Comprobaciones semánticas estáticas

Pueden realizarse en tiempo de compilación y no dependen de un contexto de ejecución determinado.

- Comprobación del ámbito de las variables. Control del uso de las variables
- Comprobaciones de tipo. Comprobación de la compatibilidad de operadores y operandos
- Comprobación de la unicidad de las variables. Las variables deben declararse solo una vez (en la misma región o bloque).
- Comprobación de la declaración de las variables. Una variable debe declararse (implícita o explícitamente) antes de usarse.
- Comprobaciones en los argumentos. Los métodos deben llamarse con el número y el tipo de los argumentos correcto.
- Otras. Comprobación del flujo de control, relaciones de herencia, comprobación de la unicidad de la clases y de los métodos en las clases, etc.

Comprobaciones semánticas dinámicas

Dependen de un determinado contexto de ejecución.

- Verificación del estado del "stack" y del "heap"
- Verificación de posibles "overflow" y "underflow"
- Verificación de posibles divisiones por cero
- Verificación de direcciones e índices en variables indexadas
- ...

Los requerimientos dependen del Lenguaje

Ámbito (alcance) de las variables

Define el segmento de programa en el que una variable es accesible.
Relaciona la declaración de una variable con su uso.

- Si el ámbito de la variable está completamente determinada por su posición en el programa, entonces se denomina Lenguaje de ámbito estático
  - ⇒ La mayor parte de los lenguajes son de ámbito estático.
- Si, por el contrario, el ámbito de la variable depende del estado durante la ejecución del programa, entonces se denomina Lenguaje de ámbito dinámico
  - ⇒ Solo unos pocos lenguajes tienen ámbito dinámico (Lisp, SNOBOL)
    (Lisp ha cambiado a un ámbito principalmente estático)

- En muchos lenguajes, un mismo nombre puede identificar objetos diferentes en distintas partes de un programa.
  (los diferentes ámbitos de un mismo nombre no se pueden solapar)
- La mayor parte de los lenguajes de ámbito estático (incluyendo `C`, `C++`, o `Java`) permiten anidamiento de declaraciones.
  (las declaraciones internas "ocultan" las externas)
- Los métodos no necesitan definirse en la clase que se utilizan, pero si en alguna clase antecesora. Los métodos pueden ser redefinidos: En `Java` o `C++` (no en `C` o `Python`), se puede utilizar el mismo nombre para más de un método.
- Declaraciones explícitas o implícitas
  - `Java` y `C++` requieren declaraciones explícitas; `C` es más "indulgente".
  - `Python` permite declaraciones implícitas.
  - `Fortran` declara implícitamente todas las variables usadas.
  - En lo que respecta al compilador, siempre debe haber una declaración.

EJEMPLO DE LEB: C

```
main()
{
.    int a = 0;
.    int b = 0;
.    {
.    .    int b = 1;
.    .    {
.    .    .  int a = 2;
.    .   B2  printf(``%d %d\n'', a, b);
.    .    .
B0   .    }
.   B1    {
.    .    .  int b = 3;
.    .   B3  printf(``%d %d\n'', a, b);
.    .    .
.    .    }
.    .    printf(``%d %d\n'', a, b);
.    }
.    printf(``%d %d\n'', a, b);
}
```


COMPROBACIONES DE TIPO

- **Tipos** Un tipo es un conjunto de valores junto con un conjunto de operaciones sobre dichos valores.

  Las clases son una instanciación de esta moderna noción de tipo.

- **Sistema de tipos** El sistema de tipos de un *Lenguaje de Programación* especifica que operaciones son válidas para cada tipo

  El objetivo de la comprobación de tipos es asegurar que las operaciones se utilizan con los tipos correctos

- **Tipos y operaciones** La mayoría de las operaciones son legales solo para los valores de algunos tipos
  - En `C` no tiene sentido añadir un puntero de función a un entero
  - Tiene sentido sumar dos números enteros
  - Sin embargo, ¡ ambos tienen la misma implementación en ensamblador !

COMPROBACIONES DE TIPO

- **Utilidad de los sistemas de tipos**
  - Detección de errores
    - Errores de memoria, como la de utilizar un número entero como un puntero.
    - Violaciones de los límites de la abstracción, como el uso de un campo privado desde fuera de una clase
  - Ayuda a la compilación
    - En `Python` (p.ej. en $x + y$) el sistema de tipos aporta muy poca información respecto a los tipos de x e y, por lo que el código debe ser general
    - En `C`, `C++` y `Java`, el código para $x + y$ será más pequeño y eficiente debido a que las representaciones son conocidas.

COMPROBACIONES DE TIPO

- **Sitema de Tipos**
  - Sistema de tipos estático.- Toda o la mayor parte de la comprobación de tipos se realiza en tiempo de compilación (`C`, `Java`, `C++`). El sistema de tipos estáticos es generalmente muy rico.
  - Sistema de tipos dinámico.- Casi toda la comprobación de tipos se realiza en tiempo de ejecución (`Scheme`, `Python`, `Ruby`). El sistema de tipos estáticos es generalmente poco importante.
  - Sin tipos.- Sin comprobación de tipos (código máquina).

➢ La "Guerra de tipos"

Conflicto de puntos de vista entre sistemas de tipos estático y dinámico.

➢ Los que proponen sistemas de tipos dinámicos

- Los sistemas de tipos estático son restrictivos. En general requieren más trabajo para hacer cosas razonables.
- El prototipado rápido es complicado en el marco de un sistema de tipos estático.

➢ Los que proponen sistemas de tipos estáticos

- La comprobación de tipos estática captura la mayor parte de los errores del programa en tiempo de compilación.
- Evita la sobrecarga de cálculo en la comprobaciones de tipo en tiempo de ejecución.

➢ Compatibilidad (equivalencia) de tipos

➢ **Nominal**. Cada nombre de tipo representa un tipo distinto (`Java`, `Ada`, `C#`, ...)

➢ **Estructural**. Dos tipos son equivalente si tienen la misma estructura, después de sustituir todos los nombres de tipo por las expresiones de tipo que representan (`Algol`, `Modula`, `C`, `Pascal`, ...)

```
typedef struct {              typedef struct {
  int  id;                      int   identificador;
  char nombre [50];             char descripcion [50];
  } cliente;                    } producto;

cliente    e1, e2;
producto  c1, c2;
```

Equivalencia nominal : `e1` ≡ `e2` y `c1` ≡ `c2`;

Equivalencia estructural: `e1` ≡ `e2` ≡ `c1` ≡ `c2`

➢ Conversión de tipos Se permite un tipo en un contexto donde se esperaba otro.

```
int  a, c;   short b;

      a = b * c
```

➢ **Implícita (coerción)** introducida por el compilador

➢ **Explícita** introducida por el programador en el programa fuente (casting).

Sistema		Usuario	
`a = b;`	✔	`a = (int) b;`	✔
`b = a;`	✘	`b = (short) a;`	✔ (warning)

➢ Sobrecarga de operadores (y funciones)

Un mismo nombre de operador (o función) tiene distintos significados.

Ejemplo

En `Java` el operador + denota el operador de concatenación de cadenas o el operador de suma. En `C++` se permite al usuario implementar operadores sobrecargados, en `Java`, no.

También se puede sobrecargar una función:

```
void  alfa(float a, float b) { ... }
void  alfa(int a, int b) { ... }
```

La sobrecarga se resuelve en tiempo de compilación

COMPROBACIONES DE TIPO

➢ Polimorfismo

Se denomina a cualquier fragmento de código que se puede ejecutar con argumentos de diferentes tipos.

Ejemplo

Dado un segmento de código en el lenguaje ML [Aho et al.2007]

```
fun length (x) =
  if null(x) then 0 else length(tail(x)) + 1;

length([``sun'', ``non'', ``true'']) + length([10,9,8,7])
```

El polimorfismo se resuelve en tiempo de ejecución

COMPROBACIONES (ESTÁTICAS) DE TIPO

⇒ Representación: **Sistema de Tipos**

Sistema de tipos.- Es una representación formal de los tipos de un lenguaje. Asigna *expresiones de tipo* a las distintas partes de un programa y define la equivalencia y la compatibiliad de tipos así como la inferencia de tipos.

Expresión de tipo.- Una Expresión de tipo es un tipo básico o se forma mediante un *constructor de tipos* aplicado a otras expresiones de tipo.

⇒ Interpretación: **ETDS que implementa el Sistema de Tipos**

- ➢ Determinar el tipo de objetos y expresiones ⇒ Inferencia de Tipos
- ➢ Comprobar el tipo de las operaciones ⇒ Comprobación de Tipos

COMPROBACIONES (ESTÁTICAS) DE TIPO: EJEMPLO

	Inferencia de de tipos	Comprobación de tipos
tipos básicos	$int \quad x$	$z = x + y$
punteros	$char \quad *p$	$z = *p$
vectores	$int \quad A\,[\,27\,]$	$z = A[i]$
registros	$struct\,\{\,int\,c_1;\ char\,c_2;\ int\,c_3\,\}\,R$	$z = R.c_1$
funciones	$int \quad F\,(\,int\,p_1,\ char\,p_2,\ int\,p_3)$	$z = F\,(a,\ b,\ c)$

CASO DE ESTUDIO: SISTEMA DE TIPOS

➢ **Expresiones de Tipo**.- Una Expresión de Tipo (ET), se define como:

1.– un tipo básico: *tcarácter*, *tentero*, *treal*, *tlógico*, ..., *terror* y *tvacio*,

2.– el nombre de una ET,

3.– un constructor de tipos aplicado a una ET:

- ➢ punteros: $tpuntero(\,T\,)$, donde T es una ET;
- ➢ vectores: $tvector(\,I,\,T\,)$, donde I denota la información de los índices y T es una ET,
- ➢ registros: $tregistro(\,(N_1 \times T_1) \times \ldots \times (N_k \times T_k)\,)$, donde $N_1, \ldots, N_k$ son nombres y $T_1, \ldots, T_k$ son ET.
- ➢ funciones: $tfunción(\,D,\,R\,)$, donde D (dominio) y R (rango) son ET,

  Dominio $D \equiv (\,t_{p_1} \times t_{p_2} \times \ldots \times t_{p_n}\,)$ $\quad t_{p_i}$ tipo del i-ésimo parámetro

CASO DE ESTUDIO: ETDS COMPROBACIÓN DE TIPOS 1/4

Declaración de objetos

P ⇒ LD	
LD ⇒ LD D	
⇒ D	
D ⇒ DV ;	**insTdS**(DV.n, "variable-global", DV.t);
DV ⇒ T id	DV.n = id.n; DV.t = T.t;
⇒ T * id	DV.n = id.n; DV.t = `tpuntero`(T.t);
⇒ T id [cte]	SI ¬[cte.t = `tentero` ∧ cte.num > 0] MenError(.) SINO DV.n = id.n; DV.t = `tvector`(cte.num, T.t);
T ⇒ char	T.t = `tcarácter`;
⇒ int	T.t = `tentero`;
⇒ `float`	T.t = treal;
⇒ `bool`	T.t = tlógico;

insTdS: Inserta en la TDS toda la información de un objeto. **MenError**: Genera un cierto mensaje de error.
P: Programa; **LD**: Lista de Declaraciones; **D**: Declaración; **DV**: Declaración de variables; **T**: Tipo;

CASO DE ESTUDIO: ETDS COMPROBACIÓN DE TIPOS 2/4

Declaración de objetos (cont.)

T ⇒ struct { LC }	T.t = `tregistro`(LC.t)
LC ⇒ LC DV ;	LC.t = LC'.t ⊗ (DV.n ⊗ DV.t);
⇒ DV ;	LC.t = (DV.n ⊗ DV.t);
D ⇒ DV ;	**insTdS**(DV.n, "variable-global", DV.t);
⇒ T id (PF) { DL LI }	**insTdS**(id.n, "función", `tfunción`(PF.t, T.t));
DL ⇒ DL DV ;	**insTdS**(DV.n, "variable-local", DV.t);
⇒ ϵ	
PF ⇒ ϵ	PF.t = `tvacio`;
⇒ LF	PF.t = LF.t;
LF ⇒ DV , LF	LF.t = DV.t ⊗ LF'.t; **insTdS**(DV.n, "parámetro", DV.t);
⇒ DV	LF.t = DV.t; **insTdS**(DV.n, "parámetro", DV.t);

PF: Parámetros Formales; **DL**: Declaraciones Locales; **LI**: Lista de Instrucciones;
LC: Lista de Campos; **LF**: Lista de parámetros Formales.

CASO DE ESTUDIO: ETDS COMPROBACIÓN DE TIPOS 3/4

Expresiones

E ⇒ E mod E	SI ¬[E_1.t = E_2.t = `tentero`] { E.t = `terror`; MenError(.); } SINO E.t = `tentero`;
⇒ cte	E.t = cte.t;
⇒ id	SI ¬[**obtTdS**(id.n, id.t)] { E.t = `terror`; MenError(.); } SINO E.t = id.t;
⇒ * id	SI ¬[**obtTdS**(id.n, id.t) ∧ (id.t = `tpuntero`(tap.t))] { E.t = `terror`; MenError(.); } SINO E.t = tap.t;
⇒ id [E]	SI ¬[**obtTdS**(id.n, id.t) ∧ (id.t = `tvector`(id.nel, id.tel)) ∧ (E_1.t = `tentero`)] { E.t = `terror`; MenError(.); } SINO E.t = id.tel;
⇒ id . id	SI ¬[**obtTdS**(id_1.n, id_1.t) ∧ (id_1.t = `tregistro`(id_1.lc) ∧ **obtCampo**(id_1.lc, id_2.n, id_2.t)] { E.t = `terror`; MenError(.); } SINO E.t = id_2.t;

obtTdS: Función que obtiene la información de un objeto, devuelve el valor *falso* si el objeto no esta en la TDS.
obtCampo: Función que obtiene el tipo asociado al nombre de un elemento del *struct*, devuelve el valor *falso* si el nombre del elemento no existe. **E**: Expresión;

CASO DE ESTUDIO: ETDS COMPROBACIÓN DE TIPOS 4/4

Expresiones (cont.)

E ⇒ id (PA)	SI ¬[**obtTdS**(id.n, id.t) ∧ (id.t = `tfunción`(id.td, id.tr)) ∧ (id.td = PA.t)] { E.t = `terror`; MenError(.); } SINO E.t = id.tr;
PA ⇒ ϵ	PA.t = `tvacio`;
⇒ LA	PA.t = LA.t;
LA ⇒ E , LA	LA.t = E.t ⊗ LA'.t;
⇒ E	LA.t = E.t;

Instrucciones

I ⇒ id = E ;	SI ¬[**obtTdS**(id.n, id.t) ∧ (id.t = E.t)] MenError(.);
⇒ * id = E ;	SI ¬[**obtTdS**(id.n, id.t) ∧ (id.t = `tpuntero`(id.tap)) ∧ (id.tap = E.t)] MenError(.);
⇒ id [E] = E ;	SI ¬[**obtTdS**(id.n, id.t) ∧ (id.t = `tvector`(id.nel, id.tel)) ∧ (E_1.t = `tentero`) ∧ (id.tel = E_2.t)] { MenError(.); }
⇒ id . id = E ;	SI ¬[**obtTdS**(id_1.n, id_1.t) ∧ (id_1.t = `tregistro`(id_1.lc)) ∧ **obtCampo**(id_1.lc, id_2.n, id_2.t) ∧ (id_2.t = E.t)] MenError(.);
⇒ while (E) I	SI (E.t ≠ `tlógico`) MenError(.);

PA: Parámetros Actuales; **LA**: Lista de parámetros Actuales; **I**: Instrucciones.