

Sottoprogrammi

- Premessa: \exists due fondamentali meccanismi di *astrazione* nei LP $\left\langle \begin{array}{l} \text{processo (prima)} \\ \text{dati (dopo)} \end{array} \right.$
- Sottoprogramma: promuove $\left\langle \begin{array}{l} \text{astrazione} \\ \text{riuso} \end{array} \right.$
- Caratteristiche generali $\left\langle \begin{array}{l} \text{unico entry point} \\ \text{chiamante sospeso durante l'esecuzione del chiamato} \\ \text{terminazione del subprog} \rightarrow \text{controllo torna al chiamante} \end{array} \right.$

Sottoprogrammi (ii)

- **Definizione** del subprog: descrive $\left\langle \begin{array}{l} \text{interfaccia} \\ \text{azioni} \end{array} \right\rangle$ dell'astrazione

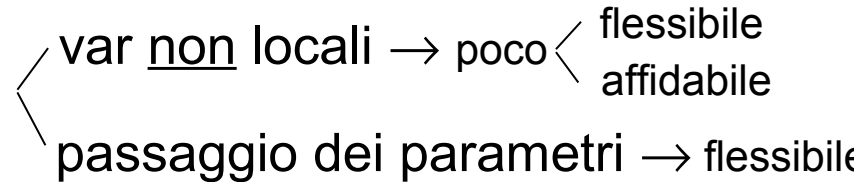
- **Chiamata** del subprog: richiesta di exec del subprog

- **Header**: intestazione del subprog $\left\{ \begin{array}{l} \text{SUBROUTINE SOMMA (parametri)} \\ \text{procedure SOMMA (parametri)} \\ \text{void somma (parametri)} \end{array} \right.$ FORTRAN
Ada
C

- **Profilo**: $\left\langle \begin{array}{l} \text{numero} \\ \text{ordine} \\ \text{tipo} \end{array} \right\rangle$ dei parametri formali

- **Protocollo**: profilo + tipo valore di ritorno (se \exists)

Sottoprogrammi (iii)

- \exists 2 modi di accesso ai dati da manipolare 
 - var non locali \rightarrow poco $\left\{ \begin{array}{l} \text{flessibile} \\ \text{affidabile} \end{array} \right.$
 - passaggio dei parametri \rightarrow flessibile

`max(a, b)`
- A volte, necessario trasferire computazione (invece che dati) \rightarrow nome del subprog
`integrale(f(x: real): real, x1: real, x2: real): real`
- Differenza parametri $\left\{ \begin{array}{l} \text{formali} \rightarrow \text{binding}(\text{formale}, \text{indirizzo}): \text{alla chiamata} \\ \text{attuali} \rightarrow \text{in generale, expr} \end{array} \right.$
- Corrispondenza $\left\{ \begin{array}{l} \text{formali} \\ \text{attuali} \end{array} \right. \left\{ \begin{array}{l} \text{implicita (posizionale)} \\ \text{esplicita (attuali in ordine qualsiasi)} \\ \text{mista} \end{array} \right.$
 - `SOMMA(LUNG => 10, LISTA => L, RIS => R)`
 - `SOMMA(10, RIS => R, LISTA => L)`
dopo: persa la corrispondenza posizionale

Sottoprogrammi (iv)

- Valori di default: quando non specificato il parametro attuale (Ada, FORTRAN 90, C++)

Ada

```
function STIPENDIO(LORDO: FLOAT; CLASSE: INTEGER := 1; ALIQUOTA: FLOAT) return FLOAT;  
STIP := STIPENDIO(2500.0, ALIQUOTA => 0.32);
```

implicito: CLASSE = 1

valori di default in coda

C++

```
float stipendio(float lordo, float aliquota, int classe = 1);  
stip = stipendio(2500.0, 0.32);
```

- LP senza parametri di default → corrispondenza isomorfa
Però: utilità dello scollamento (es: `printf` in C)

- Subprog = artificio per “estendere” il LP < **procedura**: nuova istruzione
funzione: nuovo operatore `A + abs(B)`

- Funzioni: non “pure” → pb effetti collaterali

Sottoprogrammi (v)

Scelte progettuali:

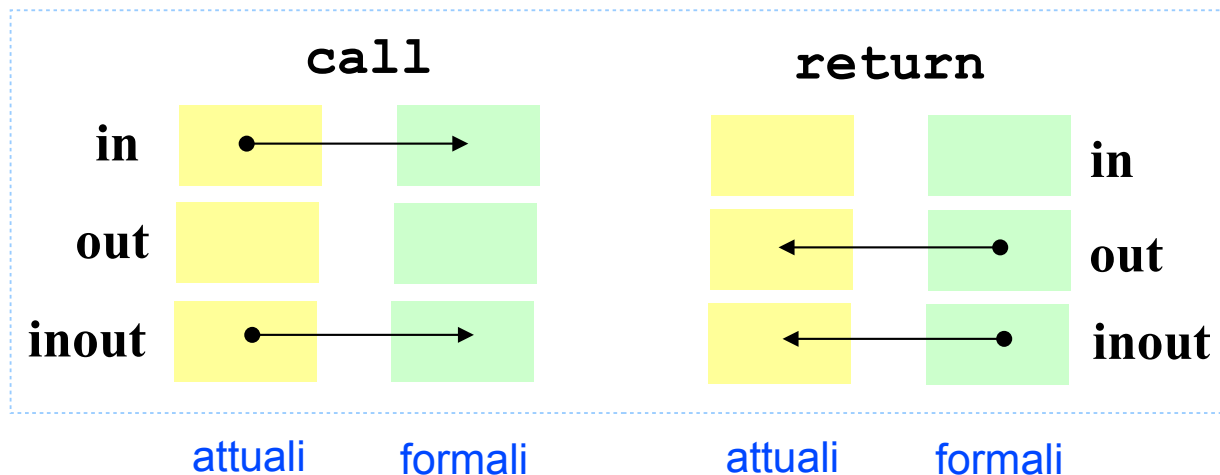
1. Quali metodi di passaggio dei parametri ?
2. \exists type checking dei parametri (attuali rispetto a formali) ?
3. Variabili locali allocate $\left\langle \begin{array}{l} \text{staticamente} \\ \text{dinamicamente} \end{array} \right\rangle$?
4. Se \exists passaggio di subprog S $\left\langle \begin{array}{l} \text{quale ambiente di referenziazione di S} \\ \exists \text{ type check. dei parametri nella chiamata di S} \end{array} \right\rangle$?
5. Possibile innestare definizioni di sottoprogrammi ?
6. Possibile overloading dei sottoprogrammi ?
7. Permessi effetti collaterali nelle funzioni ?
8. Quali tipi possono essere restituiti dalle funzioni ?

Passaggio dei Parametri

- Differenza modello < **concettuale**
linguistico (\approx implementazione del concettuale nel LP)

`SUB(p1, p2, ..., pn)`

- Modelli concettuali < **in** : riceve il dato dal
out : trasmette il dato al
inout : riceve/trasmette il dato dal/al } corrispondente attuale



`int max(a,b)`

`void max(a,b,m)`

`void incr(&v,n)`

Modelli Linguistici di Passaggio dei Parametri

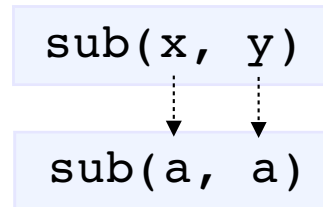
1. Passaggio per valore (**in** mode)

2. Passaggio per risultato (**out** mode)

nessun valore passato al subprog
parametro formale assegnato all'attuale al `return`
attuale: deve essere una variabile (non expr!)

Problemi:

- Collisione parametri attuali:



⇒ valore di a: quello del formale assegnato per ultimo

- Binding(*attuale*, *indirizzo*) $\left\langle \begin{array}{l} \text{call} \\ \text{return} \end{array} \right\rangle ?$

`list[ind]` ⇒ se cambia `ind` → cambia l'indirizzo!

3. Passaggio per valore-risultato (**inout** mode)

Modelli Linguistici di Passaggio dei Parametri (ii)

4. **Passaggio per reference** (**inout** mode) \Rightarrow trasmesso al subprog un path di accesso (puntatore) al parametro

Svantaggi:

- Rallentamento accesso (indiretto) al parametro formale
- Possibilità di alterazione del parametro formale anche quando non richiesto
- Creazione di alias

Possibili collisioni fra:

▪ Parametri attuali:

`void f(int x, int y)`

\Rightarrow

`f(a, a)`

▪ Elementi di array:

`f(v[i], v[j])`

\Rightarrow

se $i=j \rightarrow \text{alias}(x, y)$

▪ Formali e var non locali

```
procedure P
  var x: integer;
  procedure Q(var y: integer);
  begin
    ... x ...
  end;
begin
  ... Q(x) ...
end;
```

$\rightarrow \text{alias}(x, y)$

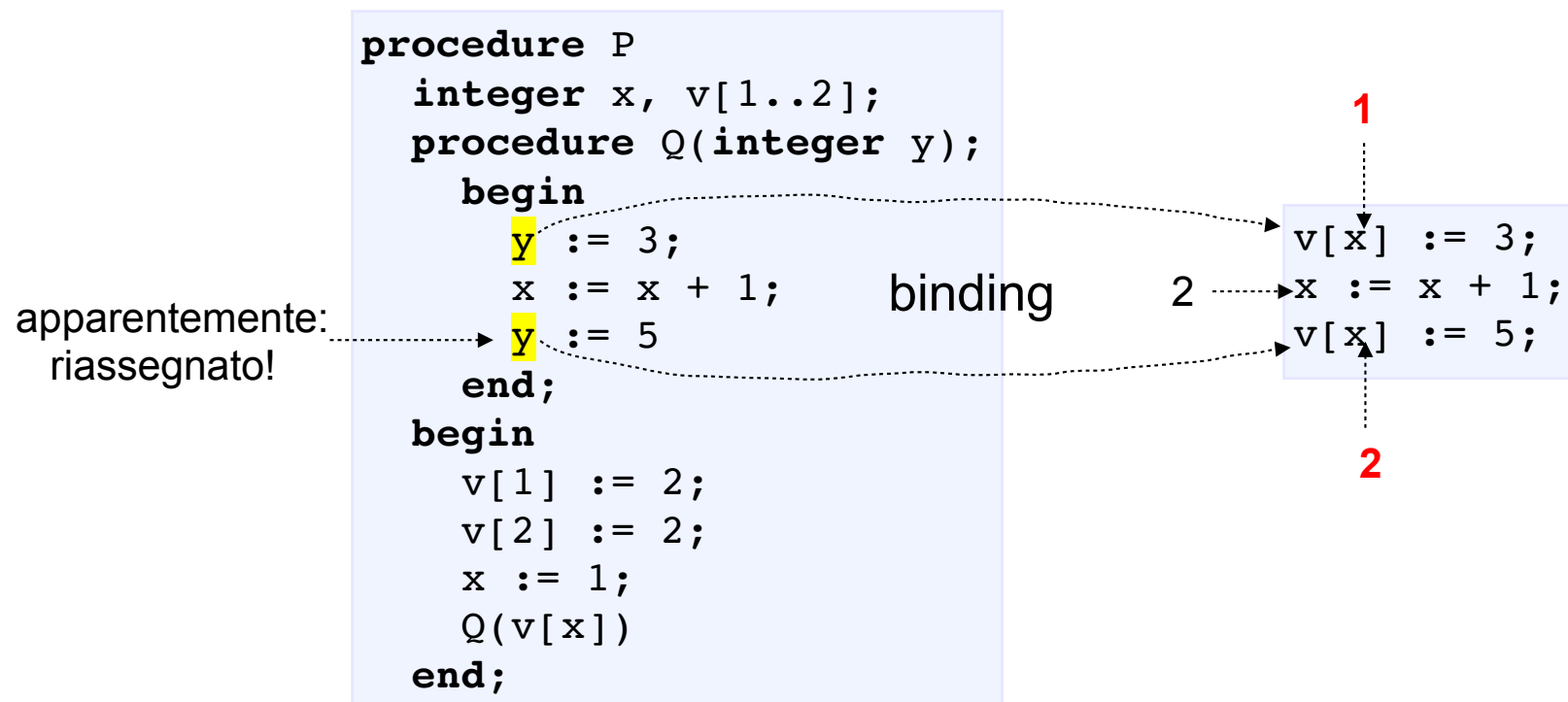
Modelli Linguistici di Passaggio dei Parametri (iii)

5. Passaggio per nome (**inout** mode che non corrisponde ad una singola implementazione: modalità linguistica *polimorfa*)

Semantica: par attuale sostituisce *testualmente* il corrispondente par formale

Binding del formale con $\left\{ \begin{array}{l} \text{valore} \\ \text{indirizzo} \end{array} \right\}$ quando il par è $\left\{ \begin{array}{l} \text{assegnato (ind)} \\ \text{referenziato (ind, val)} \end{array} \right\}$

↓
expr
var



Modelli Linguistici di Passaggio dei Parametri (iv)

Osservazioni (passaggio per nome):

- a) Ragon d'essere: flessibilità (consistente con LP con binding ritardato) {

APL: binding(*var*, *tipo*)

OOPL
- b) Forma del parametro attuale: stabilisce la modalità di passaggio

<i>Forma par attuale</i>	<i>Modalità di passaggio</i>
var scalare	reference
expr costante	valore
elem di array	?
expr con var	?

-> quando cambia l'indice
-> expr valutata \forall accesso al formale:
se var cambia \rightarrow cambia anche expr!

Modelli Linguistici di Passaggio dei Parametri (v)

c) \exists casi di semplici operazioni non implementabili con passaggio per nome:

```
void swap(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```



```
int i = 2,
    v[5] = {1, 3, 5, 7, 9};
swap(i, v[i]);
```

Diagram showing the caller code. A dashed arrow points from the value 2 to the variable i. Another dashed arrow points from the value 2 to the expression v[i]. A third dashed arrow points from the value 5 to the expression v[i].

 semantica

```
temp = i;
5 → i = v[i];
v[i] = temp;
```

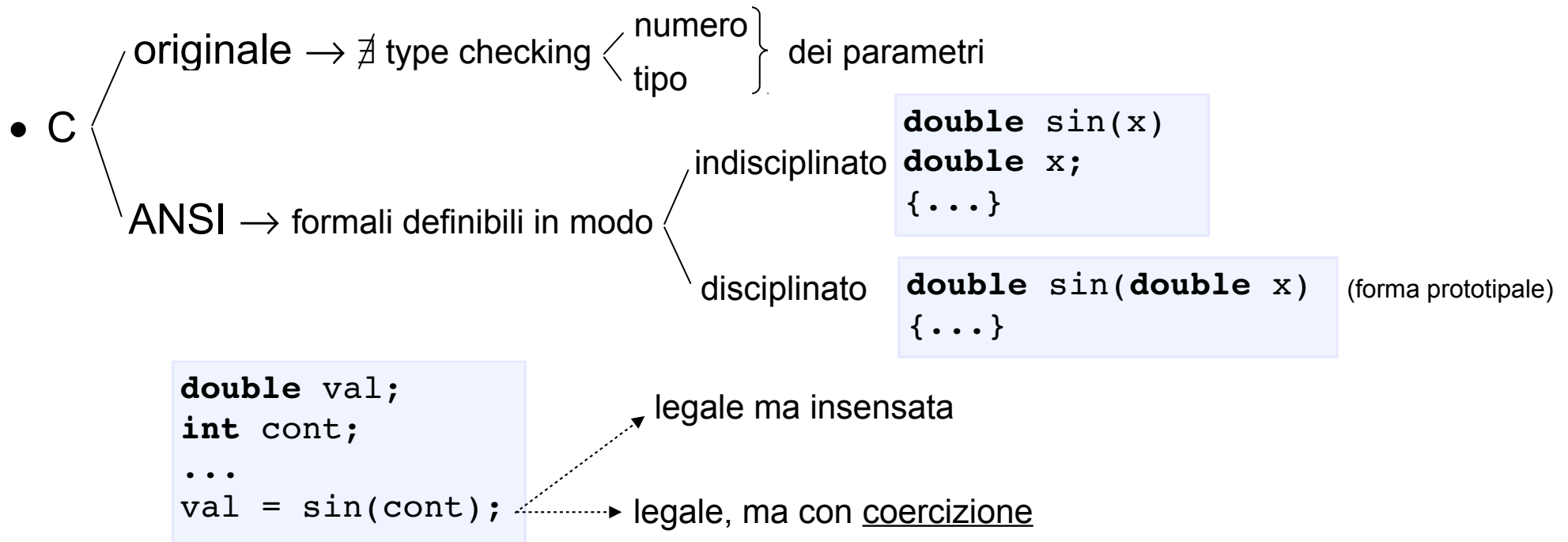
Diagram showing the execution of the swap function. A dashed arrow points from the value 2 to the variable temp. Another dashed arrow points from the value 2 to the variable i. A third dashed arrow points from the value 5 to the expression v[i].

out of range!

Type Checking dei Parametri (Formali ↔ Attuali)

- Favorisce l'affidabilità del sw, altrimenti: piccoli errori tipografici → non intercettati!

`R := SUB(1);` \Rightarrow se parametro formale di tipo reale → errore!



- C++: solo forma prototipale, però `f(int x, float y, ...)` (numero variabile di parametri)

Passaggio di Sottoprogrammi

- Esempio: subprog che computa l'integrale di una funzione → unico per ogni f !
- OSS: se necessario $\left\{ \begin{array}{l} \text{solo il corpo del subprog} \rightarrow \text{basta passare il puntatore} \\ \text{anche parametri} \rightarrow \text{pb del type checking} \end{array} \right.$

```
procedure integra(function f(x: real): real; da, a: real; var ris: real);  
...  
var val: real;      type checking statico!  
begin  
    ... val := f(da); ...  
end;
```

- C/C++: passato il puntatore alla $f \rightarrow$ tipo = protocollo della $f \rightarrow$ possibilità di type checking!

```
float media(int *v, int n){...}  
...  
float (*pf)(int *, int), ris;  
int valori[MAX];  
...  
pf = &media;  
...  
ris = (*pf)(valori, 100);
```

include tutti i tipi dei parametri

anche: pf passabile come parametro attuale

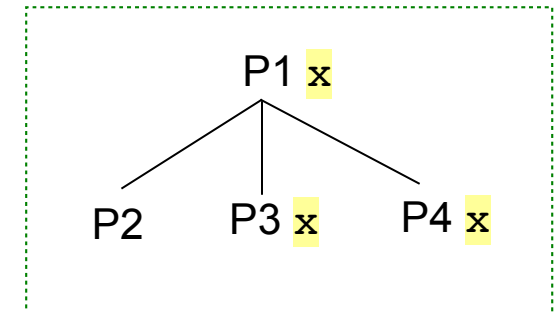
```
float integrale(float (*pf)(float),  
               float da,  
               float a);  
...  
    val = (*pf)(da);  
...
```

Passaggio di Sottoprogrammi (ii)

- Problema: quale ambiente di referenziazione per le istruzioni nel corpo del sub passato?
- Oss: sub entra in gioco in 3 contesti: definizione, passaggio, chiamata.
- Scelte progettuali:
 - ◀ **deep** binding: ambiente in cui è definito sub
 - ◀ **ad-hoc** binding: ambiente della call che passa sub come parametro attuale
 - ◀ **shallow** binding: ambiente della call a sub

```
procedure P1;  
  var x: integer;  
  procedure P2;  
    begin  
      write('x=', x)  
    end;  
  procedure P3;  
    var x: integer;  
    begin  
      x := 3;  
      P4(P2)  
    end;  
  procedure P4(P);  
    var x: integer;  
    begin  
      x := 4; P  
    end;  
  begin  
    x := 1; P3  
  end;
```

$x = \begin{cases} 1 \leftarrow \text{deep binding} & (P1) \\ 3 \leftarrow \text{ad-hoc binding} & (P3) \\ 4 \leftarrow \text{shallow binding} & (P4) \end{cases}$



Oss:

- LP a blocchi con scope statico → deep binding = soluzione naturale
- Ad-hoc binding: mai adottato (scorrelazione tra
 - ◀ sub passato
 - ◀ ambiente di passaggio)
- Shallow binding: da alcuni LP con scope dinamico (SNOBOL)

Overloading dei Sottoprogrammi

- Similitudine con overloading degli operatori
- Def: Sottoprogramma overloaded = sub omonimo di una altro nello stesso AR
- Oss:
 1. \forall versione del sub overloaded \rightarrow profilo distinto (numero/ordine/tipo dei parametri formali)
 2. $\text{Binding}(\text{nome}, \text{sub})$: determinato dai parametri attuali (chiamata)
 3. \exists LP con sub overloaded predefiniti: (Ada) `PUT` (string, int, float)
 4. Quando LP non ammette expr miste (Ada) \rightarrow tipo di ritorno della f: discriminante se stesso profilo

intero..... \rightarrow `i * f(a,b)` \Rightarrow `int f(int x, int y)`

reale..... \rightarrow `r * f(a,b)` \Rightarrow `float f(int x, int y)`

5. \exists LP (C++, Java, Ada) con possibilità di overloading da parte dell'utente (scorrelazione semantica)

`sort` array di $\begin{cases} \text{int} \\ \text{float} \end{cases}$

6. Overloading di sub con parametri di default \rightarrow possibile ambiguità

```
void f(float b = 0.0);  
void f();  
...  
f();
```

chiamata ambigua \rightarrow errore di compilazione