Capítulo 6. Generación de Código: Gestión de Memoria

1 Objetivo de esta Fase

Cuando un programador escribe la definición de una variable (como el "*int cont;*" del ejemplo siguiente), lo que le está encargando al compilador es que busque por él una dirección de memoria que esté libre (al programador le es indiferente cuál sea ésta), de tal manera que a partir de ahora se referirá a dicha dirección de memoria por un nombre en vez de por un número difícil de recordar. De la misma manera, cuando el programador realiza posteriormente la asignación a la variable *cont*, le está encargando al compilador que guarde un 5 en la zona de memoria a la que le haya asignado dicho nombre[1].

```
DATA
        int cont;
CODE
        cont = 5;
```

Esta es la labor de esta fase del compilador; asignar una dirección libre a cada variable del programa. De esta manera, en la fase del siguiente capítulo (*Selección de Instrucciones*), cada referencia a una variable será sustituida por la dirección a la que representa[2]. Supóngase una entrada como:

```
DATA
        float f1;
        int i1;
        float f2;
        int i2;
CODE
```

Para este programa, esta fase modificaría el AST recibido del semántico asignando una dirección a cada nodo *VarDefinition* (direcciones 0, 4, 6 y 10 respectivamente). El AST, una vez acabada esta fase, quedaría así:

```
Programa →                                  2:11  5:10   float f1; ... int i2;
·  definiciones  List<VarDefinition> =
·  ·  VarDefinition →                       2:11  2:12   float f1;
·  ·  |  RealType →                         null  null
·  ·  |  "f1"
·  ·  |  0  int
·  ·  VarDefinition →                       3:9  3:10    int i1;
·  ·  |  IntType →                          null  null
·  ·  |  "i1"
·  ·  |  4  int
·  ·  VarDefinition →                       4:11  4:12   float f2;
·  ·  |  RealType →                         null  null
·  ·  |  "f2"
·  ·  |  6  int
·  ·  VarDefinition →                       5:9  5:10    int i2;
·  ·  |  IntType →                          null  null
·  ·  |  "i2"
·  ·  |  10  int
·  sentencias  List<Sentencia> =
```

[1] Al programar en código máquina, en vez de usar variables, había que indicar expresamente en qué dirección se quería escribir.

[2] En realidad, la fase de Gestión de Memoria incluye muchos más aspectos que el asignar direcciones (como por ejemplo determinar la representación de cada símbolo en memoria). Aquí se ha tratado exclusivamente lo necesario para este lenguaje.

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

Como en toda fase, se comenzará extrayendo los requisitos. Se incluyen a continuación los requisitos de *Descripción del Lenguaje.pdf* que corresponden a esta fase:

- En la sección *DATA* se realizan las definiciones de variables (no puede haber definiciones en la sección *CODE*).
- Las variables solo pueden ser de tipo entero o tipo real (*float*). Las variables de tipo entero ocupan 2 bytes y las reales 4.

El primer requisito supone que no habrá variables locales y por tanto todas las direcciones son absolutas (no habrá direccionamiento relativo).

3.1.2 Metalenguaje Elegido

En esta fase se añade más información a las definiciones de variables: su dirección. Por tanto, se utilizarán de nuevo las Gramáticas Atribuidas (GAt), ya que permiten indicar qué información se va a añadir en forma de atributos (las direcciones) y cómo obtienen estos su valor mediante reglas semánticas (cómo asignar la dirección de cada variable).

3.1.3 Especificación en el Metalenguaje

A continuación, se incluye el contenido de la gramática abstracta que se puede encontrar en el documento "Gestión de Memoria. *Gramática Atribuida.pdf*"

Símbolo	Predicados	Reglas Semánticas
program → *definitions*:varDefinition* *sentences*:sentence*		$program.definitions_i.address = \sum program.definitions_j.type.size \mid 0 \le j < i$
varDefinition → *type*:type *name*:String		
intType:type → λ		intType.size = 2
realType:type → λ		realType.size = 4
print:sentence → *expression*:expression		
assignment:sentence → *left*:expression *right*:expression		
arithmeticExpression:expression → *left*:expresión *operator*:String *right*:expression		
variable:expression → *name*:String		
intConstant:expression → *valor*:String		
realConstant:expression → *valor*:String		

Tabla de atributos

Categoría	Nombre	Tipo Java	H/S	Descripción
VarDefinition	address	int	Heredado	Dirección de memoria donde se ubicará el valor de la variable
type	size	int	Sintetizado	Número de bytes que ocupan los valores de dicho tipo

La primera regla semántica simplemente indica que la dirección de una variable es la suma de los tamaños de las variables anteriores.

Nótese que no se ha definido ningún predicado. Esto es lógico ya que con el semántico acabaron las fases de análisis y, por tanto, ya no hay errores que detectar. Simplemente se usa la parte de las gramáticas atribuidas que nos permite añadir más información al árbol (atributos y reglas semánticas).

3.2 Implementación

3.2.1 Dirección de las variables

El primer paso es, siguiendo la tabla de atributos de la gramática atribuida anterior, añadir una propiedad *dirección* a las definiciones de variables:

```
public class VarDefinition extends AbstractAST {

    private int address;

    public int getAddress() {
        return address;
    }

    public void setAddress(int address) {
        this.address = address;
    }

    // El resto igual...

}
```

3.2.2 Tamaño de los tipos

Para poder asignar una dirección libre se necesita saber cuánta memoria va ocupando cada variable. Como el tamaño de una variable depende de su tipo, modificamos éstos para que nos indiquen su tamaño (size):

```
public interface Type extends AST {
    public int getSize();
}
```

Siguiendo literalmente la forma de implementar los atributos de una Gramática Atribuida habría que añadir también el método *setSize* al interface *Tipo*. Aunque podría hacerse, se ha optado por simplificar la implementación ya que en este caso no es realmente necesario.

A continuación, hay que redefinir el método *getSize* en cada uno de los tipos para que devuelvan su tamaño:

```
public class IntType extends AbstractType {


    public int getSize() {
        return 2;
    }
       // El resto igual
}
```

```
public class ReaType extends AbstractType {


    public int getSize() {
        return 4;
    }
       // El resto igual
}
```

3.2.3 Asignación de las direcciones

Para implementar este *visitor*, en el proyecto ya existe un fichero "*codegeneration/MemoryAllocation.java*" que sugiere donde colocar esta nueva clase. En dicho fichero se tiene un esqueleto al que solo le falta añadir los métodos *visit* para los nodos de nuestra gramática. De nuevo se puede optar, o bien por escribirlos a mano, o bien copiarlos del fichero "*_PlantillaParaVisitor.txt*" generado por *VGen* en la carpeta *visitor*.

Una vez hecho esto hay que plantearse sobre qué nodos de AST se debe actuar en esta fase. En este caso, solo hay que implementar el método *visit* de *Programa*. Esto es así porque el atributo *dirección* de *VarDefinition* es heredado (y por tanto debe asignarlo el padre - en este caso el nodo Programa).

Al implementar dicho método *visit*, se aprovecha para hacer una pequeña optimización consistente en recorrer sólo los hijos *VarDefinition*, ya que en este lenguaje no pueden aparecer definiciones de variables locales entre las sentencias.

```
public class MemoryAllocation extends DefaultVisitor {

    // class Programa { List<DefVariable> definiciones; List<Sentencia> sentencias; }
    public Object visit(Program node, Object param) {

        int currentAddress = 0;

        for (VarDefinition child : node.getDefinitions()) {
            child.setAddress(currentAddress);
            currentAddress += child.getType().getSize();
        }
        return null;
    }

}
```

4 Ejecución

Se modifica el fichero *source.txt* para comprobar que el compilador asigna adecuadamente las direcciones de memoria:

```
DATA
        float f1;
        int i1;
        float f2;
        int i2;

CODE
```

Al ejecutar la clase *Main,* se obtendrá un AST con las direcciones asignadas a las variables. Sin embargo, en *'AST.html'* no aparecerán dichas direcciones. Aunque no es obligatorio, es muy sencillo modificar la traza para incluir esta información.

4.1 Ampliar la traza (opcional)

La clase *ASTPrinter* (generada por VGen) se limita a imprimir en cada nodo quiénes son sus hijos ignorando otros atributos adicionales que pudiera tener dicho nodo (es decir, sólo imprime aquella información declarada en la gramática abstracta). Se hace de esta manera para mantener la traza del AST lo más limpia posible y así facilitar su revisión.

Las direcciones de las variables son el único caso en el que sería realmente recomendable aumentar la traza ya que no hay otra forma de validar los valores asignados. Para añadir las direcciones a la traza, basta con añadir directamente en la clase *ASTPrinter* la línea que le falta al método *visit* de *VarDefinition* (línea resaltada con fondo amarillo)[3]:

```
//  class VarDefinition { Type type;  String name; }
public Object visit(VarDefinition node, Object param) {
    int indent = ((Integer) param).intValue();

    printName(indent, "VarDefinition", node, false);

    visit(indent + 1, "type", "Type", node.getType());
    print(indent + 1, "name", "String", node.getName());

    // ESTA ES LA LINEA NUEVA <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
    print(indent + 1, "address", "int", node.getAddress());

    return null;
}
```

Si se vuelve a ejecutar la clase *Main* se observará que *'AST.html'* está actualizada con las direcciones de las variables:

[3] Esta sería la misma clase *ASTPrinter* que se obtendría si se añade en la gramática abstracta el atributo *"address:int"* en el nodo *VarDefinition* y se vuelve a pasar por *VGen*. Pero, por una línea, no merece la pena dicho proceso. Además, se recomienda usar con mucha precaución VGen después de que se hayan empezado a modificar las clases del AST para evitar sobreescribir el código que se haya añadido manualmente a las mismas (tipo, dirección, etc).

```
Programa →                                  2:11  5:10   float f1; ... int i2;
·  definiciones  List<VarDefinition> =
·  ·  VarDefinition →                         2:11  2:12   float f1;
·  ·  |  RealType →                          null  null
·  ·  |  "f1"
·  ·  |  0  int
·  ·  VarDefinition →                         3:9  3:10    int i1;
·  ·  |  IntType →                           null  null
·  ·  |  "i1"
·  ·  |  4  int
·  ·  VarDefinition →                         4:11  4:12   float f2;
·  ·  |  RealType →                          null  null
·  ·  |  "f2"
·  ·  |  6  int
·  ·  VarDefinition →                         5:9  5:10    int i2;
·  ·  |  IntType →                           null  null
·  ·  |  "i2"
·  ·  |  10  int
·  sentencias  List<Sentencia> =
```

Se puede ver que el *visitor* ha asignado correctamente las direcciones 0, 4, 6 y 10 respectivamente (en verde en el fragmento anterior).

5 Resumen de Cambios

Fichero	Acción	Descripción
Comprobador de Tipos. Gramática Atribuida.pdf	Creado	Gramática atribuida que indica cómo se asignan las direcciones a las variables
VarDefinition.java	Modificado	Se le añade la propiedad *dirección*
Type.java	Modificado	Se le añade la propiedad de solo lectura *size*
IntType.java	Modificado	Implementación de *getSize* para devolver 2
RealType.java	Modificado	Implementación de *getSize* para devolver 4
MemoryAllocation.java	Modificado	Asigna las direcciones a todas los *VarDefinition*
source.txt	Modificado	Define distintas variables para poder validar las direcciones asignadas
ASTPrinter.java	Modificado	Se añade el valor del atributo *address* (dirección) a la traza de los nodos *VarDefinition*