La Gestione della Memoria

Dispensa del corso di Linguaggi e Traduttori

A.A. 2005-2006

Giovanni Semeraro

Da qui in poi, terminata l'analisi, cominceremo la parte di **sintesi** svolta dal compilatore.

Per comprendere questa parte e' necessario sapere come l'informazione in un programma in esecuzione e' immagazzinata ed utilizzata.

Ci occuperemo quindi delle strategie di allocazione della memoria utilizzate in un'ampia classe di linguaggi di programmazione.

L'allocazione della memoria puo' essere cosi' suddivisa:

- statica;
- a stack;
- a heap.

Allocazione statica

In questo caso dobbiamo decidere a tempo di compilazione esattamente dove l'oggetto risiederà in memoria.

Per prendere questa decisione è necessario che:
1) La dimensione di ogni oggetto sia nota;
2) Solo una occorrenza di un oggetto sia disponibile durante tutta l'esecuzione del programma.

Non sono quindi permesse stringhe o array di lunghezza variabile, poiche' la loro dimensione non e' nota staticamente.

Anche le procedure innestate non sono permesse in quanto a tempo di compilazione non e' noto quante procedure e le rispettive variabili locali saranno attive (non ha senso allocarle preventivamente tutte).

Inoltre, non sono permesse chiamate ricorsive, in quanto staticamente non si sa il numero delle attivazioni.

FORTRAN e' un linguaggio che rispetta i precedenti criteri.

Una allocazione statica della memoria è molto semplice da realizzare. L'indirizzo degli oggetti si inserisce direttamente nella tabella dei simboli. Poichè inoltre e' nota la dimensione di ogni oggetto si puo' seguire il seguente procedimento.

Alla prima variabile è assegnato un certo indirizzo A all'inizio di una area di memoria attribuita ai dati. Alla seconda A + n1, se n1 sono le unita' di memoria richieste dalla prima variabile, alla terza A + nl +n2 ecc.

Esempio di una TS per il FORTRAN (immaginiamo che i reali occupino 8 bytes e gli interi 4):

```
REAL MAXPRN,RATE
INTEGER IND1, IND2
REAL PRIN(100), YRINT(5,100), TOTINT
```

Name	Type	Dimension	...	Address
MAXPRN	R	0		264
RATE	R	0		272
IND1	I	0		280
IND2	I	0		284
PRIN	R	1		288
YRINT	R	2		1088
TOTINT	R	0		5088

L'indirizzo puo' essere sia **assoluto** che **relativo.**
Normalmente è relativo, cioè riferito ad una base o indirizzo iniziale dell'area dei dati. In questo modo il codice assoluto e' generato dal loader ed e' **rilocabile** perche' il programma può risiedere in aree differenti ad ogni esecuzione.

Allocazione (dinamica) a stack

In questo caso cadono le limitazioni 1) e 2) dette precedentemente.

E' tipica dei linguaggi a blocchi quali Algol, Pascal, C ecc.

La strategia di allocazione può essere modellata mediante uno stack, chiamato *run-time* stack.

Ogni blocco puo' essere visto come proprietario di un'area dati locale che viene inserita/eliminata dal run-time stack quando il blocco inizia/termina.

Procedure e Funzioni (richiami)

Parametri:

I parametri costituiscono, in genere; il mezzo di comunicazione tra unità chiamante ed unità chiamata.

Parametri attuali (specificati nella chiamata) e formali (specificati nella definizione) devono corrispondersi in **numero, posizione** e **tipo.**

Legame dei parametri, all'atto della chiamata.

Variabili locali:

Nella parte dichiarazione di un sottoprogramma (procedura o funzione) possono essere dichiarati costanti, tipi, variabili (dette locali) ed altri sotto programmi (identificatori **locali** al blocco del sottoprogramma).

```
procedure saltabianchi;
var Car:char;
begin
      repeat
            read(Car)
      until Car<>' '
end;
```

Alla variabile Car si può far riferimento solo nel corpo della procedura saltabianchi. II tempo di vita di Car è quello dell'attivazione della procedura saltabianchi.

Ad ogni attivazione di saltabianchi viene creata una associazione tra il nome Car ed uno spazio di memoria:

Car ----> <locazione>

Alla fine dell'attivazione tale associazione viene distrutta. Se la procedura viene attivata di nuovo, viene creata una nuova associazione.

Non c'è correlazione tra i valori che Car assume durante le varie attivazioni della procedura saltabianchi.

Variabili non locali o globali:

Nell'ambito del blocco di un sottoprogramma si può però far riferimento anche ad identificatori dichiarati esternamente **(non locali):**

```
program main (input,output);
var C: char;                                    {*}
      procedure saltabianchi;
            begin
                  repeat
                        read(C)
                  until C<>' '
            end;
begin   ...; saltabianchi;... end.
```

C è una **variabile non locale** della procedura saltabianchi **(globale,** dichiarata nel programma principale).

Tecniche di legame dei parametri:

Si intende l'associazione fra parametri attuali e parametri formali che avviene al momento dell'attivazione di una unità di programma.

Meccanismi piu' comuni:

- **Legame per valore** (Pascal);
- **Legame per indirizzo** (Pascal/Fortran).
- **Legame per valore-risultato** (non presente in Pascal)

Esempio:

Procedura P (con parametro formale pf), chiamata con parametro attuale pa:

```
P(pa);
```

Legame per valore:

Al momento della chiamata, viene valutato pa (può essere un'espressione), creata una locazione per il parametro formale pf e memorizzato in tale locazione il valore di pa:

pa ----> valore di pa
pf ----> valore di pa

Il parametro formale pf si comporta come una **variabile locale** alla procedura P. Viene creata al momento dell'attivazione di P ed inizializzata al **valore di pa.**

Alla fine dell'esecuzione del sottoprogramma P:

pa ----> valore di pa
pf ----> nuovo valore.

II valore delle variabili che compaiono nell'espressione pa nel programma chiamante rimane **inalterato.**

Parametri passati per valore servono per dare **valori in ingresso** al sottoprogramma.

II nuovo valore di pf **non** viene restituito al sottoprogramma chiamante.

Passaggio dei parametri per valore: esempio

Procedura che scambia due variabili X, Y (di tipo integer) se X>Y.

```
program pervalore (input,output);

var X,Y: integer;

procedure scambia1 (A, B: integer);
var T:integer;
begin
      if A>B
      then     begin
                 T:=A;
                 A:=B;
                 B:=T
                 end
end;

begin                          {corpo del main program}
      readln(X,Y);
      scambia1(X,Y);
      writeln(X,Y)
end.
```

Dati in ingresso i valori: 33 5
qual è il risultato stampato?

Alla chiamata:

X ---->33 ====> A ---->33
Y ---->5 ====> B ---->5

Alia fine dell'attivazione della procedura scambia1:

X ----> 33
Y ----> 5

Legame per indirizzo (o riferimento):

II parametro formale pa deve essere una **variabile.** Al momento dell'attivazione del sottoprogramma P, viene calcolato l'***indirizzo*** di pa, creata una locazione per il parametro formale pf e memorizzato in tale locazione l'indirizzo di pa:

pa ---->
valore di pa
pf ----> (indirizzamento indiretto)

Il parametro formale pf si comporta come una **variabile locale** alla procedura P. Viene creata al momento dell'attivazione di P ed inizializzata **all'indirizzo di pa.**

Alla fine dell'esecuzione del sottoprogramma P:

pa ---->
nuovo valore di pa
pf ---->

Il valore di pa nel programma chiamante viene alterato.

Parametri passati per indirizzo servono per dare **valori in uscita** dall'unità chiamata all'unità chiamante.

Passaggio dei parametri per indirizzo: esempio

Procedura che scambia due variabili X, Y (di tipo integer) se X>Y.

```
program perindirizzo (input,output);

var X,Y:integer;

procedure scambia2 (var A, B: integer);
var T:integer;
begin
      if A>B then
            begin
                  T:=A;
                  A:=B;
                  B:=T
            end
end;

begin                        {corpo del main program}
      readln(X,Y);
      scambia2(X,Y);
      writeln(X,Y)
end.
```

Alla chiamata:

```
X   ---->                          Y   ---->
              33                                  5
A   ---->                          B   ---->
```

Alla fine dell'attivazione della procedura scambia2:

```
X   ---->5                                 Y   ---->33
```

Legame per valore-risultato:

Il parametro formale pa deve essere una **variabile.** Al momento dell'attivazione del sottoprogramma P, viene calcolato il valore di pa, creata una locazione per il parametro formale pf e memorizzato in tale locazione il ***valore*** di pa (simile al passaggio per valore):

pa ----> valore di pa
pf ----> valore di pa

Durante l'attivazione di P, le modifiche fatte al valore di pf non influenzano quello di pa.

pa ----> valore di pa
pf ----> nuovo valore di pf

Solo alla fine dell'esecuzione del sottoprogramma P, il nuovo valore di pf viene assegnato al parametro attuale pa:

pa ----> nuovo valore
pf ----> nuovo valore

Il valore di pa nel programma chiamante viene alterato (simile al passaggio per indirizzo).

Parametri passati per valore-risultato servono per dare sia **valori in ingresso** che **valori in uscita.**

E' una tecnica di legame non presente in Pascal.

Esempio:

```
program legapar (input,output);
var N:integer;
procedure P ( ?          X: integer);
      begin
            X:=X+1;
            writeln(N);                    {1}
            writeln(X)                     {2}
      end; begin
      N:=3; P(N);
      writeln(N)                           {3}
end.
```

Se X è legato per valore abbiamo le stampe:

{1} 3
{2} 4
{3} 3

Se X è legato per indirizzo:

{1} 4
{2} 4
{3} 4

Se X è legato per valore-risultato:

{1} 3
{2} 4
{3} 4

Effetti collaterali in procedure e funzioni:

Si chiama effetto collaterale **(side effect)** provocato dall'attivazione di un'unità di programma una qualunque modifica delle variabili non locali a tale attivazione.

Sono accettabili nelle procedure, ma da evitare nelle funzioni (compito generalmente lasciato al programmatore). Una funzione dovrebbe solo fornire un valore in uscita.

Rappresentano una interazione potenziale tra unità chiamante ed unità chiamata. Occorre porre attenzione nell'uso di **parametri passati per indirizzo** e **variabili non locali.**

```
program effetticollateralil (output); var
B: integer;
    function f (var A:integer): integer;
    begin
        A:=2*A;
        f:=A
    end;
begin
    B:=l;
    writeln(2*f(B));            {1}
    B:=l;
    writeln(f(B)+f(B))          {2}
end.
```

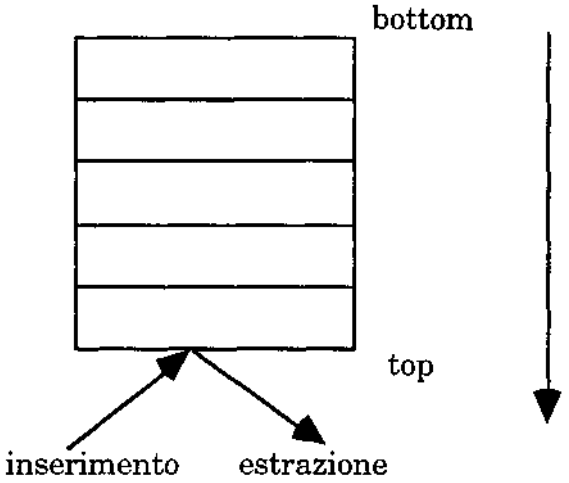
Fornisce valori diversi, pur essendo attivata con lo stesso parametro attuale. L'istruzione {1} stampa 4 mentre l'istruzione {2} stampa 6.

Esempio modello a tempo di esecuzione a stack per il Pascal

Aree di memoria: area dati globale, heap, stack

Anche in Pascal (come in tutti i linguaggi Algol-like nei quali è presente la nozione di blocco) le attivazioni dei sottoprogrammi sono realizzate utilizzando una zona di memoria gestita seguendo una disciplina a pila **(stack).**

Stack, è un multiinsieme in cui l'ultimo elemento inserito è il primo ad essere estratto (LIFO, Last In First Out).



Ad ogni attivazione di un'unità di programma viene creato un **record d'attivazione** ed inserito in cima alla pila.

Record d'attivazione:

Un record d'attivazione rappresenta le informazioni relative ad una specifica attivazione. In particolare:

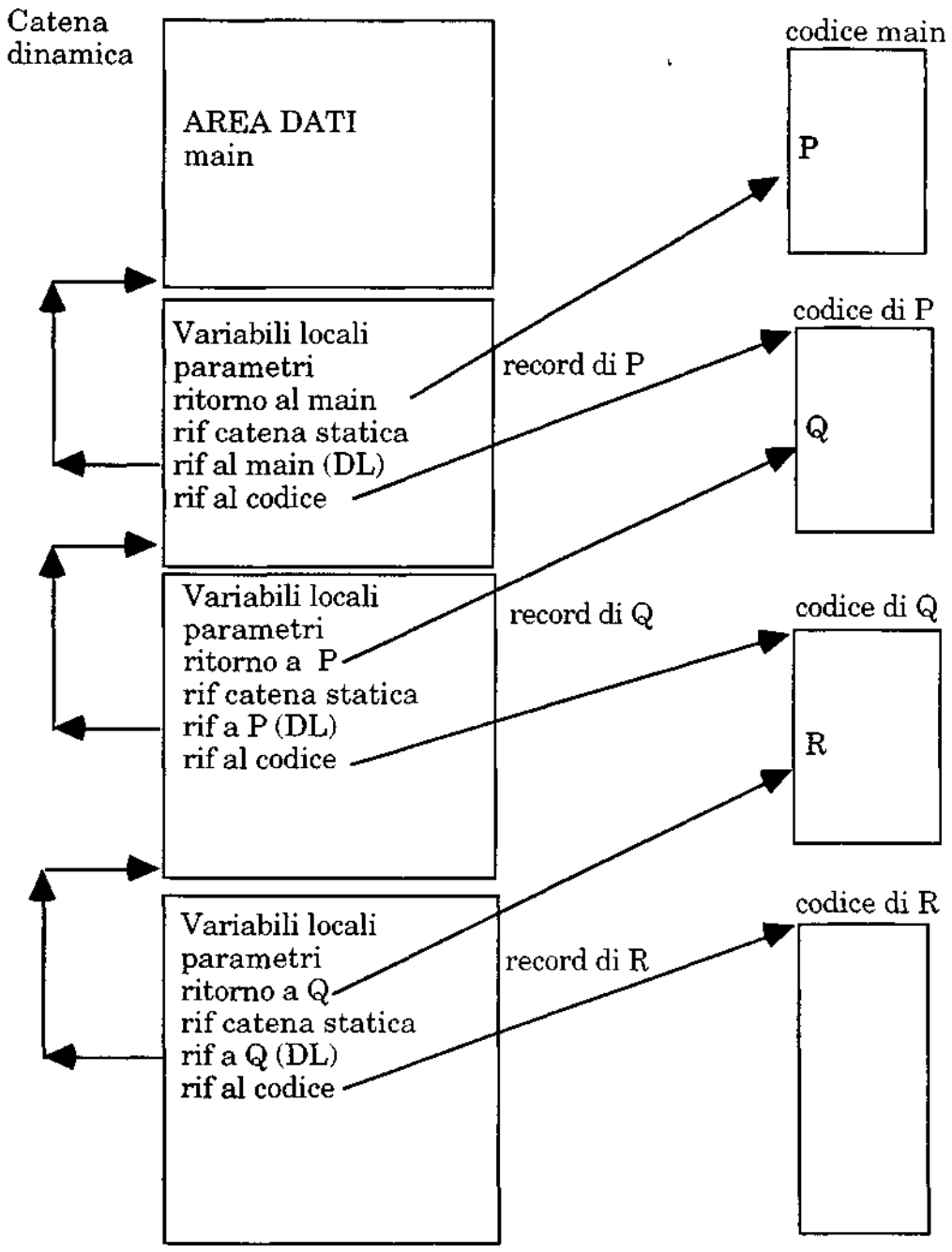
1) nome del sottoprogramma attivato e riferimento al codice;
2) punto di ritorno al chiamante **(return address): è** l'indirizzo dell'istruzione da eseguire al termine della attivazione;
3) riferimento di catena statica **(static link): è** un riferimento all'ambiente visto "staticamente" dal sotto-programma (variabili non locali);
4) parametri formali e loro legame con quelli attuali;
5) variabili locali;
6) riferimento al record di attivazione precedente sulla pila (catena dinamica, **dynamic link): è** un riferimento all'ambiente del chiamante.

La dimensione del record di attivazione di un'unità di programma è nota staticamente. Durante la compilazione, un simbolo di variabile (locale o parametro) viene legato al suo "offset" all' interno del record di attivazione **(variabili semistatiche).**

Al termine dell'esecuzione (return) il record di attivazione viene deallocato dallo stack. Viene quindi rilasciata la memoria allocata sia per le variabili locali che per i parametri formali. Si perde traccia del legame tra parametri formali ed attuali.

L'esecuzione prosegue dall'istruzione memorizzata in 2).

Immaginiamo che P (attivato dal main program) abbia attivato Q che ha attivato R (catena dinamica). Avremo:

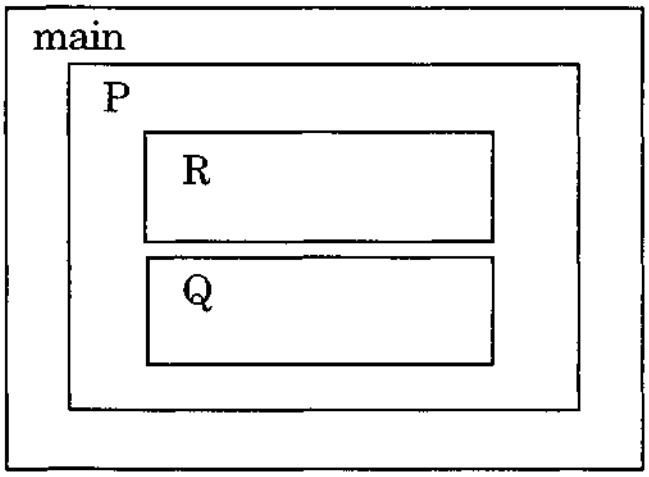


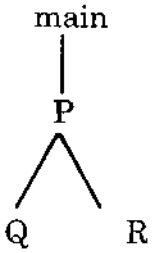
La catena dinamica rappresenta la storia delle attivazioni delle unità di programma.

Attivazioni: main --> P --> Q --> R

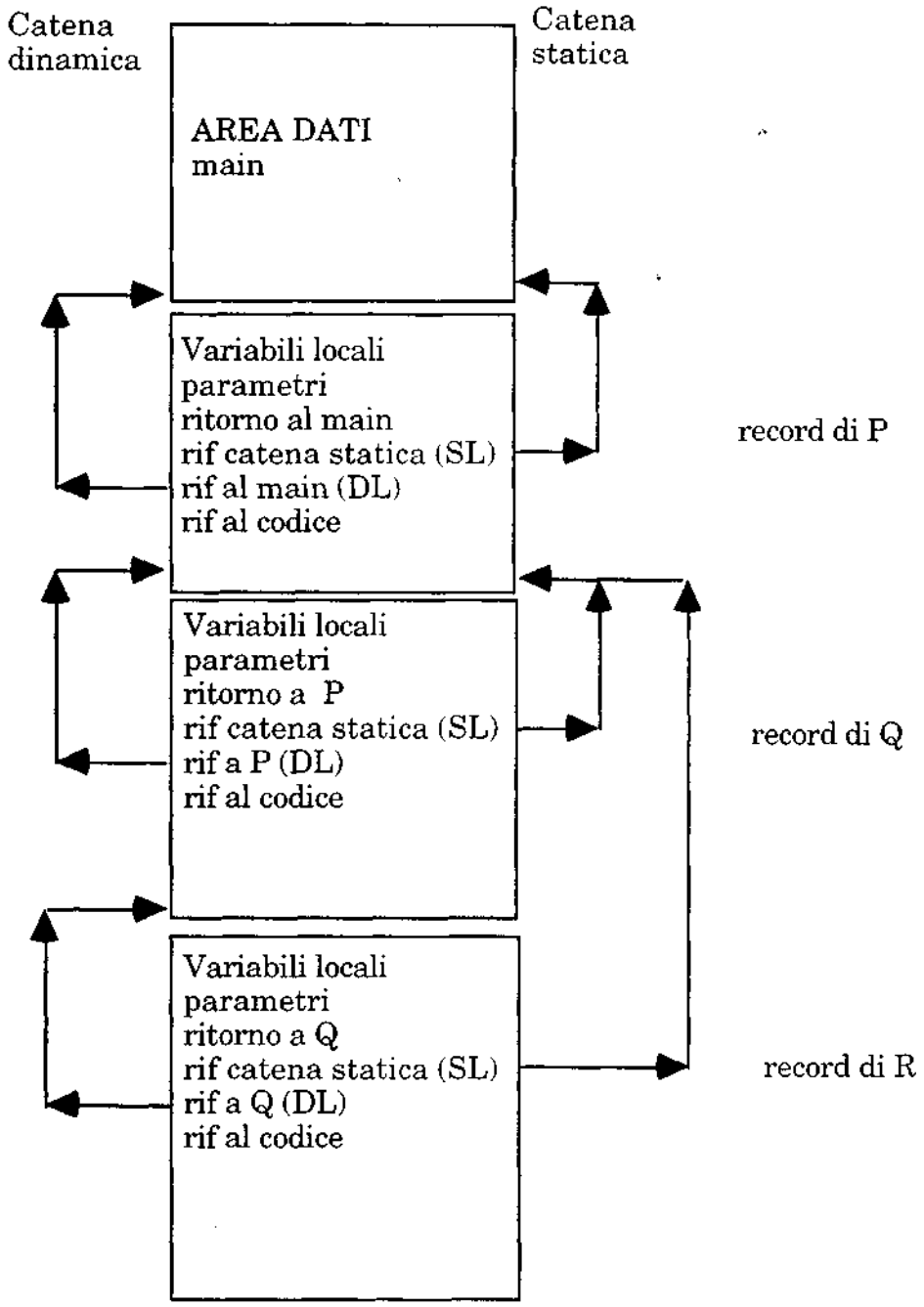
La catena statica fa riferimento alia gerarchia di sottoprogrammi dovuta alle dichiarazioni. Indica dove cercare (in quale ambiente) i riferimenti per le variabili non locali.

Il riferimento per la variabile non locale A usata in un sottoprogramma SP va cercato nel record di attivazione relativo all'attivazione più recente dell'unità di programma che contiene la dichiarazione di A visibile in SP.



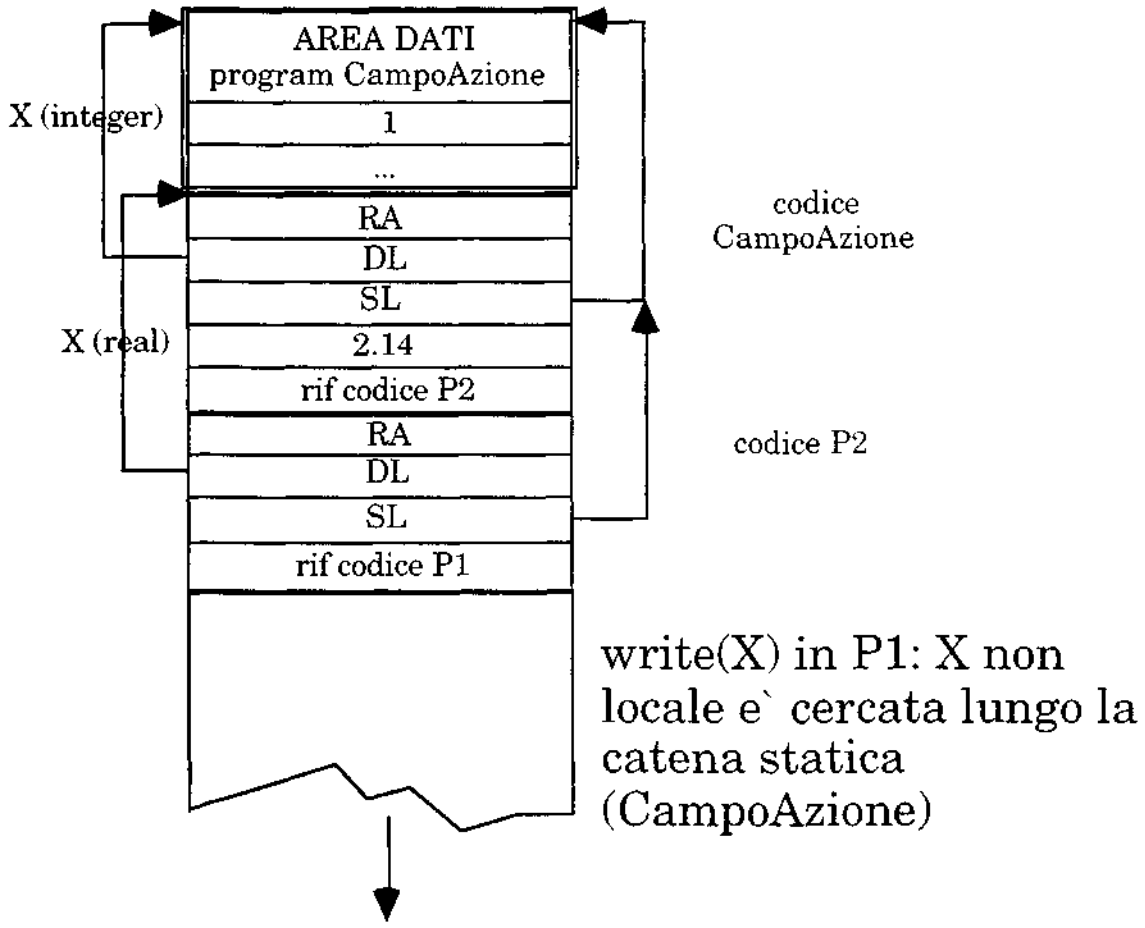


Nell'esempio, il riferimento di catena statica di R è il record di attivazione più recente di P (non di Q).



Se per le variabili non locali di R non viene trovato un riferimento nel record di attivazione di P, si prosegue lungo la catena statica (area dati del main).

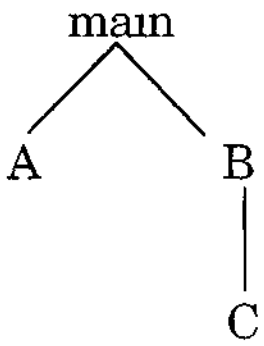
```
program CampoAzione (...);
var X:integer;
        procedure P1;
        begin
            writeln(X)                {X del main}
        end;
      procedure P2;
      var X:real;
          begin
              X:=2.14;                {X di P2}
              P1
          end;
begin                  {corpo di CampoAzione}
      X:=1;
      P2
end.
```



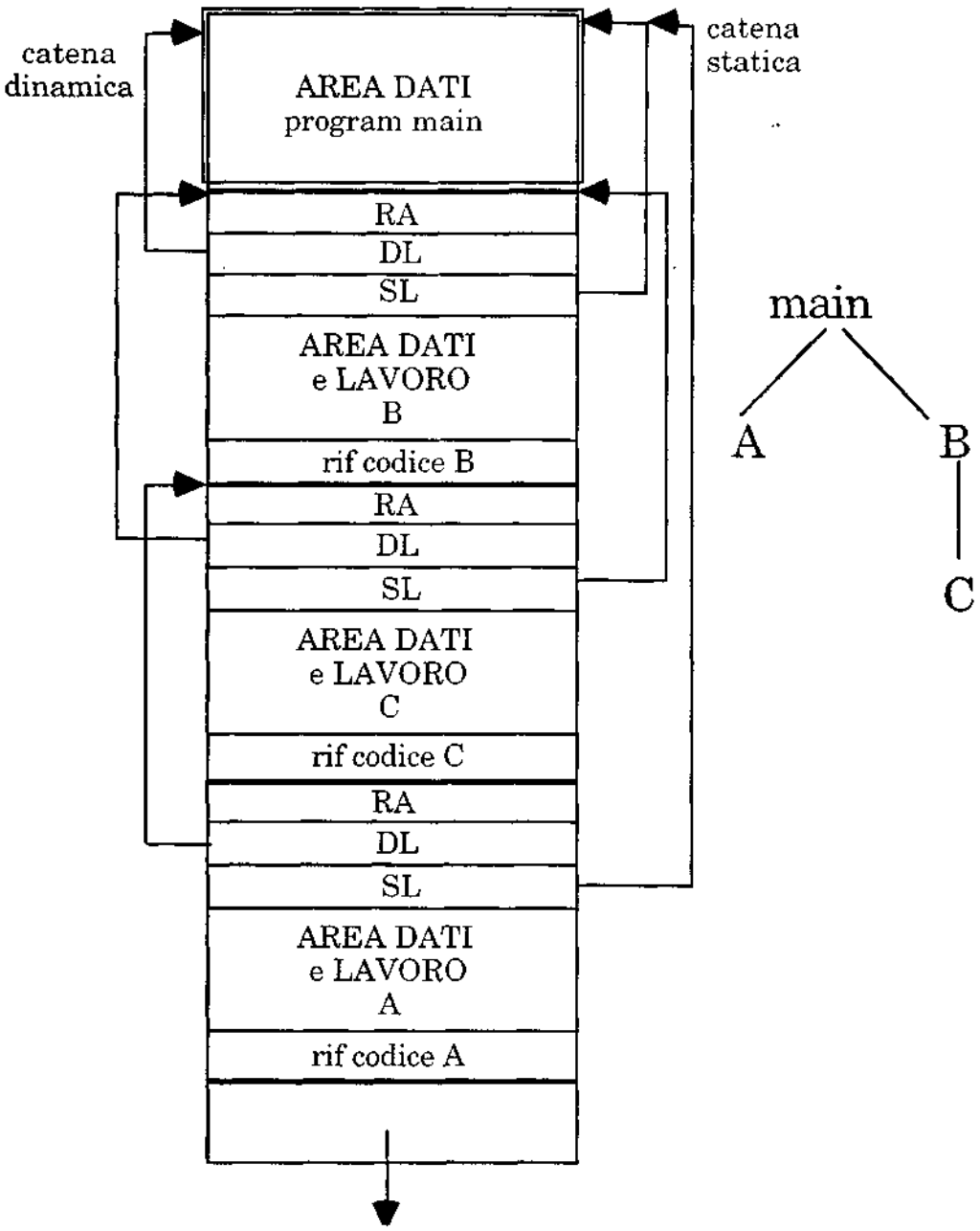
```
program main (...);
...
      procedure A(...);
      ...
      end;       {A}

      procedure B (...);

      ...
            procedure C(...); begin
                  ...; A(...); ...
            end;       {C} begin
            ...; C(...); ...
      end;       {B}
begin                  {corpomain}
      ...; B(...); ...
end.
```



main --> B --> C --> A


catena
dinamica
catena
statica
AREA DATI
program main
RA
DL
SL
AREA DATI
e LAVORO
B
rif codice B
RA
DL
SL
AREA DATI
e LAVORO
C
rif codice C
RA
DL
SL
AREA DATI
e LAVORO
A
rif codice A
main
A
B
C

Parametri e variabili locali:

Una "locazione" per ciascun parametro formale e ciascuna variabile locale (la locazione sarà costituita dal numero di byte necessari a memorizzare il dato -tipo).

Per le **funzioni,** si può immaginare che venga allocata anche una cella dove memorizzare il valore restituito dalla funzione (identificatore funzione).

Parametri con legame per valore:
Nella locazione viene scritto il valore assunto dal parametro attuale all'atto della chiamata.

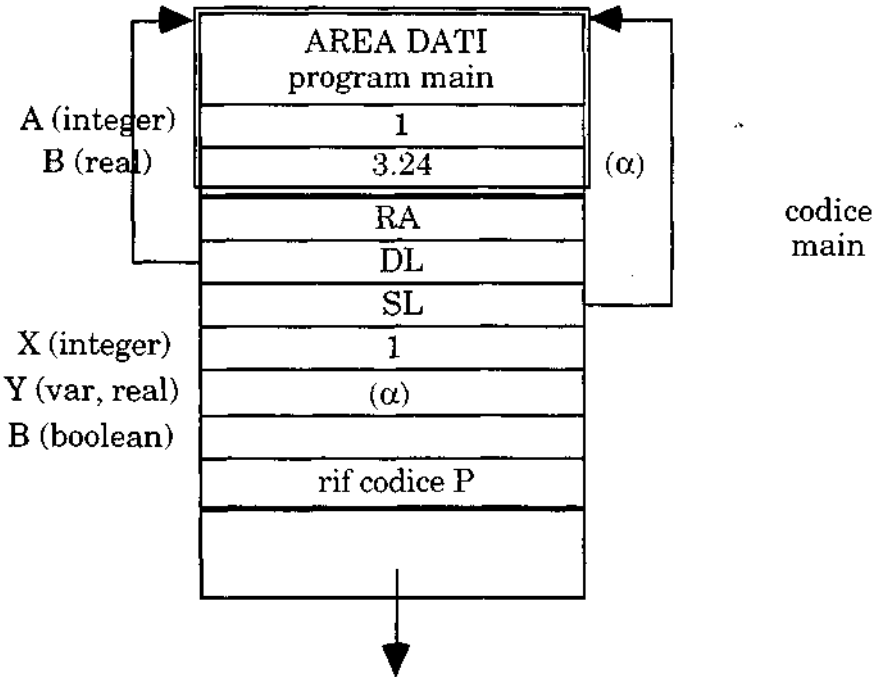
Parametri con legame per indirizzo:
Nella locazione viene scritto l'indirizzo della variabile che compare come parametro attuale all'atto della chiamata.

```
program main (input, output);
var A:integer; B:real;
      procedure P (X: real; var Y:real);
      var B: boolean;
      begin
            Y:=Y*2;X:=X*2;...
      end;
begin
      readln(A,B);
      P(A,B);
      ...

end.
```

In ingresso:

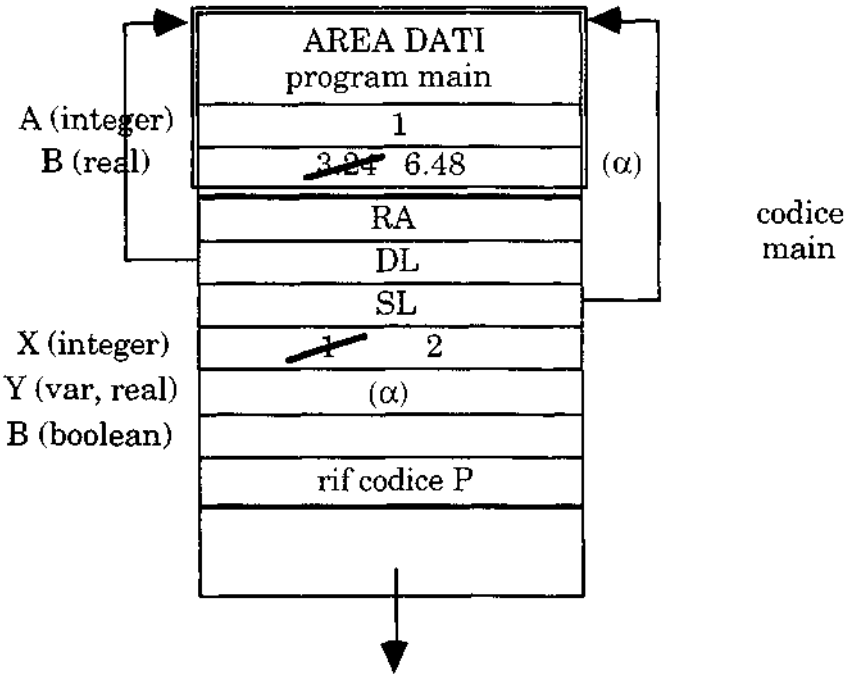
```
1        3.24
```



Dopo l'esecuzione delle istruzioni:

Y:= Y*2; X:=X*2

(codice di P), si ha:



La ricorsione:

Una funzione matematica è definita ricorsivamente quando nella sua definizione compare un riferimento (chiamata) a se stessa.

Esempio:

Funzione fattoriale su interi non negativi:

f(n) = n!

definita ricorsivamente come segue:

$$f(n) = \begin{cases} 1 & \text{se } n=0 \\ n*f(n-l) & \text{se } n>0 \end{cases}$$

Usando il metodo induttivo si specifica come tale funzione si comporta nel caso base e nel passo generico.

Induzione matematica: immaginanado di avere x_k, costruisci x_{k+1}.

Informalmente, il calcolo del fattoriale di un numero n viene ricondotto al calcolo del fattoriale di n-1, fino a raggiungere un caso base (fattoriale di 0), noto.

Metodo particolarmente utile per alcuni problemi (intrinsecamente ricorsivi) o che lavorano su strutture dati ricorsive (liste, alberi).

Esempi di problemi ricorsivi:

1) Generare l'n-esimo **numero di Fibonacci:**

$$fib(n) = \begin{cases} 0 & \text{se } n=0 \\ 1 & \text{se } n=l \\ fib(n-l)+fib(n-2) & \text{altrimenti} \end{cases}$$

2) Calcolo del **minimo di una sequenza di elementi:**

[al, a2, a3, ...] [al | [a2, a3, ...]]

min([a1]) = al
min([a1,a2])= al se a1<a2; altrimenti a2
min([a1 | Z]) = min([a1,min(Z)])

sviluppando si ottiene:

min([a1 | Z]) = min([a1,min([a2, min([a3,...])])])

3) **lunghezza** di una sequenza:

lung([]) = 0

lung([a1|Z]) = l+lung(Z)

4) **Appartenenza** di un elemento ad una lista:

member(Item,List)=false, se empty(List)
member(Item,List)= true, se not empty(List)
e Item = head(List)
member(Item,List) = member(Item,tail(List)).

5) Insieme dei **numeri naturali,** definito induttivamente come:

0 succ(0) succ(succ(0))....

Funzione somma:

sum(N,M) = N se M=0
succ(sum(N,pred(M))) altrimenti

Calcolo del fattoriale di un numero:

```
program main (input,output); type
positivi= 0..maxint;
var N:positivi;

    function fattoriale (n:positivi): positivi;
    begin
        if n=0 then fattoriale:=l
        else fattoriale:= n*fattoriale(n-l)
    end;            {fattoriale}

begin
    readln(N);
    writeln('fattoriale', fattoriale(N))
end.
```

§

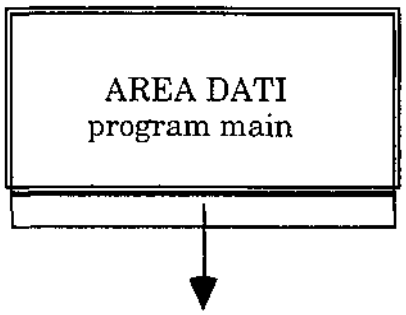
Numero di Fibonacci:

```
    function fib (n:positivi): positivi; begin
        case n of
            0: fib:=0;
            1: fib:=1;
        else fib:= fib(n-l)+fib(n-2)
        end
end;                {fibonacci}
```

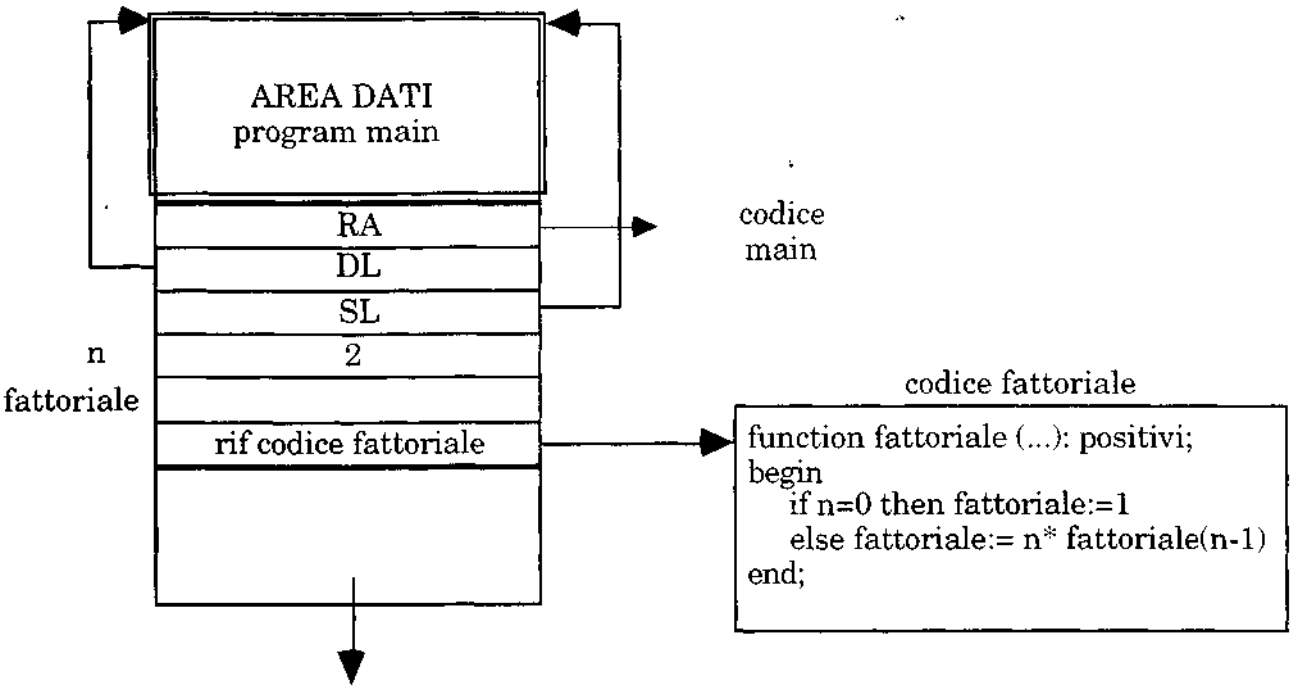
Esercizio:
Pila di attivazioni per la valutazione di fattoriale(2):

```
program main (output);
type positivi: 0.. maxint;
      function fattoriale (n: positivi): positivi;
      …
      end;              {fattoriale}
begin
      write(fattoriale(2))
end.
```

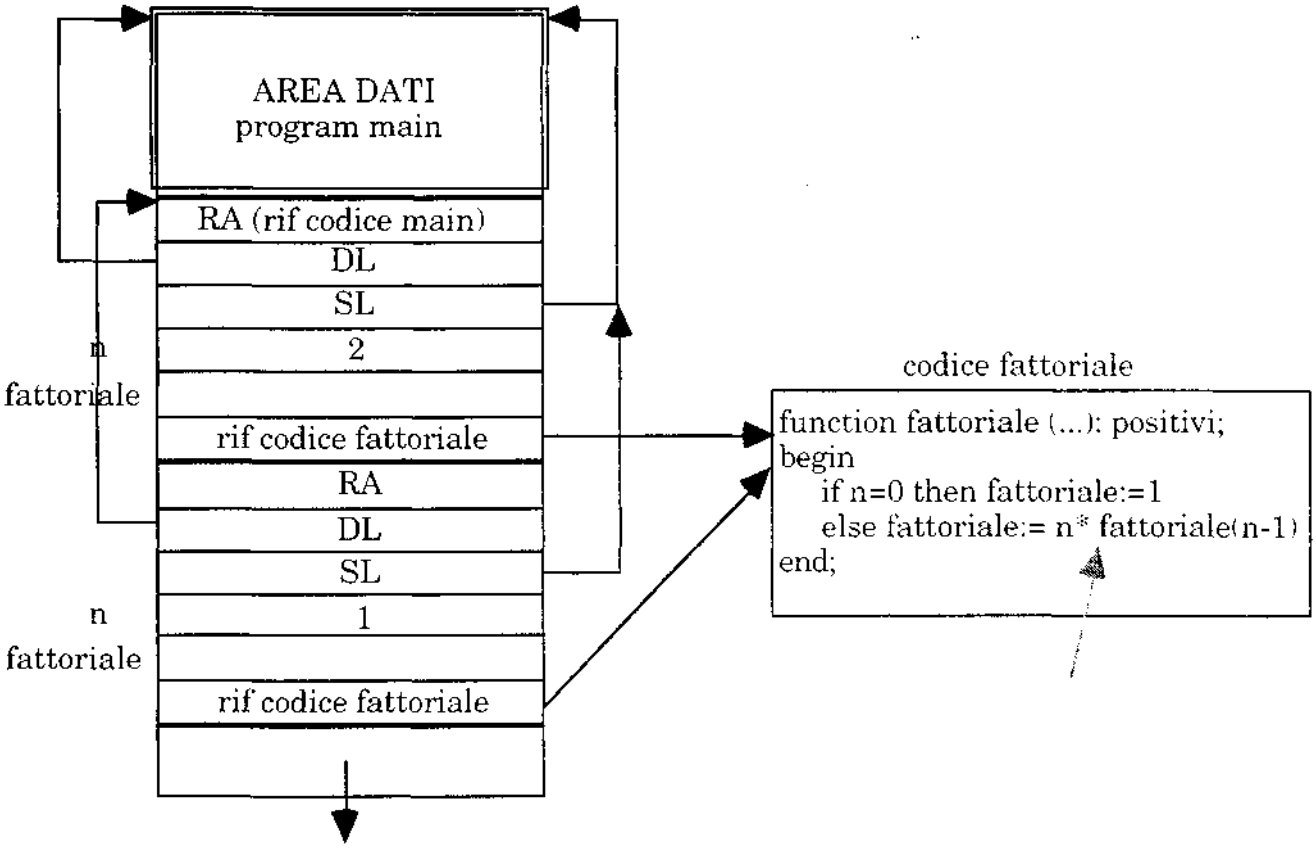
All'inzio del main program:



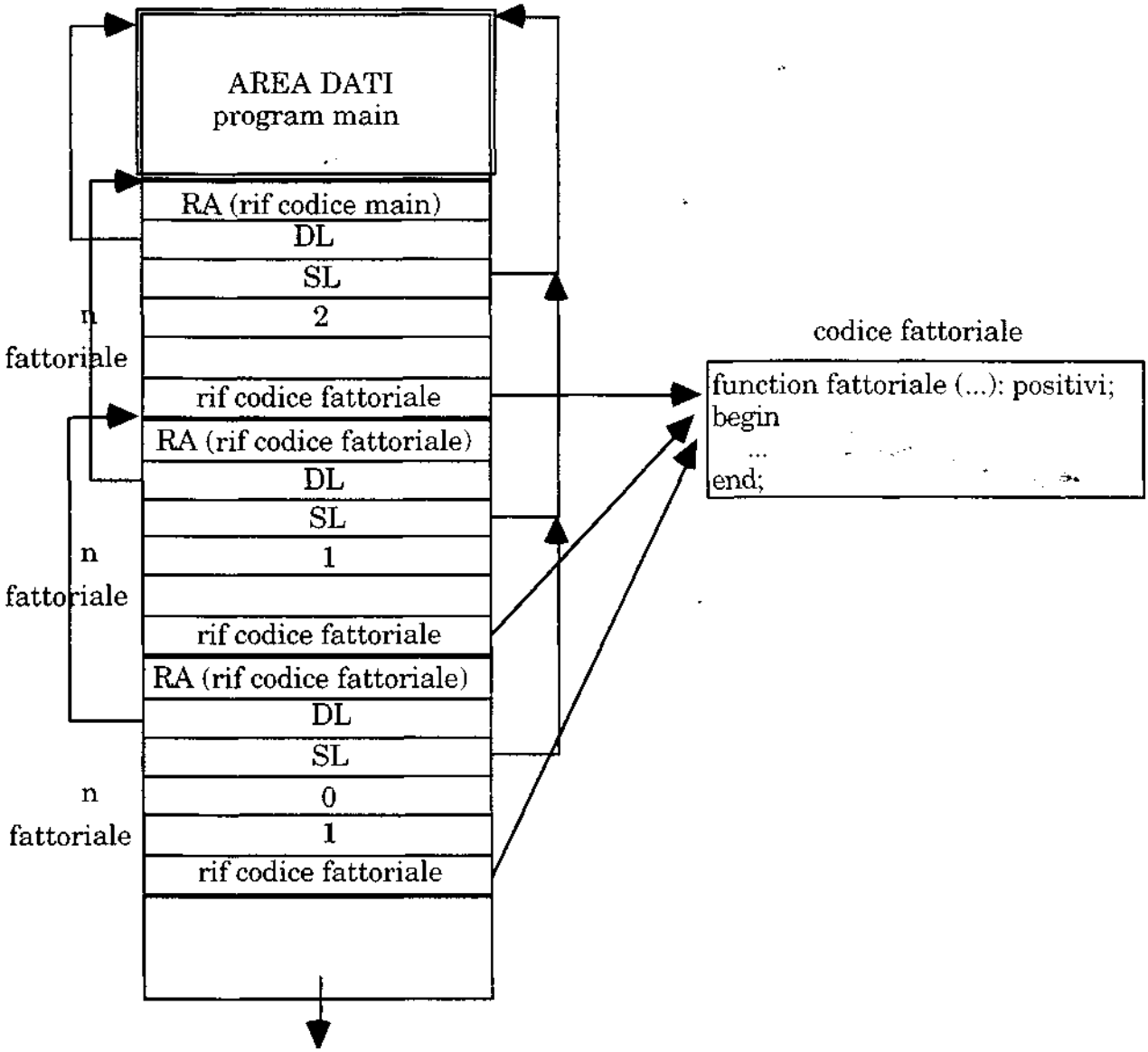
Dopo la prima attivazione della funzione fattoriale (fattoriale(2)):



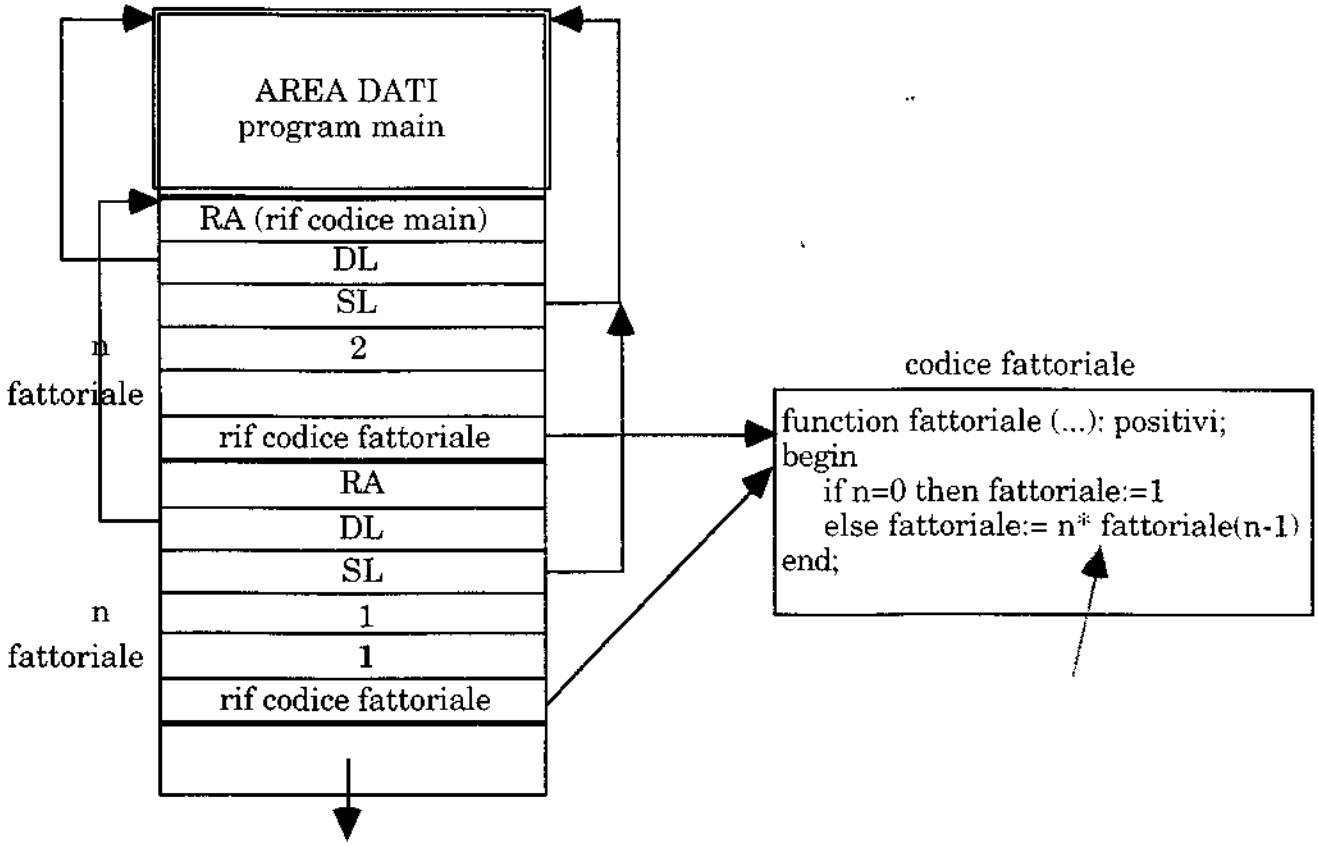
Dopo la seconda attivazione (fattoriale(1):



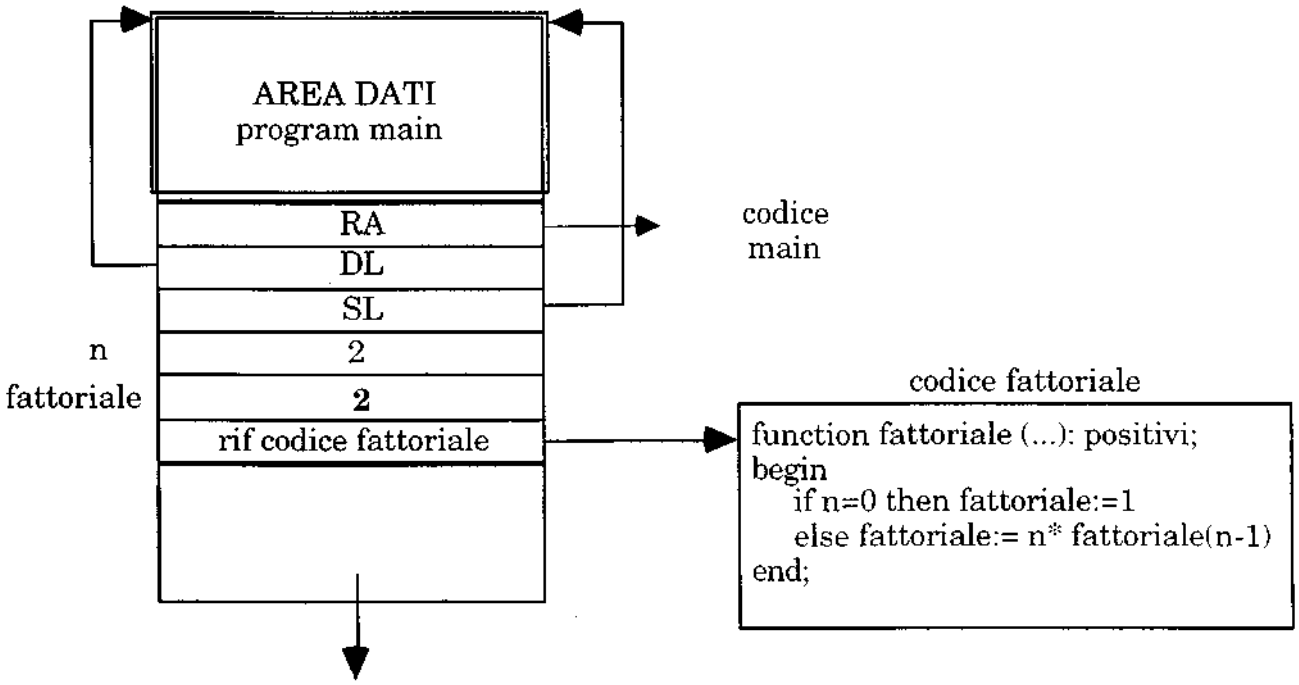
Dopo la terza attivazione (fattoriale(0)):



Termine della terza attivazione (return):



Termine della seconda attivazione (return):



Al termine della prima attivazione di fattoriale, viene restituito al main program il valore 2.

Ricorsione ed iterazione:

Soluzioni ricorsive sono generalmente più vicine alla definizione matematica di certe funzioni:

- serie ricorsive, quali ad esempio i numeri di Fibonacci;
- funzioni matematiche ricorsive (fattoriale, etc.).

Versioni iterative sono generalmente più efficienti di una soluzione ricorsiva (sia in termini di memoria che di tempo di esecuzione).

Schemi di (sotto)programmi ricorsivi del tipo:

$P(x_k)$ ≡ **if** caso base (x_0)
then termina
else expr($P(x_{k-1})$)

sono esprimibili con costrutti iterativi come segue:

$P(x_k)$ ≡ inizializzazione (caso base, x_0)
while B **do** expr(x_{k-1})

Non tutti i linguaggi di alto livello supportano procedure ricorsive (FORTRAN).

Calcolo del fattoriale (versione iterativa):

```
function fattoriale (n:positivi): positivi;
var I, F:positivi;
begin
      F:= 1;            {caso base}
      I:=0;
      while I<n do begin
                  I:=I+1;
                  F:=F*I
            end;
fattoriale := F
end;              {fattoriale iterativo}
```

Esercizi:

Disegnare i record di attivazione generati dall'esecuzione dei seguenti programmi:

1)

```
program mainl (output);
var x: real;
      procedure stranoloop;
      const limit=3;
            function ripeti (cont: integer): real;
            begin
                  if cont<limit then
                        begin
                              x:=x*0.5;
                              ripeti:=ripeti(cont+1)
                        end
                  else ripeti :=x
            end;              {ripeti}
      begin
            writeln(ripeti(1))
      end;              {stranoloop}
begin
      x:=1.0E-l;
      stranoloop
end.
```

Indicare qual è il risultato stampato e come avviene l'accesso alla variabile globale x da parte della funzione ripeti.

2)

```
program main2 (input,output);
      function g: integer;
      var x, cont: integer;
            function f (val: integer): integer;
            begin
                  if not eof(input) then
                        begin
                              readln(x);
                              f:=f(val+x)
                         end
                  else f:=val
            end;              {f}
      begin
            g:=f(0)
      end;                                  {f}
begin
      writeln(g)
end.
```

Indicare qual è il risultato stampato se in ingresso vengono dati i valori:
5
2
4

Attivazioni: main --> g --> f(0) --> f(5) --> f(7) --> f(11)

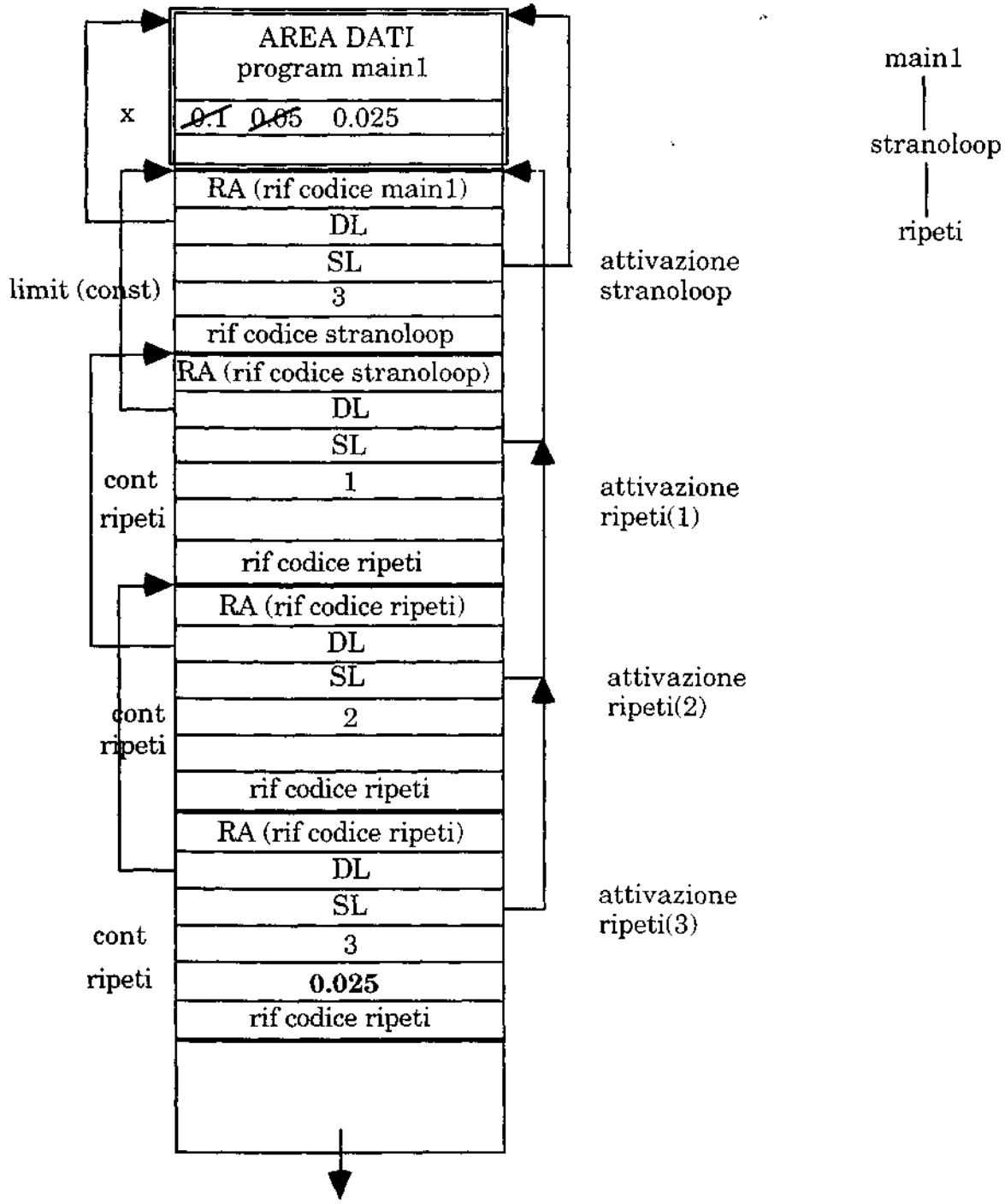
3)

```
program main3 (output);
var n:integer;
      procedure fib(n: integer; var k:integer); function f
            (i, p, s, n: integer): integer; begin
                  if i=n then f:=s
                  else f:=f(i+l, s, p+s, n)
             end;              {f}
      begin
            if n<0 then k:=0
            else k:=f(1, 0,1, n)
      end;              {fib}
begin
      fib(3,n); writeln(n)
end.
```

Indicare qual è il risultato stampato.

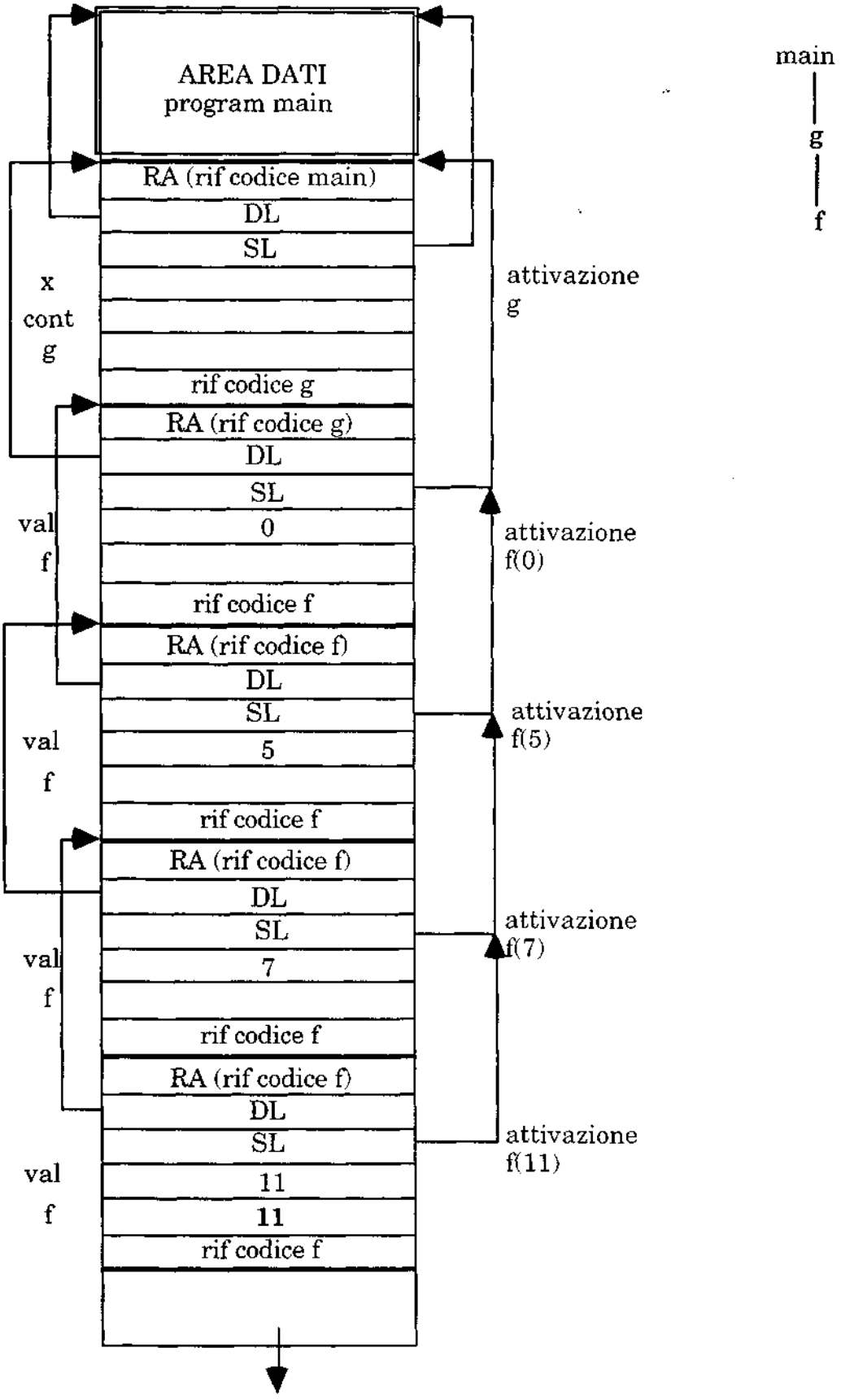
Soluzioni:

1)



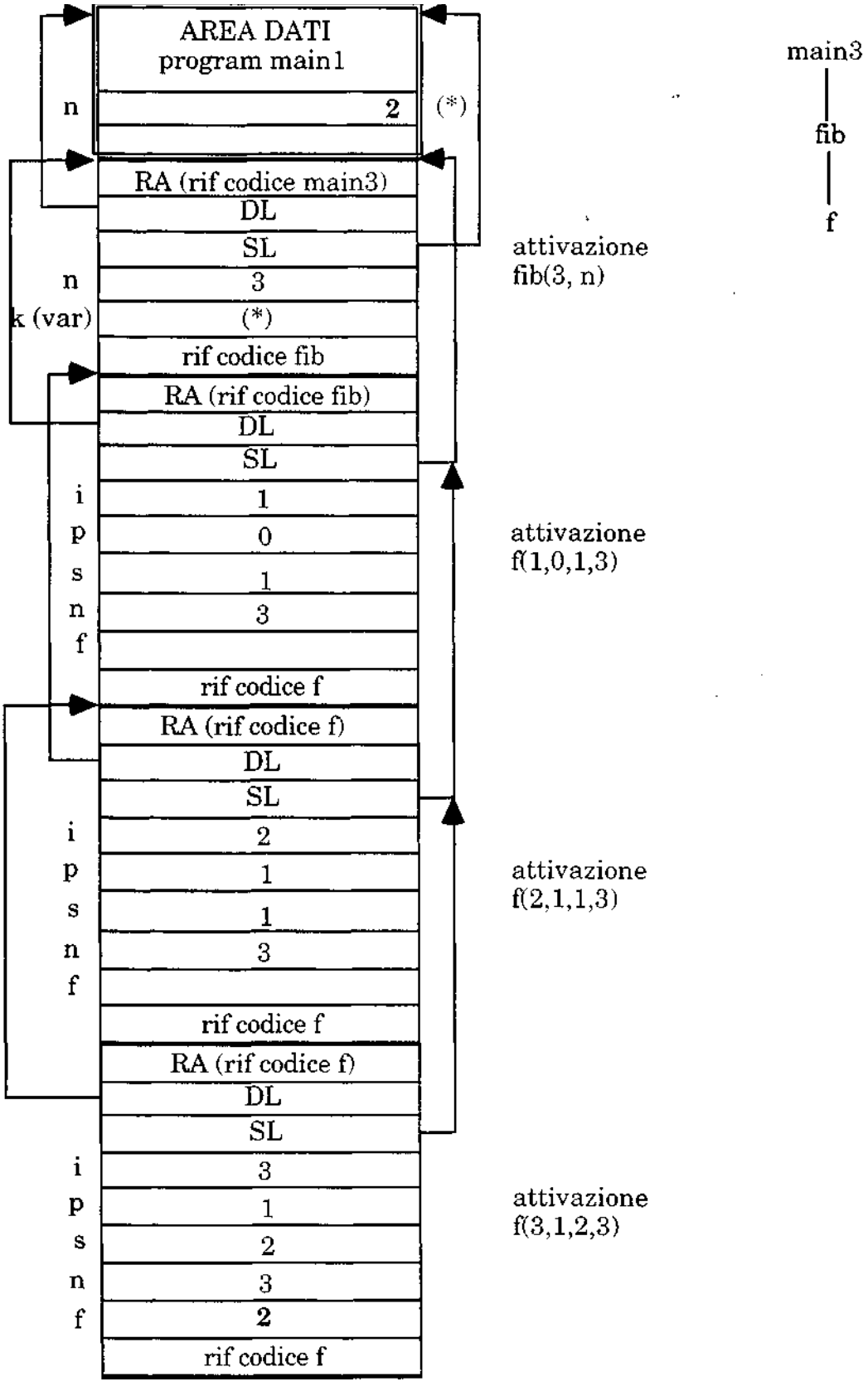
Stampa: 2.5 E-2

2)


AREA DATI
program main
RA (rif codice main)
DL
SL
x
cont
g
rif codice g
RA (rif codice g)
DL
SL
val
f
0
rif codice f
RA (rif codice f)
DL
SL
val
f
5
rif codice f
RA (rif codice f)
DL
SL
val
f
7
rif codice f
RA (rif codice f)
DL
SL
val
f
11
11
rif codice f
attivazione
g
attivazione
f(0)
attivazione
f(5)
attivazione
f(7)
attivazione
f(11)
main
g
f

Stampa: 11

3)


AREA DATI
program main1
n
2
(*)
RA (rif codice main3)
DL
SL
n
3
k (var)
(*)
rif codice fib
RA (rif codice fib)
DL
SL
i
1
p
0
s
1
n
3
f
rif codice f
RA (rif codice f)
DL
SL
i
2
p
1
s
1
n
3
f
rif codice f
RA (rif codice f)
DL
SL
i
3
p
1
s
2
n
3
f
2
rif codice f
attivazione
fib(3, n)
attivazione
f(1,0,1,3)
attivazione
f(2,1,1,3)
attivazione
f(3,1,2,3)
main3
fib
f

Stampa: 2

Ricorsione Tail:

Si é in presenza di **Tail Recursion** quando la chiamata ricorsiva di una procedura P è l'ultima istruzione del codice di P.

Consente di ottimizzare lo spazio di memoria allocato sullo stack.

Esempio:

```
function f (x, y:integer): integer;
begin
      if x=0 then f:=y
      else
            if x>0 then f:=f(x-l,y+l)
            else f:=f(x+l,y-l)
end;
```

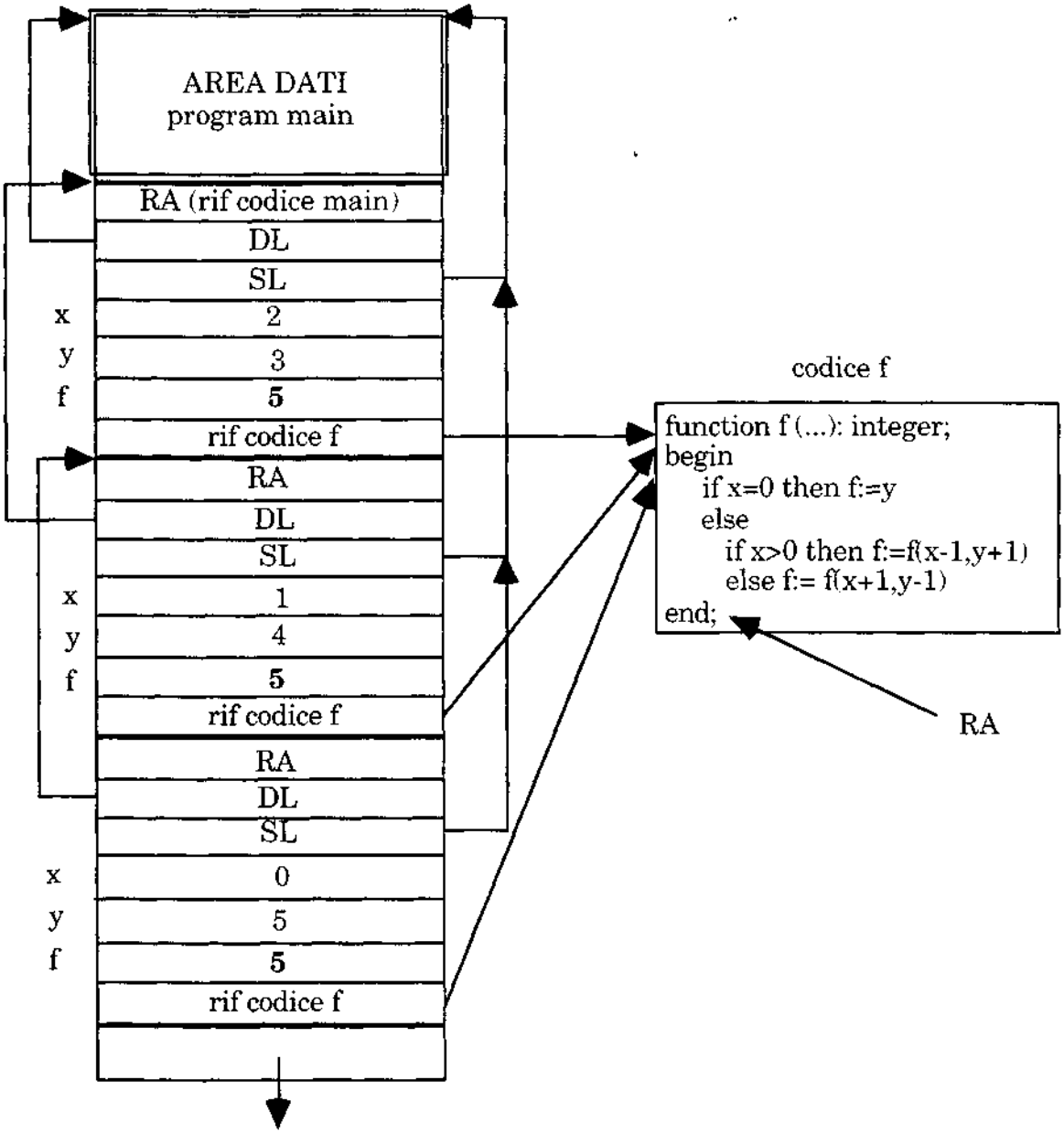
(in pratica, somma x ad y)

Corrisponde ad un ***processo computazionale iterativo.***

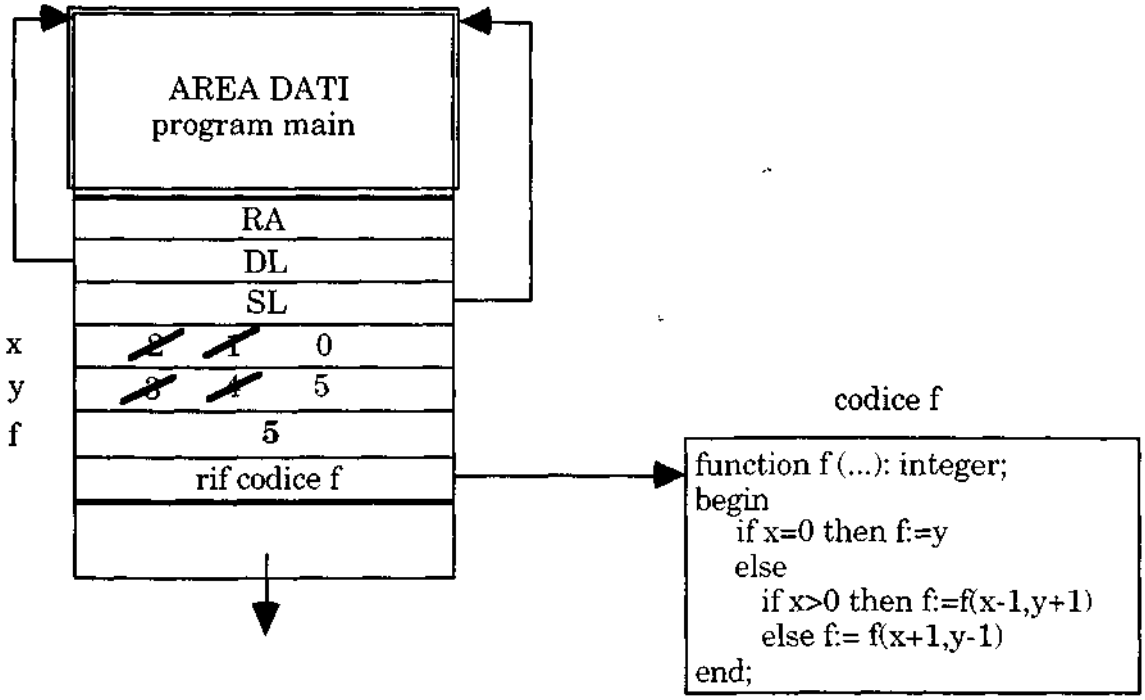
Un processo computazionale è ***ricorsivo*** quando è caratterizzato da una catena di operazioni posticipate , il cui risultato è disponibile solo dopo che l'ultimo anello della catena si è concluso.

In un processo computazionale ***iterativo,*** ad ogni passo é disponibile una frazione del risultato.

Nel main: writeln(f(2,3));



In pratica l'ottimizzazione consiste nell'utilizzare il medesimo record di attivazione per tutte le attivazioni successive della funzione tail ricorsiva. E' come se si avesse una deallocazione anticipata del record di attivazione.

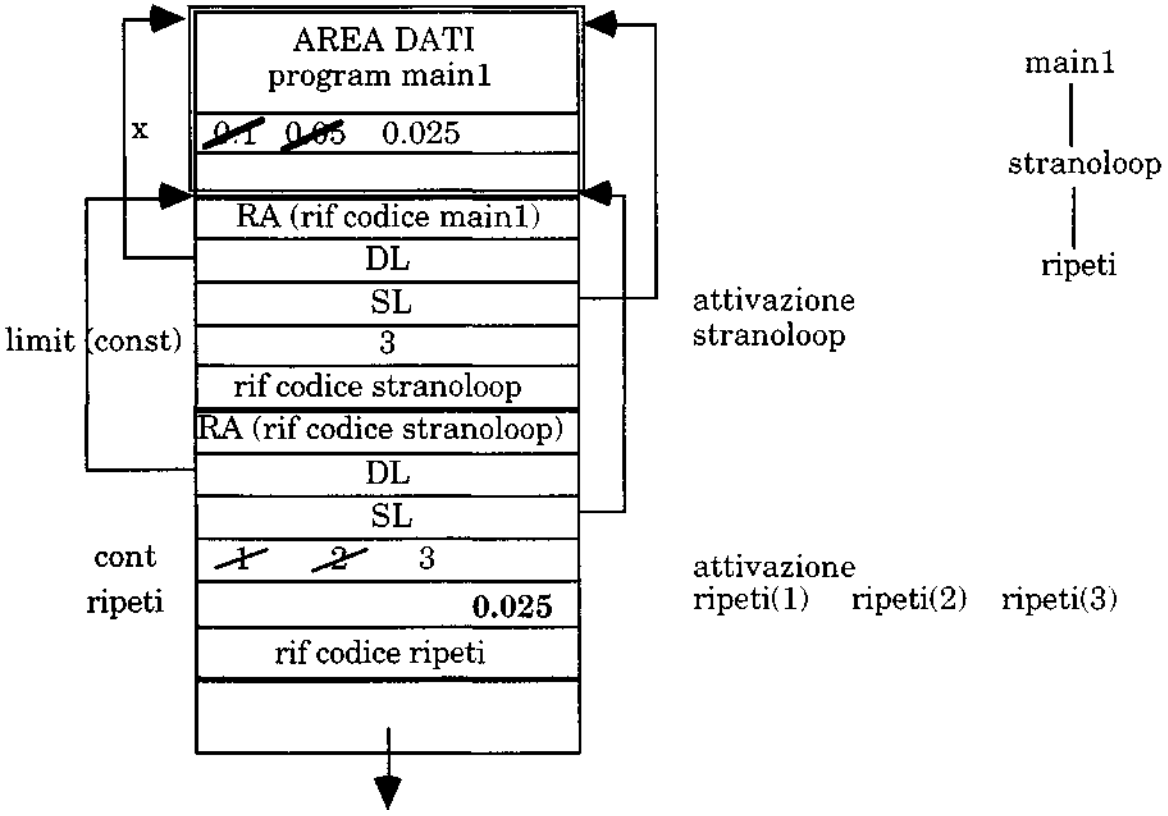

AREA DATI
program main
RA
DL
SL
x
y
f
2
1
0
3
4
5
5
rif codice f
codice f
function f (...): integer;
begin
if x=0 then f:=y
else
if x>0 then f:=f(x-1,y+1)
else f:= f(x+1,y-1)
end;

Esercizi:

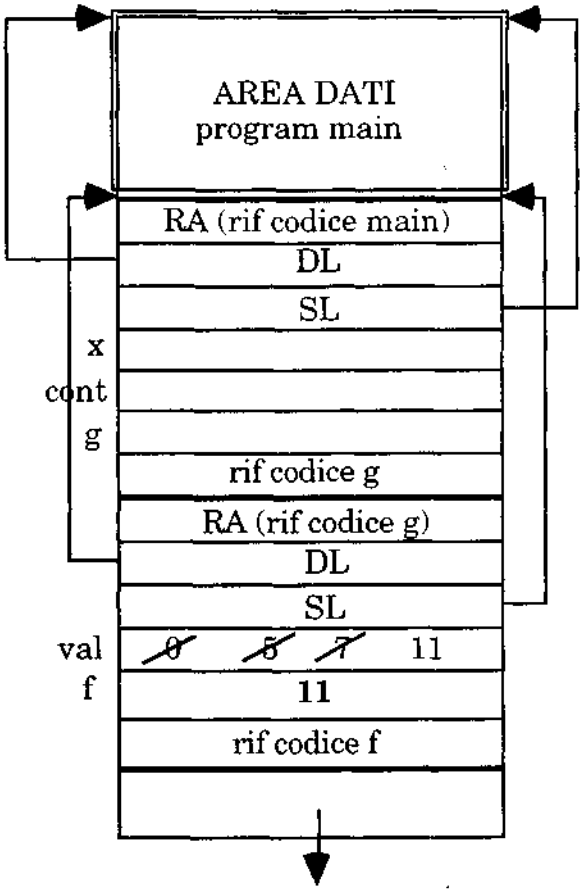
Discutere come cambiano i record di attivazione degli esercizi 1), 2) nel caso la macchina astratta Pascal realizzi in modo efficiente la ricorsione tail?

Soluzioni:

1)



2)



attivazione
g

attivazione
f(0) f(5) f(7) f(11)

In alcuni casi è possibile scrivere versioni tail-ricorsive di funzioni e procedure.

Esempio:

```
function fattoriale (n:positivi): positivi;
begin
      if n=0 then fattoriale:=1
      else fattoriale: = n*fattoriale(n-l)
end;              {fattoriale}
```

Versione tail-ricorsiva:

```
function fatt (n:positivi): positivi;

      function fatt_tail (val,cont:positivi): positivi;
      begin
            if cont<=n
           then fatt_tail:=fatt_tail(val*cont,cont+l)
           else fatt_tail:=val
      end;
begin
      fatt:=fatt_tail(1,1)
end;              {fattoriale - versione tail recursive}
```

Allocazione della memoria ad Heap

Nel caso dell'allocazione a stack trattata precedentemente, le esigenze di memoria si devono tutte conoscere nel momento in cui si entra in un nuovo blocco.

Questo non e' vero in alcuni casi. Si pensi ad array o stringhe di lunghezza indefinita oppure alle strutture dati a lista od ad albero presenti in tanti linguaggi di programmazione (Pascal, Lisp, Prolog).

In questo caso si utilizza una gestione ad **Heap** della memoria.

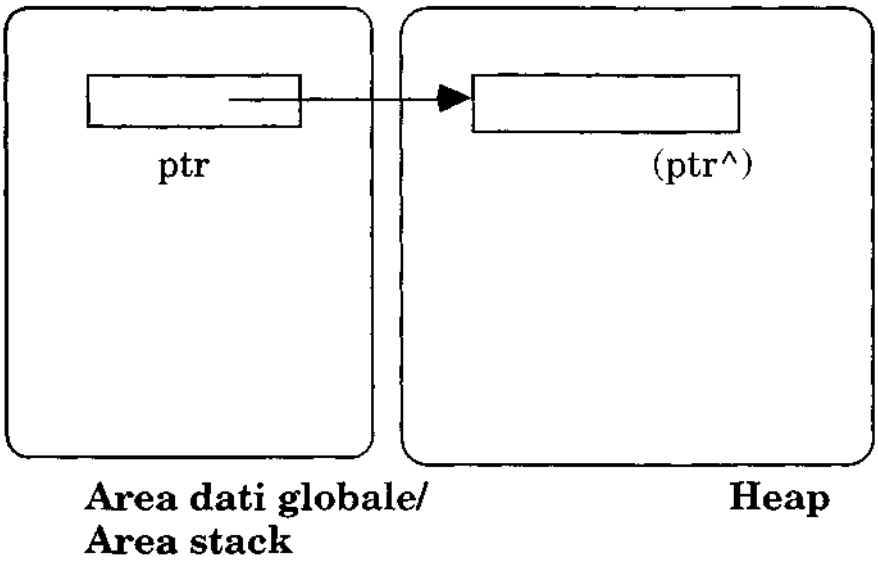
In pratica, l'Heap e' costituito da un blocco di memoria ampia e contigua. Quando c'è richiesta di memoria, un manager a tempo di esecuzione attribuisce tale memoria. Quando lo spazio non e' piu' necessario, il manager cerca se possibile di rilasciare la memoria per renderla disponibile ad altri.

L'allocazione della memoria nell'heap puo' essere **esplicita** od **implicita.**

In Pascal, ad esempio, l'allocazione e la deallocazione sono esplicite.

Strutture dati dinamiche e ricorsive: il tipo pointer in Pascal:

Consente di definire strutture dati collegate (liste, alberi, grafi, etc.) e di allocare dinamicamente "variabili" (dette ***variabili dinamiche).***



Le locazioni di memoria per le variabili puntate sono prelevate da un'area di memoria **(heap)** che equivale ad una lista libera.

type <identificatore-ptr>= ^<tipo-puntato>

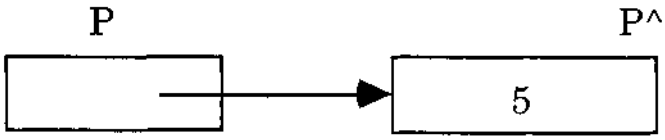
Variabili del tipo <identificatore-ptr> fanno riferimento a variabili di tipo <tipo-puntato>.

Per consentire controlli statici di tipo, un puntatore Pascal deve puntare (referenziare) dati di un tipo prefissato.

```
type puntatore_intero= ^integer;
var P: puntatore_intero;

begin
...;

new(P);          {creazione dell'oggetto P^}
P^:=5; ...
end.
```

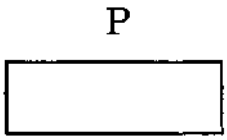


La variabile P conterrà solo riferimenti a celle che contengono dati interi.

P^, ***variabile puntata*** (di tipo intero).

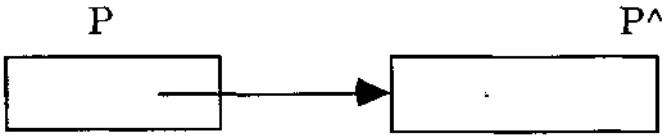
A qualunque variabile di tipo pointer può essere assegnato il valore **nil**.

P:=**nil**;



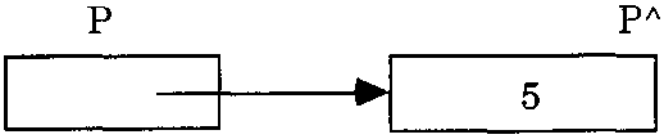
Per creare la ***variabile puntata*** occorre chiamare la procedura standard **new:**

new(P)**;**



L'inizializzazione di una variabile puntatore mediante ***new*** provoca la creazione di una variabile dinamica allocata nell'area ***heap.***

P^:=5;



Dereferencing:
Il dereferenziamento di una variabile puntatore si ottiene con l'operatore ^.

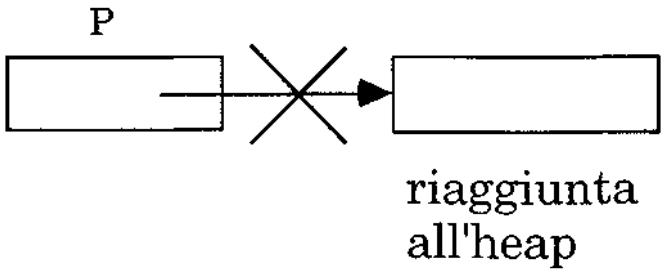
P^ variabile puntata o ***referenziata***

II ***tempo di vita*** della "variabile referenziata" P^ non è affatto in relazione con il blocco di programma in cui viene eseguita l'operazione ***new***, nè con il blocco in cui è dichiarato P.

E' compito del programmatore rilasciare la memoria occupata da P^, quando questa non serve più.

Si rilascia la locazione di memoria referenziata da un puntatore con la procedura stardard **dispose:**

dispose(P);



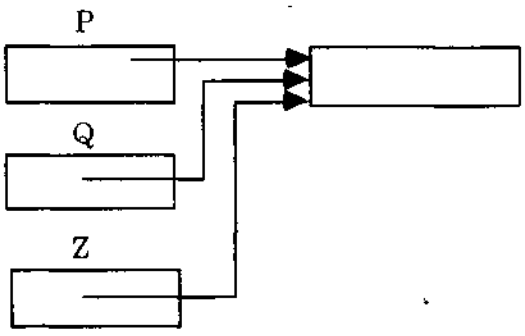
Il tempo di vita di P^ inizia con l'esecuzione di new(P) fino all'esecuzione di dispose(P). ***Dopo*** questa operazione P^ ***non esiste più.***

Il valore di P é indefinito, dopo questa operazione.

Più variabili puntatore possono riferire il medesimo oggetto:

var P,Q,Z: puntatore_intero;

new(P);
Q:=P;
Z:=Q;



Se l'area di memoria puntata da P viene rilasciata:

dispose(P)**;**

Z e Q diventano riferimenti pendenti **(*dangling references).***

Definizione di strutture dati ricorsive attraverso pointer:

In una definizione di un tipo pointer:

type *puntatore* = ^ *tipo;*

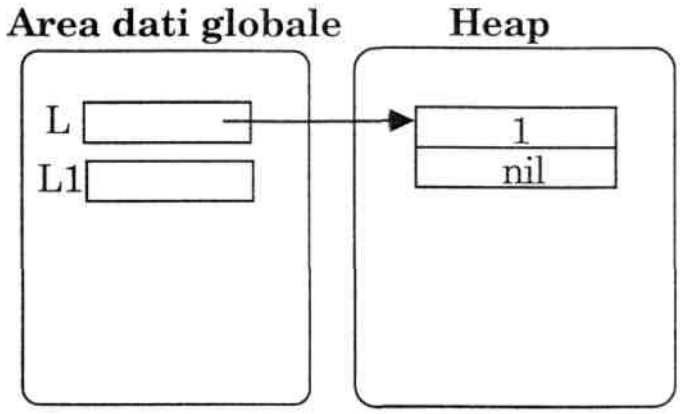
l'identificatore *tipo* può non essere ancora stato definito (unica eccezione consentita in Pascal).

La sua definizione può quindi utilizzare, a sua volta, il tipo *puntatore* che si sta definendo per realizzare una ***struttura dati ricorsiva.***

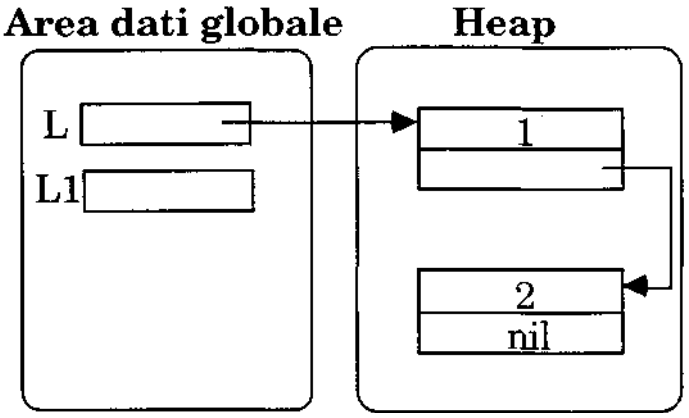
Esempio:

```
program es_pointer;
type    Lista = ^ Elemento;
          Elemento = record
                valore: integer;
                next: Lista
                end;
```

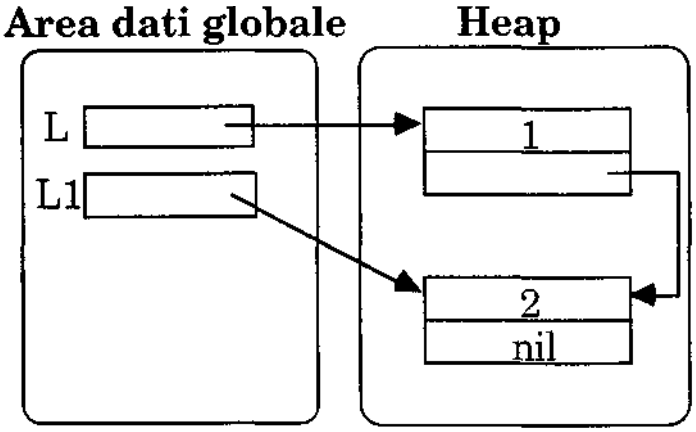
```
var       L, L1: Lista; begin
      L:= nil;
      new(L); L^.valore:= 1; L^.next:= nil;
```



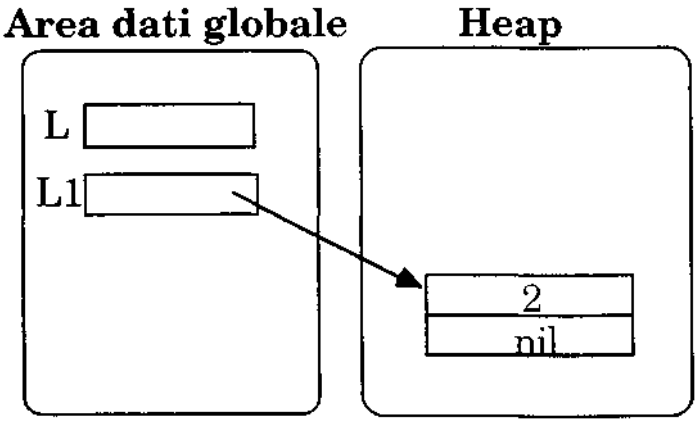
```
new(L^.next);
L^.next^.valore:=2;
L^.next^.next:=nil;
```




```
L1:=L^.next;
```




```
dispose(L);
```



end.

Dereferencing:
Il dereferenziamento di una variabile puntatore si ottiene con l'operatore ^.

P^ variabile puntata o ***referenziata***

Nota Bene: eseguire in sequenza le due operazioni:

new(P); P:=nil;

non ha senso (si preleva spazio dallo heap senza utilizzarlo!!)

Quando per le liste si utilizza la rappresentazione collegata realizzata con **record** e **pointer** é importante rilasciare memoria man mano che questa non viene utilizzata più.

Il recupero di spazio di memoria non più utilizzato da liste prende il nome di ***garbage collection.***

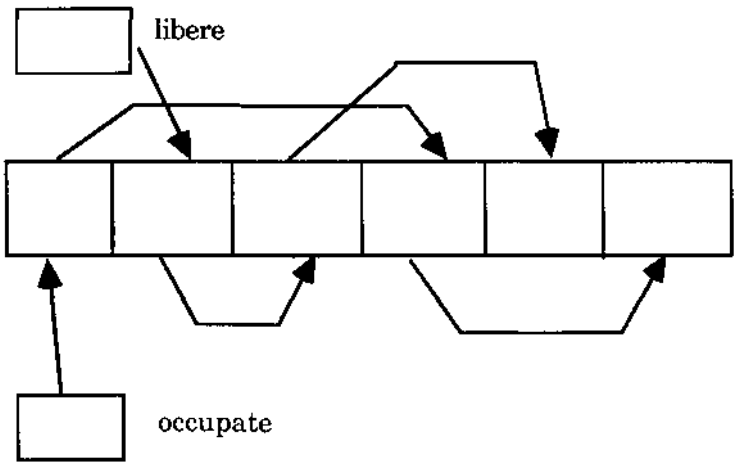
Se fatto "metodicamente" con la procedura **dispose,** si dice "on the fly".

Tecniche di allocazione della memoria ad heap

E' differente a seconda che la deallocazione sia implicita od esplicita. L'allocazione implicita avviene ad esempio in linguaggi quali il linguaggio LISP ogni volta che con l'operatore di *cons* si crea una nuova lista. L'allocazione esplicita e' tipica di linguaggi quali Pascal.

Allocazione esplicita:

La forma piu' semplice di allocazione dinamica coinvolge blocchi di lunghezza prefissata. Tali blocchi possono essere legati in liste di liberi od occupati.



Quando i blocchi sono allocati e poi deallocati, la memoria puo' diventare frammentata. Quindi un heap e' costituito da blocchi alternati che possono essere liberi o frazionati.
Il problema diventa grave se i blocchi allocati possono essere di lunghezza variabile perche' in quel caso potremmo non riuscire ad allocare la memoria richiesta, perche' in teoria libera, ma frammentata. Ci sono varie politiche per ridurre questo problema (ad esempio, una politica puo' essere quella di allocare sempre per primi i blocchi piu grandi).

Deallocazione implicita

Utilizzando funzioni quali *list*, *cons* in LISP, si costruiscono nuove strutture dati. Durante la computazione tali celle possono non essere più referenziate.

Quando lo spazio di memoria è stato completamente utilizzato esse vanno recuperate: GARBAGE COLLECTOR

II problema e' riconoscere se un blocco di memoria e' ancora in uso.

Assumeremo che un blocco e' in uso se e' possibile per l'utente riferire le informazioni nel blocco.

Il riferimento al blocco puo' avvenire mediante un puntatore o mediante una catena di puntatori. Il compilatore deve quindi sapere la posizione dei puntatori (si possono immaginare in una zona fissa della memoria)

Si possono utilizzare due approcci:

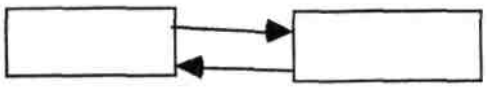
- Contare i riferimenti;
- Tecniche di marcatura.

Contare i riferimenti

Si tiene traccia del numero di blocchi che puntano direttamente al blocco in esame.
Se il valore del contatore e' zero, puo' essere deallocato. E' abbastanza costoso.

Esempio (assegnamento di due puntatori): p:=q;
II count del blocco puntato da p è decrementato di 1, quello di q incrementato di 1.

Problema: situazioni cicliche fra puntatori.



Non sono referenziati da nessuno accessibile, ma il loro contatore e' uguale a 1.

Tecniche di marcatura

Periodicamente si esegue un algortimo detto "mark and sweep" in 2 fasi:

1) **Marcatura** (alterando un bit oppure una tabella) delle celle dello heap che contengono variabili dinamiche usate correntemente dal programma;

2) **Recupero** delle celle non marcate come libere. Si recuperano nella lista libera tutte le celle dello heap non marcate e si toglie la marcatura alle altre.