Lezione 4 – Programmazione Strutturata

Programmazione

Modulo 2 - Programmazione e progettazione

Unità didattica 1 – Dalla programmazione non strutturata alla strutturata

Marco Anisetti

Università degli Studi di Milano - SSRI - CdL Online

Programmazione strutturata

Nasce dalla critica del costrutto del salto incondizionato, che rappresentava, negli anni sessanta, lo strumento fondamentale per la definizione di algoritmi complessi

Dijkstra fu tra i primi a evidenziare gli effetti deleteri dei salti incondizionati sulla leggibilità e modificabilità del software (spaghetti code)

Esistono delle strutture predefinite da usare per costruire il programma

Il flusso è controllato da queste strutture riducendo «i gradi di libertà» del programmatore

Programmazione strutturata (2)

Obiettivo: ottenere il più possibile un flusso ordinato di istruzioni dall'inizio alla fine (abolizione dei salti incondizionati)

Idealmente ottenibile considerando una sequenza lineare di operazioni, senza alternative

Nella realtà avviene attraverso regole che portano ad effetti equivalenti all'esecuzione sequenziale di operazioni

Programmazione strutturata (3)

Ogni struttura di controllo ha un solo punto di ingresso e un solo punto d'uscita

- Si parla di **blocco strutturato**

Un blocco è come se fosse un'unica istruzione

Un programma è **una sequenza di blocchi**

Il flusso di esecuzione è evidente dalla struttura del codice

- Flusso tra i blocchi

Programmazione strutturata (4)

Tutti i linguaggi di programmazione di alto livello moderni sono strutturati

La strutturazione del codice ha portato ad evidenti vantaggi sulla facilità di progettazione e di verifica del codice

Enfasi sulla struttura e sul flusso, per leggibilità/manutenibilità non per migliorare efficienza

Costrutto: dispositivo sintattico che permette di combinare tra loro istruzioni elementari creando blocchi strutturati

Costrutti fondamentali

[Sequenza]

Le istruzioni vengono eseguite una dopo l'altra a seconda dell'ordine in cui sono scritte

[Selezione]

L'esecuzione di un blocco di istruzioni viene scelta tra due possibili in base al valore di una condizione

[Iterazione]

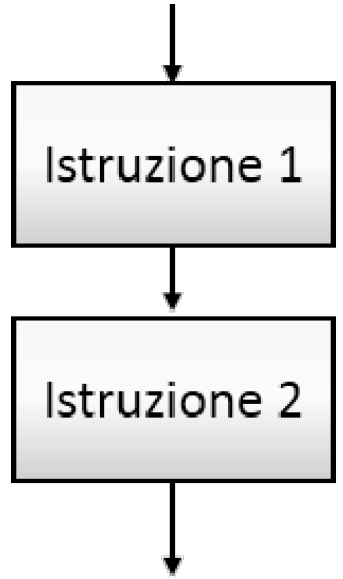
L'esecuzione di una o più istruzioni viene ripetuta in base al valore di una condizione

NOTA: non esiste più il salto come costrutto fondamentale

Sequenza

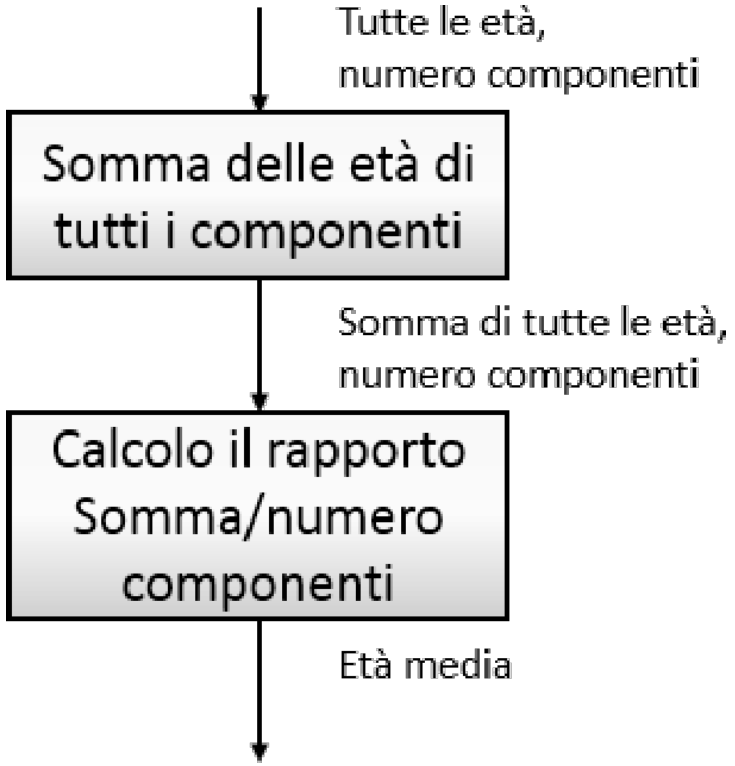
Le istruzioni sono eseguite nello stesso ordine in cui compaiono nel programma, cioè secondo la sequenza in cui sono scritte.

[Moltiplicazione tra due numeri]
1: read(x)
2: read(y)
3: write(*x*∗*y*)



Sequenza: esempio

Calcolo dell’età media dei componenti di un nucleo familiare



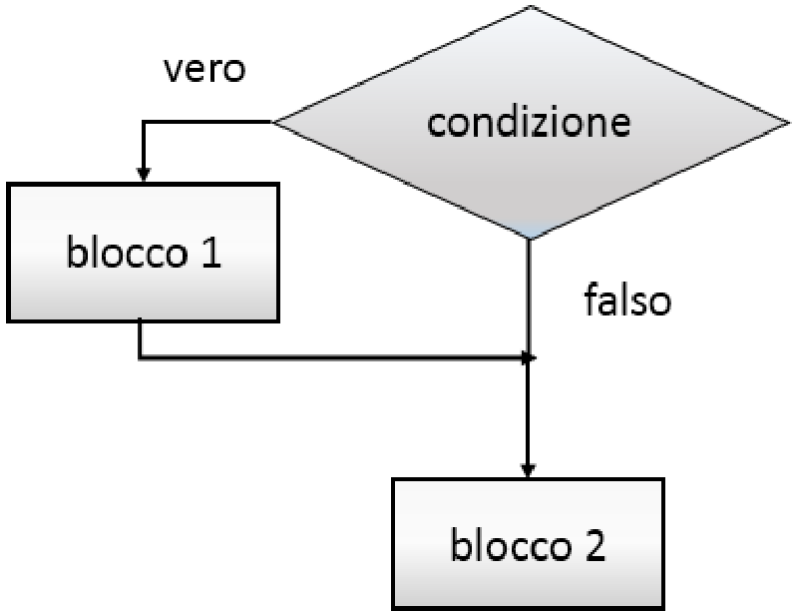
Selezione (if-then)(1)

[Sintassi generica]

if *condizione*
    then
        blocco 1
endif
blocco 2

[non strutturata]

if *condizione*
    then <goto blocco 1>
<goto blocco 2>
blocco 1
blocco 2



Selezione (if-then)(2)

[Esecuzione]

Valutazione della condizione:

VERA Vengono eseguite le istruzioni del blocco 1

In seguito si esegue blocco 2 essendo la prima istruzione che segue il costrutto

FALSA l'esecuzione prosegue direttamente dalla prima istruzione che segue il costrutto ovvero blocco 2

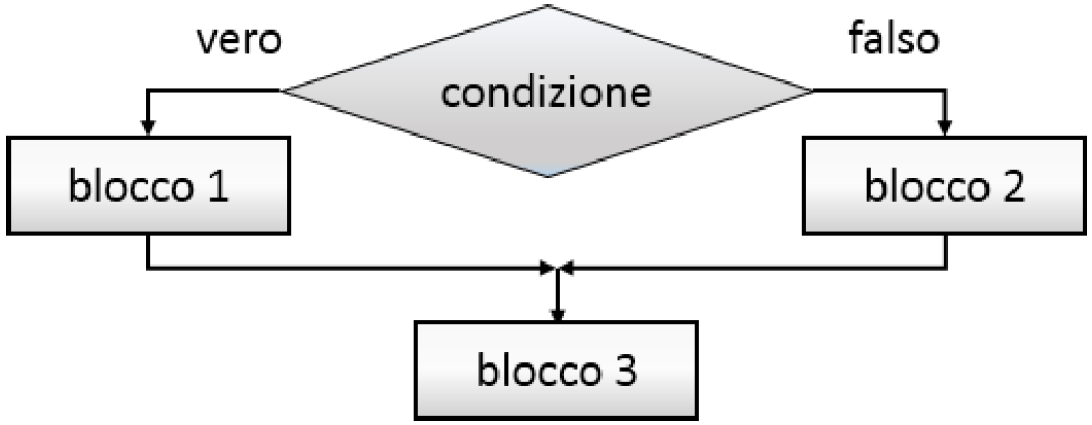
Selezione (if-then-else)(1)

[Sintassi generica]

if *condizione*
  then
    blocco 1
  else
    blocco 2
endif
blocco 3

[non strutturata]

if *condizione*
  then <goto blocco 1>
blocco 2
<goto blocco 3>
blocco 1
blocco 3



Selezione (if-then-else)(2)

[Esecuzione]

(1) Valutazione della *condizione*

VERA Vengono eseguite le istruzioni del blocco 1

FALSA Vengono eseguite quelle del blocco 2

(2) L'esecuzione procede con le istruzioni in blocco 3

Selezione (if-then-else)(3)

L'uso dell'else equivale ad un doppio uso dell'if-then:

[Esempio]

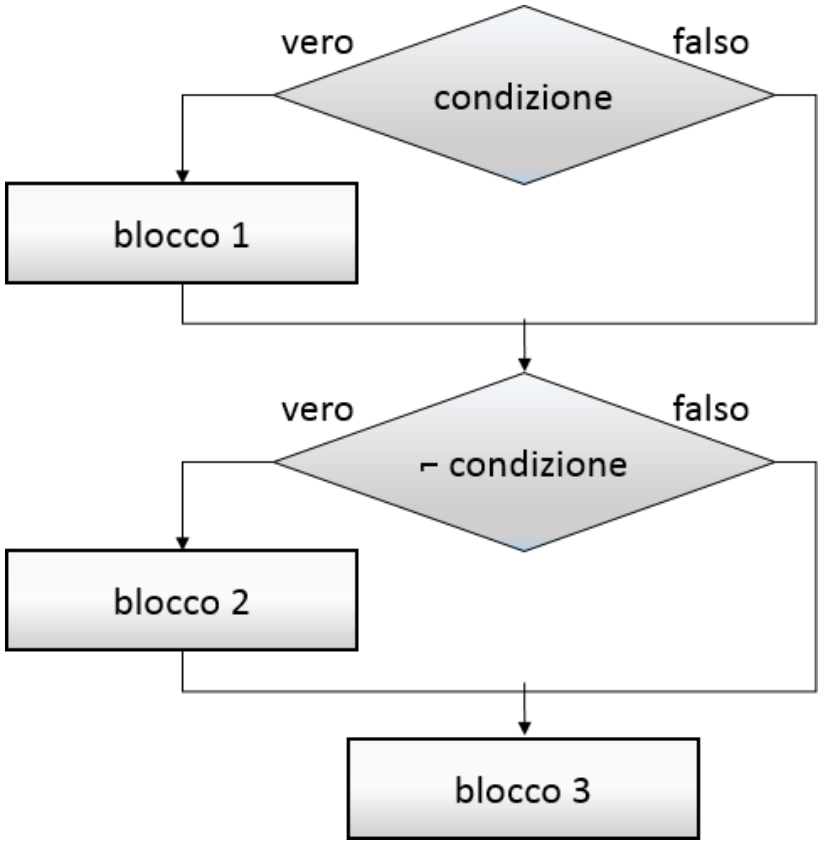
if *condizione*

then blocco 1

if *not condizione*

then blocco 2

blocco 3



Esempio(1)

Rivediamo l'esercizio del calcolo della divisione tra due numeri controllando che il divisore sia diverso da zero

1. read(dividendo)
2. read(divisore)
3. **if divisore ≠ 0 then goto 6**
4. **write("errore: divisione per zero")**
5. **goto 8**
6. **res <- dividendo/divisore**
7. **write(res)**
8. **;**

Esempio(1)

Rivediamo l'esercizio del calcolo della divisione tra due numeri controllando che il divisore sia diverso da zero

Togliamo i goto usando la selezione

```
if (divisore !=0) {
      res <- dividendo/divisore;
      write(res);
      }
   else
      write("errore: divisione per zero");
```

1. read(dividendo)
2. read(divisore)
3. if divisore ≠ 0 then goto 6
4. write("errore: divisione per zero")
5. goto 8
6. res <- dividendo/divisore
7. write(res)
8. ;

Esempio(1)

Rivediamo l'esercizio del calcolo della divisione tra due numeri controllando che il divisore sia diverso da zero

Togliamo i goto usando la selezione

```
if (divisore !=0) {
      res <- dividendo/divisore;
      write(res);
      }
   else
      write("errore: divisione per zero");
```

Le **{ }** definiscono un blocco: racchiudono istruzioni in sequenza

endif serviva per capire quando termina la struttura *if then* o *if then else* ma possiamo ometterlo

Iterazione (post-condizionato)(1)

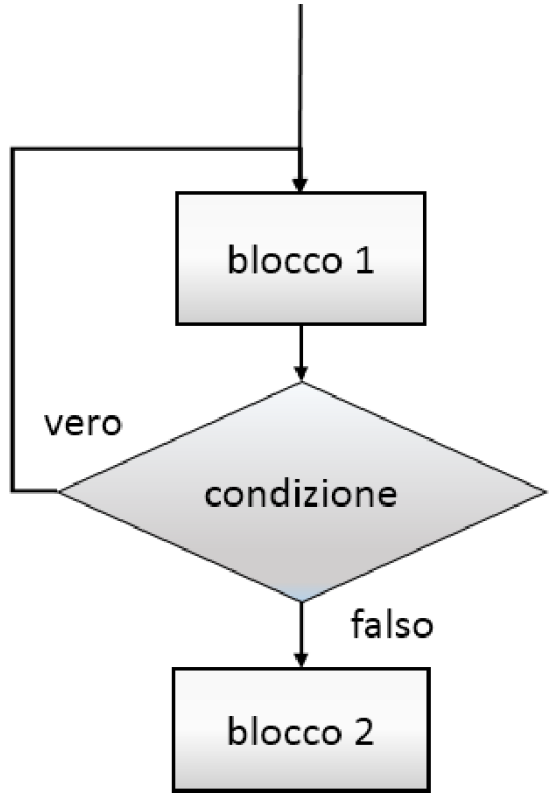
Chiamato anche ciclo repeat-until

[Sintassi]

```
do
   blocco 1
while condizione
blocco 2
```

[Non strutturato]

```
blocco 1
if condizione
then <goto blocco 1>
blocco 2
```



Iterazione (post-condizionato)(2)

[Esecuzione]

(1) Viene eseguito blocco 1

(1) Viene valutata condizione:

VERA Si ritorna al punto (1)

FALSA Si prosegue con la prima istruzione scritta dopo il costrutto iterativo

- Il blocco 1 è eseguito almeno una volta
- L'iterazione termina quando la condizione diventa falsa

Esercizio

Somma dei primi 1000 numeri interi senza utilizzare la formula di Gauss ($\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$), ma iterativamente

Esercizio: Soluzione

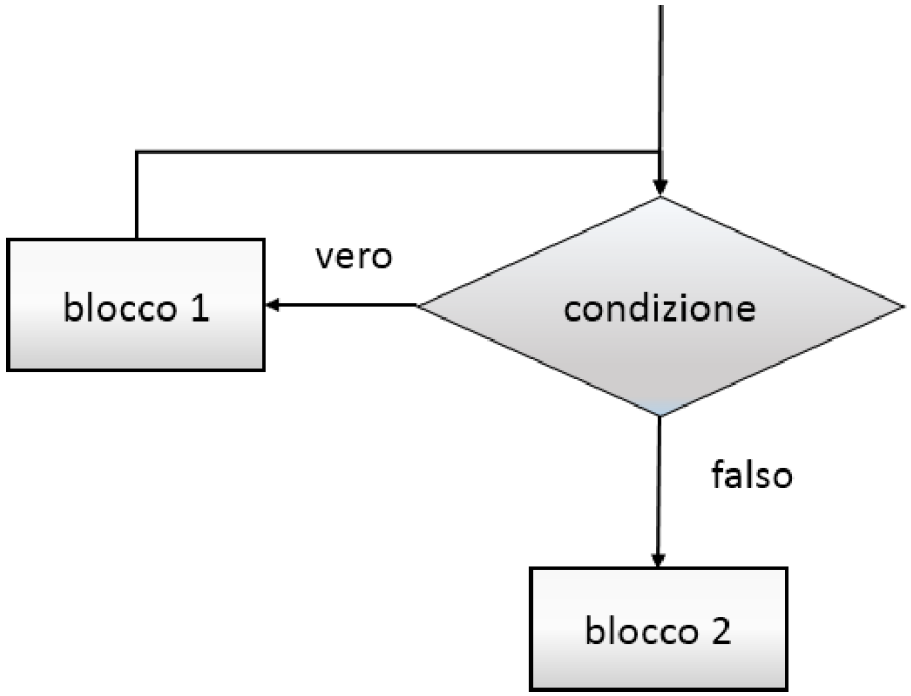
Somma dei primi 1000 numeri interi senza utilizzare la formula di Gauss ($\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$), ma iterativamente

```
somma =0;
n = 1;
do
   {
   somma = somma + n;
   n = n+1;
   }
while (n<= 1000)
write(somma);
```

Iterazione (pre-condizionato)(1)

[Sintassi]

while *condizione* do
    blocco 1
endwhile
    blocco 2



[Non strutturato]

etichetta:

if not condizione
then <goto blocco 2>
blocco 1
<goto etichetta>
blocco 2

Iterazione (pre-condizionato)(2)

[Esecuzione]

(1) Viene valutata la condizione:

VERA Viene eseguito blocco 1 quindi si torna al punto (1)

FALSA L'esecuzione riprende dalla prima l'istruzione che segue il costrutto iterativo

- Il blocco 1 può essere eseguito anche zero volte
- L'iterazione termina quando la condizione diventa falsa

Esempio iterazione

Il comportamento dell'iterazione pre-condizionata può essere simulato combinando un iteratore post-condizionato e una selezione:

```
if (condizione) {
      do
         blocco 1
      while (condizione)
      }
```

Esempio: rimozione dei doppi

Ora possiamo rivedere l'esempio sotto non strutturato e tradurlo usando le strutture che abbiamo visto

```
1: read(x);
2: if x=0 then goto 8;
3: writeln(x);
4: read(next);
5: if next=x then goto 4;
6: x <- next;
7: goto 2;
8: ;
```

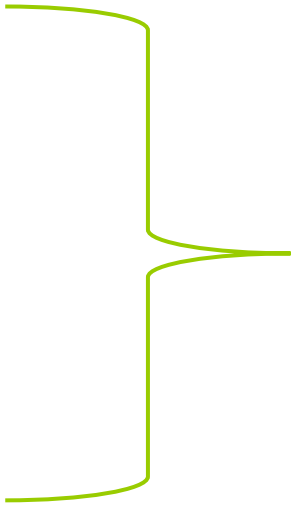
Esempio: rimozione dei doppioni

Ecco la versione strutturata

```
read(x);
while (x != 0) {




}
```



```
1.  read(x);
2.  if x=0 then goto 8;
3.  writeln(x);
4.  read(next);
5.  if next=x then goto 4;
6.  x <- next;
7.  goto 2;
8.  ;
```

Esempio: rimozione dei doppioni

Ecco la versione strutturata

```
read(x);
while (x != 0) {




}
```

condizione per restare nel ciclo

ometto ***do*** e ***endwhile***
uso il blocco ***{}*** per delimitare il corpo del while

```
1.  read(x);
2.  if x=0 then goto 8;
3.  writeln(x);
4.  read(next);
5.  if next=x then goto 4;
6.  x <- next;
7.  goto 2;
8.  ;
```

Esempio: rimozione dei doppioni

Ecco la versione strutturata

```
read(x);
while (x != 0) {
   write(x);



   x = next;
}
```

```
1.  read(x);
2.  if x=0 then goto 8;
3.  writeln(x);
4.  read(next);
5.  if next=x then goto 4;
6.  x <- next;
7.  goto 2;
8.  ;
```

Esempio: rimozione dei doppioni

Ecco la versione strutturata

```
read(x);
while (x != 0) {
   write(x);
   do
      read(next);
   while x==next
   x = next;
}
```

```
1.  read(x);
2.  if x=0 then goto 8;
3.  writeln(x);
4.  read(next);
5.  if next=x then goto 4;
6.  x <- next;
7.  goto 2;
8.  ;
```

Lezione 5 – Eliminazione dei salti

Programmazione

Modulo 2 - Programmazione e progettazione

Unità didattica 1 – Dalla programmazione non strutturata alla strutturata

Marco Anisetti

Università degli Studi di Milano - SSRI - CdL Online

Programma proprio (1)

Un programma si definisce proprio se gode delle seguenti proprietà

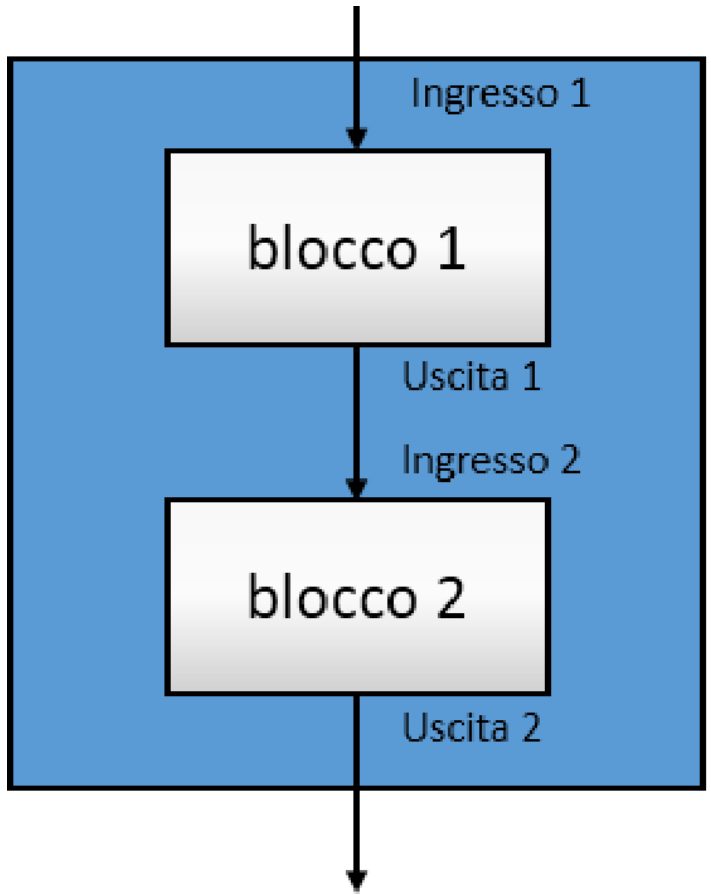
- Ha un unico ingresso ed un'unica uscita
- Ogni elemento appartiene ad un percorso che porta dall'ingresso all'uscita

L'ingresso è l'unica via di accesso al programma e l'uscita è l'unica via di terminazione del programma

Il programma proprio può essere suddiviso in blocchi

I blocchi sono strutture proprie (godono delle stesse due proprietà dei programmi propri)

Programma proprio (2)



Programma proprio

Programmi equivalenti

Due programmi che realizzano lo stesso algoritmo usando operazioni differenti si dicono equivalenti

In modo formale due programmi propri P e Q sono equivalenti se: $P(X) = Y = Q(X)$

Ovvero se applicati agli stessi argomenti X forniscono lo stesso risultato Y

Teorema di Böhm Jacopini

Dato un programma proprio *P* è possibile costruire un programma strutturato *S*(*P*) equivalente a *P*

- Il teorema permise di dimostrare che la potenza di calcolo dei programmi strutturati non è inferiore a quella dei programmi che usano il goto (**completezza**)
- Il programma strutturato *S*(*P*) è costituito da combinazioni di tre oggetti logici fondamentali:

  Sequenza, Selezione, Iterazione

[Completezza]

Tutti i programmi esprimibili tramite istruzioni di salto (*goto*) o diagrammi di flusso (*flow-chart*) possono essere riscritti utilizzando esclusivamente le tre strutture di controllo fondamentali (eliminazione dei salti)

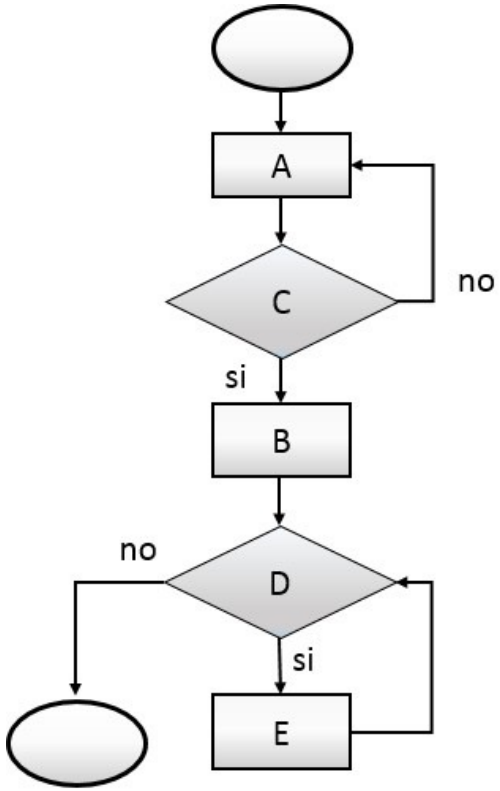
Dimostrazione Teorema di Böhm Jacopini

La dimostrazione del teorema porta a del codice strutturato in maniera da avere un unico ciclo (di tipo pre-condizionato) e una serie di variabili booleane di supporto per il mantenimento di un ordine preciso nelle delle istruzioni

Le variabili booleane sono a guardia di ogni operazione e permettono di eseguire una istruzione per ogni ciclo while in maniera simile ad un program counter

```
while ....
   if ... {istruzione 1}
.
.
   if ... {istruzione n}
```

Teorema di Böhm Jacopini: esempio



$b_A \leftarrow$ True
cond$\leftarrow$ True

while(cond)
{
if(b_A){A; $b_A \leftarrow$ False; $b_C \leftarrow$ True; }
if(b_C){if(C) $b_B \leftarrow$ True; else $b_A \leftarrow$ True;
$b_C \leftarrow$ False}

if(b_B){B; $b_B \leftarrow$ False; $b_D \leftarrow$ True;}

if(b_D){if(D) $b_E \leftarrow$ True; else cond $\leftarrow$ False
$b_d \leftarrow$ False}
if(b_E){E; $b_E \leftarrow$ False; $b_D \leftarrow$ True;}
}

Teorema di Böhm Jacopini: conclusioni

E’ sempre possibile scrivere un qualsiasi programma usando solo i costrutti della programmazione strutturata

Si possono eliminare i salti di un programma non strutturato ottenendo un programma strutturato a patto di inserire un certo numero di selezioni aggiuntive

Il risultato di questa trasformazione è un programma strutturato non meno intricato del programma originale ma leggibile

Esiste una trasformazione che dimostra il teorema e che porta a risultati più rilevanti

- Ashcroft e Manna

Eliminazione dei salti

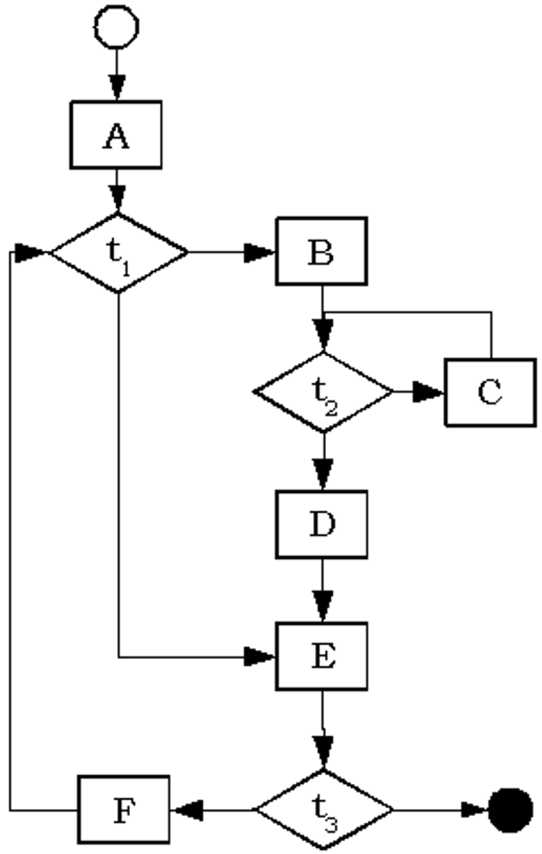
Vediamo anche una dimostrazione alternativa al teorema tramite la trasformazione di Ashcroft e Manna

La trasformazione di Ashcroft e Manna(1)

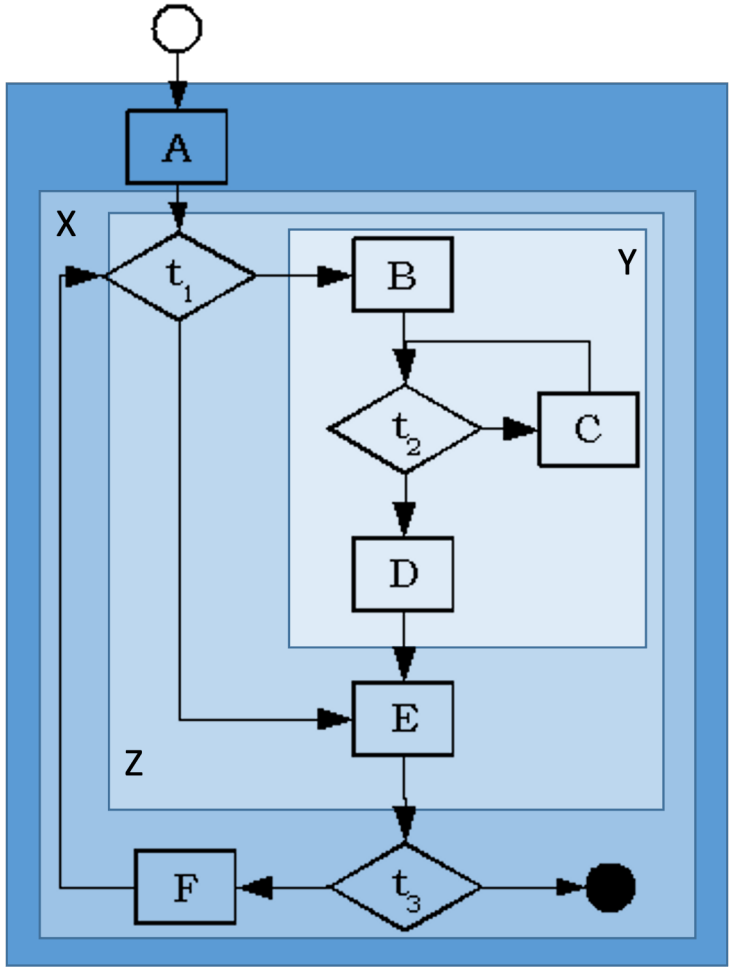
Tecnica costruttiva che sfrutta i concetti di programma proprio e di scomposizione gerarchica di un programma in sottoprogrammi propri

La dimostrazione sfrutta il diagramma di flusso come rappresentazione non considerando gli stati iniziale e finale e permette di preservare la struttura logica del programma

Esempio di trasformazione(1)

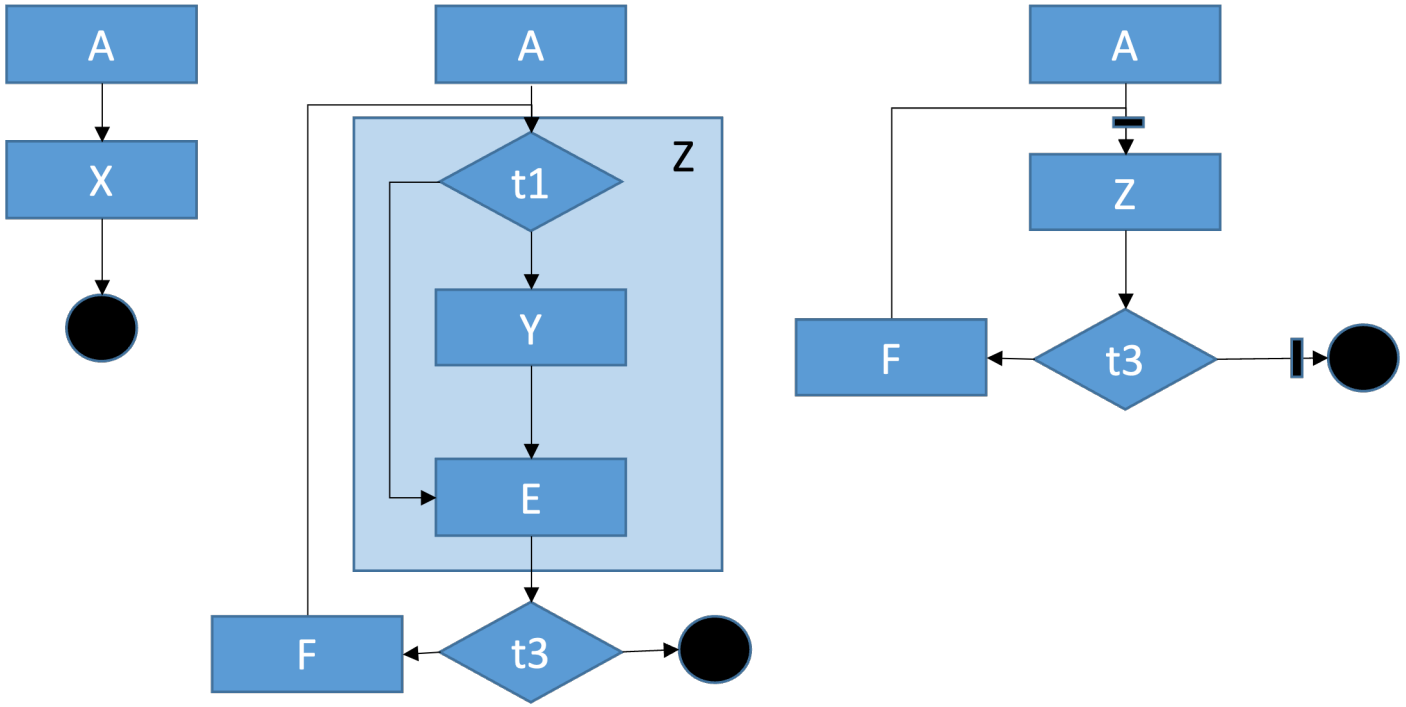


Esempio di trasformazione(2)



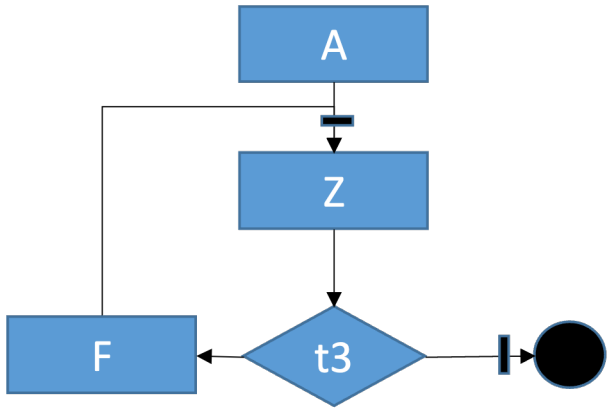
Ricerca programmi propri

Esempio di trasformazione(3)

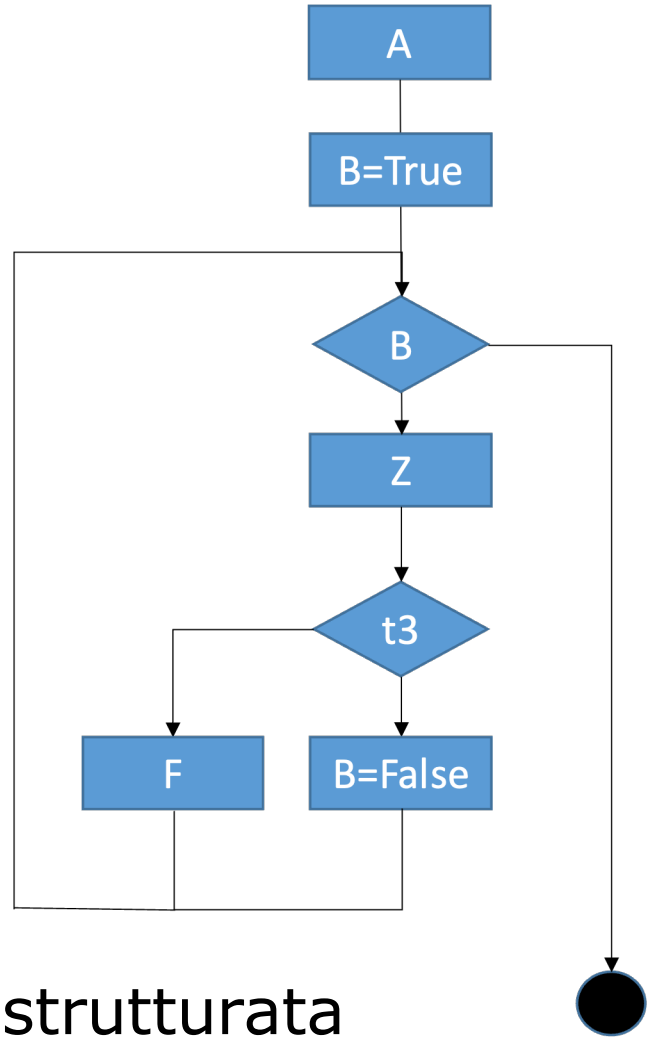


Ristrutturazione e indivituazione dei punti di taglio

Esempio di trasformazione(4)




```
A;
B= true;
while B
    Z;
    if t3 then B=false
          else   F
endwhile
```



Traduzione della parte non strutturata

Costrutti mini

In realtà i 3 costrutti ridotti ai soli sequenza e iterazione per codificare qualsiasi algoritmo

E' possibile rappresentare una selezione come un costrutto iterativo inserendo una variabile booleana in aggiunta.

```
if condizione
        blocco 1

b <- condizione
while b {
     blocco 1
     b <- false
     }
```

Strutture di controllo ed invarianti

Come abbiamo visto il programma evolve

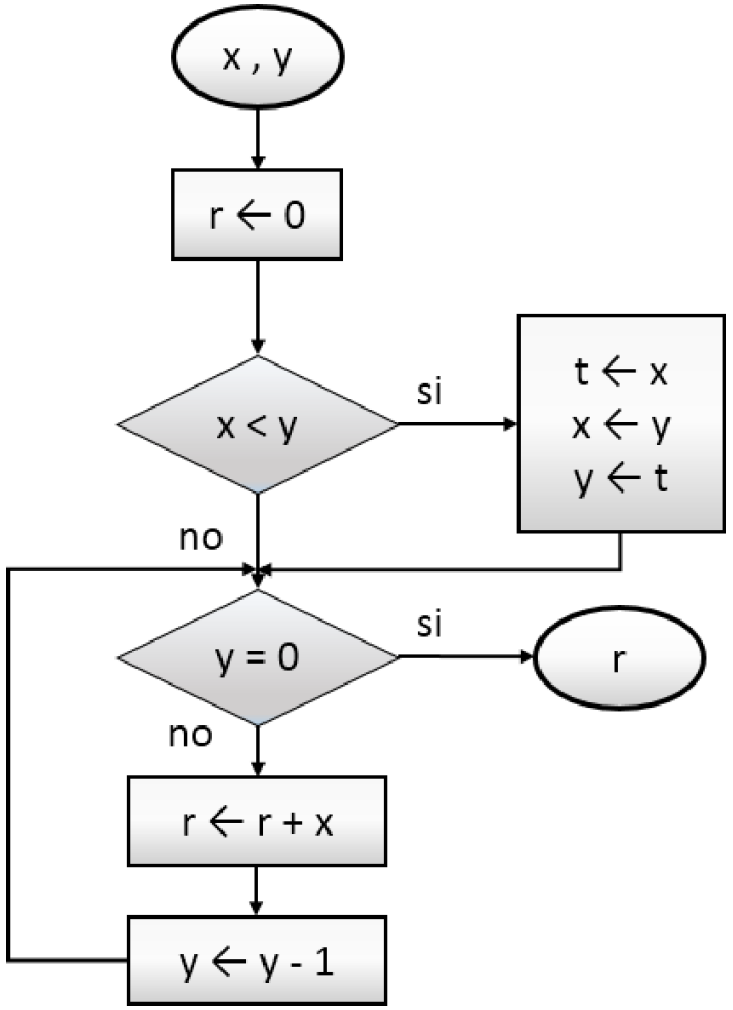
Una invariante è una asserzione vera in qualunque momento della esecuzione del programma

Una asserzione è una condizione vera o falsa circa lo stato della computazione (e.g. $x > y$)

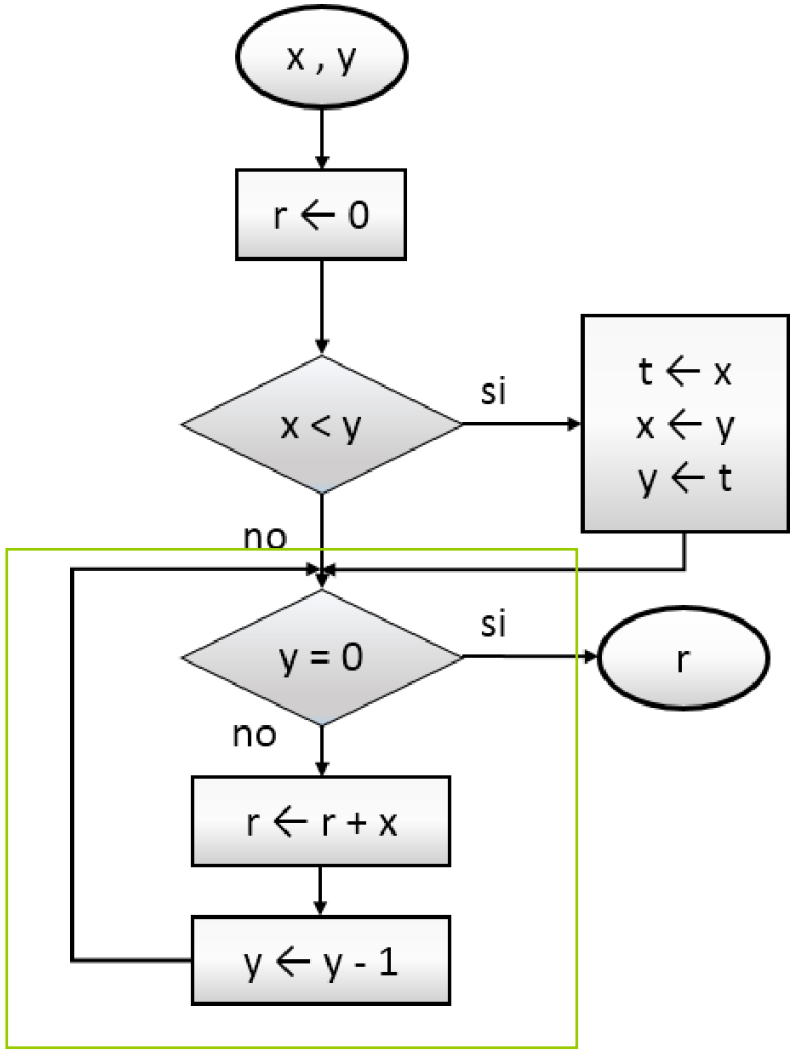
Problema: La correttezza del codice è una proprietà dinamica della computazione e non statica del codice scritto

Gli invarianti sono associati ad un punto del programma ma ci danno delle indicazioni circa le ***proprietà della sua computazione***

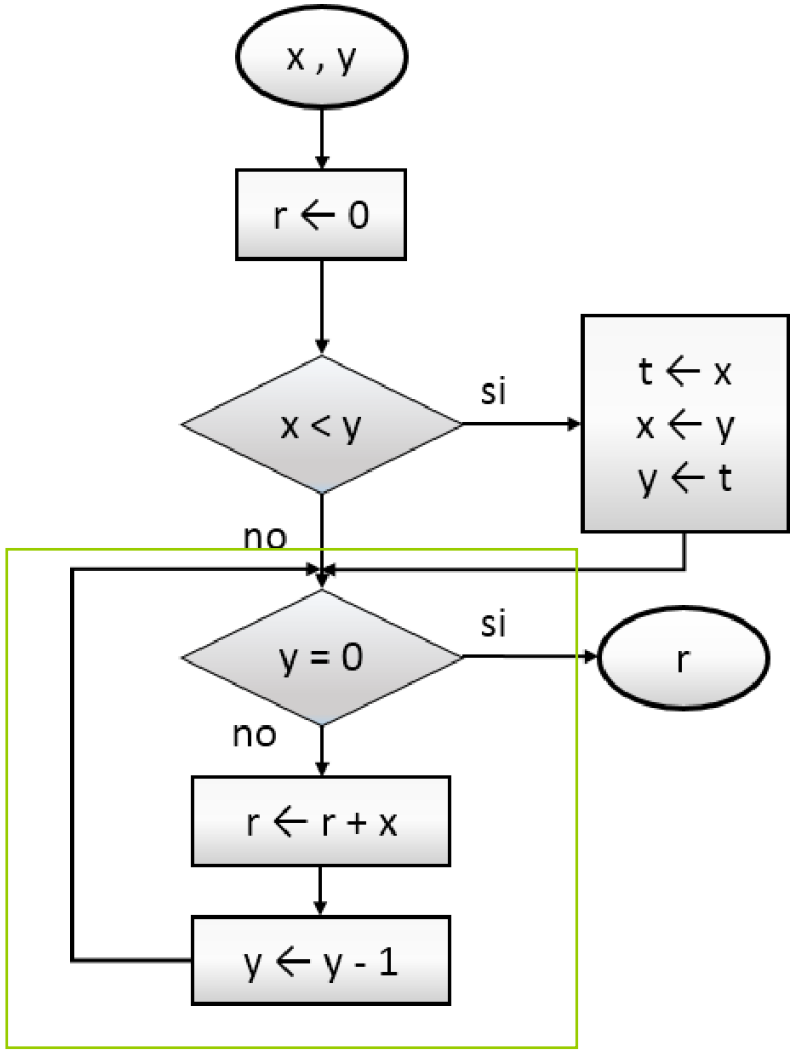
Invariante di ciclo



Invariante di ciclo

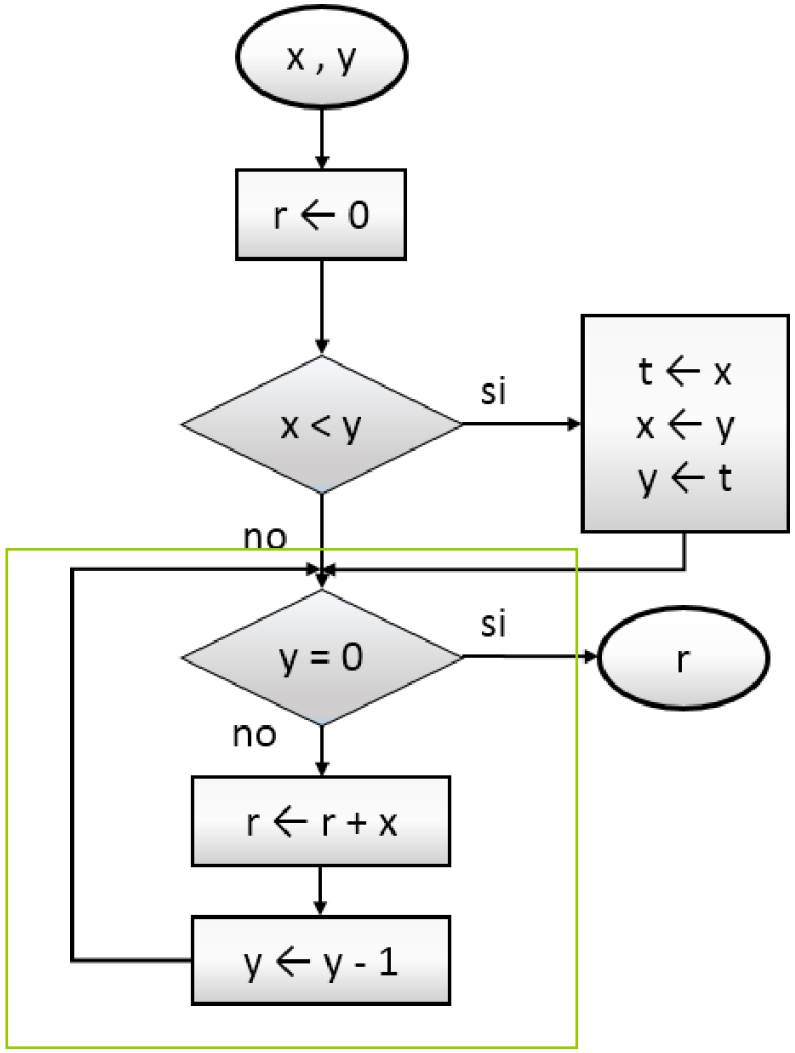


Invariante di ciclo



Invariante ?

Invariante di ciclo



Invariante ?

Dentro il ciclo sicuramente $y>0$

Turing equivalenza

Il Teorema di Jacopini-Bohm ci dice di conseguenza che ogni funzione calcolabile con una macchina di Turing è anche calcolabile con un programma scritto in un linguaggio che utilizzi i costrutti fondamentali

Implica che i linguaggi di programmazione sono Turing-equivalenti ovvero stessa potenza espressiva della macchina di Turing

Programmazione strutturata ed efficienza

Il fatto che un programma non strutturato possa essere riscritto in modo strutturato non implica che le due versioni del programma abbiano la stessa dimensione o la stessa efficienza

Esiste infatti un interessante teorema [*] che afferma che per ogni intero n, esiste un programma di $\bar{n}$ istruzioni che usa i salti, che non può essere convertito in un programma strutturato di meno di $1 * 3^{\sqrt{n}}$ istruzioni, a meno che tale programma non sia più lento di un fattore $\frac{1}{2} log_2 n$

[*] Ricard A. DeMillo, Stanley C. Eisenstat, and Richard J. Lipton. Spacetime trade-offs in structured programming: An improved combinatorial embedding theorem. Journal of the ACM, 1980.