A.A. 2013/2014

Corso di Laurea in Matematica

LABORATORIO INFORMATICO

La definizione di “Informatica” è complessa.

La parola stessa, contenente parti delle parole “informazione” e “automatica”, però suggerisce che può essere considerata come l'insieme delle teorie e tecniche che permettono di rappresentare e trattare automaticamente l'informazione

L'ACM (Association for Computing Machinary) fornisce la seguente definizione:

Studio sistematico degli algoritmi che descrivono e trasformano l'informazione: la loro teoria, analisi, progettazione, efficienza, realizzazione e applicazione.

Algoritmo

La definizione precedente mette in luce che in informatica la soluzione di un problema consiste nel proporre un algoritmo risolutivo e poi codificarlo in un programma che possa essere eseguito da un calcolatore.

Formalmente, per algoritmo si intende una successione finita di passi o istruzioni che definiscono le operazioni da eseguire su dei dati (=istanza del problema): in generale un algoritmo è definito per risolvere ogni istanza di un problema di un certo tipo.

La parola algoritmo deriva dal nome del matematico persiano Muhammad ibn Mūsa 'l-Khwārizmī ritenuto uno dei primi a riferirsi al concetto di algoritmo

Generalmente un algoritmo è sempre definito supponendo che esso interagisca con un ambiente esterno dal quale acquisisce dati e verso il quale comunica dati o messaggi.

I dati su cui opera un'istruzione sono forniti all'algoritmo dall'esterno oppure sono il risultato di istruzioni eseguite precedentemente.

Proprietà dell'algoritmo

Finitezza: ogni istruzione va eseguita in un tempo finito e deve essere eseguita un numero finito di volte;

Generalità: un algoritmo deve fornire soluzione per tutti i problemi di una classe;

Non ambiguità: i passi devono essere univoci, evitare paradossi, contraddizioni e ambiguità.

Inoltre un algoritmo deve essere **corretto** ed **efficiente**, ossia arrivare alla soluzione giusta e nel modo più veloce possibile, usando la minore quantità di memoria possibile.

Un semplice esempio: la somma dei primi n numeri naturali.

0. inizio;
1. leggi n;
2. poni i = 0 e s = 0;
3. se i > n
    - 3.1 stampa s (che è il risultato);
    - 3.2 vai al passo 7;
4. aumenta s di i (s := s+i);
5. incrementa i di 1 (i := i+1);
6. torna al passo 3.;
7. fine.

Finitezza: **ok**. Però se il passo 5 o il passo 3.2 fossero omessi, l'algoritmo non terminerebbe.

Generalità: **ok**. La quantità n è generica e a meno che non sia grande tanto da non poter essere contenuto in memoria, l'algoritmo funzione per ogni n.

Non ambiguità: **ok.** Però se togliessi il passo 2, non sarebbero definiti s e i al passo 3.

Correttezza: cosa succede se al passo 3 ci fosse ≥ invece di >? L'algoritmo non sarebbe corretto.
Come fare per correggerlo mantenendo il ≥?

Provare a riscrivere l'algoritmo utilizzando al passo 3 il controllo con il < e il ≤.

Un altro esempio: il Massino Comune Divisore (MCD) fra due numeri

Si ricorda il procedimento di Eulero:

MCD (a,b) = MCD (b,r)   se $r \neq 0$;
MCD (a,b) = b   se $r = 0$,

dove a,b sono numeri naturali con $a > b$ e $r = a \bmod b$ (resto della divisione di a con b).

Esempi:
MCD (105,90) = MCD (90,15) = MCD (15,0) = 15;
MCD (27,10) = MCD (10,7) = MCD (7,3) = MCD (3,1) = MCD (1,0) = 1.

Algoritmo:

1. inizio;
2. acquisisci i valori di a e b;
3. se b > a, scambia i valori di a e b;
4. calcola r (resto della divisione di a con b);
5. se r = 0, salta al passo 8;
6. sostituisci a con b e b con r;
7. torna al passo 4;
8. scrivi il risultato che è b;
9. fine.

Finitezza: **ok**. L'unico problema potrebbe verificarsi al passo 7 dove si dice di tornare al passo 4. Ma notare che r sicuramente assumerà il valore 0.

Non ambiguità: **ok.**

Generalità: **ok.** Notare che l'algoritmo funziona anche se a = b.

In genere negli algoritmi le istruzioni sono semplici:

· lettura e scrittura (acquisizione e stampa);
· operative (operazioni aritmetiche o di assegnamento);
· di controllo;
· di salto.

Dagli esempi risolta inoltre che lo stesso concetto può essere espresso in più modi:

· leggi, acqisisci, ...;
· stampa, scrivi, ...;
· vai a, salta a, ...;
· poni, assegna, ...;
· incrementa, somma, aggiungi, aumenta, ...

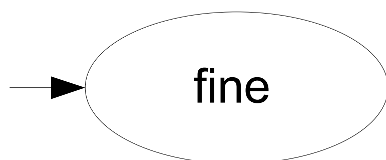
Si rende allora necessario uno pseudolinguaggio il più possibile uniforme per la scrittura degli algoritmi. Notare che non si parla per adesso di uno specifico linguaggio di programmazione.

Una possibile stesura degli algoritmi può essere realizzata tramite il *diagramma a blocchi* (o *diagramma di flusso* o *flow chart*).

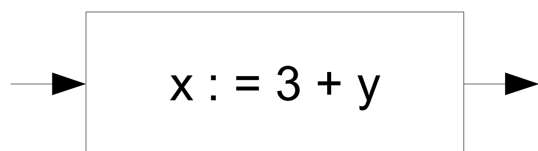
Nel diagramma le varie istruzioni dell'algoritmo sono rappresentate con simboli grafici, detti *blocchi elementari,* opportunamente collegati. Essi sono:

blocco di inizio

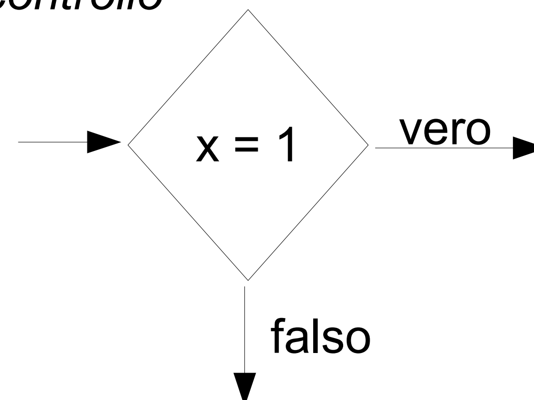
blocco di fine



blocco di elaborazione o azione



blocco di controllo



blocco di input/output

Proprietà e osservazioni

In un diagramma a blocchi sono presenti:

• un blocco di inizio e uno di fine;
• un numero finito di blocchi di elaborazione (o azione) e di input/output;
• un numero finito di blocchi di controllo.

Inoltre:

- ogni blocco di elaborazione (o azione) o di input/output ha una freccia in ingresso e una in uscita;
- ogni blocco di controllo ha una freccia in ingresso e due in uscita;
- ogni freccia entra in un blocco e si raccorda con un'altra freccia (escluso la freccia in uscita del blocco di inizio e quella in entrata del blocco di fine);
- ogni blocco è raggiungibile dal blocco di inizio;
- il blocco di fine è raggiungibile da ogni altro blocco.

Il blocco di controllo contiene una *condizione*.

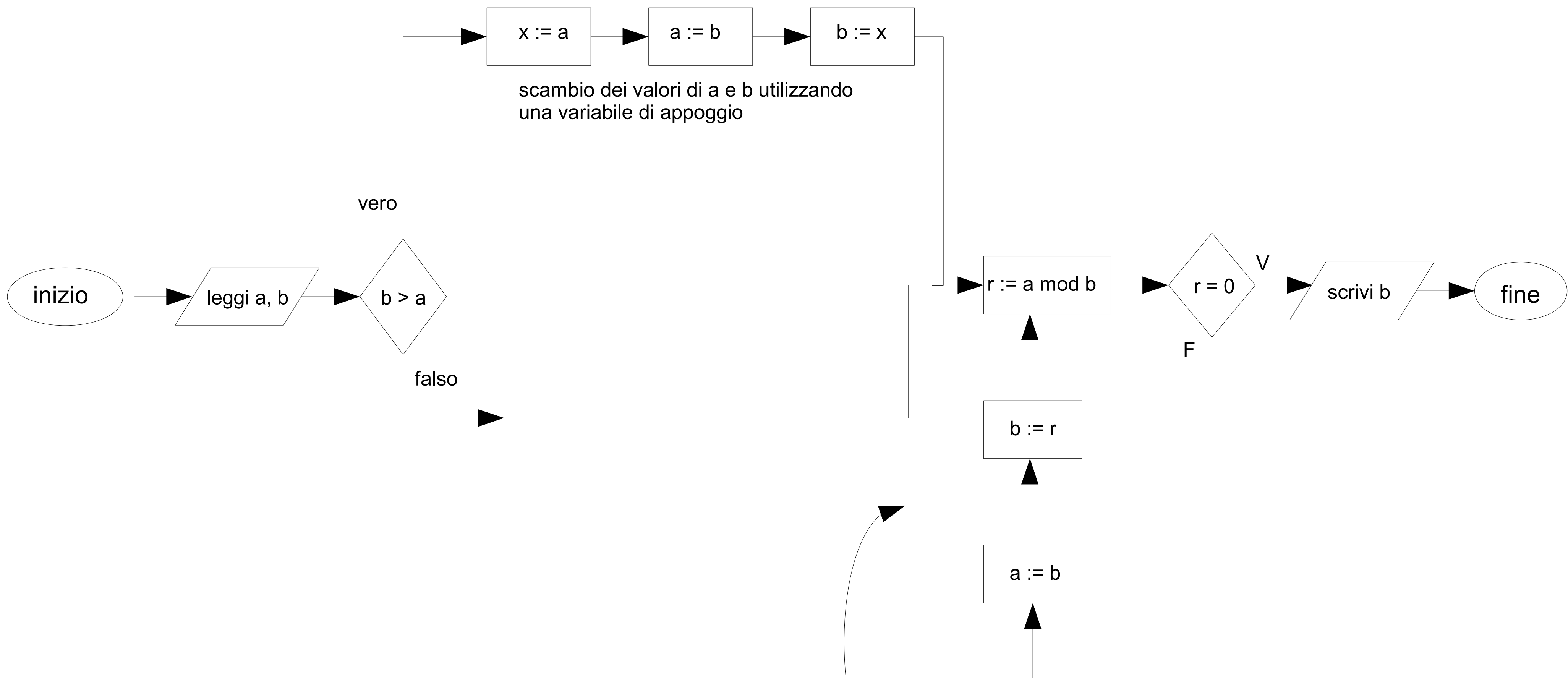
Def.

Una *condizione* è un predicato definito sulle variabili dell'algoritmo;

Un *predicato* è una proposizione in cui compaiono delle variabili;

Una *proposizione* è un costrutto linguistico del quale si può dire se è vero o falso.

Diagramma di flusso dell'algoritmo per l'MCD



Da notare l'ordine con cui si effettuano
gli assegnamenti a := b e b := r.

Possiamo affermare che la base di ogni linguaggio di programmazione è formata da **espressioni, comandi** e **dichiarazioni**

Un'**espressione** è un'entità sintattica la cui valutazione produce un valore oppure non termina e allora è indefinita.

Un'espressione può essere un'entità singola oppure è costituita da un operatore applicato ad un certo numero di operandi che a loro volta sono espressioni.

La notazione più usata per le espressioni è la *notazione infissa*, in cui l'operatore (binario) è posto fra le sue espressioni che rappresentano gli operandi. Si noti che per evitare ambiguità nell'applicare gli operatori agli operandi sono necessarie parentesi e opportune regole di precedenza, ben note.

Es. (a + b) * (c + d)

Nella *notazione prefissa* (o *polacca*, da Lukasiewicz) invece l'operatore precede gli operandi. In tale notazione non servono parentesi ma deve essere nota l'arietà degli operatori.

Es. *+ab+cd

Nella *notazione postfissa* (o *polacca inversa*) l'operatore invece segue gli operandi. Anche in tale notazione non servono parentesi a patto che si conosca l'arietà degli operatori.

Es. ab+cd+*

Un **comando** è un entità sintattica la cui valutazione non necessariamente restituisce un valore.

Un comando che abbiamo già usato negli esempi precedenti è l'**assegnamento** (il cui simbolo è :=). Esso è il comando base che permette di modificare il valore delle variabili.

Una **variabile** può essere vista come una locazione (di memoria) alla quale si può dare un nome e che può contenere dei valori, modificabili in seguito ad assegnamenti.

Nel particolare assegnamento x := x + 1 (l'effetto è quello di assegnare alla variabile x il suo valore incrementato di 1), il nome x a sinistra di := indica la locazione, mentre l'occorrenza di x a destra di := indica il suo valore. Questa distinzione importante si formalizza usando i termini *l-valore* (l = left) e *r-valore* (r = right). I primi indicano sostanzialmente la locazione mentre gli altri sono i valori che possono essere contenuti in essa.

L'assegnamento, in generale, può essere visto come un operatore binario la cui notazione in forma infissa è *exp1 := exp2.* Il suo significato è il seguente: calcola lo l-valore di exp1, determinando una locazione *loc;* calcola lo r-valore di exp2; modifica il contenuto di *loc* sostituendo il valore calcolato a quello precedente.

Prima di essere utilizzate le variabili devono essere dichiarate.

Una **dichiarazione** è un costrutto che permette di creare un'associazione fra un nome (ammesso) e una locazione (la cui dimensione dipende dal tipo di dichiarazione).

Scrivendo, ad esempio, *int x* si dichiara che x è una variabile di tipo intero. Sarà allocata una certa quantità di memoria, diversa da quella necessaria per la dichiarazione *float x.* Quest'ultima indica che x è una variabile (di tipo float) che può memorizzare numeri più grandi rispetto ad una variabile di tipo int. Inoltre una variabile di tipo float può contenere numeri con cifre dopo la virgola.

Una variabile deve essere dichiarata anche prima di un assegnamento che la riguarda:

```
int x;      // dichiarazione
x := 5;     // assegnamento
```

che può essere condensato in

```
int x := 5;
```

All'interno di un programma le istruzioni possono essere organizzate in **blocchi.**

Un **blocco** è una regione del programma, identificata con un inizio e una fine, che può contenere dichiarazioni locali a quella regione.

Facciamo un esempio:

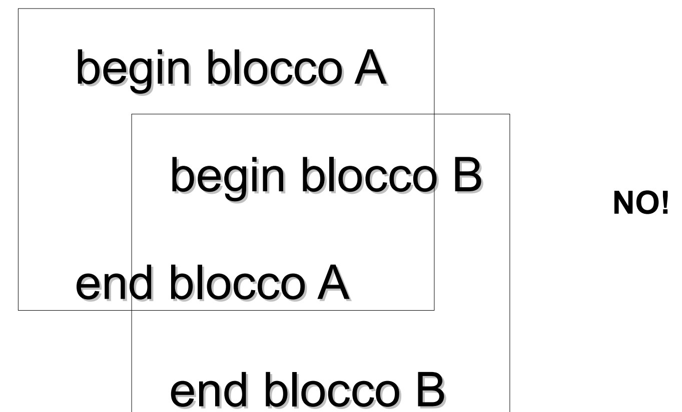
(per designare l'inizio di un blocco si usa *begin*, mentre per la sua fin *end*)

```
A  begin   int x := 2;

        B  begin   int y := 3;
                   y := y + 1;
           end

           x := x + 1;
           stampa x;
   end
```

Nell'esempio a fianco ci sono due *blocchi annidati* (blocco A e blocco B). In un programma si possono avere blocchi annidati o in sequenza. Non si può però verificare una situazione come in figura sotto:

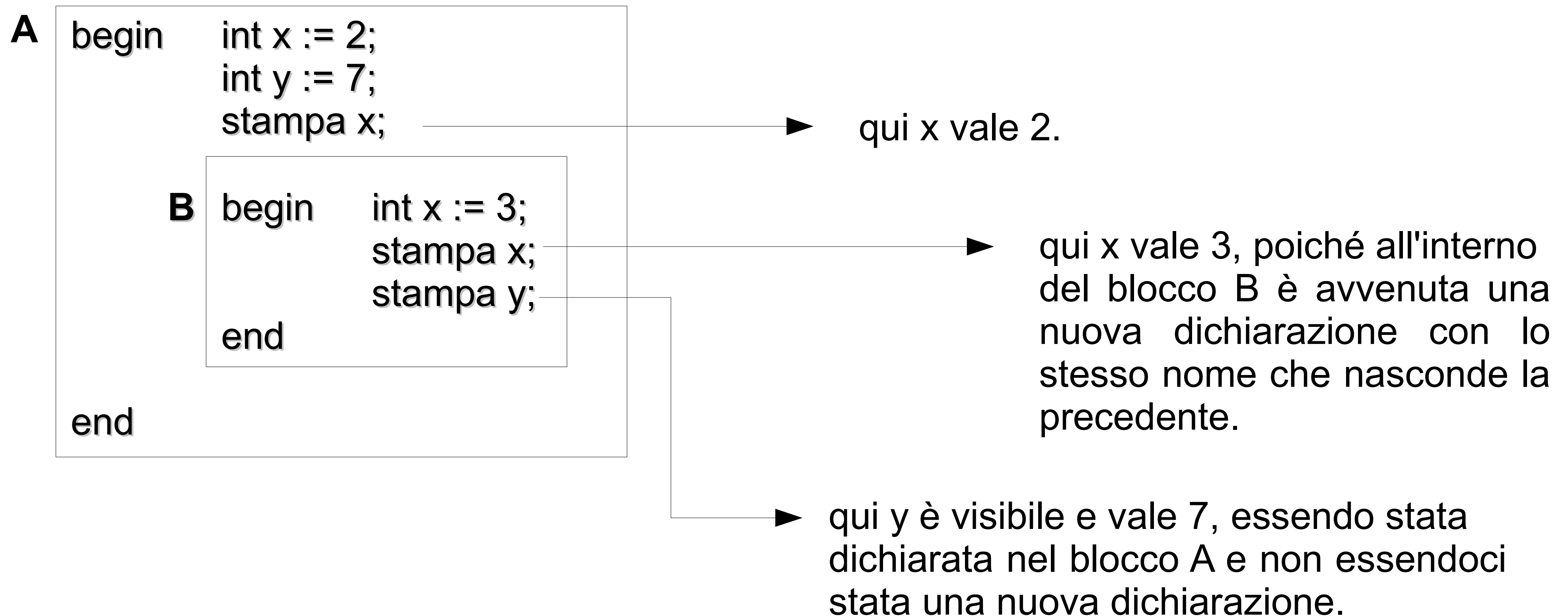


Non si può chiudere un blocco se quelli aperti al suo interno non sono già stati chiusi.

Regola di visibilità per i blocchi

Una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome. In tal caso, nel blocco in cui compare la ridefinizione la nuova dichiarazione nasconde la precedente.

Cosa significa?



In entrata al blocco:

- si creano le associazioni fra i nomi dichiarati localmente al blocco e i relativi oggetti denotati;
- sono disattivate le associazioni per quei nomi già esistenti all'esterno del blocco che siano ridefinite al suo interno.

In uscita dal blocco:

- sono distrutte le associazioni fra i nomi dichiarati localmente al blocco e i relativi oggetti denotati;
- si riattivano le associazioni per i nomi, già esistenti all'esterno del blocco, che erano stati ridefiniti al suo interno.

Esempio blocchi

```
A begin   int a := 1;

  B begin   int b := 2;
            int c := 2;

    C begin   int c:= 3;
              int d;
              d := a + b + c;
              stampa d;
      end

    D begin   int e;
              e := a + b + c;
              stampa e;
      end

    end

  end
```

Nel blocco **C** *a* e *b* sono visibili, la variabile *c* vale 3 (non 2) e *d* vale 6;

Dopo la chiusura del blocco **C** la variabile *d* non è più visibile. Se chiedessi di stampare *d* avrei un errore. Lo stesso per la variabile *e* dopo la chiusura del blocco **D**.

Nel blocco **D**, la variabile *c* vale 2, non 3. L'assegnamento c := 3 del blocco **C** viene disattivato alla chiusura di **C**, mentre viene riattivato quello effettuato all'apertura del blocco **B**. Nel blocco **D** la variabile *e* vale 5.

Le variabili *a* e *b* sono visibili nel blocco **D**.

Se nell'esempio precedente le istruzioni fossero organizzate
in un unico blocco:

```
begin   a := 1;
        b := 2;
        c := 2;
        c := 3;
        d := a + b + c;
        stampa d;
        e := a + b + c;
        stampa e;
end
```

la variabile *c*, dopo il suo secondo assegnamento, vale sempre 3; pertanto sia *d* sia *c* valgono 6.

L'introduzione dei blocchi conduce al concetto di *ambiente.*

Ambiente locale: costituito dalle associazioni per nomi dichiarati localmente ad un blocco;

Ambiente non locale: costituito dalle associazioni relative ai nomi che sono visibili all'interno di un blocco ma che non sono stati dichiarati localmente;

Ambiente globale: costituito dalle associazioni create all'inizio dell'esecuzione del programma. Contiene dunque le associazioni per i nomi che sono usabili in tutti i blocchi del programma.

Alla luce di quanto detto sopra, riprendendo l'*Esempio blocchi,* possiamo dire:

- L'ambiente di **B** è costituito dall'ambiente locale contenente l'associazione per i due nomi *b* e *c* e dall'ambiente globale contenente l'associazione per *a*.

- L'ambiente di **C** è costituito dall'ambiente locale contenente l'associazione per i due nomi *c* e *d* e dall'ambiente non locale contenente lo stesso ambiente globale di prima e anche l'associazione per il nome *b* che è ereditata dall'ambiente del blocco **B**.

- L'ambiente di **D** è costituito dall'ambiente locale contenente l'associazione per *e*, il solito ambiente globale e quello non locale che contiene l'associazione per *a, b* e *c.*

L'assegnamento costituisce un comando base per un linguaggio di programmazione. Ci sono comunque altri comandi. Molto importanti e fondamentali sono i *comandi per il controllo di sequenza*.

Possiamo affermare che essi possono essere divisi in:

1) comandi per il controllo di sequenza esplicito;
2) comandi condizionali (o di selezione);
3) comandi iterativi.

1) Comandi per il controllo di sequenza esplicito

- *comando sequenziale*. E' indicato con un ; (punto e virgola).

Es.

C1; C2; … ; Cn;

Usato per separare comandi. Un comando inizia quando il precedente è concluso. Può essere visto come un operatore di concatenazione fra comandi. Notare che in qualche linguaggio (C, C++, Java) esso è un terminatore di comandi, ossia il ; è sempre richiesto alla fine di un comando anche se questo non è seguito da nessun altro comando. L'ultimo comando di un blocco, ad esempio, in linguaggio C, è sempre seguito dal punto e virgola.

- *comando composto*.

```
begin
…
end
```

Con il comando composto è possibile raggruppare una serie di comandi. Con esso si delimita un blocco di comandi.

- *comando con etichetta.* Serve per etichettare un particolare comando.

Es.

label: C   label è un'etichetta e C un comando.

- *goto.* E' un'istruzione di salto.

Es.

goto A   trasferisce il controllo al punto in cui è presente l'etichetta A.

```
begin
        int i:=1;
        salto: stampa i;
        i := i + 1;
        se i < 10 goto salto;
end
```

Il goto appare un comando naturale ma:

- non è essenziale per l'espressività di un linguaggio: il teorema di Böhm – Jacopini assicura che un programma che usi il goto può essere tradotto in un programma equivalente che non usi il goto;
- usare il goto può rendere il codice incomprensibile, poiché permette di effettuare salti <u>incondizionati</u> da una parte all'altra del programma.
Per tali motivi il goto non è quasi più usato

Vedremo altre istruzioni di salto: break, continue, return.

2) Comandi condizionali (o di selezione)

- *if* Si usa nella forma:

  if Bexp then C1 else C2,

  dove Bexp è un'espressione booleana, C1 e C2 cono comandi. Se Bexp è vera allora si esegue C1, altrimenti C2.

  Si può trovare anche senza else:

  if Bexp then C1.

  In questo caso se Bexp è falsa non si esegue C1 ma si passa al comando successivo all'if.

- *if annidati* if Bexp1 if Bexp2 then C1 else C2

  Solitamente il ramo else si riferisce allo if più interno:

```
if Bexp1
    begin
        if Bexp2 then C1
        else C2
    end
```

- *else - if* (if a più rami) E' un alternativa agli if annidati:

<u>if</u> Bexp1 <u>then</u> C1

<u>else if</u> Bexp2 <u>then</u> C2

<u>else if</u> Bexp3 <u>then</u> C3

…

<u>else if</u> Bexpn <u>then</u> Cn

<u>else</u> Cn+1

Le espressioni si valutano in ordine. Se una risulta vera si esegue il comando associato e si termina la catena. L'ultimo <u>else</u> gestisce il caso “nessuno dei precedenti” e èuò essere omesso.

- *case* E' una specializzazione dell'if a più rami.

case Exp of
label1: C1
label2: C2
…
labeln: Cn
else Cn+1 (si può omettere)

L'espressione Exp produce un risultato di tipo compatibile con quello delle etichette label1, label2, ..., labeln che sono delle costanti.
Si valuta Exp e si esegue il comando corrispondente all'etichetta con lo stesso valore di Exp se essa è presente, altrimenti si esegue il comando Cn+1. Se il ramo else non fosse presente e nessuna etichetta ha il valore di Exp, si esegue il comando successivo al case.

Il comando *switch* in linguaggio C

In linguaggio C il comando case prendo il nome switch. La sua sintassi è la seguente:

```
switch (Exp) {
      case label1: C1;
      case label2: C2;
      …
      case labeli: Ci;
      ...
      case labeln: Cn;
      default: Cn+1
}
```

In linguaggio C le parentesi graffe delimitano un blocco di istruzioni

- si valuta Exp e il controllo passa al comando con etichetta che coincide con Exp. Se non ci sono etichette Exp, si esegue il comando con etichetta default. Se non c'è etichetta default si esce dallo switch e si esegue il comando subito seguente.

- una volta entrati in un ramo, il controllo "fluisce" nei rami successivi, compreso defalut. Per eseguire solo il comando con etichetta Exp bisogna scrivere *break* dopo il comando stesso. Allora, eseguito il comando, si esce dallo switch senza eseguire i comandi seguenti a quello eseguito. L'istruzione *break* è un istruzione di salto.

3) Comandi iterativi

Non usare il comando goto limita molto le possibilità di un linguaggio di programmazione. Ci sono però due meccanismi che assicurano il potere espressivo. l'iterazione strutturata e la ricorsione. L'iterazione strutturata può essere determinata o indeterminata.

- *iterazione indeterminata*

  - ciclo *while*

    while Bexp do C

    1. valutare l'espressione booleana Bexp;
    2. se Bexp è vera eseguire il comando C e tornare al passo 1, altrimenti il ciclo termina.

  - ciclo *until repeat*

    repeat C until Bexp

    1. eseguire il comando C;
    2. valutare Bexp: se Bexp è falsa torna al passo 1, altrimenti il ciclo termina.

Notare che

- nel ciclo *until repeat* il comando C viene eseguito almeno una volta;
- il ciclo *unitl repeat* equivale a:

```
C;
while (not Bexp) do C;
```

(not Bexp è la negazione di Bexp)

- *iterazione determinata*

  - ciclo *for* Generalmente è usato nella forma:

for I = **inizio** to **fine** by **passo** do

C

I è l'indice o contatore o variabile di controllo, **inizio** e **fine** sono espressioni di tipo intero, **passo** è una costante.

1. valutare le espressioni **inizio** e **fine** e memorizzare i risultati;
2. inizializzare I con il valore di **inizio**;
3. se I > **fine**, il ciclo termina;
4. eseguire il comando C e incrementare I del valore di **passo**;
5. tornare al passo 3.

Se **passo** è negativo il test al passo 3 è “se I < fine”.

Notare che:

- è sconsigliato modificare la variabile di controllo nel corpo del ciclo (comando C);
- usato in questa forma il ciclo *for* ha sempre termine, a differenza dell'iterazione indeterminata (in linguaggio C ciò può essere non vero);
- nella forma generalmente usata il ciclo *for* può essere sostituito con uno *while*:

```
for i = 1 to n by k do
        C1
```

```
i := 1;
while i ≤ n do
      begin     C1;
                i := i+k;
      end
```

Le istruzioni di salto *break* e *continue* nei cicli (istuzioni di salto).

- usato in un ciclo il *break* trasferisce il controllo in un punto immediatamente successivo alla fine del ciclo.

Es. (stabilire se un numero n è primo)

```
leggi n;
for d=2 to n-1 (by 1) do
      if n mod d= 0 then break;
if d<n then
      stampa “n è divisibile per d”;
else
      stampa “n è primo”
```

Appena viene trovato un divisore di n, non importa terminare il ciclo for ma esso viene interrotto con il break. Notare che se il ciclo for non viene interrotto, alla sua uscita la variabile d vale n.

- il *break* in genere interrompe il ciclo più interno; elude un solo livello di annidamento:

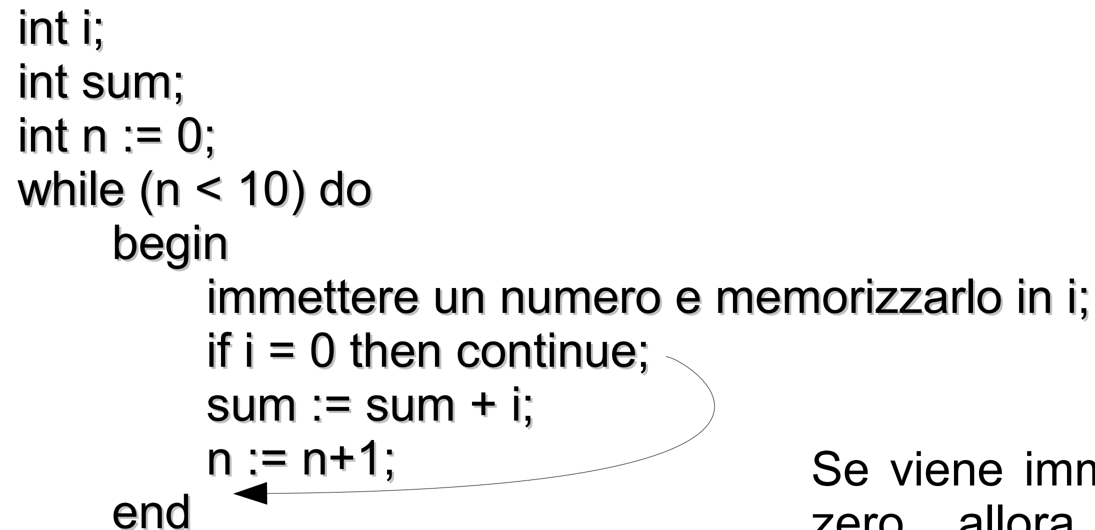
Es.

```
while P do
      begin
            for i=1 to n do
                  begin
                  ....        break
                  end
      end
```

Nell'esempio accanto, l'istruzione break fa uscire dal ciclo for ma non dallo while.

- l'istruzione *continue* trasferisce il controllo in un punto immediatamente precedente al corpo del ciclo:

Es. (leggere 10 numeri diversi da zero e sommarli)



Se viene immesso un numero uguale a zero, allora il continue fa terminare l'iterazione corrente e si inizia quella seguente.

Riassumendo:

- *break* fa uscire dal ciclo più interno in cui si trova;
- *continue* termina l'iterazione corrente e forza l'inizio di quella successiva.

Pseudocodice utilizzato nel corso.

1. Comando di *assegnamento*, della forma “V := E”, dove V è una variabile ed E è un'espressione.
Tale comando assegna alla variabile il valore dell’espressione;

2. Comando if then else, della forma “if P then C1 else C2”, dove P è una condizione, C1 e C2 sono comandi. L’effetto è quello di eseguire C1 se nello stato di calcolo P è vera, altrimenti quello di eseguire C2.

3. Comando *composto*, della forma “begin C1 C2 . . .Cm end” dove C1,C2, . . . ,Cm sono comandi. L’effetto è quello di applicare i comandi C1, C2, . . . , Cm nell’ordine.

4. Comando for, della forma “for i=1 to n do C”, dove *i* è una variabile intera e C è un comando.
Invece *n* è una variabile intera il cui valore è positivo e non viene modificato da C. L’effetto è quello di eseguire C ripetutamente per *n* volte, una per ciascun valore 1, 2, . . . , *n* di *i*.

5. Comando while, della forma “while P do C”, dove P è una condizione e C un comando. Se la condizione P è vera, il comando C viene eseguito; questo viene ripetuto finché la condizione diventa falsa. Nota che la condizione viene sempre valutata una volta in più rispetto al numero di esecuzioni di C.

6. Comando repeat, della forma “repeat C1 C2 . . .Cm until P”, dove P è una condizione e C1, C2, . . . , Cm sono m comandi. La sequenza di comandi C1, C2, . . . , Cm viene eseguita ripetutamente fino a quando la condizione P risulta vera; la condizione P viene verificata sempre al termine di ogni esecuzione del ciclo. Nota che la sequenza di comandi viene eseguita almeno una volta e che P è una condizione di uscita dal ciclo.

7. Comando *“con etichetta”*, della forma “e: C” dove e è un'etichetta e C un comando. Permette di rappresentare univocamente un comando specifico all’interno di un programma.

8. Comando goto, del tipo “goto e”, dove e è un'etichetta. Il suo effetto è quello di rimandare all’esecuzione del comando con etichetta e.

In un problema di una certa complessità possono esserci delle componenti ognuna delle quali concorre a fornire la soluzione globale. Conviene allora scomporre il problema generale per gestire meglio la complessità. La soluzione si ottiene componendo in modo opportuno le soluzioni ai sottoproblemi.

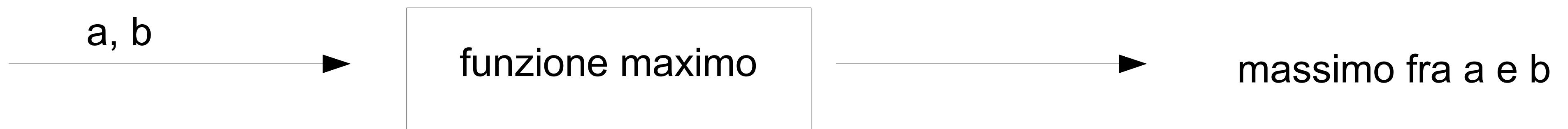
ES. trovare il massino fra tre numeri.
Un possibile algoritmo potrebbe essere quello a fianco:

```
1. leggi a, b, c; (i tre numeri)
2. int MAX;
3. if a > b begin
        if b > c
            MAX := a;
        else begin
            if a > c
                MAX := a;
            else MAX := c;
            end
   end
4. else begin
      if b< c
          MAX := c;
      else
          MAX := b;
   end
5. stampa MAX;
6. Fine
```

Ma se ci fosse a disposizione una funzione, che chiamiamo maximo, che riceve in ingresso due numeri e fornisce in uscita il massimo fra i due numeri, l'algoritmo risulterebbe alquanto più snello (pagina seguente):

1. leggi a, b, c;
2. int RIS, A;
3. A = maximo (a,b);
4. RIS = maximo (A,c);
5. stampa A;
6. Fine.

La funzione maximo riceve in ingresso due numeri e restituisce il massimo.



Il modo con cui la funzione maximo calcola il massimo fra due numeri non viene specificato all'interno dell'algoritmo principale. Deve comunque essere esplicitato quando si definisce la funzione.

Ci vuole un supporto linguistico che faciliti e renda possibile tale suddivisione in sottoproblemi. Esso si chiama **funzione**. E' una porzione di codice con un nome e capace di scambiare informazioni con il resto del codice mediante parametri. Ciò si traduce in

- definizione della funzione (dichiarazione di una funzione);
- uso della funzione (chiamata di una funzione).

La funzione scambia informazioni col resto del codice con:

- Parametri:
    - formali (nella definizione);
    - attuali (nella chiamata);

- Valore di ritorno: è il risultato della funzione. Si ottiene con l'istruzione **return** che termina l'esecuzione della funzione corrente e restituisce il controllo al punto che segue la chiamata della funzione (l'istruzione return è un'istruzione di salto).

L'esempio del massimo fra tre numeri rivisto:

```
leggi a, b, c;
int RIS, A;
A = maximo (a,b);
RIS = maximo (A,c);
stampa A;
int maximo (int x, int y)
    begin
        if x > y
            return x;
        else
            return y;
    end
```

Nella definizione della funzione si riconosce il nome (maximo), il tipo del valore di ritorno (int prima di maximo), parametri formali (int x, int y) e il corpo delimitato da begin e end. Si nota anche che costituisce una parte di codice all'interno di tutto il programma.

Quando comincia l'esecuzione del programma, arrivati alla terza riga, vine chiamata la funzione maximo con parametri attuali a e b (che in un caso reale devono essere numeri interi). Il controllo passa dunque alla funzione maximo (sesta riga): i parametri formali vengono sostituiti ordinatamente con quelli attuali e si eseguono i comandi all'interno del corpo della funzione. Non appena si incontra l'istruzione return, si interrompe l'esecuzione della funzione e il controllo torna al punto seguente la chiamata della funzione. Notare che nell'esempio sopra l'else nella dichiarazione della funzione può essere omesso.

Un altro esempio: la media fra due numeri.

```
int x, y, z;
leggi x, y, z;
float a, b, c; (float è un tipo per i numeri con cifre dopo la virgola)
a = average (x,y);
b = average (x,z);
c = average (y,z);
stampa a, b e c;
float average (int x, int y)
    begin
        return ( x + y ) / 2;
    end
```

Il corpo della funzione average è equivalente a:

```
begin
    int m=( x + y ) / 2;
    return m;
end
```

Esistono funzioni che non hanno un valore di ritorno. A volte esse vengono chiamate *procedure*. In linguaggio C, anche se non ci sono valori di ritorno, continuano a chiamarsi funzioni.

In linguaggio C, anche se una funzione non ha valore di ritorno, esso va comunque specificato con la parola riservata **void.**

Es.

```
void print_count (int m) {
        printf (“%d”,n)
}
```

printf è il comando per la stampa in linguaggio C. Per la sua sintassi si rimanda ad un manuale del linguaggio.

La funzione print_count non ha valore di ritorno e infatti non presenta l'istruzione return. Il suo effetto è quello di stampare il valore della variabile intera n.