Sottoprogrammi

I Sottoprogrammi

Qualche nozione introduttiva

DVAL: dominio dei valori denotabili, ovvero oggetti ai quali è possibile dare un nome: data object (variabili), indirizzi, sottoprogrammi, parametri, tipi, costanti, operazioni primitive, letterali

Ambiente: associazione ρ tra nomi e oggetti

$$\rho : \texttt{identificatori} \rightarrow \texttt{DVAL}$$

Controllo Sequenza Sottoprogrammi

Problema: trasferimento controllo dal chiamante al chiamato

copy-rule: (visione banale di sottoprogramma) l'effetto di una chiamata al sottoprogramma P è lo stesso che si otterrebbe sostituendo ogni invocazione di P con il suo corpo

In realtà non è mai implementata come macroespansione, bensì come `branch&link` HW, ovvero come salto più memorizzazione in una cella del return-point

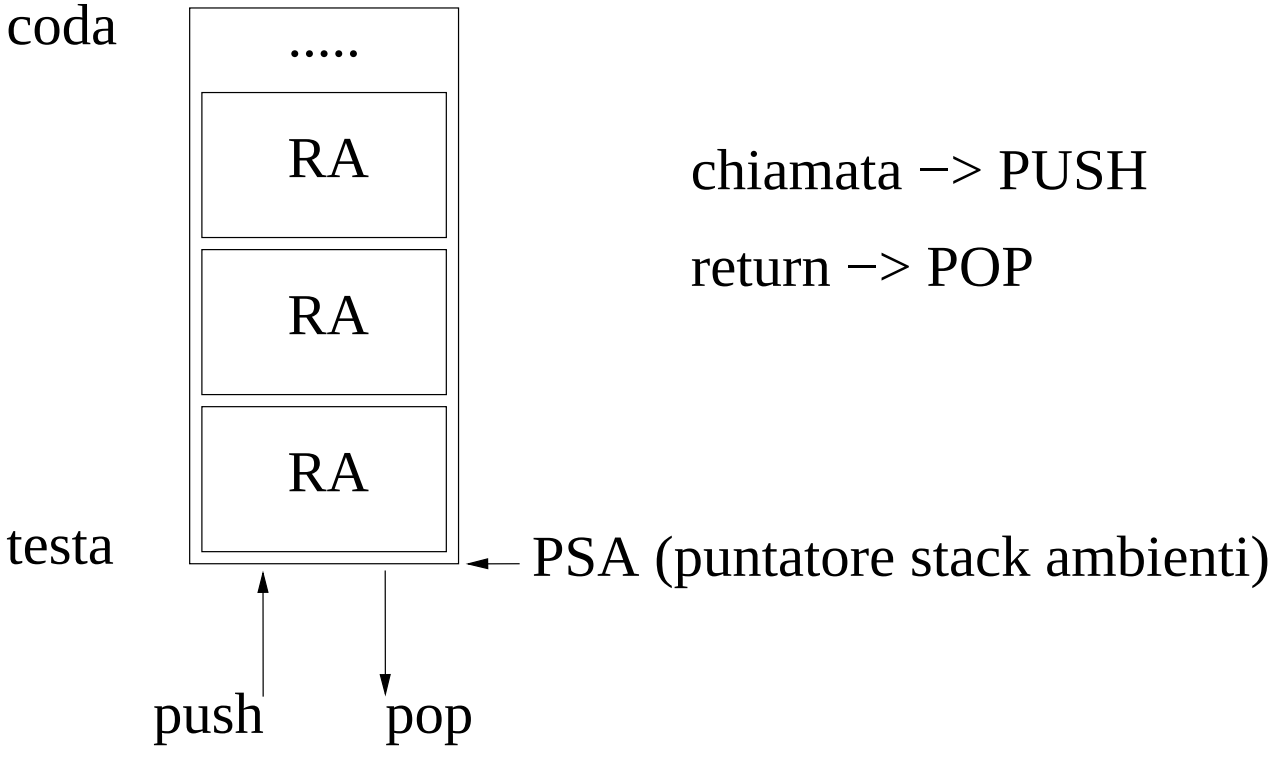
limite: funziona purchè non ci sia ricorsione (infinite sostituzioni)

Sottoprogrammi Ricorsivi

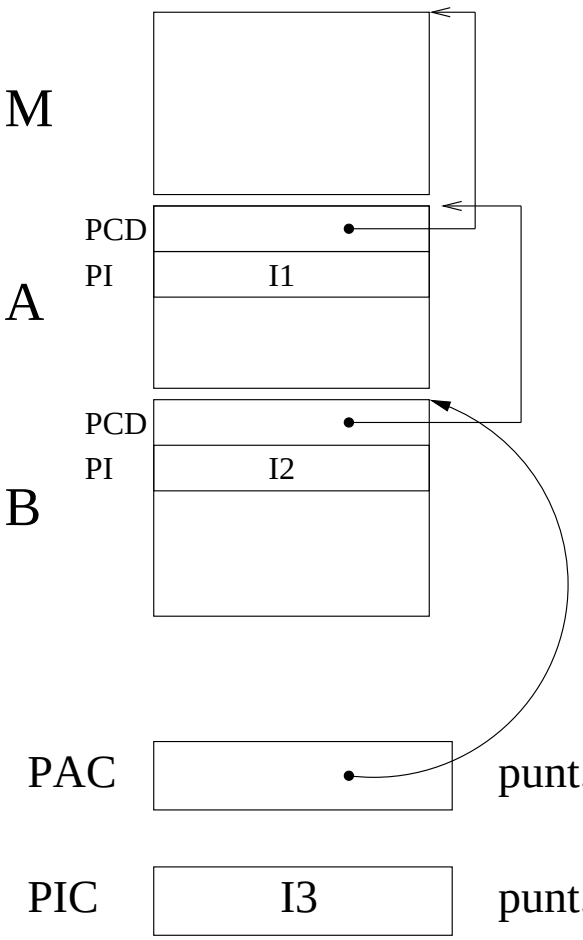
Per poter avere la **ricorsione** $\rightarrow$ `attivazione`:

- parte fissa (segment code, SC)
- parte dinamica (record di attivazione, RA)

Record di attivazioni organizzati a stack

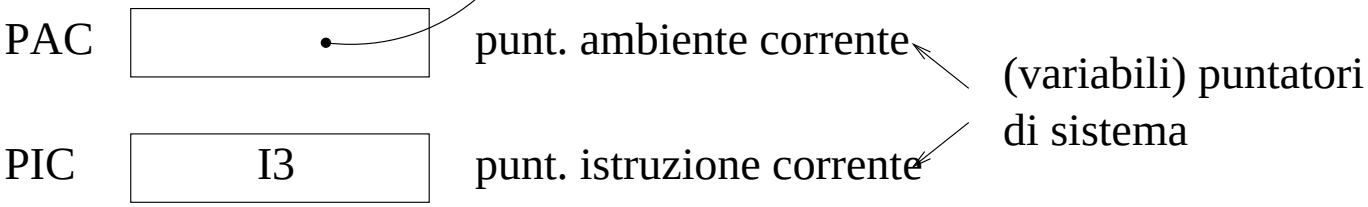


```
M          A          B
...        ...        I3
call A     ...        ...
I1         ...        ...
...        call B     ...
...        I2         ...
...        ...        ...
```



RETURN POINT = (PCD,PI)

PCD: punt. di catena dinamica
PI: puntatore istruzione



stato della macchina all'inizio dell'esecuzione di B

Controllo Dati

Problema:
come fornire i dati a operazioni & sottoprogrammi

ovvero

qual è l'ambiente di riferimento per un nome?

Due sono i problemi principali:

1. un nome può denotare oggetti diversi
   (es. variabili locali)

2. un oggetto può essere denotato da più nomi
   (es. passaggio parametri) $\rightarrow$ alias

Quali sono le operazioni che possono essere fatte sull'ambiente da parte dei linguaggi di programmazione?

1. creazione di un'associazione

   `<nome,oggetto>`

   es. dichiarazioni, parametri formali

2. uso dell'ambiente
   es. utilizzo identificatori

3. disattivazione associazione
   es. P chiama $Q \Rightarrow$ certe associazioni di P vengono disattivate

4. riattivazione associazione
   es. Q restituisce controllo a P

5. distruzione associazioni

Il concetto di **blocco**

```
begin
  D      ⇒ dichiarazioni locali
  C      ⇒ comandi
end
```

Esempio:

```
x = 5;
{ int x;
  x = 7;
  printf(``%d '', x);
}
printf(``%d '', x);
```

output: 7 5

Un blocco è come una procedura senza parametri per la quale sia stata utilizzata la copy-rule

Scoping

Problema:
Quando

`<nome,oggetto>`

è usabile (visibile)?

ovvero

quando esiste ed è attiva?

ovvero

qual è l'ambiente di riferimento?

Ambiente:

- locale (AL)
- globale (AG)
- non locale (ANL)

AL: insieme associazioni create/attivate caratterizzanti un blocco/sottoprogramma

AG: insieme associazioni condivise da tutti i blocchi/sottoprogrammi

ANL: insieme associazioni usabili ma non locali (a volte si precisa che ANL e AG sono disgiunti; il libro di testo dice che AG è sottoinsieme di ANL)

Le regole di scoping riguardano essenzialmente ANL

Scoping:

dinamico: le regole di visibilità sono legate all'esecuzione (Lisp)

statico: le regole di visibilità sono legate alla struttura (sintassi) del programma: è la tecnica più usata nei linguaggi moderni (C, C++, Java, Pascal, ML...)

Ambiente Locale

Notazione:

$P \to Q$: procedura P chiama Q

$P \hookrightarrow Q$: procedura P termina restituendo il controllo al chiamante Q

Ad esempio in

$$P \to Q \to R \hookrightarrow Q \hookrightarrow P$$

cosa accade all'ambiente locale di Q?

La parte facile:

$Q \to R$: quando il controllo passa a R, AL di Q viene disattivato

$R \hookrightarrow Q$: quando il controllo viene nuovamente passato a Q, il suo AL viene riattivato

La parte delicata:

$$P \to Q \qquad \text{e} \qquad Q \hookrightarrow P$$

Due soluzioni possibili

1. dinamica (ALD)
2. statica (ALS)

ALD: Ambiente Locale Dinamico

$P \to Q$: viene **creato** AL di Q

$Q \hookrightarrow P$: viene **distrutto** AL di Q

ALS: Ambiente Locale Statico

$P \to Q$: viene **riattivato** AL di Q

$Q \hookrightarrow P$: viene **disattivato** AL di Q

Esempio: l'opzione "static" in C crea un ambiente locale statico

```
void f()
{ static int x = 0;
  x++;
  printf(``%d '', x);
  f();
}
...
f();
```

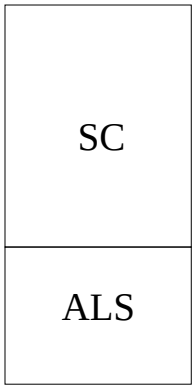
output: 1 2 3 4 5...

Cosa accade senza "static"?

Implementazione Ambiente Locale

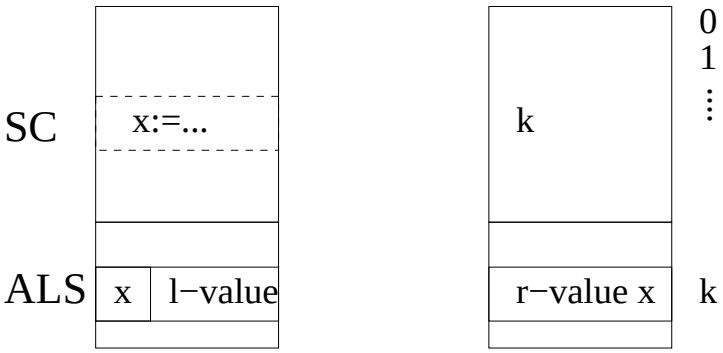
1. Ambiente Locale Statico

Tabella ambiente locale statico associata permanentemente al segmento di codice



Nelle implementazioni concrete:

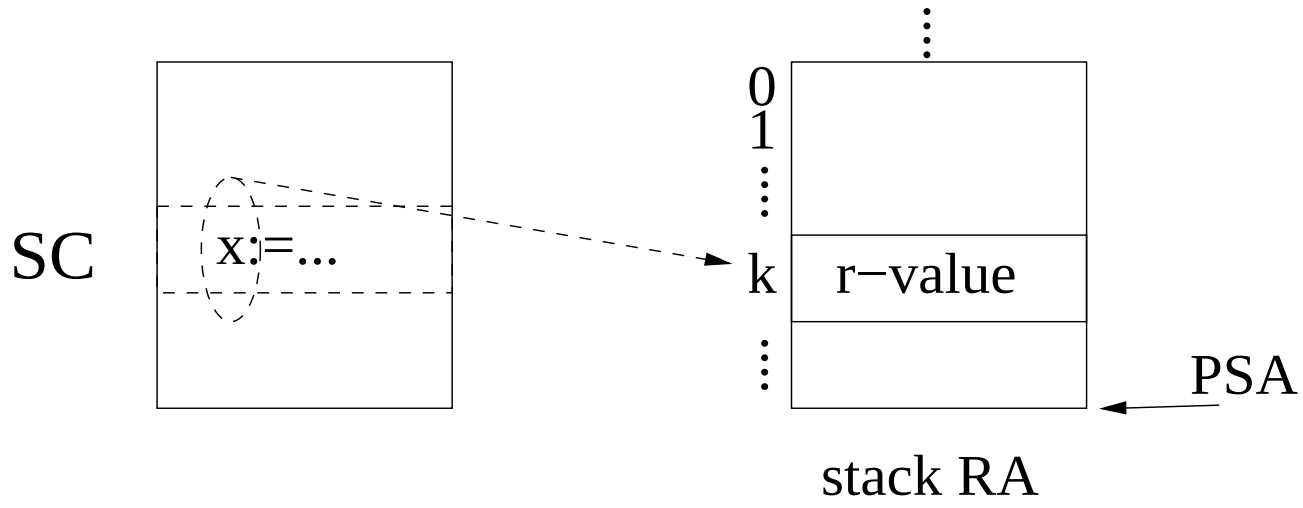
- i nomi sono offset
- ALS è una sequenza di r-value



2. Ambiente Locale Dinamico

L'ambiente locale è parte del record di attivazione

Anche in questo caso un nome locale di un sottoprogramma è compilato come offset, però questa volta all'interno del RA



Ambiente Globale (AG)

Tabella comune a tutti i sottoprogrammi (compreso il main)

Implementazioni concrete: trattato come un record (i nomi sono compilati come campi di un record). È sufficiente che SC contenga l'indirizzo di base di AG.

Riferimenti Non Locali

```
proc  Q()
  begin
    ...
    x
    ...
  end
```

Se x non è locale, qual è l'associazione da usare per x?

Risposta: **regole di scoping**

(Useremo una sintassi ispirata al Pascal per spiegare i concetti di base)

Scoping Dinamico (cenni)
(ambiente non locale dinamico)

Se x è un nome non locale viene ricercata l'ultima associazione per x

L'implementazione di questa ricerca non locale è semplice se:

ambiente locale dinamico + stack di RA

infatti è sufficiente scandire lo stack dei RA all'indietro partendo dalla testa

Purtroppo la ricerca dell'associazione non è fattibile in tempo costante

Esistono implementazioni efficienti che ricercano l'associazione non locale in tempo costante.

Esempio: `Shallow-Binding`

La seguente tabella

xyz	0/1	valore

nome flag oggetto denotato

ci dice se l'associazione è attiva

Se $P \rightarrow$ Q allora vengono disattivate le associazioni di P che sono in conflitto con quelle di Q

Occorre tener traccia (in uno stack) delle associazioni in conflitto per poi ripristinarle quando $Q \hookrightarrow P$

Perché lo scoping dinamico è stato di fatto abbandonato?

$\rightarrow$ impossibile type checking statico!

```
begin
  real x, y;
  proc P;
    begin
      x:=x+y
    end;
  begin
    real x, y;
    call P        → durante esecuzione
  end;              di P x e y reali
  begin
    integer x, y;
    call P        → durante esecuzione
  end               di P x e y interi
end
```

Scoping Statico
(ambiente non locale statico)

Ad ogni identificatore è associata staticamente una dichiarazione. Tale associazione è costante durante il tempo di esecuzione.

Per un'analisi più rigorosa, associamo ad ogni programma un **albero**, detto **di scoping**:

(diamo un nome diverso ad ogni blocco (i sottoprogrammi hanno già un nome))

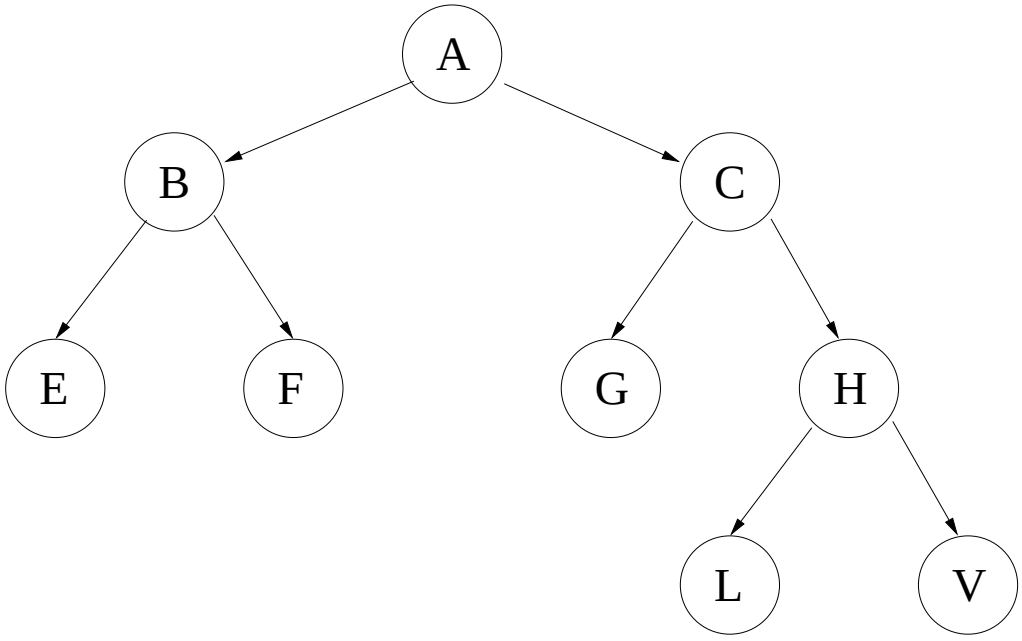
nodi albero $\rightarrow$ nomi di blocchi e sottoprogrammi

1. la radice è il nome del blocco più esterno

2. Q è figlio di P se
   - Q è sottoblocco diretto di P
   - Q è un sottoprogramma dichiarato in P

```
A: begin
     proc B;
       begin
       E: begin
          ...
          end
       F: begin
          ...
          end
       end {B}
     C: begin
        G: begin
           ...
           end
        proc H;
           begin
             L: begin
                ...
                end
             V: begin
                ...
                end
           end {H}
        end {C}
   end {A}
```

Albero di scoping del programma pag. precedente



Regola di Scoping Statico

Sia x un'occorrenza di un riferimento non locale in un sottoprogramma/blocco P

1. l'ambiente non locale che fornisce l'associazione corretta per x è quello dell'antenato Q più vicino a P nel quale x è dichiarato

2. se non esiste alcun antenato Q che dichiara x viene generato un errore (controllo effettuato in fase di compilazione)

Con la sintassi adottata l'ambiente globale è l'ambiente del sottoprogramma/blocco più esterno

Se il linguaggio definisce un ambiente globale esterno a sottoprogrammi/blocchi la regola di scoping è così riscritta

1. `<come prima>`

2. se non esiste alcun Q antenato di P che dichiara x allora x è cercato nell'ambiente globale. Se non viene trovato viene generato un errore (in compilazione)

Implementazione Scoping Statico (con ambiente locale dinamico)

`Osservazione`: lo stack di RA fornisce un ordinamento temporale tra gli ambienti locali (inutile per lo scoping statico) ma non fornisce alcuna indicazione sulla struttura sintattica del programma

L'informazione "statica" sulla struttura sintattica (albero di scoping) è implementata tramite i puntatori di catena statica (PCS)

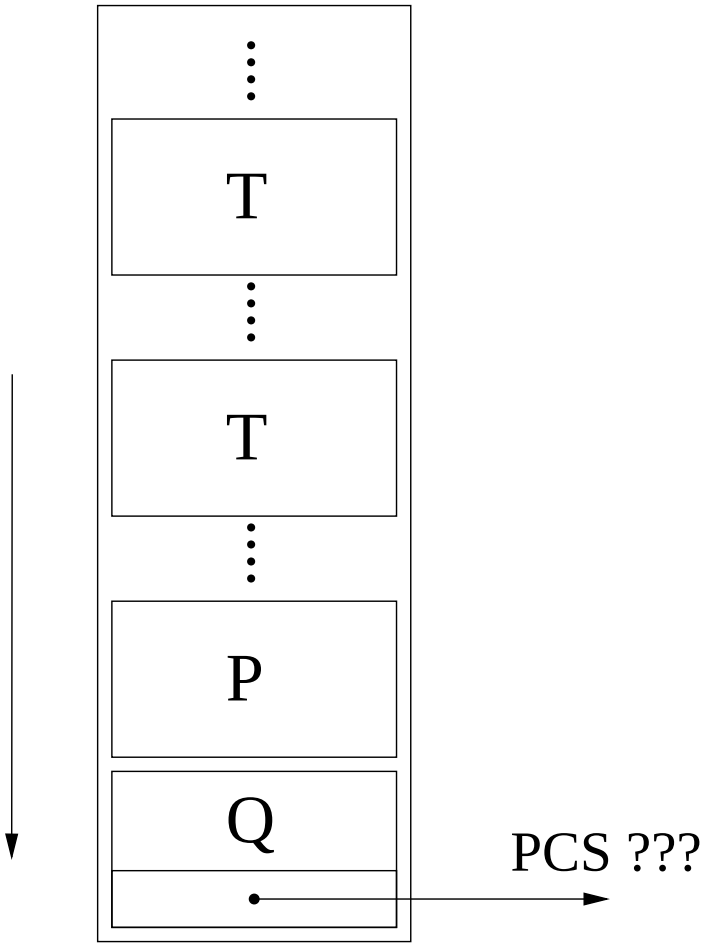
Il PCS di un RA di un sottoprogramma/blocco punta al RA del sottoprogramma/blocco Q antenato di P nell'albero di scoping, individuato mediante la regola di scoping statico

Pertanto viene aggiunto ad ogni RA un puntatore di catena statica. Vediamo come

Supponiamo che

$$P \to Q$$

$\to$ viene fatto un push del RA di Q nello stack dei RA



Supponiamo che Q sia figlio di T ma che nello stack ci siano più occorrenze di T

Siano α, β due nodi dell'albero di scoping

Se

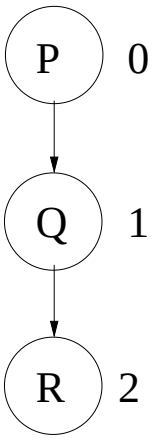
$$\alpha \rightarrow \beta$$

definiamo

$$\#(\alpha, \beta) = \texttt{profondita}(\alpha) - \texttt{profondita}(\texttt{padre}(\beta))$$

N.B.
se $\alpha \rightarrow \beta \Rightarrow$ il padre di β deve essere antenato di α

Esempio:



$$Q \rightarrow R \quad \Rightarrow \#(Q, R) = 0$$

$$R \rightarrow Q \quad \Rightarrow \#(R, Q) = 2$$

`Algoritmo per la determinazione del PCS`

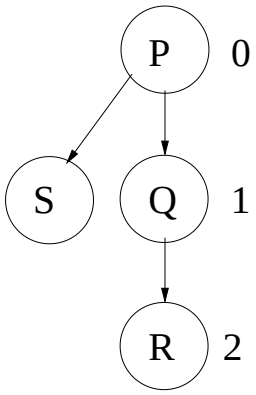
Se

$$P \rightarrow Q$$

allora

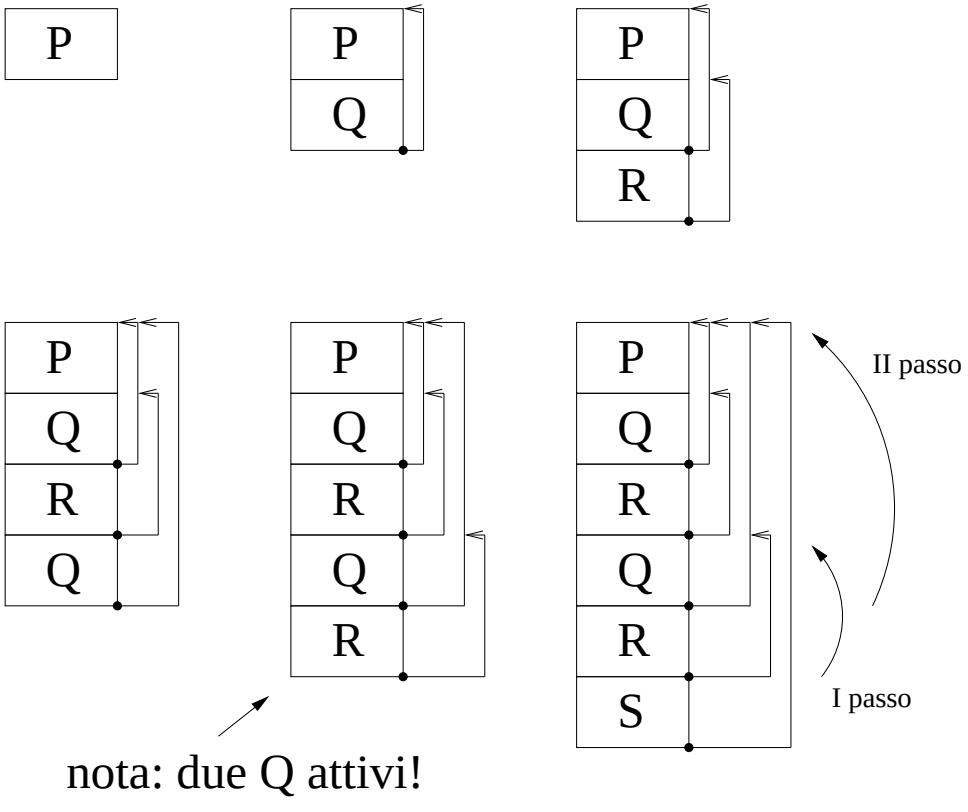
1. viene messo sullo stack il RA di Q (RA_Q)
2. viene calcolato $\#(P, Q)$
3. viene calcolato l'indirizzo a del RA del sottoprogramma/blocco T che dichiara Q effettuando $\#(P, Q)$ passi di catena statica partendo dal PCS del RA dell'invocante P
4. PCS di RA_Q è il valore di a

Esempio

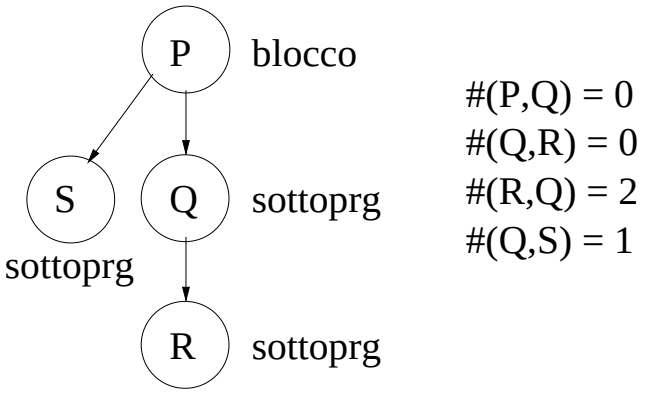


#(P,Q) = 0
#(Q,R) = 0
#(R,Q) = 2
#(R,S) = 2

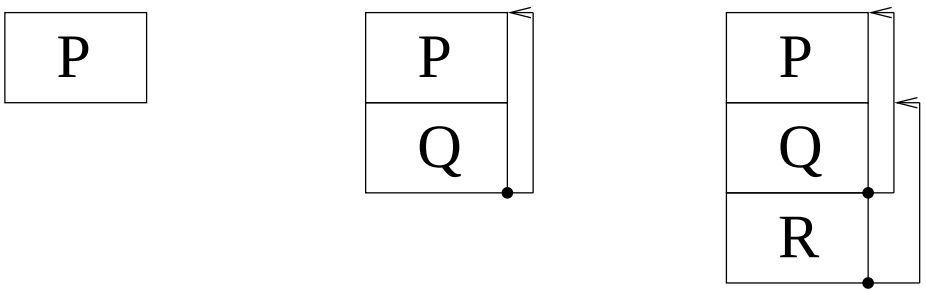
$$P \rightarrow Q \rightarrow R \rightarrow Q \rightarrow R \rightarrow S$$

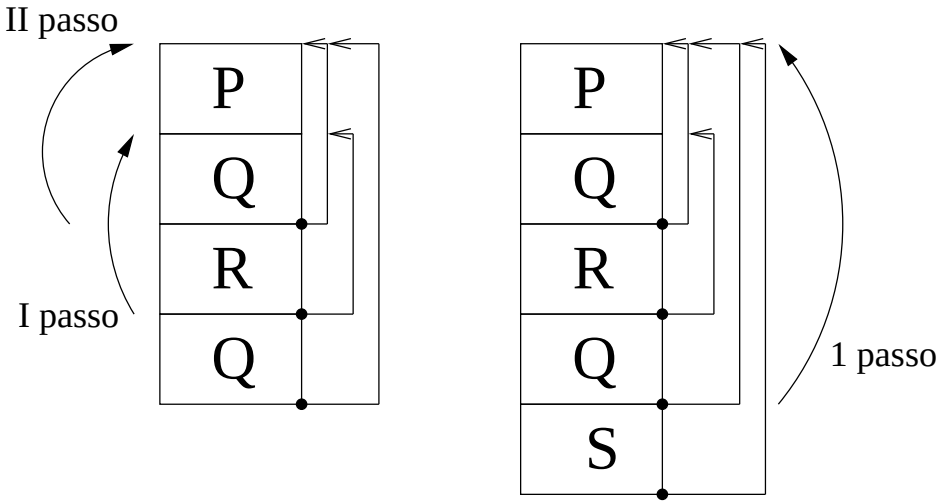


Esempio



$$P \rightarrow Q \rightarrow R \rightarrow Q \rightarrow S$$





Più generalmente, usando l'albero di scoping:

Se il sottoprogramma/blocco P usa il nome N

```
#(P,N) = profondita(P) -
         profondita(sttprg/blc che dichiara N)
```

Ogni riferimento non locale N nel sottoprogramma/blocco P è compilato come

$$< x, y >$$

dove

$x = \#(P, N)$

y = posizione (offset) di N nel modello (template) del RA del sottoprg/blocco che dichiara N

Se $x = 0$ N è compilato come y (N è locale!)

Passaggio Parametri

Assumiamo:
ambiente locale dinamico + scoping statico

Notazione:
`(a) proc P(x)` $\rightarrow$ `x parametro formale`
`(b) call P(E)` $\rightarrow$ `E par. effettivo o argomento`

Scriviamo

$$\texttt{call P(x} \Leftarrow_{\alpha} \texttt{E)}$$

per indicare che P è dichiarata come in `(a)` e che è invocata come in `(b)` associando l'argomento E al parametro formale x

α indica il tipo di passaggio di parametri adottato

I parametri formali sono a tutti gli effetti variabili locali

```
proc P(x)
  begin
    int y
    ...
  end
```

`x, y` : variabili locali

I parametri formali sono allocati nel record di attivazione

Con

$$[\texttt{call P(x} \Leftarrow_\alpha \texttt{E)}]_\sigma$$

indichiamo i passi dell'interprete per eseguire il passaggio del parametro rispetto allo stato σ della macchina (σ = stack di RA, memoria...)

Passaggio per Costante

$$[\texttt{call P(x} \Leftarrow_{\texttt{cost}} \texttt{Y)}]_\sigma =$$

r-value della variabile `Y` diventa r-value di `x` nel RA di `P`

`P` non può modificare `x`

Nei linguaggi moderni è una variante del passaggio per valore

Passaggio per Valore

$$[\texttt{call P(x} \Leftarrow_{\texttt{val}} \texttt{E)}]_\sigma =$$

1. viene valutata l'espressione `E` in σ: $\texttt{v} = [\texttt{E}]_\sigma$
2. nel RA di `P` viene assegnato `v` alla variabile `x` (`x` è locale!)

Passaggio per Value-Result

$$[\texttt{call P(x} \Leftarrow_{\texttt{val-res}} \texttt{Y)}]_\sigma =$$

1. viene calcolato r-value(Y) rispetto a σ: $\texttt{v} = [\texttt{Y}]_\sigma$
2. nel RA di P viene assegnato v alla var. locale x
3. al momento di ritornare al chiamante viene valutato r-value(x) = w nello stato σ_1 corrente. Il valore w viene assegnato alla variabile Y nel chiamante

Passaggio per Result

$$[\texttt{call P(x} \Leftarrow_{\texttt{res}} \texttt{Y)}]_\sigma =$$

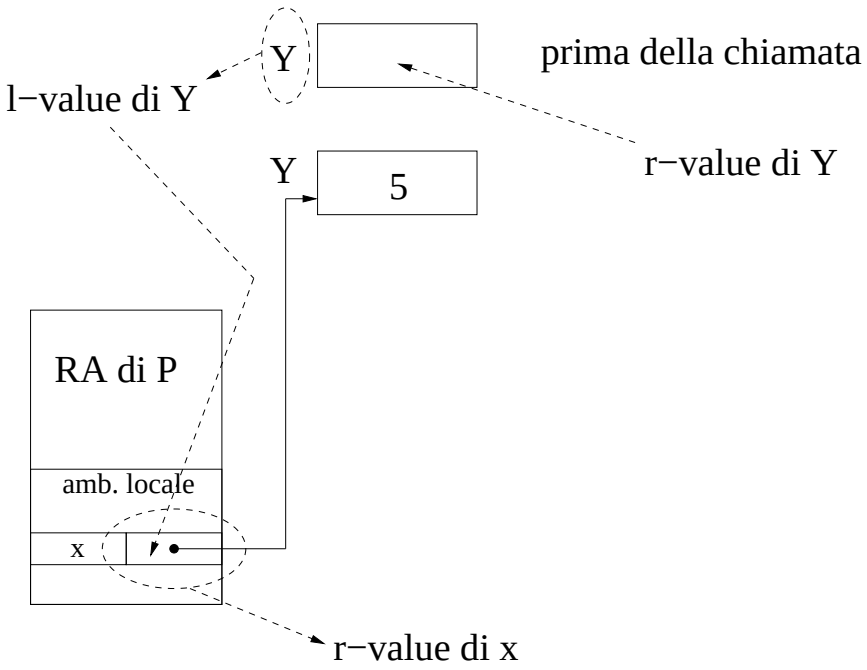
Viene effettuato solo il passo (3) del metodo `value-result`

Passaggio per Riferimento (Indirizzo)

$$[\texttt{call P(x} \Leftarrow_{\texttt{ref}} \texttt{Y)}]_\sigma =$$

1. viene calcolato in σ: `addr` = l-value(`Y`)
2. viene assegnato a `x` `addr` come l-value

Il passo (2) è poco realistico: in realtà `x` è un puntatore e `addr` è assegnato all'r-value di `x` che accede all'r-value di `Y` tramite un accesso indiretto nascosto al programmatore



Un accesso a `x` (`es. x:=5`) significa prendere la cella con indirizzo `a = r-value(x)` e assegnare 5 a quella cella

Passaggio per Nome

$$[\texttt{call P(x} \Leftarrow_{\texttt{nome}} \texttt{E)}]_\sigma =$$

`x` non è implementato come una vera var. locale

Viene creata una coppia

$$< (\texttt{E}), \rho >$$

detta `thunk` dove

ρ è l'ambiente di rif. per la valutazione di `E` in σ, e

`(E)` è il codice compilato di `E`

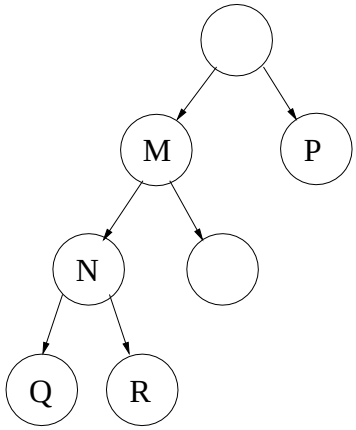
Alla variabile `x` è associato il thunk

Quando in P deve essere valutato il parametro formale x, in realtà viene valutata (E) nell'ambiente (RA) ρ: l'implementazione tipica è di passare un sottoprogramma senza parametri con la stessa tecnica discussa nei lucidi seguenti

Passaggio di Procedure

$$[\texttt{call P(x} \Leftarrow_{\texttt{proc}} \texttt{R)}]_\sigma = ?$$

La situazione è complicata




```
proc R
  begin ··· z ··· end

proc Q
  begin
    ··· call P(R) ···
  end

proc P(x:  proc)
  bein
    ··· call x ···
  end
```

Quale ambiente non locale (statico) per R quando P la invoca tramite la call?

`call P(R)`

in esecuzione viene calcolata la chiusura (**`thunk`**)

$$\texttt{T} = < \texttt{r}, \texttt{a} >$$

dove

- `r`: puntatore codice compilato (in questo caso di `R`)
- `a`: indirizzo RA (ambiente) pertinente per `R` (ottenuto facendo i passi di catena statica necessari partendo dal RA del sottoprogramma invocante, ovvero che include la call, in questo caso `Q`)

`T` è associato alla variabile locale `x`

Che accade quando P esegue

`call x`

?

- viene caricato RA di R usando `r`
- il puntatore di catena statica del RA di R assume il valore `a`

Più generalmente

$$[\texttt{call P(x} \Leftarrow_{\texttt{proc}} \texttt{R)}]_\sigma =$$

$$= [\texttt{call P(x} \Leftarrow (\texttt{k}, \texttt{m}))]_\sigma$$

1. usando il RA del chiamante (quello che include la call) viene calcolata la chiusura

   $$< \texttt{r}, \texttt{a} >$$

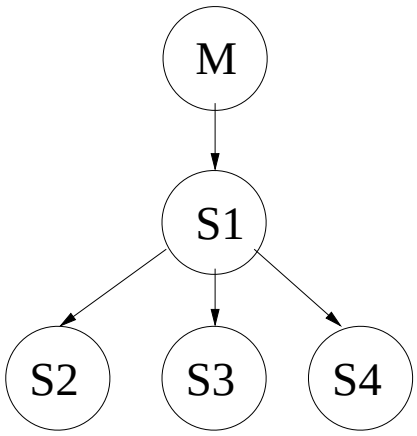
   dove `r` e `a` sono calcolati facendo `k` passi di catena statica e poi prendendo l'offset `m` nel RA raggiunto

2. ad `x` è associata la chiusura $< \texttt{r}, \texttt{a} >$

Dal punto di vista utente, le regole di scoping per le variabili non locali di R sono quelle che sarebbero state applicate se R fosse stata chiamata al momento della `call P(R)`

```
program esempio 1;
proc S1;
  var x:  integer;
     proc S2;
       begin
         writeln(x)
       end; {S2}
     proc S4 (proc SX);
       var x:  integer;
       begin
         x:=4;
         SX
       end; {S4}
     proc S3;
       var x:  integer;
       begin
         x:=3;
         S4(S2)
       end; {S3}
  begin
     x:=1;
     S3
  end; {S1}
begin
  S1
end.  {main}
```

cont. esempio 1

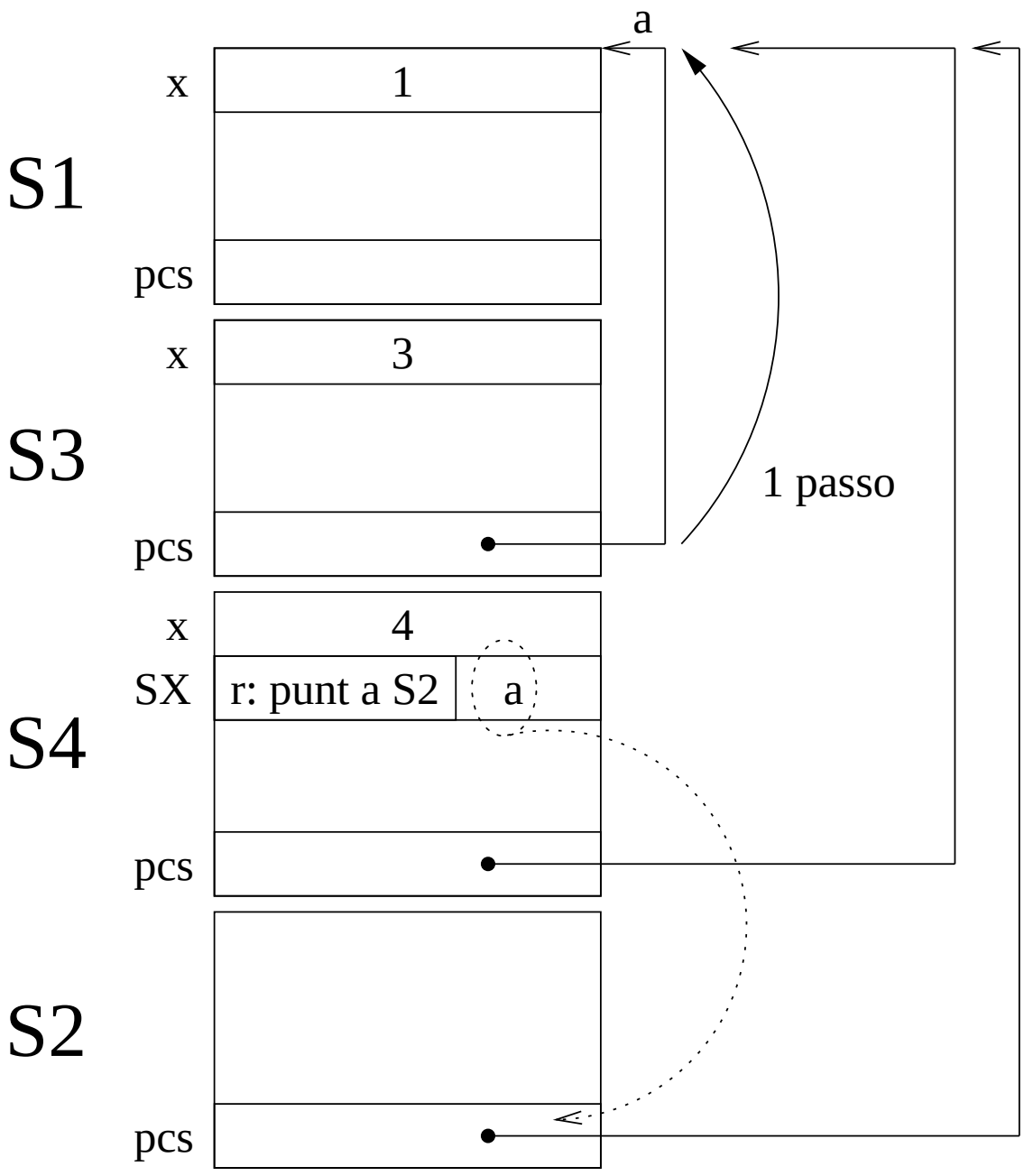


$$M \rightarrow S1 \rightarrow S3 \rightarrow S4 \rightarrow S2$$

Nota: durante l'esecuzione di S3, in occasione della chiamata S4(S2), viene calcolato $\#(S3, S2) = 1$ e quindi al parametro SX di S4 viene associata la chiusura

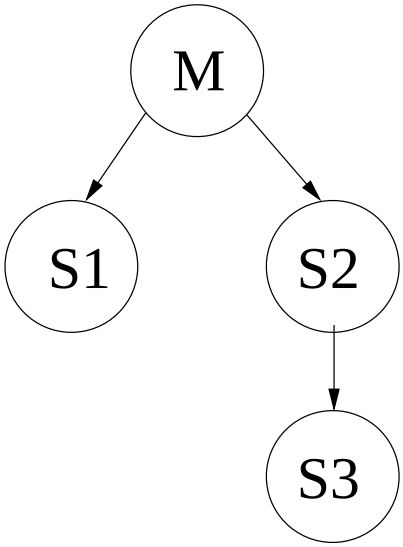
<puntatore a S2,
puntatore a RA ottenuto percorrendo 1 passo
di catena statica a partire dal RA di S3
correntemente attivo>

cont. esempio 1



```
program esempio 2;
  proc S1;
    begin
    ...
    end; {S1}
  proc S2 (proc SX);
    var x:  real;
    proc S3;
      begin
        x:=0.0
      end; {S3}
    begin
      SX;
      S2(S3)
    end; {S2}
begin
  S2(S1)
end.  {main}
```

cont. esempio 2

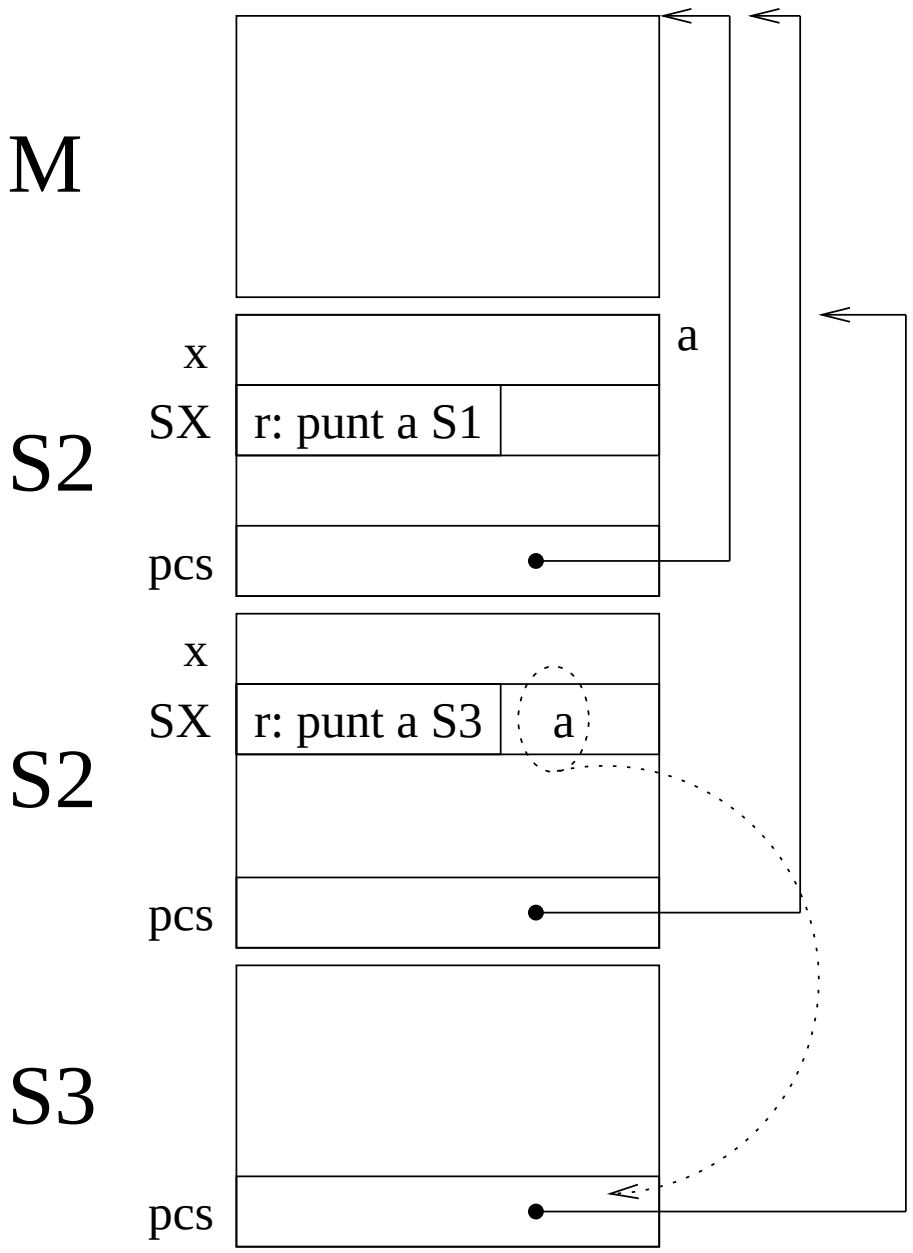


$$M \rightarrow S2 \rightarrow S1 \hookrightarrow S2 \rightarrow S2 \rightarrow S3$$

Nota: durante l'esecuzione di S2, in occasione della chiamata S2(S3), viene calcolato $\#(S2, S3) = 0$ e quindi al parametro SX di S2 viene associata la chiusura

```
<puntatore a S3,
puntatore a RA ottenuto percorrendo 0 passi
di catena statica a partire dal RA di S2
correntemente attivo>
```

cont. esempio 2



Pertanto l'x riferito da $S3$ non è quello dell'attivazione di $S2$ più recente, ma di quella precedente!