7. Comprobaciones de tipos

1. Sistema de tipos.
2. Comprobaciones semánticas
3. Ámbito de declaraciones
4. Comprobaciones de tipo
5. Ejemplo de sistema de tipos.

1. Sistema de tipos

Sistema de tipos

Sistema de tipo:

- Está formado por un conjunto de reglas que asignan expresiones de tipo a las construcciones de un lenguaje y que definen la equivalencia de tipos, la compatibilidad de tipos y la inferencia de tipos.

Comprobador de tipos:

- Implementación de un sistema de tipos.

Expresión de tipo

- Una expresión de tipo es un tipo básico o un constructor de tipos aplicado a una o más expresiones de tipo.

- El conjunto de tipos básicos y constructores de tipo depende del lenguaje fuente. Una **expresión de tipo** será:
  - *Un tipo básico: tentero, treal, tcar, tlogico, terror y tvacio.*
  - *El nombre de una expresión de tipo.*
  - *Un constructor de tipos aplicado a expresiones de tipo:*
    - *tpuntero (T)*
    - *tvector ($I_1 x I_2 x \ldots x I_k$, T)*
    - *$T_1 x T_2$*
    - *D -> R*
    - *testructura($(N_1 \times T_1) \times (N_2 \times T_2) \times \ldots \times (N_k \times T_k)$)*

2. Comprobaciones semánticas

Comprobaciones dinámicas vs estáticas

Comprobación dinámica:

Se realiza durante la ejecución del programa objeto (en tiempo de ejecución).

Comprobación estática:

Se realiza en tiempo de compilación.

Algunas comprobaciones de tipo solo pueden realizarse en tiempo de ejecución.

Comprobaciones estáticas

Para almacenar la información semántica de los objetos que aparecen en el programa fuente se utiliza la Tabla de Símbolos (TDS).

- *Comprobaciones del ámbito de las declaraciones*
- *Comprobaciones de tipo: Comprobación de que los tipos de los operandos y operadores de las expresiones son compatibles.*
- *Comprobación de declaración: Un identificador no puede usarse antes de ser declarado.*
- *Comprobación de unicidad: Los identificadores no pueden definirse más de una vez dentro del mismo bloque.*
- *Comprobación de parámetros: Los métodos (o funciones) deben invocarse con el número y tipo de parámetros adecuado,*
- *Otras: Comprobaciones de flujo de control, relaciones de herencia, unicidad de clases y métodos,…*

Comprobaciones dinámicas

Dependen del contexto de la ejecución.

Ejemplos:

- *Verificar estado de la pila (stack) y montículo (heap)*
- *Verificación de desbordamientos (overflow y underflow).*
- *Divisiones por cero*
- *Verificaciones de direcciones e índices en variables indexadas*

…

- Un lenguaje es fuertemente-tipado si **prohíbe**, de forma que la implementación del lenguaje pueda asegurar su cumplimiento, la aplicación de una **operación** a cualquier objeto que **no la soporte**.

- Un lenguaje es estáticamente tipado, si es **fuertemente tipado** y las comprobaciones de tipo pueden realizarse en **tiempo de compilación**.
  - Pocos lenguajes son estáticamente tipados en el sentido riguroso. Pero se suele aceptar que lo son Ada (en su mayor parte) ó C

- Dinámicamente tipados: Lisp, Smalltalk, lenguajes de script,.. En general lenguajes con ámbito dinámico.

3. Ámbito de declaraciones

Ámbito de una variable

- El ámbito de una declaración define el segmento de programa en el que se aplica la declaración (la variable definida es accesible).
- Relaciona la declaración de la variable con su uso
- Lenguaje de ámbito estático (o léxico)

  Si el ámbito está completamente determinado por su posición en el código fuente.
- Lenguaje de ámbito dinámico

  Si el ámbito depende del estado durante la ejecución del programa.

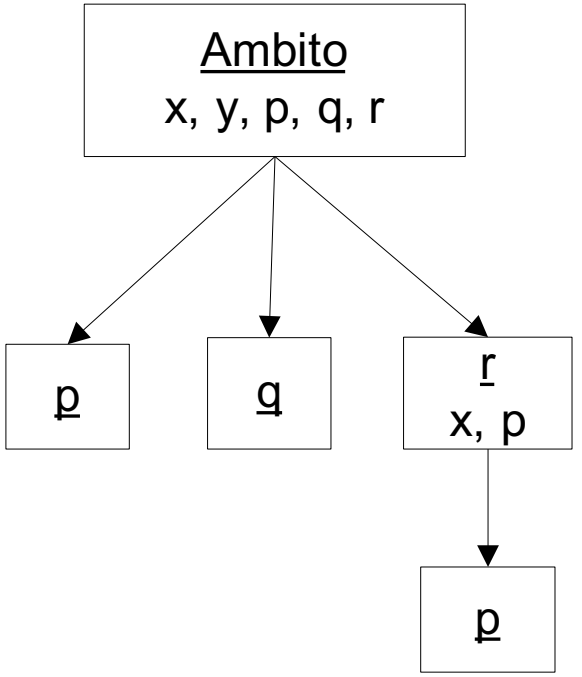
Ámbito estático vs dinámico

Ámbito	función	Variables	Valor de y
	Dinámico	Dinámico	6
	Estático	Dinámico	10
	Estático	Estático	5

Ámbito estáticos (léxico) vs. ámbito dinámico

```
int x, y ;
void function p {
    x = x * 2; }
void function q {
    p(); }
void function r {
     int x;
     void function p {
          x=x+1 ; }
     x = 5; q(); y = x; }
void function principal {
    x = 0; r(); write(y);
}
```

Ámbito estático

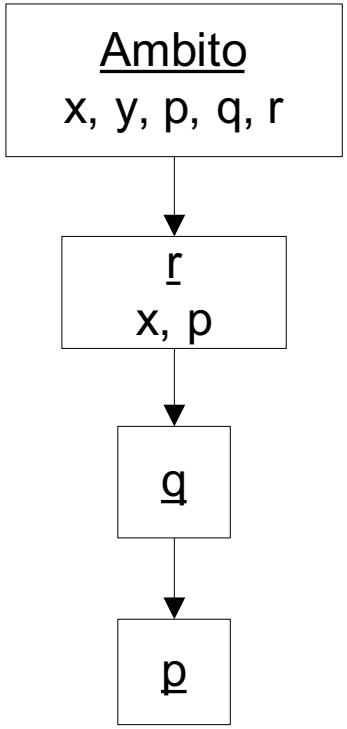


Ámbito estático vs dinámico

Ámbito	función	Variables	Valor de y
	Dinámico	Dinámico	6
	Estático	Dinámico	10
	Estático	Estático	5

```
int x, y ;
void function p {
    x = x * 2; }
void function q {
    p(); }
void function r {
     int x;
     void function p {
          x=x+1 ; }
     x = 5; q(); y = x; }
void function principal {
    x = 0; r(); write(y);
}
```

Ámbito dinámico



Comprobaciones de ámbito

- Un mismo nombre puede identificar objetos distintos en distintas partes del programa, pero no se pueden solapar sus ámbitos.
- En lenguajes con ámbitos estáticos se permite anidamiento de declaraciones (C, C++, Java,…)
- Los método no necesitan estar declarados en la clase que lo utiliza si lo están en una antecesora.
- Los métodos pueden redefinirse

Ejemplo ámbito estático

```
int  a, b;
void function f1 (int p1, p2){
      int  c, d ;
      int function f2 (int p3, p4, p5){
            int e, c ;
             … }
      void function f3 (int p6, p7){
            f2(3,4,5) ; }
      f3(6,7);  … }
int function f4 {
      int f ;
       …  }
void function principal {
      f1(1,2) ;  }
```

4. Comprobaciones de tipos

Equivalencia de tipos

- En un lenguaje tipado estáticamente toda definición de un objeto (constante, variable, subrutina,…) debe especificar el tipo del objeto.

- ¿Cuándo son equivalente dos expresiones de tipo?
  - *Equivalencia por nombre:* Dos expresiones de tipo son equivalentes si tienen el mismo nombre.
  - *Equivalencia estructural:* Dos expresiones de tipo son equivalentes si representan expresiones estructuralmente equivalentes después de sustituir todos los nombres por las expresiones de tipo que representan.

Ejemplos

- Equivalencia estructural: Algol 68, Módula-3, C (casi todo), ML, Pascal (primeros compiladores)
- Equivalencia por nombre: Java, C#, Ada, Pascal estándar.

Compatibilidad de tipos

- No siempre se exige que 2 tipos sean iguales (equivalentes), puede bastar con que sean compatibles en el contexto en el que aparecen.

Ej.
```
int b;  float c, a;
    a = b + c ;
```

Conversiones de tipo

Coerción:

- Se permite un tipo en un contexto donde se esperaba otro.
- La implementación del lenguaje debe convertir automáticamente al tipo esperado: Conversión de tipos implícita introducida por el compilador.
- Puede requerir código para la comprobación de tipos en tiempo de ejecución.

Conversión explícita:

- Cuando el programador debe indicar explícitamente la conversión en el programa fuente.

Conversiones de tipo

Sobrecarga

- Un **operador** de función puede representar **diferentes** operaciones según el contexto en el que se usa.
- La sobrecarga se resuelve determinando qué ocurrencia del símbolo sobrecargado se está empleando en cada **contexto**.
- Se resuelve en tiempo de **compilación**

Tipos de referencia genéricos

- Se permiten objetos "contenedores" que apuntan a otros objetos
- Ej. C y C++: void *

Polimorfismo

Un cuerpo de **código** funciona con objetos de **varios tipos.**

Puede (o no) necesitar comprobaciones de tipo en tiempo de ejecución.

- Polimorfismo paramétrico explícito
  - El código recibe un tipo como parámetro
  - La inferencia de tipos asigna (infiere) un tipo en tiempo de *ejecución* a cada objeto o expresión.
  - El programador puede definir clases con parámetros de tipo.
- Polimorfismo de subtipo (de herencia)
  - Permite a una variable de tipo T referirse a un objeto de cualquier tipo derivado a partir de T
  - Puede implementarse en tiempo de *compilación*
  - Ej. C++, Java, Eiffel

Inferencia de tipos

- Consiste en calcular (inferir) el tipo de un objeto o expresión.
- A veces no es sencillo
- El resultado de un **operador** aritmético suele tener el mismo tipo que los operandos.
- Una **asignación** suele tener el tipo de la expresión de su lado izquierdo

5. Ejemplo de sistema de tipos

Declaraciones (1/6)

$P \rightarrow D \quad D\_F$

$T \rightarrow$ **int** { *T.tipo = tentero ; }*

| **float** { *T.tipo = treal ; }*

| **bool** { *T.tipo = tlogico ; }*

| **struct { C }** { *T.tipo = testructura (C.tipo) };*

| T_1 * { *T.tipo = tpuntero(T_1.tipo) ;}*

$D \rightarrow D \; D$

| ε

| T **id ;** { ***InsTds*** *(id.nom,"variable",T.tipo) ; }*

| T **id** L_I **;** { ***InsTds*** *(id.nom,"variable",tvector(L_I.tipo,T.tipo) ;}*

$D\_F \rightarrow$ T **id** (P_F) { ***InsTds*** *(id.nom,"funcion", P_F.tipo -> T.tipo) ; }*

{ D1 L_Inst } D_F

| ε

Índices declaración array (2/6)

*/**** Índices de declaración de array ****/*

L_I → **[cte]** *{ si (cte.tipo ≠ tentero) **OR** (cte.valor < 0)*

ent { yyerror() ; L_I.tipo = terror ; }

sino L_I.tipo = cte.valor ; }

| L_I$_1$ **[cte]** *{ si (cte.tipo ≠ tentero) **OR** (cte.valor < 0)*

ent { yyerror(); L_I.tipo = terror ; }

sino si L_I$_1$.tipo == terror ent L_I.tipo = terror

sino L_I.tipo = L_I$_1$.tipo x cte.valor ; }

/**** Miembros de estructuras ****/

C → T **id** *{ C.tipo = (id.nom x T.tipo) ; }*

| C1 **;** T **id** *{ C.tipo = C1.tipo x (id.nom x T.tipo) ; }*

(3/6)

*/**** Parámetros formales ****/*

P_F → L_PF { *P_F.tipo = L_PF.tipo ; }*

| ε { *P_F.tipo = tvacio ; }*

L_PF → T **id** { ***InsTds*** *(id.nom,"parametro",T.tipo) ;*

L_PF.tipo = T.tipo ; }

| L_PF_1 , T **id** { ***InsTds***(*id.nom,"parametro",T.tipo) ;}

{ *$L\_PF.tipo = L\_PF_1.tipo$ x T.tipo ; }*

Expresiones (4/6)

E → **cte** *{ E.tipo = cte.tipo ; }*

| **id** *{ Si* ***NOT ObtTds****(id.nom,* E.tipo*)*
ent { E.tipo = terror ; yyerror()} ; }

| **id** L_E *{ Si* ***ObtTds****(id.nom, tvector(*I,tipo*))* ***AND***
NumDimensiones*(I) == L_E.ndim* ***AND*** *(L_E.tipo ≠ terror)*
ent E.tipo = tipo sino { E.tipo = terror ; yyerror() ; } }

| **id** P_A *{ Si* ***ObtTds****(id.nom,* D->R*)* ***AND*** *(D == P_A.tipo)*
ent E.tipo = R sino { E.tipo = terror ; yyerror() ; } }

| * **id** *{ Si* ***ObtTds****(id.nom, tpuntero(tipo)) ent E.tipo = tipo*
sino {E.tipo = terror ; yyerror() ; } }

| & **id** *{ Si* ***ObtTds****(id.nom, tipo) ent E.tipo = tpuntero(tipo)*
sino { E.tipo = terror ; yyerror() ; }}

| **id**$_1$. **id**$_2$ *{Si* ***ObtTds****(id$_1$.nom, testructura(tipo))* ***AND***
BuscarCampo*(tipo, id$_2$.nom,* tipo-campo*)*
ent E.tipo = tipo-campo sino {E,tipo = terror ; yyerror() ; } }

(5/6)

/**** Índices de un array ****/

L_E → [E] { Si E.tipo == tentero
ent L_E.tipo = tentero sino L_E.tipo = terror ;
L_E.ndim = 1 ; }

L_E → L_E1 [E] { Si E.tipo == tentero **AND** L_E1.tipo == tentero
ent L_E.tipo = tentero sino L_E.tipo = terror;
L_E.ndim = L_E1.ndim + 1 ; }

/**** Parámetros actuales ****/

P_A → ε { P_A.tipo = tvacio ; }
| (L_PA) { P_A.tipo = L_PA.tipo ; }

L_PA → E { L_PA.tipo = E.tipo ; }
| L_PA$_1$, E { L_PA.tipo = L_PA$_1$.tipo x E.tipo) ; }

(6/6)

/**** Algunas instrucciones ****/

I → id **=** E **;** { *Si* ***NOT ObtTds****(id.nom, id.tipo)* ***OR*** *id.tipo ≠ E.tipo*

ent { yyerror(); I.tipo = terror ;}

sino I.tipo = id.tipo ; }

| **while** (E) I { *Si E.tipo ≠ tlogico ent { yyerror(); I.tipo = terror ; }*

sino I.tipo = tvacio ; }

| **{** L_Inst **}**

L_Inst → L_Inst I

| ε

Ejercicio

S → **id :=** E

| S ; S

| **for (** S ; E ; S **)** S

E → E ≺ E

| E # E

| **id**

ETDS que realice la comprobación de tipos. Los operadores ≺ y # son sobrecargados.

≺ Operador de orden que puede tener operandos enteros y booleanos.

Suma de enteros, el 'o' lógico o la concatenación de cadenas de caracteres, según los operandos sean enteros, booleanos o cadenas respectivamente.

La tabla de símbolos se supone iniciada con los tipos de cada identificador.

En ' for (S1 ; E ; S2) S3 ' las ocurrencias del terminal S1 y de S2, deben contener alguna instrucción de asignación y el no terminal E debe reescribirse de forma que contenga algún identificador que ocurra en la parte izquierda de alguna asignación que aparezca en S1 o en S2.

Solución ejercicio

$S \rightarrow$ **id :=** E	S.ident := { id.nom }
	Si BuscaTipo(id.nom) <> E.tipo **ent** yyerror();
| S_1 **,** S_2	S.ident := S_1.ident $\cup$ S_2.ident;
| **for (** S_1 **;** E **;** S_2 **)** S	**Si** S_1.ident=$\varnothing$ **or** S_2.ident=$\varnothing$ **ent** yyerror()
	sino si E.ident $\cap$ (S_1.ident $\cup$ S_2.ident) = $\varnothing$ **ent** yyerror()
	sino si E.tipo <> tlogico **ent** yyerror() ;
	S.ident := $\varnothing$;
$E \rightarrow E_1 \prec E_2$	**Si** E_1.tipo <> E_2.tipo **or** (**not** E_1.tipo **in** [tentero, tlogico])
	ent yyerror() ; E.tipo := terror
	sino E.tipo := E_1.tipo ; E.ident := E_1.ident $\cup$ E_2.ident ;
| $E_1 \oplus E_2$	**Si** E_1.tipo <> E_2.tipo **or** (**not** E_1.tipo **in** [tentero, tlogico, tcad])
	ent yyerror() ; E.tipo := terror
	sino E.tipo := E_1.tipo ; E.ident := E_1.ident $\cup$ E_2.ident ;
| **id**	E.tipo := BuscaTipo (id.nom) ; E.ident := { id.nom }

Ejercicio

Dada la siguiente gramática para definir objetos en 2 y 3 dimensiones, donde un punto (*P*) está representado por números (*num*) separados por comas:

S → **space** E **begin** LO **end**

E → **2D** | **3D**

LO → **line (** LP **)** LO | **square (** LP **)** LO | ε

LP → P | LP **:** P

P → **num** | P **,** **num**

Ejemplo:

```
space 2D
  begin
    line (5, 2 : 9, 3)
  end
```

Escribe un ETDS que realice las siguientes comprobaciones semánticas:

a) Si el espacio es *2D*, todos los puntos deben estar definidos por 2 números, si es *3D* por 3 números.

b) Los objetos *line* están definidos por 2 puntos y los objetos *square* por 3 puntos.

S → space E begin LO end	{ LO.spc = E.spc ; }
E → 2D	{ E.spc = 2 ; }
| 3D	{ E.spc = 3 ; }
LO → line (LP) LO_1	{ LP.spc = LO.spc ; LO_1.spc = LO.spc ; } { if (LP.pto != 2) yyerror("Linea mal definida"); }
| square (LP) LO_1	{ LP.spc = LO.spc ; LO_1.spc = LO.spc ; } { if (LP.pto != 3) yyerror("Cuadrado mal definido"); }
| ε	
LP → P	{ LP.pto =1 ; if (P.num != LP.spc) yyerror("Punto mal definido"); }
| LP_1 : P	{ LP_1.spc = LP.spc ;} { LP.pto = LP_1.pto + 1 ; if (P.num != LP.spc) yyerror("Punto mal definido"); }
P → num	{ P.num = 1 ; }
| P_1 , num	{ P.num = P_1.num + 1 ; }