FUNZIONI E PROCEDURE

Breve storia dei sottoprogrammi

- Astrazione di una sequenza di istruzioni
- Un frammento di programma (sequenza di istruzioni) risulta utile in diversi punti del programma
  - Riduco il "costo della programmazione" se posso dare un nome al frammento e qualcuno per me inserisce automaticamente il codice del frammento ogni qualvolta nel "programma principale" c'è un'occorrenza del nome
    - ✓**macro** e **macro-espansione**

MACRO in C

```
#define MULT(x, y) x * y
```

Cosa viene assegnato a z?

```
int z = MULT(3 + 2, 4 + 2);
```

13!!!

```
int z = 3 + 2 * 4 + 2;
// 2 * 4 valutato prima
```

Code Inlining

3

Breve storia dei sottoprogrammi

- Si riduce anche l'occupazione di memoria se esiste un meccanismo che permette al programma principale
  - ✓di trasferire il controllo ad una unica copia del sottoprogramma memorizzata separatamente
  - ✓di riprendere il controllo quando l'esecuzione del frammento è terminata
  - ✓Meccanismo supportato direttamente dall'hardware (**codice rientrante**)

4

Breve storia dei sottoprogrammi

- Ancora meglio se permettiamo **astrazione via parametrizzazione**
  - Astraendo dall'identità di alcuni dati
  - La cosa è possibile anche con le macro e il codice rientrante
    - ✓Macroespansione con rimpiazzameno di entità diverse
    - ✓Associazione di informazioni variabili al codice rientrante

5

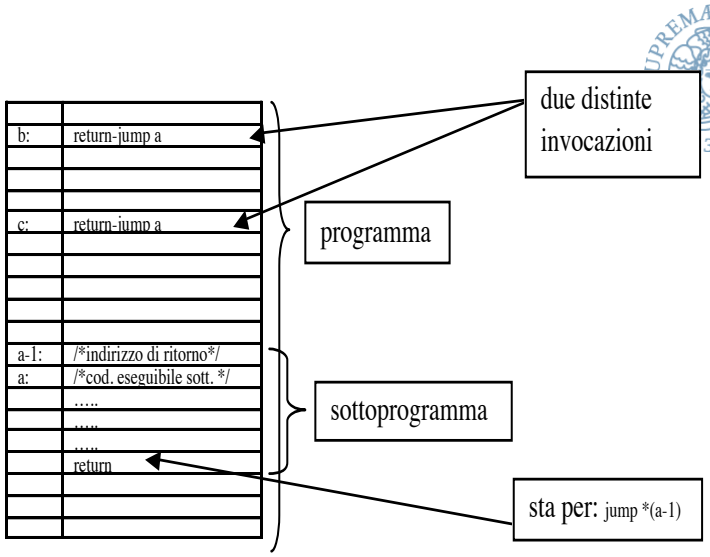
Cosa fornisce l'hardware?

- Primitiva di **return jump** con opportune strutture ausiliarie
- Viene eseguita (nel programma chiamante) l'istruzione

  return jump a

  memorizzata nella cella **b**
  - Il controllo viene trasferito alla cella **a** (entry point della subroutine)
  - L'indirizzo dell'istruzione successiva del chiamante (**b + 1**) viene memorizzato in qualche posto noto, per esempio nella cella (**a - 1**) (**punto di ritorno**)
- Quando nella subroutine si esegue una operazione di return
  - il controllo ritorna all'istruzione (del programma chiamante) memorizzata nel punto di ritorno

6



7

Archeologia: FORTRAN

- Una **subroutine** è un pezzo di codice compilato, a cui sono associati
  - Una cella destinata a contenere (a tempo di esecuzione) i punti di ritorno relativi alle chiamate
  - Alcune celle destinate a contenere i valori degli eventuali parametri
  - Ambiente locale è statico

8

Semantica della subroutine à la FORTRAN

- Si può definire facilmente attraverso la ***copy rule statica*** ("**macroespansione**")
  - Ogni chiamata di sottoprogramma è *testualmente rimpiazzata* da una copia del codice
    - ✓facendo qualcosa per i parametri
    - ✓ricordandosi che le dichiarazioni sono eseguite una sola volta
- Il sottoprogramma non è semanticamente qualcosa di nuovo è solo un (importante) strumento metodologico (astrazione!)

9

Semantica della subroutine à la FORTRAN

- Osservazione: non è compatibile con la ricorsione
  - la macroespansione darebbe origine ad un programma infinito
  - l'implementazione à la FORTRAN (con un solo punto di ritorno) non permetterebbe di gestire più attivazioni presenti allo stesso tempo
- Il fatto che le subroutine FORTRAN siano concettualmente una cosa statica fa sì che
  - non esista di fatto il concetto di attivazione
  - l'**ambiente locale** sia necessariamente **statico**

10

Attivazione

- Se ragioniamo in termini di attivazioni, la semantica può essere ancora definita da una ***copy rule***, ma ***dinamica***
  - ogni chiamata di sottoprogramma è ***rimpiazzata a tempo di esecuzione*** da una copia del codice
- Il sottoprogramma è ora semanticamente qualcosa di nuovo
- Ragionare in termini di attivazioni
  - rende naturale la ricorsione
  - porta ad adottare la regola dell'**ambiente locale dinamico**

11

Le strutture di implementazione

- Invece delle informazioni **staticamente** associate al codice compilato di FORTRAN
  - punto di ritorno, parametri, ambiente e memoria locale
- si usano i **record di attivazione**
  - contenenti le stesse informazioni

  ma associati dinamicamente alle varie chiamate di sottoprogrammi
- Dato che l'accesso ai sottoprogrammi segue una politica LIFO
  - l'ultima attivazione creata nel tempo è la prima che ritorna

  possiamo organizzare i record di attivazione in una pila

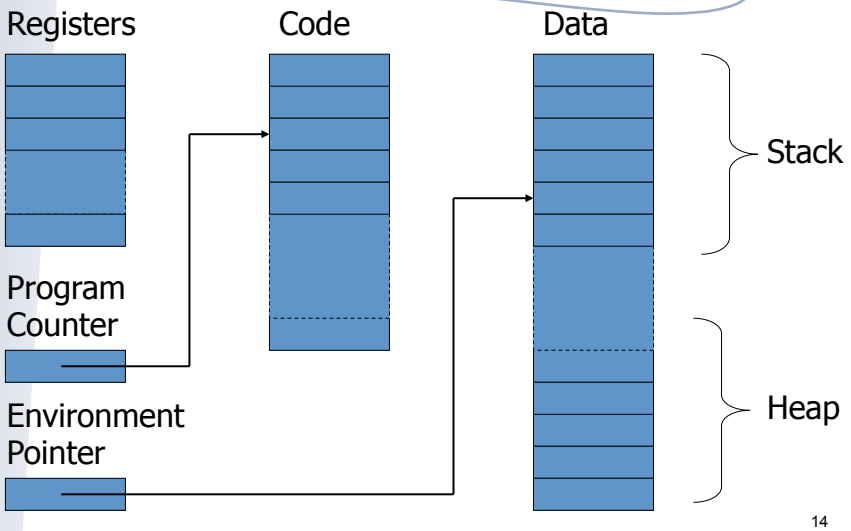
12

Cosa è un sottoprogramma vero

- Astrazione procedurale (operazioni)
  - Astrazione di una sequenza di istruzioni
  - Astrazione via parametrizzazione
- Luogo di controllo per la gestione dell'ambiente e della memoria
  - In assoluto, l'aspetto più interessante dei linguaggi, intorno a cui ruotano tutte le decisioni semantiche importanti
  - Binding: statico o dinamico

13

Modello di macchina Hw



14

Meccanismo Call/Return di sottoprogramma

- Chiamante
  - Crea una istanza del record di attivazione
  - Salva lo stato dell'unità corrente di esecuzione
  - Effettua il passaggio dei parametri
  - Inserisce il punto di ritorno
  - Trasferisce il controllo al chiamato
- Chiamato (prologo):
  - Salva il valore corrente di Environment Pointer (EP) e lo memorizza nel link dinamico.
  - Definisce il nuovo valore di EP
  - Alloca le variabili locali

15

Meccanismo Call/Return di sottoprogramma

- Chiamato (epilogo)
  - Eventuale passaggio di valori (dipende dalla modalità di passaggio dei parametri - lo vediamo dopo)
  - Il valore calcolato dalla funzione viene trasferito al chiamante
  - Ripristina le informazioni di controllo (il vecchio valore di EP salvato come link dinamico)
  - Ripristina lo stato di esecuzione del chiamante
  - Trasferisce il controllo al chiamante

16

Come si realizza?

- Partiamo dalla cosa più semplice: i blocchi
  - Sostanzialemente delle procedure senza none e senza parametri

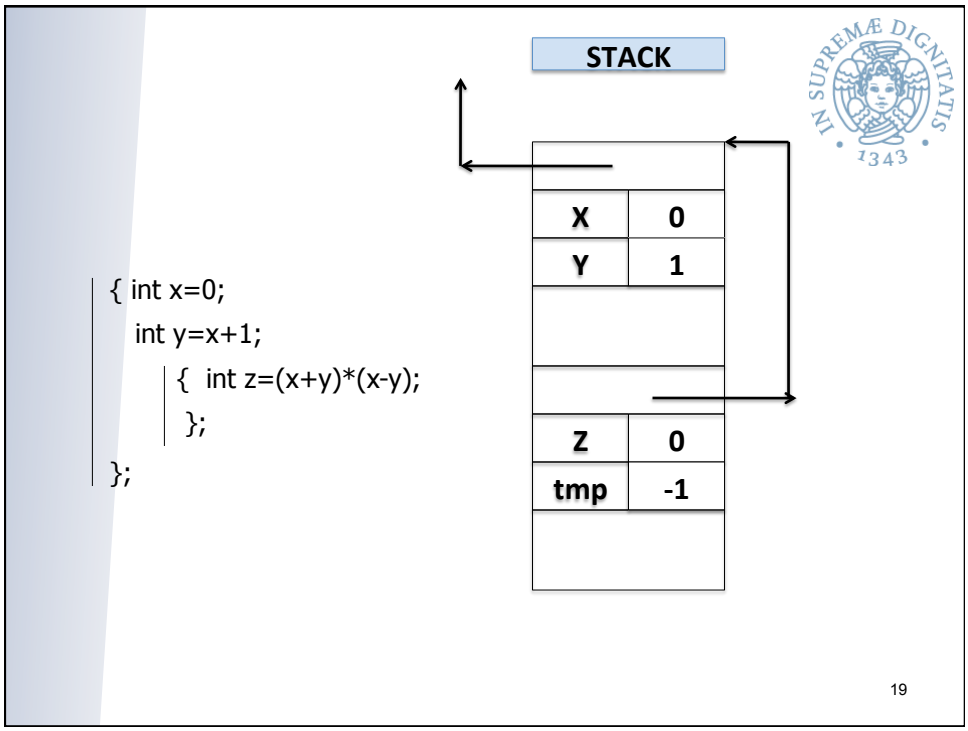
In-line Blocks

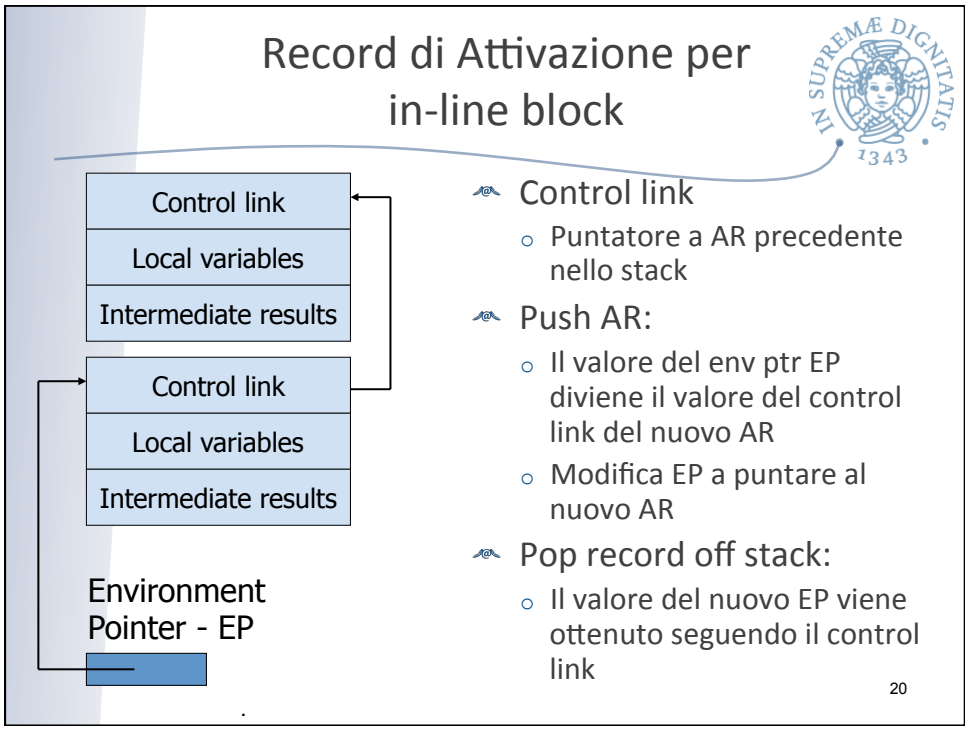
- Record di attivazione -- Activation record
  - Tipo di dati di sistema memorizzato nello stack
  - Gestisce l'ambiente locale
- Esempio

```
{ int x=0;
    int y=x+1;
        {  int z=(x+y)*(x-y);
         };
};
```

```
Push AR con spazio per x, y
Assegna i valori a x, y
        Push AR per blocco interno
        Assegna valore a z
        Pop AR per blocco interno
Pop AR per blocco esterno
```

Occorre prevedere spazio per memorizzare i risultati intermedi (x+y), (x-y)


STACK
{ int x=0;
int y=x+1;
{ int z=(x+y)*(x-y);
};
};
X
0
Y
1
Z
0
tmp
-1
19


Record di Attivazione per in-line block
Control link
Local variables
Intermediate results
Control link
Local variables
Intermediate results
Environment Pointer - EP
Control link
Puntatore a AR precedente nello stack
Push AR:
Il valore del env ptr EP diviene il valore del control link del nuovo AR
Modifica EP a puntare al nuovo AR
Pop record off stack:
Il valore del nuovo EP viene ottenuto seguendo il control link
20

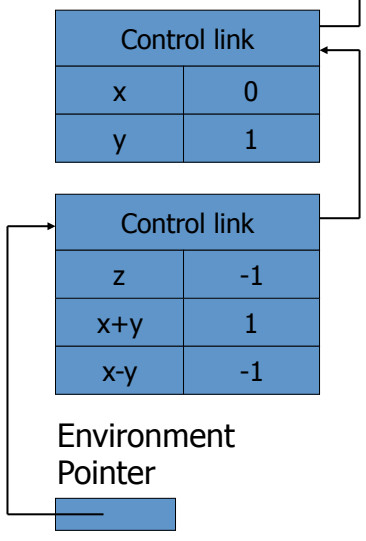
Record Attivazione

Control link Puntatore di catena dinamica
Variabili locali
Risultati intermedi

21

Esempio Completo

```
{ int x=0;
   int y=x+1;
      {  int z=(x+y)*(x-y);
       };
};
```



22

E le regole di scope?

- Variabili e ambiente
  - x, y locali al blocco esterno
  - z locale al blocco interno
  - x, y non locali per il blocco interno

```
{ int x=0;
  int y=x+1;
      {  int z=(x+y)*(x-y);
      };
};
```

- Static scope
  - Riferimenti non locali si risolvono nel più vicino blocco esterno
- Dynamic scope
  - Riferimenti non locali si risolvono nel AR precedente sullo stack

Nel caso di in-line block le due nozioni coincidono

23

Analisi

- Il meccanismo dello stack dei record di attivazione è un meccanismo efficiente
- Per risolvere un riferimento locale basta accedere al record di attivazione in testa allo stack (tramite EP) e poi cercare il nome nell'ambiente locale memorizzato nel record di attivazione
- Maggiore efficienza se potessimo eliminare i nomi dal codice in esecuzione (dettagli in seguito)

24

Funzioni e procedure

- Procedure (Algol)

```
procedure P (<pars>)
   begin
        <local vars>
        <proc body>
   end;
```

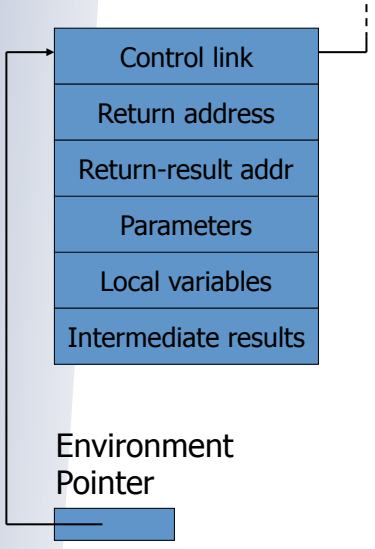
- Funzioni (C)

```
   <type> function f(<pars>)
{
        <local vars>
        <function body>
}
```

- Cosa ci deve stare nel record di attivazione?
  - Parametri
  - Indirizzo di ritorno
  - Variabili locali, risultati intermedi
  - Valore restituito (caso particolre di risultato intermedio)
  - Spazio per il valore restituito al momento del ritorno

25

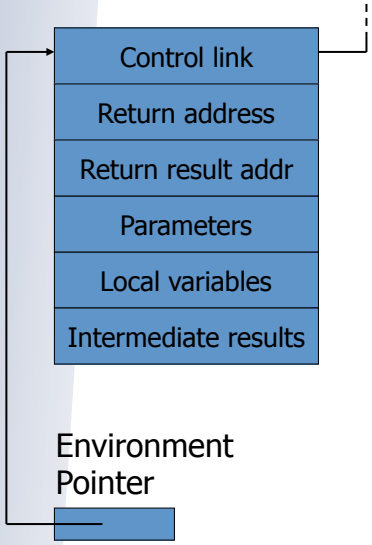
Funzioni: struttura AR



- Return address
  - Indirizzo della istruzione da eseguire quando viene restituito il controllo al chiamante
- Return-result address
  - Indirizzo nel AR del chiamante dove memorizzare il risultato
- Parameters
  - Parametri della funzione

26

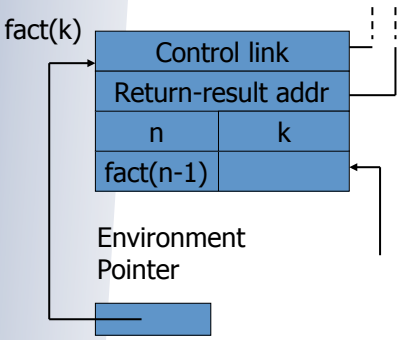
Esempio



- Il solito fattoriale
  fact(n) = if n<= 1 then 1
  else n * fact(n-1)
- Return result address
  - Indirizzo dove memorizzare fact(n)
- Parameter
  - Associazione tra n e il valore del parametro attuale
- Intermediate result
  - Spazio per memorizzare il valore di fact(n-1)

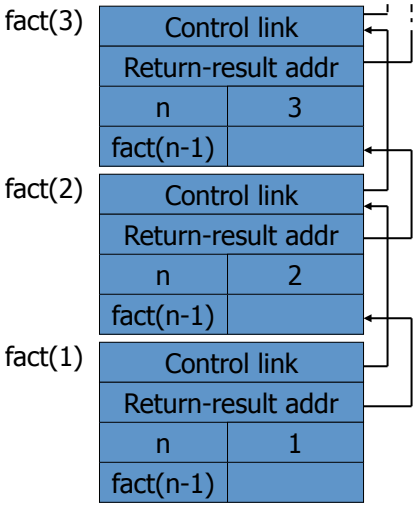
27

Call & …



fact(n) = if n<= 1 then 1
else n * fact(n-1)

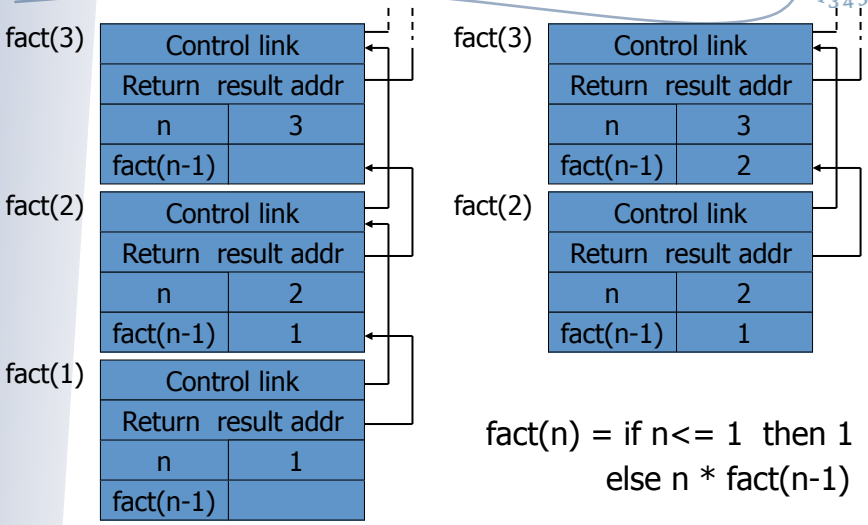
Per semplicità non inseriamo il valore del return address



Continua →

28

….. & return



fact(n) = if n<= 1 then 1
else n * fact(n-1)

29

Altri aspetti

- Passaggio dei parametri
  - Per valore: copiare il valore del parametro attuale nello spazio previsto nel record di attivazione
  - Per riferimento: copiare il valore del ptr nel record di attivazione
- Variabili globali
  - Le variabili globali sono memorizzate nel record di attivazione che sta in fondo allo stack (il primo a essere creato)
- Esaminate questi aspetti con un semplice debugger!!

30

Passaggio dei parametri

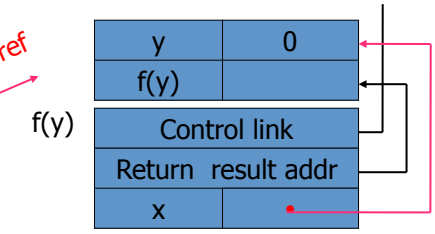
- L-values & R-values: Assegnamento y := x
  - ✓ Identificatore sulla sinistra dell'assegnamento denota la locazione e viene solitamente chiamato L-value
  - ✓ Identificatore sulla destra fa riferimento al contenuto della locazione e viene chiamato R-value
- Per riferimento
  - Memorizzare L-value (indirizzo di x) nel record di attivazione
  - Il corpo della funzione può modificare il parametro attuale
  - Aliasing: parametro formale e parametro attuale
- Per valore
  - Memorizzare R-value (contenuto di x) nel record di attivazione
  - Il corpo della funzione non può modificare il valore del parametro attuale
  - Non abbiamo aliasing

31

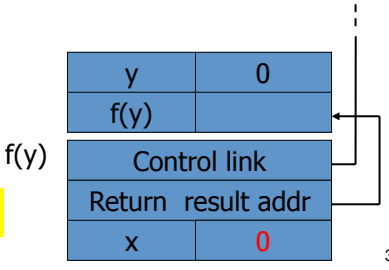
Esempio

```
function f (x) =
   { x = x+1; return x;  }
var y = 0;
println (f(y)+y);
```

pass-per-ref



pass-per-val



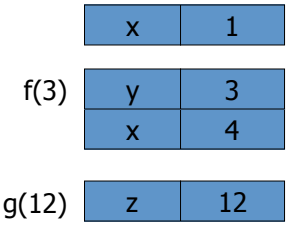
Cosa stampa nei due casi?

32

Variabili non locali

- Due alternative
  - Static scope (scoping statico)
  - Dynamic scope (scoping dinamico)
- Esempio

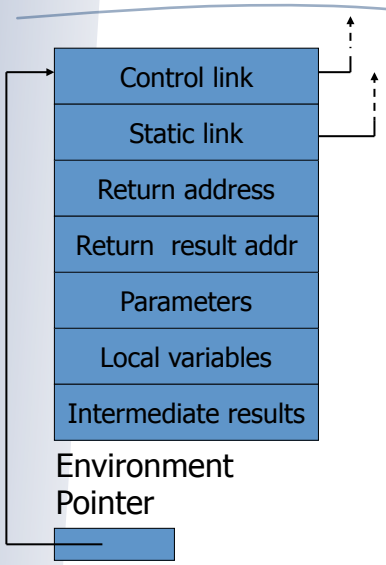
```
var x=1;
function g(z) {
    return x+z; }
function f(y) {
    var x = y+1;
    return g(y*x);  }
f(3);
```



Quale è il riferimento corretto di x nel valutare x+z ?

33

Scoping Statico



- Control link
  - Puntatore al AR che era in testa alla pila
- Static link
  - Puntatore al AR che contiene il blocco più vicino che racchiude il codice in esecuzione
- Analisi
  - Control link memorizza il flusso dinamico di esecuzione
  - Static link dipende dalla struttura sintattica del programma

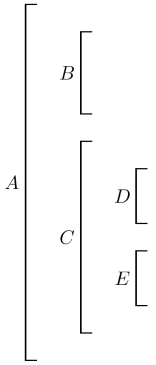
34

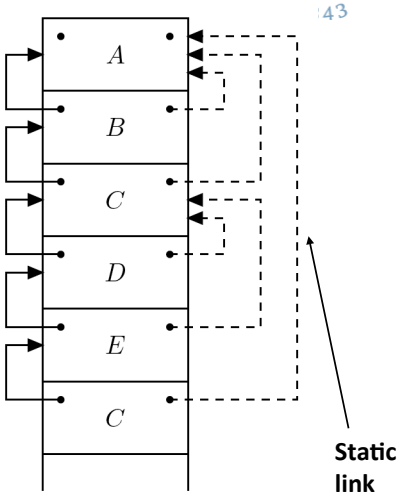
Static Link

- Lo static link del AR di una funzione A è il puntatore al record di attivazione del blocco dove A è stata dichiarata.
- La catena statica di un AR implementa la struttura sintattica dell'AR sulla catena dinamica
- *Risolvere un riferimento non locale significa trovare l'istanza del record di attivazione dove il riferimento non locale è stato dichiarato*

35

- Sequenza di chiamate a run time
  A, B, C, D, E, C

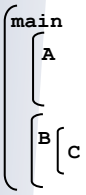


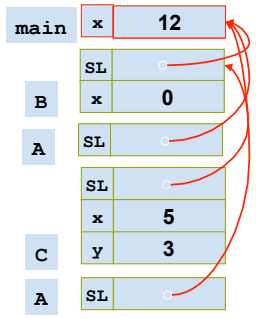


36

Esempio

```
{int x;
 void A(){
    x=x+1;}
 void B(){
   int x;
   void C (int y){
      int x;
     x=y+2; A();
   }
   x=0; A(); C(3);
 }
 x=10;
 B();
}
```





37

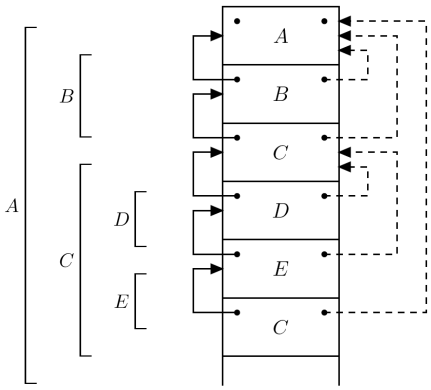
Come viene determinata la catena statica a run-time

- Quali operazioni deve effettuare il supporto a tempo di esecuzione per determinare il link statico del chiamato?
  - È il chiamante a determinare il link statico del chiamato
- Info a disposizione del chiamante:
  - annidamento statico dei blocchi (determinata dal compilatore staticamente)
  - proprio AR

38

Il chiamante ***C*** "conosce" l'annidamento dei blocchi:

- quando ***C*** chiama ***P***, sa se la definizione di ***P*** è:
  - immediatamente inclusa in ***C*** (k=0);
  - in un blocco k passi fuori ***C***
  - nessun altro caso possibile (perché)?
- nel caso a destra:
  - chiamate: **A, B, C, D, E, C**
- con i dati di catena statica:
  - **A; (B,0); (C,1); (D,0); (E,1); (C,2)**



- Se k=0:
  - ***C*** passa a ***P*** un puntatore al proprio AR
- Se k>0:
  - ***C*** risale la propria catena statica di **k** passi e passa a ***P*** il puntatore all'AR così determinato

39

Static Depth

- Si può eterminare staticamente il valore dell'annidamento delle procedure.
- Esempio:

```
Main {
   A {
        B {
        }B
   }A

   C{
   }C
}Main
```

40

Static Depth

- **Static Depth (SD)** = la profondità statica della dichiarazione
- SD può essere determinato staticamente: dipende solo dalla struttura sintattica del programma

```
Main {                  -- SD = 0
   A {                  -- SD = 1
       B {              -- SD =2
       }B
  }A

  C{                    --SD = 1
  }C
}Main
```

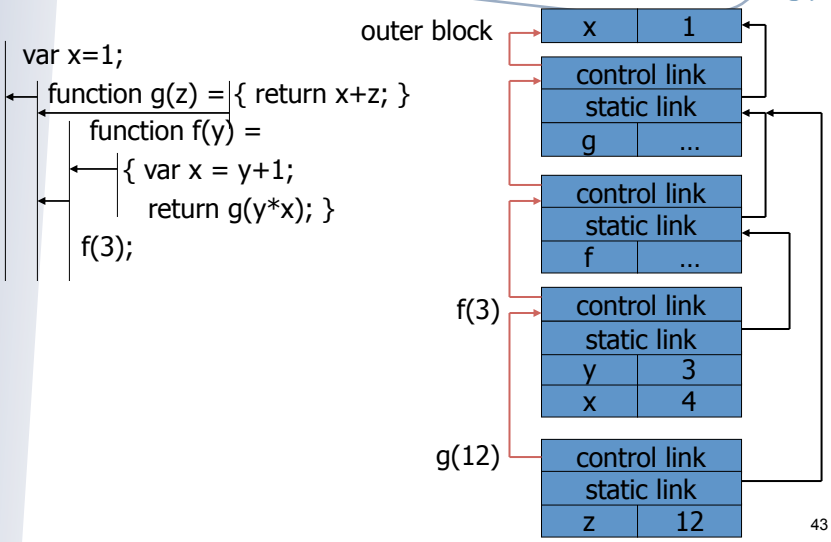
41

Chiamato esterno al chiamante

- Le regole dello scoping statico assicurano che perchè il chiamato sia visibile si deve trovare in un blocco esterno che includa il blocco del chiamante
- Questo implica che l'AR che contiene la dichiarazione del chiamato è già presente sullo stack.
- Assumiano che
  - ✓ **SD(Chiamante) = n**
  - ✓ **SD(Chiamato) = m**
  - ✓ Distanza statica tra chiamante e chiamato **n-m=k**
  - ✓ Il chiamante deve fare **k** passi lungo la sua catena statica per definire il valore del puntatore della catena statica del chiamato

42

Static scope with access links



Funzioni come valori

- Nei **linguaggi funzionali** le funzioni tipicamente sono **valori esprimibili** (possono essere risultato della valutazione di espressioni)
- Consideriamo i seguenti due casi:
  - Funzione passata come parametro attuale (semplice)
  - Funzione restituita come risultato di una altra funzione: può essere utilizzata nel seguito della computazione (più complicato)

Parametri funzionali

Haskell

```
int x = 4;
  fun f(y) = x*y;
    fun g(h) = let
          int x=7
           in
           h(3) + x;
    g(f);
```

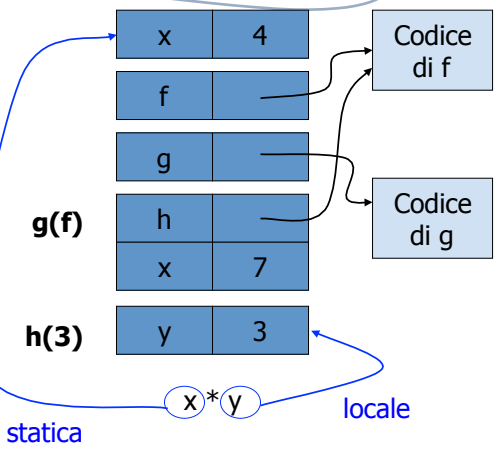
Pseudo-JavaScript

```
{ var x = 4;
  { function f(y) {return x*y};
    { function g(h) {
          var x = 7;
          return h(3) + x;
      };
      g(f);
} } }
```

Due dichiarazioni per la variabile **x**
Quale deve essere usata nella chiamata **g(f)**?

Parametri funzionali: Static Scope

```
int x = 4;
  fun f(y) = x*y;
    fun g(h) =
      let
        int x=7
      in
        h(3) + x;
    g(f);
```



Come si determina?

Non è un gioco

```
{var x = 4;
  {function f(y) {return x*y;}
    {function g(h) {
          var x = 7;
          return h(3) + x;}
    g(f);}
  }
}
```

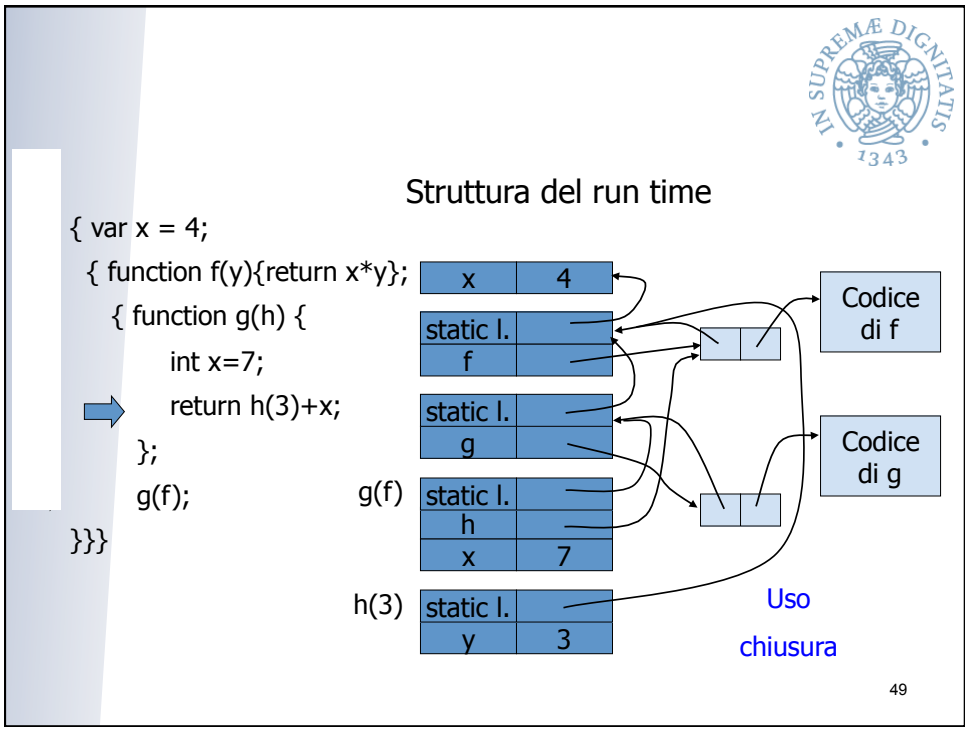
Valutiamo questo codice JavaScript su repl.it

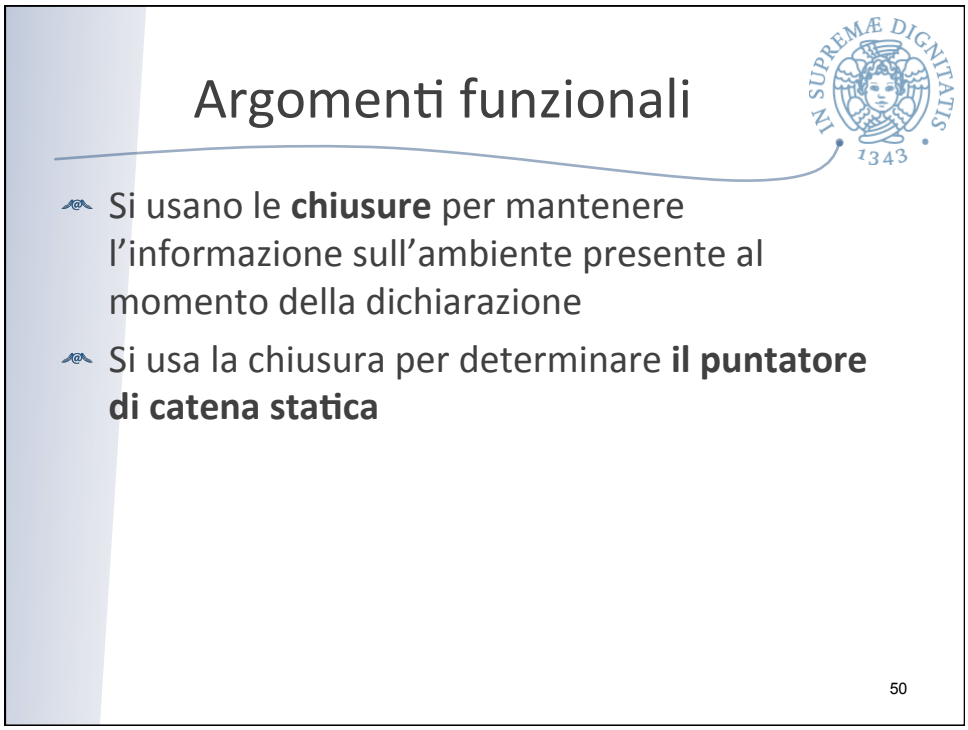
47

Chiusure

- Il valore di una funzione trasmessa come parametro è una coppia denominata **chiusura**
- *closure* = ⟨*env_dichiarazione*, *codice_funzione* ⟩
- Quando il parametro formale (funzionale) viene invocato,
  - Si alloca sullo stack AR della funzione
  - Si mette come valore del **puntatore di catena statica** il puntatore a *env_dichiarazione*

48


Struttura del run time
{ var x = 4;
{ function f(y){return x*y};
{ function g(h) {
int x=7;
return h(3)+x;
};
g(f);
}}}
x
4
static l.
f
static l.
g
g(f)
static l.
h
x
7
h(3)
static l.
y
3
Codice di f
Codice di g
Uso chiusura
49


Argomenti funzionali
Si usano le chiusure per mantenere l'informazione sull'ambiente presente al momento della dichiarazione
Si usa la chiusura per determinare il puntatore di catena statica
50

Funzioni come risultato

- Funzione che restituisce come valore una nuova funzione
  - Bisogna congelare l'ambiente dove la funzione è "dichiarata"
- Esempio

  function compose(f,g)

  {return function(x) { return g(f (x)) }};
- Funzione "dichiarata" dinamicamente
  - La funzione può avere variabili non locali
  - Valore restituito è una chiusura ⟨**env, code**⟩
  - **Attenzione:** l'AR cui punta **env** non può essere distrutto finché la funzione può essere usata: **retention**

Funzioni con "stato"

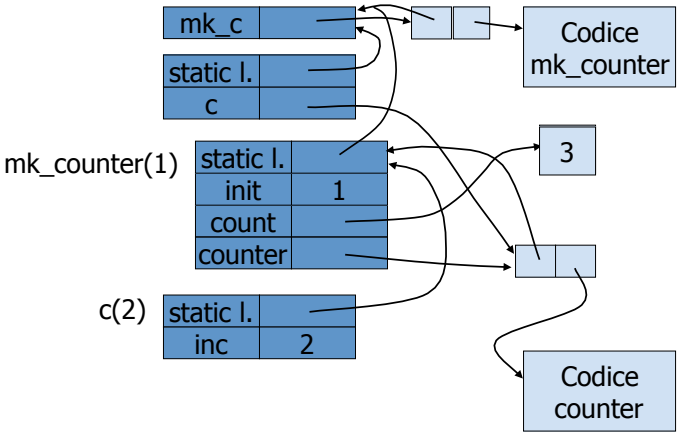
OCaml

- L'ambiente non locale di una funzione può essere sfruttato per incapsulare una variabile, dotando la funzione di "stato".

```
# let mk_counter (init : int) =  begin
      let count = ref init in
      let counter(inc:int) =
       (count := !count + inc; !count)
       in counter   (* funzione come risultato *)
   end;;
val mk_counter : int -> int -> int = <fun>
# let c = mk_counter 1;;
val c : int -> int = <fun>
# c(2) + c(2);;  (* cosa restituisce? *)
```

Riferimento a variabile non locale, preservata tra diverse attivazioni

```
let mk_counter (init : int) =  begin
    let count = ref init in
      let counter(inc:int) =
       (count := !count + inc; !count)
        in counter
    end;;
let c = mk_counter 1;;
c(2) + c(2);;
```



SCOPE DINAMICO

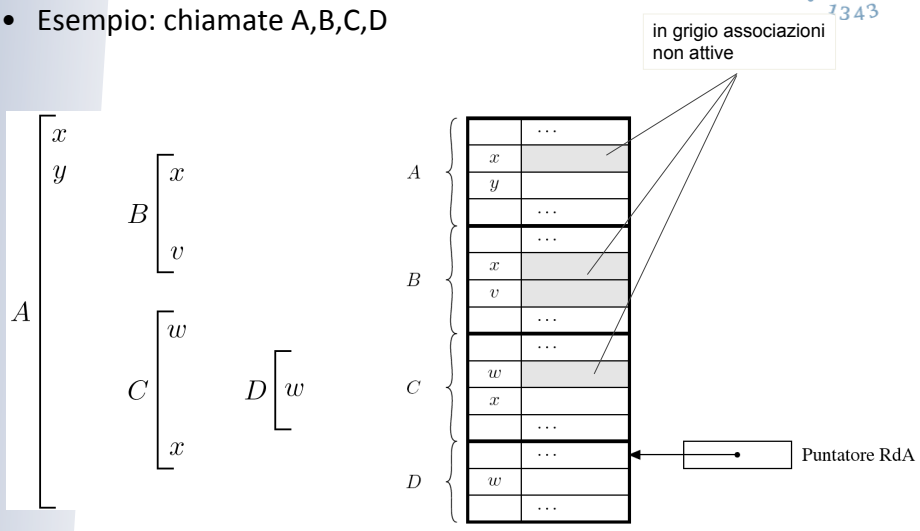
Regole Scope dinamico

- Con scope dinamico l'associazione nomi-oggetti denotabili dipende
  - dal flusso del controllo a run-time
  - dall'ordine con cui i sottoprogrammi sono chiamati
- La regola generale è semplice: l'associazione corrente per un nome è quella determinata per ultima nell'esecuzione (non ancora distrutta)

55

Implementazione ovvia

- Ricerca per nome risalendo la pila
- Esempio: chiamate A,B,C,D



56

Record di attivazione

- La strutturazione dei vari campi del record di attivazione cambia a seconda del linguaggio e dell'implementazione
- Gli identificatori generalmente non vengono memorizzati nel AR (se il linguaggio ha controllo statico dei tipi) ma sono sostituiti dal compilatore con un indirizzo relativo (offset) rispetto ad una posizione fissa del AR

57

esempio

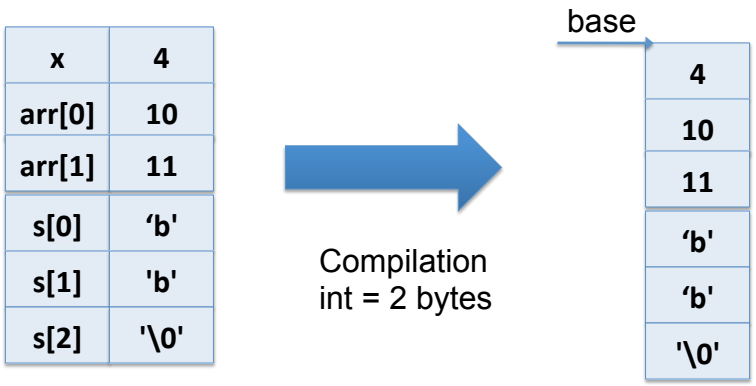

```
{ int x;
  int arr[2];
  char * s;
  x= 4;
  arr[0]=10;
  arr[1]=11;
 s = "bb";
}
```

x	4
arr[0]	10
arr[1]	11
s[0]	'b'
s[1]	'b'
S[2]	'\0'

C-Standard: C-string consists of an array
of characters terminated by the null character '\0'

58

Calcolo Offset



access(x) = base
access(arr[1]) = base + 4bytes

59

SCOPING STATICO E ANALISI STATICA

60

- Ambiente non locale con scoping statico:
  - numero di passi che a tempo di esecuzione devono essere fatti lungo la catena statica per trovare l'associazione (non locale) per l'identificatore "x" è uguale alla differenza fra le profondità di annidamento del blocco in cui "x" è dichiarato e quello in cui è usato
- Ogni *riferimento* a un identificatore *ide* nel codice può essere staticamente tradotto in una coppia (m,n) di numeri interi
  - m è la differenza fra le profondità di nesting dei blocchi (0 se *ide* si trova nell'ambiente locale)
  - n è la posizione relativa – offeset - (partendo da 0) della dichiarazione di *ide* fra quelle contenute nel blocco

61

Valutazione

- Efficienza nella rappresentazione:
  - l'accesso diventa efficiente (non c'è più ricerca per nome)
- si può economizzare nella rappresentazione degli ambienti locali che non necessitano più di memorizzare i nomi

62