

## Astrarre sul controllo

Procedure, passaggio dei parametri, eccezioni

- Procedure e funzioni: astrazione sul controllo
- Modalità di passaggio dei parametri
- Funzioni di ordine superiore
  - funzioni come parametro
  - funzioni come risultato
- Gestori delle eccezioni

## Astrazione

Meccanismo per gestire la complessità:

- identifica proprietà importanti di ciò che si vuole descrivere
- nasconde, ignora gli aspetti secondari
- permette di descrivere e concentrarsi solo sulle questioni rilevanti

Permette di:

- evidenziare tratti comuni in strutture diverse (matematica)
- gestire la complessità:
  - spezzare un sistema complesso in sottosistemi
  - descritti astrattamente, senza entrare nel dettaglio

In informatica:

- astrazione sul controllo (istruzioni)
- astrazione sui dati

## Astrazione sul controllo

Definizione di procedura-funzione principale meccanismo di astrazione

- assegno un nome ad una parte di codice
- descrivo la parte rilevante del comportamento
  - come interagisce con la restante parte del codice
- ignoro i dettagli implementativi

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

Possiamo:

- usare `log`
  - conoscendo le sue specifiche
  - senza conoscerne nei dettagli l'implementazione
- specificare `log`
  - senza conoscere l'implementazione
- implementare `log` (scrivere il codice)
  - rispettando le specifiche
  - senza conoscere il contesto in cui verrà usato

## Procedure funzioni

Distinguiamo tra:

- **funzioni**: restituiscono un valore
- **procedure**: non restituiscono nulla,
- in C e derivati: procedure caso particolare di funzioni che restituiscono un valore di tipo `void` (avente un unico elemento) solo funzioni

Terminologia:

- dichiarazione/definizione

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

`n` **parametro formale**

- uso/chiamata

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

`y+3` **parametro attuale**

## Meccanismi di comunicazione tra (funzione) chiamante - chiamata

- **Valore restituito**

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

- **Passaggio di parametri**  
varie modalità di passaggio

- **Modifiche ambiente non locale**

- in questo caso il meccanismo di astrazione meno efficace
- chi usa la funzione deve conoscere anche quali modifiche sono svolte
- nei linguaggi funzionali questa possibilità è assente più astratti

## Passaggio dei parametri

Come avviene l'interazione **chiamante** – **chiamato** attraverso i parametri

Distinguo tra:

- parametri d'ingresso: main -> proc
- parametri d'uscita: main <- proc
- parametri d'ingresso – uscita : main <-> proc

Varie modalità di passaggio con diversa:

- semantica
- implementazione
- costo

## Passaggio dei parametri

Due modi principali:

- per **valore** (call by value):
  - il parametro **formale** è una **variabile locale**
  - il parametro **attuale** viene valutato il suo **valore** assegnato al formale
  - attuale: **r-value**, una qualsiasi espressione
  - si comporta come l'**assegnamento**
  - modifiche al formale non si ripercuotono sull'attuale
  - parametro in ingresso: main -> proc
- per **riferimento** (call by reference)
  - il parametro **formale** contiene un riferimento (indirizzo) all'attuale;
  - attuale: **l-value**, deve rappresentare una locazione
  - ogni accesso al formale comporta un accesso all'attuale (aliasing)
  - parametro di ingresso-uscita: main <-> proc modifiche al formale passano all'attuale

## Passaggio per valore

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

- Il formale **x** è una var locale (nel RdA)
- alla chiamata, l'attuale **y+1** è valutato e assegnato a **x**
  - viene fatta un'operazione di assegnamento **x = y+1**
- nessun legame tra **x** nel corpo di **foo** e **y** nel chiamante
  - se **x** ha tipo semplice (modello a valore)

## Passaggio per valore

- la sua semantica definita in termini della semantica dell'assegnamento
- può essere costoso per dati di grande dimensione, se l'operazione di assegnazione lo è
- Pascal (default), Scheme, C (unico modo);
- Implementazione:
  - nel RdA alloco una variabile associata al parametro formale
  - al momento della chiamata, il parametro attuale viene valutato e nel RdA inserito il suo valore

## Passaggio per riferimento (per variabile)

```
void foo (ref int x){ x = x+1;}  
...  
y = 1;  
foo(y);  
foo(V[y+1]);
```

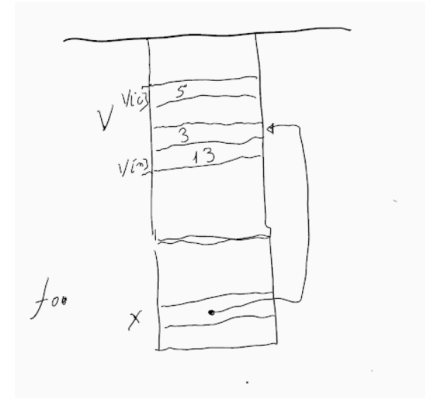
- viene passato un riferimento (indirizzo, puntatore)
- l'attuale deve essere un l-value (un riferimento)
- il formale, x, diventa un alias dell'attuale ,y
  - considerato a basso livello, non presente nei linguaggi più recenti
- trasmissione bidirezionale tra chiamante e chiamato

## Nei linguaggi di programmazione

- raramente meccanismo di default: Fortran
- in alcuni linguaggi offerto come opzione:  
C++, PHP, Visual Basic .NET, C#, REALbasic, Pascal  
`foo (ref int x)`  
`foo (var x : int)`
- in altri simulabile:
  - C, ML, Rust simulato passando un puntatore, riferimento, indirizzo di memoria ([call by address](#))
  - Java simulato usando i tipi riferimento, o il [call by sharing](#)

## Implementazione passaggio per riferimento

- nello stack di attivazione inserisco un puntatore
  - efficiente nel passaggio
- un diverso accesso ai dati nel chiamato:
  - il parametro formale è implementato come [puntatore](#),
  - trattato nel codice come una [variabile locale](#),



## Simulazione in C

Tramite puntatori:

```
void foo (int *x){ *x += 1;}  
...  
y = 1;  
foo(&y);  
printf("y = %i \n", y); // stampa "y = 2"
```

## Differenze tra implementazione e simulazione

In C++

```
void foo(int &value) {value++;}
```

```
foo(x)
```

In C

```
void foo(int *value) { (*value)++; }
```

```
foo(&x)
```

## Call-by-sharing

Nei linguaggi con modello a [riferimento](#)

- Python, Java, Ruby, JavaScript  
dove l'assegnazione crea la condivisione di una stessa copia

anche il passaggio dell'argomento, nominalmente per valore, crea una condivisione (sharing) tra chiamato e chiamante.

- argomento  $x$  tipo riferimento (assegnazione usa il modello a riferimento):  
strutturato, array, class,
- le modifiche nella funzione del parametro formale si ripercuotono sul parametro attuale
- attuale e formale riferimenti a una singola copia dello stesso oggetto

## Call-by-sharing: esempi in Python

```
def f(a_list):  
    a_list += [1]
```

```
m = []  
f(m)  
print(m)
```

stampa [1], **ma**

```
def f(a_list):  
    a_list = [1]
```

```
m = []  
f(m)  
print(m)
```

stampa [],

nella procedura un nuovo vettore viene creato e assegnato a a\_list

## Call by sharing: esempi in Java:

Le classi sono un tipo riferimento, l'assegnamento avviene copiando il riferimento (puntatore)

```
class A {  
    int v;  
}  
A y = new A(3)  
...  
void foo (A x){ x.v += 1;}  
...
```

```
foo (y);
```

Lo stesso esempio con la classe Integer avrebbe un comportamento diverso

$x = x + 1$ ; crea un nuovo intero e lo assegna a  $x$

Il nome “call-by-sharing” non standard in ambito Java, si parla di call-by-value

Il nome call-by-sharing mette in evidenza che

- il passaggio del parametro crea una copia condivisa
- eventuali modifiche nel parametro formale si possono ripercuotere sul parametro attuale
- se al parametro formale si assegna un nuovo elemento, nessuna modifica sull'attuale

importante sapere:

- se il passaggio del parametro crea una copia condivisa o una nuova copia
- se l'assegnamento modifica l'oggetto puntato o crea un nuovo oggetto

infatti:

- copia condivisa e oggetti mutabili comportamento call-by-reference
- copia condivisa e oggetti immutabili (l'assegnazione crea una nuova istanza), comportamento call-by-value

Passaggio per valore:

- semantica semplice: si crea una copia locale
- implementazione abbastanza semplice,
- costoso il passaggio per dati di grande dimensione
- efficiente il riferimento al parametro formale
- necessità di altri meccanismi per comunicare main <- proc

Passaggio per riferimento:

- semantica complessa: si crea aliasing
- implementazione semplice
- efficiente il passaggio
- un po' più costoso il riferimento al parametro formale (indiretto)

Call-by-sharing

- a seconda dei casi, simile a call-by-value o a call-by-reference

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; C, C++: const)

- il formale trattato come una costante
- nella procedura non è permessa la sua modifica:
  - no all'assegnamento;
  - no al passaggio ad altre procedure che possono modificarlo (per riferimento)
- controllo statico del compilatore:
  - vincolo blando in C
  - forte in Java

## Passaggio per costante

Implementazione a discrezione del compilatore:

- parametri “piccoli” passati per valore
- parametri “grandi” passati per riferimento

In C:

```
int foo(const char *a1, const char *a2){  
    /* le stringhe a1 a2 non possono essere modificate */  
}
```

In Java: final

```
void foo (final A a){  
    //qui a non può essere modificato  
}
```

## Passaggio per risultato

Duale del passaggio per valore: main <- proc

```
void foo (result int x) {x = 8;}  
...  
y = 1;  
foo(y);  
foo(V[y+1]);
```

- l'attuale deve essere un l-value, y o V[y+1]
- al ritorno da foo, il valore di x è assegnato all'attuale y o V[y+1]
- nessun legame tra y iniziale del chiamante e x nel corpo di foo
  - non è possibile trasmettere dati alla procedura mediante il parametro
- pragmatica: usato quando una funzione deve restituire più di un singolo risultato
- Ada: out

## Passaggio per valore/risultato

Insieme valore+risultato. Pragmatica: main <-> proc evitando aliasing

```
void foo (value-result int x)  
    { x = x+1; x := y + 1; }  
...  
y = 8;  
foo(y);
```

- l'attuale deve essere un l-value
- il formale x è 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
- costoso per dati grandi: copia
- Ada: in out (ma solo per dati piccoli; per dati grandi passa riferimento)

## Passaggio per nome

L'espressione del parametro attuale viene passata, senza valutazione, alla procedura

Regola di copia:

- non si forza la valutazione del parametro attuale al momento della chiamata
- una chiamata alla procedura P produce l'effetto di
  - eseguire il corpo di P valutando, ogni volta che il parametro formale viene usato nel corpo l'espressione attuale

```
int sel (name int x, y) {  
    return (x==0 ? 0 : y);}  
...  
z = sel(w, z/w);
```

- Introdotto in Algol-W come il default

## Simulazione passaggio per riferimento

- Può simulare il passaggio per riferimento:

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

- Se il parametro formale appare a sinistra dell'assegnamento, il compilatore controlla che il parametro attuale sia un l-value

## Valutazione multipla

Se un parametro formale appare più volte nel codice il parametro attuale viene valutato più volte

- se la valutazione comporta effetti collaterali questi vengono ripetuti

```
int V[] = {0, 1, 2, 3};
int y = 1;
void fie (name int x){
    int y = 2;
    x = x + y;
}
fie( V[y++] );
```

## Cattura delle variabili - ambiente di valutazione

```
int y = 1;
void fie (name int x){
    int y = 2;
    x = x + y;
}
fie(y);
```

- conflitto (e “cattura”) di variabili
- in quale ambiente valutare il parametro formale x
  - nel corpo procedura? No
  - nell'ambiente della chiamata `fie(y)`? Sì
- Si realizza una “macro espansione”, in modo semanticamente corretto evitando la [cattura delle variabili](#)

## Implementazione: chiusura

Al momento della chiamata:

- oltre all'espressione del parametro attuale `exp` bisogna fornire
- l'ambiente di valutazione del parametro `env`

La coppia `<exp, env>` prende il nome di **chiusura**.

Come rappresentare la chiusura?

- **exp** : un puntatore al codice di `exp`
- **env** : un puntatore di catena statica (sullo stack) al record di attivazione del blocco di chiamata



## Uso:

- complesso da implementare.
- Algol 60 e W: implementazioni storiche
- quasi non utilizzato nei linguaggi imperativi attuali opzionale in Nim, Scala
- default nei linguaggi funzionali lazy: Haskell, ma in una versione più efficiente
  - **call-by-need**, valuto l'argomento al più una volta
  - con l'assenza dei side-effect, valutazioni multiple restituiscono sempre lo stesso valore
    - call-by-need e call-by-name stesso comportamento
    - call-by-need più efficiente
- chiusure utilizzate per il passaggio di funzioni come argomento

## Parametri di default

In molti linguaggi,

Ada, C++, C#, Fortran, Python, JavaScript, Ruby, ...

possibili funzioni con valori di default per alcuni argomenti

- è possibile omettere alcuni argomenti
- nel qual caso la funzione usa gli argomenti di default

## Simulazione del passaggio per nome:

- si sfrutta la similitudini tra passaggio di funzione e passaggio per nome
- il parametro è una funzione senza argomenti
- il corpo della funzione non viene valutato al passaggio ma nell'ambiente del chiamante
- un **thunk**, nel gergo Algol

Esempio in Scheme:

```
(define (doublePlusOne e)
  (+ (e) (e) 1))
```

```
(define x 2)
```

```
(doublePlusOne (lambda () (+ x 3)))
```

**thunk**: meccanismo generale per simulare la valutazione lazy (per nome) in un linguaggio con valutazione eager (per valore)

## Esempio in Python

```
def printData(firstname, lastname = 'Mark', subject = 'Math'):
    print(firstname, lastname, 'studies', subject)
```

```
# 1 positional argument
printData('John')
# 2 positional arguments
printData('John', 'Gates')
printData('John', 'Physics')
```

```
# Output:
John Mark studies Math
John Gates studies Math
John Physics studies Math
```

## Parametri con nome:

L'accoppiamento parametri attuali - parametri formali viene determinato dalla posizione,

in alternativa, in alcuni linguaggi, possibile specificare esplicitamente il ruolo svolto dai parametri attuali:

```
def printData(firstname, lastname = 'Mark', subject = 'Math'):
    print(firstname, lastname, 'studies', subject)
```

*# Mixed passing is possible*

```
printData(firstname='John', subject='Physics')
printData('John', subject='Physics')
```

*# Output:*

```
John Mark studies Physics
John Mark studies Physics
```

## Funzioni come argomento, semantica

Caso standard:

- il chiamante `main` passa una funzione `f`
- funzione `f` che verrà valutata, eventualmente più volte, nel chiamato `g`

Problema principale:

- quale ambiente usare nel valutare `f` come risolvere le variabili non locali
- politiche di scope: (statico, dinamico)
- ma non solo, politiche di binding: (deep, shallow)

## Funzioni di ordine superiore

Alcuni linguaggi permettono di:

- passare funzioni **come argomento** di altre funzioni (procedure)
  - caso relativamente semplice
  - possibile in molti linguaggi imperativi
  - funzioni come **oggetti di secondo livello**
- restituire funzioni **come risultato** di funzioni
  - caso più complesso
  - possibile nelle versioni più recenti di linguaggi imperativi
  - tipico dei linguaggi funzionali
  - funzioni come **oggetti di primo livello**

## Ambiente esterno del parametro funzione: esempio

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

Tre momenti della funzione `f`

- definizione di `f`: `int f (int y)`
- passaggio di `f` come argomento: `int z = g(f)`
- chiamata di `f`, tramite il nome `h`: `h(3) + x`

Tre possibili alternative per l'ambiente esterno di `f`

## Politiche di binding

In quale ambiente viene valuta  $f$ , chiamata con il nome  $h$ , dentro  $g$ ?

- scope statico: uso l'ambiente della definizione (come sempre)
- scope dinamico: due alternative:
  - ambiente al momento della creazione del legame  $h \rightarrow f$  ossia della chiamata di  $g$  con parametro  $f$   
**deep binding**
  - ambiente al momento della chiamata di  $f$ , in  $g$ , via  $h$   
**shallow binding**

## Politiche di binding: esempio

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

- tre dichiarazioni di  $x$
- quando  $f$  sarà chiamata (tramite  $h$ )
  - quale  $x$  (non locale) sarà usata?
- in scope statico, la  $x=1$  esterna
- in scope dinamico,
  - deep binding: le  $x=4$  del blocco di chiamata
  - shallow binding: la  $x=2$  interna

## Implementazione uso delle chiusure

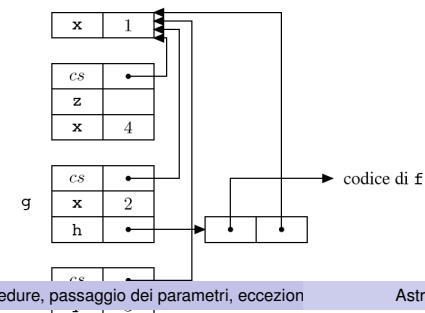
Analoga alla chiamata per nome:

alla chiamata di una procedura  $g$  con parametro attuale una procedura  $f$

- si inserisce nel RdA di  $g$ , in corrispondenza ad  $f$ , una closure  $\langle \text{code}, \text{env} \rangle$ 
  - un riferimento al codice della procedura  $f$
  - un riferimento al suo ambiente non locale di  $h$ .
- Alla chiamata della procedura  $f$  dentro  $g$ 
  - si alloca (come sempre) il record di attivazione
  - usa come puntatore alla catena statica il riferimento (ambiente) fornito dalla chiusura  $\text{env}$ .

## Deep binding e chiusure, scope statico

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



## Scope dinamico: implementazione

### Shallow binding

- non necessita la **chiusura**
  - alla parametro funzione si associa solo il codice
  - per accedere a x, risali la pila
  - uso delle strutture dati solite (A-list, CRT)

### Deep binding

- al parametro funzione si associa la chiusura
- per “ricordarsi” il RdA della chiamata
- parte della pila va saltata
  - per le procedure chiamate via parametro devo usare un link statico

## Deep binding e chiusure, scope dinamico

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

## Deep vs shallow binding con scope statico

A prima vista deep o shallow non fa differenza con scope statico

- 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 la procedura passata come parametro,
  - la procedura viene dichiarata più volte
- la politica di binding cambia la dichiarazione da prendere in considerazione,
- accade in presenza di ricorsione

Per coerenza, viene sempre usato deep binding

- implementato con chiusure

## Esempio

```
void foo (int f(), int x){
    int fie(){
        return x; }
    int z;
    if (x==0) z=f()
    else foo(fie,0);}
int g(){
    return 1;}
...
foo(g,1)
```

- due chiamate ricorsive di foo
- con due diversi valori di x
- fie dichiarata due volte, con due x non locali diversi
- quale dichiarazione considerare?

## Funzioni come argomento in C

```
void mapToInterval ( void (*f)(int), int max ) {
    for ( int ctr = 0 ; ctr < max ; ctr++ ) {
        (*f)(ctr);
    }
}
void print ( int x ) {
    printf("%d\n", x); }
void main () {
    ...
    mapToInterval(print, 100) /* notare print senza parentesi */
```

L'asterisco nel parametro formale, \*f necessario

Implementazione semplificata:

- in C non ci sono funzioni annidate,
- non servono chiusure,
- basta un puntatore alla funzione,
  - che verrà valutata nel ambiente globale

## Funzioni come risultato, caso semplice:

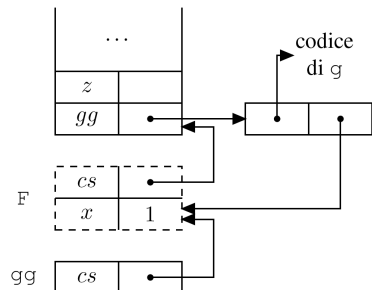
Si restituisce codice e ambiente non locale

```
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.

## Funzioni come risultato, caso complesso

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



## Implementazione delle funzioni come risultato

Linguaggi funzionali:

- Uso delle chiusure, ma ...
- i record di attivazione possono dover persistere indefinitamente
  - perdita proprietà dello stack (LIFO)
- non si usa uno stack di attivazione
  - record di attivazione sullo heap
  - le catene statica e dinamica collegano i record
  - invoca il **garbage collector** quando necessario

Linguaggi imperativi, con funzioni di ordine superiore:

- gestione sofisticata dei RdA

## Implementazione funzioni di ordine superiore

Caso più semplice:

- Funzioni come argomento
  - si passa una chiusura

Caso più complesso:

- Funzione ritornata da una chiamata di procedura
  - occorre mantenere l'ambiente della funzione restituita:
  - disciplina a pila per i RdA non funziona più.

## Funzionamento:

- si definiscono **blocchi protetti**, blocco di codice, con associato:
  - insieme di **eccezioni** sollevabili all'interno,
  - relativi **gestori**
- il programma può sollevare l'eccezione **raise an exception**
  - la normale computazione viene interrotta
  - si cerca:
- un **gestore** per l'eccezione sollevata,
  - codice da eseguire solo nel caso venga sollevata quell'eccezione
  - relativa ad un **blocco protetto**
- La ricerca del gestore comporta:
  - terminazione comandi correnti
  - uscita da cicli
  - uscita da procedure
    - RdA vengono deallocati

## Gestione delle eccezioni: "uscita strutturata"

Meccanismo per gestire eventi eccezionali:

- errori
- situazioni non previste
- ma anche terminare in anticipo la computazione, perché si sa già il risultato

Primi linguaggi gestivano attraverso istruzioni di salto: GOTO

- esempio di uso del GOTO non sostituibile dai meccanismi soliti (cicli, procedure)
- con GOTO difficile implementare correttamente l'uscita da una procedura, bisogna gestire lo stack dei RdA
- per una gestione corretta si introduce un costrutto apposito: le **eccezioni**

## Riassumendo

Tre costrutti:

- definizione delle **eccezioni**
- definizioni di **blocchi protetti**, con relativi **gestori** delle eccezioni
- **sollevamento** dell'eccezione

## Esempio Java

- la funzione `average` calcola la media di un vettore;
- se il vettore è vuoto, solleva un'eccezione
- le eccezioni sono sottoclassi di una classe `Throwable`
- devo dichiarare, `throws` se una funzione può generare un'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);
};
...
try{...
    average(W);
    ...
}
catch (EmptyExcp e) {write('Array_vuoto'); }
```

## Esempio Java: gestione di un'eccezione

- il gestore è legato al blocco di codice protetto
- l'esecuzione del gestore rimpiazza la parte di blocco che doveva essere ancora eseguita
- possibile assegnare il parametro `x` in `EmptyExcp`
- comunico dei valori attraverso l'eccezione

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

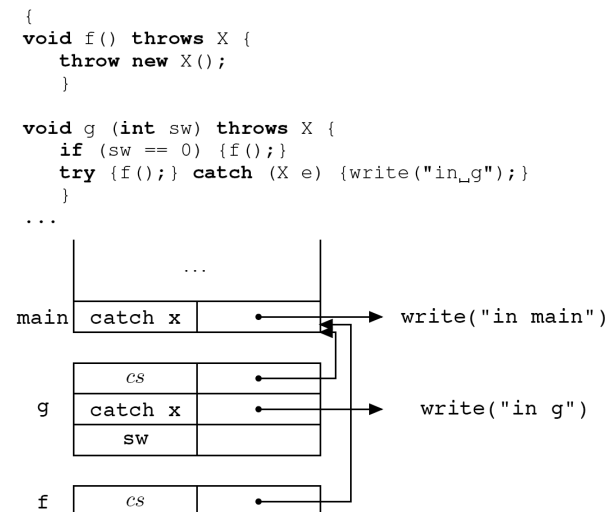
int average(int[] V) throws EmptyExcp(){
    if (length(V)==0) throw new EmptyExcp();
    else {int s=0; for (int i=0, i<length(V), i++) s=s+V[i];}
    return s/length(V);
};
...
try{...
    average(W);
    ...
}
catch (EmptyExcp e) {write('Array_vuoto'); }
```

## Propagare un'eccezione

Se l'eccezione non è gestita nella procedura corrente:

- la procedura termina, l'eccezione è **ri-sollevata nel punto di chiamata**
  - l'eccezione è propagata lungo la **catena dinamica** si toglie un RdA, e si passa il controllo al chiamante
- fino a quando si incontra un gestore o si raggiunge il top-level,
  - fornisce un gestore default, tipicamente:  
ferma un programma con un messaggio di errore

## Eccezione si propaga lungo la catena dinamica



## Eccezione si propaga lungo la catena dinamica

Conseguenza:

- a tempo di esecuzione non posso determinare il gestore di un'eccezione generata dentro una funzione (f)

## Codice

```
(define-struct node (left right))

(define tree (make-node (make-node 10 9)
                       (make-node 0 (make-node 1 5))))

(define (fold-product-aux nd)
  (cond
    [(number? nd)
     (if (equal? nd 0) (raise "zero-found") nd)]
    [(node? nd)
     (* (fold-product-aux (node-left nd))
        (fold-product-aux (node-right nd)))]))

(define (fold-product nd)
  (with-handlers
    ([ (lambda (e) (equal? e "zero-found")) (lambda (e) 0) ])
    (fold-product-aux nd)))
```

## Uso delle eccezioni per aumentare l'efficienza

Programma ricorsivo Scheme che

- moltiplica tutti i nodi di un albero binario,
- ogni nodo contiene un intero.

Soluzione per restituire immediatamente il risultato quando un nodo contiene 0:

- definisco una funzione ausiliaria di visita dell'albero genera l'eccezione `found-zero` se trova uno 0.
- funzione principale chiama l'ausiliaria con un gestore dell'eccezione
  - se è stata generata l'eccezione `found-zero`, restituisce 0
  - altrimenti il risultato della funzione ausiliaria,
- evito di dover continuare la visita quando so già il risultato finale

In Scheme non serve dichiarare le eccezioni prima di usarle

## Implementare le eccezioni

Soluzione naturale:

- all'inizio di un blocco protetto (try):
  - alloco un nuovo RdA con la lista dei gestori
- quando un'eccezione è sollevata:
  - cerco il gestore nel RdA corrente,
  - se non trovo scendo nella lista dei RdA alla ricerca di un gestore

ma inefficiente:

- ad ogni ingresso e uscita in un blocco protetto modificare lo stack
- nel caso più frequente, in cui non si verificano eccezioni, viene svolto lavoro inutile



## Implementare le eccezioni

Una soluzione più efficiente è quella di costruire una mappa dei blocchi protetti

- RdA inserito solo in corrispondenza delle procedure
- nel RdA di ogni procedura, una mappa dei blocchi protetti nella procedura per ogni blocco:
  - indirizzo di inizio - fine
  - quali eccezioni gestisce
  - i gestori delle eccezioni
- alla chiamata di un'eccezione, usando il suo indirizzo si consulta la mappa per decidere quale gestore invocare

## Esercizi: eccezioni

Descrivere il comportamento del seguente programma con ognuno dei diversi meccanismi di passaggio dei parametri: valore, riferimento, costante, valore-risultato, nome. La valutazione delle espressioni viene fatta da sinistra a destra.

```
{ int y=1
  void f(___ int x) throws E(){
    if (x++ == y++)
      {throw new E();}
    x++;
  }
  try{ f(y) } catch (E){write(y++)};
  write(y);
}
```

## Esercizi: passaggio dei parametri

Descrivere il comportamento del seguente programma con ognuno dei diversi meccanismi di passaggio dei parametri: valore, riferimento, costante, valore-risultato, nome. La valutazione delle espressioni viene fatta da sinistra a destra.

```
int y=1
void f(___ int x){
  x++; y++;
  write(x);
  x++;
}
f(y);
write (y);
```

## Scope

- Si mostri l'evoluzione dello stack di attivazione e dell'output del seguente frammento di programma C-like con: scope statico, assegnamento che calcola l'l-value prima dell'r-value, valutazione degli argomenti da sinistra a destra, indici dei vettori che iniziano da 0:

```
char x[10] = "abcdefghij";
int i = 0;
char magic(ref char a, name j){
  char c = a;
  x[j] = x[++i] = c++;
  write(a, j) ;
  return c ;
}
write(magic(x[i++], i++) );
write(x, i);
```

Si mostri l'evoluzione dello stack di attivazione e dell'output del seguente programma espresso C-like con:

- scope statico e deep binding,
- assegnamento che calcola l-value dopo r-value,
- valutazione delle espressioni da destra a sinistra,
- argomenti chiamate da destra a sinistra e
- indici dei vettori iniziati da 0 (% è il modulo aritmetico,  $-1\%3 = 2$ ).

```
int x=4, y=2, v[3]={x--, y, ++y};
int F(int R(valres int), ref int y){
    int x=3;
    write(R(y));
    return(y -= x);}
int G(name int x){
    int H(valres int w){
        return(v[(w++)%3]);
    }
    write(v[(y++) %3] );
    return(F(H, x) - y++);
}
write(G(v[x %3]));
```