Linguaggi di Programmazione I – Lezione 2

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
http://sette.dnsalias.org

11 marzo 2010

Panoramica della lezione

Modello imperativo

Memoria

- Consiste in un insieme di "contenitori di dati" ...
  - Ad es. parole (o celle) della memoria centrale
  - Tipicamente rappresentate dal loro indirizzo
- ... associati ai valori in essi contenuti
  - I valori delle variabili
- Dunque (concettualmente) la memoria è
  - Una funzione da uno spazio di locazioni ad uno spazio di valori
  - mem(loc) = "valore contenuto in loc"

Assegnazioni

- Definizione grammaticale (sintassi):

```
<assegnazione> ::= <name> <assignment-operator> <expression>
```

  <name> rappresenta la locazione dove viene posto il risultato mentre in <expression> sono specificati una computazione e i riferimenti ai valori necessari alla computazione.

  Esempio in Pascal:

```
a := b + c;
```

- Esecuzione (significato, semantica):

  Il valore di <expression> va memorizzato nell'indirizzo rappresentato da <name>.

  Il valore di <expression> dipenderà dai valori contenuti negli indirizzi degli argomenti di <expression> rappresentati dai nomi di questi ottenuto seguendo le prescrizioni del codice associato al suo nome.

Modello imperativo

- Simula le azioni dell'elaboratore a livello di linguaggio macchina.
- I programmi sono descrizioni di sequenze di modifiche della "memoria" del calcolatore.
- Ogni unità di esecuzione consiste di 4 passi:
  1. ottenere indirizzi delle locazioni di operandi e risultato;
  2. ottenere dati di operandi da locazioni di operandi;
  3. valutare risultato;
  4. memorizzare risultato in locazione risultato.
- Si caratterizza per l'uso dei nomi come astrazione di indirizzi di locazioni di memoria.

Nomi (di variabili o di parametri di procedure)

- Le variabili di un programma costituiscono una memoria?
- E i parametri formali delle funzioni/procedure? Non proprio (sia concettualmente che per come sono realizzati)

Ambiente (di esecuzione)

- Insieme di nomi di variabili e parametri ...
  - Non indirizzi di memoria, piuttosto identificatori
- ... associati a qualcosa da cui si può risalire al valore della variabile o del parametro.
- Concettualmente l'ambiente è: una funzione da un insieme di identificatori (i nomi) a un insieme di ...?

  env(id) = ???
- Il codominio della funzione dipende dal paradigma computazionale del linguaggio.

Esempi di ambiente

- Nel paradigma imperativo, la funzione env associa gli identificatori a locazioni di memoria, le quali, a loro volta, sono associate (funzione mem) al contenuto di memoria

  Il valore di una variabile x è mem(env(x))
- Nel paradigma funzionale, non esiste la funzione mem e la funzione env associa direttamente gli identificatori al contenuto della memoria.
- Attenzione:

  nel paradigma imperativo la funzione env identifica una associazione *immutabile* (nomi costantemente associati a locazioni di memoria); nel paradigma funzionale, ovviamente, non è così.

Esempio: assegnazione

In una assegnazione:

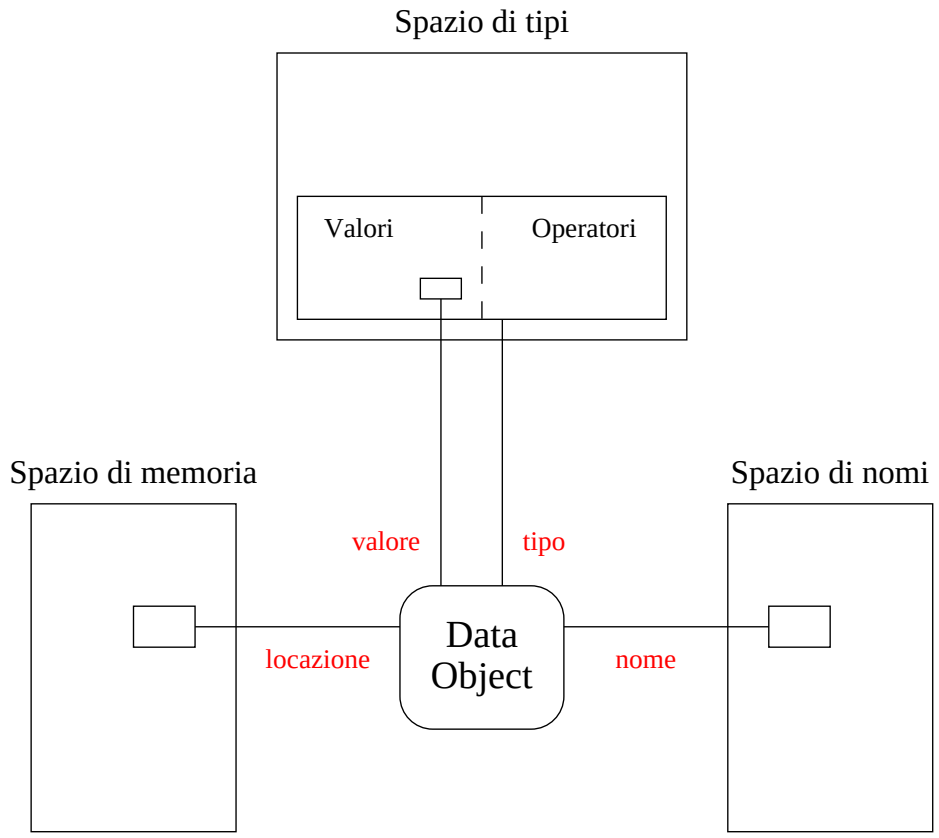
```
x := x + 1;
```

la x di sinistra indica la locazione associata al nome (cioè env(x))
la x di destra indica il valore della variabile (cioè mem(env(x))

Data Object

- Un *data object* è la quadrupla (L, N, V, T), ove:
  - L: locazione.
  - N: nome.
  - V: valore.
  - T: tipo.
- Un *legame* è la determinazione di una delle componenti.

Legami



Modifiche di legami

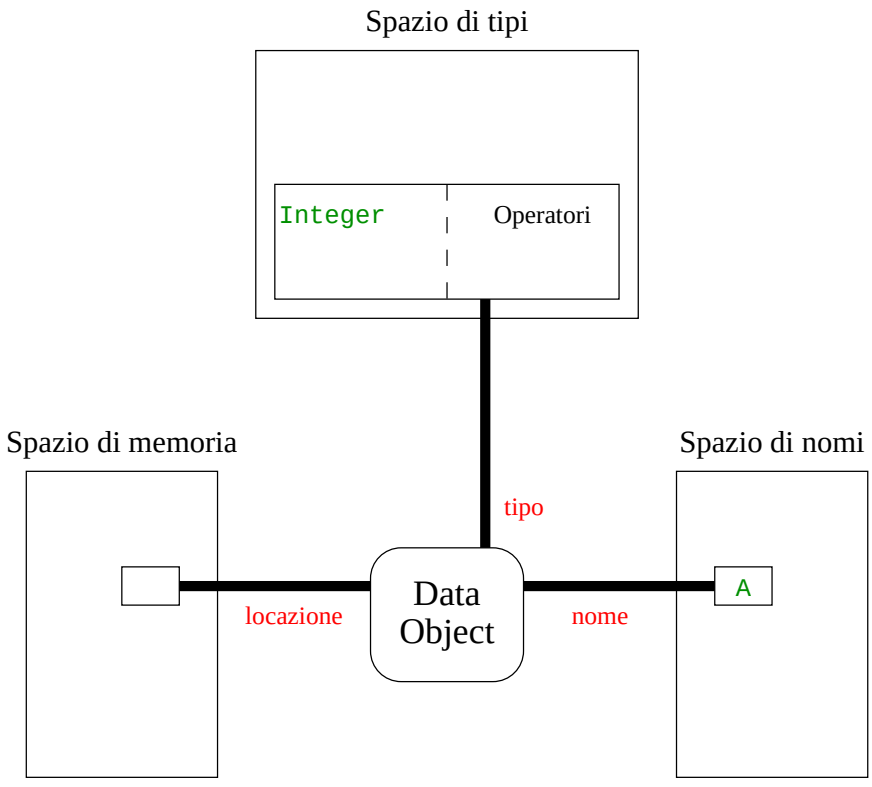
Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).
2. Durante il caricamento in memoria (load time).
3. Durante l'esecuzione (run time).

Possiamo notare che:

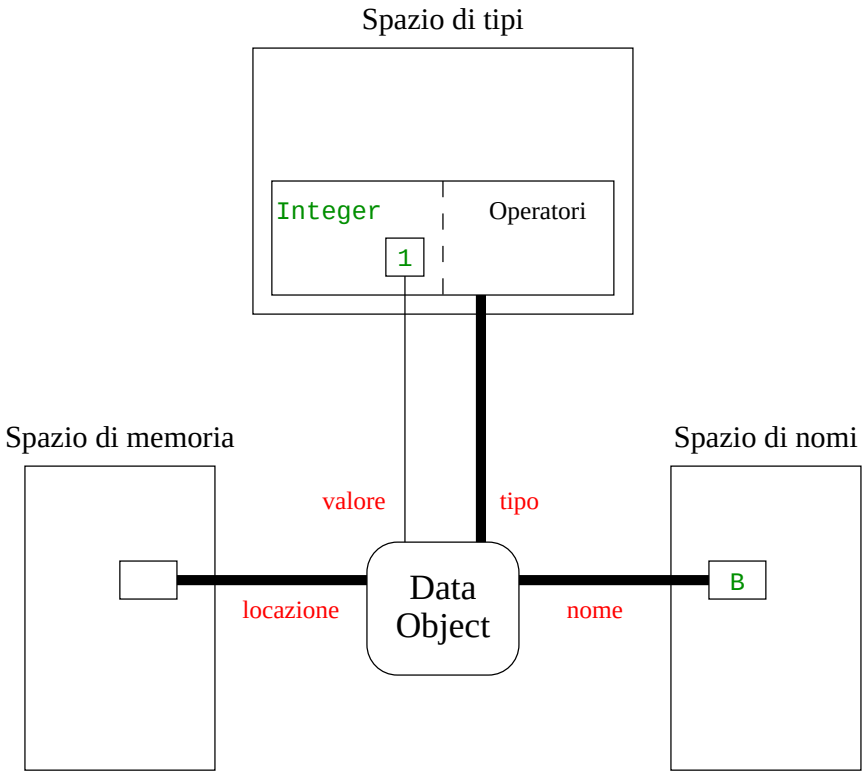
- il *location binding* avviene durante il caricamento in memoria;
- il *name binding* avviene durante la compilazione, nell'istante in cui il compilatore incontra una dichiarazione;
- il *type binding* avviene (di solito, si veda dopo) durante la compilazione, nell'istante in cui il compilatore incontra una dichiarazione di tipo; un tipo è definito dal sottospazio di valori (e dai relativi operatori) che un *data object* può assumere.

Esempio 1



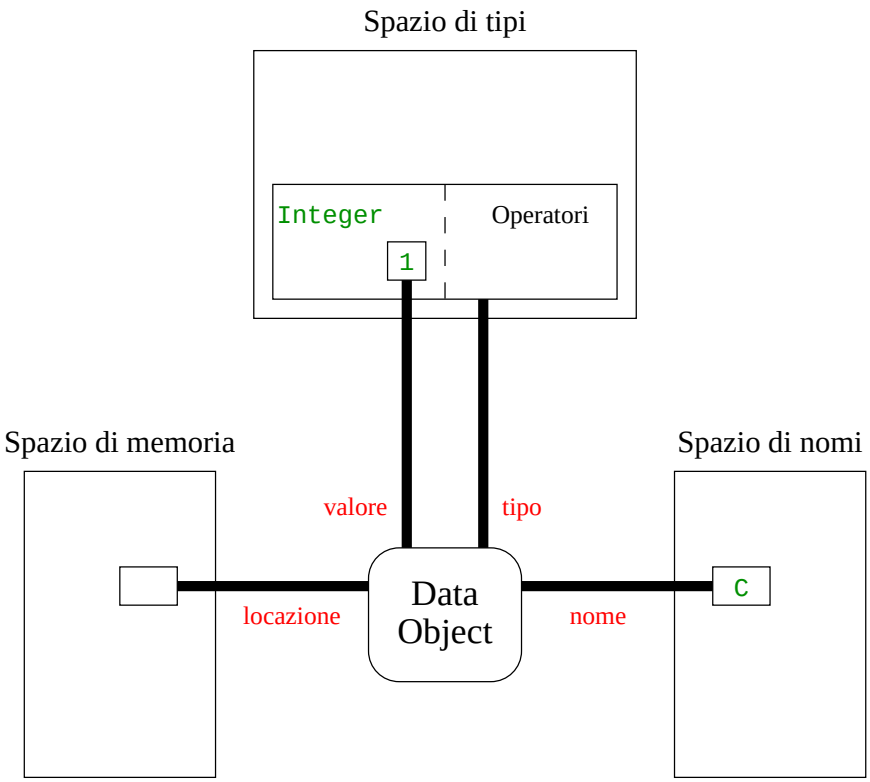

```
A: integer;
```

Esempio 2



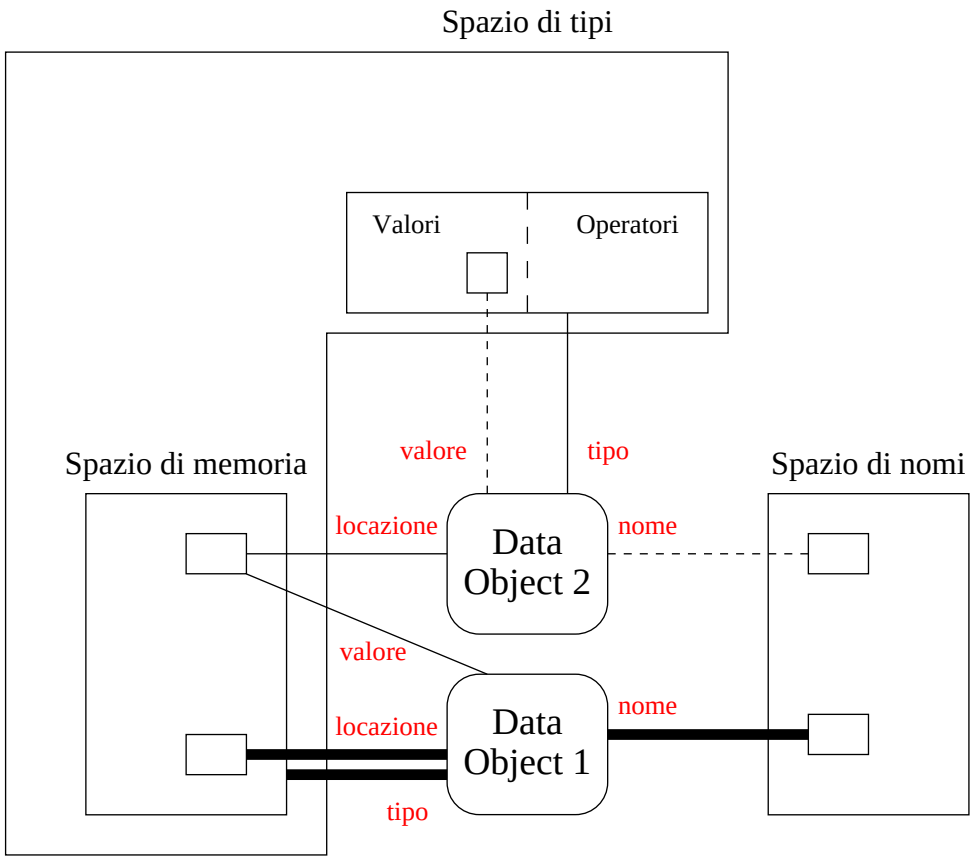

```
B: integer:= 1;
```

Esempio 3




```
C: constant integer:= 1;
```

Il puntatore (1)



Il puntatore (2)

- 2 *data object* coinvolti.
- Il secondo può non avere un legame di nome o di valore.
- La deallocazione è necessaria, perché la modifica del legame di valore genera di solito dati non più accessibili per nome o riferimento.
- Alcuni linguaggi possiedono meccanismi di recupero automatico di memoria (*garbage collector*).

Legame di tipo

- Per definizione è correlato al legame di valore.
- Sia quando si instaura, sia quando viene modificato, occorrerebbe controllare (type checking) la consistenza con il legame di valore.
- Un linguaggio è *dinamicamente tipizzato* se il legame (e le variazioni di legame) e di conseguenza anche il controllo di consistenza (se avviene) avvengono durante l'esecuzione.

  Esempio: nei linguaggi di scripting

  ```
  x=1; ... x= "abc";
  ```

  Inizialmente il tipo è numerico, poi è stringa (il legame di tipo cambia in seguito ad un cambio del legame di valore).
- Un linguaggio è *staticamente tipizzato* se il legame avviene durante la compilazione; in questo caso il controllo di consistenza (se avviene) può avvenire in entrambe le fasi.

Type checking (1)

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.
Può avvenire: a) durante la compilazione, b) durante l'esecuzione, c) per nulla.

- Un linguaggio è *fortemente tipizzato* se il controllo di consistenza avviene sempre: il più possibile durante la compilazione e, negli altri casi, durante l'esecuzione.
  - Un linguaggio fortemente tipizzato è anche staticamente tipizzato.
  - Java è fortemente tipizzato (vedremo poi).
  - Pascal è quasi fortemente tipizzato (una sola eccezione di assenza di controllo: i record con varianti).
  - Linguaggi didatticamente rilevanti.

Type checking (2)

- Un linguaggio è *debolmente tipizzato* se il controllo di consistenza può non avvenire affatto in numerosi casi.
  - C è debolmente tipizzato:
    in esso esistono le operazioni di *casting*, che consentono di forzare, in esecuzione, l'interpretazione di un qualunque valore secondo un qualunque tipo (anche un tipo diverso da quello a cui il valore è stato precedentemente associato);
    esistono puntatori a void, che godono, in esecuzione, di conversione di tipo implicita verso qualunque altro tipo puntatore;
    esistono le *unioni*, che consentono di interpretare una collezione di dati correlati secondo diverse attribuzioni di tipo indipendenti.

Il linguaggio perfetto (1)

Sarebbe bello se esistesse un linguaggio Turing completo, in cui il controllo di consistenza di tipo avvenisse completamente durante la compilazione (e in cui il compilatore non generasse più errori del necessario) – Linguaggio Perfetto (LP).

- LP, se esistesse, sarebbe staticamente e fortemente tipizzato (il più "forte" di tutti i fortemente tipizzati).
- LP non può esistere.

Il linguaggio perfetto (2)

- Se LP esistesse, allora il compilatore, per essere capace di decidere la correttezza dell'ultima assegnazione in:

```
int x;
P;
x = "pippo";
```

  dove `P` è un generico programma, dovrebbe essere capace di decidere la terminazione di P, quindi LP non sarebbe Turing completo, contro l'ipotesi.

- I compilatori, in situazioni come la precedente, presumono la terminazione di `P` e segnalano un errore nell'ultima linea.
  Per esempio, questo programma Pascal non compila:

```
program wrong (input, output);
var i: integer;
begin
  if false then i:= 3.14;
           else i:= 0;
end.
```

  Ma un tale programma verrebbe correttamente eseguito: il compilatore ha segnalato un errore di troppo.

Blocchi di istruzioni

Necessità

Istruzioni raggruppate in blocchi per meglio definire:

- ambito delle strutture di controllo;
- ambito di una procedura;
- unità di compilazione separata;
- ambito dei legami di nome.

Definizioni

Blocchi che definiscono l'ambito di validità di un nome contengono due parti:

- una sezione di dichiarazione del nome;
- una sezione che comprende gli enunciati sui quali ha validità il legame.

Definiamo un piccolo pseudo-linguaggio per rappresentare un blocco:

```
...
BLOCK A;
  DECLARE I;
BEGIN A
...               {I DEL BLOCCO A}
END A;
...
```


Ambito di validità di legami

Essenzialmente due tipi di ambito di validità (scoping):

In ambito statico o lessicale.

Blocchi annidati vedono e usano i legami dei blocchi più esterni (*legami non locali*) e, di solito, possono aggiungere legami locali o sovrapporne di nuovi.

In ambito dinamico

Concetto qui esaminato solo in relazione ai blocchi annidati, ma che assume il proprio senso maggiore quando vi sono procedure chiamanti e chiamate. In questo caso la procedura chiamata vede e usa i legami visti e usati dalla procedura chiamante.

Esamineremo tutti i dettagli nella prossima lezione.

Static scoping

```
PROGRAM P;
  DECLARE X;
BEGIN P
  ...               {X da P}
  BLOCK A;
    DECLARE Y;
  BEGIN A
    ...             {X da P, Y da A}
    BLOCK B;
      DECLARE Z;
    BEGIN B
      ...           {X da P, Y da A, Z da B}
    END B;
    ...             {X da P, Y da A}
  END A;
  ...               {X da P}
  BLOCK C;
    DECLARE Z;
  START C
    ...             {X da P, Z da C}
  END C;
  ...               {X da P}
END P;
```


Mascheramento

```
PROGRAM P;
  DECLARE X,Y;
BEGIN P
  ...               {X e Y da P}
  BLOCK A;
    DECLARE X,Z;
  BEGIN A
    ...             {X e Z da A, Y da P}
  END A;
  ...               {X e Y da P}
END P;
```

Allocazione statica di memoria

Si dice *allocazione statica di memoria* (load-time) quando le variabili conservano il proprio valore ogni volta che si rientra in un blocco (il legame di locazione è fissato e costante al tempo di caricamento). Se il linguaggio prevede ciò, allora, per esempio:

```
PROGRAM P;
  DECLARE I;
BEGIN P
  FOR I:=1 TO 10 DO
    BLOCK A;
      DECLARE J;
    BEGIN A
      IF I=1 THEN
        J:=1;              {I da P, J da A}
      ELSE
        J:=J*I;
      END IF
    END A;
END P;
```


Allocazione dinamica di memoria

- Si dice *allocazione dinamica di memoria* (run-time) quando il legame di locazione (e anche di nome) è creato all'inizio dell'esecuzione di un blocco e viene rilasciato a fine esecuzione.
- Essa è realizzata attraverso il *record di attivazione* di un blocco.
  - Un record di attivazione contiene tutte le informazioni sull'esecuzione del blocco necessarie per riprendere l'esecuzione dopo che essa è stata sospesa.
  - Può contenere informazioni complesse (come si vedrà in seguito), ma per realizzare un legame dinamico di locazione in blocchi annidati è sufficiente che contenga le locazioni dei dati locali più un puntatore al record di attivazione del blocco immediatamente più esterno.

Stack di attivazione

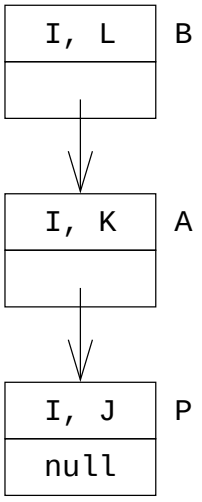
In ogni momento dell'esecuzione lo *stack di attivazione* contiene i record "attivi":

1. il top dello stack contiene sempre il record del blocco correntemente in esecuzione;
2. ogni volta che si entra in un blocco, il record di attivazione del blocco viene posto sullo stack (push);
3. ogni volta che si esce da un blocco, viene eliminato il record al top dello stack (pop).

Esempio di allocazione dinamica

All'ingresso in B

Contenuto dello stack di attivazione




```
PROGRAM P;
  DECLARE I,J;
BEGIN P
  BLOCK A;
    DECLARE I,K;
  BEGIN A
    BLOCK B;
      DECLARE I,L;
    BEGIN B
      ...          {I e L da B, K da A, J da P}
    END B;
    ...            {I e K da A, J da P}
  END A;
  BLOCK C;
    DECLARE I,N;
  BEGIN C
    ...            {I e N da C, J da P}
  END C;
...                {I e J da P}
END P;
```


Bibliografia

Bibliografia

- Dershem, Jipping. *Programming Languages: Structures and Models*. Cap. 1 e 2; cap. 3, par. 3.1.1 - 3.1.5, 3.1.12, 3.2.

Esercizio 1

Si considerino i seguenti blocchi annidati. Per ciascun blocco, determinare i legami di ogni variabile. Evidenziare le variabili locali e non locali.

```
PROGRAM P;
  BLOCK B1;
    DECLARE A,B,C;
  BEGIN B1
    BLOCK B2;
      DECLARE C,D;
    BEGIN B2
      BLOCK B3;
        DECLARE B,D,F;
      BEGIN
        ...
      END B3;
      ...
    END B2;
    BLOCK B4;
      DECLARE B,C,D;
    BEGIN B4
      ...
    END B4;
    ...
  END B1;
END P;
```


Esercizio 2

Tracciare il contenuto dello stack di attivazione durante l'esecuzione del seguente pseudo-programma.

```
PROGRAM P;
  DECLARE X,Y;
BEGIN P
  BLOCK A;
    DECLARE X,Y,Z;
  BEGIN A
    BLOCK B;
      DECLARE Y;
    BEGIN B
      BLOCK C;
        DECLARE X,Y;
      BEGIN C
        ...
      END C;
      BLOCK D;
        DECLARE Z;
      BEGIN D
        ...
      END D;
      ...
    END B;
  END A;
  BLOCK E;
    DECLARE Z;
  BEGIN E
    BLOCK F;
      DECLARE X;
    BEGIN F
      ...
    END F;
    ...
  END E;
  ...
END P;
```