

Linguaggi di Programmazione

Corso di Laurea in "Informatica"

Astrarre sul controllo

Valeria Carofiglio

(Questo materiale è una rivisitazione del materiale
prodotto da Nicola Fanizzi)

Obiettivi

- astrazione funzionale
 - ☒ sottoprogrammi
 - ☒ passaggio parametri
- funzioni di ordine superiore
 - ☒ funzioni come parametro
 - ☒ funzioni come risultato
- eccezioni

Astrazione

(gestire la complessità)

Nascondere qualcosa nella descrizione di un fenomeno per poter meglio studiarlo

Nei LdP

Astrarre sul controllo

Per nascondere dettagli procedurali

Astrarre sui dati

Per definire e utilizzare tipi di dati sofisticati senza rif.

Alla loro implementazione

Sottoprogrammi

(gestire la complessità dei programmi)

- **metodologia di sviluppo**

■ Soluzione di un problema = composizione di soluzioni parziali

■ I LdP devono consentire

- Decomposizione + ricomposizione
- Sottoprogramma, procedura o funzione
- Differenze nei vari LdP

Sottoprogrammi

definizione e uso

- sezione di codice
 - ☒ identificata da un nome (*foo*)
 - ☒ dotata di un proprio ambiente locale (*a,b,tmp*)
 - ☒ capace di scambiare informazioni con l'ambiente circostante per mezzo di parametri
 - parametri espliciti (*a,b*)
 - valore di ritorno
 - ambiente non locale
- meccanismi linguistici
 - ☒ definizione
 - intestazione
 - corpo
 - ☒ uso (chiamata)

```
int foo(int a, int b) {
```

```
    int tmp=b;
```

```
    if (tmp==0) return a;
```

```
    else return a+1;
```

```
}
```

```
int x;
x = foo(3,0);
```

Sottoprogrammi

meccanismi di comunicazione: parametri

- **param.** formali nella definizione del sottoprogramma
 - ☒ Sono sempre nomi
 - ☒ Si comportano come le var. locali (ambiente)
 - ☒ come variabili legate (*bound*)
 - la ridenominazione non ha effetti sulla semantica
- **param. attuali** nella chiamata del sottoprogramma
 - ☒ Possono essere nomi o anche espressioni
 - dipende anche da come avviene l'accopp. Par.Formali \leftrightarrow Par.Attuali
- ☒ **regola generale:** numero e tipo devono coincidere
- ☒ Altre regole di compatibilità dei tipi
- ☒ Possibilità di sottoprogrammi con un numero variabile di parametri

Sottoprogrammi

meccanismi di comunicazione: valori di ritorno

- le funzioni calcolano un valore
 - ☒ scambio di informazioni non solo tramite parametri
- meccanismo linguistico per la restituzione del valore
 - ☒ return (C e derivati)
 - ☒ assegnazione al nome della funzione (Pascal)
- **procedure:** sottoprogrammi che non restituiscono un valore
 - ☒ interagiscono attraverso
 - parametri
 - ambiente non locale
 - ☒ meccanismo meno sofisticato e più pericoloso
 - ☒ tipo void (C e derivati)

astrazione funzionale

mediante l'uso dei sottoprogrammi

- Sottoprogramma come Componente SW che fornisce un servizio all'ambiente
 - ☒ non importa come si realizza il servizio
 - ☒ basta conoscere l'interfaccia per richiederli
 - una funzione per ogni servizio
- separazione
 - ☒ come richiedere il servizio (interfaccia)
 - ☒ come è implementato

astrazione funzionale mediante l'uso dei sottoprogrammi

astrazione funzionale: principio metodologico

- ☒ specifica, intestazione e corpo indipendenti (anche dal contesto)
 - il cliente non dipende dal corpo ma solo dall'intestazione
- ☒ trasparenza: sostituzione con altro corpo con la stessa semantica
- ☒ implementato attraverso i sottoprogrammi

Passaggio parametri

- modalità di legame tra parametri attuali e parametri formali (comunicazione permessa & implementazione)

☒ **fissata al momento della def. del sottoprogramma**

☒ **può essere specifica per ogni singolo parametro**

☒ **vale per ogni uso (chiamata) del sottoprogramma**

- Semantica (del tipo di comunicazione): indotta dalle tre classi di parametri definiti da un LdP

☒ **ingresso: dal chiamante al chiamato**

☒ **uscita: dal chiamato al chiamante**

☒ **ingresso/uscita: bidirezionale**

Passaggio parametri

specifiche tecniche implementative

- Che tipo di comunicazione
- Che forma di parametro
- Quale semantica della modalità
- Quale implementazione canonica
- Quale costo

Passaggio parametri

per valore

- modalità: parametro in ingresso

Comunicazione: dal chiamante al chiamato

Ambiente locale della procedura esteso con il binding tra il parametro formale e una nuova variabile

Forma del parametro attuale:

- Espressione generica

Passaggio parametri

per valore

- **Semantica**

- ☒ Espressione viene valutata prima della chiamata
- ☒ r-value viene assegnato al parametro formale

OSSERVAZIONI

- durante l'esecuzione:
 - ☒ nessun rapporto tra parametro formale e parametro effettivo
- al termine
 - ☒ il parametro viene distrutto insieme all'ambiente locale

Passaggio parametri

per valore

- **Implementazione:** allocazione a pila:

 p. formale, nel Rda della procedura

- alla chiamata si copia il valore del parametro attuale nel parametro formale (ad opera della sequenza di chiamata)

- **costo:**

 per strutture dati molto grandi, copia dispendiosa

 accesso a costo minimo (come le altre var. locali del corpo)

Passaggio parametri per valore

```
int y = 1;  
void foo(int x) {  
    x = x+1;  
}  
...  
y = 1;  
foo(y+1);  
  
// y vale 1
```

- Se non indicato esplicitamente si tratta di passaggio per valore
- La variabile y non cambia mai il suo valore
- All'uscita da foo → Rda associato distrutto (con tutto l'ambiente locale)

Passaggio parametri

per riferimento

- Meccanismo a basso livello presente in molti linguaggi

☒ Pascal (modificatore var)

☒ Altri (C e derivati): dipende dalla str. dati e presenza puntatori

Passaggio parametri

per riferimento

- modalità: parametro di ingresso/uscita
- Comunicazione: bidirezionale
- Forma parametro attuale: Espressione dotata di l-valore (anche semplice variabile)

Passaggio parametri

per riferimento

- **Semantica:**
 - Alla chiamata:
 - valutazione dell' l-valore
 - estensione dell'ambiente locale con un binding tra il parametro formale e l' l-valore del parametro attuale (alias)
 - normalmente una variabile (con due nomi diversi)
 - all'uscita
 - distruzione del legame, non della variabile

OSSERVAZIONI

- Ogni modifica al **parametro formale** è una modifica al **parametro attuale**

Passaggio parametri per riferimento

- Implementazione: allocazione a pila:
 ▸ p. formale, nel RDA della procedura
 - Alla chiamata si copia l' l-valore del param. attuale (sequenza di chiamata)
- costo: abbastanza basso
- sia per la memorizzazione dell'indirizzo
- sia per l'accesso: costo di un indirizzamento indiretto
- Poco usato dai linguaggi moderni

Passaggio parametri per riferimento

```
int y = 0;  
void foo(reference int x) {  
    x = x+1;  
}  
...  
foo(y);  
// y vale 1
```

- Per riferimento: indicato esplicitamente
- X è un nome per y
- La variabile y cambia il suo valore
- All'uscita da $\text{foo} \rightarrow \text{RdA}$ associato distrutto (con tutto l'ambiente locale)

```
int[] V = new V[10];  
int i = 0;  
void foo(reference int x) {  
    x = x+1;  
}  
...  
V[1] = 1;  
foo(V[i+1]);  
// V[1] vale 2
```

Passaggio parametri

per costante

- **Modalità:** Parametro di ingresso (altro nome *read-only*)

☒ Comunicazione dal chiamante al chiamato

- **Forma del parametro Attuale:** espressioni generiche

☒ vincolo statico: nessuna modifica del parametro formale permessa nel corpo

- né diretta: assegnazione
- né indiretta: passaggio a sottoprogrammi che li modifichino

Passaggio parametri

per costante

- **semantica:** come quella del passaggio per valore
 - ☒ Espressione viene valutata prima della chiamata
 - ☒ r-value viene assegnato al parametro formale
- **implementazione:** dipende dalla macchina astratta
 - piccole dimensioni: come nel passaggio per valore (copia)
 - grandi dimensioni: come nel passaggio per riferimento
- **costo:** dipende dall'implementazione

Passaggio parametri

per risultato

- Modalità: Parametro di uscita (duale del passaggio per costante)

☒ Comunicazione dal chiamato al chiamante

☒ ambiente locale esteso: aggiungendo un binding tra il param. formale ed una nuova variabile

- Forma del parametro attuale: espressione dotata di l-value

Passaggio parametri

- Semantic:
per risultato

■ Valuta lo l-valore del parametro attuale

- Se esiste piu' di un parametro, in quale ordine si valutano? (ordine di assegnazione all'indietro)
- Quando viene determinato l'l-valore

■ al termine del sottoprogramma

- prima della distruzione dell'ambiente locale copia del valore del param. formale nello l-valore del param. Attuale

OSSERVAZIONI

■ durante l'esecuzione nessun legame tra i due parametri

Passaggio parametri

per risultato

- **implementazione:** analoga a passaggio per valore: allocazione a pila:
 - ☒ p. formale, nel RDA della procedura
 - ☒ alla chiamata si copia il valore del parametro attuale nel parametro formale (ad opera della sequenza di chiamata)
- **costo:** idem passaggio per valore

Passaggio parametri per risultato

```
void foo(result int x) {  
    x = 2;  
}  
  
int y = 1;  
foo(y);  
// y vale 2
```

- Per risultato: indicato esplicitamente
- X è un nome per y
- La variabile y cambia il suo valore (diff. Con passaggio per valore)

prima della distruzione
dell'ambiente locale copia del
valore del param. formale nello
l-valore del param. Attuale

- All'uscita da foo → Rda associato distrutto (con tutto l'ambiente locale)

Passaggio parametri

per valore-risultato

- Modalità: parametro di ingresso/uscita

■ Comunicazione: bidirezionale

■ Combina passaggio per valore (di entrata) a passaggio per risultato (di sola uscita)

- Forma del parametro attuale: espressione dotata di l-value

Passaggio parametri

per valore-risultato

- Semantică:

 alla chiamata:

- Valuta lo l-valore del parametro attuale
- Assegna lo l-valore del parametro attuale al parametro formale

 al termine del sottoprogramma

- prima della distruzione dell'ambiente locale copia del valore del param. formale nella locazione indicata dallo l-valore del parametro attuale.

Passaggio parametri per valore-risultato

- **Implementazione:**

■ come per il passaggio per valore (allocazione a pila):

■ p. formale, nel RdA della procedura

■ alla chiamata si copia il valore del parametro attuale nel parametro formale (ad opera della sequenza di chiamata)

- ma in Ada è come per riferimento per strutture grandi (allocazione a pila):

■ p. formale, nel RdA della procedura

■ Alla chiamata si copia l' l-valore del param. attuale (sequenza di chiamata)

- anche se non semanticamente corretta

Passaggio parametri

per valore-risultato

```
void foo(valueresult int x) {  
    x = x+1;  
}  
...  
int y = 5;  
foo(y);  
// y vale 6
```

Passaggio parametri

per valore-risultato

```
void foo(reference/valueresult int x  
reference/valueresult int y  
reference int z) {  
    y = 2;  
    x = 4;  
    if (x==y) z=1;  
}  
.  
int a = 3;  
int b = 0;  
foo(a,a,b);
```

x e y hanno valori diversi solo in assenza di aliasing (il passaggio per rif. Ha aliasing → I parametri formali e parametri attuali sono due rif allo stesso oggetto. → la condizione x==y è sempre vera)

- nel passaggio per valore-risultato non c'è aliasing
- ☒ La chiamata `foo(a,a,b)` termina senza la modifica di b
- altrimenti (passaggio di x e y per riferimento)

☒ La chiamata a `foo(a,a,b)` **terminerebbe con b che vale 1**

Passaggio per nome

definizione

- modalità simile a quella per riferimento ma semanticamente più precisa (Algol)
- **regola di copia (semantica ridotta a sintassi)**
 - ☒ sia x parametro formale di una funzione f ed a un'espressione compatibile con x
 - ☒ una chiamata a f con parametro A attuale a è semanticamente equivalente all'esecuzione del corpo di f nel quale tutte le occorrenze di x sono sostituite con a

Passaggio per nome

osservazione 1

- Caso: presenza di più variabili con lo stesso nome

```
int x = 0;  
int foo(name int y) {  
    int x = 2;          X (parametro attuale ) è  
    return x+y;        stata catturata dalla  
}                      dichiarazione locale  
...  
int a = foo(x+1);
```

1) valutando il *return* nell'ambiente locale di *foo*,
si avrebbe *return x+x+1*; ossia si restituisce 5

2) se il corpo fosse {*int z = 2; return z+y;*} si avrebbe

N. Fanizzi - Linguaggi di Programmazione (c) 2006

Passaggio per nome

Osservazione 1 (cont.)

- Caso: presenza di più variabili con lo stesso nome
 - 1. variabile catturata dalla dichiarazione locale
 - 2. sostituzione senza cattura di variabile
 - valutazione nell'ambiente del chiamante anziché del chiamato
- quindi nel passaggio per nome
 - ☒ semantica: regola di copia
 - ☒ sostituzione: senza cattura
 - viene sostituito sempre il parametro attuale + il suo ambiente
- ☒ fissato a momento della chiamata

Passaggio per nome

Osservazione 2

- Caso: valutazione del parametro attuale ad ogni sua occorrenza del parametro formale

- ☒ secondo la regola di copia

- ☒ conseguenze per gli effetti collaterali

```
int i = 2;  
int fie(name int y) {  
    return y+y;  
}  
.  
int a = fie(i++);  
// i vale 4 ed a vale 5
```

- ☒ La regola di copia impone la valutazione di `i++` per due volte

- la prima restituisce 2 e porta il valore di `i` a 3
- la seconda restituisce 3 e porta il valore di `i` a 4

Passaggio per nome

confronto con altre modalità

- Differenza con la modalità valore-risultato

```
void fiefoo( valueresult/name int x  
            valueresult/name int y ) {  
    x = x+1  
    y = 1;  
}  
  
...  
int i = 1;  
int[] A = new int[5];  
A[1] = 4;  
fiefoo(i,A[1]);
```

☒ per valore risultato (ma anche per riferimento):

A[1] vale 1 ed i vale 2 e il resto dell'array A intatto

☒ per nome: A[1] vale 4 ed i vale 2 e il resto dell'array A Programmazione (c) 2006

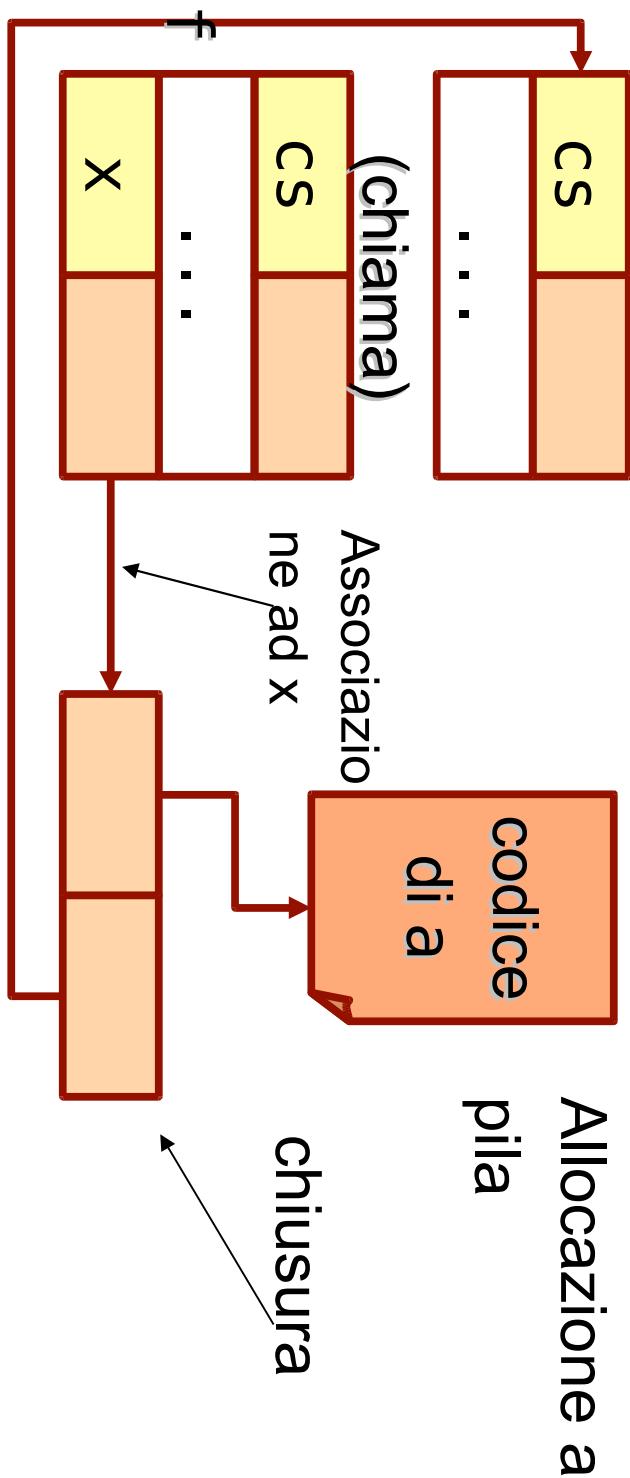
ma modifica anche A[2] che vale 1

Passaggio per nome implementazione

- Necessita per il chiamante di passare
 - ☒ espressione testuale (*nome*):
 - ☒ ambiente
 - almeno tutte le var. libere, ossia non associate, nell'espressione
- **chiusura** = (*espressione, ambiente*) → associazioni per le variabili non locali
- alla chiamata: passaggio della chiusura
- ☒ quando il chiamato incontra un rif. al parametro formale,
risolve il riferimento valutando la prima componente secondo la seconda componente della chiusura

Passaggio per nome

implementazione



- chiusura: coppia di puntatori
 - ☒ al codice di valutazione dell'espressione
 - ☒ di catena statica al blocco dell'ambiente locale nel quale valutare l'espressione
- alla chiamata

- ☒ Il chiamante costruisce una chiusura (**codice di a**, **puntatore al proprio Rda**)
- ☒ Associa la chiusura creata al parametro formale **x** nel Rda del chiamato

Passaggio per nome

sommario

- modalità di ingresso e uscita
- parametro formale non corrisponde ad una variabile locale al sottoprogramma
- parametro Attuale: espressione generica
- possibile aliasing (tra p formale e p. attuale)
 - ☒ p. attuale valutato come l-valore se p. formale compare a sinistra di un'assegnazione
- semantica: regola di copia
- implementazione:
 - ☒ estensione dell'ambiente locale con binding tra p. formale e chiusura
 - ☒ valutazione ad ogni riferimento al p. formale
- modalita costosa e complessa

Funzioni di ordine superiore

- Funzioni che hanno come parametro o risultato altre funzioni
 - ☒ Parametri: molti linguaggi
 - ☒ Risultato: pochi linguaggi
 - Paradigma funzionale
- Considereremo solo funzioni definibili nell'ambiente globale
 - ☒ Come in C

Funzioni come parametri

esempio

- f passata come parametro attuale a g
- f chiamata tramite parametro formale h

```
int g(int h(int n))  
{  
    int x = 2;  
    return h(3) + x;  
}
```

☒ In quale ambiente non locale viene valutata f?

```
{  
    int x = 4;  
    int z = g(f);  
}
```

Funzioni come parametri

esempio

- f passata come parametro attuale a g
- f chiamata tramite parametro formale h
- x è definita 2 volte

☒ In quale ambiente non locale viene valutata f?

...
...
...
...
...

```
{ int x = 4;  
int z = g(f);  
}
```

Funzioni come parametri

Grado di variabilità ulteriore

W indipendente dalla visibilità (scope)

- linguaggi con visibilità statica
 - W deep binding
- linguaggi con visibilità dinamica
 - W Sia deep sia shallow binding

Funzioni come parametri

Visibilità statica e deep binding

- Visibilità statica:

L'ambiente non locale è nel blocco che contiene la definizione di *f*

```
int g(int h(int n))
```

```
{
```

```
    int x = 2;
    return h(3) + x;
```

```
}
```

- La *x* in *f* alla chiamata è quella del blocco esterno

```
...
```

- *h(3)* restituisce 4

- *g* restituisce 6



Momento di creazione
del legame tra *h* e *f*
(deep binding)

```
{ int x = 1;
    int f(int y)
    {
        return x + y;
    }
```

```
int g(int h(int n))
{
    int x = 2;
    return h(3) + x;
}
```

Funzioni come parametri

Visibilità dinamica e deep binding

```
{ int x = 1;
```

```
int f(int y)
{ return x + y; }
```

```
int g(int h(int n))
```

```
{ int x = 2;
return h(3) + x;
}
```

```
...
```

```
{ int x = 4;
int z = g(f);
}
```

Momento di creazione
del legame tra **h** e **f**
(deep binding)

- **Visibilità dinamica:**
L'ambiente non locale è nel blocco che chiama **f**

- La **x** in **f** alla chiamata è quella locale al blocco della chiamata a **g(f)**



- **h(3)** restituisce 7
- **g** restituisce 9



Funzioni come parametri

Visibilità dinamica e shallow binding

```
{ int x = 1;
```

```
int f(int y)
{ return x + y; }
```

```
int g(int h(int n))
```

```
{ int x = 2;
return h(3) + x;
}
```

- La `x` in `f` alla chiamata è quella locale al blocco della chiamata a `g`



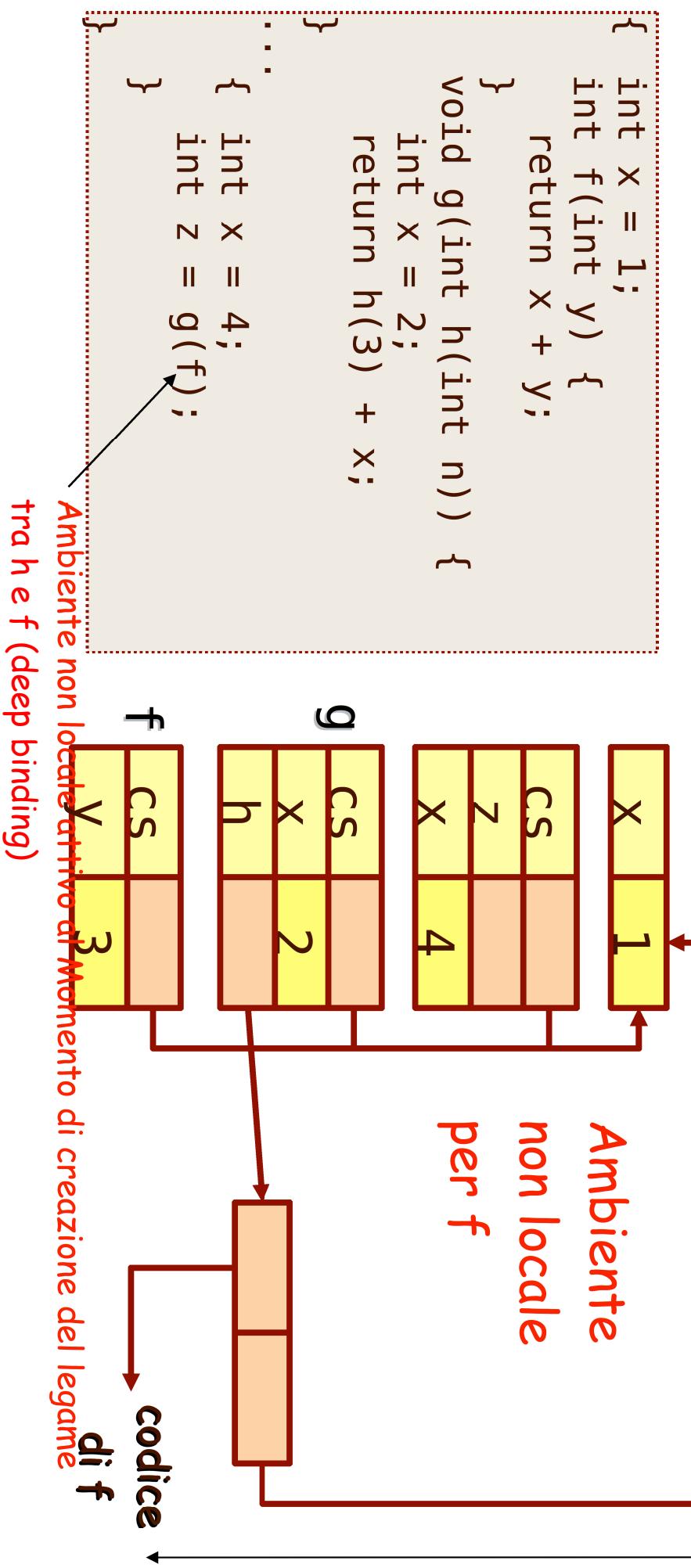
- `h(3)` restituisce 5
- `g` restituisce 7

Momento di chiamata
di `f` tramite `h`
(shallow binding)

```
{ int x = 4;
int z = g(f);
}
```

Funzioni come parametri

implementazione: SCOPE STATICO E DEEP BINDING

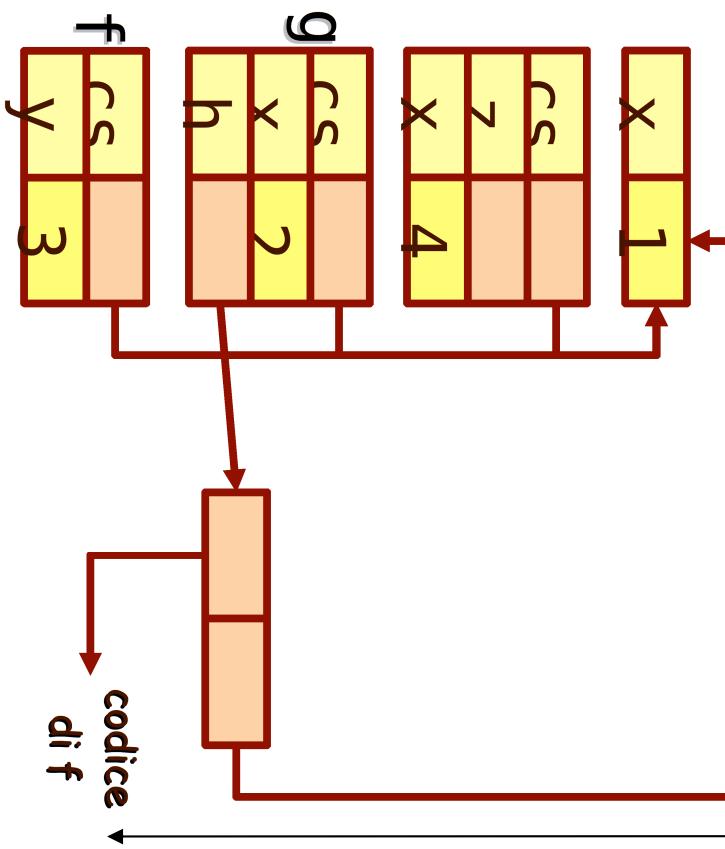


- All'invocazione diretta di `f`: info statiche sul livello di annidamento della def. Di `f` rispetto al blocco nel quale `f` è chiamata. → inizializzazione del punt. Di `cs` del Rda di `f`.

Funzioni come parametri

implementazione: SCOPE STATICO E DEEP BINDING

```
{  
    int x = 1;  
    int f(int y) {  
        return x + y;  
    }  
    void g(int h(int n)) {  
        int x = 2;  
        return h(3) + x;  
    }  
    ...  
}
```



- ☒ Alla chiamata di *g*, *f* si lega ad *h* (deep binding)

- Il chiamante (*g*) costruisce una chiusura

- (codice di *f*, ambiente non locale per valutare *f*)

- Macc.Astra: (1) Trova il codice al quale trasferire il controllo (prima parte della chiusura); (2) Associa la seconda componente della chiusura al puntatore di CS del chiamato (*ff*)

Funzioni come parametri

implementazione: SCOPE STATICO E DEEP BINDING

OSSERVAZIONI

Il linguaggio deve permettere la definizione di funzioni con ambiente non locale (def di funzioni all'interno di blocchi annidati)

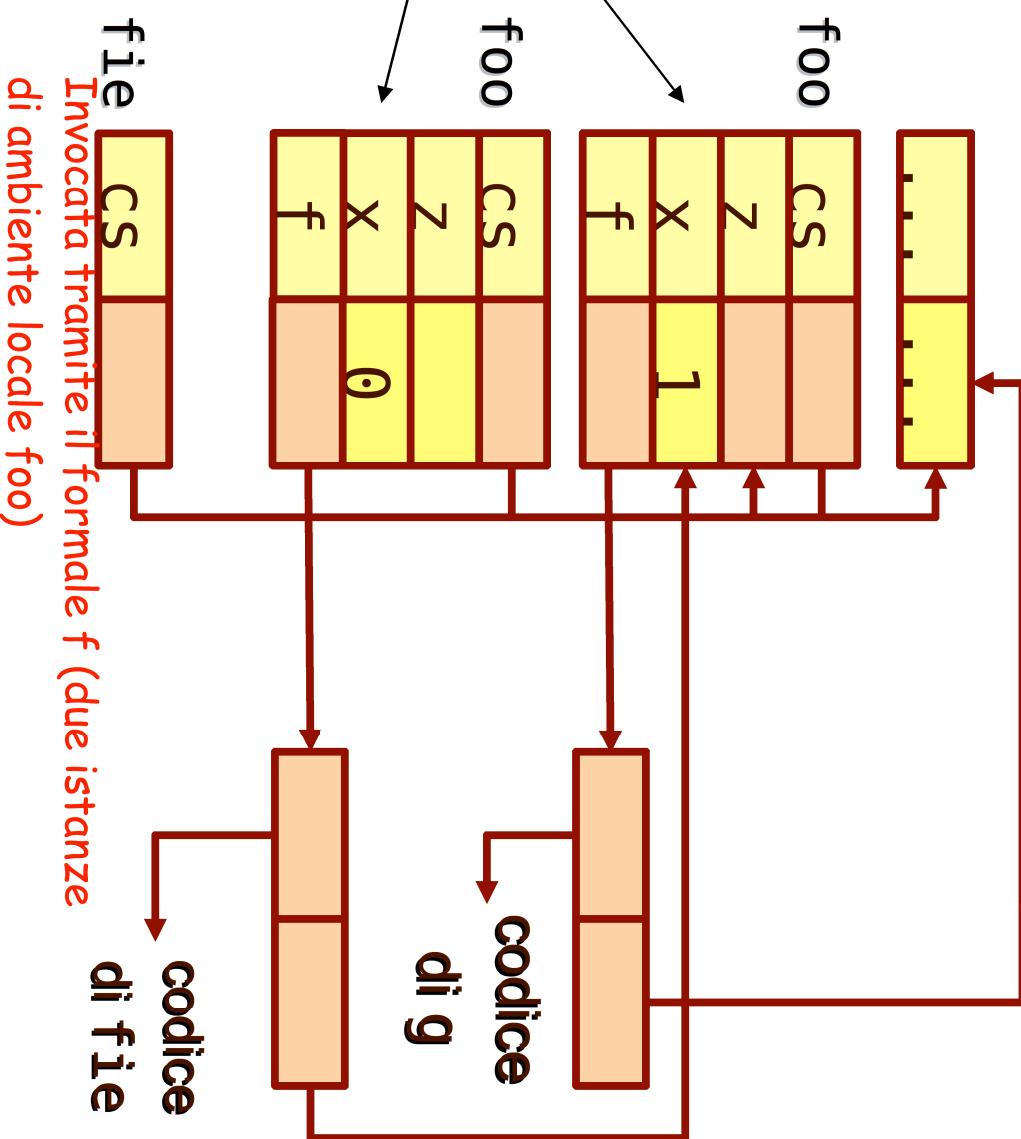
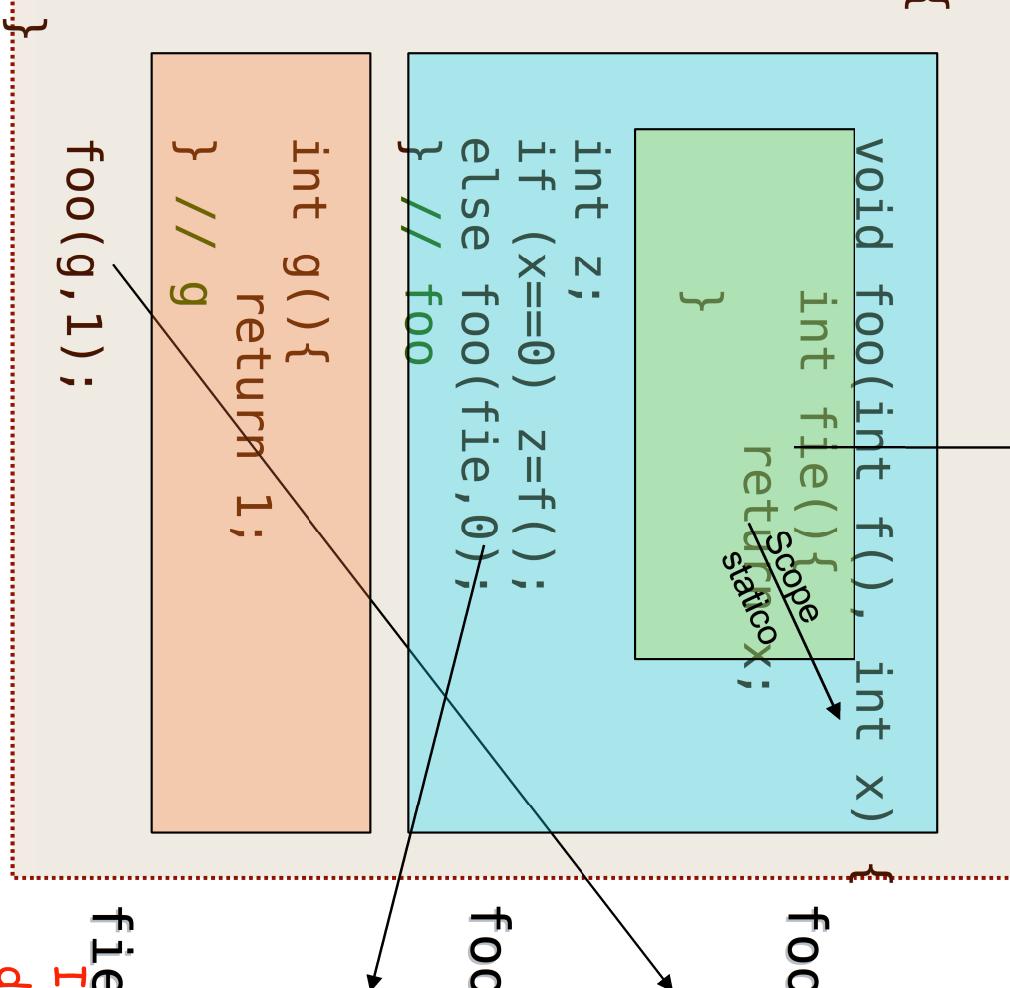
In C: non c'è ambiente non locale → non c'è bisogno di chiusure
(è sufficiente il puntatore al codice della funzione passata)

Funzioni come parametri

C'è differenza tra deep e shallow binding, in scope statico?

Quale ambiente non locale per x
con mutua ricorsione?

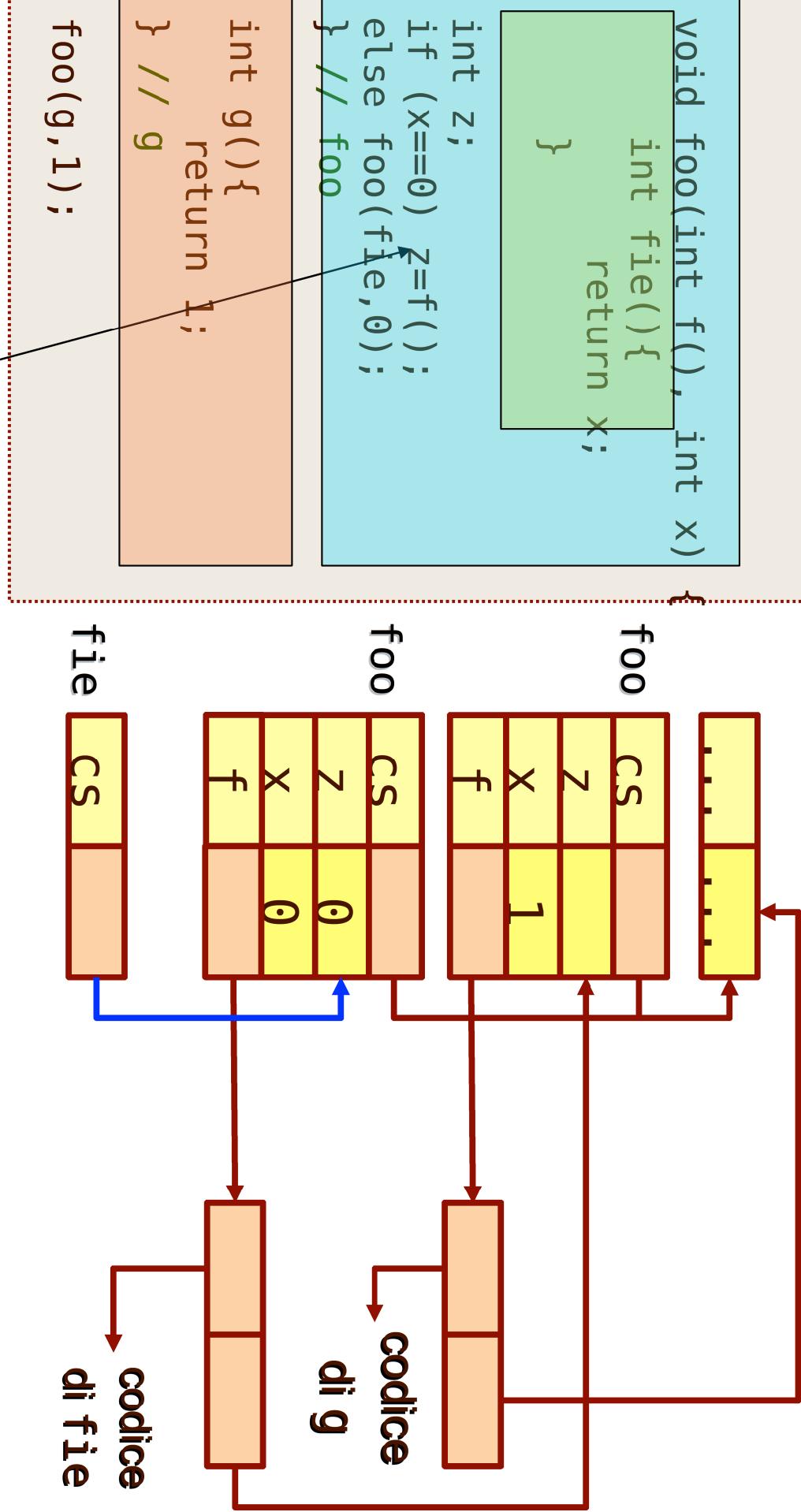
shallow binding



Le regole di scope non dicono nulla su quale istanza di x si debba usare nel corpo di f

Funzioni come parametri

SOLUZIONE: POLITICHE DI BINDING



Funzioni come parametri **definire l'ambiente**

- Regole di visibilità
 - ☒ Garantite dalla struttura a blocchi
- Eccezioni alle regole di visibilità
 - ☒ Ridefinizione nomi
 - ☒ Possibilità di uso dopo la dichiarazione
- Regole di scope
- Regole del passaggio parametri
- Politica di binding

Funzioni di ordine superiore come risultato

- Permette la *creazione dinamica di funzioni* (a run-time)

☒ Servono per la valutazione della funzione:

- codice della funzione + ambiente

☒ Restituisce una chiusura

- Macchina astratta: chiamata a chiusura
- Puntatore di catena statica determinato tramite la chiusura

Funzioni di ordine superiore

come risultato

Tipo delle funzioni che non prendono alcun argomento e restituiscono un intero

```
{ int x = 1;  
void->int F() {  
    int g() {  
        return x+1;  
    }  
    return g;  
}
```

RISULTATO di una funzione:
che restituisce una funzione:
chiusura
(codice della funzione, ambiente)

- Es. gg() e' la funzione successore
 - Il suo valore è una chiusura (valore ottenuto dinamicamente)
 - Il suo puntatore di cs è determinato dall'ambiente della chiusura

Funzioni di ordine superiore

come risultato

- Possibilità di restituire funzioni dall'interno di un blocco annidato

☞ Possibilità che il suo ambiente faccia riferimento ad un nome sulla pila

```
void->int F() {  
    x=1;  
    int g(){  
        return x+1;  
    }  
    return g;  
}  
  
void->int gg = F();  
int z = gg();
```

Blocco
annidato

Funzioni di ordine superiore

come risultato

```
void->int F() {
```

```
    x=1;
```

```
    int g(){
```

```
        return x+1
```

```
}
```

```
return g;
```

```
}
```

```
}
```

- Possibilità di restituire funzioni da un blocco annidato
 - ☒ L'ambiente potrebbe far riferimento ad un nome sulla pila

- Da distruggere all'uscita! (→ dangling reference)

Serve un'altra struttura

dinamica (heap+garbage

collector)

- Ambienti di durata illimitata



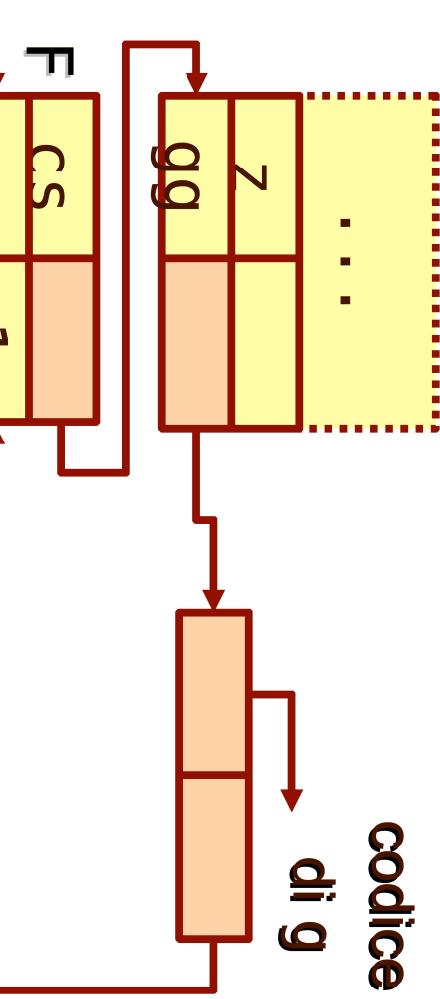
Ling. funzionali

Abbandonare
disciplina a PILA

N. Fanizzi ° Linguaggi di I

CS

gg



codice

di g

gg

X è locale ad

F, all'uscita da

F x distrutta

disciplina a PILA

Funzioni di ordine superiore

come risultato

Possibilità di restituire funzioni

da un blocco annidato nei vari LdP

☒ Restrizioni per garantire che non si verifichi un riferimento ad ambienti disattivati

- No funzioni annidate (C, C++)
- Restituzione di funzioni solo non annidate (Modula-2, Modula-3)
- Vincolo sullo scope delle funzioni annidate che sono restituite (ADA)

eccezioni

- Evento che non deve/può essere gestito nel flusso di controllo normale

☒ Errore di semantica dinamica

- Div. per zero, overflow, ...
- Terminazione esplicita del flusso corrente (computazione parallela)

• Grande variabilità tra i linguaggi di programmazione

☒ Specifica

- definizione
- lancio
- gestione

(costrutti di astrazione)

Interruzione della esecuzione

Risalita della pila di sistema

Ricerca di un gestore di eccezione

Eccezioni:

Un LdP deve specificare

- quali eccezioni sono gestibili e come definire?

☒ Eccezioni della macchina astratta

- Alla violazione semantica dinamica

☒ Eccezioni definite dall'utente

- ML, Ada: Valori di un tipo speciale
- C++: valori qualsiasi (contiene un valore generato dinamicamente, passato al gestore)
- Java: oggetto di una qualsiasi classe
☒ (sottoclassa di Throwable)

• come sollevare un'eccezione (una volta definita)?

☒ Implicitamente (macchina astratta)

☒ Esplicitamente: istruzioni appropriate (utente)

Eccezioni: Specifica

grande variabilità nei LdP

- **come va gestita un'eccezione?**

azioni da compiere

- Definizione di una capsula attorno ad una sezione di codice (*blocco protetto*) in cui possono verificarsi eccezioni da gestire
- Definizione di un gestore dell'eccezione al quale trasferire il controllo in caso di eccezione
 - ☒ Staticamente legata al blocco protetto
- ☒ come riprendere l'esecuzione

Eccezioni: esempio

```
class EccezioneVettoreVuoto extends Throwable {x=0;};  
int media(int[] v) throws EccezioneVettoreVuoto() {  
    int s=0;  
    if (length(v)==0) throw new EccezioneVettoreVuoto();  
    else  
        for (int i=0; i<length(v); i++) s=s+v[i];  
    return s/length(v);  
};  
.  
.  
try {  
    m = media(w);  
    .  
}.  
catch (EccezioneVettoreVuoto e)  
{fwrite("Array Vuoto"); }
```

Definizione

Possono
essere eccezioni
tutte le istanze
della classe
EccezioneVettoreVuoto

Eccezioni: esempio

```
class EccezioneVettoreVuoto extends Throwable {x=0;};
int media(int[] v) throws EccezioneVettoreVuoto() { ← Definizione
    int s=0;
    if (length(v)==0) throw new EccezioneVettoreVuoto(); ← di media
    else
        for (int i=0; i<length(v); i++) s=s+v[i];
    return s/length(v);
}
try {
    m = media(w);
    ...
} catch (EccezioneVettoreVuoto e)
    {fwrite("Array Vuoto"); }
```

Parola chiave:
Introduce
Lista delle eccezioni
sollevabili nel corpo
della funzione
EccezioneVettoreVuoto

Eccezioni: esempio

```
class EccezioneVettoreVuoto extends Throwable {x=0;};  
int media(int[] v) throws EccezioneVettoreVuoto() { ←  
    int s=0;  
    if (length(v)==0) throw new EccezioneVettoreVuoto();  
    else  
        for (int i=0; i<length(v); i++) s=s+v[i];  
    return s/length(v);  
}  
.  
.  
.  
try {  
    m = media(w);  
    ..  
}  
catch (EccezioneVettoreVuoto e)  
{write("Array Vuoto"); }
```

Definizione

di media

Parola chiave:
Solleva una eccezione

Eccezioni: esempio

```
class EccezioneVettoreVuoto extends Throwable {x=0;};
int media(int[] v) throws EccezioneVettoreVuoto() { ←  
    int s=0;  
    if (length(v)==0) throw new EccezioneVettoreVuoto();  
    else  
        for (int i=0; i<length(v); i++) s=s+v[i];  
    return s/length(v);  
}  
.  
.  
.  
try {  
    m = media(w);  
    ...  
}  
catch (EccezioneVettoreVuoto e)  
{fwrite("Array Vuoto"); }
```

Definizione

di media

Parola chiave:
Blocco protetto

Eccezioni: esempio

```
class EccezioneVettoreVuoto extends Throwable {x=0;};
int media(int[] v) throws EccezioneVettoreVuoto() {
    int s=0;
    if (length(v)==0) throw new EccezioneVettoreVuoto();
    else
        for (int i=0; i<length(v); i++) s=s+v[i];
    return s/length(v);
}
...
try {
    m = media(w);
    ...
}
catch (EccezioneVettoreVuoto e)
    {fwrite("Array Vuoto"); }
```

Definizione
di media

Parola chiave:
Gestore dell'eccezione

Eccezioni: esempio

Se il vettore v è vuoto →

media solleva una

eccezione (istanza di
classe EccezioneVettoreVuoto)



```
class EccezioneVettoreVuoto extends Throwable {x=0;};  
  
int media(int[] v) throws EccezioneVettoreVuoto() {  
    int s=0;  
    if (length(v)==0) throw new EccezioneVettoreVuoto();  
    else  
        for (int i=0; i<length(v); i++) s=s+v[i];  
    return s/length(v);  
}  
  
try {  
    m = media(w);  
    ...  
} catch (EccezioneVettoreVuoto e)  
{write("Array Vuoto");}
```

MacC.Astr. :

-Interrompe esecuzione

-Propaga Eccezione

-chiusura di tutti i blocchi

in esecuzione (fino a

capsula try che intrappa

(catch) l'eccezione.

-controllo da try a catch

(in assenza di try →

gestore di default)

Eccezioni: gestione **(dipendenza dai LdP. Tratti comuni)**

- *Ogni eccezione ha un nome*
(ossia non è un evento anonimo)
- Menzionato nei costrutti throw e catch
- *Si possono anche inglobare dei valori nell'evento (no nell'esempio)*
- Interpretabili per decidere come reagire all'evento
 - Campo x nell'esempio

Eccezioni: gestione

(dipendenza dai LdP. Tratti comuni)

- In caso di mancata gestione

☒ La propagazione avviene attraverso

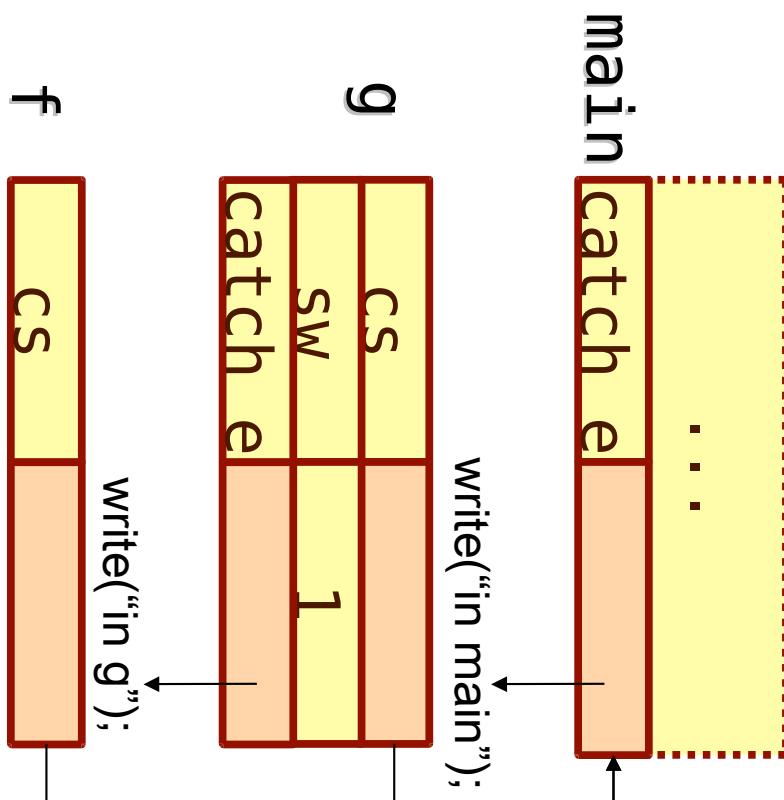
- la terminazione e
 - il salto al chiamante lungo la catena dinamica
(anche se il gestore è legato staticamente al suo blocco protetto)
 - ... fino al raggiungimento di un gestore o alla gestione di default del top level
- ☒ regola: 'eccezione gestita dall'ultimo gestore posto sulla pila'

Classe di eccezioni

supposta già dichiarata

(**throwable**) gestione a carico dell'ultimo gestore sulla pila

```
void f() throws E {  
    throw new E();  
}  
  
void g(int sw) throws E {  
    if (sw==0) f();  
    try {f();} catch (E e) { write("in g"); }  
}  
...  
try { g(1); } catch (E e) { write("in main"); }
```



Eccezioni:

Blocchi protetti (in cui puo' essere sollevata l'eccezione)

Eccezioni: **implementazione**

- Pila di esecuzione

- ☒ All'entrata nel blocco protetto

- Puntatore al gestore corrispondente nel RdA del blocco + tipo di eccezione cui si riferisce

- ☒ Uscita "normale"

- Eliminazione dal RdA del puntatore al gestore

- ☒ In caso di eccezione

- Macch.Astr. cerca un gestore nel RdA corrente
- Se non lo si trova

- ☒ Ripristina lo stato della macchina
- ☒ Elimina il record di attivazione corrente
- ☒ Solleva l'eccezione nuovamente

DIFETTO

La Pila viene sempre manipolata, anche nell'uscita normale

eccezioni implementazione

- Implementazione costosa
 - ☒ Ogni volta che si entra in un blocco protetto la macch.Astratta manipola la pila
 - computazione anche quando l'eccezione non si verifica
- ☒ ottimizzazioni in fase di compilazione
- Particolari sul testo: (lettura)