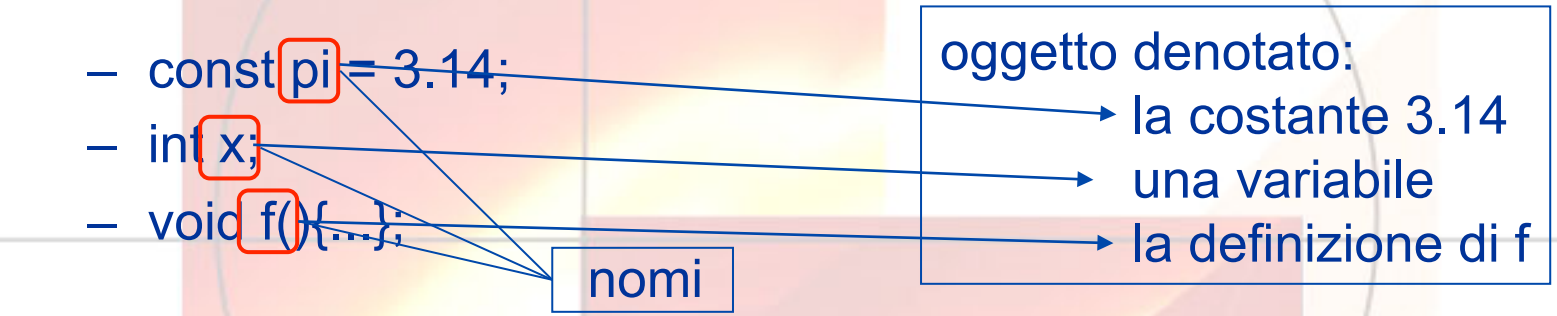
Nomi

nomi, ambiente, blocchi, regole di scope

M. Gabbrielli, S. Martini
Linguaggi di programmazione: principi e paradigmi
McGraw-Hill Italia, 2005

Nomi

- nome
  - sequenza di caratteri usata per *denotare* qualche cos'altro



- Nei linguaggi i nomi sono spesso identificatori (token alfa-numerici)
  - ma possono essere anche altro (+, := …)
- L'*uso* di un nome serve ad indicare l'oggetto denotato

- Oggetti simbolici più facili da ricordare
- Astrazione
  - dati (variabili, tipi ecc.)
  - controllo (sottoprogrammi)

Oggetti denotabili

- Oggetto *denotabile*
  - quando può essergli associato un nome
- Nomi definiti dall'utente
  - variabili, parametri formali, procedure (in senso lato), tipi definiti dall' utente, etichette, moduli, costanti definite dall' utente, eccezioni
- Nomi definiti dal linguaggio
  - tipi primitivi, operazioni primitive, costanti predefinite.
- Terminologia:
  - *Legame* (binding), o *associazione*, tra nome e oggetto

Binding time

- Statico
  - Progettazione del linguaggio
    - Tipi primitivi, nomi per operazioni e costanti predefinite ecc.
  - Scrittura del programma
    - definizione di alcuni nomi (variabili, funzioni ecc.) il cui legame sarà completato più tardi
  - Compilazione (+collegamento e caricamento)
    - legame di alcuni nomi (var globali)
- Dinamico
  - Esecuzione
    - legame definitivo di tutti i nomi non ancora legati (pe. variabili locali ai blocchi

Terminologia

In italiano usiamo:

- *binding* = legame = associazione
- *environment* = ambiente
- *scope* = portata, estensione (anche: ambito, campo d'azione)

- *lifetime* = vita, o tempo di vita

Ambiente

Ambiente:

insieme delle associazioni fra nomi e oggetti denotabili esistenti a run-time in uno specifico punto del programma ed in uno specifico momento dell'esecuzione

Dichiarazione:

meccanismo (implicito o esplicito) col quale si crea un'associazione in ambiente

```
int x;
int f (){
    return 0;
}
type T = int;
```

Ambiente, 2

Lo stesso nome può denotare oggetti distinti

in punti diversi del programma

Aliasing

nomi diversi denotano lo stesso oggetto

passaggio per riferimento

puntatori

ecc.

```
int *X, *Y;          // X,Y puntatori a interi
X = (int *) malloc (sizeof (int));
                     // allocata la memoria puntata
*X = 5;              // * dereferenzia
Y=X;                 // Y punta alla stesso oggetto di X
*Y=10;
write(*X);
```

Blocchi

- Nei linguaggi moderni l'ambiente è *strutturato*

- Blocco:
  - regione testuale del programma, identificata da un segnale di inizio ed uno di fine, che può contenere dichiarazioni *locali* a quella regione

    - `begin...end` — Algol, Pascal
    - `{...}` — C, Java
    - `(...)` — Lisp
    - `let...in...end` — ML

  - anonimo (o in-line)
  - associato ad una procedura

Perché i blocchi

- Gestione locale dei nomi

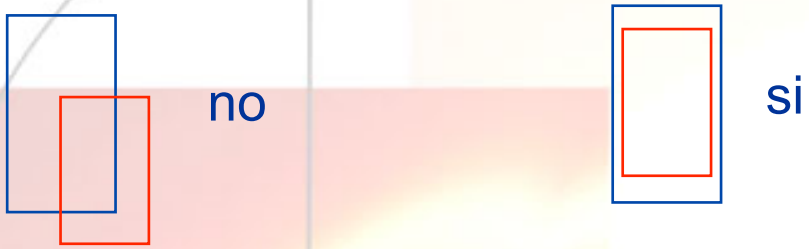
```
{int tmp = x;
 x=y;
 y=tmp
}
```

  - chiarezza
  - ognuno può scegliere il nome che vuole

- Con un'opportuna allocazione della memoria (=> vedi dopo):
  - ottimizzano l'occupazione di memoria
  - permettono la ricorsione

Annidamento

- Blocchi sovrapposti solo se annidati



- Regola di visibilità (preliminare)
  - Una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome (che *nasconde*, o *maschera*, la precedente).

Suddividiamo l'ambiente

- L'ambiente (in uno specifico blocco) può essere suddiviso in
  - ambiente locale: associazioni create all'ingresso nel blocco
    - variabili locali
    - parametri formali
  - ambiente non locale : associazioni ereditate da altri blocchi
  - ambiente globale: quella parte di ambiente non locale relativo alle associazioni comuni a tutti i blocchi
    - dichiarazioni esplicite di variabili globali
    - dichiarazioni del blocco più esterno
    - associazioni esportate da moduli ecc.

Esempio

```
A:{int a =1;

    B:{int b = 2;
       int c = 2;

       C:{int c =3;
          int d;
          d = a+b+c;
          write(d)
       }

       D:{int e;
          e = a+b+c;
          write(e)
       }
    }
}
```

Operazioni sull'ambiente

- Creazione associazione nome-oggetto denotato (naming)
  - dichiarazione locale in blocco
- Riferimento oggetto denotato mediante il suo nome (referencing)
  - uso di un nome
- Disattivazione associazione nome-oggetto denotato
  - una dichiarazione maschera un nome
- Riattivazione associazione nome-oggetto denotato
  - uscita da blocco con dichiarazione che maschera
- Distruzione associazione nome-oggetto denotato (unnaming)
  - uscita da blocco con dichiarazione locale

Operazioni sugli oggetti denotabili

- Creazione
- Accesso
- Modifica (se l'oggetto è modificabile)
- Distruzione

- Creazione e distruzione di un oggetto non coincidono con creazione e distruzione dei legami per esso

Alcuni eventi fondamentali

1. Creazione di un oggetto
2. Creazione di un legame per l'oggetto
3. Riferimento all'oggetto, tramite il legame
4. Disattivazione di un legame
5. Riattivazione di un legame
6. Distruzione di un legame
7. Distruzione di un oggetto

Il tempo tra 1 e 7 è la **vita** (o il tempo di vita: *lifetime*) **dell'oggetto**

Il tempo tra 2 e 6 è la **vita dell'associazione**

vita

La vita di un oggetto *non* coincide con la vita dei legami per quell'oggetto

- Vita dell'oggetto più lunga di quella del legame:

variabile passata ad una proc per riferimento (Pascal: var)

```
procedure P (var X:integer); begin… end;
…
var A:integer;
…
P(A);
```

Durante l'esecuzione di P esiste un legame tra X e un oggetto che esiste prima e dopo tale esecuzione.

vita, 2

- Vita dell'oggetto più breve di quella del legame:

area di memoria dinamica deallocata

```
int *X, *Y;
…
X = (int *) malloc (sizeof (int));
Y=X;
…
free (X);
X=null;
```

Dopo la `free` non esiste più l'oggetto, ma esiste ancora un legame ("pendente") per esso (`Y`): *dangling reference*

Regole di scope

- Come deve essere interpretata la regola di visibilità?

  Una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome (che nasconde, o maschera, la precedente)

  – in presenza di procedure

    cioè di blocchi che sono eseguiti in posizioni diverse dalla loro definizione

  – in presenza ambiente non locale (e non globale)

Qual è lo scope?

```
{int x=10;
void foo () {
    x++;
    }
void fie (){
    int x = 0;
    foo();
    }
fie();

}
```

quale x incrementa foo?

un riferimento non-locale in un blocco B può essere risolto:

nel blocco che *include sintatticamente* B

scope statico

nel blocco che è *eseguito immediatamente prima* di B

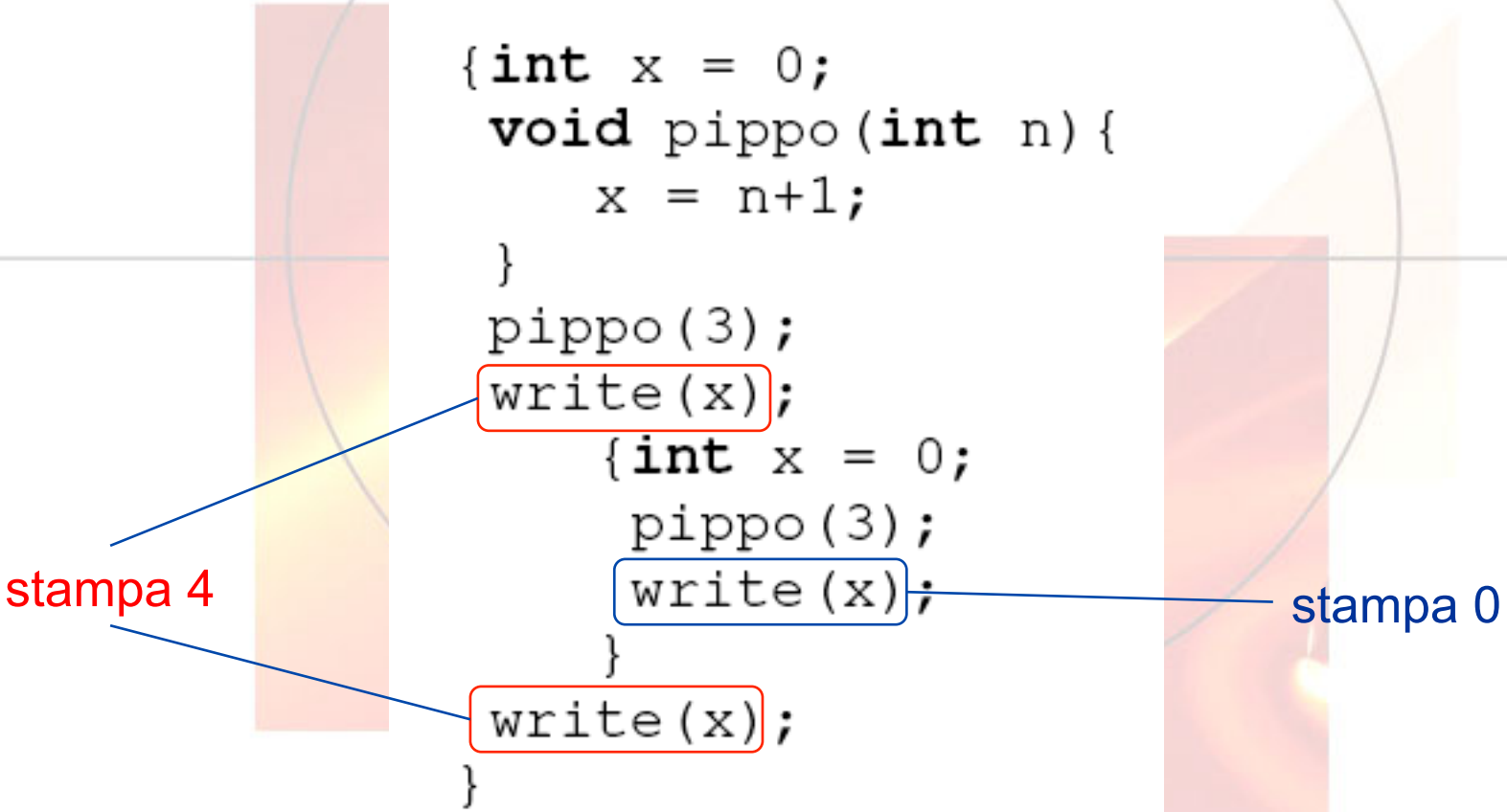
scope dinamico

Cosa stampa questo programma ?

```
{int x = 0;
 void pippo(int n){
     x = n+1;
 }
 pippo(3);
 write(x);
     {int x = 0;
      pippo(3);
      write(x);
     }
 write(x);
}
```

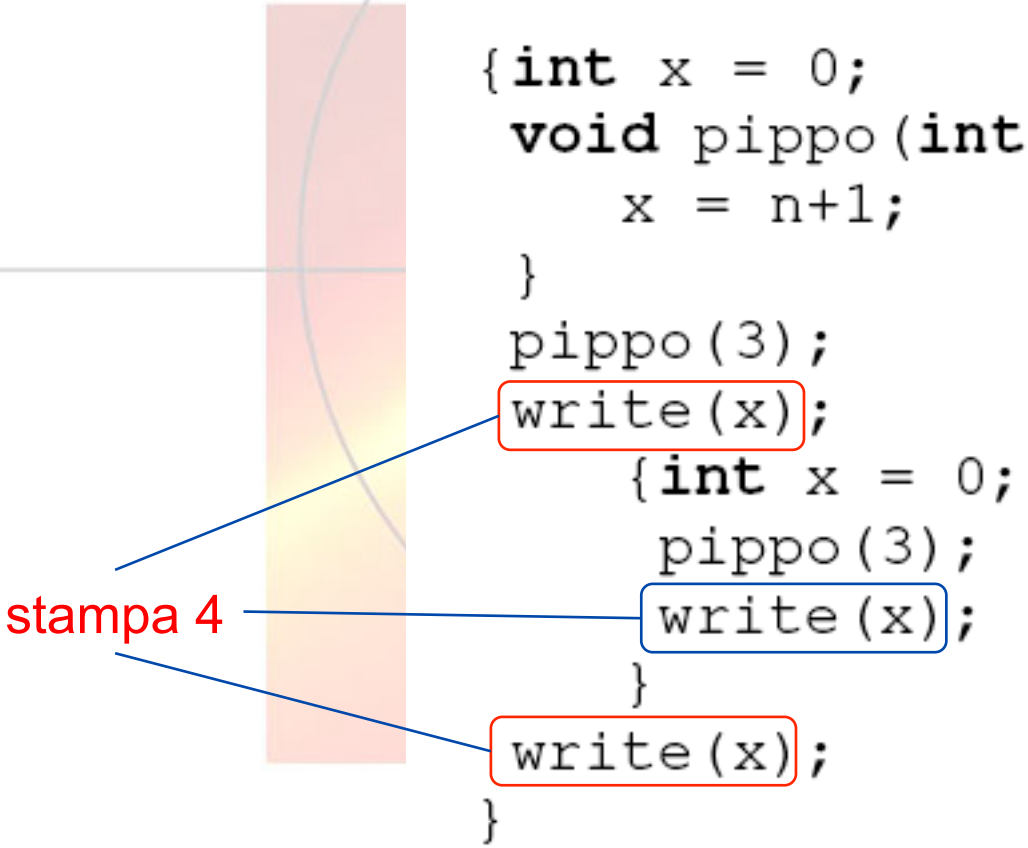
Scope statico

- Un nome non locale è risolto nel blocco che *testualmente* lo racchiude



Scope dinamico

- Un nome non locale è risolto nel blocco attivato *più di recente* e non ancora disattivato



Scope statico: independenza dalla posizione

```
{int x=10;
void foo () {
    x++;
    }
void fie (){
    int x=0;
    foo();
    }
fie();
foo();
}
```

- il corpo di **foo** è parte dello scope della x più esterna
- la chiamata di **foo** è compresa nello scope della x più interna
- **foo** può essere chiamata in molti contesti diversi
- l'unico modo in cui **foo** può essere compilata in modo univoco è che il riferimento a x sia sempre quello più esterno

La chiamata di **foo** interna a **fie** e quella nel main accedono alla stessa variabile: la x esterna

Scope statico: independenza dai nomi locali

```
{int x=10;
void foo () {
    x++;
    }
void fie (){
    int y =0;
    foo();
    }
fie();
foo();
}
```

la modifica del locale `x` in `y` dentro `fie`

- modifica la semantica del programma in scope dinamico
- non ha alcun effetto in scope statico

Un principio di indipendenza:
Ridenominazioni consistenti dei nomi locali di un programma non devono avere effetto sulla semantica del programma stesso

Scope dinamico: specializzare una funzione

- **`visualizza`** è una procedura che rende a colore sul video una certa forma

```
...
{var colore = rosso;
 visualizza(testo);
}
```

Scope statico vs dinamico

- Scope statico (*scoping statico, statically scoped, lexical scope*).
  - informazione completa dal testo del programma
  - le associazioni sono note a tempo di compilazione
  - dunque: principi di indipendenza
  - concettualmente più complesso da implementare ma più efficiente
  - Algol, Pascal, C, Java, ...

- Scope dinamico (*scoping dinamico, dynamically scoped*).
  - informazione derivata dall'esecuzione
  - spesso causa di programmi meno ``leggibili''
  - concettualmente più semplice da implementare, ma meno efficiente
  - Lisp (alcune versioni), Perl

- Differiscono solo in presenza congiunta di
  - ambiente non locale e non globale
  - procedure

Attenzione: C

- Algol, Pascal, Ada, Java permettono di annidare blocchi di sottoprogrammi

  non possibile in C:

  - funzioni definite solo nel blocco più esterno
  - dunque in una funzione l'ambiente è partizionato in locale e globale
  - non si presenta il problema dei non-locali

Questo non vuol dire che la regola di scoping (statico o dinamico) sia indifferente in C !

Significa solo che è semplice determinare dove risolvere un non-locale:
ogni non-locale viene risolto nell'ambiente globale
(ovvero: ci sono solo locali e globali)

```
int x=10;
void foo () {
   x++;
}
void fie(){
   int x =0;
   foo();
}
main(){
   fie();
   foo();
}
```

Attenzione: C

Una situazione di “scope dinamico” si realizza in C con le “macro”:

```
int x=10;
int next_x(int delta) { return x + delta; }
#define NEXT_X(delta) x+delta
main(){
   int x=5;
   printf("%d, %d\n", next_x(4), NEXT_X(4));
}
```

La definizione di C dice che **`define`** corrisponde ad una sostituzione testuale.

Ma poteva anche dire:
la semantica di **`define`** corrisponde ad una chiamata in scoping dinamico e lasciare poi all’implementazione come realizzarla

Determinare l'ambiente

- L'ambiente è dunque determinato da
  - regola di scope (statico o dinamico)
  - regole specifiche, p.e.
    - quando è visibile una dichiarazione nel blocco in cui compare?

discuteremo più avanti

  - regole per il passaggio dei parametri
  - regole di *binding* (shallow o deep)
    - intervengono quando una procedura P è passata come parametro ad un'altra procedura mediante il formale X

Alcune regole specifiche

- Dov'è visibile una dichiarazione all'interno del blocco in cui essa compare?
  - a partire dalla dichiarazione e fino alla fine del blocco
    - *Java: dichiarazione di una variabile*

```
{a=1;    // no!
 int a;
 …
}
```

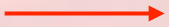
  - sempre (dunque anche *prima*) della dichiarazione
    - *Java: dichiarazione di un metodo*
      - permette metodi mutuamente ricorsivi

```
{void f(){
    g(); // si
 }
 void g(){
    f(); // si
 }
 …
}
```

E alcuni problemi

- Pascal
  - lo scope di una dichiarazione è l'intero blocco dove essa appare - eccetto i buchi
  - ogni identificatore deve essere dichiarato prima di essere usato

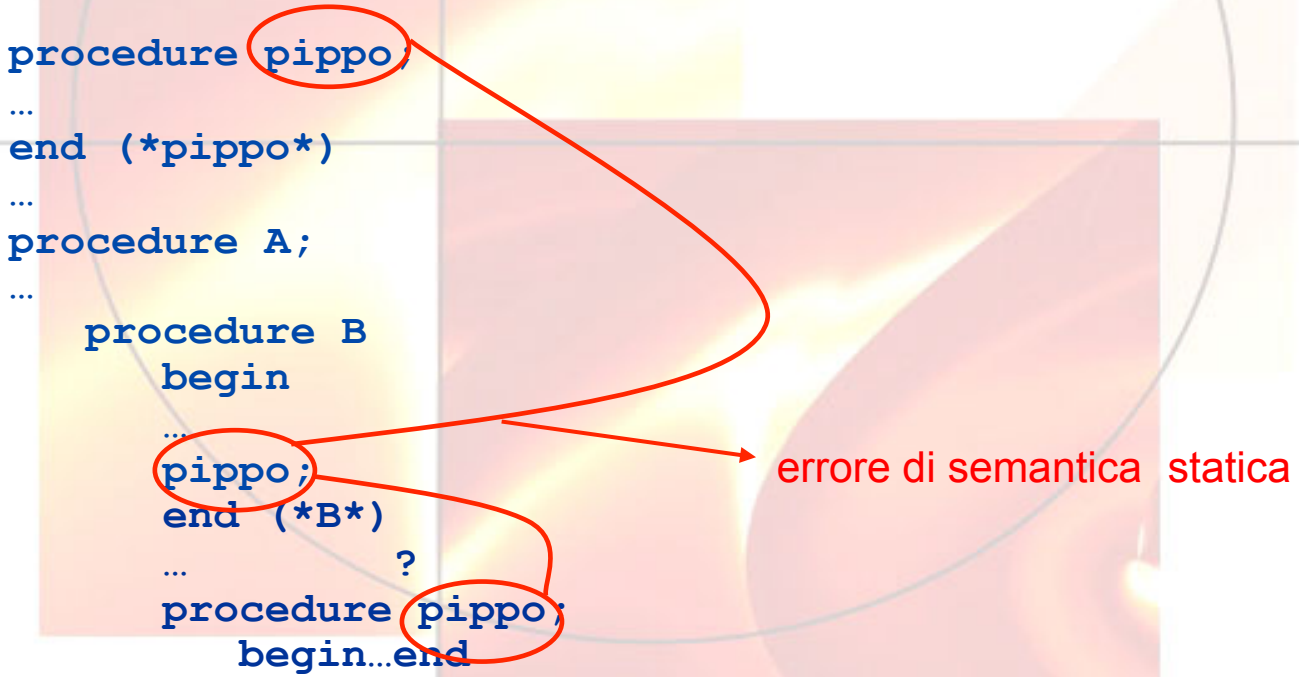
```
const a = -1;

procedure pippo;
   const b = a;
   …
   const a = 0;
```



errore di semantica statica
o in alcuni casi `b = -1` !

analogamente

- Pascal
  - lo scope di una dichiarazione è l'intero blocco dove essa appare - eccetto i buchi
  - ogni identificatore deve essere dichiarato prima di essere usato



Mutua ricorsione

Mutua ricorsione (funzioni, tipi) in linguaggi dove un nome deve essere dichiarato prima di essere usato?

- rilasciare tale vincolo per funzioni e/o tipi
  - Java per i metodi
  - Pascal per tipi puntatore

Java

```
{void f(){
    g();
 }
 void g(){
    f();
 }
}
```

Pascal

```
type lista = ^elem;
     elem = record
              info : integer;
              next : lista;
            end
```

- definizioni *incomplete*

Ada

```
type elem;
type lista is access elem;
type elem is record
             info: integer;
             next: lista;
           end
```

C

```
struct elem;
struct elem{
    int info;
    elem *next;
}
```