



NOMI, BINDING

Nomi



- ✎ Un nome in un linguaggio di programmazione è esattamente quello che immaginate
 - La maggior parte dei nomi sono definiti dal programma (gli identificatori)
 - Ma anche i simboli speciali (come '+' o 'sqrt') sono nomi
- ✎ Un nome è una sequenza di caratteri utilizzata per denotare simbolicamente un oggetto

Nomi e Astrazioni



- L'uso dei nomi realizza un primo meccanismo elementare di astrazione
- **Astrazione sui dati**
 - ✓ La dichiarazione di una variabile permette di introdurre un nome simbolico per una locazione di memoria astrando sui dettagli della gestione della memoria
- **Astrazione sul controllo**
 - ✓ La dichiarazione del nome di una funzione (procedura) è l'aspetto essenziale nel processo di astrazione procedurale

Entità denotabili



- ✎ Entità denotabili: elementi di un linguaggio di programmazione a cui posso assegnare un nome
- ✎ Entità i cui nomi sono **definiti dal linguaggio** di programmazione (tipi primitivi, operazioni primitive, ...)
- ✎ Entità i cui nomi sono **definiti dall'utente** (variabili, parametri, procedure, tipi, costanti simboliche, classi, oggetti...)

Binding e scope



- ✎ Un *binding* è una associazione tra un nome e un oggetto
- ✎ Lo *scope* di un binding definisce quella parte del programma in cui il binding è attivo

Binding time



- ✎ Il **Binding Time** definisce il momento temporale in cui viene definita una associazione
- ✎ Più in generale il binding time definisce il momento in cui vengono prese le decisioni relative alla gestione delle associazioni

Esempi di binding time



- **Language design time** – progettazione del linguaggio
 - ✓ Binding delle operazioni primitive, tipi, costanti
- **Program writing time** – tempo di scrittura del programma
 - ✓ Binding dei sottoprogrammi, classi, ...
- **Compile time** – tempo di compilazione
 - ✓ Associazioni per le variabili globali
- **Run time** – tempo di esecuzione
 - ✓ Associazioni tra variabili e locazioni, associazioni per le entità dinamiche

Domanda



- ☞ In Java, qual'è il binding time per l'associazione tra il nome di un metodo e il codice effettivo del metodo?
- ☞ Program time?
- ☞ Compile time?
- ☞ Run time?

Statico & Dinamico



- ✎ Il termine **static (dynamic) binding** è solitamente utilizzato per fare riferimento ad una associazione attivata prima di mandare (dopo aver mandato) il programma in esecuzione
- ✎ Molte delle caratteristiche dei linguaggi di programmazione dipendono dalla scelta del binding time statico o dinamico
- ✎ I linguaggi “compilati” cercano di risolvere il binding staticamente
- ✎ I linguaggi “interpretati” devono risolvere il binding dinamicamente

Ambiente

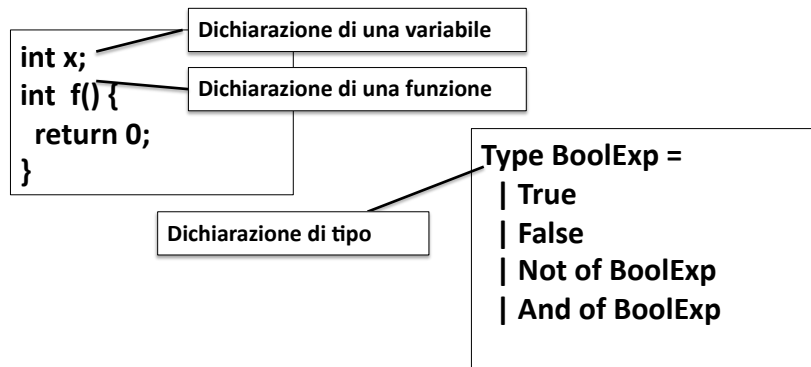


- ✎ L'ambiente è definito come l'insieme delle associazioni nome-oggetto esistenti a runtime in uno specifico punto del programma ed in uno specifico momento dell'esecuzione
- ✎ ***Nella macchina astratta del linguaggio, per ogni nome e per ogni sezione del programma l'ambiente determina l'associazione corretta.***

Ambiente e Dichiarazioni



- Le dichiarazioni sono il costrutto linguistico che permette di introdurre associazioni nell'ambiente



Ambiente e dichiarazioni



```

{ int x;
  x = 22;
  { char x;
    x = 'a';
  }
}
  
```

Lo stesso nome, la variabile *x*, denota due oggetti differenti

Ambiente e dichiarazioni



```
Class A { ...}
```

```
A a1 = new A();  
A a2 = new A();  
:  
a1 = a2;
```

Aliasing: diversi nomi
per lo stesso oggetto

Blocchi



- ✦ Un blocco è una regione testuale del programma che può contenere dichiarazioni
 - C, Java {...}
 - OCAML let ... in
- ✦ Blocco associato a una procedura: corpo della procedura con le dichiarazioni dei parametri formali
- ✦ Blocco in-line: meccanismo per raggruppare comandi

Blocchi



```
A: { int aVar;
    aVar= 2;
    B: { char aVar;
        aVar= 'a'; }
    }
```

blocco in-line

```
apri blocco A;
apri blocco B;
chiudi blocco B;
chiudi blocco A;
```

Politica di accesso ai blocchi: LIFO

I cambiamenti dell'ambiente avvengono all'entrata e all'uscita dai blocchi (anche annidati)

Tipi di ambiente



- ✦ **Ambiente locale:** l'insieme delle associazioni dichiarate localmente, compreso le eventuali associazioni relative ai parametri
- ✦ **Ambiente non locale:** associazioni dei nomi che sono visibili all'interno del blocco ma non dichiarati nel blocco stesso
- ✦ **Ambiente globale:** associazioni per i nomi usabili da tutte le componenti che costituiscono il programma

Tipi di ambiente: esempio in C



```
#include <stdio.h>

int main(){
  A:{ int a =1 ;
    B:{ int b = 2;
      int c = 2;
      C:{ int c = 3;
        int d;
        d = a+b+c;
        printf("%d\n",d);
      }
      D:{ int e;
        e = a+b+c;
        printf("%d\n",e);
      }
    }
  }
}
```

Ambiente locale di C:
 associazioni per **c** e **d**:
 quella per **c** disattiva quella di **B**

Ambiente non locale per C:
 associazione per **b** ereditata da **B**
 associazione globale per **a**

Ambiente Globale:
 associazione per **a**

Cosa stampa?

Tipi di ambiente: esempio in Java



```
public class Prova{
  public static void main(String [] args){
    A:{ int a =1 ;
      B:{ int b = 2;
        int c = 2;
        C:{ int c = 3;
          int d;
          d = a+b+c;
          System.out.println(d);
        }
      D:{ int e;
        e = a+b+c;
        System.out.println(e);
      }
    }
  } } }
```

NB: in **Java** non è possibile ri-dichiarare una variabile già dichiarata in un blocco più esterno

```
$ javac Prova.java
Prova.java:7: c is already defined in main(java.lang.String[])
  C:{ int c = 3;
      ^
```



IMPLEMENTAZIONE DELL'AMBIENTE

Ambiente (env)



- ✎ Tipo (polimorfo) utilizzato nella semantica e nelle implementazioni per mantenere il binding tra nomi e valori di un opportuno tipo
- ✎ La specifica definisce il tipo come funzione
- ✎ L'implementazione che vedremo utilizza le liste

Ambiente: interfaccia



```
# module type ENV =
sig
  type 't env
  val emptyenv : 't -> 't env
  val bind : 't env * string * 't -> 't env
  val bindlist : 't env * (string list) * ('t list)
    -> 't env
  val applyenv : 't env * string -> 't
  exception WrongBindlist
end
```

21

Ambiente: semantica



```
# module Funenv:ENV =
struct
  type 't env = string -> 't
  exception WrongBindlist
  let emptyenv(x) = function (y: string) -> x
    (* x: valore default *)
  let applyenv(x,y) = x y
  let bind(r, l, e) =
    function lu -> if lu = l then e else applyenv(r,lu)
  let rec bindlist(r, il, el) = match (il,el) with
  | ([],[]) -> r
  | i::il1, e::el1 -> bindlist (bind(r, i, e), il1, el1)
  | _ -> raise WrongBindlist
end
```

22

Ambiente: implementazione



```
# module Listenv:ENV =
struct
  type 't env = (string * 't) list
  exception WrongBindlist
  let emptyenv(x) = [("",x)]
  let rec applyenv(x,y) = match x with
  | [(_,e)] -> e
  | (il,e1) :: x1 -> if y = il then e1
                    else applyenv(x1, y)
  | [] -> failwith("wrong env")
  let bind(r, l, e) = (l,e) :: r
  let rec bindlist(r, il, e1) = match (il,e1) with
  | ([],[]) -> r
  | i::il1, e::e11 -> bindlist (bind(r, i, e), il1, e11)
  | _ -> raise WrongBindlist
end
```

23

Scope



- ☞ Lo *scope* di un binding definisce quella parte del programma in cui il binding è attivo
 - **Scope statico o lessicale**: è determinato dalla struttura sintattica del programma
 - **Scope dinamico**: è determinato dalla struttura a tempo di esecuzione
- ☞ Come vedremo, differiscono solo per l'**ambiente non locale**

Operazioni su ambienti



Entrando in un blocco:

- creazione delle associazioni fra i nomi locali al blocco e gli oggetti denotati
- disattivazione delle associazioni per i nomi ridefiniti

Uscendo dal blocco:

- distruzione delle associazioni fra i nomi locali al blocco e gli oggetti denotati
- riattivazione delle associazioni per i nomi che erano stati ridefiniti

Operazioni su ambienti



- ✦ **Naming:** creazione di associazione fra nome e oggetto denotato (dichiarazione locale al blocco o parametro)
- ✦ **Referencing:** riferimento ad un oggetto denotato mediante il suo nome (uso del nome per accedere all'oggetto denotato)
- ✦ **Disattivazione** di associazione fra nome e oggetto denotato (la nuova associazione per un nome maschera la vecchia associazione, che rimane disattivata fino all'uscita dal blocco)
- ✦ **Riattivazione** di associazione fra nome e oggetto denotato (una vecchia associazione che era mascherata è riattivata all'uscita da un blocco)
- ✦ **Unnaming:** distruzione di associazione fra nome e oggetto denotato (es: ambiente locale all'uscita di un blocco)

Regole di Scope



```
A: { int x=0;
    void proc( ){x=1;}
    B: { int x;
        proc( );
    }
    printf("%d\n", x);
}
```

quale valore di x viene stampato?

Assumiamo **scoping statico**, quindi la variabile **x** nel corpo di **proc** è legata alla dichiarazione che la precede sintatticamente, cioè quella in **A**.

Regole di visibilità



- ✦ Una dichiarazione locale in un blocco è visibile:
 - in quel blocco
 - in tutti i blocchi in esso annidati
 - salvo ri-dichiarazioni dello stesso nome (mascheramento, *shadowing*)
- ✦ La regola di visibilità (cioè l'annidamento) è basata:
 - sul testo del programma (**scope statico**)
 - sul flusso di esecuzione (**scope dinamico**)

Scoping statico



- ✦ Le dichiarazioni locali in un blocco includono solo quelle presenti nel blocco, e non quelle dei blocchi in esso annidati
- ✦ Se si usa un nome in un blocco, l'associazione valida è quella locale al blocco; se non esiste, si prende la prima associazione valida a partire dal blocco immediatamente esterno, dal più vicino al più lontano; se non esiste, si considera l'ambiente predefinito del linguaggio; se non esiste, errore
- ✦ Se il blocco ha un nome, allora il nome fa parte dell'ambiente locale del blocco immediatamente esterno (come per le procedure)

Scoping statico: esempio



```

int main(){
  int x = 0;
  void proc(int n) {x = n+1;}
  proc(2);
  printf("%d\n",x);
  { int x = 0;
    proc(3);
    printf("%d\n",x);
  }
  printf("%d\n",x);
}

```

Cosa stampa?

_____ Stampa 3

_____ Stampa 0

_____ Stampa 4

Discussione



- ✦ Lo scope statico permette di determinare tutti gli ambienti di un programma staticamente, osservando la struttura sintattica del programma
 - controlli di correttezza a compile-time
 - ottimizzazione del codice a compile-time
 - possibile il controllo statico dei tipi

- ✦ Gestione a run-time articolata
 - gli ambienti non locali di funzioni e procedure evolvono diversamente dal flusso di attivazione e disattivazione dei blocchi

Il C



Il linguaggio C e' un linguaggio di programmazione che non prevede annidamento di procedure e funzioni:

I nomi delle funzioni (procedure) sono globali.

La ricorsione



In alcuni linguaggi una funzione deve essere dichiarata “lessicalmente” prima del suo uso.

Nel caso di funzioni mutualmente ricorsive questa regola comporta una separazione tra la dichiarazione e la definizione.

Scope dinamico



- ✦ L’associazione valida per un nome x , in un punto P di un programma, è la più recente associazione creata (in senso temporale) per x che sia ancora attiva quando il flusso di esecuzione arriva a P
- ✦ Come vedremo lo scope dinamico ha una gestione a run-time semplice
 - Vantaggi: flessibilità nei programmi
 - Svantaggi: difficile comprensione delle chiamate delle procedure e controllo dei tipi statico non è possibile



```
int N;  
void main() {N=1; A(); printf("main: ",N);}
```

```
void A()  
int N;  
{ N = 2; B(); printf("A: ", N); }
```

```
void B() { N = 3; }
```

Static scope: "A: 2 . main 3" (N in B e' globale)

Dynamic scope: "A 3 . main: 1" (N in B riferisce alla dichiarazione in A)



ALTRI ASPETTI

```

type lista = ptr^elemento;
type elemento = record
  informazione: intero;
  successivo: lista
end

```


errore: elemento è usato prima di essere definito; in Pascal si può fare, ma solo con i puntatori

```

type elemento;
type lista = access elemento;
type elemento is record
  informazione: intero;
  successivo: lista
end

```

in ADA si usano dichiarazioni incomplete



```

procedure pippo (A: integer); forward;

procedure pluto (B: integer)
begin
  pippo(3);
end

procedure pippo;
begin
  pluto(4)
end

```

in Pascal si usano definizioni incomplete per le funzioni mutuamente ricorsive; in C si può fare liberamente

