

Capitolo 9

Astrazione sul controllo:
sottoprogrammi ed eccezioni

Argomenti

- Astrazione sul controllo
- Modalità di passaggio dei parametri
- Funzioni di ordine superiore
 - funzioni come parametro
 - funzioni come risultato
- Gestori delle eccezioni

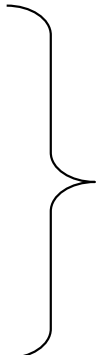
Astrazione

- identificare proprietà importanti di cosa si vuole descrivere
- concentrarsi sulle questioni rilevanti e ignorare le altre
- cosa è rilevante dipende dallo scopo del progetto

Astrazione sul controllo

- Sottoprogrammi, blocchi, parametri

```
double P (int x) {  
    double z;  
    /* CORPO DELLA FUNZIONE  
    return expr;  
}
```

- Specifica P
 - Scrivi P
 - Usa P
- 
- senza conoscere
il contesto

Parametri

- Terminologia:

- dichiarazione/definizione

```
int f (int n) {return n+1;}
```



Parametro formale

- uso/chiamata

```
x = f(y+3);
```



Parametro attuale

Una funzione comunica col chiamante

- Valore restituito

```
int f() {return 1;}
```

- Parametri

- **main** → **proc**
- **main** ← **proc**
- **main** ↔ **proc**

- Ambiente non locale

Modalità di passaggio dei parametri

- Due modi principali:
 - per valore:
 - il valore dell'attuale è assegnato al formale, che si comporta come una variabile locale
 - pragmatica: $\text{main} \rightarrow \text{proc}$
 - attuale qualsiasi; modifiche al formale non passano all'attuale
 - per riferimento (per variabile)
 - è passato un riferimento (indirizzo) all'attuale; i riferimenti al formale sono riferimenti all'attuale (*aliasing*)
 - pragmatica: $\text{main} \leftrightarrow \text{proc}$
 - attuale: variabile; modifiche al formale passano all'attuale

Passaggio per valore

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



qui y vale 1

- Il formale `x` è una var locale (sulla pila)
- Alla chiamata, l'attuale `y+1` è valutato ed il valore è assegnato al formale `x`
- Nessun legame tra `x` nel corpo di `foo` e `y` nel chiamante
- Al ritorno da `foo`, `x` viene distrutto (tolto dalla pila)
- Non è possibile trasmettere info da `foo` al chiamante mediante il parametro
- Costoso per dati grandi: copia
- Java, Scheme, Pascal (default), C e Java (unico modo);

Passaggio per riferimento (per variabile)

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



qui y vale 2

- Viene passato un riferimento (indirizzo; puntatore)
- Il formale `x` è un alias di `y`
- L'attuale deve essere un L-valore ("una variabile")
- Al ritorno da `foo`, viene distrutto il (solo) legame tra `x` e l'indirizzo di `y`
- Trasmissione bidirezionale tra chiamante e chiamato
- Efficiente nel passaggio, ma indirezionale nel corpo
- Pascal (var); in C simulato passando un puntatore...

Passaggio per costante (o read-only)

- Il passaggio per valore garantisce la pragmatica main → proc a spese dell'efficienza
 - dati grandi sono copiati anche quando non sono modificati
- Passaggio read-only (Modula-3; ANSI C: `const`)
 - nella procedura non è permessa la modifica del formale (controllo statico del compilatore: no alla sin di assegnamento; non passato per riferimento ad altre proc)
 - implementazione a discrezione del compilatore:
 - parametri “piccoli” passati per valore
 - parametri “grandi” passati per riferimento
- In Java: `final`

```
void foo (final int x){ //qui x non può essere  
    modificato  
}
```

Passaggio per risultato

- Duale del passaggio per valore. Pragmatica: `main` \leftarrow `proc`

```
void foo (result int x) {x = 8;}
```

```
...
```

```
y = 1;  
foo(y);
```



qui y vale 8

- Il formale `x` è una var locale (sulla pila)
- Al ritorno da `foo`, il valore di `x` è assegnato all'attuale `y`
- Nessun legame tra `x` nel corpo di `foo` e `y` nel chiamante
- Al ritorno da `foo`, `x` viene distrutto (tolto dalla pila)
- Non è possibile trasmettere info dal chiamante a `foo` mediante il parametro
- Costoso per dati grandi: copia
- Ada: `out`

Passaggio per valore/risultato

- Insieme valore+risultato. Pragmatica: main \leftrightarrow proc

```
void foo (value-result int x)
    { x = x+1; }
```

...

```
y = 8;
foo(y);
```



qui y vale 9

- Il formale `x` è a tutti gli effetti una var locale (sulla pila)
- Alla chiamata, il valore dell'attuale è assegnato al formale
- Al ritorno, il valore del formale è assegnato all'attuale
- Nessun legame tra `x` nel corpo di `foo` e `y` nel chiamante
- Al ritorno da `foo`, `x` viene distrutto (tolto dalla pila)
- Costoso per dati grandi: copia
- Ada: `in out` (ma solo per dati piccoli; per dati grandi passa riferimento)

Valore e riferimento: morale

- Passaggio per valore:
 - semantica semplice: il corpo non ha necessità di conoscere come la procedura verrà chiamata (*trasparenza referenziale*)
 - implementazione abbastanza semplice
 - potenzialmente costoso il passaggio; efficiente il riferimento al parametro formale
 - necessità di altri meccanismi per comunicare main ← proc
- Passaggio per riferimento:
 - semantica complessa; *aliasing*
 - implementazione semplice
 - efficiente il passaggio; un po' più costoso il riferimento al parametro formale (un indiretto)

Passaggio per nome

- Regola di copia:
una **chiamata** alla procedura **P** è la stessa cosa che eseguire il **corpo** di **P** dopo aver **sostituito** i parametri attuali al posto dei parametri formali
- “Macro espansione”, realizzata in modo semanticamente corretto
- Apparentemente semplice...

Passaggio per nome

- In Algol-W era il default

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



y = y+1;

in questo caso l'effetto è quello del passaggio per riferimento (che non esiste in Algol W)

vediamo un caso più delicato...

Passaggio per nome

```
int y;  
void fie (name int x){  
    int y;  
    x = x + 1; y = 0;  
}
```

```
...  
y = 1;  
fie(y);
```

```
{ int y;  
y = y+1;  
y = 0; }
```

conflitto (e “*cattura*”) di variabili !

una domanda sensata: in quale ambiente avviene la valutazione dell'attuale dopo la sostituzione?

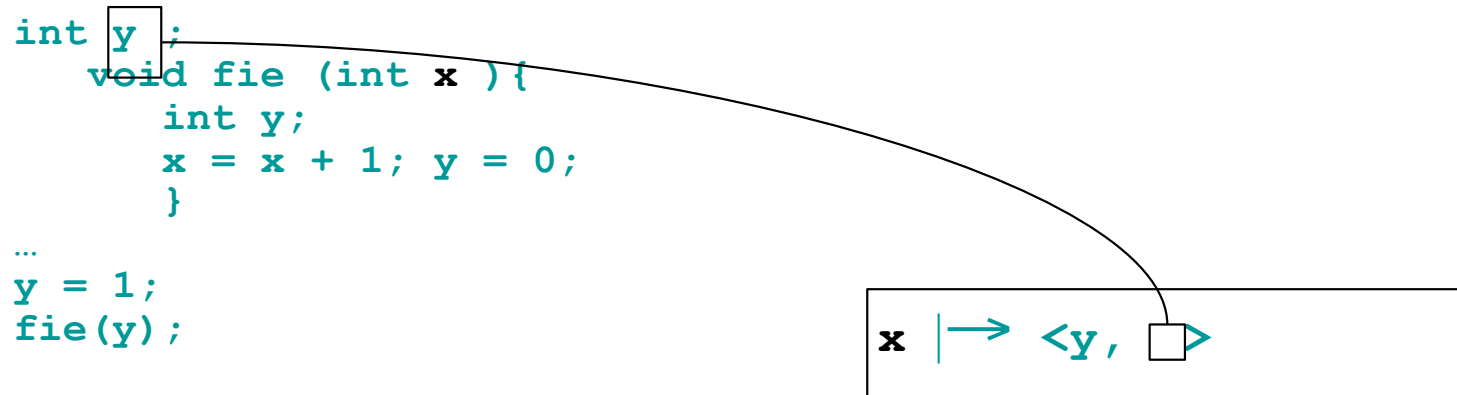
scoping statico: in quello del chiamante !

Le due variabili *y* e *y* sono diverse !

fie incrementa l'attuale (*y*) attraverso il formale *x* e crea e distrugge una nuova *y*

Passaggio per nome

- Viene passata una coppia: $\langle \text{exp}, \text{amb} \rangle$
 - exp è il parametro attuale (testo, non valutato)
 - amb è l'ambiente di valutazione (in scoping statico)
- *Ogni volta* che il formale è usato, exp è valutata in amb .



Aliasing possibile tra formale e attuale; attuale deve valutare ad un L-val se il formale compare a sin di assegnamento

Pragmatica: $\text{main} \leftrightarrow \text{proc}$

Costoso: passaggio di ambiente + valutazione ogni volta

Solo Algol 60 e W. Ma implementazione fondamentale...

Passaggio per nome: implementazione

- Come passare la coppia $\langle \text{exp}, \text{env} \rangle$?
 - un puntatore al testo di *exp*
 - un puntatore di catena statica (sullo stack) al record di attivazione del blocco di chiamata
 - cioè una chiusura

Le chiusure servono a passare funzioni come argomenti ad altre procedure

Parametro per nome = funzione nascosta senza argomenti che valuta il parametro nell'ambiente del chiamante (un *thunk*, nel gergo Algol)

Funzioni di ordine superiore

- Alcuni linguaggi permettono di:
 - passare funzioni come argomenti di procedure
 - restituire funzioni come risultato di procedure
- Entrambi i casi: come gestire l'ambiente della funzione ?
- Caso più semplice
 - Funzioni come argomento
 - Occorre un puntatore al record di attivazione all'interno della pila: passa una chiusura !
- Caso più complicato
 - Funzione ritornata da una chiamata di procedura
 - Occorre mantenere il record di attivazione della funzione restituita: disciplina a pila non funziona!!

Funzioni come parametro di procedure

- Il passaggio per nome è un caso particolare:
si passa una fne senza argomenti; corpo=exp.
- Esempio: ispirato a Pascal
ricorda: in ANSI C puoi passare fni, ma **non** ci sono funzioni annidate (e dunque non servono chiusure, basta un puntatore)

```
int x=4; int z=0;
int f (int y){
    return x*y;}
void g ( int h(int n) ) {
    int x;
    x = 7;
    z = h(3) + x ;
end;

...
{int x = 5;
  g(f);
}
```

- tre dichiarazioni di x
- quando f sarà chiamata (tramite h)
quale x (non locale) sarà usata?
- in scope *statico*, certo la x esterna
- in scope *dinamico*, ha senso sia la x del blocco di chiamata che la **x** interna

Politiche di binding

- Quando una procedura viene passata come parametro, si crea un riferimento tra un nome (par formale: h) e una procedura (par attuale: f).

Pb: quale ambiente non locale si applica al momento dell'esecuzione di f (in quanto chiamata via h)?

- ambiente al momento della creazione del legame $h \rightarrow f$?
 - ~~deep binding~~
- ambiente al momento della chiamata di f via h ?
 - *shallow binding*

sempre
usato con
scope
statico

può aver
senso con
scope
dinamico

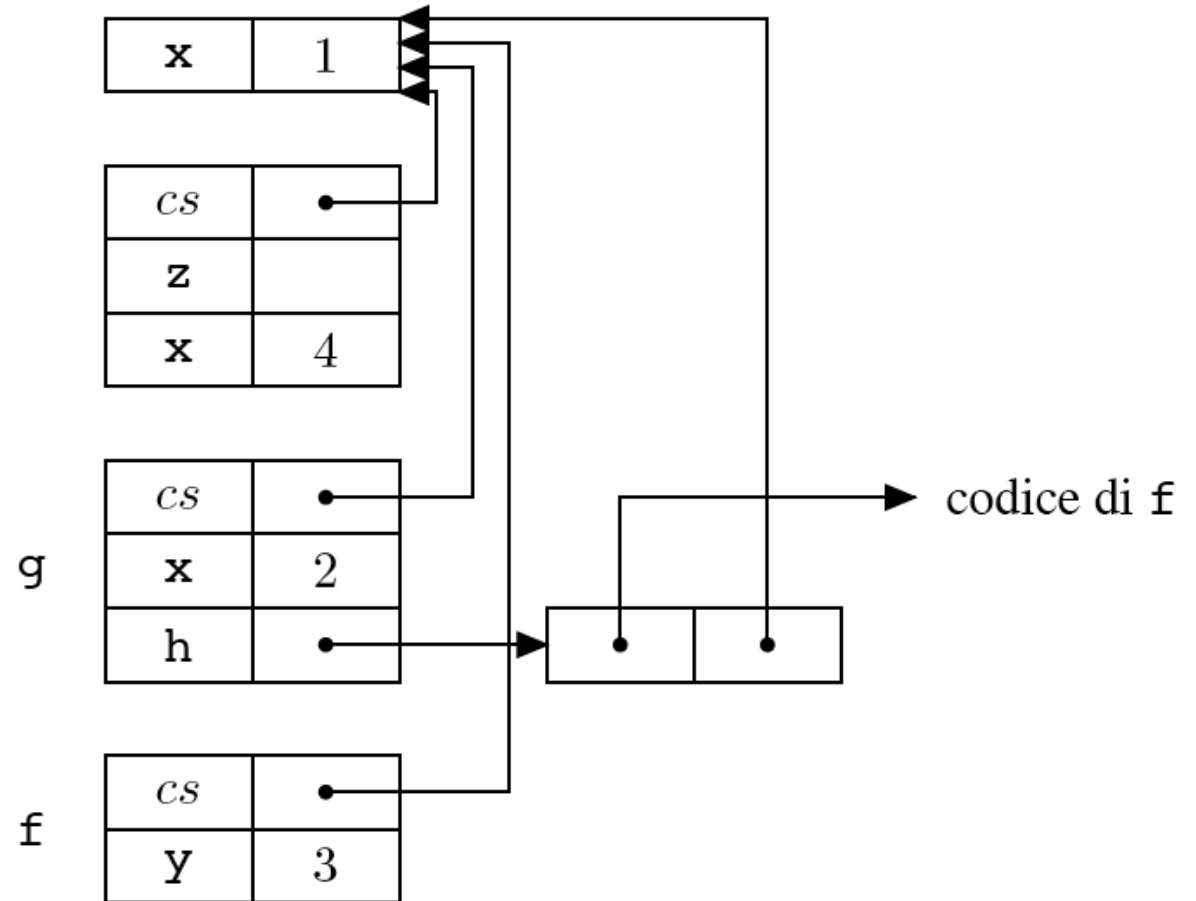
Deep e shallow binding: esempio

```
{int x = 1;
int f(int y) {
    return x+y;
}
void g (int h(bool b)) {
    int x = 2;
    return h(3) + x;
}
...
{int x = 4;
  int z = g(f);
}
```

- $h(3)$ invoca f , che accede a x quale x ?
- Scope statico
 - deep: $x=1$ (esterna)
 - shallow: $x=1$ rossa (esterna)
 - *la regola di scope è sufficiente!*
- Scope dinamico
 - deep: $x=4$ (blocco di chiamata)
 - shallow: $x=2$ (blocco interno)

Deep binding e chiusure

```
{int x = 1;  
int f(int y) {  
    return x+y;  
}  
void g (int h(bool b)) {  
    int x = 2;  
    return h(3) + x;  
}  
...  
{int x = 4;  
  int z = g(f);  
}
```



Chiusure

- Passare dinamicamente sia il legame col codice della funzione, che il suo ambiente non locale, cioè una chiusura `<code, env>`
- Alla chiamata di una procedura passata per parametro
 - alloca (come sempre) il record di attivazione
 - prendi il puntatore di catena statica dalla chiusura

Riassumendo: parametri funzioni (e per nome)

- Chiusure per mantenere puntatore all'ambiente statico del corpo di una funzione
- Alla chiamata, il puntatore di catena statica determinato mediante la chiusura
- Tutti i puntatori di catena statica puntano sempre indietro nella pila
 - record di attivazione possono essere “saltati” per accedere a var non locali
 - de-allocazione dei record di attivazione secondo stretta politica a pila (lifo: last-in-first-out)

Scope dinamico: implementazione

- Shallow binding
 - non necessita di alcuna attenzione
 - per accedere a x, risali la pila
 - uso delle strutture dati solite (A-list, CRT)
- Deep binding
 - usa necessariamente qualche forma di chiusura per “congelare” uno scope da riattivare più tardi

Deep vs shallow binding con scope statico

- Riassumendo:
 - scope dinamico
 - è possibile sia deep binding
 - implementato con chiusure
 - che shallow binding
 - non necessita di implementazione ulteriore
 - scope statico
 - sempre usato deep binding
 - implementato con chiusure
 - a prima vista deep o shallow non fa differenza
 - è la regola di scope statico a stabilire quale non locale usare
- *Non è così:* vi possono essere dinamicamente
 - più istanze del blocco che dichiara il nome non-locale
 - accade in presenza di ricorsione

Restituire una funzione, caso semplice

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

La proc F ritorna una chiusura.

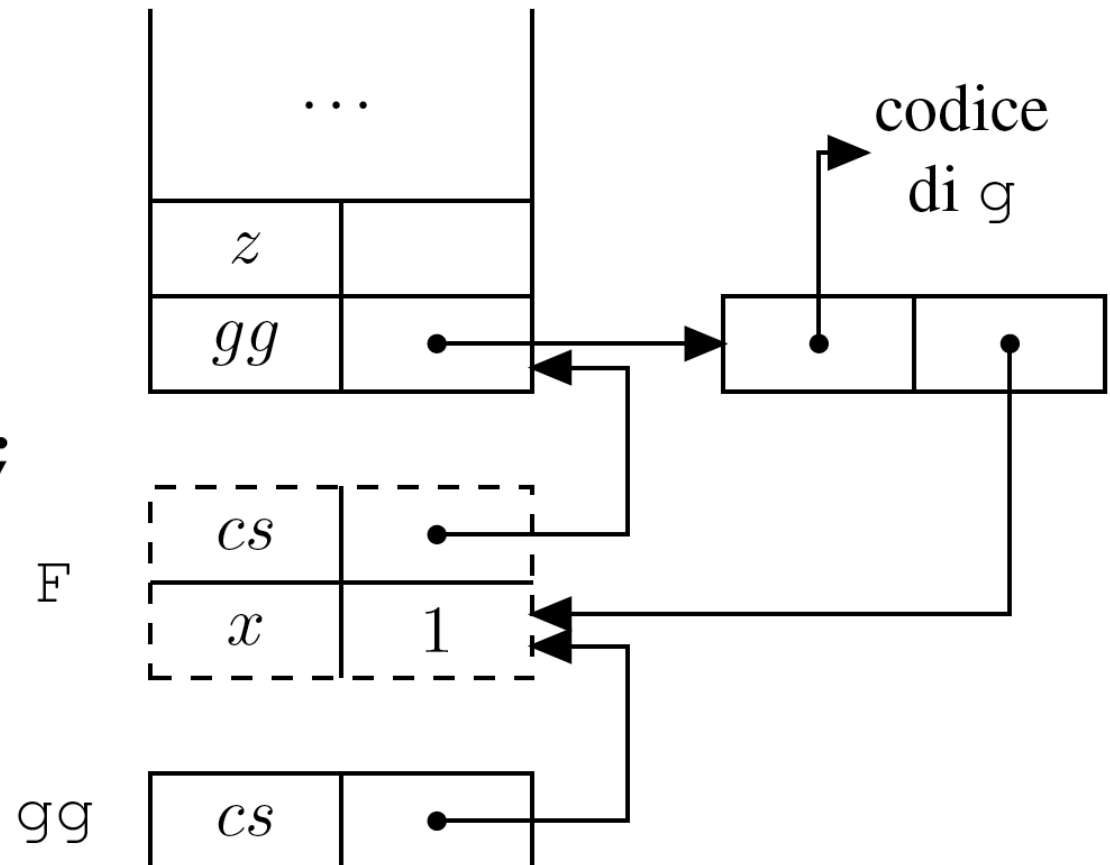
Modifica la macchina astratta per gestire una “chiamata a chiusura” come in `gg()`;

Restituire una funzione, caso complesso

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

La proc F ritorna una chiusura.

Ma dove sta l'associazione per x dopo che F termina?



Morale: funzioni come risultato

- Uso delle chiusure, ma...
- I record di attivazione persistono indefinitamente
 - perdita proprietà della pila (lifo)
- Come implementare la “pila” in questo caso:
 - non deallocare esplicitamente
 - record di attivazione sullo heap
 - le catene statica e dinamica collegano i record
 - invoca il garbage collector quando necessario

Eccezioni: “uscita strutturata”

- Terminare parte di una computazione
 - Saltar fuori di un costrutto
 - Passando dati attraverso il salto
 - Ritornando il controllo al più recente punto di gestione
 - Record di attivazione non più necessari sono deallocati
 - altre risorse, incluso spazio sullo heap, possono essere liberate
- Due costrutti
 - Dichiarazione del gestore dell'eccezione
 - posizione e codice di trattamento
 - Comando/espressione per sollevare eccezione

Spesso usate per condizioni inusuali o eccezionali, ma non necessario.

Esempio

- La funzione **average** calcola la media di

un vettore; se il vettore è vuoto, solleva eccezione

```
class EmptyExcp extends Throwable {int x=0;};

int average(int[] V) throws EmptyExcp() {
    if (length(V)==0) throw new EmptyExcp();
    else {int s=0; for (int i=0, i<length(V), i++) s=s+V[i];}
    return s/length(V);
};
```

solleva l'eccezione

eccezione

```
...
try {
    ...
    average(W);
    ...
}
```

intrappola e gestisce
l'eccezione

```
catch (EmptyExcp e) {write('Array_vuoto'); }
```

gestore (handler)
dell'eccezione

Sollevare un'eccezione

- Il **gestore** è legato in modo **statico** al blocco di codice protetto
- L'esecuzione del gestore rimpiazza la parte di blocco che doveva essere ancora eseguita

```
class EmptyExcp extends Throwable {int x=0;};

int average(int[] V) throws EmptyExcp() {
    if (length(V)==0) throw new EmptyExcp();
    else {int s=0; for (int i=0, i<length(V), i++) s=s+V[i];}
    return s/length(V);
};

...
try{...
    average(W);
    ...
}
catch (EmptyExcp e) {write('Array_vuoto'); }
```

Propagare un'eccezione

- Se l'eccezione non è gestita nella routine corrente:
 - la routine termina, l'eccezione è ri-sollevata al punto di chiamata
 - se l'eccezione non è gestita dal chiamante, l'eccezione è propagata lungo la catena
 - fino a quando si incontra un gestore o si raggiunge il top-level, che fornisce un gestore default
- Vengono tolti i rispettivi frame dallo stack:
 - per ogni frame che viene tolto, ripristina lo stato dei registri
- Ogni routine ha un gestore “nascosto” che ripristina lo stato e propaga le eccezioni lungo la pila

Eccezione si propaga lungo la catena *dinamica*

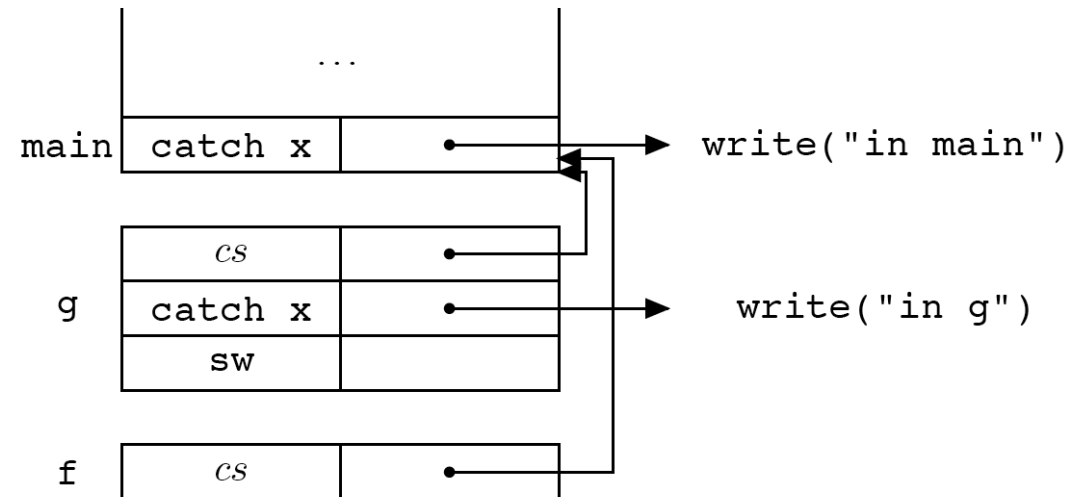
```
{  
void f() throws X {  
    throw new X();  
}  
  
void g (int sw) throws X {  
    if (sw == 0) {f();}  
    try {f();} catch (X e) {write("in_g");}  
}  
  
...  
try {g(1);}  
catch (X e) {write("in_main");}  
}
```

Quale dei due gestori viene eseguito?

Osserva: quando l'argomento di `g` (qui `g(1)`) è frutto dell'esecuzione non si conosce staticamente il gestore ...

Eccezione si propaga lungo la catena *dinamica*

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



Implementare le eccezioni

- Semplice:
 - all'inizio di un blocco protetto ():
 - metti su una pila (può essere quella di sistema) il gestore
 - quando un'eccezione è sollevata:
 - toglì il primo gestore sulla pila e guarda se è quello giusto
 - in caso contrario, solleva di nuovo l'eccezione e ripeti
- Inefficiente nel caso – il più frequente – che non si verifichi eccezione:
 - ogni **try** e **routine** devono mettere e togliere roba dalla pila
- Una soluzione migliore è quella di una *tabella di indirizzi*