Sintassi e Semantica

- **Sintassi** — forma del linguaggio
  regole per comporre frasi come sequenze di parole
- **Semantica** — significato delle frasi valide
- **Lessicali** — insieme di caratteri (alfabeto) + regole per comporre parole valide

Extended Backus-Naur Form

(a) Regole sintattiche

0 o più volte

<program> ::= { <statement>* }

<statement> ::= <assignment> | <conditional> | <loop>

<assignment> ::= <identifier> = <expr> ;

1 o più volte

<conditional> ::= **if** <expr> { <statement>$^{+}$ } |
        if <expr> { <statement>$^{+}$ } **else** { <statement>$^{+}$ }

<loop> ::= **while** <expr> { <statement>$^{+}$ }

<expr> ::= <identifier> | <number> |
        (<expr>) | <expr> <operator> <expr>

EBNF

(b) Regole Lessicali

<operator> ::= + | - | * | / | = | /= | < | > | <= | >=

<identifier> ::= <letter> <ld>*

<ld> ::= <letter> | <digit>

<number> ::= <digit>+

<letter> ::= a | b | c | . . . | z

<digit> ::= 0 | 1 | . . . | 9

Nota: * è sia la moltiplicazione aritmetica,
quindi un simbolo, che un metasimbolo

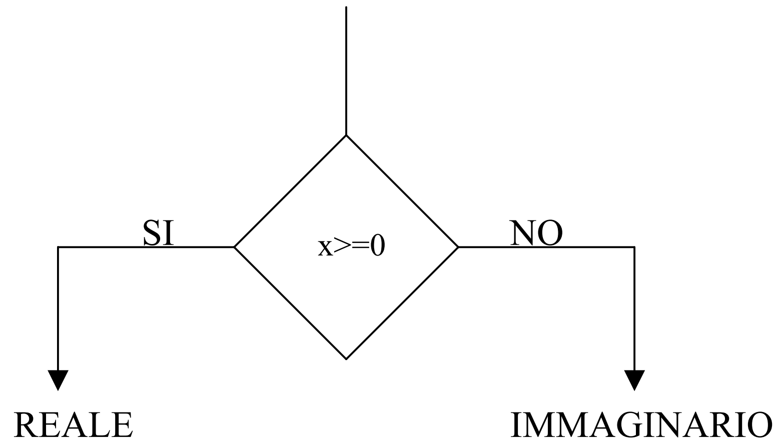
Esercizio

Utilizzando il linguaggio descritto dalla seguente EBNF:

REGOLE SINTATTICHE:

- <program> ::= { <statement>* }
- <statement> ::= <assignment> | <loop>
- <assignment> ::= <identifier> = <expr> ;
- <loop> ::= **while** <expr> { <statement> + }
- <expr> ::= <identifier> | <number> | (<expr>) | <expr> <op> <expr> | !<expr>
- <op> ::= == | <= | >= | != | && | ||
- … <identifier> e <number> rappresentano risp. identificatori C e numeri interi

scrivere una funzione in grado di predire se la radice di un numero è reale o immaginaria



- SOLUZIONE IMMEDIATA:

  IF x>=0 THEN “REALE”

  ELSE “IMMAGINARIO

- PROBLEMA:

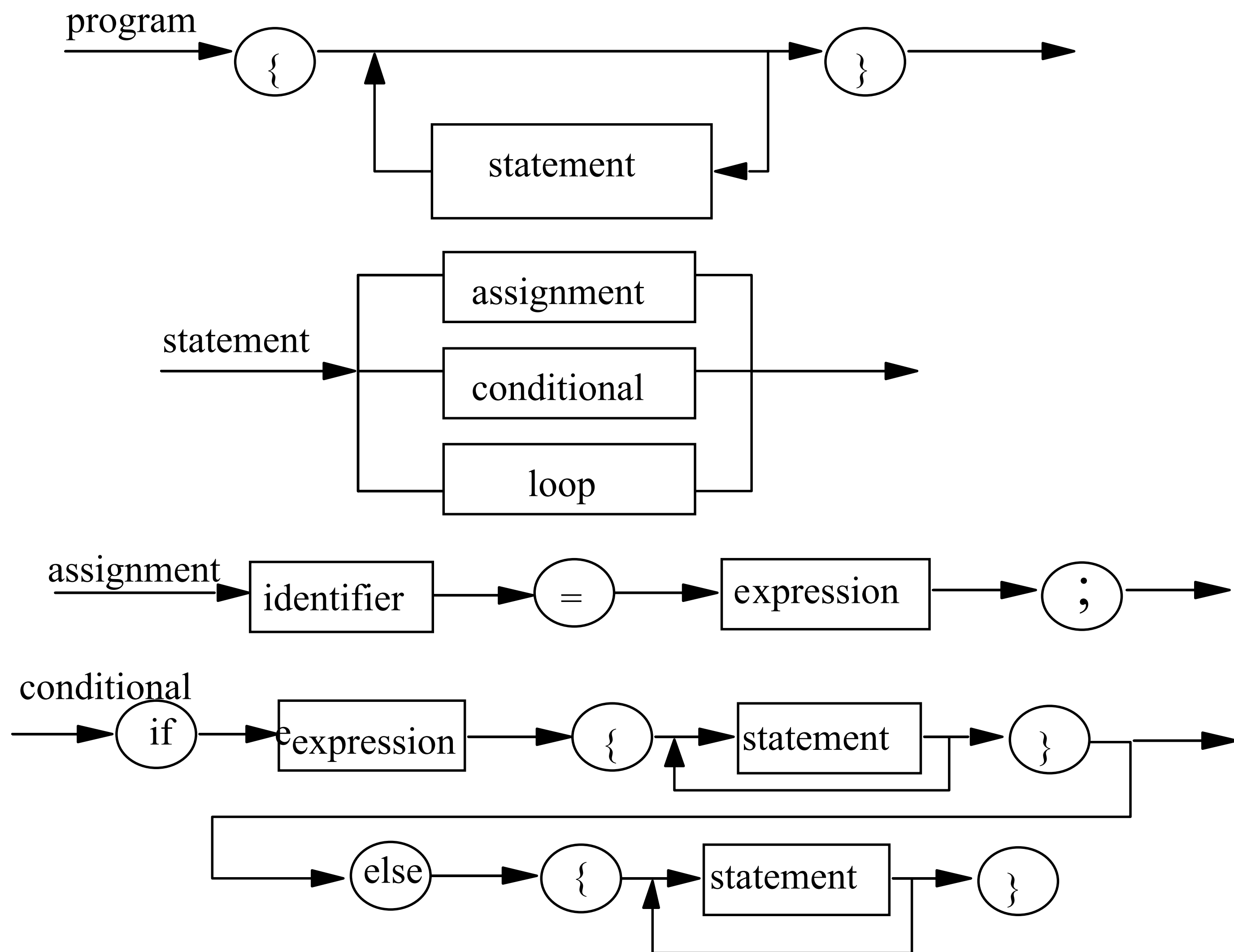
  IF non appartiene ai costrutti permessi

Soluzione

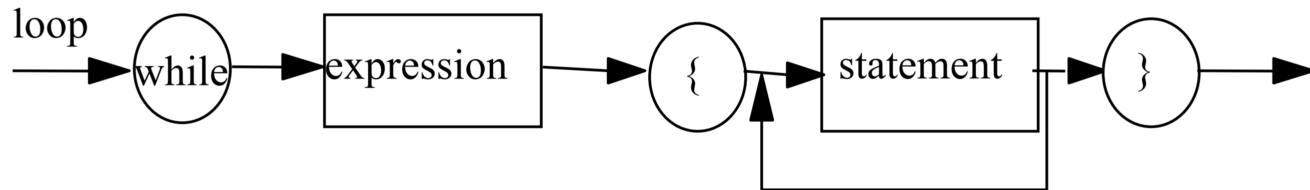
```
aux=0;
    while ((x>=0)&&(aux==0)){
          in questo caso la radice e` reale
          aux=1;
    }
    while ((x<0)&&(aux==0)) {
          in questo caso la radice e` immaginaria
          aux=1;
    }
```

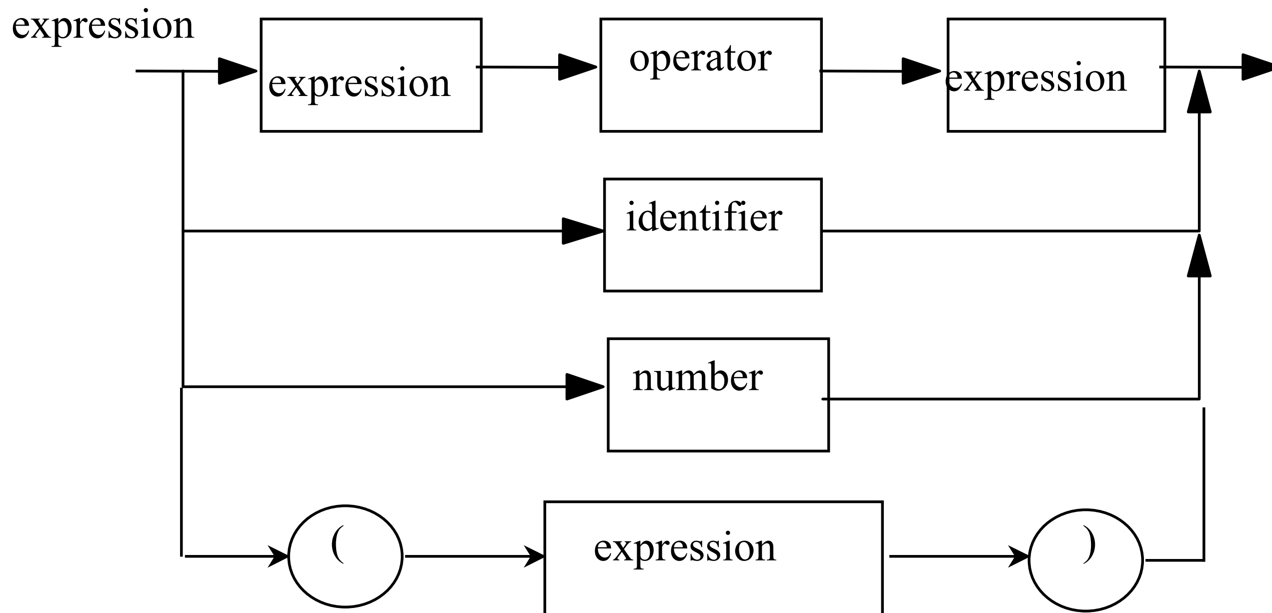
NOTA: Teorema di Jacopini-Böhm ...

Diagrammi sintattici



Diagrammi Sintattici





Sintassi astratta

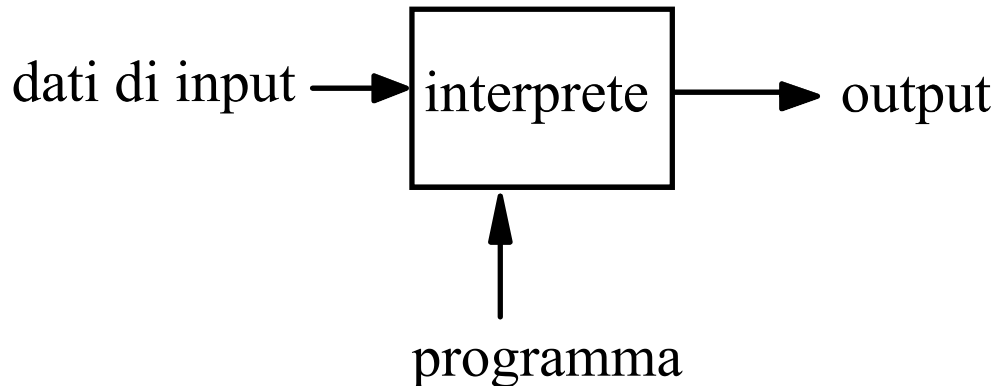
```
while (x != y) {
        . . .
};
```

```
while x ≠ y do
begin
        . . .
end
```

Pascal

C e Pascal hanno la stessa sintassi astratta,
Ma diversa sintassi concreta
(parole e modi di combinarle diverse)

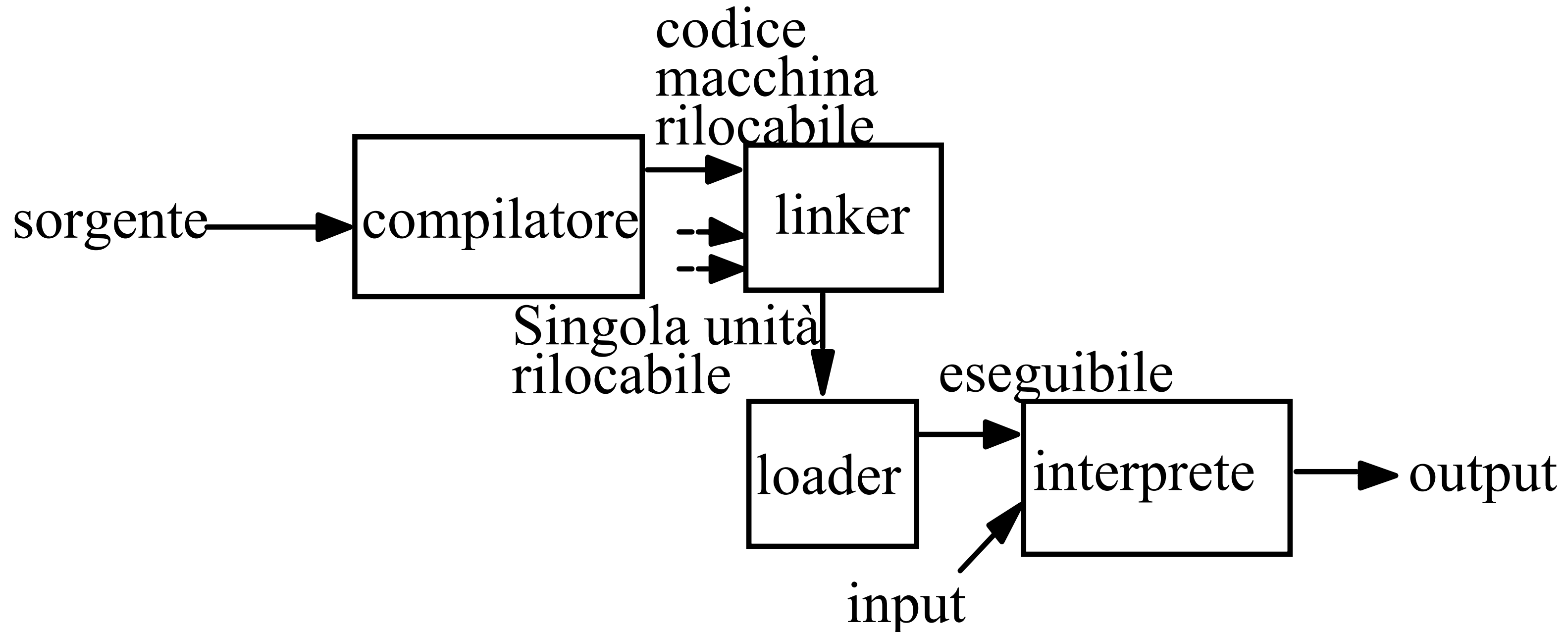
Interpretazione



L'interprete

- Prende un'istruzione
- Determina le azioni da eseguire
- Esegue le azioni

Traduzione



- Il compilatore traduce in codice macchina rilocabile (non fa riferimento a locazioni di memoria)
- Il linker a partire dai singli moduli ne produce uno unico
- Il loader risolve gli indirizzi parametrici in fisici

Il concetto di binding

Entità: variabili, routines, statements, …

Attributi

variabili: nome, tipo, locazione di memoria…

routine: nome, tipo dei parametri, tipo di passaggio

Binding descrive il legame che c'è tra entità e attributi

Descrittori memorizzano le informazioni sugli attributi

I linguaggi di programmazione differiscono in

- numero di entità
- numero di attributi da legare alle entità
- tempo di binding
  (quando un'entità viene legata a un suo attributo)

stabilità: un binding può essere modificato?

Binding statico: non può essere modificato
(altrimenti si dice dinamico)

Esempi

- FORTRAN, Ada, C, C++ sono legati a un insieme di valori alla definizione/implementazione
  - Binding statico
- Pascal
  - Binding dinamico

solitamente un binding statico

è statbilito prima dell'esecuzione

(non sempre: e.g., Pascal: read-only constant vars)

Variabili

Le variabili dei linguaggi convenzionali sono delle astrazioni della memoria

variabile = <nome, scope, tipo, l_value, r_value>

a=b

a: l_value, indirizzo della cella di memoria in cui è salvata una variabile

b: r_value, contenuto della cella

Caratterizzazione dell'astrazione:

nome della variabile: astrazione dell'indirizzo

statement di assegnamento: astrazione della modifica della cella

Nome e scope

nome (solitamente) introdotto da una dichiarazione esplicita
scope intervallo di istruzioni in cui è visibile una variabile (scope= portata). Una variabile è visibile attraverso il suo nome nel suo ambito di visibilità

<u>Il binding dello scope può essere:</u>
Statico (o lessicale): lo scope è definito dalla struttura lessicale (usato dalla maggior parte dei linguaggi)

Dinamico: lo scope è definito in termini di esecuzione del programma. L'effetto di una dichiarazione si estende a tutte le istruzioni che seguono fino a una nuova dichiarazione con lo stesso nome
(alcuni linguaggi di script: Perl, Python)

Confronto
scope statico vs scope dinamico

- Le regole di tipo dinamico sono semplici e piuttosto facili da implementare
- Svantaggi in termini di disciplina di programmazione ed efficienza di implementazione
- I programmi sono difficili da leggere

Esempio

```
main{
    int x
    {
        /* block A*/
        int x;
        ...
    }
    {
        /* block B*/
        int x;
        ...
    }
    {
        /* block C*/
        x=5;
        ...
    }
```

Dinamicamente se eseguo B poi C, la x che viene modificata in C è quella di B

Staticamente C modifica la x del main

Tipo

tipo = insieme di valori + operazioni per creare i valori, accedervi e modificarli

Proteggono le variabili da operazioni che non hanno senso

variabile = instanza del tipo

Un linguaggio può essere

- non tipizzato
- tipizzato dinamicamente
- tipizzato staticamente

Linguaggi tipizzati

- tipi built-in
- Dichiarazione di tipi → typedef int vector [10];
  - --binding fra il nome del tipo e l'implementazione (a tempo di traduzione)
  - --i nuovi tipi ereditano tutte le operazioni della struttura dati
- Tipo di dato astratto (ADT)
  - --associa nuovi tipi con operazioni da usare sulle istanze

```
typedef new_type_name {
   data structure for objects of
     type new_type_name;
   functions to manipulate data objects;
}
```

ADT in C++

```
class stack_of_char{
private:
        int size;
        char* top;
        char* s;
public:
        stack_of_char (int sz) {
                top = s = new char [size =sz];
        }
        ~stack_of_char ( ) {delete [ ] s;}
        void push (char c) {*top++ = c;}
        char pop ( ) {return *--top;}
        int length ( ) {return top - s;}
};
```

Type checking

- Verifica il corretto uso di variabili
- Linguaggi staticamente tipizzati
  - Le variabili sono fissate a un tipo prima del runtime
- Linguaggi dinamicamente tipizzati
  - Le variabili sono polimorfe
- Il type checking può essere statico
  - Per linguaggi staticamente tipizzati
  - Per certe categorie di linguaggi dinamicamente tipizzati (linguaggi OO)

l_value

Area di allocazione associata a una variabile

Lifetime (tempo di vita): periodo di tempo in cui esiste il binding tra variabile e allocazione

Allocazione della memoria:

statico vs. dinamico
automatico vs. esplicito

r_value

Valore contenuto nella zona di allocazione, interpretato secondo il tipo della variabile

Istruzioni

• accesso alle variabili attraverso l_value —*parte sinistra (left) dell'assegnamento*

• possono modificare il loro r_value —*parte destra (right) dell'assegnamento*

Binding tra variabili e valori

Solitamente dinamico

Costanti simboliche (C define, Pascal **const**) : binding statico

const float pi = 3.1416; C define

circumference= 2 * pi * radius;

Inizializzazione

Quale è il valore di una variabile quando viene creata (se non è inizializzata)?

in C int i ,j;

3 soluzioni

1. ***Ignore***: si lascia il valore che c'era prima nella locazione di memoria
2. ***System-defined initialization***: e.g., int = zero, char = blank
3. ***Undefined:*** assegna un valore "non inizializzata". A runtime bisogna controllare che l'accesso a una variabile sia fatto solo quando è definita

Routine

Unità in cui può essere decomposto un programma
Es: procedure e funzioni FORTRAN, Pascal e Ada, funzioni C
Funzioni retornano un valore; *procedure* no

```
/* somma dei primi n interi positivi*/
int sum (int n) //header {
      //body
      int i, s;
      s = 0;
      for (i = 1; i <= n ; ++i)
            s+= i;
      return s;
}
```

Routine

- Le routine hanno nome, scope, tipo, l_value, e r_value
- L'attivazione è ottenuta tramite una chiamata di routine
- Possono accedere a oggetti locali, non locali e globali
- Hanno **header (intestazione)** e **body (corpo)**

L'header defininisce il tipo della routine (**signature**)

fun: int x int -> float

La chiamata deve essere conforme al tipo della routine chiamata

l_value e r_value

- l_value locazione dove è salvato il corpo della routine
- r_value corpo correntemente legato alla routine
  - • solitamente binding statico, stabilito durante la traduzione
  - • ci sono linguaggi che supportano
    variabili di tipo routine
    variables di tipo "puntatore a routine"
```
int(*ps)(int); //ps è un puntatore a routine
ps = & sum;  //sum definita prima
int i = (*ps)(5);  // invoca la somma
```
- Routine come “oggetti di prima classe”

Rappresentazione runtime delle routine

- **codice**: istruzioni dell'unità
- **record di attivazione** (o **frame**)
    - • registra le informazioni in esecuzione (***stato***)

Ambiente di riferimento di un'istanza U

- Variabili locali di U (ambiente locale)
- Variabili non locali di U (ambiente non locale)
    - • modificabile tramite *side-effect*

Routine ricorsive

- Tutte le istanze sono composte da
  - Stesso segmento di codice
  - Diversi record di attivazione
- Binding dinamico fra un record di attivazione e il suo segmento di codice

Parametri

Supportano il flusso dell'informazione fra diversi sottoprogrammi

Distinzione fra parametri ***formali*** e ***attuali*** parametri

Corrispondenza solita tramite

- ***metodo posizionale***

Altre corrispondenze

- ***associazione per nome***

Routine generiche

```
template <class T>  void swap (T& a , T& b) {
/* la funzione non ritorna volori;
È generica rispetto al tipo T;
a e b si riferiscono alla stessa locazione come
i parametri attuali;
swap scambia I due valori*/
        T temp = a;
        a = b;
        b = temp;
}
```

Per effettuare una chiamata, il template deve essere istanziato; è implicita in C++

Overloading

```
int i, j, k;
float a, b, c;
…
i = j + k;
a = b + c;
```

+ è overloaded: addizione tra interi e addizione tra floating-point

OVERLOADING: più di un'entità legata a un nome in un dato punto e l'occorrenza del nome dà sufficienti informazione per togliere l'ambiguità sul binding

ESEMPIO   a = b + c + b( );

b denota sia una variabile che una routine

Aliasing

Contrario dell'overloading

- due nomi denostano la stessa entità in uno stesso punto
- dividono lo stesso oggetto nello stesso ambiente di riferimento
- modifico il valore di nome1, effetto visibile anche da nome2

```
int i;
 int fun (int& a);
 {  . . .
     a = a + 1;
    printf (i);
    . . .
}
 main ( )
 {  . . .
    x = fun (i);
}
```

i e a denotano lo stesso oggetto

```
int x = 0;
int* i = &x;
int* j = &x;
```

**i, *j e x sono alias*

Classificazione dei linguaggi

Linguaggi statici (FORTRAN, COBOL)

- memoria necessaria conosciuta prima dell'esecuzione
- la memoria può essere allocata prima dell'esecuzione
- non c'è ricorsione

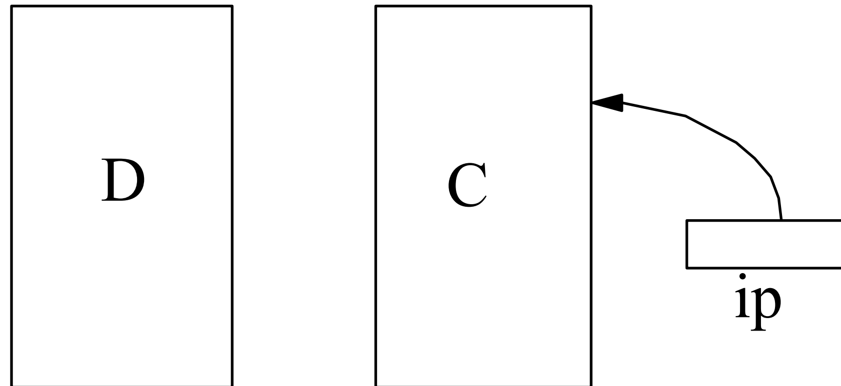
Linguaggi stack-based (ALGOL 60 e linguaggi ALGOL-like)

- memoria necessaria non conosciuta a tempo di compilazione, ma l'uso è prevedibile e segue la regola LIFO
- una politica predefinita può essere usata per allocazione/deallocazione

Linguaggi completamente dinamici

- uso della memoria non prevedibile; allocazione dinamica
- la gestione dei dati avviene come uno ***HEAP***

SIMPLESEM



ip (instruction pointer): riferimento all'istruzione corrente
2 memorie: memoria codice C, memoria dati D

Ciclo di interpretazione →

Riceve l'istruzione corrente (i.e., C[ip]);
Incrementa ip;
Esegue l'istruzione corrente

Notazione SIMPLESEM

D [X], C [X] valori memorizzati nella Xesima cella di D, C

X: l_value e D[X]: r_value

set 10, D[20]

  mette il valore salvato nella cella 20 nella cella 10

set 15, read

  il valore letto viene salvato nella cella 15

set write, D[50]

  l'output è il valore memorizzato nella cella 50

set 99, D[15]+D[33]*D[41]

  le istruzioni possono contenere espressioni complesse

Controllo di flusso

jump 47
Salto incondizionato all'istruzione salvata nella cella 47

jumpt 47, D[3] > D[8]
Il salto avviene solo se è verificata la condizione, cioè se il valore contenuto nella cella 3 è maggiore di quello contenuto nella cella 8

SIMPLESEM permette indirizzamento indiretto

```
set D[10], D[20]
jump D[13]
```

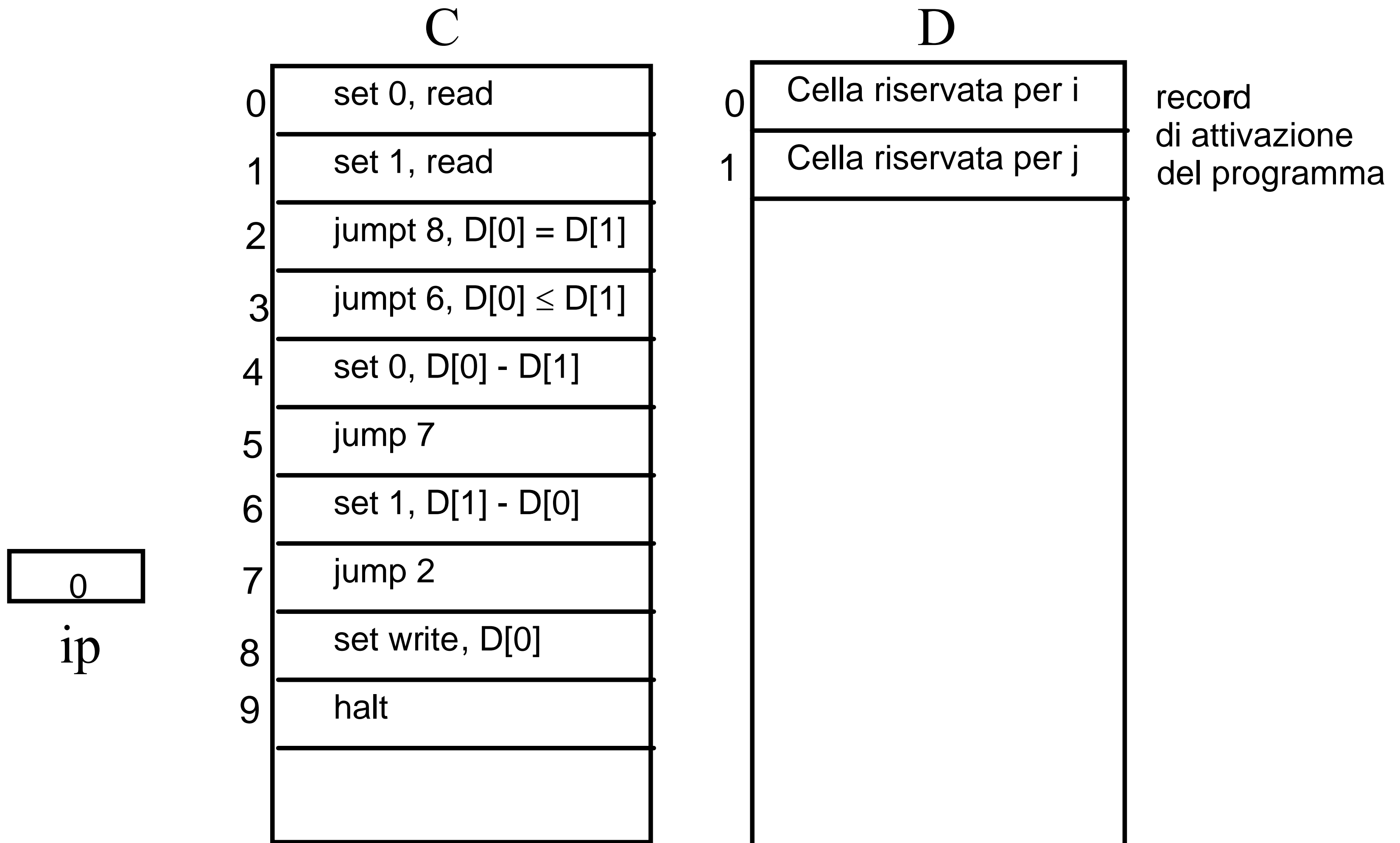
C1: linguaggio con assegnamenti semplici

Sottoinsieme del C con tipi semplici e statement (no funzioni).

Dati: interi, floating point value
array di dimensione fissa; strutture

```
main ( ) {
        int i, j;
        get (i, j);
        while (i != j)
                if (i > j)
                        i -= j;
                else
                        j -= i;
        print (i);
}
```

Stato del SIMPLESEM



Commenti

- Memoria dati: le prime due celle sono per le variabili dichiarate
- Memoria codice:
  - 0 e 1: get(I,j)
  - 2: while (se sono uguali esce)
  - 3: if
  - 4: i -=j
  - 6: ramo else j -=I
  - 8: print(i)

C2: C1 + routine

```
int i = 1, j = 2, k = 3;
alpha ( )
{
   int i = 4, l = 5;
   ...
   i+=k+l;
   ...
};
beta ( )
{
   int k = 6;
   ...
   i=j+k;
   alpha ( );
   ...
};
```

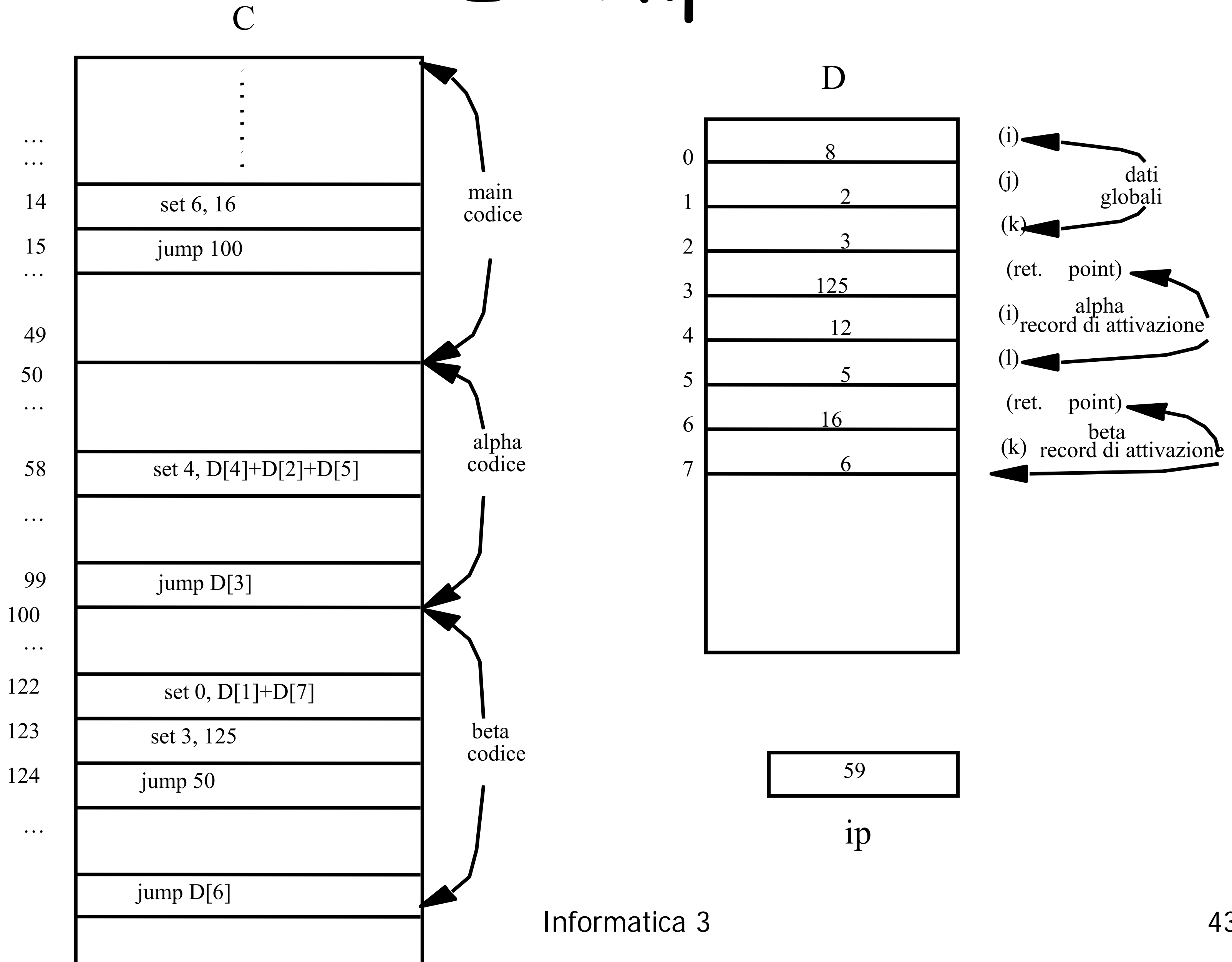
```
main ( )
{
   ...
   beta ( );
   ...
}
```

Le routine possono accedere
 ai dati locali
 ai dati globali non ridichiarati internamente
NO ricorsione, parametri, valore di ritorno

Gestione della memoria

- La dimensione del record di attivazione può essere determinata a tempo di traduzione
- I record di attivazione si possono allocare prima dell'esecuzione
- Le variabili sono legate a un indirizzo della memoria D prima dell'esecuzione
- Vantaggio: no overhead per l'allocazione
- Svantaggio: allocazione di spazio anche per routine che potrebbero non essere chiamate

Example



Compilazione separata per C2

file 1

```
int i = 1, j = 2, k = 3;
extern beta ( );
main ( )
{
    . . .
    beta ( );
    . . .
}
```

file 2

```
extern int  k;
alpha ( )
{
    . . .
}
```

file 3

```
extern int i, j;
extern alpha ( );
beta ( )
{
    . . .
    alpha ( );
    . . .
}
```

Examples

C

FORTRAN

Compilation time

le variabili locali sono legate a un offset (non un valore assoluto)

le variabili globali non possono essere legati all' offset

le chiamate di routine non possono essere legate a segmenti di codici

Link time

allocazione del codice e dei record di attivazione

tutte le informazioni che mancano vengono risolte

Ricorsione

```
int n;
 int fact ( )
{
   int loc;
    if (n > 1) {
        loc = n--;
         return loc * fact ( );
   }
   else
        return 1;
}
 main ( )
{
    getint (n);
    if (n >= 0)
        printf (fact ( ));
    else
        printf ("input error");
}
```

Assunzioni

- scope and tipizzazione statici
- dati automatici
  - dimensione del record di attivazione nota
  - offset dei dati noti
- no parametri

Conseguenze della ricorsione

Attivazioni diverse hanno
- lo stesso codice
- diverso record di attivazione

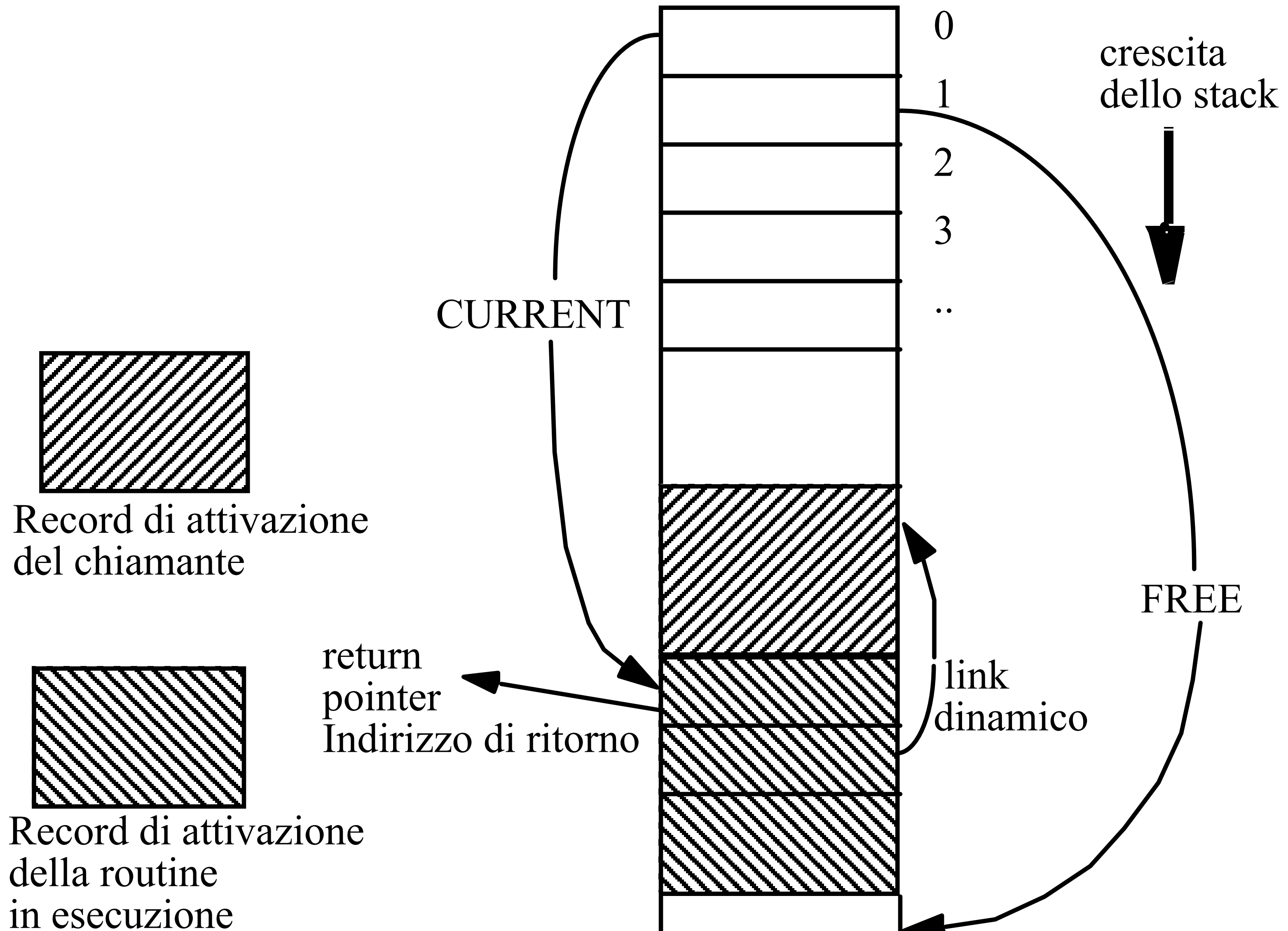
Le variabili sono legate all'offset al momento della traduzione
Il collegamento finale viene fatto a tempo di esecuzione

La cella 0 di D è usata per la base del record di attivazione corrente (CURRENT), la cella 1 per la prima cella libera (FREE)

Politica ***LIFO*** per la gestione dei record di attivazione

Le informazioni sul chiamante devono essere memorizzate
Nel record di attivazione:
- istruzione seguente da eseguire (***return pointer***)
- riferimento al record di attivazione del chiamante (***dynamic link***)

Esempio



Semantica della chiamata

set 1, D[1] + 1	creo lo spazio per l'indirizzo di ritorno
set D[1], ip + 4	l'indirizzo di ritorno è ip+4 (cioè la prima istruzione dopo quelle per inizializzare il record di attivazione)
set D[1] + 1, D[0]	la seconda cella del record di attivazione contiene l'indirizzo di ritorno del chiamante (link dinamico)
set 0, D[1]	CURRENT
set 1, D[1] + AR	FREE
jump start_addr	

Semantica del ritorno

set 1, D[0]	la prima cell libera è la base del record di attivazione della routine di cui si effettua il ritorno NOTA: la memoria non è deallocata è solo considerata libera
set 0, D[D[0] +1]	CURRENT: D[0] = indirizzo della cella che contiene l'ndirizzo di ritorno della routine che si sta deallocando D[0]+1=indirizzo del link dinamico D[D[0]+1]=base del record di attivazione del chiamante della routine da cui si effettua il ritorno
jump D[D[1]]	torno all'indirizzo di ritorno memorizzato

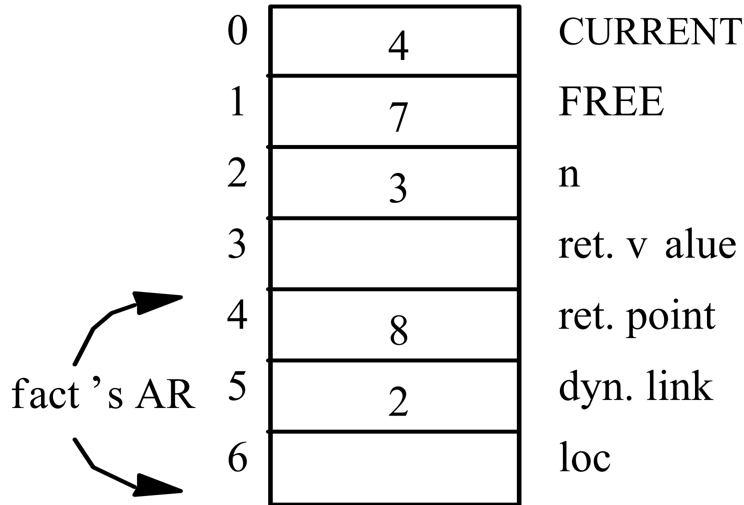
Esempio

```
0   set 2, read                 legge il valore di n
1   jumpt 10, D[2] < 0          testa il valore di n
2   set 1, D[1] + 1   chiamata a fact; spazio per i risultati salvato
3   set D[1], ip + 4            setta l'indirizzo di ritorno
4   set D[1] + 1, D[0]          setta I link dinamici
5   set 0, D[1]                 setta CURRENT
6   set 1, D[1] + 3   setta FREE (3= dimensione dell'AR di fact)
7   jump 12           12=indirizzo prima cella del codice di fact
8   set write, D[D[1] - 1]   stampa il risulatato dell chiamata
9   jump 11                     fine chiamata
10 set write, "input error"
11 halt                         fine del main
```

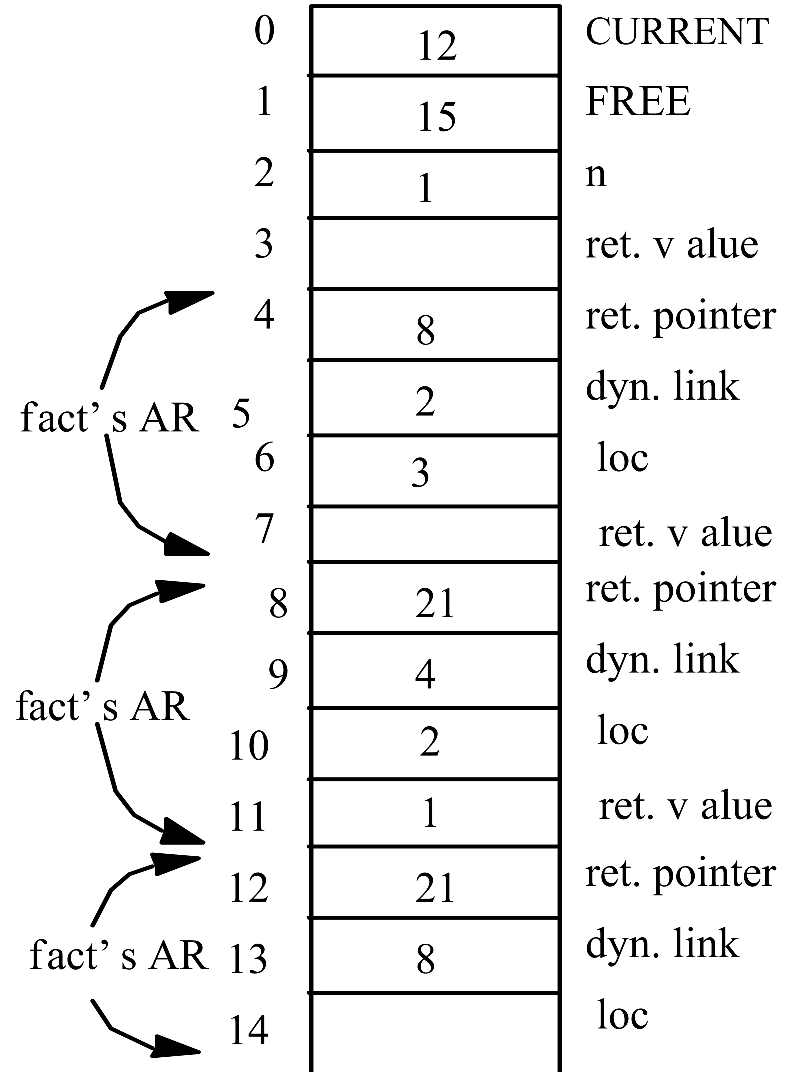
Esempio (cont.)

```
12      jumpt 23, D[2] <= 1          testa il calore di n
13      set D[0] + 2, D[2]           assegna n a loc
14      set 2, D[2] - 1              decrementa n
15      set 1, D[1] + 1   chiamata a fact, spazio per i risultati
16      set D[1], ip + 4
17      set D[1] + 1, D[0]
18      set 0, D[1]
19      set 1, D[1] + 3       3= dimensione dell'AR di fact
20      jump 12        12=indirizzo prima cella del codice di fact
21      set D[0] -1, D[D[0]+2]*D[D[1]-1]  memoriz. val. rit
22      jump 24
23      set D[0] - 1, 1
24      set 1, D[0]
25      set 0, D[D[0] + 1]
26      jump D [D[1]]
```

Snapshots della macchina



(a)



(b)

Blocchi annidati

```
int f( )
{       int x, y,w;                 //1
        while (...)
        {       int x, z;           //2
                while (. . .)
                {       int y;      //3
                }
                if (. . .)
                {       int x, w;   //4
                }
        }
        if (. . .)
        {       int a, b, c, d;     //5
        }
}
```

Soluzione

- Visibilità statica: definiamo un Activation Record (AR) che tiene conto di tutte le variabili
  - Risparmio di tempo di esecuzione
- Visibilità dinamica: la memoria viene allocata e deallocata man mano che si entra e si esce dai blocchi

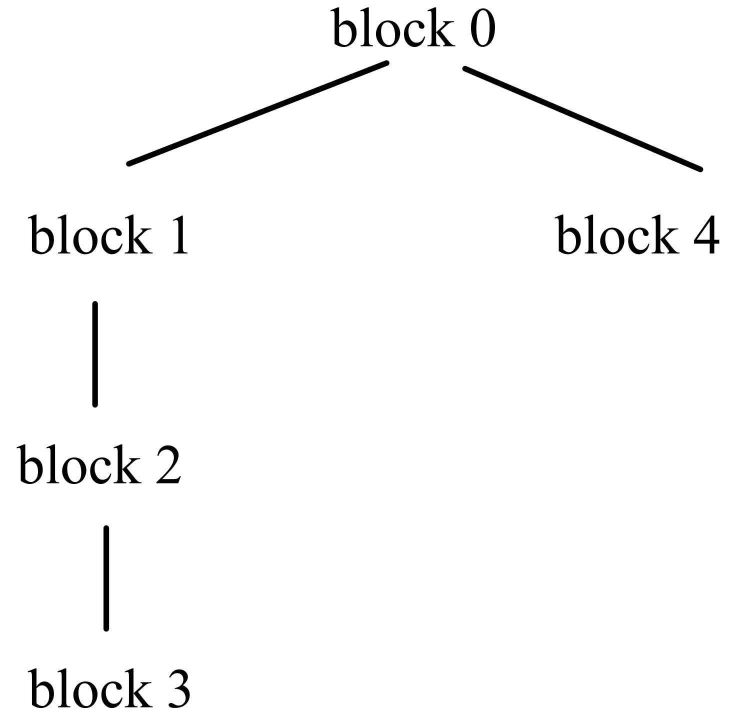
Overlayed ARs

Usiamo lo stesso insieme di celle per memorizzare variabili tra loro in mutua esclusione

return pointer
dynamic link
x in //1
y in //1
w in //1
x in //2--a in //5
z in //2--b in //5
y in //3-x in //4-c in //5
w in //4--d in //5

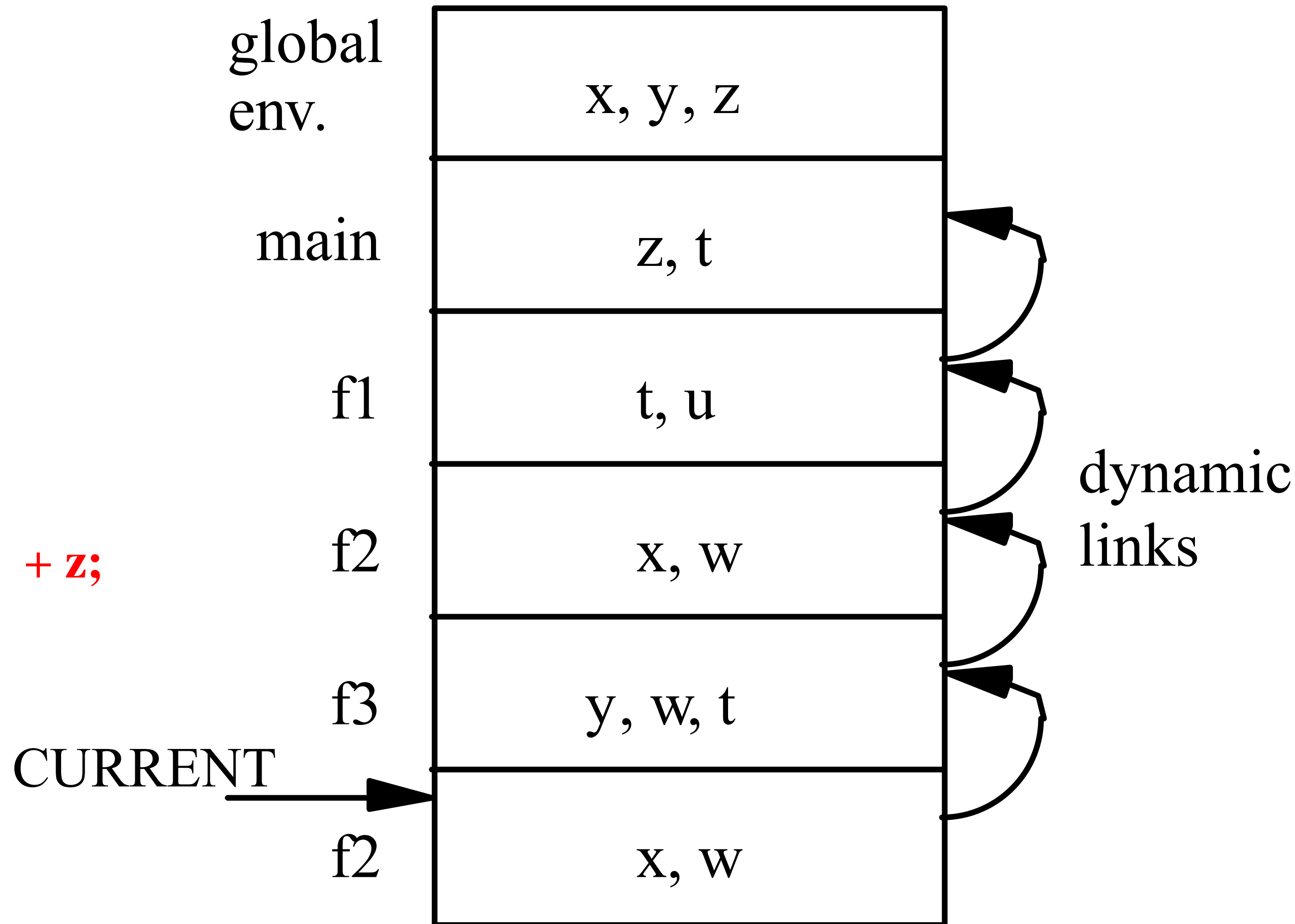
Routine nidificate

```
int x, y, z;
f1 ( )
{ //blocco 1
        int t, u;    // 1
        f2 ( )
        { //blocco 2
                int x, w;   // 2
                f3 ( )
                { //blocco 3
                        int y, w, t;  // 3
                } //fine blocco 3
                x = y + t + w  + z;
        } //fine blocco 2
} //fine blocco 1
main ( );
{ //blocco 4
        int z, t;
}  //fine blocco 4
```



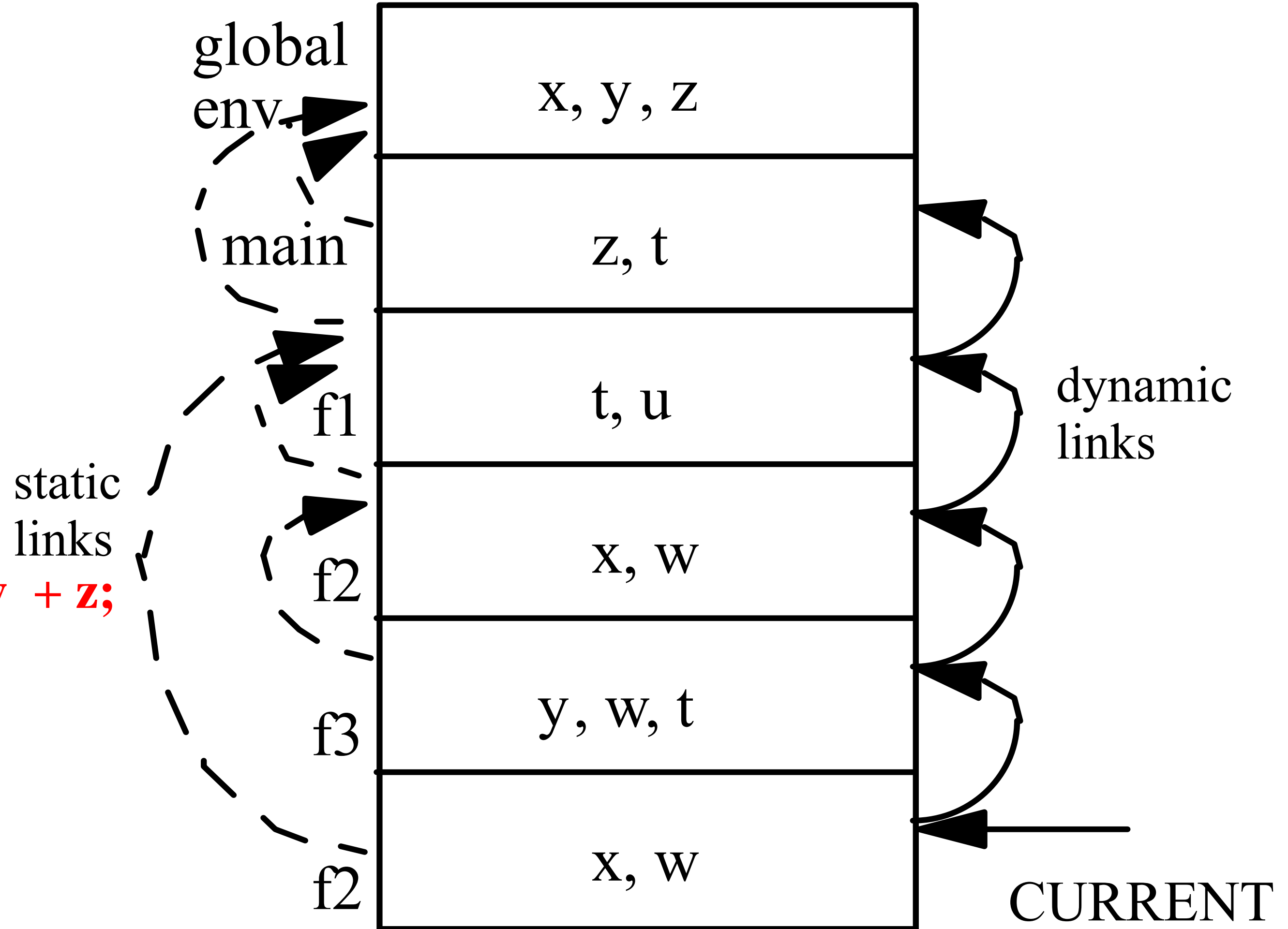
Come si accede all'ambiente non locale?

```
x, y, z
f1 ( ) {
    t, u
    f2 ( ) {
        x, w
        f3 ( ) {
            y, w, t
        }
        x = y + t + w  + z;
    }
}
main ( ) {
    z, t
}
```



Come si accede all'ambiente non locale?(cont.)

```
x, y, z
f1 ( ) {
    t, u
    f2 ( ) {
        x, w
        f3 ( ) {
            y, w, t
        }
        x = y + t + w  + z;
    }
}
main ( ) {
    z, t
}
```



Referenziare le variabili non locali

Le referenze delle variabili possono essere legate staticamente a

(d, o) ≡ (distance, offset) → ind. relativo nell' AR

→ # numero di passi che porta all'AR

Function frame pointer: **fp(d)**

puntatore all'AR che è a d passi da CURRENT

CURRENT

fp(d) = *if d=0 then* **D[0]** *else* **D [fp(d-1)+2]**

Organizzazione dell' AR:

0 ret point

1 dyn link

2 stat link

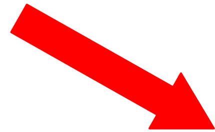
Accesso alla var: → **D[fp(d)+ o]**

Link statici

Sono salvati a runtime

basato sulla nozione di distanza a tempo di compilazione fra una chiamata di routine e la sua dichiarazione

Il chiamato è dichiarato a distanza d dal chiamante



Il link statico dovrebbe puntare all'AR che è a d passi nella catena statica originata nella chiamata

Chiamata a routine

```
set 1, D[1] + 1          setta il valore di ritorno
set D[1], ip + 5         setta il punto di ritorno
set D[1] + 1, D[0]       setta il link dinamico
set D[1] + 2, fp(d)      setta il link statico
set 0, D[1]              setta CURRENT
set 1, D[1] + AR         setta FREE
jump start_addr
```

Array dinamici (es. Ada)

type VECTOR is array (INTEGER range <>);

--definisce array con l'indice non vincolato

A: VECTOR (1..N);

B: VECTOR (1..M);

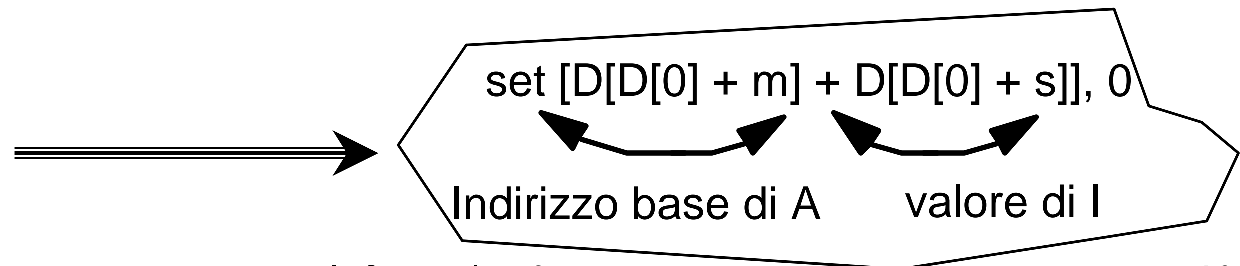
--N e M devono essere legate a qualche valore intero

--dichiarazione elaborata a runtime

In SIMPLESEM:

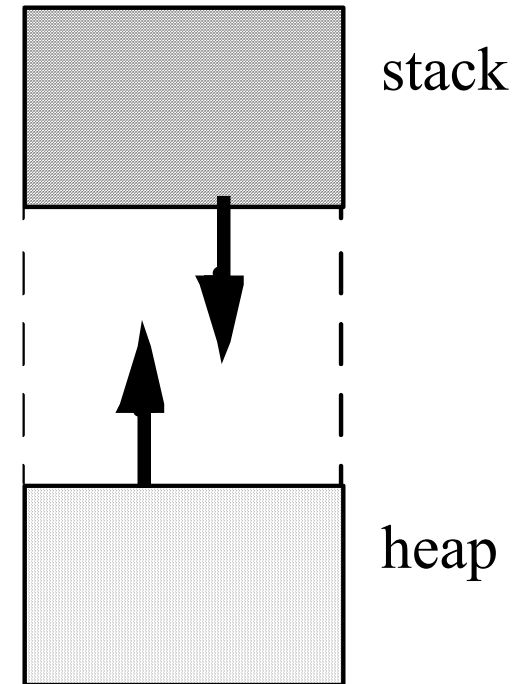
- un puntatore riservato per ogni array dinamico
- gli oggetti array sono allocati in cima agli ultimi AR allocati
- l'accesso all'array avviene indirettamente mediante puntatore

A[I] = 0
(I è a offset s
A è a offset m)



Puntatori

```
struct node {
        int info;
        node* left;
        node* right:
};
node* n = new node;
```



I nodi non possono essere allocati nello stack

--- Sono allocati nello heap

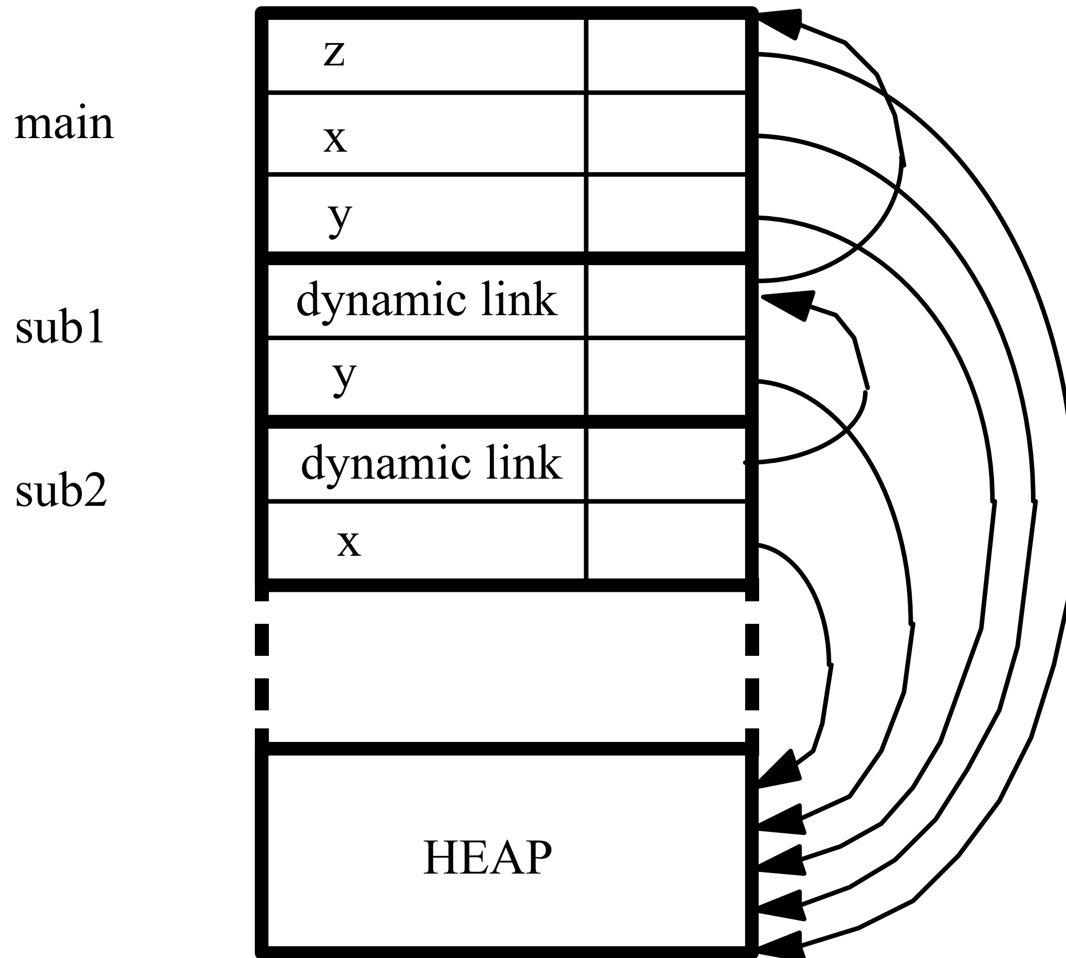
Dynamic typing e scoping

```
sub2 ( )
{
   declare x;
   . . . x  . . .;
   . . . y . . .;
   . . .
}

sub1 ( )
{
   declare y;
   . . . x  . . .;
   . . . y . . .;
   sub2 ( );
   . . .
}

main ( )
{
   declare  x, y, z;
   z = 0;
   x = 5;
   y = 7;
   sub1;
   sub2;
   . . .
}
```

dynamic typing: una variabile nell'AR è rappresentata da un puntatore all'oggetto nello heap (la dimensione può cambiare dinamicamente)

dynamic scoping: il vincolo dinamico supporta l'accesso a oggetti non locali

Vista a runtime



Esercizio

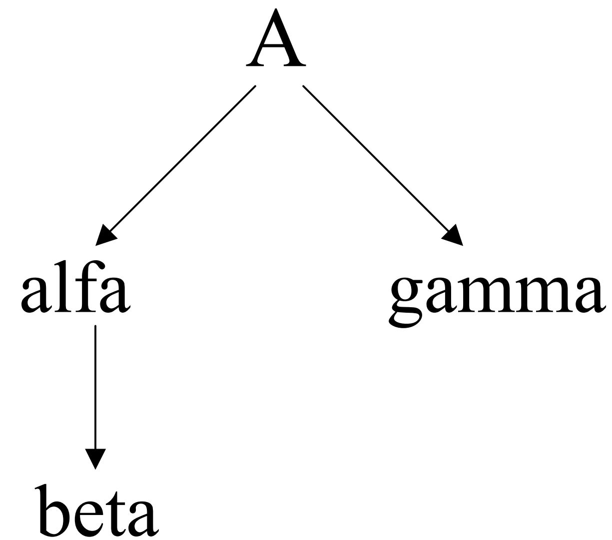
```
1.program A{
2.  integer b,y;
3.  routine alfa() {
4.          integer a,x,w;
5.          routine beta () {
6.                  integer z,g;
7.                  a=z+x+b+w;
8.          };.......
9.          x=a+y; .......
10.         };
11.         routine gamma() {
12.                 integer a,x,z,w; .......
13.          };
14.}
```

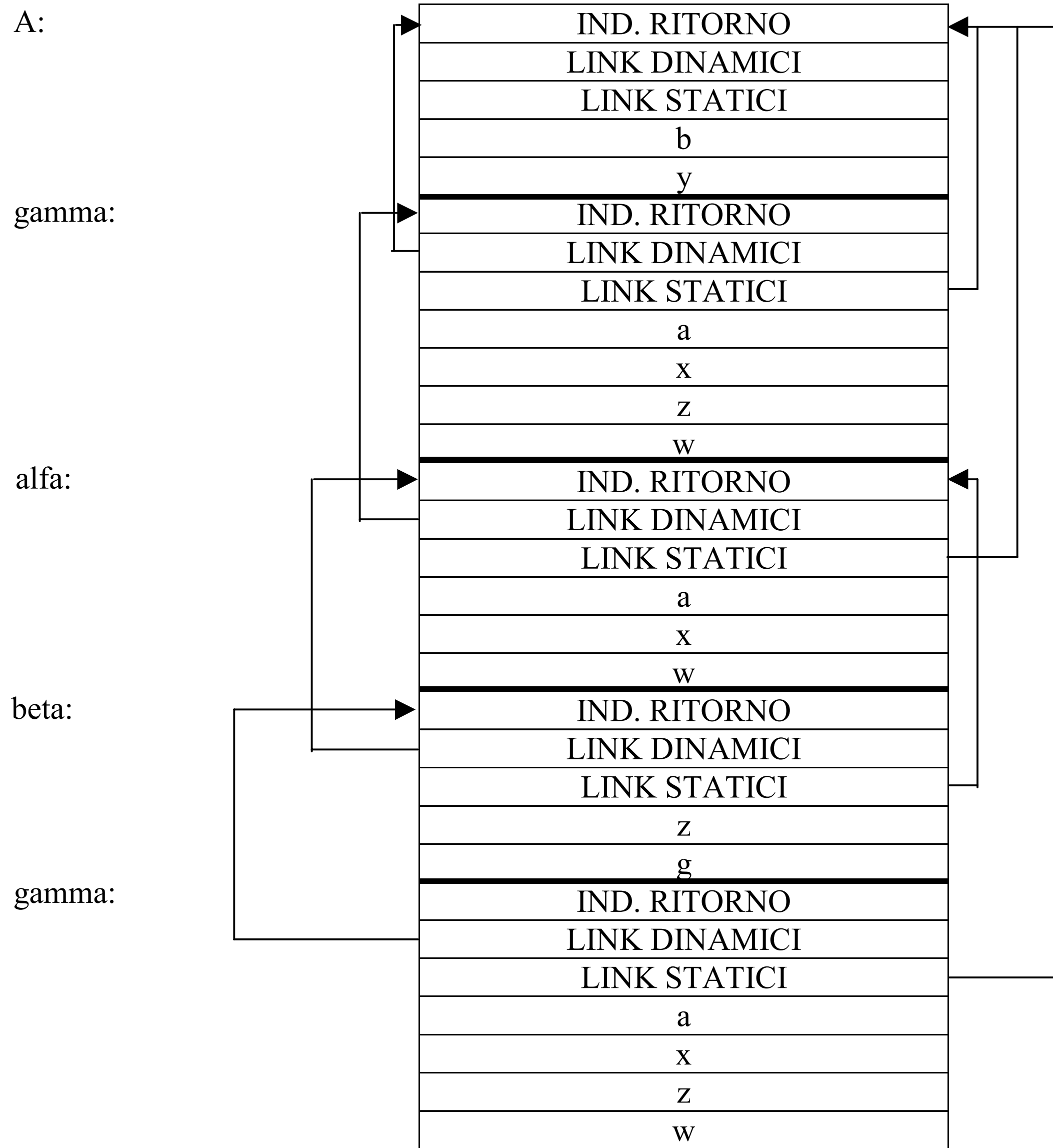
Considerata la seguente catena di chiamate:

A→gamma →alfa →beta →gamma

- schizzare lo stato della macchina astratta
  - link statici
  - link dinamici

- Annidamento statico dei moduli




A:
IND. RITORNO
LINK DINAMICI
LINK STATICI
b
y
gamma:
IND. RITORNO
LINK DINAMICI
LINK STATICI
a
x
z
w
alfa:
IND. RITORNO
LINK DINAMICI
LINK STATICI
a
x
w
beta:
IND. RITORNO
LINK DINAMICI
LINK STATICI
z
g
gamma:
IND. RITORNO
LINK DINAMICI
LINK STATICI
a
x
z
w

Variabili: <distanza, offset>

- Linea 7: a=z+x+b+w;

  a=<1,3>

  z=<0,3>

  x=<1,4>

  b=<2,3>

  w=<1,5>
- Linea 9: x=a+y;

  x=<0,4>

  a=<0,3>

  y=<1,4>

Regole di scope statico

Variabili: <distanza, offset>

- Linea 7: a=z+x+b+w;

  a=<1,3>

  z=<0,3>

  x=<1,4>

  b=<3,3>

  w=<1,5>

- Linea 9: x=a+y;

  x=<0,4>

  a=<0,3>

  y=<2,4>

Regole di scope dinamico

Passaggio dei parametri

- Dati
  - Per riferimento
  - Per copia
  - Per nome
- Routine
  - Casi di linguaggi con scope statici e dinamici

Passaggio per riferimento

Anche: per condivisione

Il chiamante passa l'indirizzo del parametro attuale

Il riferimento al parametro formale è trattato come un Riferimento indiretto

Cosa accade se il parametro attuale è un'espressione o una costante?

Passaggio per riferimento(cont.)

Il chiamante

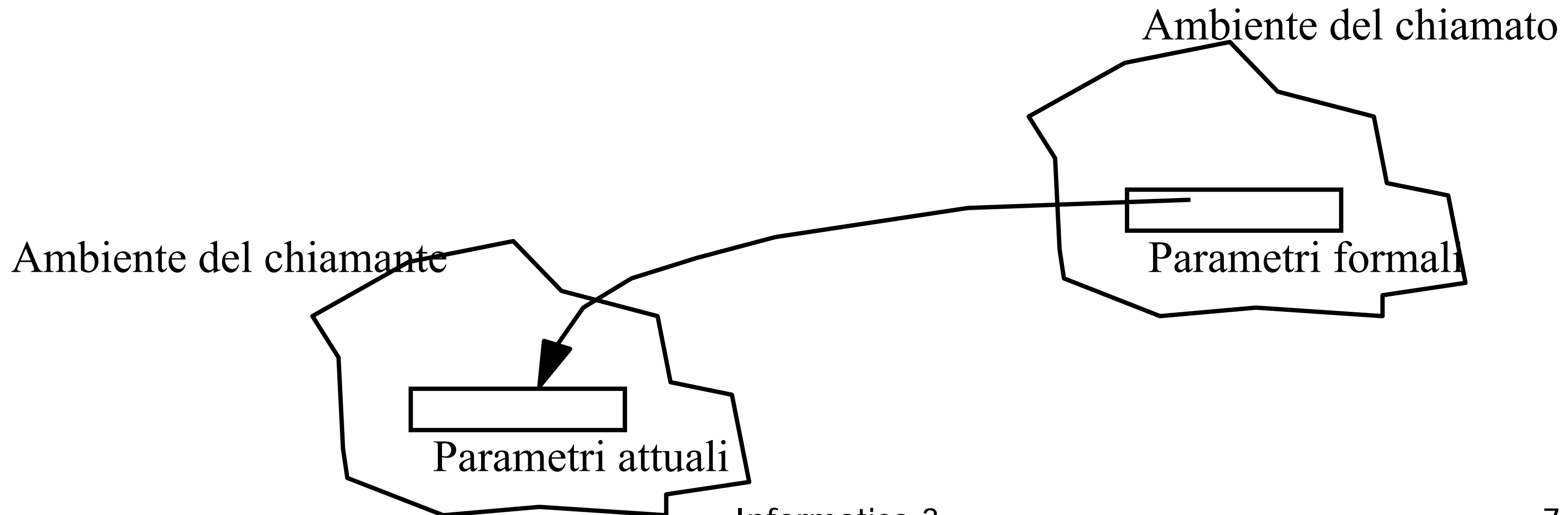
set D[0] + off, fp(d) + o

Se il parametro attuale è per riferimento:

set D[0] + off, D[fp(d) + o]

Accesso alla variabile tramite indirizzamento indiretto

x = 0; → set D[D[0] + off], 0



Passaggio per copia

Non c'è condivisione: I parametri sono variabili locali

Per valore

- il valore del parametro attuale viene copiato nel parametro formale. (la copia avviene solo dal chiamante al chiamato)

Per risultato

- il valore del parametro formale viene copiato nel parametro attuale. (la copia avviene solo dal chiamato al chiamante)

Per valore risultato

- Combinaziobe dei due precedenti

Valore-Risultato vs Riferimento

```
i = j;
a [i] = 10;
foo (a[i], a[j]);
. . .

a = 10;
. . .
goo (a);
```

```
foo (x, y) {
      . . .
      x = 0;
      y ++;
}

goo (x) {
   . . .
   a = 1;  //a is nonlocal and visible
   x = x + a;
}
```

Effetti differenti nei seguenti casi:

- due parametri formali diventano alias
- un parametro formale e una variabile non locale sono alias

Passaggio per nome

Definito dalla sostituzione del nome
I parametri formali denotano una locazione nell'ambiente del Chiamante.
Legame con uno specifico l-value dinamicamente, ogni volta è usato

```
void swap (int a, b) {
int temp;
        temp = a;
        a = b;
        b = temp;
};      swap (i,a[i])
```

se i = 3, a [3] = 4 prima, poi avrò i=4 e a[4]=3 invece di a[3] =3

Un altro problema

```
int c;
. . .
void swap (int a, b) {
int  temp;
        temp = a; a = b;
        b = temp; c ++
}

y ( );
int c, d;
{
        swap (c, d);
};
```

Valutazione

- FORTRAN: per riferimento
- ALGOL 60: per nome (è possibile anche per valore)
- SIMULA 67: per valore, per riferimento e per nome
- Pascal and Modula-2: per valore e per riferimento
- C: per valore, per riferimento coi puntatori
- Ada: il passaggio dei parametri è basato sulla loro funzione
  in (per gli input), out (per gli output)
  inout (per input/output) (in è il default)

Esercizio

```
1.  program esempio;
2.  var x:integer
3.          procedure p(y:integer)
4.          begin
5.                  x:=5;
6.                  y=y+x;
7.          end;
8.  begin
9.          x:=10;
10.         p(x);
11.                 write(x);
12. end
```

Analizzare il valore stampato a seconda della tipologia di passaggio di parametri considerata

By reference

(parametri formali e attuali dividono la stessa area di memoria)

# Riga	x	y
9	10	
3	10	10
5	5	5
6	10	10
11	10	

By value(par. attuale copiato in quello formale che sostituisce la var. locale nella procedura)

# Riga	x	y
9	10	
3	10	10
5	5	10
6	5	15
11	5	

By result

(par. formale copiato in quello attuale al termine della procedura)

# Riga	x	y
9	10	
3	10	0
5	5	0
6	5	5
11	5	

By value/result

# Riga	x	y
9	10	
3	10	10
5	5	10
6	5	15
11	15	

By name

(nel testo della procedura i par. formali sostituiscono quelli attuali)

# Riga	x	y
9	10	
3	10	10
5	5	5
6	10	10
11	10	

Routine come parametri

```
1         int u, v;
2         a ( )
3         {
4                 int y;
5                 . . .
6         };
7         b (routine x)
8         {
9                 int u, v, y;
10                c ( )
11                {...
12                y = . . .;
13                        . . .
14                };
15                x ( );
16                b (c);
17                . . .
18        }
19        main ( )
20        {
21                b(a);
22        };
```

Routine come parametri (cont.)

Informazioni da passare al chiamato:

- riferimento al codice della routine
- ambiente non locale della routine (static link: SL)

2 casi:

(a) il parametro attuale è nello scope del chiamante
(b) era un parametro formale che era passato al chiamante

Caso (a): Il link statico è un puntatore all'AR che dista d passi nella catena statica originata dal chiamante

Caso (b): Il link statico è quello che era stato passato al chiamante

Esercizio

```
program pippo
var a,b,c:integer
    procedure p(procedure x)
    var a,b,d:integer;
           procedure q
           var b,e:integer;
           ...............
           end q;
    if c=0 then
           begin
                      c:=c+1;
                      x(q);
           end
    else x;
    end p;
 ...............
c:=0;
p(p);
end;
```

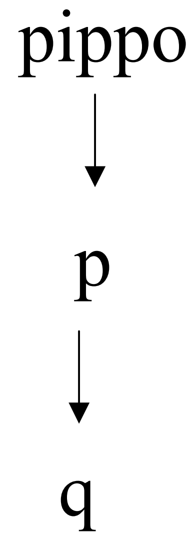
schizzare lo stato della macchina astratta, tracciando link statici e dinamici, fino a quando q viene chiamata per la prima volta.

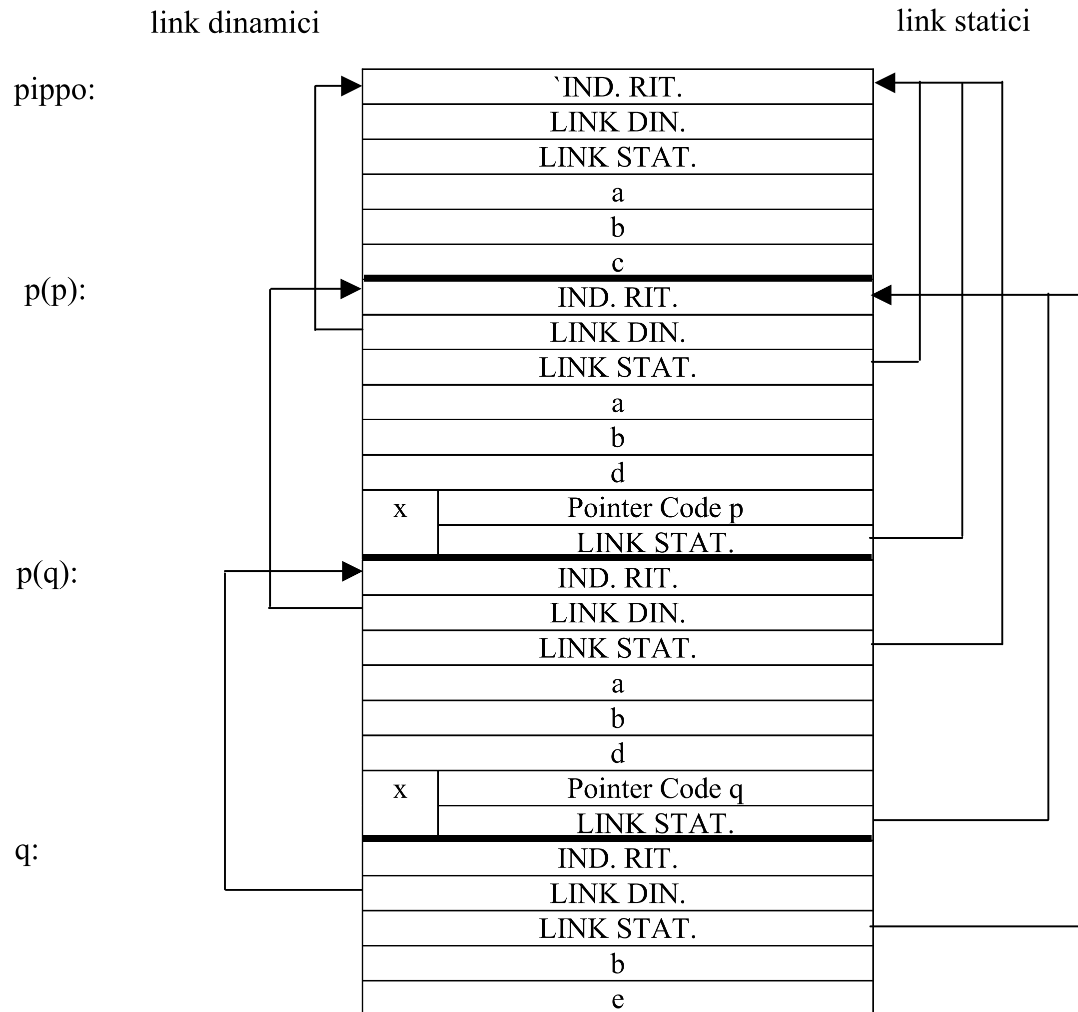
Catena chiamate

- Pippo pone c=0 e chiama p(p)
- In p(p) si entra nel ramo then dell'if ponendo quindi c=1 e chiamando p(q)
- In p(q) si entra nel ramo else dell'if quindi si chiama q
- Quindi la sequenza delle chiamate e':

pippo → p(p) → p(q) → q

Annidamento statico




link dinamici
link statici
pippo:
`IND. RIT.
LINK DIN.
LINK STAT.
a
b
c
p(p):
IND. RIT.
LINK DIN.
LINK STAT.
a
b
d
x
Pointer Code p
LINK STAT.
p(q):
IND. RIT.
LINK DIN.
LINK STAT.
a
b
d
x
Pointer Code q
LINK STAT.
q:
IND. RIT.
LINK DIN.
LINK STAT.
b
e

Esercizio

```
program pippo
var l,m,n:real;
   procedure q(procedure w)
        procedure p
        var r,s:real;
        ....................
        end
   if n>0 then w
   else begin
        n:=n+1.77;
        w(p);
   end
.......................
n:= -0.5;
q(q);
end
```

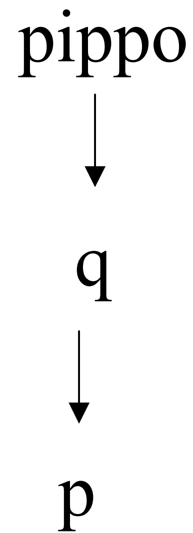
schizzare lo stato della macchina astratta, tracciando link statici e dinamici, fino alla chiamata di w effettuata nel ramo then della procedura q. Si scriva inoltre come viene tradotto l’accesso alla variabile n in termini della coppia (distanza offset) [scope statico].

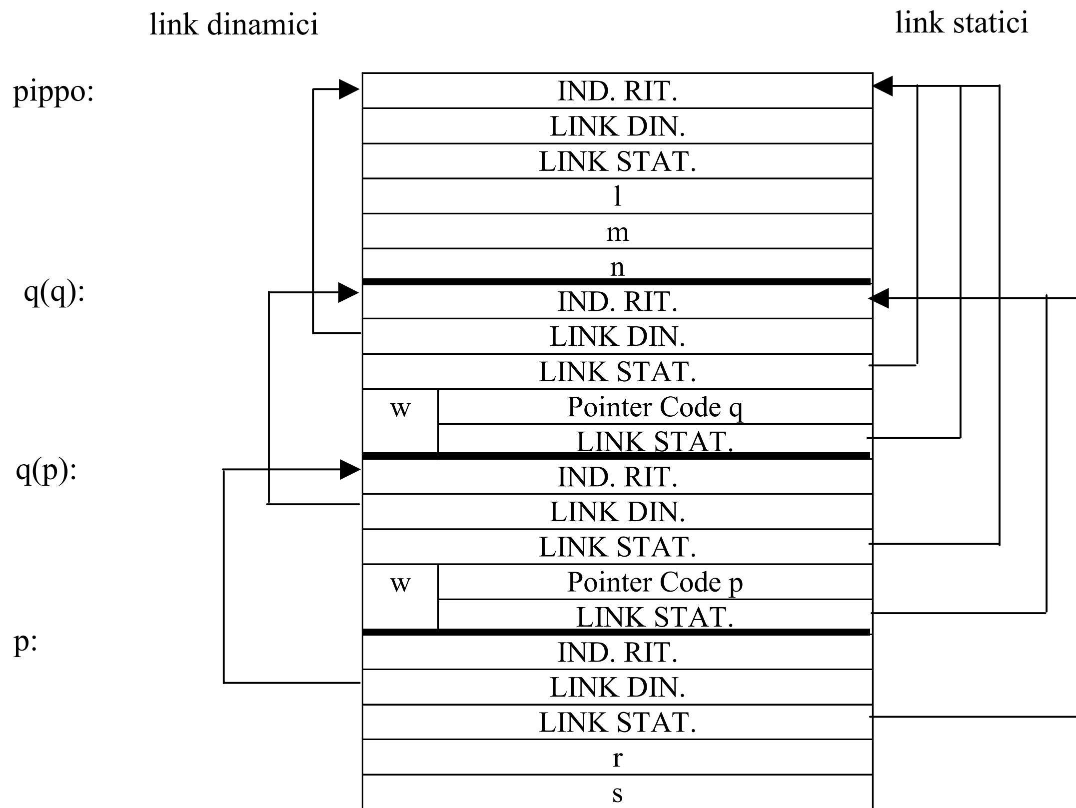
Sequenza chiamate

- Pippo pone n=-0.5 e chiama q(q)
- In q(q) si entra nel ramo else dell'if ponendo quindi n=-0.5+1.77=1.27 e chiamando q(p)
- In q(p) si entra nel ramo then dell'if quindi viene chiamata p
- Quindi la sequenza delle chiamate e':

pippo → q(q) → q(p) → p

Annidamento statico





link dinamici
link statici
pippo:
IND. RIT.
LINK DIN.
LINK STAT.
l
m
n
q(q):
IND. RIT.
LINK DIN.
LINK STAT.
w
Pointer Code q
LINK STAT.
q(p):
IND. RIT.
LINK DIN.
LINK STAT.
w
Pointer Code p
LINK STAT.
p:
IND. RIT.
LINK DIN.
LINK STAT.
r
s

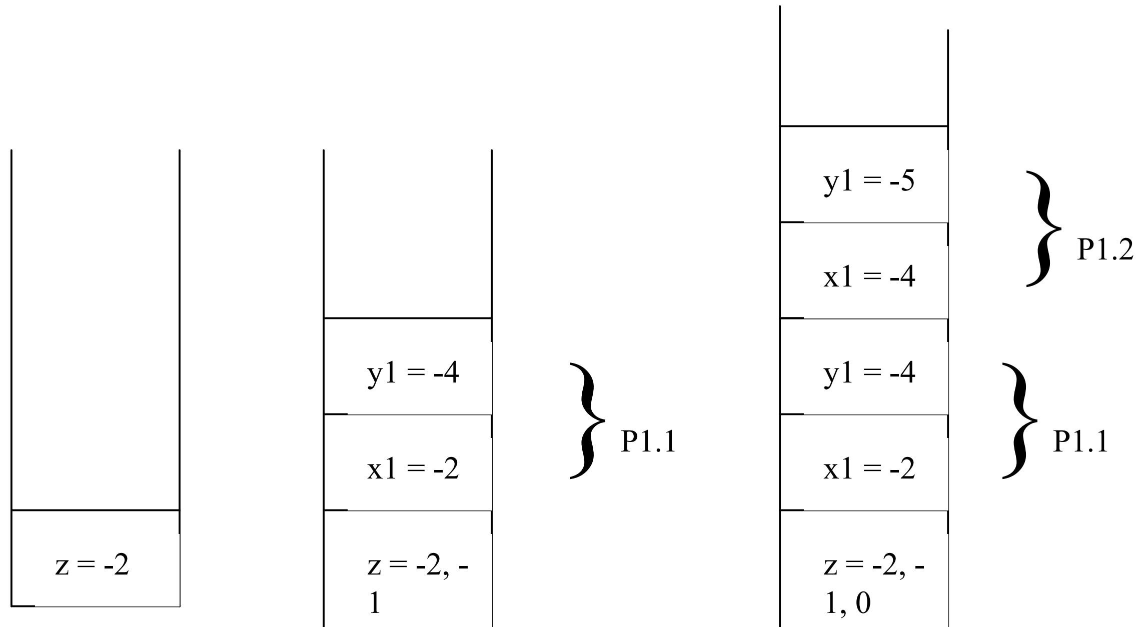
Accesso alla variabile n

- Accesso alla variabile n da pippo: n=<0,5>
- Accesso alla variabile n per la prima chiamata di q: n=<1,5>
- Accesso alla variabile n per la seconda chiamata di q: n=<1,5> (valgono regole di scope statico)

Esercizio: Simplesem+ Passaggio parametri

```
main ()
{  int z
   void P1 (int x1)
   {     int y1
         void P2 (int x2)
         {        int y2
                  y2 = y1; x2 = y1 + 1; z = y2 + 1
         };
         y1 = x1 + z; z = z + 1;
         if z < 0 P1(y1);
         P2(y1);
         z = z + y1
   }
z = -2; P1(z); printf (z)
}
```

Esecuzione mediante passaggio per valore_risultato, regola della catena statica



y2 = -5

x2 = -5, -4

y1 = -5

x1 = -4

y1 = -4

x1 = -2

z = -2, -1, 0, -4

P2.1

P1.2

P1.1

y1 = -5, -4

x1 = -4

y1 = -4

x1 = -2

z = -2, -1, 0, -4, -8

P1.2

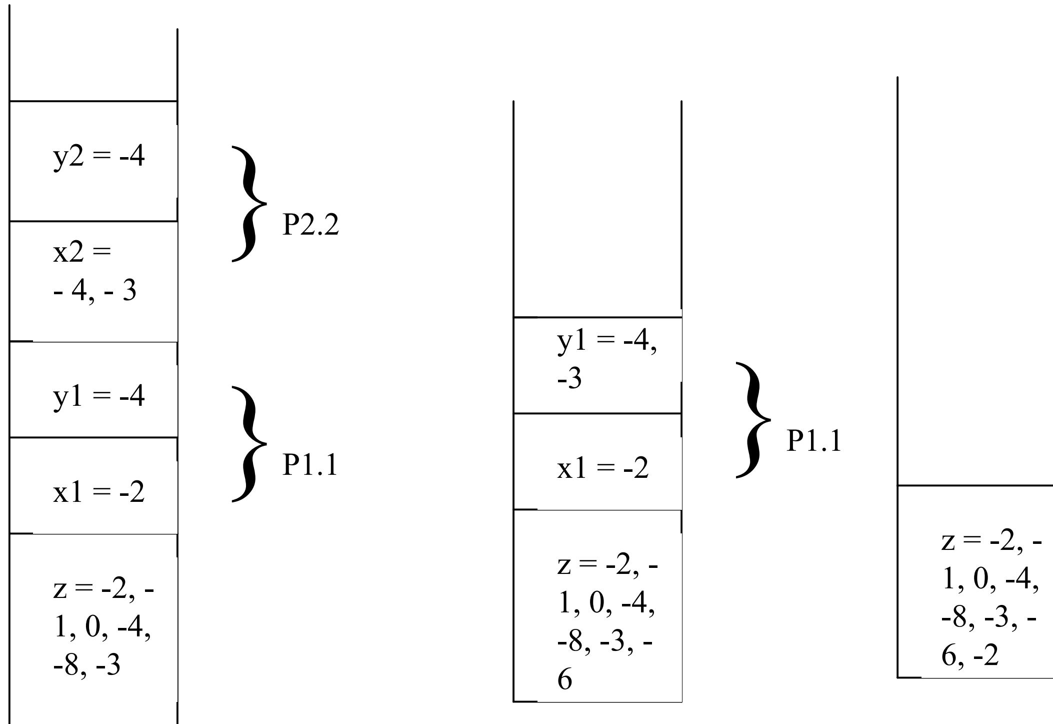
P1.1

y1 = -4

x1 = -2

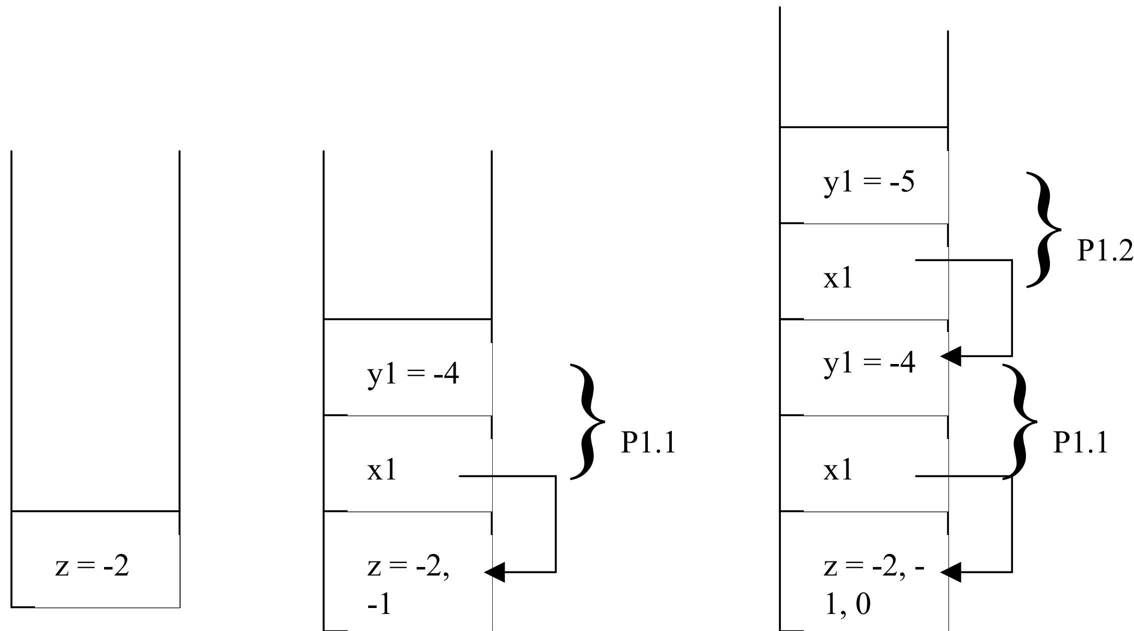
z = -2, -1, 0, -4, -8

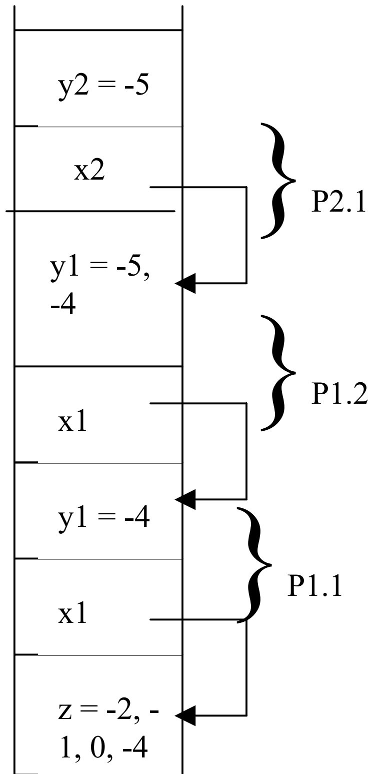
P1.1

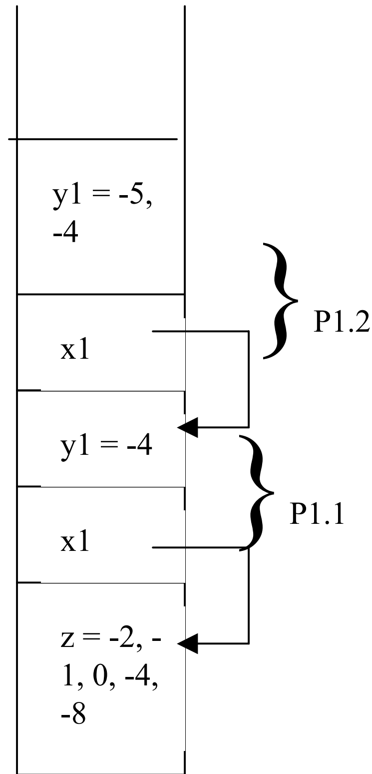


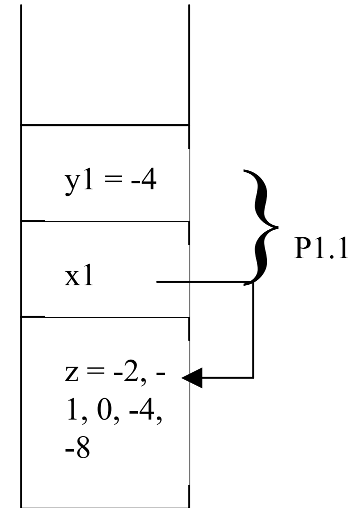
Print: z = -2

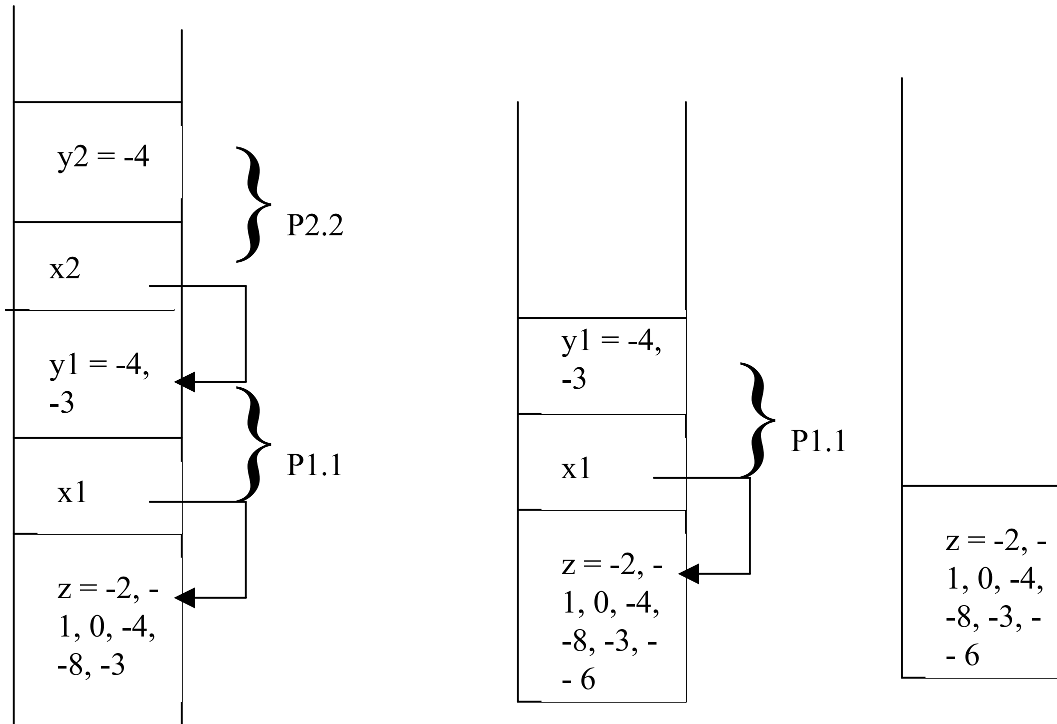
Esecuzione mediante passaggio per indirizzo, regola della catena statica




y2 = -5
x2
P2.1
y1 = -5,
-4
x1
P1.2
y1 = -4
P1.1
x1
z = -2, -
1, 0, -4


y1 = -5,
-4
P1.2
x1
y1 = -4
P1.1
x1
z = -2, -
1, 0, -4,
-8


y1 = -4
P1.1
x1
z = -2, -
1, 0, -4,
-8



Print: z = -6

Esecuzione mediante regola della catena dinamica

- Nulla cambia usando la regola della catena dinamica in luogo di quella statica.
  - L'ambiente non locale di P1 è costituito dalla sola variabile globale z;
  - quello di P2 è costituito ancora da z e dall'istanza di y1 comuqnue immediatamente sottostante nella stack.