilitar la validación del código generado.

#### *Ventaja1: Mejorar mensajes de error*

La primera ventaja afecta a la implementación de las fases de análisis del traductor. A la hora de implementar dichas fases se detectarán errores en el fichero de entrada. Si no se ha guardado en el AST la posición de cada símbolo, lo único que se puede decir al usuario es que en el fichero hay un error, pero no dónde está. Si se tienen las posiciones se podrán dar mensajes como los siguientes:



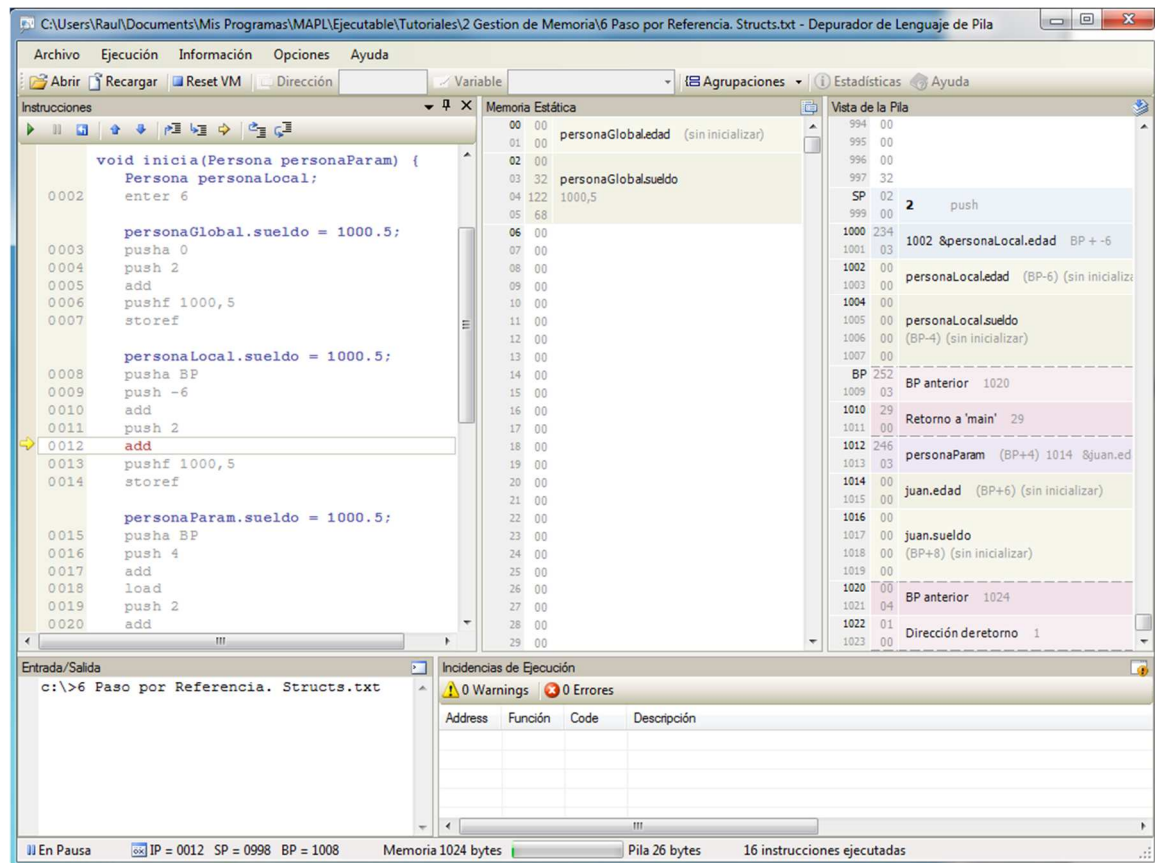
Esto será muy útil para el alumno, tanto durante el desarrollo de la práctica como en el examen final, cuando su compilador señale errores sobre una entrada aparentemente correcta.

#### *Ventaja 2: Generación de código*

La segunda ventaja afecta a las fases de síntesis del traductor. Es probable que la implementación inicial del generador de código no produzca el código correcto. En este caso es necesario saber cuanto antes qué código se está generando mal.

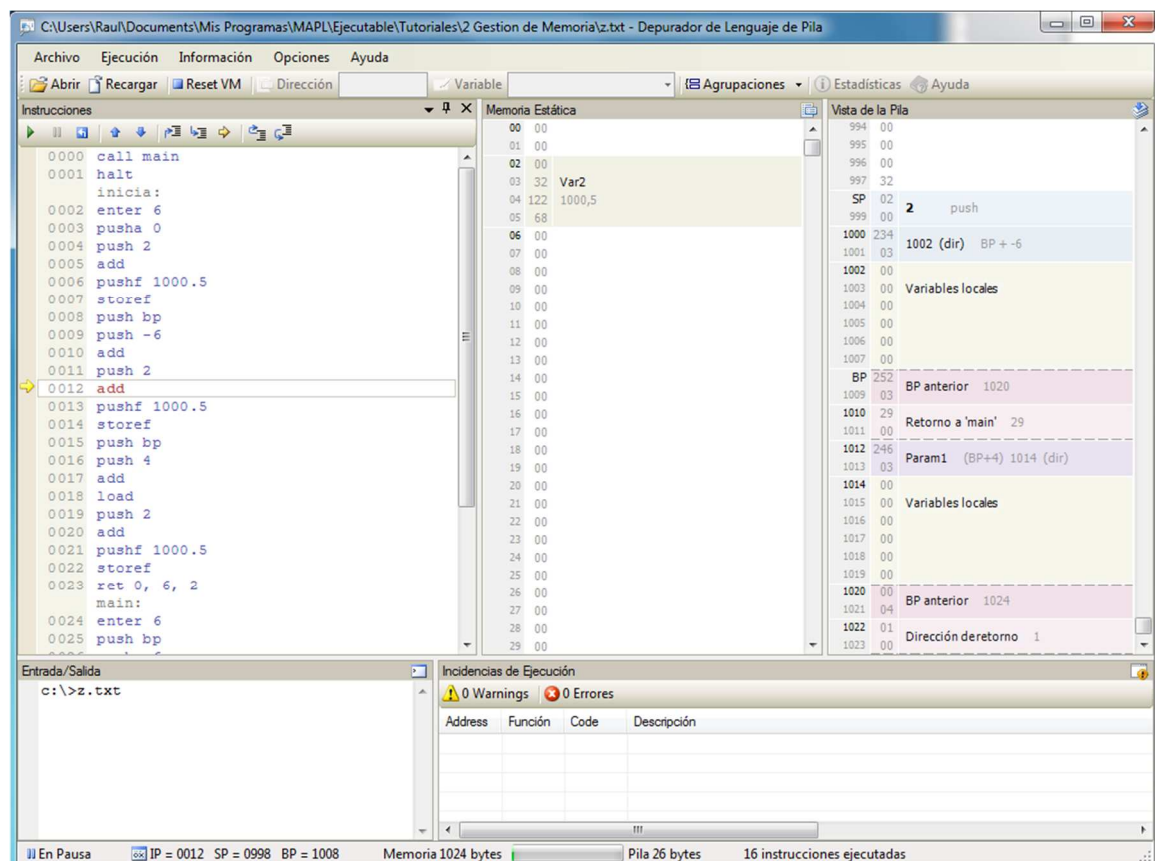
Para ello *GVM* ofrece directivas que, si se les suministra la posición de las sentencias, permiten asociar cada instrucción *MAPL* con el código de alto nivel a partir del cual fue generada. De esta manera es inmediato saber qué plantilla de código del compilador es la que está generando las instrucciones incorrectas.

En el ejemplo siguiente se puede ver que la instrucción *add* (en rojo) ha sido generada por la asignación al campo *suelto*.



The screenshot shows the MAPL debugger interface. The main window displays the assembly code for the function `inicia`. The memory view shows the state of memory, including the stack and global variables. The stack view shows the current stack frame, including the return address and local variables. The entry/output window shows the command `c:\>6 Paso por Referencia. Structs.txt`. The status bar at the bottom indicates the current instruction pointer (IP) is 0012, stack pointer (SP) is 0998, base pointer (BP) is 1008, and 16 instructions have been executed.

Véase el mismo programa sin la información de posición. Puede observarse que sería más difícil saber a qué parte del compilador pertenece la misma instrucción `add`:



The screenshot shows the MAPL debugger interface for a different program. The main window displays the assembly code for the function `main`. The memory view shows the state of memory, including the stack and global variables. The stack view shows the current stack frame, including the return address and local variables. The entry/output window shows the command `c:\>z.txt`. The status bar at the bottom indicates the current instruction pointer (IP) is 0012, stack pointer (SP) is 0998, base pointer (BP) is 1008, and 16 instructions have been executed.

Por todo ello se puede ver que, aunque guardar las posiciones de los símbolos en el AST no es obligatorio, es de una gran ayuda para la validación del compilador. En los siguientes apartados se mostrará cómo obtener fácilmente estas posiciones con *VGen*.

### Cálculo automático de las posiciones con *VGen*

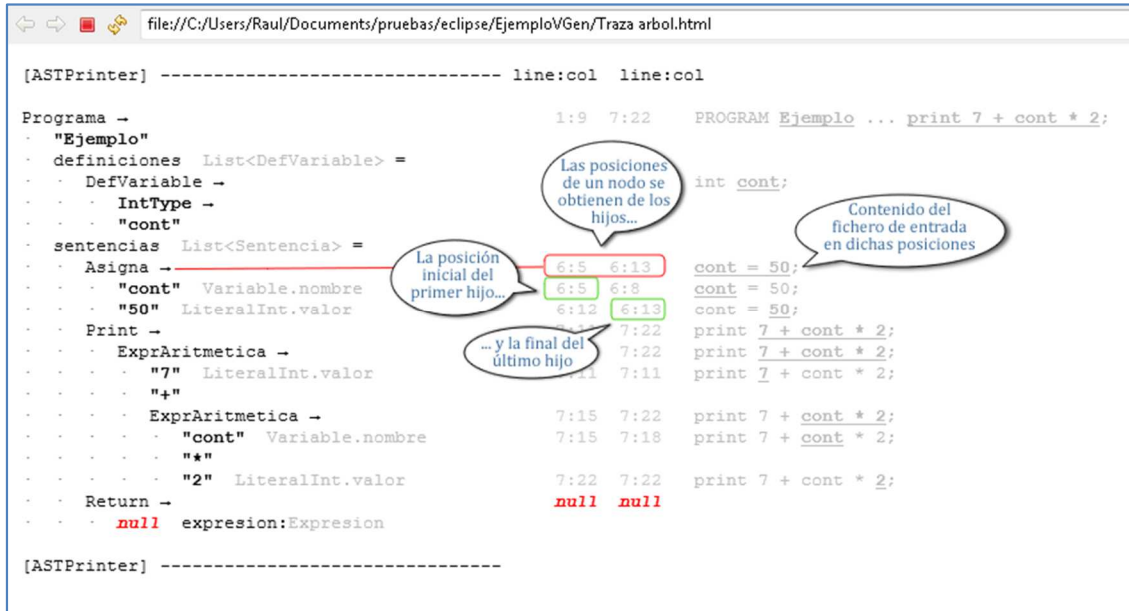
Al dejar objetos *Token* en *yylval* (*Yacc*), los símbolos terminales guardan su posición del fichero. *VGen* genera código en cada nodo del AST para que éstos obtengan también sus posiciones a partir de las posiciones de sus nodos hijos; la posición inicial será la de su primer hijo y su posición final será la del último<sup>6</sup>. Este código, que es el mismo en todas las clases, se ha sacado a una clase base de la que derivan todos los nodos llamada *AbstractTraceable*.

Para obtener esta funcionalidad no hay que hacer nada adicional en *Yacc*. De hecho, se puede utilizar un fichero *Yacc* no creado para *VGen* y simplemente dejando un *Token* en *yylval* aparecerán las posiciones en los nodos *automáticamente*.

Supóngase este ejemplo en el que se muestra una regla de *Yacc* que se limita a enlazar un nodo con sus hijos:

```
// Yacc
asignacion: expr '=' expr ';' { $$ = new Asigna($1, $3); }
```

*VGen* genera el código que obtiene automáticamente las posiciones de cada nodo. Se puede ver con *ASTPrinter* que los nodos tienen ya su posición inicial y final.



The screenshot shows the ASTPrinter tool interface. On the left, the AST structure is displayed, showing the hierarchy of nodes. On the right, the source code is shown with the corresponding AST nodes mapped to it. Annotations highlight the automatic assignment of positions to nodes:

- "La posición inicial del primer hijo..." points to the start position of the first child node.
- "... y la final del último hijo" points to the end position of the last child node.
- "Las posiciones de un nodo se obtienen de los hijos..." points to the start and end positions of a node.
- "Contenido del fichero de entrada en dichas posiciones" points to the source code snippet.

<sup>6</sup> En realidad, la posición inicial será la del primer hijo que tenga posición inicial. Del mismo modo la posición final será la del último hijo que tenga ésta.

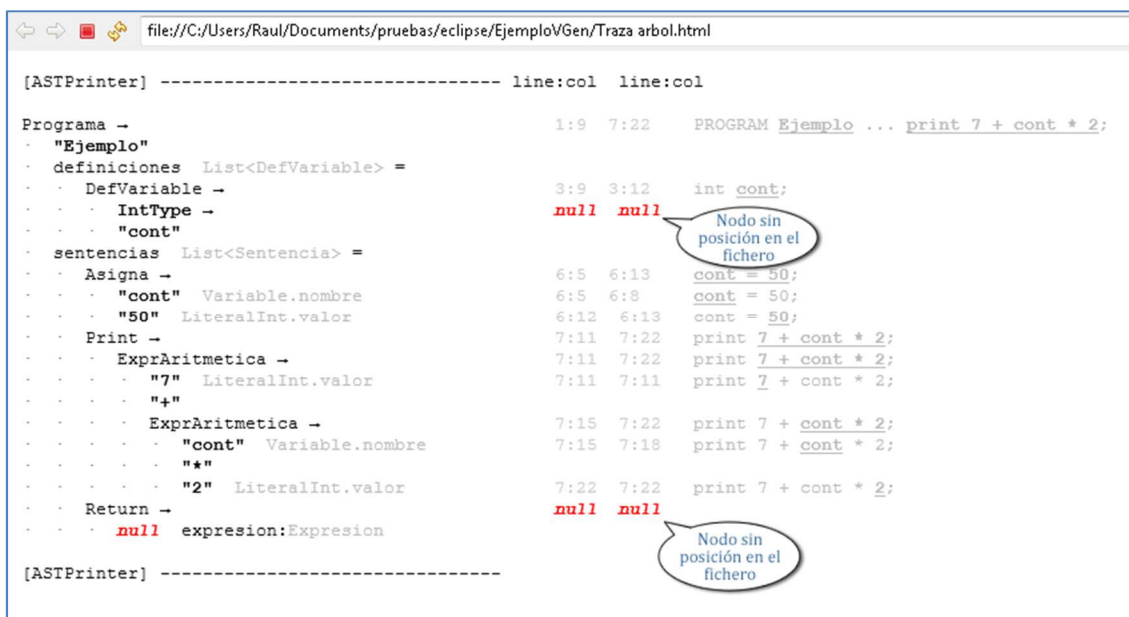
Este cálculo de posiciones se hace en el constructor de cada nodo invocando a un método llamado *searchForPositions*, el cual se hereda de la clase *AbstractTraceable*.

```
public class Asigna extends AbstractSentencia7 {  
  
    public Asigna(Expression left, Expression right) {  
        this.left = left;  
        this.right = right;  
  
        searchForPositions(left, right); // Obtener linea/columna de los hijos  
    }  
  
    // Resto de la clase omitida...  
}
```

Por tanto, partiendo de los *Token*, las posiciones de los nodos van ascendiendo en el árbol hasta tener las posiciones del nodo raíz del AST.

### Nodos sin posiciones

En algunas ocasiones poco frecuentes algún nodo puede quedarse sin posiciones asignadas. Si se quiere saber de manera rápida si algún tipo de nodo se ha quedado sin posiciones basta con abrir el fichero "*Traza arbol.html*" generado por *ASTPrinter*:



```
[ASTPrinter] ----- line:col  line:col  
  
Programa - 1:9  7:22  PROGRAM Ejemplo ... print 7 + cont * 2;  
- "Ejemplo"  
- definiciones List<DefVariable> =  
-   DefVariable - 3:9  3:12  int cont;  
-   - IntType - null null  
-   - "cont"  
- sentencias List<Sentencia> =  
-   Asigna - 6:5  6:13  cont = 50;  
-   - "cont" Variable.nombre 6:5  6:8  cont = 50;  
-   - "50" LiteralInt.valor 6:12  6:13  cont = 50;  
-   Print - 7:11  7:22  print 7 + cont * 2;  
-   - ExprAritmetica - 7:11  7:22  print 7 + cont * 2;  
-   - - "7" LiteralInt.valor 7:11  7:11  print 7 + cont * 2;  
-   - - "+"  
-   - - ExprAritmetica - 7:15  7:22  print 7 + cont * 2;  
-   - - - "cont" Variable.nombre 7:15  7:18  print 7 + cont * 2;  
-   - - - "*"   
-   - - - "2" LiteralInt.valor 7:22  7:22  print 7 + cont * 2;  
-   Return - null null  
-   - null expresion:Expression  
  
[ASTPrinter] -----
```

Se puede observar que se han quedado dos nodos sin posición: *IntType* y *Return*.

Dado que un nodo saca su posición de sus hijos, un nodo se quedará sin posición si no tiene atributos, es decir, si en su constructor no se le pasa ningún hijo. Puede comprobarse en los dos fragmentos de *Yacc* que eso es precisamente lo que ocurre con estos nodos:

```
return: 'RETURN' expr ';' { $$ = new Return($2); }  
      | 'RETURN' ';'      { $$ = new Return(null); } // Sin hijos
```

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

<sup>7</sup> La clase *AbstractSentencia* deriva de *AbstractTraceable*, por lo que *Asigna* hereda su implementación



Sin embargo, esto no suele ser un problema. Aunque algún nodo se quede sin posición, lo habitual será que esta situación no se propague a sus nodos padre al haberlas podido extraer éste de otros de sus hijos. De hecho, el algoritmo intenta agotar todas las posibilidades de encontrar posiciones de entre los hijos:

- Si alguno de los hijos es una lista, se extrae de ésta el primer/último hijo que tenga la posición buscada.
- Incluso aunque los atributos de los nodos sean *String* o tipos primitivos, se obtiene la posición de sus *Token* antes de que éstos sean convertidos a tipos primitivos y descartados (ver apartado *Conversiones Automáticas*).

Puede verse que, a pesar de que *IntType* no tiene posiciones, aun así, su nodo padre *DefVariable* ha obtenido sus posiciones a partir del *String* "cont".

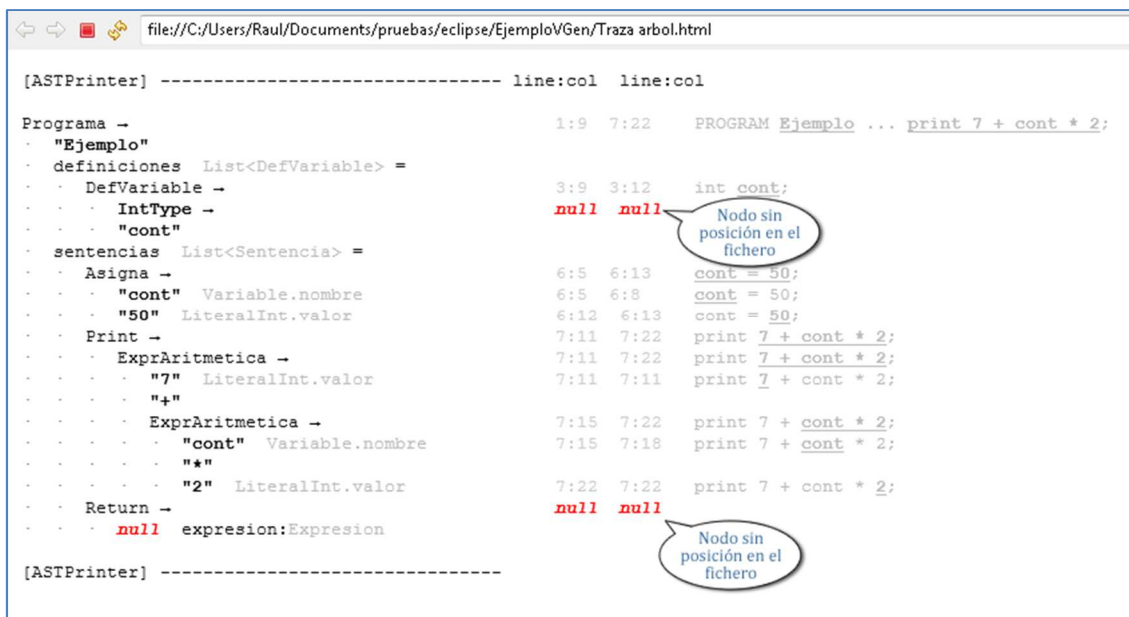
### Necesidad de la posición en los nodos

Hay que destacar que el objetivo *no es que todos* los nodos del AST tengan posiciones. Se trata de que la tengan al menos aquellos nodos de los cuales se vaya a necesitar su posición (para dar un mensaje de error o mostrar el código generado como se vio anteriormente). Por tanto, lo que se propone es dejar que el mecanismo de *VGen* trate de obtener todas las posiciones que pueda. Normalmente **serán suficientes** para el resto de las fases del compilador. Y, solo en el improbable caso de que se detecte algún nodo para el que se necesite la posición y no la tenga, se proceda a añadirla manualmente como se explicará en la siguiente sección.

### Asignación manual de posiciones

Como se acaba de comentar lo normal es que *VGen* obtenga todas las posiciones necesarias para las demás fases del compilador. Puede haber algún caso poco habitual en el que se necesite la posición de un nodo que no haya podido obtenerse automáticamente. Para estos casos *VGen* ofrece una forma sencilla de asignar las posiciones manualmente.

Uno de estos casos es la sentencia *Return* de la gramática del ejemplo anterior. Como se vio anteriormente no tiene posiciones en el AST:



```
[ASTPrinter] ----- line:col  line:col
Programa ->                                1:9  7:22  PROGRAM Ejemplo ... print 7 + cont * 2;
- "Ejemplo"
- definiciones List<DefVariable> =
-   DefVariable ->                        3:9  3:12  int cont;
-   - IntType ->                          null null
-   - "cont"
- sentencias List<Sentencia> =
-   Asigna ->                             6:5  6:13  cont = 50;
-   - "cont" Variable.nombre
-   - "50" LiteralInt.valor
-   Print ->                             7:11  7:22  print 7 + cont * 2;
-   - ExprAritmetica ->
-   -   "7" LiteralInt.valor
-   -   "+"
-   - ExprAritmetica ->
-   -   "cont" Variable.nombre
-   -   "*"
-   -   "2" LiteralInt.valor
-   Return ->
-   - null expresion:Expresion
[ASTPrinter] -----
```

En el caso del *Return*, sí es necesario obtener sus posiciones en el fichero para que a la hora de generar el código se pueda usar la directiva *#line* de *MAPL* y así poder indicar qué instrucciones se han generado para esta sentencia.

Ya se mostró que la razón por la que el *Return* no tiene las posiciones del fichero es porque, al no tener una expresión (es opcional), ha entrado por la segunda regla y se ha creado sin nodo hijo. Por tanto, no ha podido extraerse información de posición:

```
return: 'RETURN' expr ';' { $$ = new Return($2); }  
      | 'RETURN' ';'      { $$ = new Return(null); } // Sin hijos
```

En este caso basta con indicarle al nodo del AST, mediante el método *setPositions*, otro símbolo del cual se quiera obtener su posición del fichero (independientemente de que sea hijo o no<sup>8</sup>).

```
return: 'RETURN' expr ';' { $$ = new Return($2); }  
      | 'RETURN' ';'      { $$ = new Return(null).setPositions($1); }
```

En el caso anterior, lo que haría *setPositions* sería poner en el nodo *Return* las posiciones del token *RETURN*:



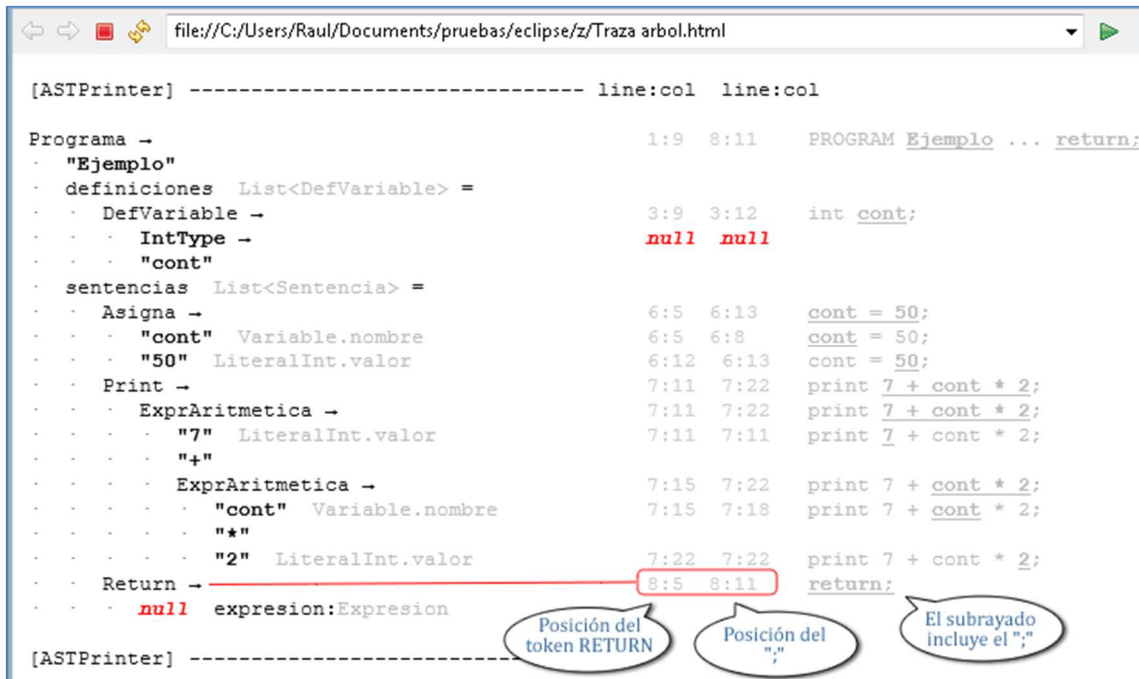
```
[ASTPrinter] ----- line:col  line:col  
  
Programa →                               1:9  8:10  PROGRAM Ejemplo ... return;  
- "Ejemplo"  
- definiciones List<DefVariable> =  
-   DefVariable →                               3:9  3:12  int cont;  
-   - IntType →                               null  null  
-   - "cont"  
-   sentencias List<Sentencia> =  
-   - Asigna →                               6:5  6:13  cont = 50;  
-   -   "cont" Variable.nombre               6:5  6:8   cont = 50;  
-   -   "50" LiteralInt.valor                 6:12  6:13  cont = 50;  
-   - Print →                               7:11  7:22  print 7 + cont * 2;  
-   -   ExprAritmetica →                     7:11  7:22  print 7 + cont * 2;  
-   -   - "7" LiteralInt.valor                 7:11  7:11  print 7 + cont * 2;  
-   -   - "+"  
-   -   - ExprAritmetica →                     7:15  7:22  print 7 + cont * 2;  
-   -   -   "cont" Variable.nombre             7:15  7:18  print 7 + cont * 2;  
-   -   -   "*"   
-   -   -   "2" LiteralInt.valor                 7:22  7:22  print 7 + cont * 2;  
-   - Return →                               8:5  8:10  return;  
-   - null expresion:Expresion  
  
[ASTPrinter] -----
```

Incluso se puede aprovechar para dar una posición final más exacta que incluya hasta el punto y coma final del *return*:

```
return: 'RETURN' expr ';' { $$ = new Return($2); }  
      | 'RETURN' ';'      { $$ = new Return(null).setPositions($1, $2); }
```

<sup>8</sup> En realidad, el mecanismo automático lo único que hace es llamar a *setPositions* con los hijos. Cuando no hay hijos lo que hay que hacer es llamarlo explícitamente con otros símbolos de los que extraer sus posiciones (no se utiliza ninguna otra información de ellos).

De esta manera cogerá la posición inicial del primer símbolo y la final del último – el punto y coma<sup>9</sup>:



```
[ASTPrinter] ----- line:col  line:col
Programa →                               1:9  8:11  PROGRAM Ejemplo ... return;
- "Ejemplo"
- definiciones List<DefVariable> =
-   DefVariable →                        3:9  3:12  int cont;
-   - IntType →                          null null
-   - "cont"
-   sentencias List<Sentencia> =
-   - Asigna →                            6:5  6:13  cont = 50;
-   -   "cont" Variable.nombre           6:5  6:8   cont = 50;
-   -   "50" LiteralInt.valor            6:12  6:13  cont = 50;
-   -   Print →                          7:11  7:22  print 7 + cont * 2;
-   -   - ExprAritmetica →               7:11  7:22  print 7 + cont * 2;
-   -   -   "7" LiteralInt.valor         7:11  7:11  print 7 + cont * 2;
-   -   -   "+"
-   -   -   ExprAritmetica →             7:15  7:22  print 7 + cont * 2;
-   -   -   - "cont" Variable.nombre     7:15  7:18  print 7 + cont * 2;
-   -   -   - "*"
-   -   -   - "2" LiteralInt.valor       7:22  7:22  print 7 + cont * 2;
-   - Return →                          8:5  8:11  return;
-   - null expresion:Expresion

[ASTPrinter] -----
```

Nótese que esto último de añadir la posición del punto y coma no es realmente necesario, ya que la posición que se obtiene únicamente con \$1 sería suficientemente precisa.

### Asignación manual a los tipos

El otro nodo que en el ejemplo anterior había quedado sin posiciones era *IntType*. Sin embargo, a diferencia del *Return*, la posición de este nodo no es necesaria. Si se quiere indicar algún error en la definición de la variable puede observarse que su padre, el nodo *DefVariable*, sí tiene posiciones. Por tanto, en este caso no se necesitaría hacer nada más<sup>10</sup>. En general se recomienda dejar el fichero de *Yacc* lo más independiente posible de *VGen*<sup>11</sup>.

Solo como ejemplo del uso de *VGen*, se muestra la forma de obtener las posiciones en estos nodos (que sería idéntica a la del *Return*):

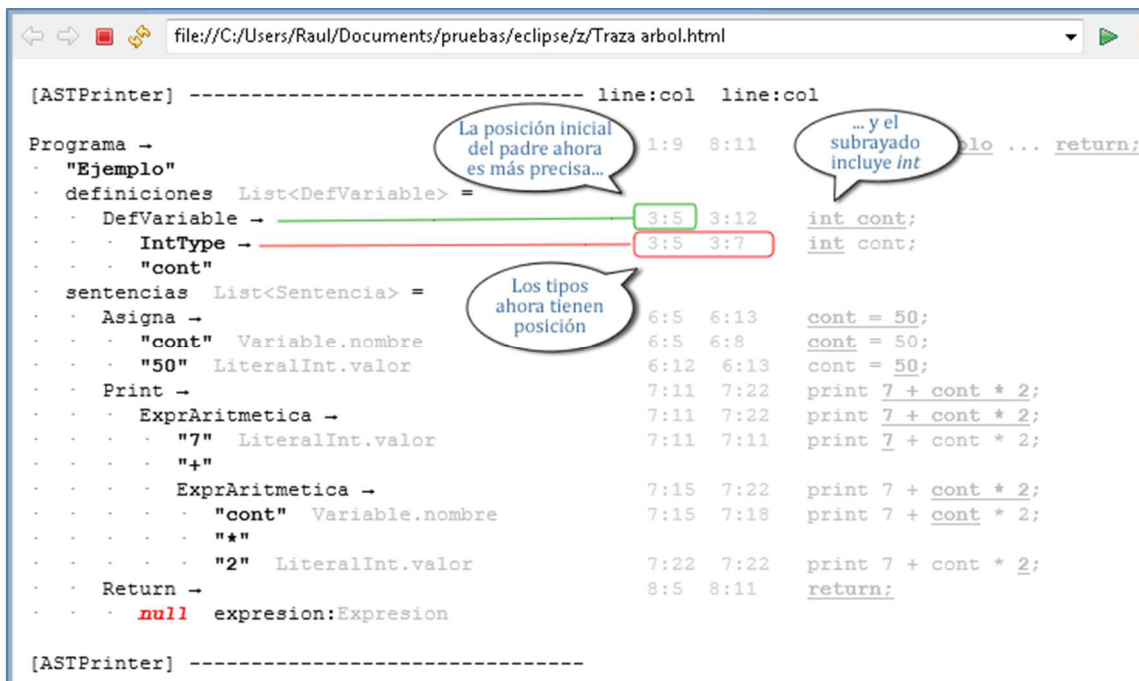
```
tipo: 'INT' { $$ = new IntType().setPositions($1); }
      | 'REAL' { $$ = new RealType().setPositions($1); }
```

<sup>9</sup> La estructura del nodo *Return* no se modifica; sigue sin tener hijos y solo se asignan sus posiciones.

<sup>10</sup> En el lenguaje del alumno posiblemente el único caso en el que haya que usar el mecanismo manual de asignación de posiciones (*setPositions*) sea en el caso del *Return*.

<sup>11</sup> De hecho, la mayoría de los ficheros *Yacc* podría usarse con *VGen* sin tener que modificarlos.

El nuevo AST quedaría así:



```

[ASTPrinter] ----- line:col  line:col
Programa →
- "Ejemplo"
- definiciones List<DefVariable> =
-   DefVariable → 3:5 3:12  int cont;
-   IntType → 3:5 3:7  int cont;
-   "cont"
- sentencias List<Sentencia> =
-   Asigna →
-   - "cont" Variable.nombre
-   - "50" LiteralInt.valor
-   Print →
-   - ExprAritmetica →
-   -   "7" LiteralInt.valor
-   -   "+"
-   -   ExprAritmetica →
-   -   - "cont" Variable.nombre
-   -   - "*"
-   -   - "2" LiteralInt.valor
-   Return →
-   - null expresion:Expresion

[ASTPrinter] -----
  
```

Nótese como *DefVariable*, aunque ya tenía posiciones, ahora mejora su precisión al tener dos hijos con posiciones.

### Mejora de la precisión de las posiciones

Supóngase el siguiente programa de entrada:

```

DATA
  int total;
CODE
  print total;
  
```

El fragmento del AST que corresponde con el *print* y muestra sus posiciones sería el siguiente:

```

-   Print → 10:11 10:15  print total;
-   - "total" Variable.nombre 10:11 10:15  print total;
  
```

Nótese que las posiciones del *print* no incluyen el propio token *PRINT* ni el punto y coma. Son las mismas que el lexema *"total"* ya que es su único hijo:

```

sentencia: 'PRINT' expr ';' { $$ = new Print($2); }
  
```

Como ejemplo del uso de otras características de posicionamiento de *VGen*, y no porque sea necesario en este ejemplo, se muestra a continuación como mejorar las posiciones anteriores.

Lo primero que podría hacerse sería incluir la posición del token *PRINT*. Para ello se indica que la posición inicial (*startPosition*) se tome de dicho token en vez de usar el hijo del nodo:

```

sentencia: 'PRINT' expr ';' { $$ = new Print($2).startPosition($1); }
  
```

Nótese como ahora la posición de inicio se corresponde con el inicio real de la sentencia *print*:

```

-   Print → 10:5 10:15  print total;
-   - "total" Variable.nombre 10:11 10:15  print total;
  
```

En vez de mejorar la posición de inicio de la sentencia, se podría querer mejorar la posición final incluyendo el punto y coma (por ejemplo, para que *GVM* muestre el código generado después del punto y coma de la sentencia y no antes de este). Para ello bastaría indicar que se tome la posición final del punto y coma en vez de tomarla del hijo:

```
sentencia: 'PRINT' expr ';' { $$ = new Print($2).endPosition($3); }
```

La posición final ahora incluye el punto y coma que finaliza el *print*:

```
. . Print → 10:11 10:16 print total;  
. . . "total" Variable.nombre 10:11 10:15 print total;
```

Si se quisiera hacer ambas cosas a la vez y conseguir las posiciones precisas tanto del inicio como del final de la sentencia podría hacerse de la siguiente manera:

```
'PRINT' expr ';' { $$ = new Print($2).startPosition($1).endPosition($3); }
```

O bien usar el método *setPositions*, que combina los dos métodos anteriores:

```
sentencia: 'PRINT' expr ';' { $$ = new Print($2).setPositions($1,$3); }
```

En los dos casos anteriores el resultado sería el mismo:

```
. . Print → 10:5 10:16 print total;  
. . . "total" Variable.nombre 10:11 10:15 print total;
```

Una vez más, incidir en que lo anterior es un ejemplo de uso de *VGen* para aquellos casos esporádicos en los que se necesite mayor precisión en las posiciones. Pero el uso adecuado de *VGen* no consistirá en polucionar el fichero de especificación de *Yacc* con código que intente establecer las posiciones precisas de inicio y final de todos los nodos del AST. El objetivo de *VGen* es que se tenga toda la información realmente necesaria en el árbol *de manera automática sin necesidad de incluir ninguna llamada a setPositions en Yacc*. Y, de necesitarlo, que haya, a lo sumo, una o dos llamadas a dicho método y solamente cuando en fases posteriores (semántico o generación de código) se eche en falta alguna posición.

## Apéndice I. Opciones en línea de comando de VGen

VGen ofrece las siguientes opciones en línea de comandos. Sin embargo, *lo habitual* será invocarlo sin ninguna opción. Se incluyen únicamente para posibles usos específicos que se le quisiera dar al código generado. En general, se recomienda no usar estas opciones.

Opción	Descripción
<b>-object</b>	Mediante esta opción se le indica a VGen que no genere los constructores con parámetros <i>Object</i> . Por tanto, habrá que hacer las conversiones de los atributos \$n (que son de tipo <i>Object</i> ) en Yacc.
<b>-public</b>	Con esta opción los atributos de los nodos se generan como atributos <i>Java</i> con visibilidad <i>public</i> . Por tanto, no se generan <i>getter</i> y <i>setters</i> de cada uno.
<b>-null</b>	Por defecto, a la hora de generar la clase <i>DefaultVisitor</i> (implementación base del <i>Visitor</i> de la que derivan todos los demás), VGen genera código que comprueba si un hijo es <i>null</i> antes de recorrerlo y evitar así un <i>NullPointerException</i> . Esta opción elimina dicho código. Sin embargo, no se recomienda usar esta opción a menos de que se esté seguro que ningún AST tendrá hijos a <i>null</i> .
<b>-token</b>	Usar esta opción si en Yacc no se usa la clase <i>Token</i> con <i>yylval</i> . Se elimina el código de las funcionalidades adicionales: la gestión de posiciones (línea y columna) y las conversiones automáticas de los objetos <i>Token</i> a los tipos primitivos de <i>Java</i> . No es necesario usar esta opción aunque no se use la clase <i>Token</i> en <i>yylval</i> . El código funcionará correctamente igualmente, ya que el código generado lo detectará y no hará nada. Esta opción es simplemente para ni siquiera generar dicho código que no va a hacer nada.

## Apéndice II. Especificación de VGen

### Léxico

```
%ignorecase // No se diferencian mayúsculas y minúsculas

CATEGORIES { return Parser.CATEGORIES; }
NODES      { return Parser.NODES; }
"->"|"="    { return Parser.FLECHA; }

[a-zA-ZñÑ][a-zA-Z0-9_ñÑ]* { return Parser.IDENT; }

[: , ; *]    { return yycharat(0); }

"/" "([ ^ * ] | \\. + [ ^ * / ] ) * \\. + "/" { /* comentario de varias líneas */ }
"/" "/" . * { /* comentario de una línea */ }
```



## Sintaxis

```

program: 'CATEGORIES' identsOpt 'NODES' definicionNodosOpt
        | 'NODES' definicionNodosOpt
        | definicionNodosOpt

identsOpt: idents |  $\lambda$ 

idents: idents ',' 'IDENT'
        | 'IDENT'

definicionNodosOpt: definicionNodosOpt nodo |  $\lambda$ 

nodo: 'IDENT' categoriasOpt 'FLECHA' atributosOpt ';'

categoriasOpt: ':' idents |  $\lambda$ 

atributosOpt: atributosOpt atributo |  $\lambda$ 

atributo: 'IDENT' ':' 'IDENT' multivaluadoOpt
          | 'IDENT' multivaluadoOpt

multivaluadoOpt: '*' |  $\lambda$ 

```

## Validaciones semánticas. Gramática Atribuida

Nota: En general, la comparación de cadenas no distingue entre mayúsculas y minúsculas (si aparece una categoría *expr* y otra *Expr* se dará error por nombre repetido). Sin embargo, como situación particular, a la hora de comparar con las palabras reservadas de Java sí se considerarán como distintas las mayúsculas y las minúsculas (puede haber un nodo *Return* aunque *return* sea una palabra reservada).

Símbolo	Predicados	Reglas Semánticas
<b>Programa:</b> AST $\rightarrow$ <i>categorías:</i> String* <i>nodos:</i> NodoAST*	$\text{duplicados}(\text{Programa.categorías}) == \emptyset$ $\text{Programa.categorías} \cap \text{generadas} == \emptyset$ $\text{Programa.categorías} \cap \text{Programa.abstractasGeneradas} == \emptyset$ $\text{Programa.categorías} \cap \text{primitivos} == \emptyset^{12}$ $ \text{Programa.nodos}  > 0$ $\text{duplicados}(\{ \text{Programa.nodos}_i.\text{nombre} \}) == \emptyset$	$\text{nodosi.programa} = \text{Programa}$ $\text{Programa.abstractasGeneradas} = \{ \text{"Abstract"} + \text{Programa.categorías}_i \}$
<b>NodoAST:</b> AST $\rightarrow$ <i>nombre:</i> String <i>categorías:</i> String* <i>atributos:</i> AtributoAST*	$\text{NodoAST.nombre} \notin \text{generadas}$ $\text{NodoAST.nombre} \notin \text{primitivos}$ <i>// No puede llamarse igual que una categoría ni su abstracta</i> $\text{NodoAST.nombre} \notin \text{NodoAST.programa.categorías}$ $\text{NodoAST.nombre} \notin \text{NodoAST.programa.abstractasGeneradas}$ $\text{duplicados}(\text{NodoAST.categorías}) == \emptyset$ $\text{NodoAST.categorías} \subset \text{NodoAST.programa.categorías}$ $\text{duplicados}(\{ \text{NodoAST.atributos}_i.\text{nombre} \}) == \emptyset$	
<b>AtributoAST:</b> AST $\rightarrow$ <i>nombre:</i> String <i>tipo:</i> String <i>multivaluado:</i> boolean	$\text{AtributoAST.nombre} \notin \text{reservadas}$ <i>// Java no permite listas de tipos primitivos (como List&lt;int&gt;)</i> $\text{AtributoAST.multivaluado} \Rightarrow \text{AtributoAST.tipo} \notin \text{primitivos}$	$\text{si } \text{AtributoAST.nombre} == \emptyset$ $\text{AtributoAST.nombre} = \text{minus}(\text{AtributoAST.tipo})$

<sup>12</sup> Los tipos primitivos no deben poder usarse como nombre de categoría o de nodo porque entonces no se podría saber si el tipo de un atributo se refiere al nodo o al tipo primitivo con el mismo nombre.

Los atributos añadidos a los símbolos del lenguaje son:

Nodo/Categoría	Atributo	Tipo Java	H/S	Descripción
Programa	abstractasGeneradas	List<String>	Sintetizado	Clases Abstractas que se van a generar (una por cada categoría/interface: AbstractExpr, AbstractSentencia, etc)
NodoAST	programa	Programa	Heredado	Raiz del AST para poder acceder a las categorías definidas

### Funciones auxiliares

*minus*(cadena) → Devuelve la cadena convertida a minúsculas.

*duplicados*(conjunto) → Devuelve otro conjunto con los elementos que en el original se encuentren repetidos (no distingue entre mayúsculas y minúsculas).

### Conjuntos auxiliares

*reservadas*. Conjunto de las palabras reservadas de Java (*if*, *public*, *return*, *int*, ...).

*primitivos*. Conjunto de los tipos primitivos de Java (*int*, *float*, *double*, ...). Es un subconjunto del conjunto *reservadas* anterior.

*generadas* = { *AST*, *Token*, *Traceable*, *AbstractTraceable*, *Position* }. Conjunto de clases auxiliares que genera VGen. No deben usarse estos nombres en los nodos o categorías ya que las clases generadas se sobrescribirían.