Paradigma imperativo

Paradigma Imperativo

- Del paradigma imperativo faremo quelle parti che costituiscono un'importante “preistoria concettuale” per il paradigma ad oggetti

Programma nel paradigma (modello) imperativo

- Il paradigma imperativo sottolinea la visione del calcolatore come
  - singola CPU
  - + memoria

- Ricordiamo che i programmi scritti nei linguaggi appartenenti a questo paradigma consistono di
  - *Descrizioni di sequenze di modifiche della "memoria" del calcolatore.*

Note su nozione di "memoria"

- Di fatto 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*"

Note su nozione di "memoria"

- La funzione "memoria" può essere rappresentata graficamente:

0	1033
1	12
2	0
3	0
4	0
5	41
6	0

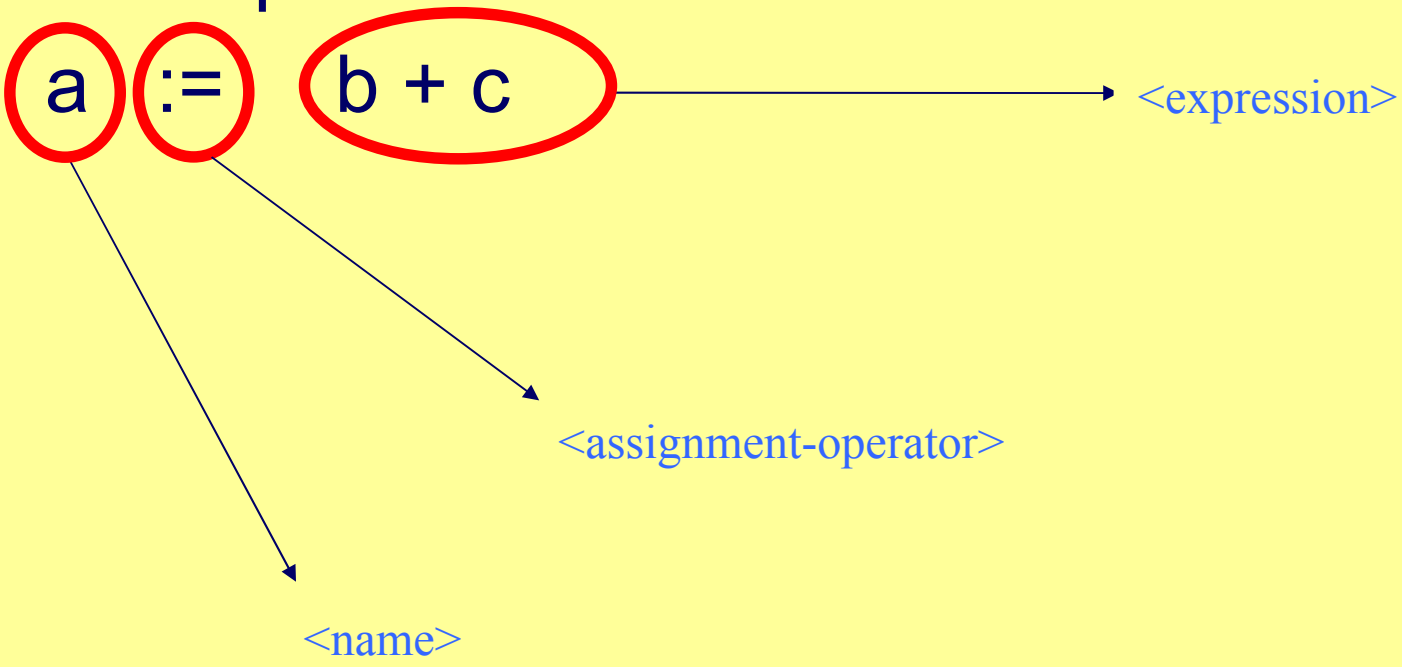
Manipolazione della memoria nel paradigna imperativo: Statement di assegnazione

- Definizione grammaticale (*sintassi*)
- < statement-di-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 assegnazione

- Esempio in Pascal

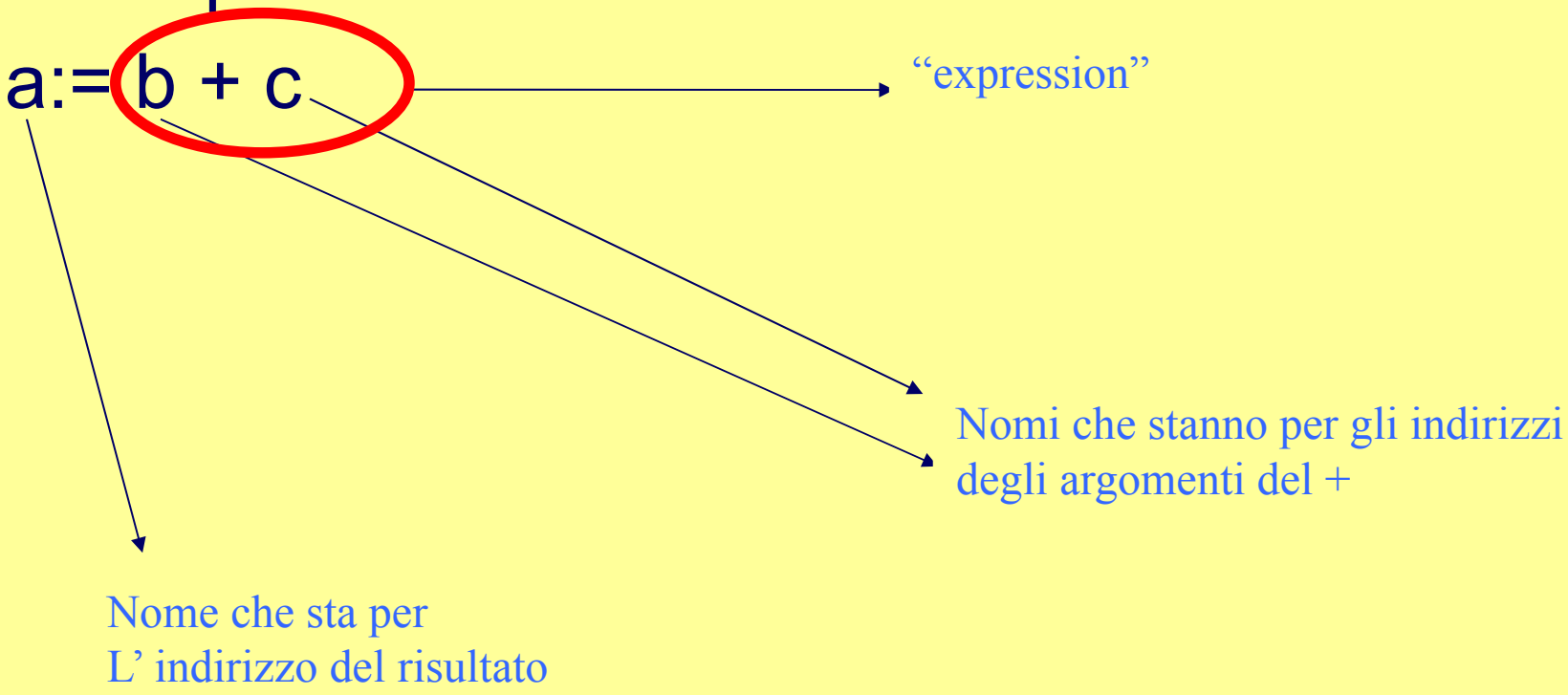


Cosa succede quando *eseguiamo* un assegnazione? (*semantica*)

- 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 variabile

Esempio assegnazione

- Esempio in Pascal



Relazioni linguaggio macchina

- Lo statement di assegnazione, a parte l'astrazione che sostituisce agli indirizzi i nomi, ricalca la tipica unità di esecuzione in un linguaggio macchina:

- 1. ottenere indirizzi delle locazioni di operandi e risultato;
- 2. ottenere valori di operandi da locazioni di operandi;
- 3. valutare risultato;
- 4. memorizzare risultato in locazione risultato.

- Quindi il paradigma imperativo si caratterizza per l'uso dei nomi come astrazione di indirizzi di locazioni di memoria.

- Raffiniamo questo concetto...

Natura concettuale delle variabili

- Le *variabili* di un programma costituiscono una *memoria*?
- E i *parametri formali* delle funzioni/procedure?

Natura concettuale delle variabili

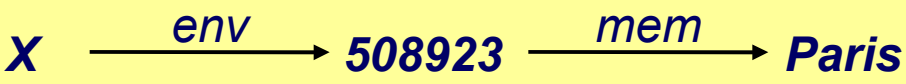
- Le *variabili* di un programma costituiscono una *memoria*?
- E i *parametri formali* delle funzioni/procedure?
- Non proprio (sia concettualmente che per come sono realizzati)

La nozione di *ambiente* o *environment*

- *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*) = ???

L'ambiente nel paradigma imperativo

- Mappa identificatori su *locazioni...*
- ...associate ai valori da una memoria
- Il *valore* di una variabile x è *mem*(*env*(x))



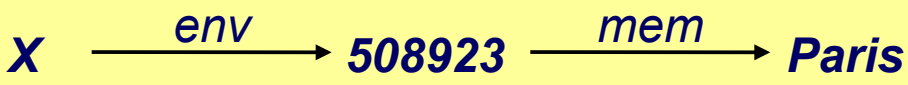
Manipolazione della memoria nel paradigna imperativo: Statement di assegnazione

Forse non ci avete mai pensato ma...

- Considerate l'assegnamento
  - X := X+1
- A *sinistra* X indica la *locazione* associata (cioè *env*(X)), destinata a contenere il risultato
- A *destra* X indica il *valore* della variabile, cioè *mem*(*env*(X))

L'ambiente nei paradigmi funzionale e imperativo

- Nel funzionale
  - Mappa direttamente gli identificatori sui valori
  - Ad es. potrebbe essere *env*(city)=”Paris”
- Nell'imperativo
  - Mappa identificatori su *locazioni...*
  - ...associate ai valori da una memoria
  - Il *valore* di una variabile x è *mem*(*env*(x))



L'ambiente nei paradigmi funzionale e imperativo

- Nel funzionale
  - Mappa direttamente gli identificatori sui valori
  - Ad es. potrebbe essere *env*(city)=”Paris”
- È dunque come una memoria? La differenza è solo nel *dominio* della funzione? (nomi piuttosto che indirizzi)

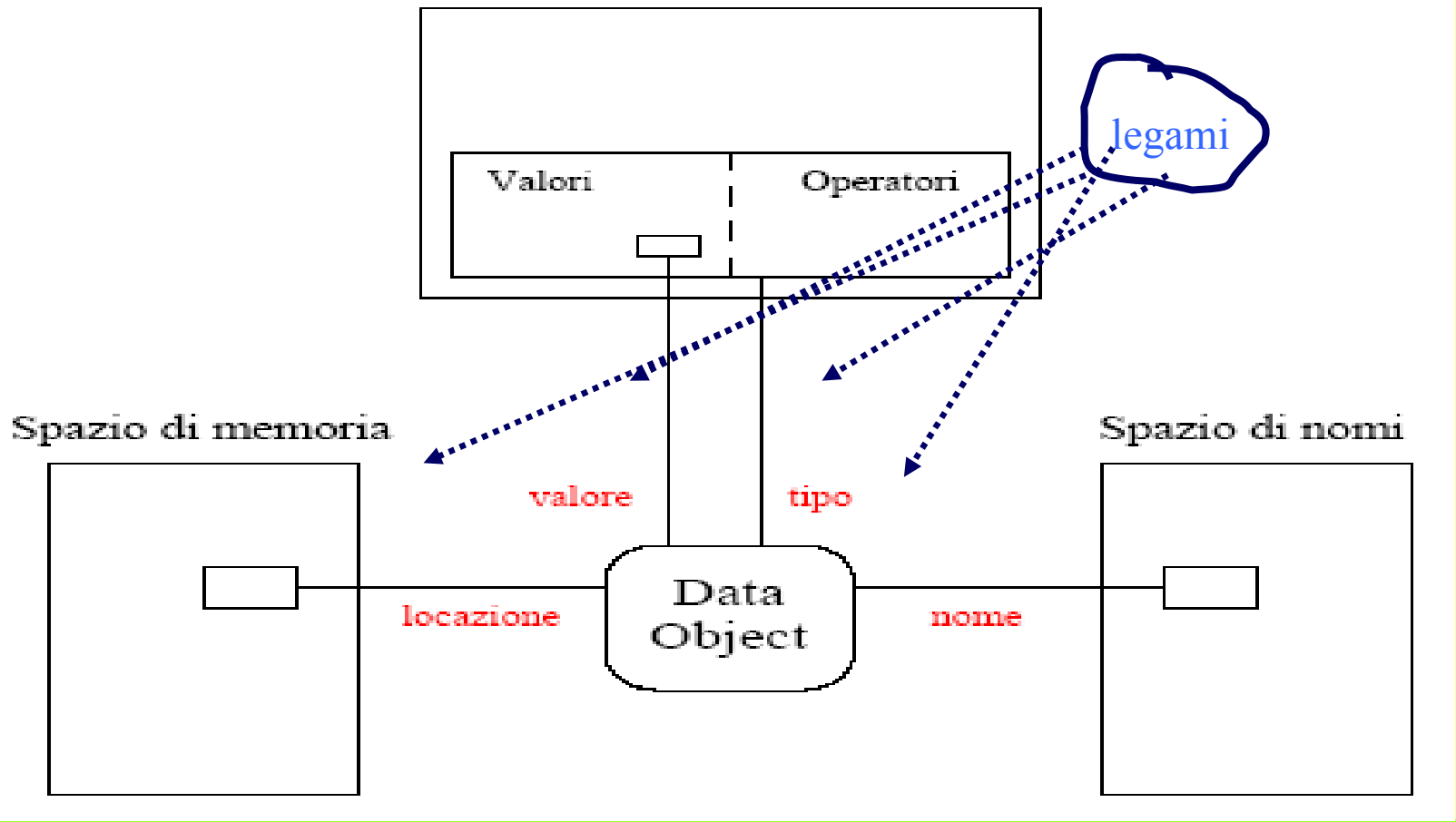
L'ambiente nei paradigmi funzionale e imperativo

- Nel funzionale
  - Mappa direttamente gli identificatori sui valori
  - Ad es. potrebbe essere *env*(city)=”Paris”
- È dunque come una memoria?
- NO: In *env* l'associazione tra identificatori e valori/locazioni è ***immutabile*** finchè esiste l'id.
  - Mentre in *mem* l'associazione locazione-valore ***cambia*** durante l'esecuzione (assegnamenti)
  - Nel paradigma funzionale niente assegnamenti!!

Legami e data object

- Il data object è l' astrazione mediante la quale noi esploreremo vari concetti del paradigma imperativo.
- E' quindi qualcosa,creato da noi, di cui ci serviamo per descrivere concetti importanti del paradigma.
  - Un data object è la quadrupla (L,N, V, T), ove:
  - L: locazione.
  - N: nome.
  - V : valore.
  - T: tipo.
- Un *legame* (*binding*) è la determinazione di una delle componenti.

Rappresentazione di un dataObject mediante un' immagine



Esempi delle 4 componenti

- Locazioni: gli interi da 0 alla dimensione della memoria della macchina (ad es 2Gb)
  - Gli indirizzi possono essere *relativi* (allo stack,...)
- Nomi (o *identificatori*): in Ada:
  - <identifier> ::= <letter> { [underline]<letter-or-digit>}
  - <letter-or-digit> ::= <letter> | <digit>
- Tipi (e valori): integer
  - Valori: da 0 a 2^{16}
  - Operatori: +, -, *, /, ...

Modifiche di un legame

- Variazioni (e quindi definizioni) di legami 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 in molti casi avviene durante il caricamento in memoria **(load time)**
  - il name binding avviene durante la compilazione **(compile time).,** nell'istante in cui il compilatore incontra una dichiarazione
  - il type binding avviene solitamente durante la compilazione **(compile time)** nell'istante in cui il compilatore incontra una dichiarazione di tipo; un tipo è definito dal sottospazio dei valori che un data object può assumere e dai relativi operatori.

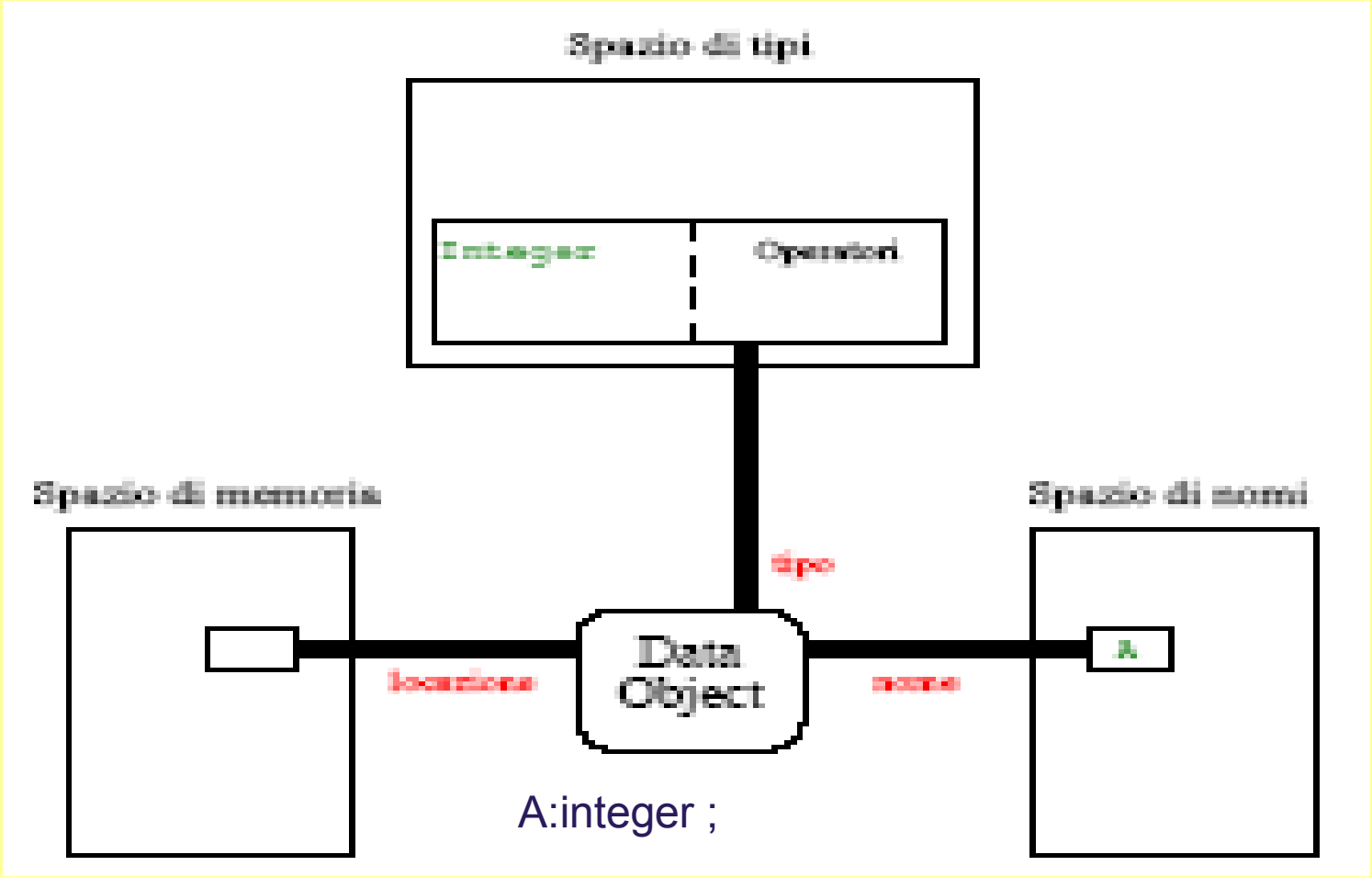
Considerazioni sul legame di tipo ed esempi

Legame di tipo

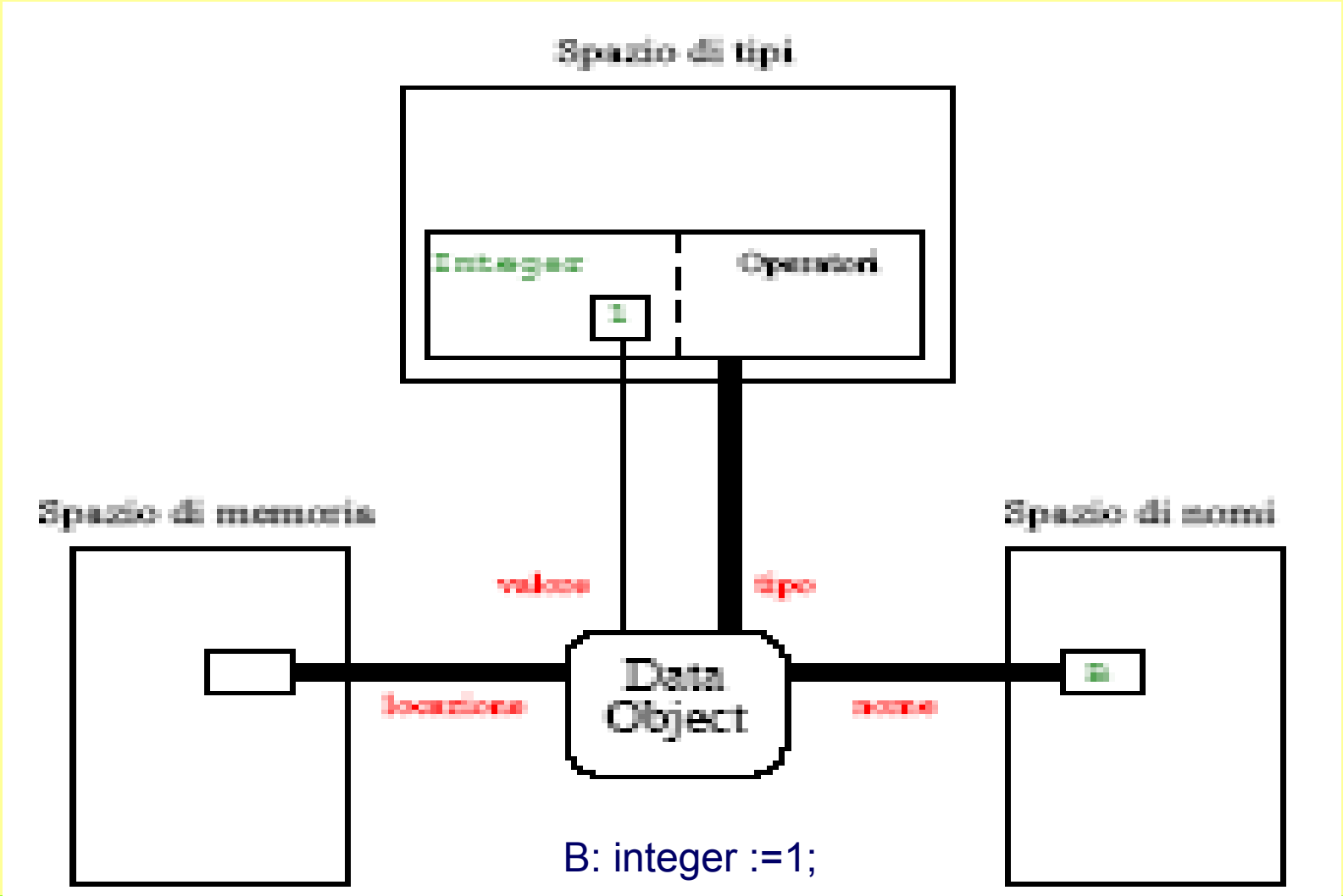
- Legame solitamente instaurato e fisso durante la compilazione tranne per linguaggi che realizzano il per il tipo **dynamic binding cioè il binding di tipo costruito a run time**. In essi il legame di tipo dipende dal valore del dato: se il valore muta durante l'esecuzione, il dato assume anche il tipo del nuovo valore.
- Esempio: nei tipici scripting language:
  - X=1; ... X="abc";
  - Inizialmente il tipo è numerico; poi stringa

Esempi in termini di data objects

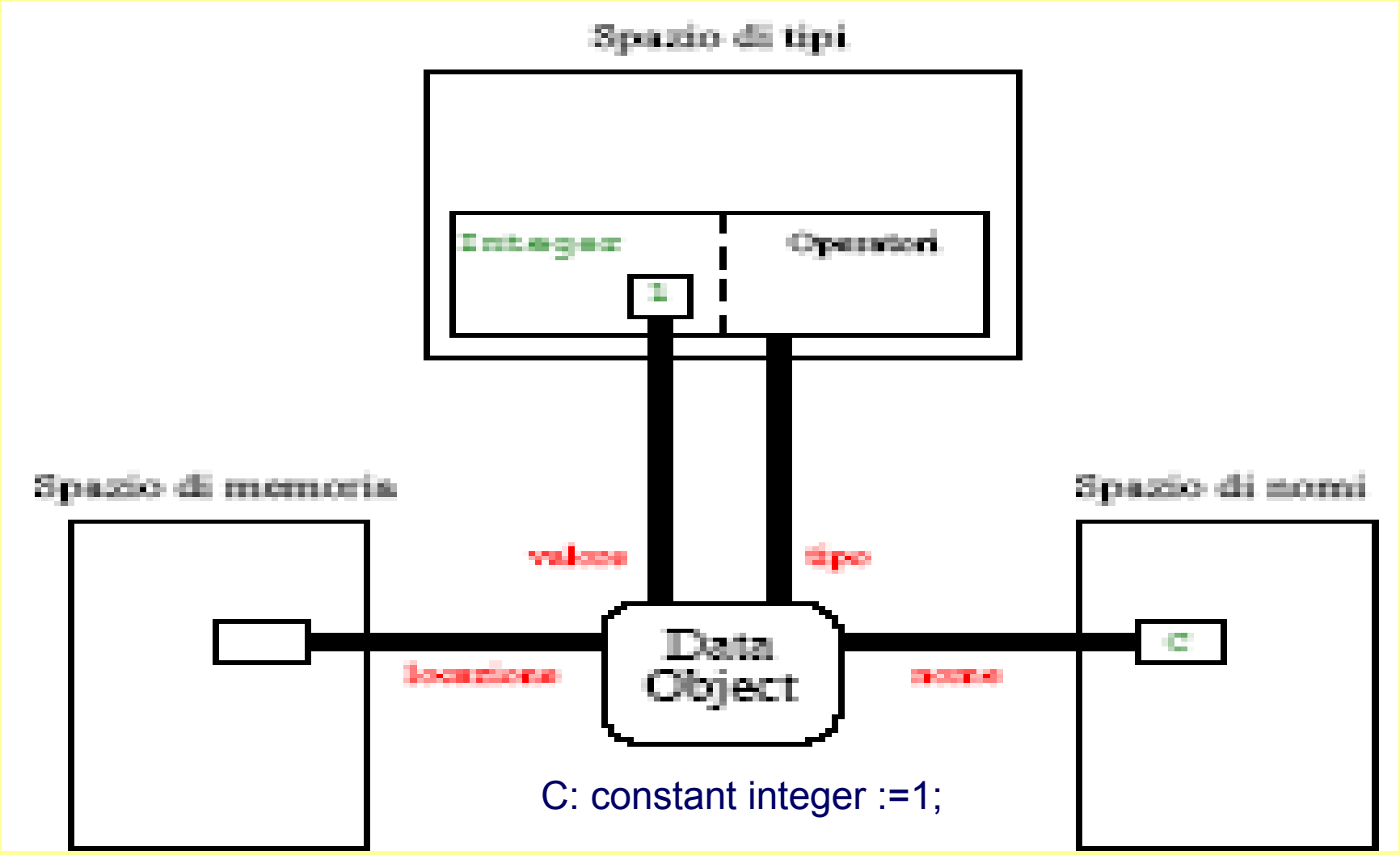
Dataobject


Spazio di tipi
Integer
Operator
Spazio di memoria
Spazio di nomi
tipo
Data Object
locazione
nome
A
A:integer ;

Dataobject B



Dataobject c



C: constant integer :=1;

Type checking (controllo di consistenza)

- Type checking è il meccanismo di controllo di consistenza della coppia valore-tipo.
- Può avvenire:
  - durante la compilazione,
  - durante l'esecuzione,
  - per nulla.

Domanda

- Se un linguaggio realizza il “dynamic binding per il tipo” quale controllo di consistenza è più adeguato?

Nomenclatura linguaggi a seconda del type checking

- ***Un linguaggio si dice fortemente tipizzato*** se i tipi vengono controllati esaustivamente (si veda il Gabbrielli & Martini)
  - il controllo avviene il più possibile durante la compilazione (controllo *statico*)
  - e quando non è possibile durante l'esecuzione (controllo *dinamico*)
  - PASCAL è quasi fortemente tipizzato – si vedano i record con varianti)
  - ADA e Java sono fortemente tipizzati
  - C/C++ non lo sono affatto (non solo per le union)!

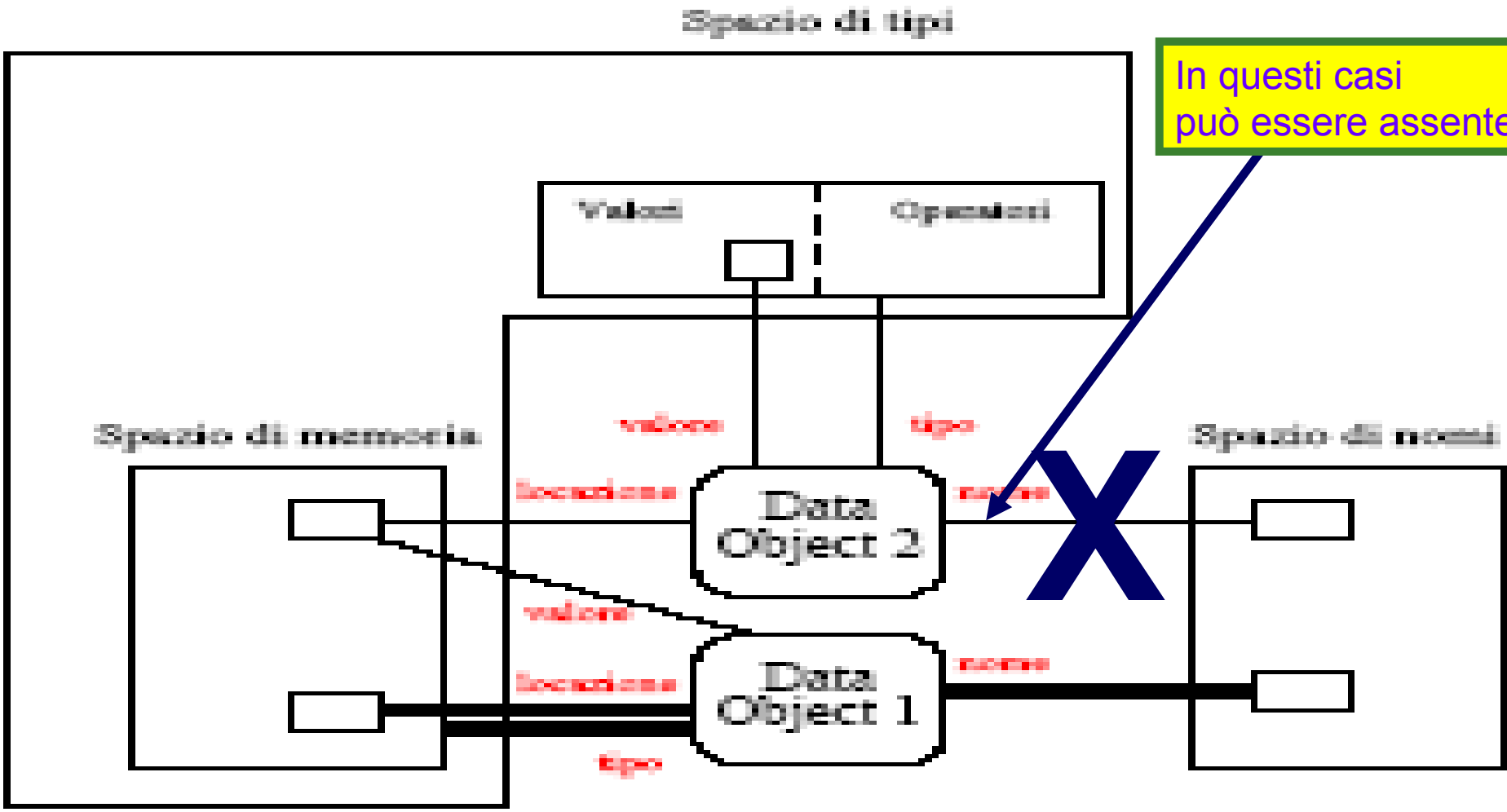
Nomenclatura linguaggi a seconda del type checking

- ***Un linguaggio è dinamicamente tipizzato*** se il legame e di conseguenza anche il controllo di consistenza avvengono durante l'esecuzione.
- ***Un linguaggio è staticamente tipizzato*** se il legame avviene durante la compilazione; in questo caso il controllo di consistenza può avvenire in entrambe le fasi.

Esempio di un tipo particolare: il puntatore

- Esempio in C:
  - char * p = & x;
  - *p = 10;
- Osservazioni:
  - 2 data object coinvolti:Il secondo può non avere un legame di nome o di valore. Considerate p = (& x)+3 ....
  - La deallocazione è necessaria, perchè la modifica del legame di valore del puntatore genera di solito dati non più accessibili per nome o riferimento.
  - Alcuni linguaggi possiedono meccanismi di recupero automatico di memoria (garbage collector).

Tipo Puntatore



Tipo puntatore

- Memorizzate per quando faremo il linguaggio Java che il
  - tipo puntatore è caratterizzato non solo dai valori : indirizzi di memoria,
  - ma anche dalle operazioni che potete fare sui “dataobject” puntatori (si veda il C).
- Poiché in Java non sono definite queste operazioni (in slang : non c’è l’aritmetica del puntatore) in Java non è definito il tipo puntatore.

Programma dal punto di vista dell' esecuzione

- Nel paradigma imperativo: il programma è una "unità di esecuzione", la più estesa.
- Frequentemente un programma è articolato in unità più piccole dette **blocchi**.
  - In C/C++: delimitati da graffe
- La più piccola unità di esecuzione indivisibile è lo **statement** .
  - L' assegnazione è lo statement mediante il quale si modifica la memoria

Cos'è un blocco

- Un blocco, qualunque sia la motivazione per definirlo, è sempre una collezione di unità di esecuzione raggruppati..

per le strutture condizionali

- Servono per delimitare e differenziare gli statements eseguibili quando la condizione è vera da quelli eseguibili quando la condizione è falsa: Sintassi del tipo:
- If <Boolean expression> then

  <block of statements>

  else <block of statements>

Lo statement “case” è un’ altro esempio.

utile mezzo nel processo di sviluppo di un programma

- Vedremo che caratterizzano l' astrazione procedurale che permette a sua volta la progettazione top down dei programmi.
- Questo tipo di blocchi ha il pregio di favorire la riutilizzabilità del codice.

Compilation Units

- Si compilano separatamente blocchi che poi vengomo riuniti insieme per l' esecuzione.
- In C/C++ ciascun *file* che costituisce un programma è una compilation unit

Ambito (scope) dei Legami

- Blocchi di statement in cui sono validi i vari binding

Riassumendo

- Istruzioni raggruppate in blocchi per meglio definire:
  - ambito delle strutture di controllo;
  - ambito di una procedura;
  - unità di compilazione separata;
  - ambito dei legami.

Adesso ci occuperemo appunto, della definizione dell'ambito di validità dei legami.

Per spiegare meglio le cose:

- Definiamo un piccolo pseudo-linguaggio per rappresentare un blocco:
- BLOCK A;

"eventuali dichiarazioni di nomi:nome1, nome2…";

- BEGIN A
- "unità di esecuzione"
- ……
- ….
- END A;

} {nome1 da A,….}

AMBITO DI VALIDITÀ del legame di *locazione*

- Due strategie possibili per i vari linguaggi:
- 1. Allocazione Statica di Memoria,
  2. Allocazione Dinamica di memoria

AMBITO DI VALIDITÀ del legame di locazione : Esempio

```
program P;
    DECLARE I;
BEGIN P;
     FOR I := 1 TO 10 DO
          BLOCK A;                                  {I from P}
              DECLARE J;
           BEGIN A
                IF I := 1 then                      {I from P, J from A}
                   J := 1;
                ELSE
                   J := J * I;
                END IF
  END A;
END P;
```

Cosa succede di J quando si esce e poi si rientra nel bloccoA?

Cosa è necessario sapere.

- Per ogni linguaggio dobbiamo conoscere quale strategia esso usa e in quali casi, fra l'Allocazione Statica e l'Allocazione Dinamica della memoria.

Allocazione statica della memoria

- Un oggetto ‘dato’ è legato alla sua locazione al tempo di caricamento (**load-time**) e questo legame dura per l’intera esecuzione del programma: quindi “J” dell’ esempio precedente può mantenere il suo valore quando si rientra in A semplicemente perché le dichiarazioni non vengono riprocessate e il legame alle locazioni ,delle variabili , locali al blocco, e all’ eventuale valore di inizializzazione viene fatto a load time.

Uso della allocazione statica di memoria

- Quando serve che le variabili conservino il loro valore ogni volta che si rientra in un blocco si richiede l' allocazione statica di memoria (cioè locazione definita a load-time).
  - Esempio: una variabile per assegnare numeri progressivi (come in posta)

Allocazione Dinamica della memoria

- Si dice allocazione dinamica di memoria (cioè allocazione definita a run-time) quando il legame di locazione, è creato all'inizio dell'esecuzione di un blocco e viene rilasciato a fine esecuzione.

.

Necessità di creare strutture per una corretta esecuzione

- Già parlando semplicemente dell' allocazione dinamica e/o statica delle locazioni ci rendiamo conto della necessità di creare nuove strutture relative all' esecuzione di un programma.
  - Parte del *supporto a run time*

Necessità di definire nuove strutture

I due fatti

1. allocazione statica di memoria della memoria, comporta che, uscendo ed entrando in (da) un Blocco, il legame alle locazioni non varia a run time,
2. allocazione dinamica comporta che, il legame alla locazione sia rilasciato, ogni volta che finisce l' esecuzione di un blocco e creato ogni volta che inizia l' esecuzione di un Blocco,

evidenziano il bisogno di una nuova struttura dati, in particolare un record, chiamato **record di attivazione, da** associare ad ogni un blocco per risolvere, tutti i problemi connessi a realizzare una corretta esecuzione compreso il governare i binding di locazione.

Record di attivazione

- ***In generale il record di attivazione serve a contenere tutta l'informazione circa una unità complessa di esecuzione, che è necessaria per rendere nuovamente attiva una sua esecuzione che è stata sospesa.***
- Per quello che riguarda l' allocazione (statica o dinamica) di memoria è necessario solo che il record di attivazione relativo ad ogni blocco contenga le locazioni per i data objects localmente ad esse legati. L' allocazione di memoria sarà dinamica se quando il blocco viene lasciato, i legami alle locazioni vengono cancellati e quando il blocco viene rieseguito, vengono riprocessate le dichiarazioni. Si avrà allocazione statica invece se i bindings di locazione permangono dopo l' uscita dal blocco e le dichiarazioni nel blocco vengono processati solo la prima volta che il blocco viene attivato.

Nesting dei blocchi

Il fatto che i Blocchi possano essere contenuti l'uno nell'altro richiede la presenza nel record di attivazione di un blocco, di un **puntatore al record del blocco** che immediatamente contiene il blocco che stiamo considerando, al quale si ripasserà il controllo quando si termina il blocco in questione. Questo permette di gestire correttamente l' uscita da un blocco. Per gestire la ripresa della computazione in un blocco, occorre inoltre, avere anche un **puntatore all' istruzione** da eseguire quando il blocco riavrà il controllo dell' esecuzione.

Altra struttura importante (stack)

Il fatto che i Blocchi siano innestati l' uno nell'altro rende necessario definire un ulteriore struttura dati associata all' esecuzione del programma, che contenga l' informazione necessaria a rappresentare correttamente attivazione-entrata (disattivazione- sospensione-uscita) dei un blocchi durante l' esecuzione del programma.

Entrare e uscire dai blocchi

- Per gestire correttamente l'entrare e l'uscire da un blocco, si usa uno stack i cui elementi sono record di attivazione, tale che:
- quando a **run-time**, si entra in un blocco Y , il record di attivazione di Y viene messo nel 'top' dello stack, quando si esce da Y, Y è tolto dal 'top' dello stack .

Blocchi dal punto di vista dell' ambito di validità del legame di nome

- Sono in genere formati da due parti:
  - Una sezione nella quale possono essere **dichiarati** i legami di nome che valgono dentro il blocco, legami detti legami locali.
  - Una sezione **eseguibile** nel quale valgono sia i legami locali che quelli non locali.

Riprendiamo il linguaggino:

- Definiamo un piccolo pseudo-linguaggio per rappresentare un blocco:
- BLOCK A;

"eventuali dichiarazioni di nomi:nome1, nome2…";

- BEGIN A
- "unità di esecuzione"
- ……
- ….
- END A;

{nome1 da A,….}

Binding locali di nomi

- “nome1 nome2…. Valgono nel blocco A in cui sono definiti e formano i cosidetti binding locali.

Legami ereditati o non locali

- Dentro un blocco valgono anche binding ereditati da altri blocchi .
- quali sono i blocchi da cui si eredita varia da linguaggio a linguaggio.
- Alcuni linguaggi richiedono una dichiarazione esplicita e separata dalle dichiarazioni dei legami locali, altri invece seguono implicitamente politiche per la determinazione dell' ereditarietà

Politiche per ereditare :

Ambito di validità detto statico o lessicale:

un blocco eredita legami locali e non locali del blocco più piccolo che contiene la sua dichiarazione .

- Ambito di validità dinamico (ha senso per le procedure) ne parleremo in seguito.

Esempio eredità statica

- Esempio 1:
- Program P;
- DECLARE X;
- BEGIN P
- …… *{X from P}*
- **BLOCK A**;
- DECLARE Y;
- BEGIN A
- …… *{X from P, Y from A}*
- BLOCK B;
- DECLARE Z;
- BEGIN B
- …… *{X from P, Y from A, Z from B}*
- END B; *{X from P, Y from A}*
- ……
- **END A;**
- …… *{X from P}*
- BLOCK C;
- DECLARE Z;
- BEGIN C
- ……. *{X from P, Z from C}*
- END C;
- END P;

Riassumendo:Ambito statico o lessicale

- Per determinare l' ambito di validità dei nomi in modo statico o lessicale si applica la seguente regola definita ricorsivamente:
- Se un nome, usato in un blocco X, è dichiarato nel blocco X, esso è legato all'oggetto specificato nella dichiarazione del nome.
- Se un nome ***non ha dichiarazioni dentro il blocco X*** , quel nome è legato allo stesso oggetto al quale è legato nel blocco immediatamente contenente la dichiarazione del blocco X .

Esempio detto "hole in the scope"

L' X definito in P dovrebbe essere visibile
Nel blocco A ,ma poiché è dichiarato in A un X
Quest' ultimo maschera il primo consistentemente alla definizione data.

- Esempio 2:
- Program P;
- DECLARE X, Y;
- BEGIN P
- ……
- ……
- BLOCK A;
- DECLARE X, Z;
- BEGIN A
- ……
- END A;
- ……
- END P \

{X from P, Y form P}

{X from A, Y from P, Z from A}

{X from P, Y from P}

Cenni all'implementazione

```
Program P;
   DECLARE I, J;                          {I from P, J from P}
   BEGINE P
BLOCK A;
   DECLARE I, K;                {I from A, J from P, K from A}
BEGIN A

BLOCK B;
   DECLARE I, L: Integer;
BEGIN B/ BLOCCO "B"  {I from B, J from P, K from A, L from B}
……
END B;
……
END A;
BLOCK C;
   DECLARE I, N;                          {I from C, N from C}
BEGINE C
……
END C;
END P;
```

In **B** possiamo rappresentare
- I con (0, 0)
- L con (0, 1)
- K con (1, 1) *ereditati*
- J con (2, 1)

mentre in **P**
- I (0,0)
- J (0,1)

Posizione della variabile all'interno del blocco di attivazione

e in **A**
- I (0, 0)
- K (0, 1)
- J (1, 1)

Indica il blocco in cui appare riapetto ad A
Preso qui come "0"

Guardare figure pag 42

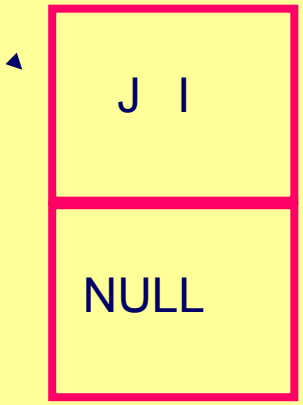
- Appunti-1.doc

Esempio gestione dinamica di mem. e static scoping per i nomi

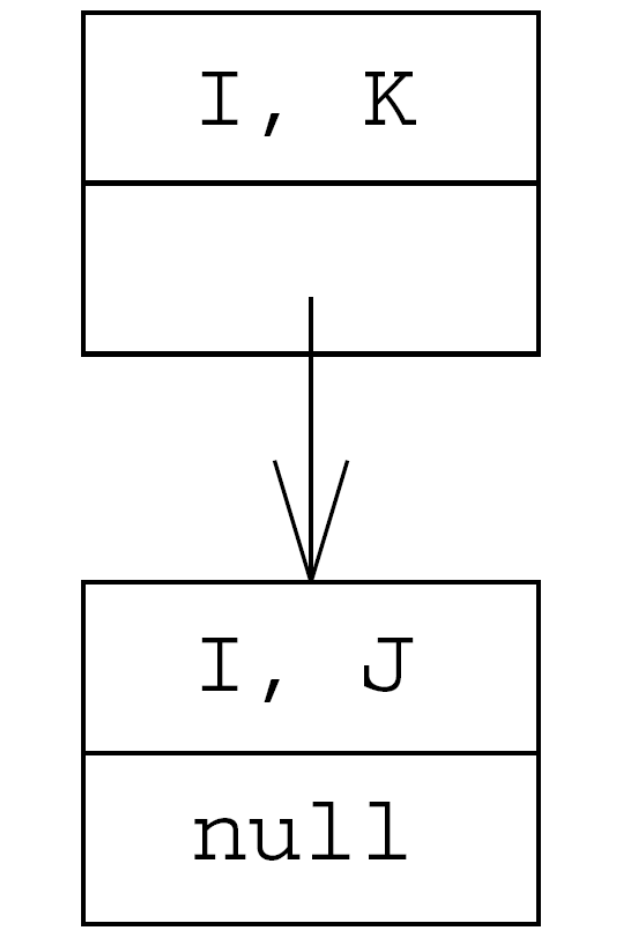
```
Program P;
   DECLARE I, J;                                  {I from P, J from P}
   BEGIN P
BLOCK A;
   DECLARE I, K;                                  {I from A, J from P, K from A}
BEGIN A
BLOCK B;
   DECLARE I, L: Integer;
BEGIN B                    BLOCCO "B"   {I from B, J from P, K from A, L from B B}
……
END B;
……
END A;
BLOCK C;
   DECLARE I, N;                                  {I from C, N from C}
BEGIN C
……
END C;
END P
```

Tutto il contenuto del rett. rosso è noto a compile time

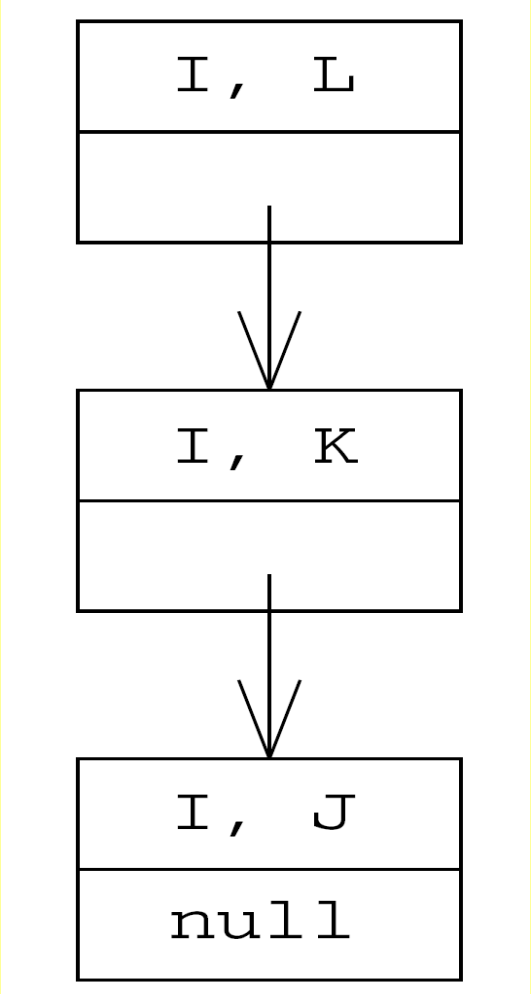
Stack di Attivazione dopo l' entrata in P



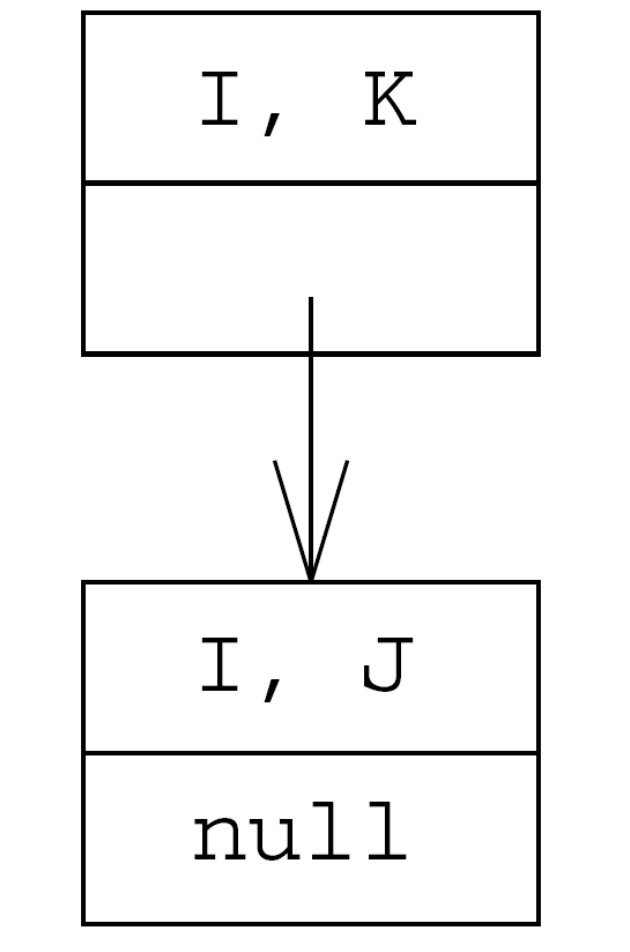
Stack di attivazione: dopo l' entrata di A



Stack di attivazione:dopo l' entrata di B



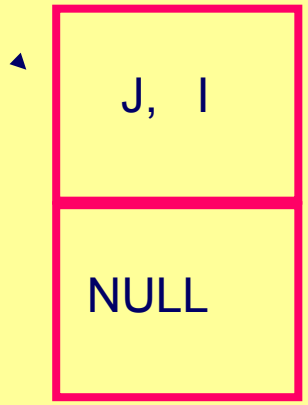
Stack di attivazione: dopo l' uscita da B



Dopo l' uscita da A

- Dopo l' uscita da A

.



Stack di attivazione: dopo l' entrata in C