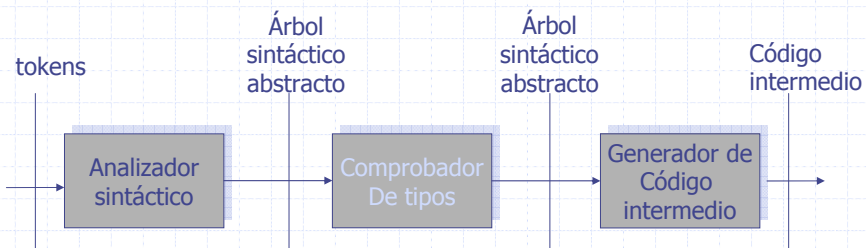


Comprobación de Tipos

- ◆ Introducción
- ◆ Sistemas de tipos
 - Expresiones de tipos
 - Sistemas de tipos
 - Comprobación estática y dinámica de tipos
 - Tablas de Símbolos
- ◆ Especificación de un comprobador de tipos sencillo
 - Conversiones y sobrecarga de tipos
- ◆ Equivalencia de expresiones de tipos

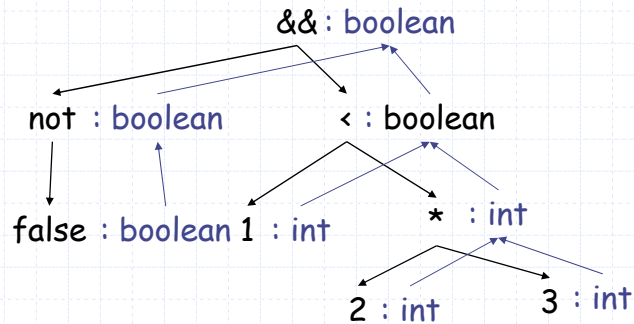
Comprobación de Tipos

- Un comprobador de tipos
 - Calcula y mantiene la información de tipos (inferencia)
 - Comprueba que el tipo de una construcción tenga sentido en su contexto según el lenguaje
- Ubicación:



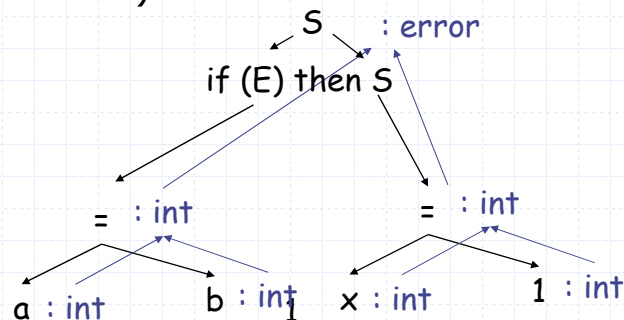
Ejemplo Comprobación de Tipos

◆ Ej. 1: Tipos para "not false && 1 < 2 * 3"



Ejemplo Comprobación de Tipos

◆ Ej. 2: Tipos para "if (a=b) x=1;" (error en Java, no en C)



◆ Es una aplicación de evaluación de gramática atribuida

- No siempre es tan sencillo

Comprobación de Tipos

- ◆ Un lenguaje especifica que operaciones son válidas para cada tipo
 - Formalización de reglas semánticas de verificación
- ◆ Se detectan errores
 - Acceso incorrecto a memoria:
 - Límites de abstracción, mal uso de estructuras, etc.
- ◆ Tipos de lenguajes:
 - *Estáticamente tipificados*: La mayoría de comprobaciones se realizan en tiempo de compilación (C, Java)
 - *Dinámicamente tipificados*: La mayoría de comprobaciones en ejecución (Scheme, LISP)
 - *No tipificados*: Ninguna comprobación (código ensamblador)

Expresiones de tipos I

- ◆ Representan el tipo de las construcciones
 - Valores posibles y operaciones que pueden aplicarse
 - Especificaciones del lenguaje para operaciones:
 - ◆ Tipo de Expresión resultante de aplicar operadores aritméticos
 - ◆ Resultado de aplicar operador &
 - ◆ Tipo de llamada a función
 - ◆ ...
 - Las clases en POO son una extensión
- ◆ Pueden ser
 - Un tipo básico
 - ◆ Boolean, Char, Integer, Real, Vacio, Error_tipo, ...
 - Un constructor de tipos aplicado a expresiones de tipos
 - ◆ array de variables de tipo básico, puntero a variable, registro,...

Expresiones de tipos II

◆ Expresiones de tipos

1. Tipos básicos
2. El nombre de un tipo es una expresión de tipo
3. Constructores de tipos y expresiones de tipos son tipo
 - a) **Arrays**
 Si T es una expresión de tipo entonces array (I, T) es una expresión de tipo que indica una matriz con elementos de tipo T y conjunto de índices I
 Var A: array[1..10] of integer es array (1..10, integer)
 - b) **Productos**
 Si T1 y T2 son expresiones de tipo, su producto cartesiano T1 x T2 es una expresión de tipo (x es asociativo por la izquierda)

Expresiones de tipos III

c) Registros

Tiene como tipo el producto de los tipos de los campos. El constructor de tipo "record" se aplica a una tupla formada con nombres de campos y tipos de campos. Ej:

```
type fila = RECORD
    direccion: integer;
    lexema: array [1..15] of Char;
END;
```

```
var tabla: array [1..10] of fila
```

fila tiene el tipo:

record ((dirección x integer) x (lexema x array(1..15, Char)))

Tabla tiene el tipo:

array(1..10, record ((dirección x integer) x (lexema x array(1..15, char))))

o: **array(1..10, Tipo_Fila)**

Expresiones de tipos IV

d) Punteros

Si T es una expresión de tipo, $\text{Pointer}(T)$ es una expresión de tipo que es un puntero a un objeto de tipo T . Ej.:

```
Var p: ^fila;
```

p es de tipo: **pointer (fila)**

e) Funciones

Si T_1 y T_2 son expresiones de tipos, entonces $T_1 \rightarrow T_2$ es la expresión de tipos de una función que toma argumentos de T_1 y los transforma en T_2

```
Function f (a, b : Char) : ^Integer;
```

f tiene el tipo: **(char x char) \rightarrow pointer (Integer)**

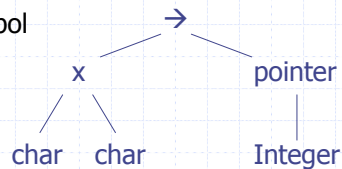
Representación de tipos

◆ En el lenguaje que implementa el compilador

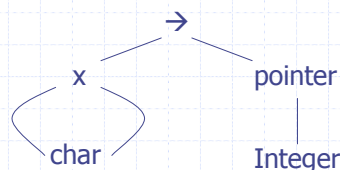
- Estructura especial de tabla de símbolos

◆ Representaciones de expresiones de tipos

- Representación en árbol



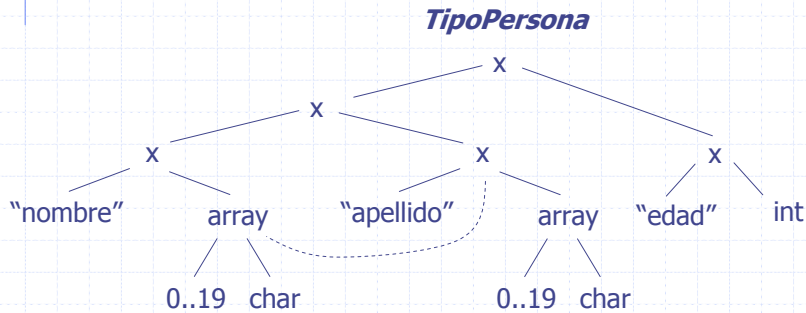
- Representación en Grafo dirigido acíclico



Representación de tipos

◆ Ejemplo representación constructores de tipos

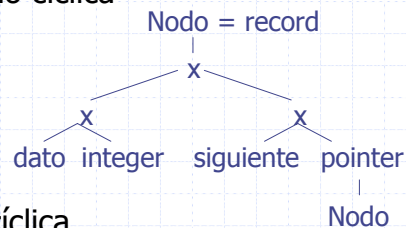
```
typedef struct{
    char nombre[20];
    char apellido[20];
    int edad;} TipoPersona;
```



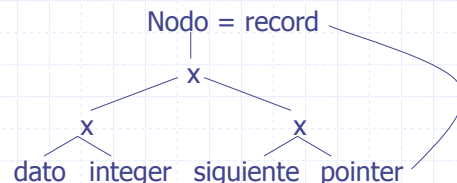
Ciclos en la representación de tipos

◆ Representaciones recursivas

■ Representación no cíclica



■ Representación cíclica



Representación de tipos

Representación tabular

- Si se definen tipos nuevos, preciso TS para añadirlos

Ej.:

```
int a;
float **b;
float *c[10]
int f(char, float, int);
```

Tabla Símbolos

Id.	Tipo	...
a	0	
b	4	
c	5	
f	8	

Tabla Tipos

Núm.	Tipo	Tam/ Tipo1	Tipo Base
0	entero	4	
1	real	8	
2	caracter	1	
3	puntero	1	1
4	puntero	1	3
5	array	10	3
6	pr. cart.	2	1
7	pr. cart.	6	0
8	función	7	0

Representación de tipos

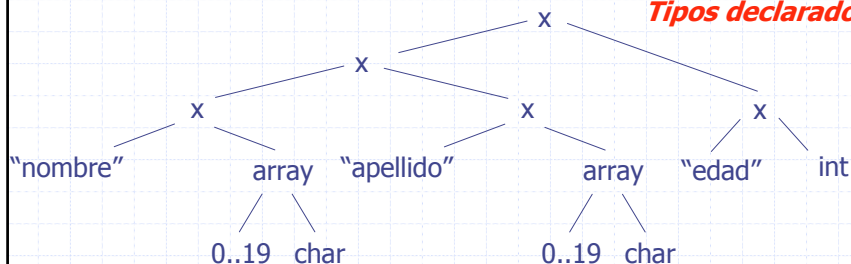
Tipos declarados y tipos "anónimos"

```
struct{
    char nombre[20];
    char apellido[20];
    int edad;} e1;
char[20] n1;
```

Tipos anónimos

```
typedef struct{
    char nombre[20];
    char apellido[20];
    int edad;} tipoPersona;
typedef char[20] cadena;
tipoPersona e2;
cadena n2;
```

Tipos declarados



Sistemas de tipos

- ◆ Sistema de tipos: conjunto de reglas para asignar expresiones de tipos a construcciones de un programa

- Un comprobador de tipos implanta un sistema de tipos
- Se pueden implementar en la definición dirigida por la sintaxis

```
VAR x: RECORD
    p_real: real;
    p_imag: real;
END;
VAR y: RECORD
    p_imag: real;
    p_real: real;
END;
x=y;                                INCORRECTO
x.p_real=y.p_imag;                 CORRECTO
```

- ◆ Comprobación del sistema de tipos

- **Estática:** Aquella que es realizada por el compilador antes de la ejecución del programa
- **Dinámica:** Aquella que es realizada al ejecutar el programa objeto

Comprobación estática y dinámica

- ◆ Sistema de tipos seguro (*sound type*): no necesita comprobar dinámicamente errores de tipos
- ◆ Lenguajes fuertemente tipificados: garantizan que los programas que aceptan se ejecutarán sin errores de tipo
- ◆ Algunas comprobaciones sólo pueden ser dinámicas:

```
Tabla: array[0..255] of Char;
i: Integer;
<< se asigna valor a i >>
Tabla [i] := 3;
```

- El compilador no puede garantizar estáticamente que el valor de la variable i no exceda los límites del array tabla
- Polimorfismo en lenguajes OO

Tablas de Símbolos I

◆ Recogen las diferentes declaraciones (explícitas) del programa

- Declaraciones de constantes (nombre, tipo, valor,...)


```
const int SIZE=20
```
- Declaraciones de variables (nombre, tipo, dirección,...)


```
Tabla TS;
int var[SIZE];
```
- Declaraciones de funciones (nombre, tipos, dirección,...)


```
int buscar (int* vector);
```
- Declaraciones de tipos (nombre, tipo, tamaño,...)


```
typedef struct Tabla{
{char* nombre;
int contador;
Tabla* siguiente;
}
```

Tablas de Símbolos II

◆ Estructura de Datos para almacenar identificadores definidos en el programa

- Se utiliza en diferentes fases del análisis y síntesis
- Funciones de **inserción**, **búsqueda** y **eliminación**
- Solución habitual con tabla de hash
 - ◆ Optimización del compilador



- Almacena la información de interés: tipo, dirección en memoria, ...

Tablas de Símbolos III

◆ Ej.:

```
int a,b;
float c,d;
char e,f;
```

NOMBRE	TIPO	TAMAÑO	DIRECCIÓN
a	Entero	4	100
b	Entero	4	104
c	Real	8	112
d	Real	8	120
e	Caracter	1	121
f	Caracter	1	122

- Dos usos en la verificación semántica
 - ◆ En la declaración de una variable no hay colisiones (buscar e insertar)
 - ◆ El uso de una variable, es el esperado según su tipo (buscar)

Tablas de Símbolos IV

◆ Gestión de ámbitos en la tabla de símbolos

- Los lenguajes con **estructura de bloques** permiten declaraciones en distintos ámbitos, se puede dar anidamiento

```
{ int x = 0;
  foo(x);
  {   int x = 1;
      bar(x);
  }
  baz(x);
}
```

◆ Idea (variable x declarada en un bloque):

- Al añadir la definición de x en un nuevo ámbito, ocultar las definiciones previas (sin eliminar)
- Al salir del bloque, eliminar esta definición de x y restaurar las anteriores

◆ Lectura hacia atrás de la tabla

Tablas de Símbolos V

◆ Tablas para estructuras registro

- Los registros tienen campos con desplazamiento fijo

```
typedef struct cplx{
    double real;
    double imag;
};
cplx a, b;
```



base de **cplx**
desplaz. campo "imag"

Tabla Símbolos

Id.	Tipo	dirección
a	1	100
b	1	116
...		

Tabla Tipos

Núm.	Tipo	Tam/ Tipo1	Tipo Base
0	real	4	
1	pr. cart.	0	0
...			

Tabla Registros

Nombre	Tipo	dirección
real	0	0
imag	0	8

Un Comprobador de Tipos Sencillo

Un lenguaje sencillo

Comprobación de tipos en expresiones

Comprobación de tipos en proposiciones

Comprobación de tipos de funciones

Un lenguaje sencillo I

- ◆ Ejemplo de lenguaje, con todos los identificadores declarados antes de uso
 - Objetivo: comprobar el tipo de toda expresión
- ◆ Tipos del lenguaje
 - Tipos básicos
 - ◆ Char
 - ◆ Integer
 - ◆ Error_Tipo
 - Tipos complejos
 - ◆ Array[n] of T es de tipo array (T, 1..n)
 - ◆ ^T es de tipo pointer (T)

Un lenguaje sencillo II

- ◆ Gramática del lenguaje

```
P → Ds E
Ds → D ; Ds | λ
D → id : T
T → char | integer | array [num] of T | ^T
E → literal | num | id | E mod E | E [ E ] | E ^
```

- D es cada declaración
- T es el tipo
- E es la expresión

Un lenguaje sencillo III

◆ Acciones semánticas de construcción de tipos

$P \rightarrow Ds E$	
$Ds \rightarrow D ; Ds$	
$Ds \rightarrow \lambda$	
$D \rightarrow id : T$	{ añadeTipo (id.entrada, T.tipo) }
$T \rightarrow char$	{ T.tipo := char }
$T \rightarrow integer$	{ T.tipo := integer }
$T_0 \rightarrow \wedge T_1$	{ $T_0.tipo := pointer(T_1.tipo)$ }
$T_0 \rightarrow array [num] \text{ of } T_1$	{ $T_0.tipo := array(1..num.val,$ $T_1.tipo)$ }

Un lenguaje sencillo IV

◆ Acciones semánticas de verificación de tipos en expresiones

■ Constantes

$E \rightarrow literal$	{ E.tipo := Char }
$E \rightarrow num$	{ E.tipo := Integer }

■ Identificadores

$E \rightarrow id$	{ E.tipo := buscaTipo (id.entrada) }
--------------------	--------------------------------------

■ Operadores

$E \rightarrow E \text{ mod } E$	{ if (E ₁ .tipo = Integer) and (E ₂ .tipo = Integer) E ₀ .tipo = Integer else E ₀ .tipo = Error_Tipo }
----------------------------------	---

Un lenguaje sencillo V

◆ Acciones semánticas de verificación de tipos en expresiones

■ Arrays

$E \rightarrow E [E]$ { if $(E_1.\text{tipo} = \text{Integer})$ and $(E_2.\text{tipo} = \text{array}(s, t) \rightarrow)$
 $E_0.\text{tipo} = t$
 else $E_0.\text{tipo} = \text{Error_Tipo}$ }

■ Punteros

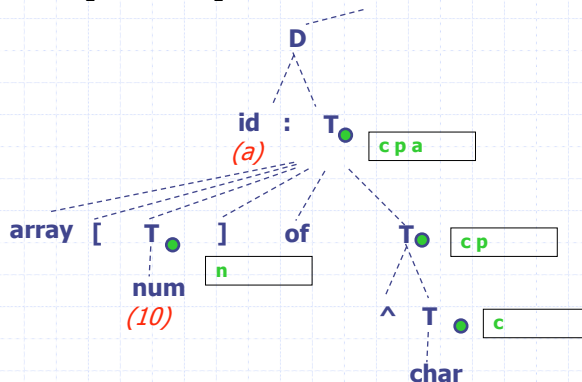
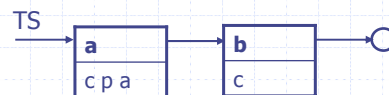
$E \rightarrow E ^$ { if $(E_1.\text{tipo} = \text{pointer}(t))$
 $E_0.\text{tipo} = t$
 else $E_0.\text{tipo} = \text{Error_Tipo}$ }

Ejemplo Verificación Tipos

a: array [10] of ^char

b: char

b=a[4 mod 3] ^

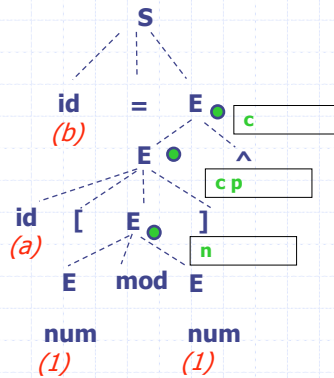
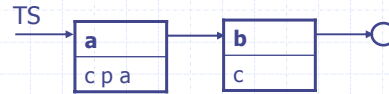


Ejemplo Verificación Tipos

a: array [10] of ^char

b: char

b=a[4 mod 3] ^



Tipos en sentencias

- ◆ Por defecto, una sentencia es de tipo nulo (vacío), salvo si es errónea
 - Propositiones if y while

$S \rightarrow \text{id} := E$	{ if (id.tipo = E.tipo) then $S_0.\text{tipo}$ = vacío else $S_0.\text{tipo} := \text{Error_Tipo}$ }
$S \rightarrow \text{if } E \text{ then } S$	{ if (E.tipo = boolean) then $S.\text{tipo} = S_1.\text{tipo}$ else $S_0.\text{tipo} := \text{Error_Tipo}$ }
$S \rightarrow \text{while } E \text{ do } S$	{ if (E.tipo = boolean) then $S.\text{tipo} = S_1.\text{tipo}$ else $S_0.\text{tipo} := \text{Error_Tipo}$ }
$S \rightarrow S ; S$	{ if ($S_1.\text{tipo}$ = vacío) and ($S_2.\text{tipo}$ = vacío) then $S_0.\text{tipo}$ = vacío else $S_0.\text{tipo} := \text{Error_Tipo}$ }

Tipos en sentencias

- Ampliar G para permitir declaración de funciones

$$T \rightarrow T' \rightarrow T \quad \{ T_0.\text{tipo} = T_1.\text{tipo} \rightarrow T_2.\text{tipo} \}$$

Ej.: int f(double x, char y) { ... }

tipo de f: double x char \rightarrow int

↑
tipos argum.

↑
tipo devolución

- Llamada a la función, con parámetros

$$E \rightarrow E (E) \quad \{ \text{if } (E_2.\text{tipo} = s) \text{ and } (E_1.\text{tipo} = s \rightarrow t) \\ \text{then } E_0.\text{tipo} = t \\ \text{else } E_0.\text{tipo} := \text{Error_Tipo} \}$$

Conversiones de tipos

- ◆ Algunos operadores pueden aplicarse a operandos de distintos tipos (promoción o coerción):

- $x + y$?
- Si el tipo de x es double y el de y int, tipo resultado? (afecta al código a generar)

$$E \rightarrow E \text{ op } E \quad \{ \text{if } (E_1.\text{tipo} = \text{Integer}) \text{ and } \\ (E_2.\text{tipo} = \text{Integer}) \text{ then } E_0.\text{tipo} = \text{Integer} \\ \text{else if } (E_1.\text{tipo} = \text{Integer}) \text{ and } \\ (E_2.\text{tipo} = \text{real}) \text{ then } E_0.\text{tipo} = \text{real} \\ \text{else if } (E_1.\text{tipo} = \text{real}) \text{ and } \\ (E_2.\text{tipo} = \text{Integer}) \text{ then } E_0.\text{tipo} = \text{real} \\ \text{else if } (E_1.\text{tipo} = \text{real}) \text{ and } \\ (E_2.\text{tipo} = \text{real}) \text{ then } E_0.\text{tipo} = \text{real} \\ \text{else } E_0.\text{tipo} = \text{tipo_error} \}$$

Tipos sobrecargados I

- ◆ Algunos operadores y funciones pueden tener distintos significado según su contexto:
 - $(4 + \text{"a"})$ y $(4 + 'a')$ son expresiones distintas en Java
 - No siempre es posible resolver sólo con los operandos cuando hay promoción automática:
 - ◆ function $"*"$ (k,j: integer) return integer
 - ◆ function $"*"$ (x,y: real) return real
 - ◆ Hay sobrecarga en Ada y en C++, no en C y Pascal
 - ◆ $"*"$ Puede tener los tipos posibles:
 - integer x integer \rightarrow integer
 - integer x integer \rightarrow real
 - real x real \rightarrow real
 - Así: $3.1 * 5$ pasa a ser $3.1 * (\text{real})5$
 $3 * 5$ es ambiguo: puede ser integer o real: $(\text{real})3 * (\text{real}) 5$
 en la expresión $2 * (3 * 5)$ tiene tipo $(i * i) \rightarrow i$
 en la expresión $z * (3 * 5)$ tiene tipo $(r * r) \rightarrow r$
- Verificaciones "de larga distancia"

Tipos sobrecargados II

- Tipado sobrecargado: si permite que las construcciones tengan más de un tipo
 - ◆ La sobrecarga permite varias declaraciones del mismo nombre
 - ◆ El extremo es una declaración para cualquier tipo:
 - Procedure swap(var x,y:anytype): plantilla (*template*)
- Se generalizan las verificaciones para considerar conjuntos de tipos posibles de una expresión:
 - ◆ Se supone que la tabla de símbolos puede contener el conjunto de posibles tipos
 - ◆ El tipo conjunto vacío se asimila con tipo_error
 - ◆ Problema similar a la "inferencia de tipos"
- Ejemplo DDS

$E' \rightarrow E$	$\{ E'.\text{tipos} = E.\text{tipos} \}$
$E \rightarrow \text{id}$	$\{ E.\text{tipos} = \text{consulta}(\text{id.entrada}) \}$
$E \rightarrow E (E)$	$\{ E_0.\text{tipos} = \{ t \mid \text{existe un tipo } s \text{ en } E_2.\text{tipos} \text{ tal que } s \rightarrow t, \text{ con } t \text{ en } E_1.\text{tipos} \} \}$

Tipos sobrecargados III

- Algunos lenguajes, como Ada, obligan a que una expresión finalmente tenga un tipo único, sino es un error
- Se introducen un nuevo atributo: **"único"**

```

E' → E      { E'.tipos:=E.tipos
              E.unico:= if E'.tipos={t} then t
                      else tipo_error}

E → id      { E.tipos:=consulta(id.entrada)}

E0 → E1 ( E2 )
{ E0.tipos:= {s' | existe un tipo s en
  E2.tipos tal que s→s' está en E1.tipos}
  t=E0.unico
  S= {s tal que s en E2.tipos y s→ t en E1.tipos}
  E2.unico:=if S== {s} then {s}, else tipo_error
  E1.unico:=if S== {s} then s→ t, else tipo_error }

```

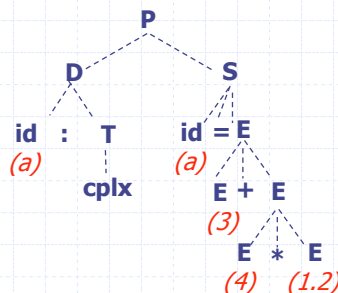
- Implementación en dos pasadas para evaluar **"tipos"** y **"único"**

Tipos sobrecargados IV

- Ej.:
a: cplx;
a=3 + 4 * 1.2

"*", "+" sobrecargados:
 int x int → int
 real x real → real
 cplx x cplx → cplx

atributos: tipos,
único ?



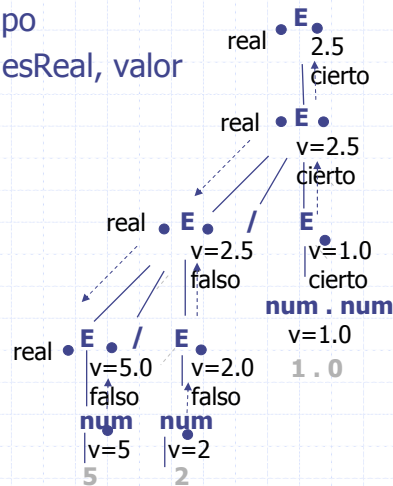
Evaluación combinada en dos pasadas

◆ Ejemplo (promoción en expresiones):

- Atributo heredado: tipo
- Atributo sintetizado: esReal, valor

$S ::= E$
 $E ::= E / E$
 | num
 | num.num

Sentencia: 5/2/1.0
 Resultado: 2.5
 No en Java/C (2.0)



Equivalencia de tipos

- ◆ Aspecto esencial en las verificaciones semánticas
- ◆ Dos posibilidades básicas:

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 sólo si son idénticas

Con la equivalencia estructural, los nombres se sustituyen por las expresiones de tipos que definen. Dos expresiones de tipos son estructuralmente equivalentes si y son idénticas al sustituirlos los tipos por sus expresiones de tipo correspondientes

Equivalencia estructural I

- ◆ Dos expresiones de tipos son estructuralmente equivalentes si son el mismo tipo básico o se forman aplicando el mismo constructor de tipos a expresiones de tipos estructuralmente equivalentes
 - integer es equivalente a integer
 - pointer (integer) es equivalente a pointer (integer)
 - ...
- ◆ Las expresiones estructuralmente equivalentes se corresponden con árboles o grafos acíclicos iguales

Equivalencia estructural II

- ◆ Algoritmo para comprobar la equivalencia estructural:

```

Function Equivale (s, t) : boolean
  if s y t son del mismo tipo básico then return true
  else if s = array(s1, s2) and t = array (t1, t2) then
    return Equivale (s1, t1) and Equivale (s2, t2)
  else if s = s1 x s2 and t = t1 x t2 then
    return Equivale (s1, t1) and Equivale (s2, t2)
  else if s = pointer (s1) and t = pointer (t1) then
    return Equivale (s1, t1)
  else if s = s1 → s2 and t = t1 → t2 then
    return Equivale (s1, t1) and Equivale (s2, t2)
  else return false
  
```

- Algunas comprobaciones pueden "relajarse" (arrays...)


Nombres de expresiones de tipos I

- ◆ Algunos lenguajes permiten nombrar tipos

```
Type enlace = ^nodo;
Var siguiente : enlace;
    ultimo: enlace;
    p : ^nodo;
    q, r : ^nodo;
```

- ◆ ¿Tienen igual tipo las variables siguiente, ultimo, p, q, r?
 - ... depende de la implementación

- ◆ Un lenguaje puede forzar el uso de tipos nombrados:

<pre>record x: pointer to real y:array[10] of int end</pre>		<pre>t1=pointer to real t2=array[10] of int t3=record x:t1 y:t2 end</pre>
---	---	---

Equivalencia de nombres

- ◆ Algoritmo para comprobar la equivalencia de nombres:

```
Function Equivale (s, t) : boolean
  if s y t son del mismo tipo básico then return true
  else if s y t son nombres de tipo
    if son iguales return true
    else return false
```

Equivalencia de nombres

◆ Solución más común en lenguajes imperativos

- Más restrictiva, más fácil de implementar

```
typedef struct{  
    char nombre[20];  
    char apellido[20];  
    int edad;} cliente;  
void registrar (cliente*p){...}  
empleado e;  
registrar(&e); //error
```

```
typedef struct{  
    char nombre[20];  
    char apellido[20];  
    int edad;} empleado;
```

