Universidad
Europea de Madrid
LAUREATE INTERNATIONAL UNIVERSITIES

ANÁLISIS SEMÁNTICO

VERIFICACIÓN DE TIPOS

© Todos los derechos de propiedad intelectual de esta obra pertenecen en exclusiva a la Universidad Europea de Madrid, S.L.U. Queda terminantemente prohibida la reproducción, puesta a disposición del público y en general cualquier otra forma de explotación de toda o parte de la misma.

La utilización no autorizada de esta obra, así como los perjuicios ocasionados en los derechos de propiedad intelectual e industrial de la Universidad Europea de Madrid, S.L.U., darán lugar al ejercicio de las acciones que legalmente le correspondan y, en su caso, a las responsabilidades que de dicho ejercicio se deriven.

Índice

Presentación 4
¿En qué consiste la verificación de tipos? 5
Sistema de tipos 7
Expresiones de tipo I 9
Expresiones de tipo II 11
Array (matriz) 11
Registro 11
Expresiones de tipo III 13
Unión 13
Expresiones de tipo IV 14
Puntero 14
Funciones 14
Clase 14
Equivalencia de tipos 16
Equivalencia estructural 16
Equivalencia nominal 16
Ejemplo I 18
Ejemplo II 20
Coerción de tipos 21
Resumen 22

© Universidad Europea de Madrid. Todos los derechos reservados.

Presentación

El objetivo de este tema es comprender las tareas necesarias para realizar la verificación de tipos como una tarea más de las que realiza el analizador semántico y consiste en los siguientes objetivos.

- Entender en qué consiste la comprobación de tipos.
- Comprender qué es un sistema de tipos.
- Identificar las distintas expresiones de tipos.
- Conocer el concepto de equivalencia de tipos en sus dos modalidades y sus implicaciones.
- Aprender a verificar tipos en un lenguaje cualquiera.
- Conocer los conceptos asociados a la conversión de tipos.

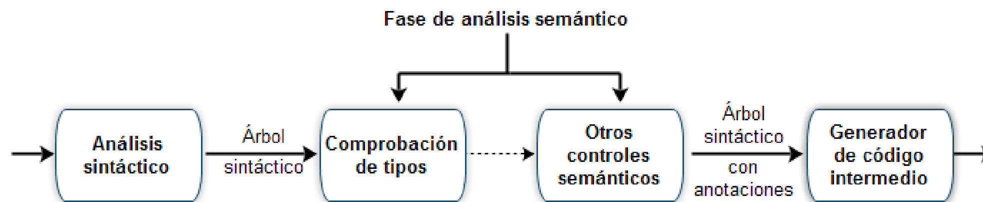
© Universidad Europea de Madrid. Todos los derechos reservados.

¿En qué consiste la verificación de tipos?

La fase de análisis semántico tiene que validar que un programa, además de ser sintácticamente correcto, es además coherente con su contexto y por tanto, tiene sentido, puesto que no vamos a sumar números con cadenas o los tipos que se utilizan se hacen de una forma adecuada, por citar alguno de los controles que podemos hacer.

Una de las tareas principales de un compilador en la fase de análisis semántico es el cálculo y mantenimiento de la información en tipos de datos **(inferencia de tipos),** y el uso de tal información para asegurar que cada parte de un programa tenga sentido bajo las reglas de tipo del lenguaje **(verificación de tipos).** Estas dos tareas por lo regular están estrechamente relacionadas, se realizan juntas, y se hace referencia a ellas simplemente como verificación de tipos (Louden, 2004). Como nos indica Louden (2004) **el tipo de datos** es un conjunto de valores junto con sus operaciones y se describe mediante una **expresión de tipo,** que es el nombre que tiene el tipo de datos, como por ejemplo **integer o float** y que además incluye otras estructuras más complejas como **arrays o registros.**

Con el objeto de no perder la visión global, es necesario el siguiente esquema conceptual, donde la comprobación de tipos es un control semántico más que hay que realizar siempre que realicemos operaciones.



Es importante comprender que dependiendo de que tareas haga el lenguaje que estamos construyendo con el compilador, la comprobación de tipos puede ir intercalada con otros controles semánticos, puesto que no dejan de ser acciones que el diseñador decide cuándo se realizan.

© Universidad Europea de Madrid. Todos los derechos reservados.

Tipos

Cada símbolo del lenguaje lleva asociado un tipo. El tipo que se asocia a cada símbolo lo utiliza el compilador para saber como debe tratarlo, así como determinar qué operaciones se pueden realizar sobre él. Por tanto un tipo de datos son un conjunto de valores que llevan implícitas ciertas operaciones. A modo de ejemplo tenemos los tipos enteros o de los reales, junto con las operaciones de suma (+), resta (-), multiplicación (*) o división (/) definidas sobre ellos.

Tarea de comprobación

Esta tarea de comprobación de tipos puede realizarse en tiempo de ejecución y se denomina comprobación de tipos **dinámica,** o bien en tiempo de compilación y se denomina comprobación de tipos **estática.**

© Universidad Europea de Madrid. Todos los derechos reservados.

Sistema de tipos

U n **sistema de tipos** son las reglas que hay que aplicar para asignar las **expresiones de tipos,** bien sean simples o complejas, a las distintas estructuras de un lenguaje. Por tanto, esa validación de las reglas que se definen para cada estructura del lenguaje es la comprobación de tipos.

Como nos indica Aho *et al* (2008) estas reglas pueden ser de dos tipos: de síntesis o de inferencia.

De todo esto se deduce que hay que realizar varias tareas:

1. **Gestión de declaraciones:** consiste en la inserción en la tabla de símbolos de los identificadores y sus tipos, teniendo en cuenta el ámbito en el que se han generado.
2. **Verificación de tipos:** a partir de la información existente en la tabla de símbolos, verifica que se cumplen las reglas asociadas a las distintas estructuras del código fuente.
3. **Inferencia de tipos:** en lenguajes que no requieren la definición de tipos, cuando hay sobrecarga de operadores o polimorfismo, hay que inferir el tipo de un dato o estructura en función de los distintos tipos de datos que intervienen

Como consecuencia del sistema de tipos que se implemente para el lenguaje en concreto, se habla de lenguajes fuertemente tipificados o débilmente tipificados:

Lenguajes fuertemente tipificados	Estos lenguajes hacen una implementación estricta de la verificación de tipos en tiempo de compilación, no permitiendo por tanto que una variable pueda usar datos de otro tipo sin que se realice una conversión antes de realizar una operación con los mismos. Esta implementación del sistema de tipos es propia de lenguajes imperativos y son ejemplo de este tipo de lenguajes Ada, Java, C, Haskell o Pascal, a los que se denomina lenguajes seguros respecto del tipo, puesto que es capaz de detectar los errores tanto en tiempo de compilación como de ejecución evitando la generación de resultados erróneos.
Lenguajes débilmente tipificados	Son lenguajes que tienen gran flexibilidad en el uso de los tipos de datos y que, por tanto, no realizan la verificación de tipos en tiempo de compilación, sino en tiempo de ejecución. En este caso, los tipos de lenguaje que tienen esta implementación son los declarativos y son ejemplo de este tipo de lenguajes Javascript, Scheme, Groovy, Perl y PHP.

Reglas de síntesis de tipos

Estas reglas construyen el tipo de una expresión a partir de los tipos de las subexpresiones y por tanto requieren que estas subexpresiones estén declaradas previamente. Son las más habituales, y son de la forma $E_1 \rightarrow T * E_2$, el tipo que tenga E_1 dependerá de los tipos que tengan T y E_2 y de lo que indique la regla respectiva para el producto cuando son del mismo tipo, por ejemplo enteros o cuando uno de ellos es real.

© Universidad Europea de Madrid. Todos los derechos reservados.

Reglas de inferencia de tipos

Determina el tipo de una construcción a partir de las formas en que se utiliza. Este mecanismo se utiliza en lenguajes que no requieren la declaración de tipos, como por ejemplo ML que comprueba los tipos pero no es necesario que se declaren los nombres de las variables.

Sobrecarga de operadores

Significa que un operador se puede utilizar con varios tipos de datos. En el caso del operador suma, "+" sirve tanto para números enteros como reales o incluso para concatenar cadenas.

Polimorfismo

Se refiere a las funciones y/o fragmentos de código que pueden ejecutarse con argumentos de distintos tipos.

Tipificados

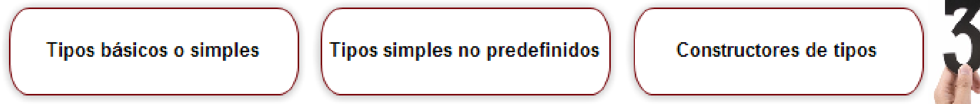
Dependiendo del autor se utiliza también la palabra **tipados.**

© Universidad Europea de Madrid. Todos los derechos reservados.

Expresiones de tipo I

Para desarrollar un sistema de tipos se utilizan las **expresiones de tipo** que es el mecanismo utilizado para definir los tipos de datos y las operaciones que se van a poder realizar sobre ellos.

Hay tres clases de expresiones de tipo:



1. **Tipos básicos o simples:** son tipos que se definen de forma explícita en el compilador, y podemos denominarlos como predefinidos en el lenguaje. En la siguiente tabla los vemos descritos.

En algunos lenguajes se les pueden añadir calificadores a los tipos básicos. Por ejemplo en C, podemos usar short o long y esto permite usar más bits para definir el valor. En el caso de short se utilizan 16 bits y en el de long 32 bits, por tanto para un entero en C, "int", podemos utilizar uno u otro (si la máquina lo permite).

2. **Tipos simples no predefinidos:** como su nombre indica son también tipos simples que no están predefinidos.
   a. **Tipos enumerados:** por ejemplo en Ada, type Dia is (lunes, martes, miércoles, jueves, viernes).
   b. **Tipos de subrango:** por ejemplo en Pascal, type número = 0...9.

En algunos lenguajes, las enumeraciones se definen en una declaración de tipo, y son verdaderos nuevos tipos. Dependiendo del lenguaje no se supone nada respecto de la forma en la que los valores se representan internamente, pero en C, los valores enumerados son todos tomados como nombres de enteros y se les asigna de forma automática los valores 0, 1, 2, etc., a menos que el usuario inicialice los valores a otros números enteros (Louden, 2002).

3. **Constructores de tipos:** a partir del conjunto de tipos simples se pueden crear nuevos tipos de datos utilizando los constructores de tipos como son los *arrays*, los registros, los punteros y las funciones por citar algunos.

Tipos básicos o simples

Tipo	Descripción	Expresión de tipo
Integer	Representación numérica con signo.	Entero
Real	Representación en punto flotante con precisión normal.	Real
Double	Representación en punto flotante con doble precisión.	Real
Boolean	Representación lógica.	Lógico
Char	Representación de carácter.	Caracter
String	Representación de cadena.	Cadena

© Universidad Europea de Madrid. Todos los derechos reservados.

Arrays

La traducción de *array* más ampliamente utilizada es la de matriz, aunque también se utiliza tabla. Cuando la matriz es de una dimensión se denomina vector. Usaremos matriz o *array* de forma indistinta.

© Universidad Europea de Madrid. Todos los derechos reservados.

Expresiones de tipo II

Como se ha mencionado, los constructores de tipo utilizan los tipos simples para crear nuevos tipos de datos. Los constructores de tipo que utilizan la mayoría de los lenguajes de programación son: **arrays, registros, uniones, punteros, funciones y clases.**

Array (matriz)

Este constructor de tipo necesita dos parámetros: el **índice y el componente.** El tipo del índice tiene que ser un entero y el tipo del componente puede ser cualquiera de los tipos básicos. La **operación asociada** con los valores de tipo matriz es la subindización.

Otro detalle a tener en cuenta es la **forma de almacenar los datos.** Normalmente las matrices usan almacenamiento contiguo, empezando por los índices más pequeños hacia los más grandes, con le objeto de permitir el uso de cálculos de desplazamiento automático durante la ejecución. **La cantidad de memoria necesaria** para una matriz será **n* tamaño**, donde n será el número de valores del tipo índice y tamaño será la cantidad de memoria necesaria para un valor del tipo componente (Louden, 2004).

Registro

Un constructor de tipo registro es una tabla de elementos, cada una con su nombre de campo y su tipo. La gran diferencia con las matrices es que en estas, los componentes son todos del mismo tipo mientras que en los registros pueden tener campos de distintos tipos. Otra diferencia es que para acceder a los distintos componentes en lugar de índices se usan nombres. Un ejemplo en C:

```
1. struct r
2.       {char c;
3.        double d;
4.        int i;
5.       };
```

La operación de acceso a un registro es mediante el operador **punto**. Si tenemos una variable de tipo registro denominada r. En el ejemplo, si escribimos r.c, accedemos a su primer componente.

© Universidad Europea de Madrid. Todos los derechos reservados.

Tipo de índice

El índice se suele definir entre corchetes [índice] o entre paréntesis (índice) dependiendo del lenguaje y se puede indicar el rango que tendrán sus valores, por ejemplo de 1 a 10 (1...10) o simplemente decir que tendrá 10 valores (empieza en el cero y termina en el nueve). En Java, por ejemplo se puede definir la longitud del *array* de forma dinámica.

- La forma de especificar un array en Pascal es: array [tipo_indice] of tipo_componente.

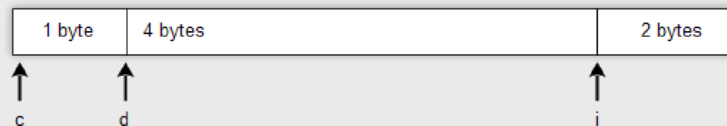
Subindización

La cual se utiliza para asignar valores a componentes o buscar valores en los componentes:

- x = y[5];
- y[7] = 85;

Registros

Los registros almacenan sus datos de forma secuencial, reservando un bloque de memoria para cada tipo de componente. En el ejemplo que hemos definido sería:



© Universidad Europea de Madrid. Todos los derechos reservados.

Expresiones de tipo III

Unión

Este constructor de tipos representa la unión de varios tipos, puesto que se forma a partir del conjunto de unión entre sus conjuntos de valores, y se corresponde con la operación de unión de conjuntos. Las uniones proporcionan una forma de manipular diferentes tipos de datos dentro de una sola área de almacenamiento.

No todos los lenguajes tienen este conjunto de tipos (Java no las tiene), el equivalente en Pascal son los ***"variant records"*** o registros variantes que utilizan un campo como discriminante del tipo que está en vigor en el registro variante. En C, un ejemplo sería:

```
1. union
2.          {char c;
3.           double d;
4.           int i;
5.          } u;
```

La variable u será lo suficientemente grande como para mantener el mayor de los tres tipos. Esto se denomina unión disjunta, puesto que cada valor se visualiza como un char, un real o un entero pero nunca como todos ellos

Cualquiera de los tres tipos puede ser asignado a u, pero el que se recupere será el último que se almacenó, por tanto es responsabilidad del programador elevar el registro del tipo que se almacenó. Se suele llevar una variable en paralelo, donde se almacena el tipo que tiene la unión, por ejemplo ***"utipo"*.**

La **operación de acceso** a una unión es mediante el operador **punto.** Si tenemos una variable de tipo unión denominada u. En el ejemplo, si escribimos u.c, accedemos a su primer componente.

Las uniones solo reservan almacenamiento para el tipo mayor, en el caso del ejemplo serán 4 bytes y estos componentes se almacenan en zonas superpuestas de la memoria.

Registro variante

Tanto C como Pascal con este mecanismo en las uniones provocan que el sistema de tipos sea inseguro, puesto que la gestión del tipo se hace por separado. Ada cuenta con un mecanismo de unión también llamado **registro variante,** como en Pascal, que obliga a utilizar el discriminante del tipo a la vez que se asigna su valor, y de esta forma el sistema de tipos es seguro. (Louden, 2002).

© Universidad Europea de Madrid. Todos los derechos reservados.

Expresiones de tipo IV

Puntero

Un constructor de tipo puntero contiene valores que son **apuntadores o punteros a valores de otro tipo,** es por tanto una dirección de memoria que debe mantener el tipo del identificador al que referencia.

Otra forma de expresarlo, un puntero es una variable que contiene la dirección de otra variable.

Ejemplo en C: int* PunteroAEntero.

Son necesarias dos operaciones, la de **referenciación** (en C es &) y la de **desreferenciación** (en C es *).

Los punteros necesitan **para almacenamiento,** el tamaño que para las **direcciones de memoria** utiliza la máquina sobre la cual se ejecutará el compilador.

Funciones

Un constructor de tipo función transforma elementos de un tipo a elementos de otro tipo. Dependiendo del lenguaje varía mucho la forma de definir una función. En el caso de C, es necesario definir las variables, tipos y parámetros utilizando punteros. Un ejemplo en Pascal:

function f (r: real): integer , donde entra un real y devuelve un entero, representándose como real → integer.

Si no devolviera nada se utiliza el tipo void, para indicar que la función no devuelve ningún valor. En el caso de funciones que vienen predefinidas en el lenguaje como puede ser en C, la función atoi entra un tipo char y devuelve un entero, char → int, o en Pascal la función mod, que entran dos enteros como argumentos de la función y esta devuelve su módulo, otro entero representándose por (integer x integer) → integer.

Clase

Este constructor de tipo se utiliza en los lenguajes orientados a objetos, siendo semejante a una declaración de registro aunque incluye la definición de operaciones, denominadas **métodos o funciones miembro** (Louden, 2004).

© Universidad Europea de Madrid. Todos los derechos reservados.

Declaraciones

Las declaraciones de clase pueden crear nuevos tipos (en C++ lo hacen), e incluso si este es el caso, las declaraciones de clase no son únicamente tipos, porque incluyen características como la herencia, o el enlace dinámico que deben ser mantenidas por estructuras de datos separadas. En el caso de la herencia, es necesario implementar la jerarquía de clases y esto se hace utilizando un grafo acíclico dirigido, y en el caso del enlace dinámico se utiliza otra estructura denominada tabla de método virtual.

© Universidad Europea de Madrid. Todos los derechos reservados.

Equivalencia de tipos

Una vez que conocemos los tipos que pueden existir y con el objeto de poder verificar que un tipo es equivalente a otro tipo, es necesario definir en qué puede consistir dicha equivalencia.

Para resolver este problema se han definido dos tipos de equivalencia: estructural y nominal

Equivalencia estructural

Dos tipos son los mismos si y solo si tienen la misma estructura.

- Dos expresiones de tipos son estructuralmente equivalentes si son el mismo tipo básico o se forman aplicando el mismo constructor de tipos sobre expresiones de tipos estructuralmente equivalentes (Aho et al, 1986).
- Las expresiones estructuralmente equivalentes se representan mediante árboles o grafos dirigidos acíclicos iguales.

En Louden (2004) se define un algoritmo para verificar este tipo de equivalencia. Este tipo de equivalencia es relativamente fácil de implementar, excepto en tipos recursivos. Se utiliza en lenguajes como FORTRAN y COBOL y de forma selectiva en C y Java.

Equivalencia nominal

Dos tipos son los mismos si además de tener la misma estructura se denominan igual.

- Esta interpretación es mucho más restrictiva, puesto que no obvia los nombres. La equivalencia de nombres considera cada nombre de un tipo como un tipo distinto, de modo que dos expresiones de tipo tienen equivalencia de nombres si y solo si son idénticas.

Ada es uno de los pocos lenguajes que ha implementado una forma muy pura de equivalencia de nombres (véase Louden, 2002). C utiliza una equivalencia de nombres para struct y union, y una equivalencia estructural para todo lo demás.

Nombres de los tipos y de los constructores de tipo

Aquí los nombres de los tipos y de los constructores de tipo se obvian y se sustituyen por los tipos:

- Real es equivalente a Real
- Pointer (char) es equivalente a pointer (char).

© Universidad Europea de Madrid. Todos los derechos reservados.

Ejemplo de equivalencia estructural

```
1.  struct A
2.  { int a;
3.    char b;
4.  };
5.
6.  struct B
7.  { int c;
8.    char d;
9.  };
10.
11. struct C
12. { char b;
13.   int a;
14. };
```

Tanto struct A como B son estructuralmente equivalentes, pero no lo es struct C puesto que los campos están invertidos de orden.

Struct y union

En el siguiente ejemplo:

```
1.  struct A
2.  { int a;
3.    char b;
4.  };
5.
6.  struct B
7.  { int c;
8.    char d;
9.  };
10.
11. struct A C;
```

En este ejemplo A, B y C son equivalentes estructuralmente, pero solo A y C son equivalentes en nombre.

© Universidad Europea de Madrid. Todos los derechos reservados.

Ejemplo I

Veamos con un ejemplo la declaración de variables y la inserción de sus tipos correspondientes en la tabla de símbolos. Se tendrá en cuenta si es un tipo simple o un constructor de tipos para la realización de las distintas operaciones.

De entre las tareas que hay que realizar para implementar un sistema de tipos, la primera de ellas es la gestión de declaraciones:

Reglas de la gramática	Reglas semánticas
P → D; S	
D → D; V	
D → V	
V → id: T	añadeTipo (id.entrada, T.tipo)
V → λ	
T → int	T.tipo := integer
T → boolean	T.tipo := boolean
T → char	T.tipo := char
T → real	T.tipo := float
T_1 → array [num] of T_2	T_1.tipo = hazNodo (array, num.valor, T_2.tipo)
T_1 → ^ T_2	T_1.tipo = puntero(T_2.tipo)

Acciones semánticas

Es de destacar que las únicas acciones semánticas que se incorporarán son las relativas a la **gestión del tipo,** no incluyendo las operaciones que pudieran corresponder.

Partimos de la siguiente gramática:

1. P → D; S
2. D → D; V | V
3. V → id: T | λ
4. T → int | boolean | char | real | array [num] of T| ^T
5. S → S; S
6. S → if E then S| id := E | while E do S
7. E → num| num.num | id | true | false | E op E | E or E | E mod E | E[E] |

Declaraciones

Siempre que haya que asignar acciones semánticas, se recomienda construir un árbol sintáctico con una sentencia válida de la gramática, y a partir del árbol identificar la acción semántica que le corresponde a cada producción de la gramática.

© Universidad Europea de Madrid. Todos los derechos reservados.

AñadeTipo

Inserta en la tabla de símbolos el identificador junto con su tipo. Se supone que antes de insertar se comprueba que este identificador no existe ya en la tabla de símbolos para este ámbito.

hazNodo

Construye un nodo de tipo array (en este caso), con un tamaño correspondiente al valor por el número de bytes necesarios para el tipo de las variables de destino del array

© Universidad Europea de Madrid. Todos los derechos reservados.

Ejemplo II

Una vez que sabemos como se gestionan las declaraciones, desde el punto de vista de los tipos que tienen los identificadores, es preciso realizar las tareas de **verificación de tipos** en las **expresiones.** Dejaremos para la coerción de tipos la producción E → E op E.

En el caso de las **sentencias,** estas no tienen tipo a priori, a no ser que así lo decida el diseñador y se suele utilizar el tipo básico vacío para indicar que no lo tienen (Aho et al, 1986).

Expresiones ❭

Sentencias

Reglas de la gramática	Reglas semánticas
E → num	E.tipo = integer
E → num.num	E.tipo = real
E → id	E.tipo = buscaTipo(id.nombre)
E → true	E.tipo = boolean
E → false	E.tipo = boolean

E_1 → E_2 or E_3	if (E2.tipo = boolean) and (E3.tipo = boolean) then
	E1.tipo = boolean
	else
	E1.tipo = tipo_error

E_1 → E_2 mod E_3	if (E_2.tipo = Integer) and (E_3.tipo = Integer)
	E_1.tipo = Integer
	else
	E_1.tipo = Error_Tipo

E_1 → E_2[E_3]	if (E_3.tipo = Integer) and (E_2.tipo = esArray(E_2.tipo) then
	E_1.tipo = E_2.tipo
	else
	E_1.tipo = Error_Tipo

Expresiones

Sentencias ❭

Reglas de la gramática	Reglas semánticas

S → id := E	if (id.tipo = E.tipo) then
	S_0.tipo = vacío
	else
	S_0.tipo := Error_Tipo

S_1 → if E then S_2	if (E.tipo = boolean) and (S_2.tipo = vacío) then
	S_1.tipo = vacío
	else
	S_1.tipo := Error_Tipo

S_1 → while E do S_2	if (E.tipo = boolean) and (S_2.tipo = vacío) then
	S_1.tipo = vacío
	else
	S_1.tipo := Error_Tipo

S_1 → S_2; S_3	if (S_2.tipo = vacío) and (S_3.tipo = vacío) then
	S_1.tipo = vacío
	else
	S_1.tipo := Error_Tipo

© Universidad Europea de Madrid. Todos los derechos reservados.

Coerción de tipos

Debido a que varios operadores están **sobrecargados,** es decir, que se utiliza el mismo operador para operaciones diferentes, es por lo que es necesario convertir los tipos antes de operar.

El caso más típico es el del operador de suma que utilizamos habitualmente para sumar dos enteros o dos reales, pero, **¿qué se hace cuando se suma un entero con un real?** Lo más lógico es pasar el entero a real y sumar dos reales, puesto que si convirtiéramos el real en entero, se perdería información.

Cuando esta conversión se realiza de forma automática (es lo que hace el lenguaje C) por el verificador de tipos, se denomina **coerción de tipos** (Louden, 2004).

La coerción de tipos se realiza en lenguajes como **C#, C++ y Java.**

Reglas de la gramática	Reglas semánticas
$E_1 \rightarrow E_2$ op E_3	if (E_2.tipo = Integer) and (E_3.tipo = Integer) then E_1.tipo = Integer else if (E_2.tipo = Integer) and (E_3.tipo = real) then E_1.tipo = real else if (E_2.tipo = real) and (E_3.tipo = Integer) then E_1.tipo = real else if (E_2.tipo = real) and (E_3.tipo = real) then E_1.tipo = real else E_1.tipo = Error_Tipo}

Esta conversión de tipos también se aplica a las asignaciones (tales como a = b;) en donde si a es real y b es entero, b se convierte en real y se hace la asignación. Es de destacar que si fuera en el sentido contrario, es decir, que a fuera entero y b real, se perdería información.

© Universidad Europea de Madrid. Todos los derechos reservados.

Resumen

En este tema hemos comprendido las tareas necesarias para realizar una verificación y comprobación de tipos, entendiendo que un **tipo de datos** es un conjunto de valores junto con sus operaciones que se describen mediante una **expresión de tipo.**

También hemos conocido lo que es un sistema de tipos como las reglas que hay que aplicar para asignar las expresiones de tipos. Estas expresiones de tipo incluyen a los tipos básicos o simples (entero, real, char, booleano, etc.) junto con los constructores de tipos (*arrays*, funciones, registros, uniones, punteros y clases), así como las tareas que hay que realizar para verificar sus tipos. Además se ha visto lo que es un lenguaje fuertemente tipificado y su opuesto.

Por otro lado se ha visto la equivalencia de tipos en sus dos vertientes la estructural y la nominal, siendo mucho más restrictiva esta última.

Finalmente se ha conocido el concepto de conversión de tipos y la denominación de esta cuando es realizada automáticamente por el compilador, y qué es coerción de tipos.

© Universidad Europea de Madrid. Todos los derechos reservados.