

Type equivalence
+
evoluzione dei sistemi di tipi
+
**classificazione dei linguaggi di
programmazione**

Ritorno al passato

- All'inizio esistevano solo i tipi elementari
 - ◆ E nessuna gerarchia di classi...
- Poi sono stati introdotti i tipi user-defined
 - ◆ Ancora niente classi
 - ◆ Inizialmente semplici ridenominazioni di tipi elementari oppure nomi di record
- Un assegnamento $x = y$ (o $x := y$) o un passaggio di parametri quando era valido?
- Lo stabilisce la nozione di *type equivalence* adottata dal linguaggio

Forme di equivalenza

■ Name equivalence

- ◆ I tipi di x e y devono avere lo stesso nome
- ◆ cioè essere lo stesso tipo

■ Structural equivalence

- ◆ I tipi di x e y devono avere la stessa rappresentazione interna
- ◆ Apparentemente più snello e flessibile, in realtà aumenta la possibilità di errori

```
Euro x;  
Dollar y;  
z = x+y ;    // che senso ha?
```

Esempi

- Pascal: name equivalence
- C (e C++): entrambe!
 - ◆ Quasi sempre structural tranne che per le struct

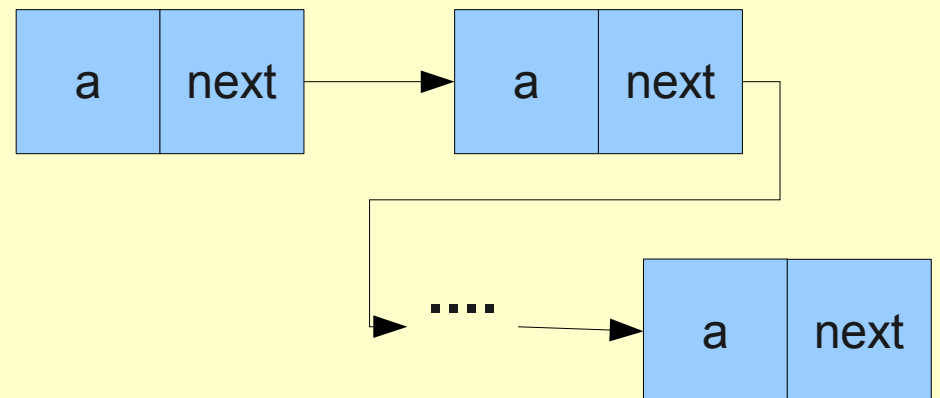
```
typedef int money;  
typedef int apples;  
  
typedef struct{ int a; } S1;  
typedef struct{ int a; } S2;  
  
int main(){  
    money x=0;  
    apples y=0;  
    int z = x+y; // non fa una piega  
    S1 a;  
    S2 b;  
    a = b; // questo invece non lo compila
```

Ecco perchè

- Verificare se due struct sono strutturalmente equivalenti richiede di verificare una proprietà chiamata *bisimulazione*
- Caso semplice: sia S1 che S2 generano tutte le catene come quella a destra

```
struct S1 { int a; struct S1* next; };
```

```
struct S2 { int a; struct S3* next; };  
struct S3 { int a; struct S2* next; };
```



- Complicazioni in presenza di record varianti

Compatibilità di tipi

- Con l'avvento dei linguaggi a oggetti l'equivalenza viene rimpiazzata da *compatibilità*
 - ◆ Qualunque sottotipo di T è compatibile con T
- Nei linguaggi O.O. più comuni la compatibilità è basata su *nome* piuttosto che *struttura*
 - ◆ Due classi con nomi diversi sono diverse
 - ◆ Anche se hanno gli stessi attributi e gli stessi metodi
 - ◆ Inoltre una classe per essere sottotipo di un'altra deve essere *esplicitamente* dichiarata tale (keyword *extends*)
 - La struttura non conta neanche in questo caso

Evoluzione dei sistemi di tipi

- Tipi di dato elementari
- Tipi user-defined (ad es. Pascal, C)
 - ◆ Solo strutture dati
- Interfacce e tipi di dato astratti (ad es. Modula, Ada)
 - ◆ Encapsulation: strutture dati accessibili solo attraverso specifiche procedure
 - ◆ Modificatori di accesso
 - ◆ Disaccoppiamento interfaccia/implementazione
- Gerarchie di tipi
 - ◆ Ed ecco finalmente i linguaggi O.O.

Caratteristiche utili alla classificazione dei linguaggi

- ◆ Paradigma di riferimento
 - imperativo, O.O, funzionale, logico
- ◆ Scoping (statico/dinamico)
- ◆ Gestione della memoria
 - Allocazione dinamica, garbage collection/esplicita, ...
- ◆ Sistema di tipi
 - strong/weak, encapsulation, equivalence/compatibility, polimorfismo (di 4 tipi), *type inference*...
- ◆ Supporto alle eccezioni
 - *Eventualmente integrato con type checking... vedi ML*
- ◆ Supporto al parallelismo
 - Memoria condivisa (*synchronized*), scambio di messaggi (nel senso *RMI*), gestione del nondeterminismo, fairness

Caratteristiche utili alla classificazione dei linguaggi

- Naturalmente determinano come si usa al meglio un dato linguaggio
- E quali sono gli errori da evitare
 - ◆ Ad es. sapendo cosa il compilatore può o non può fare per voi