Capítulo 5. Semántico: Comprobación de Tipos

1 Objetivo de esta Fase

Tal y como se indicó en el capítulo anterior, la misión del análisis semántico es comprobar la validez de cada subárbol del AST. Si cada subárbol es válido, entonces el AST completo es válido.

Dichas validaciones se han repartido en dos fases. En el capítulo anterior se implementó la Fase de Identificación. En este capítulo se diseñará e implementará la fase de Comprobación de Tipos, es decir, la parte del semántico que pretende detectar errores en una operación antes de ejecutarla (en tiempo de compilación) y sin necesidad de conocer sobre qué valores concretos operará. Lo hace comprobando el tipo de cada expresión.

Por ejemplo, ante una entrada como:

```
DATA
	float f;
	int i;

CODE
	i = 2.0;	// Error
	f = 2;		// Error
	print 1 + 2.5; // Error
	2 = i;		// Error, pero no de tipo
```

El compilador debería indicar los siguientes errores:

```
Error en Type Checking: [6:2] Los operandos deben ser del mismo tipo
Error en Type Checking: [7:2] Los operandos deben ser del mismo tipo
Error en Type Checking: [8:8] Los operandos deben ser del mismo tipo
Error en Type Checking: [9:2] Se requiere expresión modificable
```

Para poder comprobar si el tipo de cada expresión es válido, se necesita previamente haber calculado el tipo de todas ellas. Este proceso, que también realiza esta fase, es el que se conoce como *inferencia de tipos*.

Aunque el objetivo de esta fase sería únicamente comprobar los tipos, como se adelantó en el capítulo anterior, en esta fase se suelen incluir comprobaciones adicionales que no se corresponden con las tareas propias de un comprobador de tipos. Un ejemplo de estas validaciones es el último error del ejemplo anterior, en el que la causa del mismo no es que los tipos de ambos operandos de la asignación no sean compatibles (de hecho, son del mismo tipo); la causa es que no se puede modificar una constante (no es *modificable*).[1]

[1] Otro ejemplo de una validación que no pertenezca a las dos fases principales (aunque no se presenta en el lenguaje de este tutorial) sería la regla semántica de Java de que todo camino de un método con valor de retorno debe acabar en un *return*. El subárbol que formaría un nodo *DefiniciónDeMétodo* con sus hijos *Sentencia* sería inválido si entre éstas no hay los *return* necesarios (comprobación en la cual no intervienen tipos). Este sería un ejemplo de reglas de *control de flujo*, que, aunque podrían tener su propia fase aparte en el semántico, en la práctica se suelen incorporar en esta segunda.

Aunque quizás fuera más coherente realizarlas en una tercera fase con "el resto" de las validaciones, en la práctica se suelen incluir en esta segunda fase debido a que suelen ser muy pocas y por tanto no suele merecer la pena hacerlas aparte.

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

Se incluyen a continuación los requisitos de *Descripción del Lenguaje.pdf* que corresponden a la fase del Comprobador de Tipos del análisis semántico:

- La escritura (*print*) puede ser tanto de expresiones enteras como reales.
- Las operaciones aritméticas deberán ser entre operandos del mismo tipo, no habiendo conversiones implícitas. Esto quiere decir que, por ejemplo, no se puede sumar un entero con un real.
- En las asignaciones el tipo de la variable a asignar también deberá coincidir con el tipo del valor a ser asignado. Por tanto, no se puede asignar un valor real a una variable entera ni a la inversa.

3.1.2 Metalenguaje elegido

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

- La información que se necesita añadir al árbol (*atributos*). En este caso se añadirán atributos que indiquen en cada expresión su tipo y si es modificable o no.
- Cómo se obtiene el valor de lo atributos anteriores. Se utilizarán las *reglas semánticas* de la Gramática Atribuida para indicar como se infiere (calcula) el tipo de cada expresión y si es modificable o no.
- Qué condiciones deben cumplir los nodos del árbol. Mediante los *predicados* de la Gramática Atribuida se indicará qué valores de los obtenidos mediante las reglas semánticas son válidas en cada estructura del lenguaje.

3.1.3 Especificación en el Metalenguaje

En vez de comenzar la Gramática Atribuida desde cero, una alternativa es utilizar el esqueleto generado por *VGen*[2]. En la carpeta metalenguajes, éste generó un fichero '*Attribute Grammar.html*' en el cual ya aparece la gramática abstracta (primera columna), el lugar donde poner los predicados (segunda columna) y el lugar de las reglas semánticas (tercera columna). El conjunto de atributos se definiría en la segunda tabla del documento.

Una vez abierto dicho esqueleto en Word, se traducen los requisitos a la notación del metalenguaje rellenando las dos columnas de la derecha y la tabla de atributos. El resultado de este proceso está en "*Comprobador de Tipos. Gramática Atribuida.pdf*"

Símbolo	Predicados	Reglas Semánticas
program → *definitions*:varDefinition* *sentences*:sentence*		
varDefinition → *type*:type *name*:String		
intType:type → λ		
realType:type → λ		
print:sentence → *expression*:expression		
assignment:sentence → *left*:expresión *right*:expression	mismoTipo(left.type, right.type) left.modificable	
arithmeticExpression:expression → *left*:expresión *operator*:String *right*:expression	mismoTipo(left.type, right.type)	arithmeticExpression.type = left.type arithmeticExpression.modificable = false
variable:expression → *name*:String		variable.type = variable.definition.type variable.modificable = true
intConstant:expression → *valor*:String		intConstant.type = intType intConstant.modificable = false
realConstant:expression → *valor*:String		realConstant.type = realType realConstant.modificable = false

Tabla de Atributos:

Categoría	Nombre	Tipo Java	H/S	Descripción
expresion	type	Type	Sintetizado	Tipo de la expresión (operaciones que admite)
expresion	*modificable*[3]	boolean	Sintetizado	Indica si la expresión puede aparecer a la izquierda de una asignación

Funciones auxiliares:

mismoTipo*(tipoA, tipoB)* { tipoA == tipoB }

La función auxiliar *mismoTipo* realmente *no es necesaria* y hubiera sido suficiente con poner directamente como predicado "*left.tipo == right.tipo*" en lugar de usar dicha función. Sin embargo, se añade para posteriormente tener un ejemplo de cómo implementar funciones

[2] Si no se guardó la carpeta *metalenguajes* del capítulo 3, basta con volver a ejecutar *VGen* con el fichero *gramática.txt* que se creó en dicho capítulo.
[3] Nótese que el atributo *modificable* no hubiera sido necesario si se hubiera definido el primer hijo de la *asignación* como una *variable* (ver capítulo 3 donde se comenta el porqué de esta decisión)

auxiliares como las que seguramente se presentarán en la práctica del alumno (*tipoMayor*, *esPrimitivo*, *esConvertible*, etc.)

3.2 Implementación

3.2.1 Implementación de los atributos

Siguiendo la *Tabla de Atributos*, debemos añadir todas las propiedades a los nodos que les corresponda. En este caso hay que añadir *tipo* y *modificable* a las *expresiones*.

```
public interface Expression extends AST {
    public void setType(Type type);
    public Type getType();

    public void setModificable(boolean modificable);
    public boolean isModificable();
}
```

Ahora habría que añadir la implementación de los métodos anteriores en todos los nodos que implementen *Expression*. Sin embargo, si se usó *VGen*, éste ya incorporó una clase abstracta *AbstractExpressión.java* de la cual derivan todas las demás expresiones. Implementando los métodos de acceso en *AbstractExpression* no habrá que modificar el resto de los nodos *hijos*:

```
public abstract class AbstractExpression extends AbstractAST implements Expression {
    public void setType(Type type) {
        this.type = type;
    }

    public Type getType() {
        return type;
    }

    public void setModificable(boolean modificable) {
        this.modificable = modificable;
    }

    public boolean isModificable() {
        return modificable;
    }

    private Type type;
    private boolean modificable;
}
```

Si no se usó *VGen*, además de crear la clase anterior, habrá que modificar las clases de los nodos que implementan el interfaz *Expression* para que deriven de ella.

3.2.2 Implementación de Predicados y Reglas Semánticas

El analizador semántico se implementará mediante un *visitor* que recorrerá el árbol y comprobará que se cumplen todos los predicados de todos los nodos. Bastará un predicado que no se cumpla para que el programa no sea válido.

Para implementar este *visitor*, en el proyecto de eclipse ya existe un fichero "*semántico\ComprobaciónDeTipos.java*" que sugiere dónde colocar esta nueva clase. En dicho fichero se tiene un esqueleto a la que solo falta añadirle los métodos *visit* para los nodos de nuestra gramática. De nuevo, o bien se puede optar por escribirlos a mano, o bien copiarlos del fichero "visitor_*PlantillaParaVisitor.txt*" generado por *VGen*.

La *Gramática Atribuida*, además de indicar de manera precisa qué comprobaciones tiene que realizar el analizador semántico, tiene la ventaja de que supone una guía en el proceso de implementar cada uno de esos métodos *visit*.

Para cada nodo del AST se extrae de su fila de la gramática atribuida:

- La segunda columna (predicados) indica las condiciones que deben cumplirse en dicho nodo. Si no se cumplen hay que notificar un error.
- La tercera columna indica, para los atributos añadidos en el apartado anterior (tipo y modificable), qué valor hay que asignarles en dicho nodo[4].

El orden en el que se implementan estos elementos dentro de su método *visit* viene establecido por la plantilla para implementar gramáticas s-atribuidas vista en clase de teoría (lo primero sería recorrer los hijos y luego los predicados y las reglas).

> Sea *p* una regla con la forma "$padre \rightarrow hijo_1 \ldots hijo_n$"
> **visit**(Nodo *padre*) {
> Para cada $hijo_i$ { // Da igual el orden de visita de los hijos
> visitar($hijo_i$) // Al volver, el hijo tendrá ya asignados todos sus atributos
> }
> Comprobar los predicados asociados a la regla p (B(p))
> Asignar valor a los atributos de *padre* (R(p))
> }

Siguiendo esta plantilla, la fase de comprobación de tipos quedaría implementada así:

```
public class TypeChecking extends DefaultVisitor {

    public TypeChecking(ErrorManager errorManager) {
        this.errorManager = errorManager;
    }
```

[4] Esto es en el caso de los atributos sintetizados. Para atributos heredados la asignación habría que hacerla en el nodo padre. Ver la teoría para más información.

```
    //  class Assignment { Expression left;  Expression right; }
    public Object visit(Assignment node, Object param) {

        super.visit(node, param);
        predicado(mismoTipo(node.getLeft(), node.getRight()), "Los operandos deben ser del
mismo tipo", node);
        predicado(node.getLeft().isModificable(), "Se requiere expresión modificable", node
.getLeft());

        return null;
    }

    //  class ArithmeticExpression { Expression left;  String operator;  Expression right;
}
    public Object visit(ArithmeticExpression node, Object param) {

        super.visit(node, param);
        predicado(mismoTipo(node.getLeft(), node.getRight()), "Los operandos deben ser del
mismo tipo", node);

        node.setType(node.getLeft().getType());
        node.setModificable(false);

        return null;
    }

    //  class Variable { String name; }
    public Object visit(Variable node, Object param) {
        node.setType(node.getDefinicion().getType());
        node.setModificable(true);
        return null;
    }

    //  class IntConstant { String valor; }
    public Object visit(IntConstant node, Object param) {
        node.setType(new IntType());
        node.setModificable(false);
        return null;
    }

    //  class RealConstant { String valor; }
    public Object visit(RealConstant node, Object param) {
        node.setType(new RealType());
        node.setModificable(false);
        return null;
    }
```

```
    // ------------------------------------------------------
    // Funciones auxiliares

    private boolean mismoTipo(Expression a, Expression b) {
        return (a.getType().getClass() == b.getType().getClass());
    }

    private void predicado(boolean condition, String errorMessage, Position position) {
        if (!condition)
            errorManager.notify("Type Checking", errorMessage, position);
    }

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

    private ErrorManager errorManager;
}
```

4 Ejecución

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

```
DATA
        float f;
        int i;

CODE
        i = 2.0;        // Error
        f = 2;          // Error
        print 1 + 2.5; // Error
        2 = i;          // Error, pero no de tipo
```

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

```
Error en Type Checking: [6:2] Los operandos deben ser del mismo tipo
Error en Type Checking: [7:2] Los operandos deben ser del mismo tipo
Error en Type Checking: [8:8] Los operandos deben ser del mismo tipo
Error en Type Checking: [9:2] Se requiere expresión modificable
```

5 Resumen de Cambios

Fichero	Acción	Descripción
Comprobador de Tipos. Gramática Atribuida.pdf	Creado	Metalenguaje en el que se han descrito los requisitos de esta fase
Expression.java	Modificado	Se añaden los métodos de acceso para *tipo* y *modificable*
AbstractExpression.java	Modificado	Se añade la implementación de los métodos de acceso anteriores

TypeChecking.java	Modificado	Visitor que implementa la Gramática Atribuida. Comprueba los predicados de todos los nodos
source.txt	Modificado	Contienen errores que deben detectarse en esta fase