UNIVERSITÀ DEGLI STUDI DELLA TUSCIA

INFORMATICA

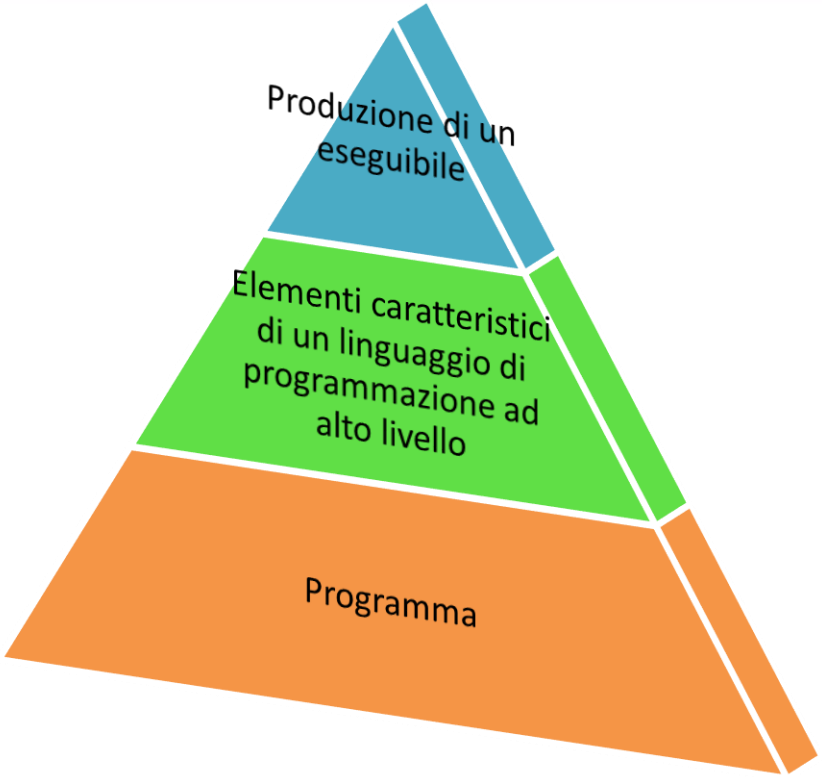
Programma

Dott. Franco Liberati
franco.liberati@unitus.it

PROGRAMMA

Argomenti della lezione

- Programma
  - Definizione
  - Diagramma di flusso
- Elementi caratteristici di un linguaggio di programmazione ad alto livello
- Produzione di un eseguibile



Algoritmo e Diagramma di flusso

PROGRAMMA

Ideazione e Stesura

- Il primo passo da compiere quando si deve realizzare un programma informatico è **la comprensione del problema, la sua formalizzazione, la definizione di quali operazioni compiere e l'ordine in cui devono essere svolte le istruzioni da impartire all'elaboratore**
- A questo è necessario aggiungere un approccio alla soluzione che preveda degli scenari di uso, dei controlli, delle verifiche e una documentazione completa ed esaustiva

PROGRAMMA

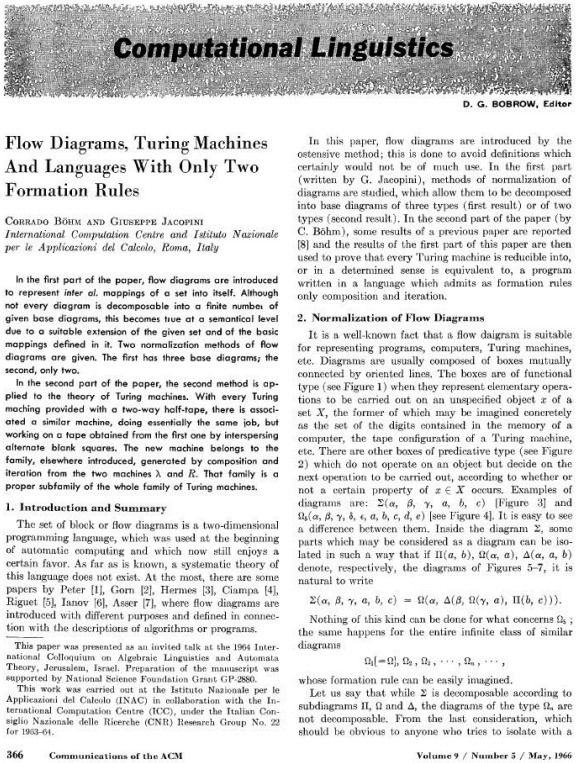
Algoritmo

- Una procedura per la risoluzione di un problema, nei termini di azioni da eseguire, e l'ordine in cui questi atti devono essere svolti realizzano un **algoritmo**
- Un algoritmo può essere espresso sfruttando uno **pseudocodice**, ovvero un linguaggio artificiale e informale che consente di riflettere su come risolvere il problema prima di scrivere il codice informatico. Lo pseudocodice consiste in istruzioni di azione e di decisioni nonché la definizione delle variabili e delle costanti
- Una formalizzazione di un problema utilizzando uno pseudocodice, se preparato con cura, può essere facilmente convertito in un programma

PROGRAMMA

Descrizione per strutture di controllo

- I lavori svolti da due professori italiani, Corrado Böhm e Giuseppe Jacopini, hanno dimostrato che **tutti i programmi possono essere scritti in termini di sole tre strutture di controllo**: sequenziale, selettiva e iterativa.
  - La **struttura sequenziale** è l'elencazione di istruzioni eseguite una dopo l'altra nell'ordine in cui sono scritte (il blocco)
  - La **struttura selettiva** consente l'esecuzione sequenziale di un gruppo di istruzioni rispetto ad un altro in accordo alla veridicità di una condizione
  - La **struttura iterativa** permette lo svolgimento sequenziale di un gruppo di istruzioni una o più volte fino al verificarsi di una condizione che ne stabilisca la terminazione e il passaggio a quanto scritto dopo



Computational Linguistics

D. G. BOBROW, Editor

Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules

CORRADO BÖHM AND GIUSEPPE JACOPINI
International Computation Centre and Istituto Nazionale per le Applicazioni del Calcolo, Roma, Italy

In the first part of the paper, flow diagrams are introduced to represent *inter al.* mappings of a set into itself. Although not every diagram is decomposable into a finite number of given base diagrams, this becomes true at a semantical level due to a suitable extension of the given set and of the basic mappings defined in it. Two normalization methods of flow diagrams are given. The first has three base diagrams; the second, only two.

In the second part of the paper, the second method is applied to the theory of Turing machines. With every Turing maching provided with a two-way half-tape, there is associated a similar machine, doing essentially the same job, but working on a tape obtained from the first one by interspersing alternate blank squares. The new machine belongs to the family, elsewhere introduced, generated by composition and iteration from the two machines λ and R. That family is a proper subfamily of the whole family of Turing machines.

1. Introduction and Summary

The set of block or flow diagrams is a two-dimensional programming language, which was used at the beginning of automatic computing and which now still enjoys a certain favor. As far as is known, a systematic theory of this language does not exist. At the most, there are some papers by Peter [1], Gorn [2], Hermes [3], Ciampa [4], Riguet [5], Ianov [6], Asser [7], where flow diagrams are introduced with different purposes and defined in connection with the descriptions of algorithms or programs.

This paper was presented as an invited talk at the 1964 International Colloquium on Algebraic Linguistics and Automata Theory, Jerusalem, Israel. Preparation of the manuscript was supported by National Science Foundation Grant GP-2880.

This work was carried out at the Istituto Nazionale per le Applicazioni del Calcolo (INAC) in collaboration with the International Computation Centre (ICC), under the Italian Consiglio Nazionale delle Ricerche (CNR) Research Group No. 22 for 1963–64.

In this paper, flow diagrams are introduced by the ostensive method; this is done to avoid definitions which certainly would not be of much use. In the first part (written by G. Jacopini), methods of normalization of diagrams are studied, which allow them to be decomposed into base diagrams of three types (first result) or of two types (second result). In the second part of the paper (by C. Böhm), some results of a previous paper are reported [8] and the results of the first part of this paper are then used to prove that every Turing machine is reducible into, or in a determined sense is equivalent to, a program written in a language which admits as formation rules only composition and iteration.

2. Normalization of Flow Diagrams

It is a well-known fact that a flow daigram is suitable for representing programs, computers, Turing machines, etc. Diagrams are usually composed of boxes mutually connected by oriented lines. The boxes are of functional type (see Figure 1) when they represent elementary operations to be carried out on an unspecified object x of a set X, the former of which may be imagined concretely as the set of the digits contained in the memory of a computer, the tape configuration of a Turing machine, etc. There are other boxes of predicative type (see Figure 2) which do not operate on an object but decide on the next operation to be carried out, according to whether or not a certain property of $x \in X$ occurs. Examples of diagrams are: $\Sigma(\alpha, \beta, \gamma, a, b, c)$ [Figure 3] and $\Omega_4(\alpha, \beta, \gamma, \delta, \epsilon, a, b, c, d, e)$ [see Figure 4]. It is easy to see a difference between them. Inside the diagram Σ, some parts which may be considered as a diagram can be isolated in such a way that if $\Pi(a, b)$, $\Omega(\alpha, a)$, $\Delta(\alpha, a, b)$ denote, respectively, the diagrams of Figures 5–7, it is natural to write

$$\Sigma(\alpha, \beta, \gamma, a, b, c) = \Omega(\alpha, \Delta(\beta, \Omega(\gamma, a), \Pi(b, c))).$$

Nothing of this kind can be done for what concerns Ω_4; the same happens for the entire infinite class of similar diagrams

$$\Omega_1[=\Omega], \Omega_2, \Omega_3, \cdots, \Omega_n, \cdots,$$

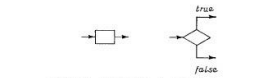
whose formation rule can be easily imagined.

Let us say that while Σ is decomposable according to subdiagrams Π, Ω and Δ_1 the diagrams of the type Ω_n are not decomposable. From the last consideration, which should be obvious to anyone who tries to isolate with a

PROGRAMMA

Programmazione strutturata

- Un algoritmo organizzato secondo la logica delle strutture di controllo è detto a **programmazione strutturata**
- Utilizzando tale approccio è possibile produrre programmi complessi, ma facili da capire (rispetto a quelli non strutturati) e quindi più comodi da verificare, correggere, modificare e perfino dimostrare in senso matematico come corretti



FIGS. 1-2. Functional and predicative boxes

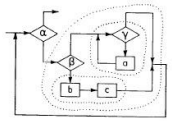


FIG. 3. Diagram of Σ

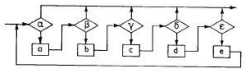
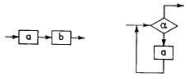
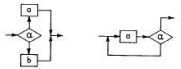


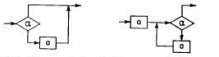
FIG. 4. Diagram of Ω_b



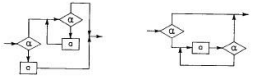
FIGS. 5-6. Diagrams of Π and Ω



FIGS. 7-8. Diagrams of Δ and Φ



FIGS. 9-10. Diagram of Δ and a diagram equivalent to Φ



FIGS. 11-12. Diagrams equivalent to Δ and Ω

Volume 9 / Number 5 / May, 1966

broken line (as was done for Σ) a part of Ω_n provided with only one input and one output, it follows that:

It is not possible to decompose all flow diagrams into a finite number of given base diagrams.

However, together with this decomposition, that could be called *strong*, another decomposition may be considered which is obtained by operating on a diagram equivalent to the one to be decomposed (that is, the diagram has to express the same transformation, whatever the meaning of the boxes contained in it may be). For instance, it may be observed that if we introduce $\Phi(\alpha, a)$ [as in Figure 8] and $\Delta(\alpha, a)$ [as in Figure 9] the diagrams of Φ, Δ and Ω become, respectively, equivalent to Figures 10, 11 and 12.

Thus, the following decompositions may be accepted:

$$\Phi(\alpha, a) = \Pi(a, \Omega(\alpha, a))$$

$$\Delta(\alpha, a) = \Delta(\alpha, \Omega(\alpha, a), a)$$

$$\Omega(\alpha, a) = \Delta(\alpha, \Phi(\alpha, a)).$$

Nevertheless, it is to be reckoned that the above statement holds even with regard to the new wider concept of decomposability. In fact, it does not seem possible[1] for every Ω_n to find an equivalent diagram which does not contain, as a subprogram, another Ω_n or an Ω of higher order. For instance, note that

$$\Omega_5(\alpha, \beta, \gamma, a, b, c) = \Delta(\alpha, \Pi(a, \Omega_5(\beta, \gamma, \alpha, b, c, a)))$$
$$= \Omega_6(\alpha, \beta, \gamma, \alpha, \beta, \gamma, a, b, c, a, b, c)$$

and similar formulas hold for all orders of Ω.

The proved unfeasibility is circumvented if a new predicate is added and if, among the elementary operations, some are assumed which either add one bit of information to the object of the computation or remove one from it. The extra bits have a stack structure (formally described below as nested ordered pairs) since it is sufficient to operate and/or take decisions only on the topmost bit.

Therefore, three new functional boxes denoted by T, F, K, and a new predicative box ω are introduced. The effect of the first two boxes is to transform the object x into the ordered pair (v, x) where v can have only the values **t** (true) or **f** (false); more precisely,

$$x \xrightarrow{T} (\mathbf{t}, x), \quad x \xrightarrow{F} (\mathbf{f}, x), \quad (\mathbf{t}, x) \xrightarrow{T} (\mathbf{t}, (\mathbf{t}, x))$$

and so on. Box K takes out from an ordered pair its second component

$$(v, x) \xrightarrow{K} x, \quad (\mathbf{t}, (\mathbf{f}, (\mathbf{t}, x))) \xrightarrow{K} (\mathbf{f}, (\mathbf{t}, x)).$$

The predicate ω is defined as

$$\omega[(v, x)] = \mathbf{t} \iff v = \mathbf{t},$$

i.e., the predicate ω is verified or not according to whether the first component of the pair is **T** or **F**; ω and K are defined only on a pair; on the contrary, all the boxes

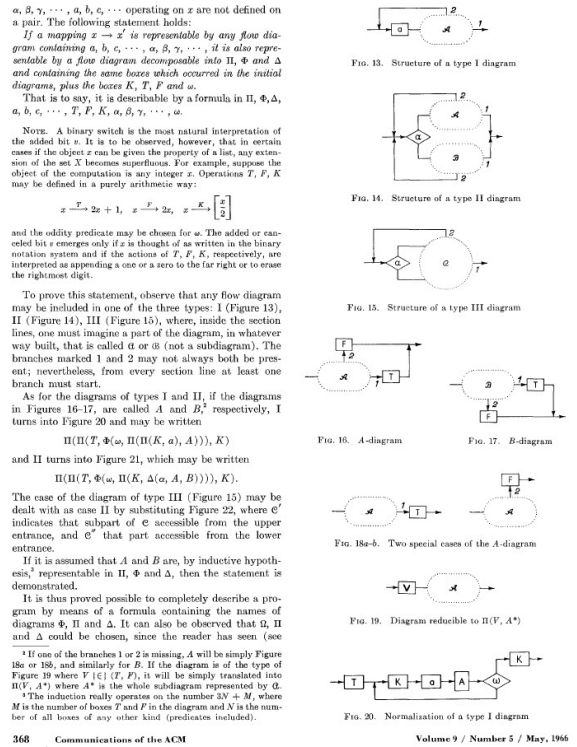
[1] We did not, however, succeed in finding a plain and sufficiently rigorous proof of this.

Communications of the ACM 367

PROGRAMMA

Diagramma di flusso

- Un programma, e le relative strutture, possono avere una descrizione grafica: il **diagramma di flusso**
- Come lo pseudocodice, questa raffigurazione è utile per descrivere un algoritmo ed offrire una rappresentazione chiara ed immediata di come e dove operano le strutture di controllo



$\alpha, \beta, \gamma, \cdots, a, b, c, \cdots$ operating on x are not defined on a pair. The following statement holds:

If a mapping $x \to x'$ is representable by any flow diagram containing $a, b, c, \cdots, \alpha, \beta, \gamma, \cdots$, it is also representable by a flow diagram decomposable into Π, Φ *and* Δ *and containing the same boxes which occurred in the initial diagrams, plus the boxes* K, T, F *and* ω.

That is to say, it is describable by a formula in $\Pi, \Phi, \Delta, a, b, c, \cdots, T, F, K, \alpha, \beta, \gamma, \cdots, \omega$.

NOTE. A binary switch is the most natural interpretation of the added bit v. It is to be observed, however, that in certain cases if the object x can be given the property of a list, any extension of the set X becomes superfluous. For example, suppose the object of the computation is any integer x. Operations T, F, K may be defined in a purely arithmetic way:

$$x \xrightarrow{T} 2x + 1, \quad x \xrightarrow{F} 2x, \quad x \xrightarrow{K} \left[\frac{x}{2}\right]$$

and the oddity predicate may be chosen for ω. The added or canceled bit v emerges only if x is thought of as written in the binary notation system and if the actions of T, F, K, respectively, are interpreted as appending a one or a zero to the far right or to erase the rightmost digit.

To prove this statement, observe that any flow diagram may be included in one of the three types: I (Figure 13), II (Figure 14), III (Figure 15), where, inside the section lines, one must imagine a part of the diagram, in whatever way built, that is called $\mathfrak{A}$ or $\mathfrak{B}$ (not a subdiagram). The branches marked 1 and 2 may not always both be present; nevertheless, from every section line at least one branch must start.

As for the diagrams of types I and II, if the diagrams in Figures 16–17, are called A and B,[3] respectively, I turns into Figure 20 and may be written

$$\Pi(\Pi(T, \Phi(\omega, \Pi(\Pi(K, a), A))), K)$$

and II turns into Figure 21, which may be written

$$\Pi(\Pi(T, \Phi(\omega, \Pi(K, \Delta(\alpha, A, B))))), K).$$

The case of the diagram of type III (Figure 15) may be dealt with as case II by substituting Figure 22, where $\mathfrak{C}'$ indicates that subpart of $\mathfrak{C}$ accessible from the upper entrance, and $\mathfrak{C}''$ that part accessible from the lower entrance.

If it is assumed that A and B are, by inductive hypothesis,[4] representable in Π, Φ and Δ, then the statement is demonstrated.

It is thus proved possible to completely describe a program by means of a formula containing the names of diagrams Φ, Π and Δ. It can also be observed that Ω, Π and Δ could be chosen, since the reader has seen (see

[3] If one of the branches 1 or 2 is missing, A will be simply Figure 18a or 18b, and similarly for B. If the diagram is of the type of Figure 19 where $V \mid \in \{T, F\}$, it will be simply translated into $\Pi(V, A^*)$ where A^* is the whole subdiagram represented by $\mathfrak{A}$.

[4] The induction really operates on the number $2N + M$, where M is the number of boxes T and F in the diagram and N is the number of all boxes of any other kind (predicates included).

FIG. 13. Structure of a type I diagram

FIG. 14. Structure of a type II diagram

FIG. 15. Structure of a type III diagram

FIG. 16. A-diagram

FIG. 17. B-diagram

FIG. 18a–b. Two special cases of the A-diagram

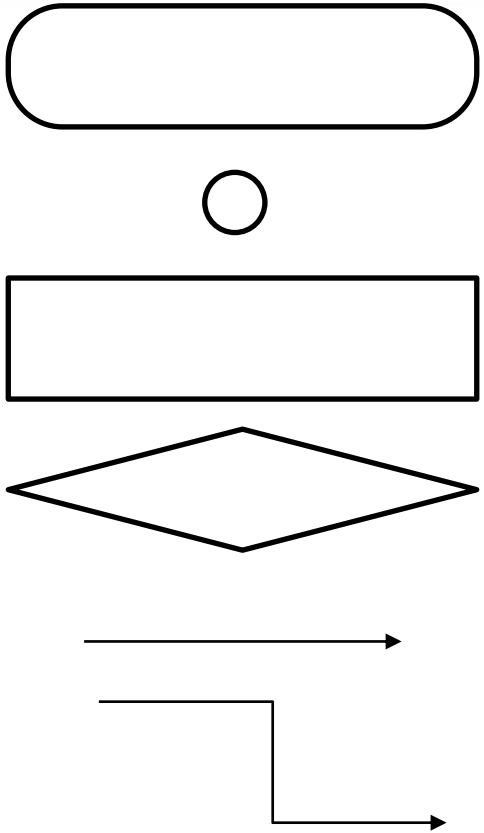
FIG. 19. Diagram reducible to $\Pi(V, A^*)$

FIG. 20. Normalization of a type I diagram

PROGRAMMA

Diagramma di flusso: simbologia

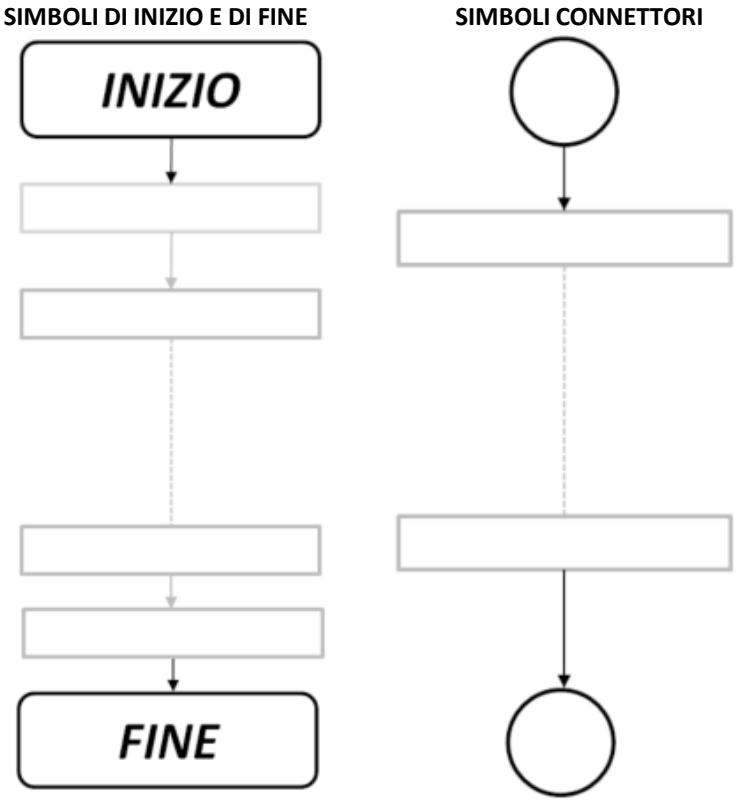
- Un diagramma di flusso è disegnato usando dei **simboli specifici**: il rettangolo con i vertici arrotondati, il cerchietto, il rettangolo e il rombo
- I simboli sono collegati da frecce chiamate **linee di flusso**
- Le linee di flusso indicano l'ordine in cui sono eseguite le istruzioni ed intraprese delle azioni



PROGRAMMA

Diagramma di flusso: simbologia

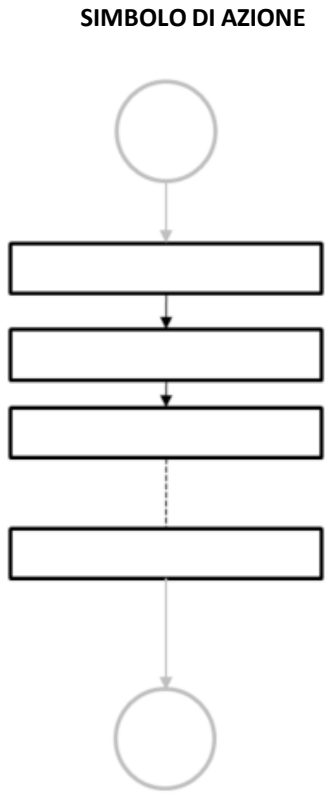
- I **rettangoli con i vertici arrotondati**, da cui si sviluppa il resto del diagramma, individuano il **punto di inizio e il punto di terminazione del programma**.
  Il primo contiene la parola *Inizio* e non prevede alcuna linea di flusso entrante; l'altro riporta il termine *Fine* e non ha alcuna linea di flusso uscente
- Quando si disegna solo una porzione di un diagramma, magari la parte più interessante dell'algoritmo, si usano dei cerchietti, chiamati anche **simboli connettori**



PROGRAMMA

Diagramma di flusso: simbologia

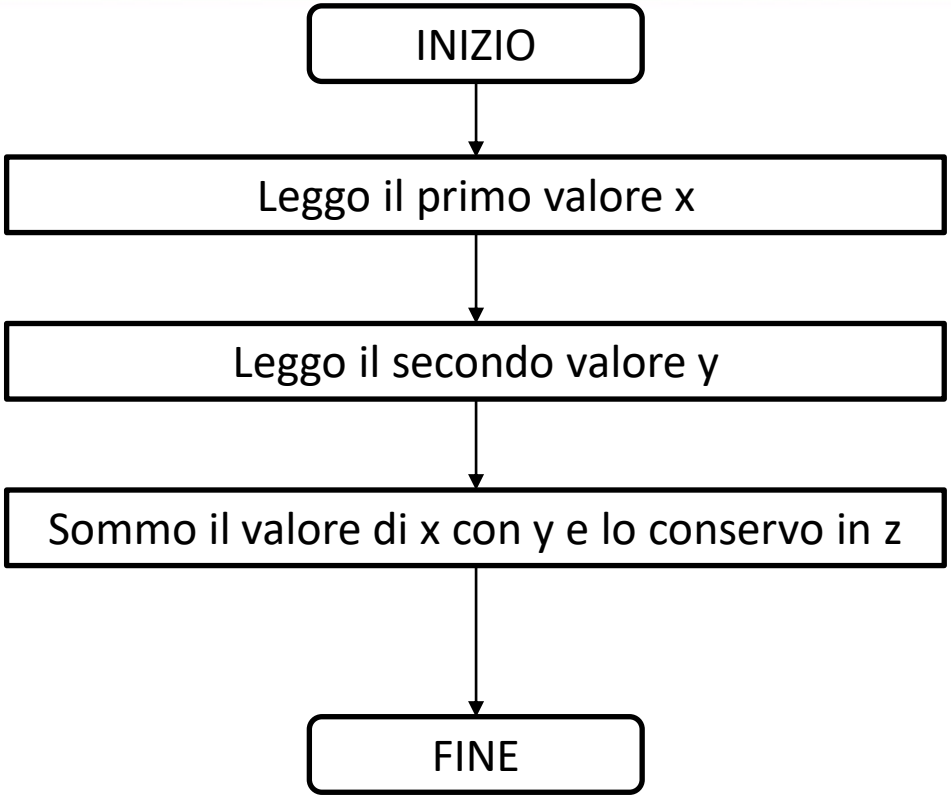
- Il **rettangolo**, o **simbolo di azione** (*functional box*), esprime la definizione di una variabile informatica o di un calcolo
- Una successione di rettangoli realizza la **struttura sequenziale** proposta da Böhm e Jacopini



PROGRAMMA

Diagramma di flusso: azione

Problema. Somma di due numeri

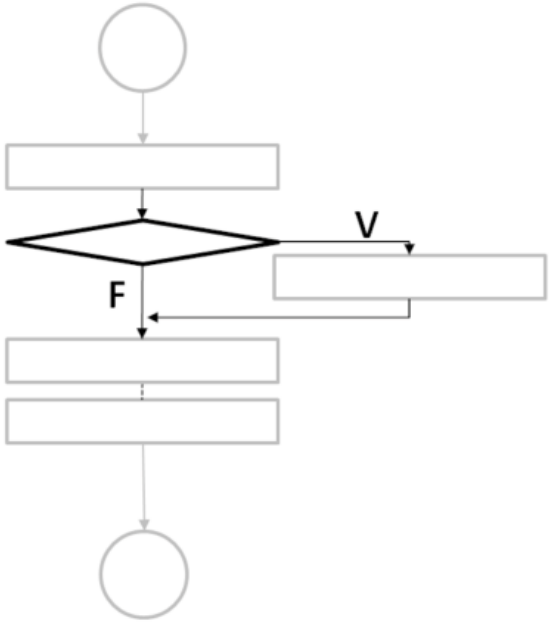


PROGRAMMA

Diagramma di flusso: simbologia

- Il **rombo**, o **simbolo di decisione** (predicative box), è la forma geometrica più interessante di un diagramma di flusso perché indica la necessità di effettuare una decisione e, in accordo all'esito (vero, true, o falso, false), indirizzare il seguito dell'elaborazione verso una struttura sequenziale o un'altra
- Una rombo realizza la **struttura selettiva** proposta da Böhm e Jacopini

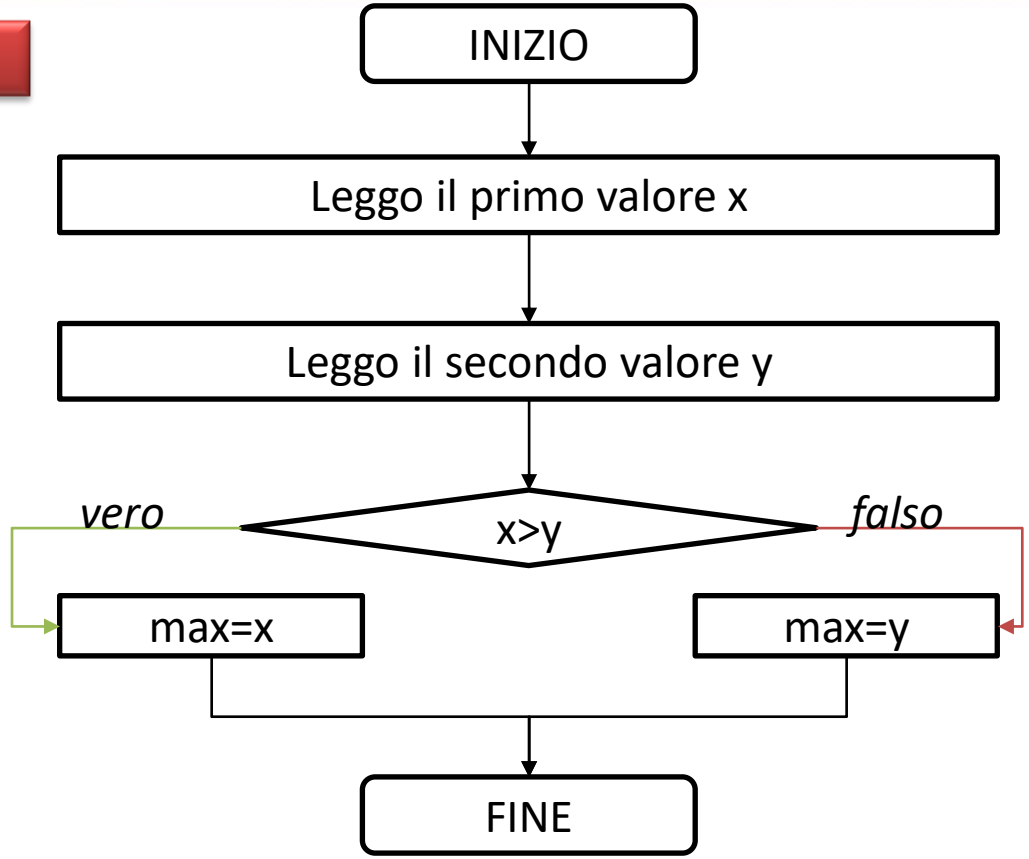
SIMBOLO DI DECISIONE



PROGRAMMA

Diagramma di flusso: azione

Problema. Massimo tra due numeri

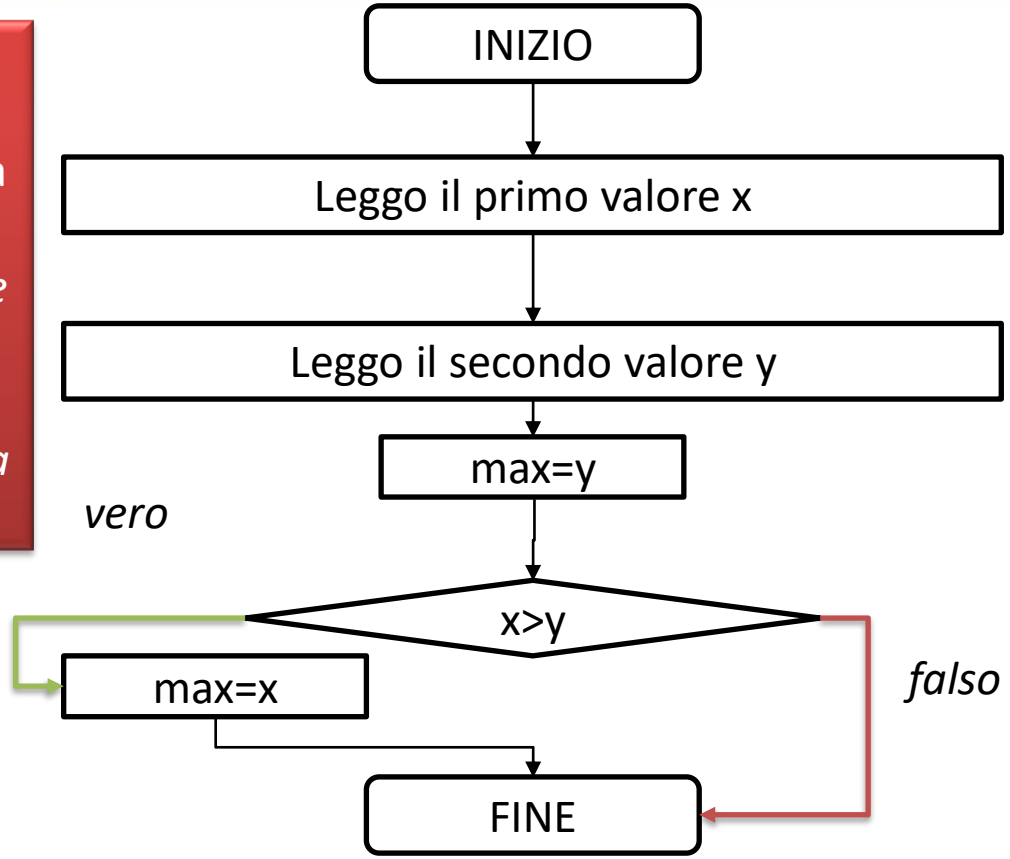


PROGRAMMA

Diagramma di flusso: azione

Problema. Massimo tra due numeri (Risoluzione proposta da una attenta studentessa in aula)

Il lucido precedente è didatticamente poco chiaro perché si applica una variante della regola di accatastamento non ancora descritta



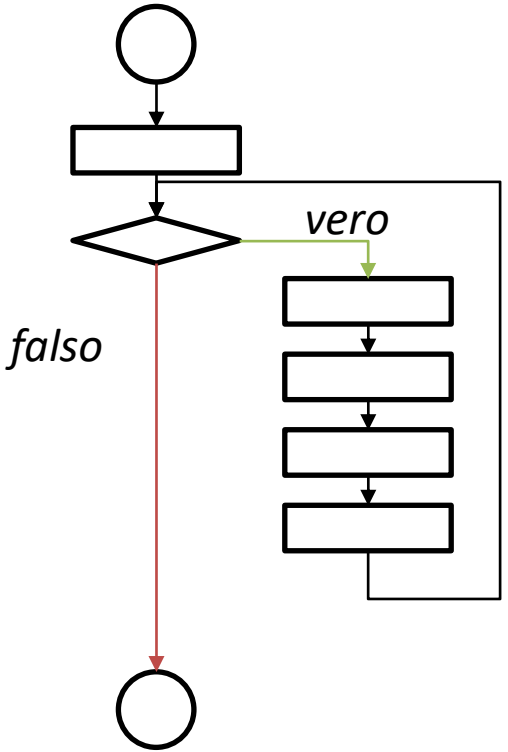
PROGRAMMA

Diagramma di flusso: simbologia

- Il **rombo**, o **simbolo di decisione** (predicative box) consente anche la formazione di un **ciclo**, ovvero la ripetizione di un certo numero di istruzioni per un numero di volte prestabilito o fino al verificarsi di una condizione

- Una rombo con un ritorno ad esso da un blocco istruzioni realizza la **struttura iterativa** proposta da Böhm e Jacopini

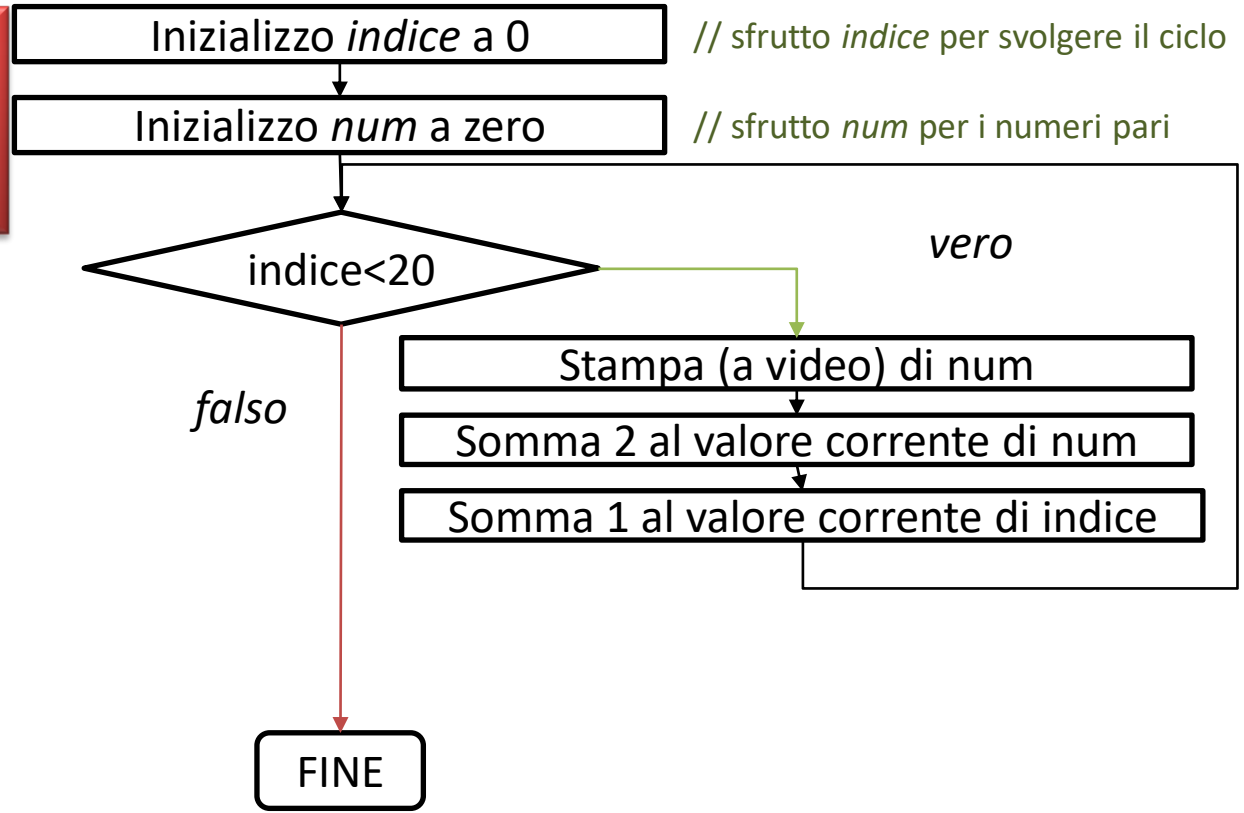
SIMBOLO DI DECISIONE USATO PER FARE UN CICLO



PROGRAMMA

Diagramma di flusso: simbologia

Problema. Stampa a video dei primi 20 numeri pari



PROGRAMMA

Diagramma di flusso: regole di composizione

- Per realizzare un **programma strutturato** è necessario applicare quattro regole al relativo diagramma di flusso:

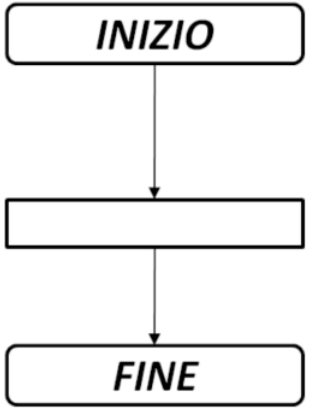
1. **Regola di Inizio**. Si implementa un algoritmo utilizzando il diagramma di flusso più semplice formato da un simbolo di inizio, un rettangolo ed un simbolo di terminazione
2. **Regola di Accatastamento**. Qualsiasi rettangolo è sostituibile da due rettangoli in sequenza
3. **Regola di Annidamento**. Ogni rettangolo può essere sostituito con una struttura di selezione
4. **Regola di Ripetizione**. Si possono applicare le regole di Accatastamento e di Annidamento quante volte si vuole e in qualsiasi ordine

PROGRAMMA

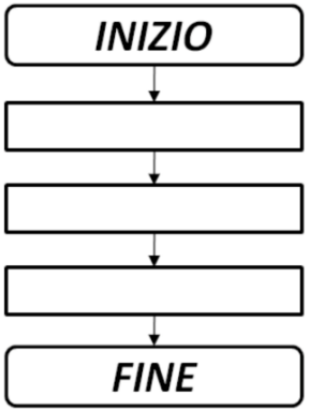
Diagramma di flusso: regole di composizione

- Applicando ripetutamente la **Regola di Accatastamento** al diagramma di flusso più semplice se ne produce un strutturato contenente molti rettangoli in sequenza, cioè si genera un blocco di istruzioni

REGOLA DI INIZIO



REGOLA ACCATASTAMENTO

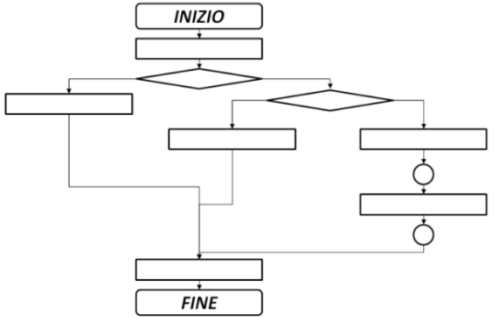


PROGRAMMA

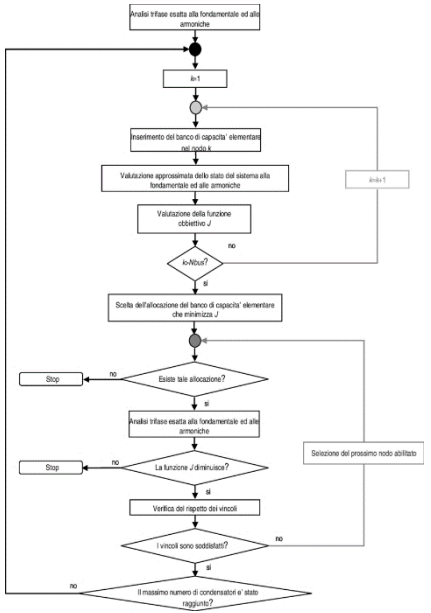
Diagramma di flusso: regole di composizione

- Utilizzando più volte la **Regola di Annidamento** al diagramma più semplice, si produce un flusso con più istruzioni di decisione annidate
- La **Regola di Ripetizione**, invece, genera strutture più ampie, molto complicate e con un annidamento più profondo

REGOLA DI ANNIDAMENTO



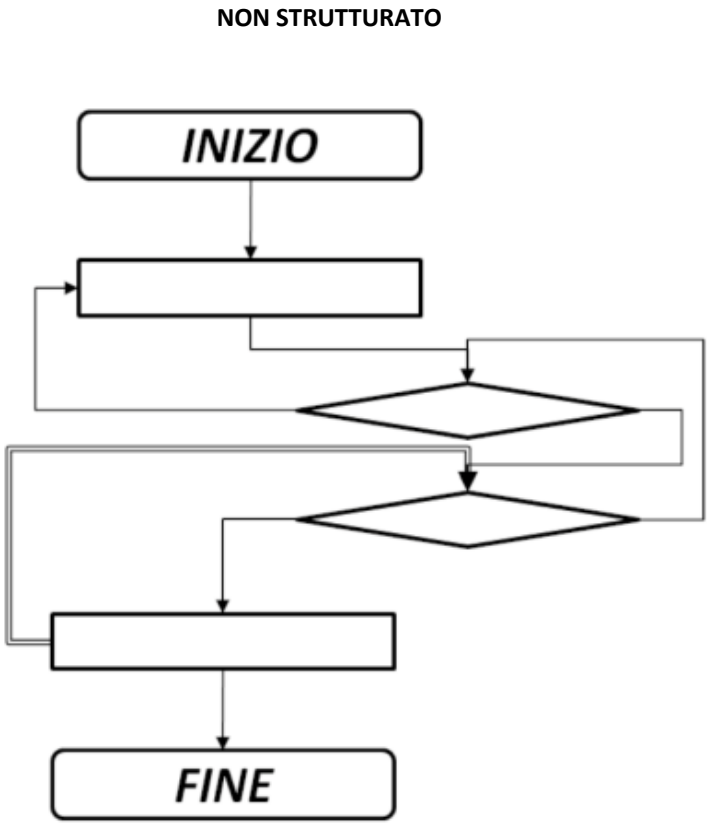
REGOLA DI RIPETIZIONE



PROGRAMMA

Diagramma di flusso: programma NON strutturato

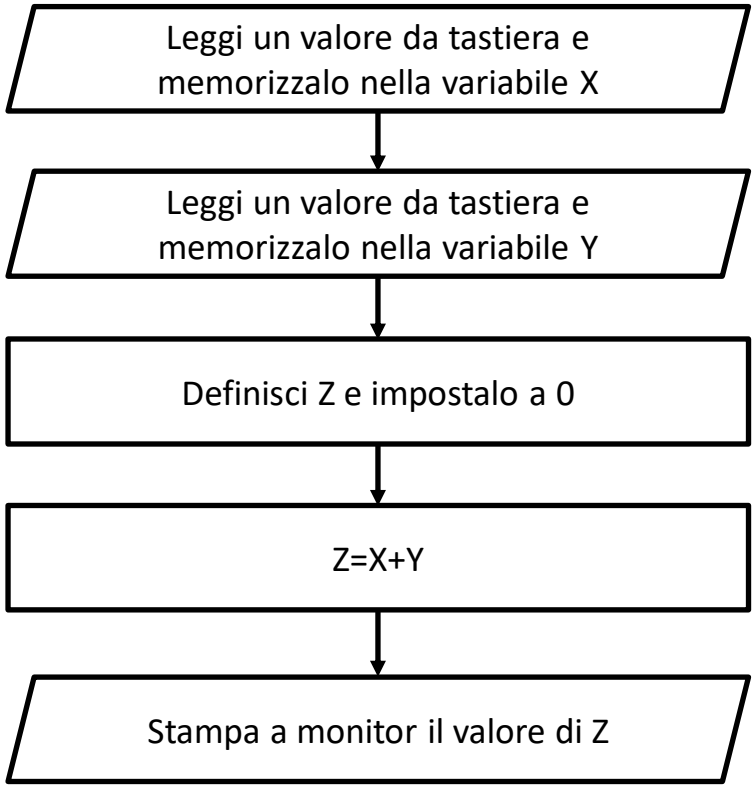
- I diagrammi di flusso che risultano dall'applicazione delle regole costituiscono l'insieme di tutti i possibili **programmi strutturati**
- Nel caso contrario si ha un **programma non strutturato** (esempio a lato, la doppia linea mostra l'incoerenza)



PROGRAMMA

Diagramma di flusso: simbolo per interazione con I/O

- Per comodità, quando si vuole richiedere l'intervento di una periferica di Input o di Output si usa il **parallelogramma**



PROGRAMMA

Commenti

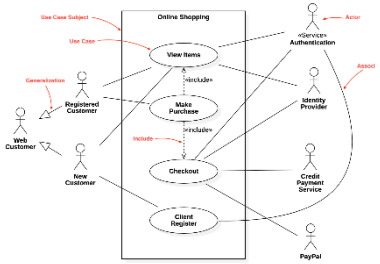
- I commenti sono delle informazioni testuali (che non sono prese in considerazione quando il codice scritto in un linguaggio ad alto livello è tradotto in un eseguibile) che specificano il senso delle variabili o delle istruzione (o gruppi di esse)
- Un codice che realizza un programma privo di commenti NON è un BUON codice
- Il commento è una parte fondamentale della documentazione
- Spesso una linea di commento è integrabile nel codice anteponendo il simbolo // (in alcuni linguaggi si usa #)
- Più linee di commento richiedono il simbolo /* di apertura e quello */ di chiusura (oppure la replica su ogni riga del simbolo #)

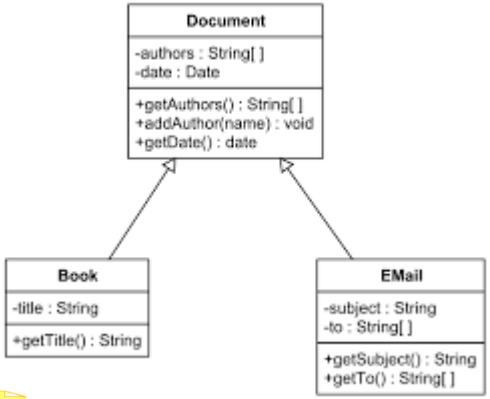
```
//############################
// Programma di: Franco Liberati
// Data: 01/01/2023
// Ultima revisione: ore 18:45
// Specifica: calcolo della media di sei numeri
// Algoritmo: una variabile indice
//(index) per il ciclo una variabile totalizzatore
//(tot) per sommare i numeri
// alla fine si procede dividendo tot per sei
// Linguaggio usato: R
// ############################
```

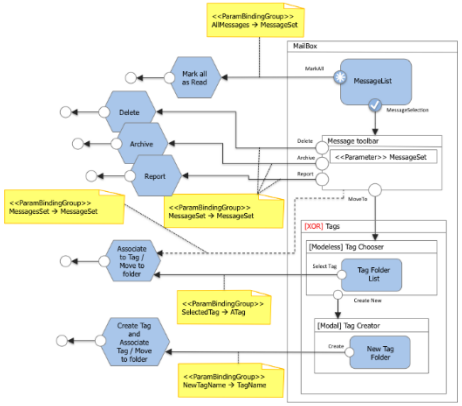
PROGRAMMA

Linguaggi di modellazione

- Un diagramma di flusso, come anche la descrizione in pseudocodice, **sono linguaggi di modellazione** (*modelling language*) ovvero espressioni formali che sono utilizzati per descrivere un qualsiasi sistema (anche di natura diversa da quella matematica e informatica)
- Attualmente, con lo sviluppo della programmazione ad oggetti, ce ne sono di più sviluppati (Unified Modeling Language, UML; Systems Modeling Language, SysML; EXPRESS-G; Interaction Flow Modeling Language, IFML), ma sono trattati in altri corsi







Programma Informatico

PROGRAMMA

Generalità

Definizione. Un programma è definito come una serie di istruzioni sequenziali

Il programma è scritto usando un **linguaggio di programmazione**

In altre parole un programma realizza un algoritmo sfruttando un linguaggio di programmazione

PROGRAMMA

Generalità

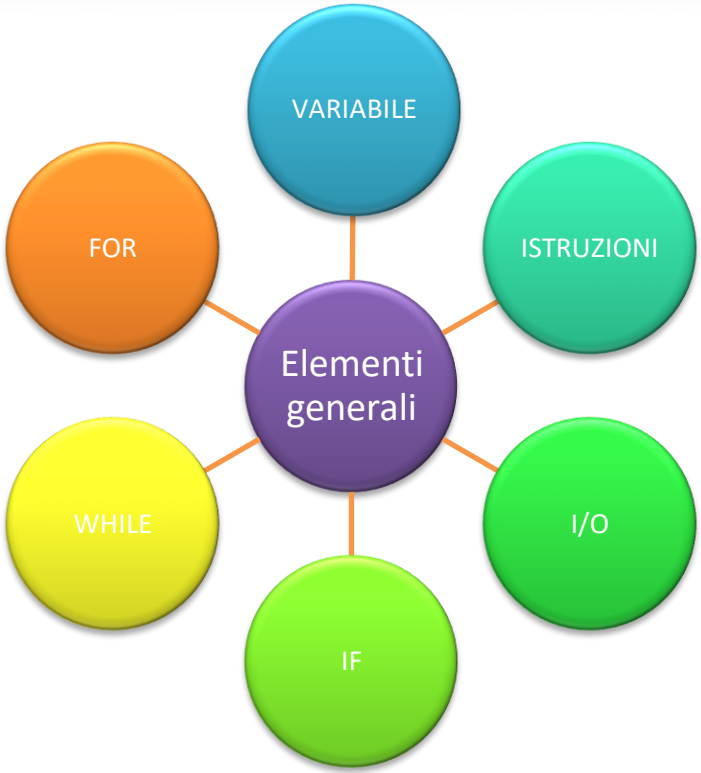
- Esistono diverti **tipi di linguaggi di programmazione**:
  - Basso livello
    - Macchina/ assemblativo
  - Alto livello
- I linguaggi sono anche **classificabili** come:
  - Imperativo
  - Funzionale
  - Dichiarativo
  - Strutturato
  - Ad Oggetti
  - Parallelo
  - Scripting

Imperativo	• BASIC,C, COBOL, FORTRAN
Funzionale	• LISP
Dichiarativo	• Mercury
Strutturato	• Pascal, C,
Ad Oggetti	• C++, Java, C#
Parallelo	• Occam, Linda
Scripting	• PHP, ASP, Python

PROGRAMMA

Elementi generali

- Un linguaggio di programmazione ad alto livello ha, in generale, i seguenti elementi fondamentali:
  - **Variabile**
    - Tipo della variabile
  - **Un insieme di istruzioni logiche aritmetiche**
  - **Interazione con dispositivi di Input/Output**
  - **IF** (selezione)
  - **WHILE** (ripetizione)
  - **FOR** (iterazione)



PROGRAMMA

Variabile

Definizione. Una variabile è una locazione di memoria che contiene un valore.

Una variabile ha un nome (identificatore) cioè una etichetta che sostituisce l'indirizzo in memoria in cui risiede il valore numerico

Osservazione:

L'identificatore è scelto dal programmatore seguendo queste regole:

- due variabili non possono avere lo stesso identificatore
- si possono usare sia le lettere sia numeri, ma il primo carattere deve essere una lettera
- in alcuni linguaggi di programmazione le maiuscole sono distinte dalle minuscole (linguaggi *case sensitive*)
- è preferibile non usare identificatori molto lunghi (ma significativi)
- le parole riservate (if, int, while, for) non possono essere usate per i nomi delle variabili

PROGRAMMA

Variabile: tipi

- I valori memorizzabili in una variabile sono caratterizzabili da un **tipo**; cioè il significato delle parole
- I tipi più comuni sono:
  - **byte** (numero intero; parola di dimensione 8bit) {esiste anche la variante **unsigned byte**}
  - **char** (numero naturale ad 8 bit rappresentato come un carattere, secondo lo standard ASCII)
  - **int** (numeri interi; parole di dimensione 32bit)
  - **unsigned int** (numeri naturali; parole di dimensione 32bit)
  - **long** (numeri interi; parole di dimensione 64bit) {esiste anche la variante **unsigned long**}
  - **short** (numeri interi; parole di dimensione 16bit) {esiste anche la variante **unsigned short**}
  - **float** (numeri in virgola mobile singola precisione)
  - **double** (numeri in virgola mobile doppia precisione)
  - **bool** (numeri booelani: due valori TRUE e FALSE)

PROGRAMMA

Variabile: dichiarazione

- Per essere usata una variabile deve essere prima dichiarata (o definita).
  La **dichiarazione** ha la seguente sintassi:
  tipo_variabile nome_variabile;

- È possibile **inizializzare** una variabile, cioè è possibile assegnare un valore (si mette nella cella di memoria un operando) utilizzando il simbolo **=**

  tipo_variabile nome_variabile ;
  nome_variabile**=***valore*;

- Per i caratteri bisogna specificare il carattere tra singoli apici

  char nome_variabile**=***'F'*;

DICHIARAZIONE:
int numero;

DICHIARAZIONE
int numero;

INIZIALIZZAZIONE
numero=4;

DICHIARAZIONE E INIZIALIZZAZIONE
int numero=4;
float numreal=56.78;
char carattere='A';

PROGRAMMA

Operatori matematici

- Un linguaggio di programmazione ad alto livello utilizza **operatori matematici**:

Funzione	Operatore	*Note*
Addizione	+	
Sottrazione	-	
Moltiplicazione	*	
Divisione	/	*la divisione tra valori di tipo interi restituisce solamente il quoziente (e non la parte frazionata!)*
Resto divisione	%	*applicabile solo ai valori di tipo interi*

ESEMPIO

```
int x=35;
int y=20;
int result=0;
float real1=45.67;
float real2=13.54;
float realresult;

result=x+y;
//valore di result=55
result=x-y;
//valore di result=15
result=x*y;
//valore di result=700
result=x/y;
//valore di result=1
realresult=real1/real2;
//valore di result=3.372968
result=x%y;
//valore di result=15
realresult=real1%real2;
//errore di compilazione: il modulo non può essere usato con numeri reali
```

PROGRAMMA

Operatori matematici: priorità

- Gli operatori matematici hanno una **precedenza di risoluzione**: %, /, *, -, +

Per evitare ambiguità bisogna usare le parentesi

OPERAZIONE LOGICHE E ARITMETICHE

```
int x;
x=5+7;
// La variabile X ha valore 12
x=1;
x=x+1;
//La variabile X risultante (a sinistra dell'uguale) ha valore 2
//cioè il valore della X corrente al cui si somma 1
int y; int z; int w;
y=6;
z=x+y;
//La variabile Z mantiene la somma di X+Y cioè Z=8
w=3+z/2;
//Che valore ha w?
```

ESEMPIO PRECEDENZA DI RISOLUZIONE

```
int x=10; int y=5; int result=0;
result=x*y+10/5-x%y =50+2-0=52
result=(x*y+10)/(5-x%y) =(50+10)/(5-0)=60/5=12
```

PROGRAMMA

Operatori bit per bit

- Un linguaggio di programmazione ad alto livello utilizza **operatori bit per bit**:

Funzione	Operatore	*Note*
AND	&	*Non si applica a numeri reali*
OR	|	*Non si applica a numeri reali*
XOR	^	*Non si applica a numeri reali*
NOT	!	*Non si applica a numeri reali*

- Gli operatori lavorano bit per bit
- Il NOT, o complementarietà, agisce su un solo operando

AND	Bit1	Bit2
0	0	0
0	0	1
0	1	0
1	1	1

OR	Bit1	Bit2
0	0	0
1	0	1
1	1	0
1	1	1

XOR	Bit1	Bit2
0	0	0
1	0	1
1	1	0
0	1	1

NOT	Bit1
1	0
0	1

PROGRAMMA

Operatori bit per bit

- Un linguaggio di programmazione ad alto livello utilizza **operatori bit per bit**:

Funzione	Operatore	*Note*
AND	&	*Non si applica a numeri reali*
OR	|	*Non si applica a numeri reali*
XOR	^	*Non si applica a numeri reali*
NOT	!	*Non si applica a numeri reali*

- Gli operatori logici lavorano bit per bit
- Il NOT, o complementarietà, agisce su un solo operando

ESEMPIO

```
int x=12;
// in binario a 32bit 00000000 00000000 00000000 00001100
int y=10;
// in binario a 32bit 00000000 00000000 00000000 00001010

int z;
z= x&y
//z ha valore  00000000 00000000 00000000 00001000
z=x|y
//z ha valore  00000000 00000000 00000000 00001110
z=!x
//z ha valore  11111111 11111111 11111111 11110011
```

PROGRAMMA

Operatore logico-matematico: shift

- L'**operatore di spostamento** (SHIFT) sposta n bit a sinistra o a destra
- La sintassi è:

Shift verso sinistra

nome_variabile_int<<n

Shift verso destra

nome_variabile_int>>n

Lo shift ha un significato logico (sposta i bit) e matematico (lo spostamento verso sinistra indica una moltiplicazione del valore della variabile per 2^n verso destra è una divisione tra interi del valore con il divisore 2^n)

SHIFT

```
int x;
x=8;
// 00000000 00000000 00000000 00001000
// shift a sinistra
y=x<<2;
// y ha valore
//00000000 00000000 00000000 00100000
//cioè y=32 ovvero 8*2^2


x=15;
// 00000000 00000000 00000000 00001111
// shift a destra
y=x>>3;
// y ha valore
//00000000 00000000 00000000 00000001
//cioè y=3 ovvero 15/2^3
```

PROGRAMMA

Variabile: riuso

- Una variabile può essere utilizzata per effettuare un calcolo ed essere riutilizzata come locazione in cui archiviare il nuovo valore
- Questa opzione è utile per creare indici e contatori

NB: *Una variabile informatica è differente da una variabile matematica perché questa ultima è un identificativo che rappresenta un valore non noto sul quale si può intervenire con le regole di risoluzione di equazioni o di calcoli algebrici*

2x=x+5 (espressione matematica risolvibile con x=5)

2*x=x+5 (espressione non corretta in ambito informatico)

ESEMPIO

```
int x=35;
//il valore attuale di x è 35
x=x+4;
//il valore attuale di x è 39
```

Dopo il calcolo il valore di x è 39 perché preleva dalla memoria all'indirizzo indicato da x il valore 35, effettua la somma con 4 e archivia il risultato di nuovo nella locazione indicata da x

PROGRAMMA

Operatori di confronto

- Un linguaggio di programmazione ad alto livello ha **operatori di confronto** che permettono il confronto tra variabili e valori

Confronto	Operatore
Uguale	==
Diverso	!=
Maggiore	>
Maggiore uguale	>=
Minore	<
Minore uguale	<=

- Sono utilizzati soprattutto nelle strutture di controllo e restituiscono il **tipo booleano**: **TRUE** o **FALSE**

ESEMPIO

```
int x=35;
int y=20;
bool result;

35==35;
//restituisce TRUE
result=(x==y);
// valore di result=FALSE
result=(x!=y);
// valore di result=TRUE
result=(x>y);
 // valore di result=TRUE
result=(x>=y);
// valore di result=TRUE
result=(x<y);
// valore di result=FALSE
```

ESEMPIO NELLA STRUTTURA DI CONTROLLO

```
int x=10; int y=5; int max=0;
if (x>y) {max=x;}
else {max=y;}
```

PROGRAMMA

Operatori logici

- Un linguaggio di programmazione ad alto livello ha **operatori logici** cioè delle operazioni tra due espressioni A e B legate da un determinato tipo di relazione, tali da dare origine a una terza proposizione C con valore vero (TRUE) o falso (FALSE)

Confronto	Operatore
AND	&&
OR	||
NOT	!

- Sono utilizzati soprattutto nella formazione di condizioni delle strutture di controllo

AND	Espr1	Espr2
F	F	F
F	F	T
F	T	F
T	T	T

OR	Espr1	Espr2
F	F	F
T	F	T
T	T	F
T	T	T

NOT	Espr
T	F
F	T

ESEMPIO

```
// y=1 se x è un multiplo di 2 e di 5
int x=10; int y;
if (x%2==0)&&(x%5==0) { y=1;}
else {y=0;}
```

PROGRAMMA

Interazione con Input e Output

- Un linguaggio di programmazione ad alto livello ha istruzioni (funzioni) che consentono l'interazione con i dispositivi di Ingresso (Input), come la tastiera, e di uscita dati (Output), ad esempio il terminale video

- In generale si usa
  READ(nome_variabile) per leggere da tastiera
  PRINT(nome_variabile) per scrivere sul monitor

PS: **PRINT** ("*messaggio*") permette la scrittura di un messaggio testuale (una stringa)

ESEMPIO

```
int x;
READ (x);
//Salvataggio del dato inserito dall'utente nella variabile x
PRINT (x)
//Stampa del dato contento in x

_____
int x;
READ(x); #se immetto 5
x=x+10;
PRINT(x) #La funzione PRINT stampa 15
```

ESEMPIO STAMPA MESSAGGI TESTUALI

```
int numero;
PRINT("\nInserire numero:");
READ (numero);
PRINT ("\nHai inserito il numero:", numero);
```

PROGRAMMA

Interazione con Input e Output 2

- Negli esempi è utilizzata la funzione di **stampa a video terminale** che ha sintassi

printf("*ID*",nome_variabile);

ID	Significato	Esempio
%d	Stampa un numero intero	int x=5 printf ("%d",x)
%x	Stampa un numero intero in base esadecimale	int x=5 printf ("%d",x)
%f	Stampa un numero reale float o double	float x=5.5 printf ("%f",x)
%c	Stampa un carattere	char x="F" printf ("%c",x)

ESEMPIO

```
int x;
READ (x);
//Salvataggio del dato inserito dall'utente nella variabile x
printf("%d",x);
//Stampa del dato contento in x

_____
int x;
READ(x); #se immetto 5
x=x+10;
printf("%d",x); #Stampa 15
```

ESEMPIO STAMPA MESSAGGI TESTUALI

```
int numero;
printf ("\nInserire numero:");
READ (numero);
printf("\nHai inserito il numero: %d ", numero);
```

PROGRAMMA

Interazione con Input e Output 2

- Negli esempi è utilizzata la funzione di **lettura da tastiera** che ha sintassi

  scanf("*ID*",&nome_variabile);

ID	Significato	Esempio
%d	Legge da tastiera un valore intero e inizializza la variabile	int x; scanf ("%d",&x)
%f	Legge da tastiera un valore reale e inizializza la variabile	float x; scanf ("%f",x)
%c	Legge da tastiera un carattere	char x; scanf ("%c",x)

ESEMPIO

```
int x;
scanf ("%d",&x);
//Salvataggio del dato inserito dall'utente nella variabile x
printf("%d",x);
//Stampa del dato contento in x

_____
int x;
scanf ("%d",&x); #se immetto 5
x=x+10;
printf("%d",x); #Stampa 15
```

ESEMPIO STAMPA MESSAGGI TESTUALI

```
int numero;
printf ("\nInserire numero:");
scanf ("%d",&numero);
printf("\nHai inserito il numero: %d ", numero);
```

PROGRAMMA

IF selezione

- Un linguaggio di programmazione ha un costrutto che realizza la **selezione**
- La sintassi è

IF (condizione) **THEN** {ISTRUZIONI}
ELSE {ISTRUZIONI}

Cioè si valuta una condizione e se questa è vera si procede con delle istruzioni (ramo THEN), altrimenti si procede con delle altre istruzioni (ramo ELSE)

ESEMPIO

```
//calcolo del massimo tra due numeri
int x=10;
int y=5;
int max=0;
if (x>y) THEN {max=x;}
         ELSE {max=y;}
```

PROGRAMMA

IF selezione 2

- Negli esempi si utilizzerà la notazione

```
if (condizione) {
        ISTRUZIONI
    }
else {
        ISTRUZIONI
    }
```

Rendendo implicito il THEN

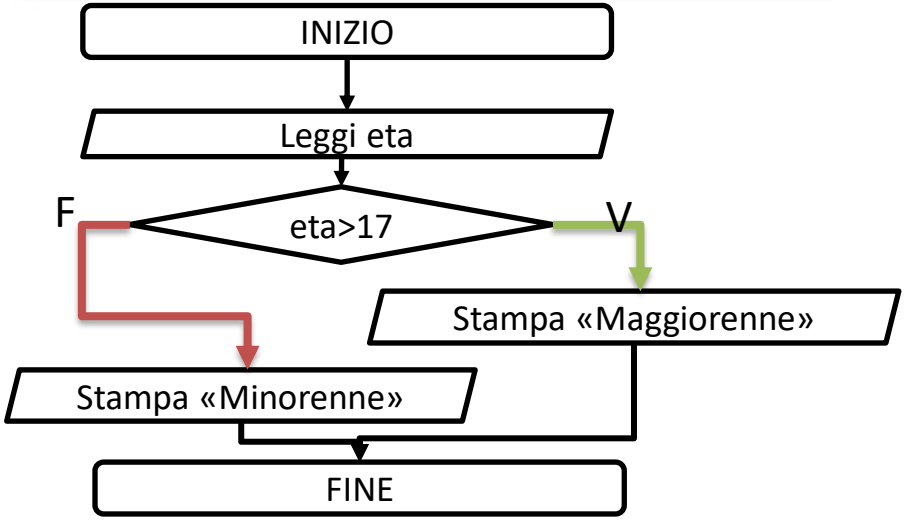
ESEMPIO

```
//calcolo del massimo tra due numeri
int x=10;
int y=5;
int max=0;
if (x>y) {max=x;}
        else {max=y;}
```

PROGRAMMA

IF selezione: esempio

L'utente inserisce la propria età e il programma restituisce una scritta se è maggiorenne (ovvero con età maggiore o uguale a 18 anni) o no



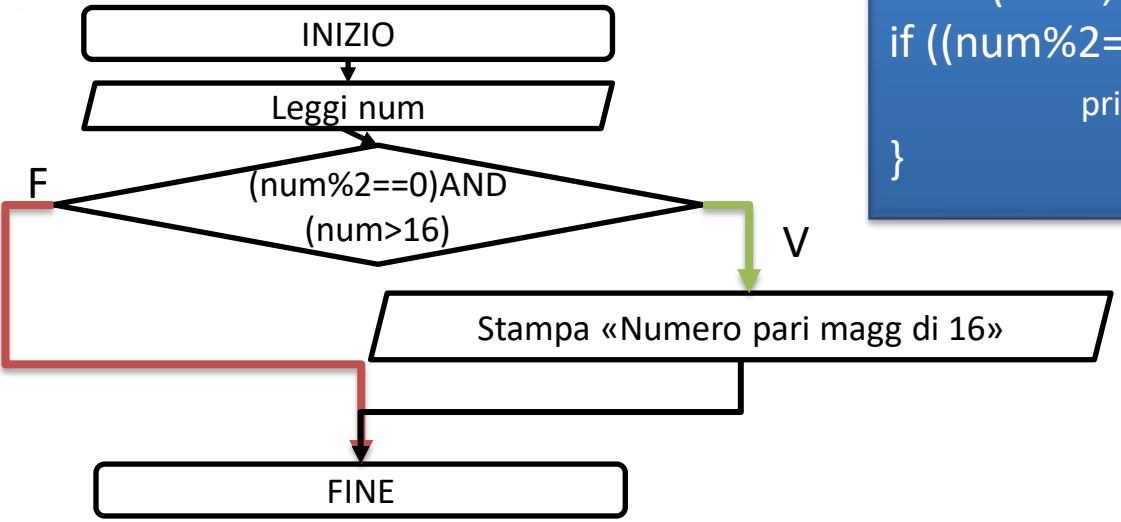
SOLUZIONE

```
int etautente;
printf("Inserisci eta dell'utente: ");
scanf("%d",&etautente);

if (etautente>17) {
        printf("MAGGIORENNE");
        }
        else {
        printf("MINORENNE");
        }
```

PROGRAMMA

IF selezione: esempio 2

L'utente inserisce un numero e il programma stabilisce se il numero è un numero pari ed è maggiore di 16



SOLUZIONE

```
//valutazione se un numero è pari e maggiore di 16
int num;
printf("Inserisci un numero: ");
scanf("%d",&num);
if ((num%2==0)&&(num>16)) {
        printf("NUMERO PARI E MAGGIORE DI 16");
}
```

PROGRAMMA

WHILE (ripetizione)

- Un linguaggio di programmazione ha un costrutto che realizza una **struttura iterativa condizionata** che esegue lo stesso blocco di codice fin quando si verifica un particolare evento
- La sintassi è:

```
WHILE(condizione)
        {
        ISTRUZIONI
        }
```

Il blocco delle istruzioni si ripete fino a quando la condizione da TRUE. Non appena la condizione restituisce FALSE il programma non considera più il blocco di istruzioni e prosegue il suo andamento sequenziale

ESEMPIO

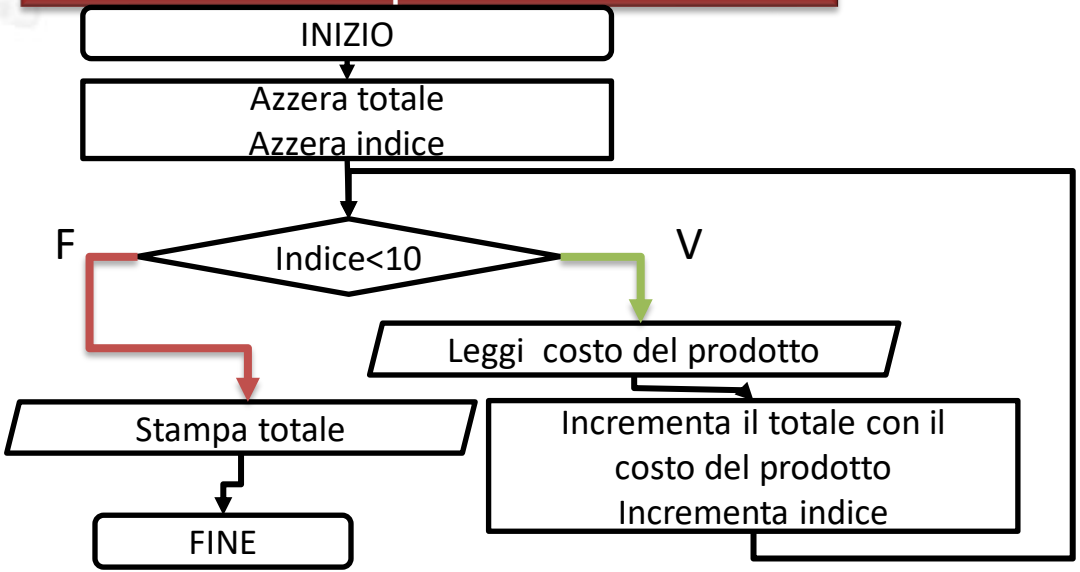
```
// stampa dei prima 10 numeri pari
int indice=0;
int pari=0;
while (indice<10){
        printf("\n%d",pari);
        pari=pari+2;
        indice=indice +1;
}
```

```
0
2
3
4
6
8
10
12
14
16
```

PROGRAMMA

WHILE: esempio

Realizzare un programma che simula la cassa di un supermercato. Leggere, da tastiera, il prezzo di 10 prodotti e restituire il totale della spesa



SOLUZIONE

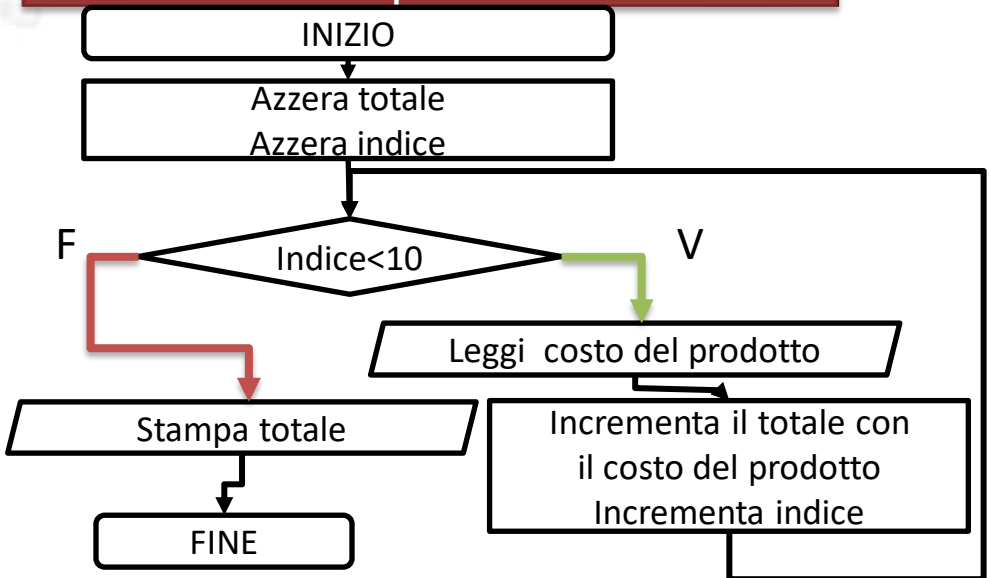
```
int indice=0;
int numero_prodotti=10;
float costo=0;
float tot=0;

WHILE (indice<numero_prodotti)
{
        printf("\nInserire il prezzo:");
         scanf("%f",&costo);
         tot=tot+costo;
}
printf("%f",tot);
```

ERRORE

PROGRAMMA

WHILE: esempio

Realizzare un programma che simula la cassa di un supermercato. Leggere, da tastiera, il prezzo di 10 prodotti e restituire il totale della spesa



SOLUZIONE

```
int indice=0;
int numero_prodotti=10;
float costo=0;
float tot=0;
while (indice<numero_prodotti)    {
printf("\nInserire il prezzo del %d prodotto: ", indice);
scanf("%f",&costo);
tot=tot+costo;
indice=indice+1;
}
printf("Totale spesa: %f euro",tot);
```

PROGRAMMA

FOR (iterazione)

- Un linguaggio di programmazione ha un costrutto che realizza una **struttura iterativa controllata** che esegue lo stesso blocco di codice per un numero prestabilito di volte
- La sintassi è:

```
FOR(indice;condizione; incremento)
                    {
                    ISTRUZIONI
                    }
```

Il blocco delle istruzioni si ripete fino a quando la condizione da TRUE

ESEMPIO

```
int indice;
int num_elementi=4;
float val=0;
float valore=0;
float media=0;
for (indice=0; indice<num_elementi;indice=indice+1){
        printf("\nInserisci il valore %d: ",indice);
        scanf("%f",&val);
        valore=valore+val;
}
media=valore/num_elementi;
printf("\nLa media è %f ",media);
```

```
Inserisci il valore 0: 5.5
Inserisci il valore 1: 4.5
Inserisci il valore 2: 3.2
Inserisci il valore 3: 4.8
La media è 4.500000:
```

PROGRAMMA

FOR iterazione: esempio

Realizzare un programma che legge un numero intero N e stampa la tabella di moltiplicazione da 1 a N

SOLUZIONE ALTERNATIVA

```
int indice;    int n;    int tabella;
printf("Inserire il numero :");
scanf("%d",&n);
printf("\nTABELLA");
for (indice=1; indice<=n;indice=indice+1){
                tabella=indice*n;
                printf("\n%d",tabella);
}
```

SOLUZIONE

```
int indice;
int n;
int tabella;
printf("Inserire il numero :");
scanf("%d",&n);
printf("\nTABELLA");
for (indice=1; indice<n;indice=indice+1){
                tabella=indice*n;
                 printf("\n%d",tabella);
}
printf("\n%d", n*n);
```

PROGRAMMA

ESERCIZIO IN CLASSE

Determinare il valore della variabile RESULT

Riga	Istruzioni
1:	int X,Y,Z;
2:	int RESULT=0;
3:	X=35;
4:	Y=55;
5:	IF (X>Y) THEN {Z=2*X-Y;}
6:	ELSE {Z=Y/2+X;}
7:	X=3;
8:	RESULT=5;
9:	WHILE (X<5)
10:	{
11:	RESULT=RESULT+Z;
12:	X=X+1;
13:	}
14:	Print (RESULT);

PROGRAMMA

Ordine istruzioni

- L'ordine in cui le istruzioni sono elaborate è detto il **controllo del programma**
- Alcune istruzioni, quelle appartenenti alla classe dei salti, consentono di variare il normale andamento sequenziale e la loro presenza nel programma implica **il trasferimento del controllo**

Riga	Istruzioni
1:	int x;
2:	int y;
3:	int z;
4:	x=35;
5:	y=55;
6:	IF (x>y)
7:	THEN z=x+y;
8:	ELSE z=x-y;
9:	PRINT (z)

CONTROLLO DEL PROGRAMMA: 1,2,3,4,5,6,8,9

TRASFERIMENTO DEL CONTROLLO: 6

PROGRAMMA

ESERCIZIO IN CLASSE

Determinare il controllo del programma, indicare i trasferimenti del controllo e il risultato finale

Riga	Istruzioni
1:	int x,y,z;
2:	x=35;
3:	y=55;
4:	IF (x>y) THEN {
5:	z=x+y;
6:	IF (z>3*x) THEN {
7:	z=3*x;
8:	}
9:	}
10:	ELSE {
11:	z=x-y;
12:	}
13	PRINT (z)

PROGRAMMA

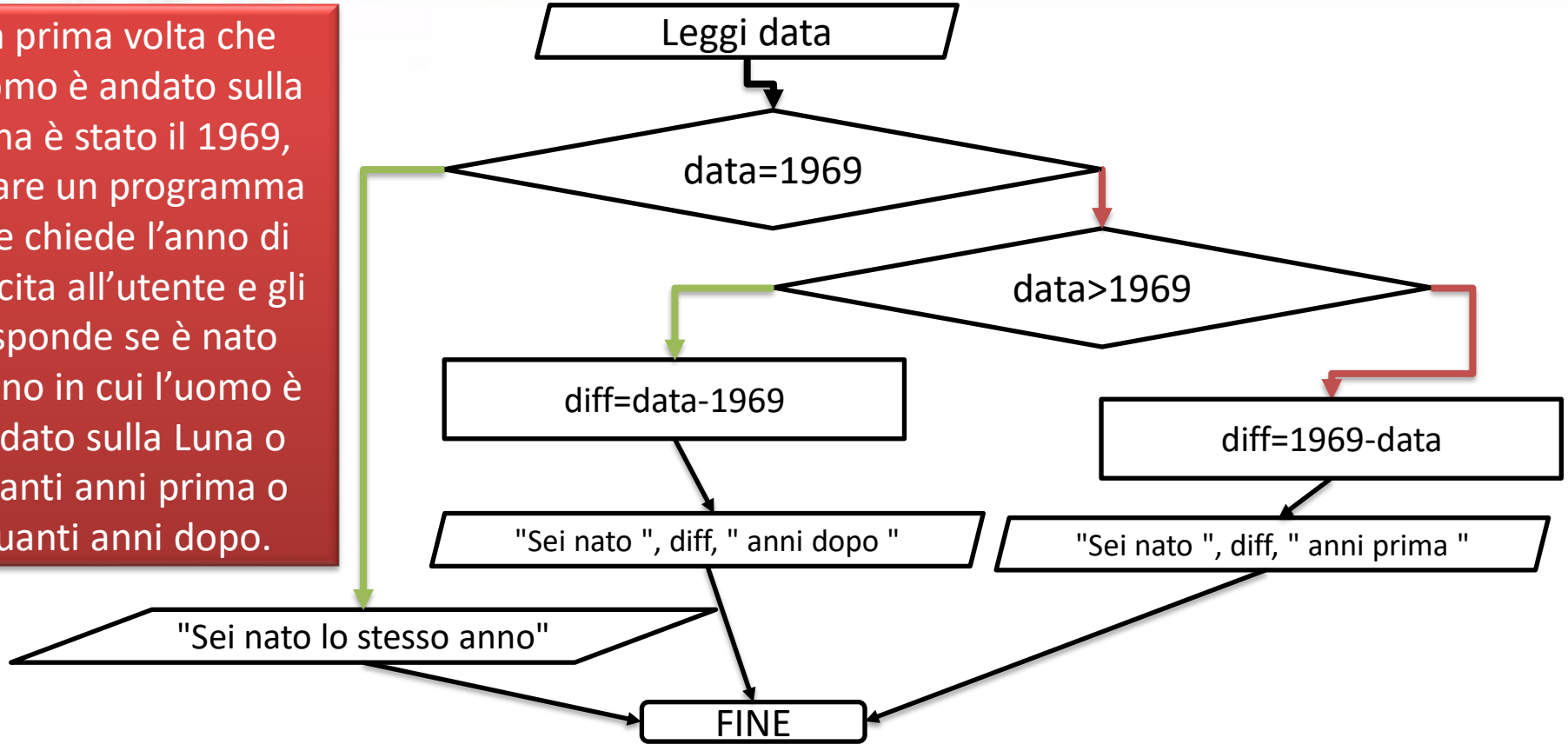
ESERCIZIO IN CLASSE

La prima volta che l'uomo è andato sulla Luna è stato il 1969, creare un programma che chiede l'anno di nascita all'utente e gli risponde se è nato l'anno in cui l'uomo è andato sulla Luna o quanti anni prima o quanti anni dopo.

PROGRAMMA

ESERCIZIO IN CLASSE

La prima volta che l'uomo è andato sulla Luna è stato il 1969, creare un programma che chiede l'anno di nascita all'utente e gli risponde se è nato l'anno in cui l'uomo è andato sulla Luna o quanti anni prima o quanti anni dopo.



PROGRAMMA

ESERCIZIO IN CLASSE

```
int anno;  int diff;
printf("\n Inserire anno di nascita: ");
scanf("%d",&anno);
if (anno==1969)  {
                printf("\nSei nato lo stesso anno dello sbarco sulla Luna");
                }
                else{
                                if (anno<1969) {
                                                                diff=1969-anno;
                                                                printf("Sei nato %d anni prima dello sbarco sulla Luna", diff);
                                                                }
                                else{
                                                diff=anno-1969;
                                                printf("Sei nato %d anni dopo dello sbarco sulla Luna", diff);
                                }
}
```

PROGRAMMA

ESERCIZIO IN CLASSE

Realizzare un programma che legge una serie di numeri interi positivi e si ferma quando o la somma di due numeri consecutivi è pari a 10 o quando un numero è uguale al precedente valore immesso

```
int precedente;
int successivo;
printf("\n Inserire un primo numero: ");
scanf("%d",&precedente);
printf("\n Inserire un secondo numero: ");
scanf("%d",&successivo);
while (!((successivo==precedente)||((precedente+successivo)==10)))          {
        precedente=successivo; // metto il valore del successivo come precedente
        printf("\n Inserire un altro numero: ");
        scanf("%d",&successivo);
        }
```

PROGRAMMA

Funzione

- Una funzione è un **sottoprogramma**
- Una funzione è caratterizzata da parametri di ingresso ed un tipo di dati in uscita
- La funzione va dichiarata prima della funzioni principale (MAIN)

- La sintassi è:

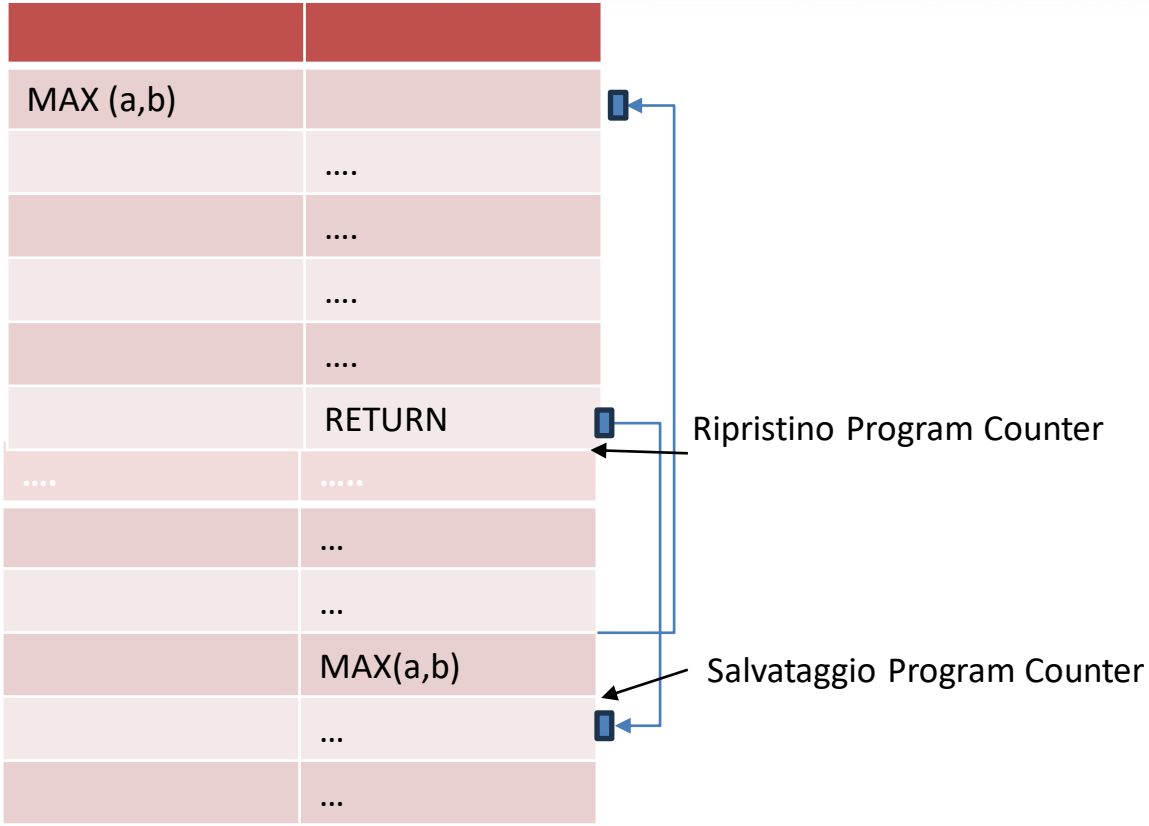
type nomefunzione (param1,parma2,…)
{
Corpo della funzione
}

FUNZIONE

```
#include <stdio.h>
int MAX(int val1,int val2){
int max;
if (val1>val2) { max=val1:}
          else { max=val2;}
return(max);
}

int main()
{
int x;int y;int z;int w;
printf("\nInserisci il primo numero:");
scanf("%d",&x);
printf("\nInserisci il secondo numero: ");
scanf("%d",&y);
w=MAX(x,y);
printf("\nIl massimo tra %d e %d è: %d",x,y,w);
printf("\nInserisci il terzo numero: ");
scanf("%d",&z);w=MAX(z,MAX(x,y));
printf("\nIl massimo tra %d e %d e %d è: %d",x,y,z,w);
scanf("%d",&w);
return 0;
}
```

PROGRAMMA

Funzione



Dal codice sorgente al codice eseguibile

PROGRAMMA

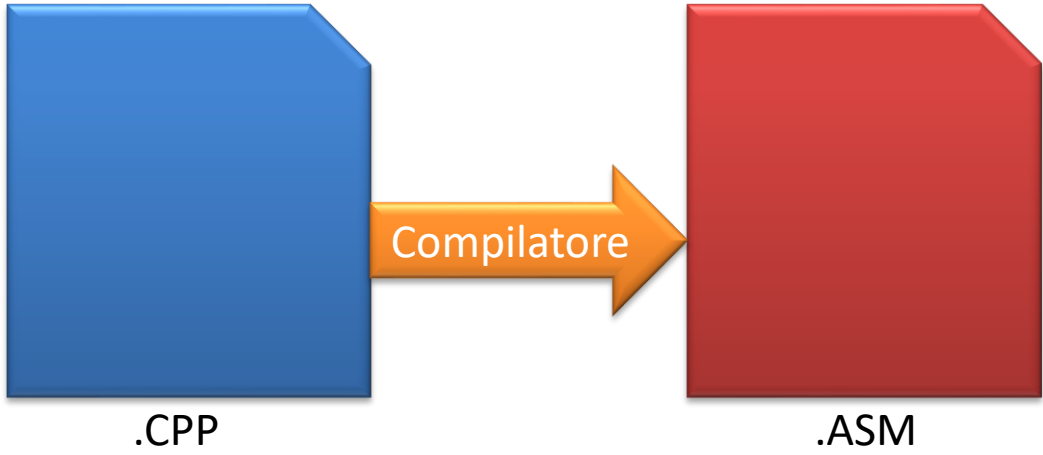
Generalità

- L'esecuzione di un programma è il punto di arrivo di una sequenza di azioni che nella maggior parte dei casi iniziano con la **scrittura di un programma in un linguaggio simbolico di alto livello**
- Le azioni principali che compongono tale sequenza nel caso in cui si parta da un linguaggio ad alto livello sono quelle che vedono l'impiego de **il compilatore, l'assemblatore, il collegatore** (linker)
- Alcuni calcolatori raggruppano queste azioni per ridurre il tempo di traduzione, ma concettualmente tutti i programmi passano sempre attraverso le fasi mostrate

PROGRAMMA

Compilatore

- Il **compilatore** trasforma, dopo un controllo sintattico, il programma scritto in un linguaggio ad alto livello in uno in linguaggio assembly, cioè in una forma simbolica che il calcolatore è in grado di capire ma, ancora, non eseguire



PROGRAMMA

Compilatore (esempio)

Linguaggio ad alto livello (linguaggio C)

```
Main ()
{
int ris=Pow (2,3);
}

int Pow(int b,int e)
{
        int t=1;
              for(i=0;i<e;i++)
                      {
                       t=t*b;
                      }
        return(t);
}
```

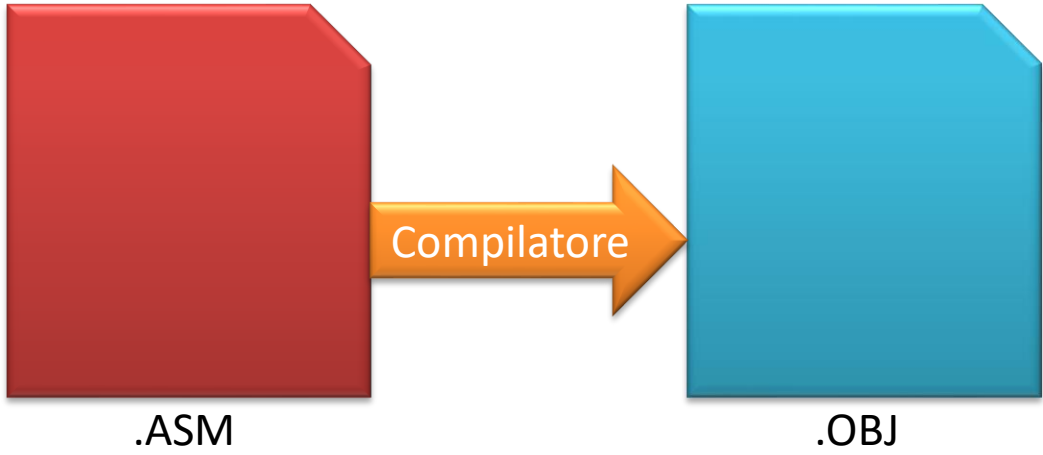
Linguaggio assembly (SPIM)

```
.text
.globl main
main:
lw $a0,base     #caricamento valore
lw $a1,espo     #caricamento valore
jal pow         #salto a funzione
sw $a2,ris      #spostamento risultato in memoria
li $v0,10
syscall
pow:
        li $t0,0    #inizializzazione contatore
        li $t1,1    #inizializzazione risultato temporaneo
        move $t3,$a0
    ciclo:
            bge $t0,$a1,fine #confronto contatore-esponente
            mul $t1,$t1,$t3  #moltiplicazione per la base
            addi $t0,1                  #incremento contatore
            j ciclo     #salto
fine:
move $a2,$t1
jr $ra                          #ritorno a funzione
.data                           #dichiarazione variabili
base: .word 2
espo: .word 3
ris: .word 0
```

PROGRAMMA

Assemblatore

- L'**assemblatore** converte un programma assembly in un file oggetto, che è una combinazione di istruzioni in linguaggio macchina, di dati e di informazioni necessarie a collocare le istruzioni in memoria nella posizione opportuna



PROGRAMMA

Collegatore (Linker)

- Quando un programma è costituito da più moduli contenuti in diversi file sorgenti, il processo di traduzione (compilazione e assemblaggio) è ripetuto per ciascun modulo
- I **file oggetto** (object) risultanti devono essere **collegati** (linked) opportunamente tra di loro all'interno di unico file eseguibile, che solo allora può essere caricato in memoria
- Per ogni modulo tradotto separatamente l'indirizzo iniziale è lo stesso: è compito del linker modificare gli indirizzi di ciascun modulo in modo che non ci siano sovrapposizioni

Modulo A	
000	...
001	x
...	...
150	Jsr B
...	...
200	...

Modulo B	
000	...
001	y
...	...
120	x
...	...
175	Jsr C
...	...
300	Ret A

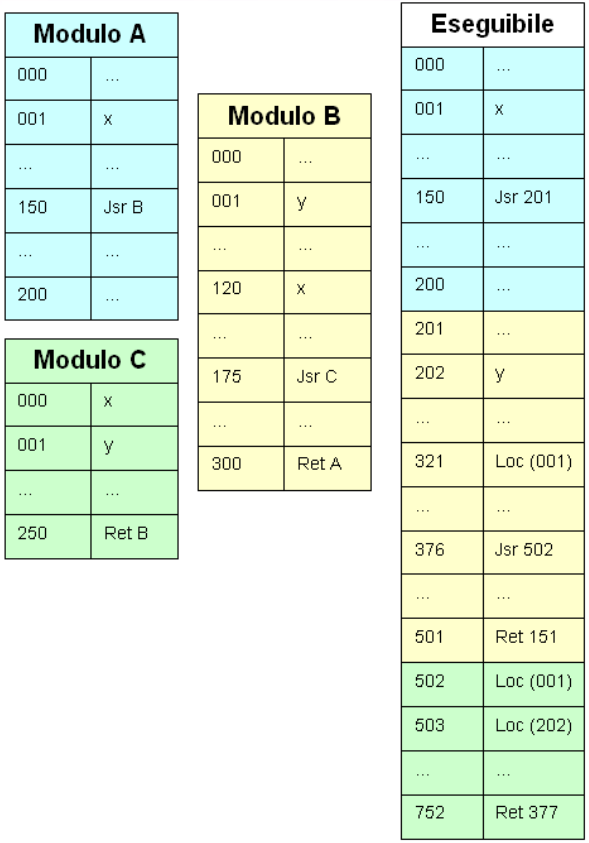
Modulo C	
000	x
001	y
...	...
250	Ret B

Eseguibile	
000	...
001	x
...	...
150	Jsr 201
...	...
200	...
201	...
202	y
...	...
321	Loc (001)
...	...
376	Jsr 502
...	...
501	Ret 151
502	Loc (001)
503	Loc (202)
...	...
752	Ret 377

PROGRAMMA

Librerie

- Il programmatore non ha necessità di programmare le funzioni di base (quali leggere un numero dalla tastiera, calcolare la radice quadrata, visualizzare una riga di testo, ecc.)
- Queste operazioni sono state programmate e compilate dal produttore del traduttore e sono a disposizione del programmatore sotto forma di funzioni(es. sqrt(x), calcola la radice quadrata; tan(x), calcola la tangente)
- I codici eseguibili(quindi già tradotti in linguaggio macchina) che realizzano queste funzioni vengono raggruppati in file detti librerie (collezioni di funzioni)



Modulo A	
000	...
001	x
...	...
150	Jsr B
...	...
200	...

Modulo B	
000	...
001	y
...	...
120	x
...	...
175	Jsr C
...	...
300	Ret A

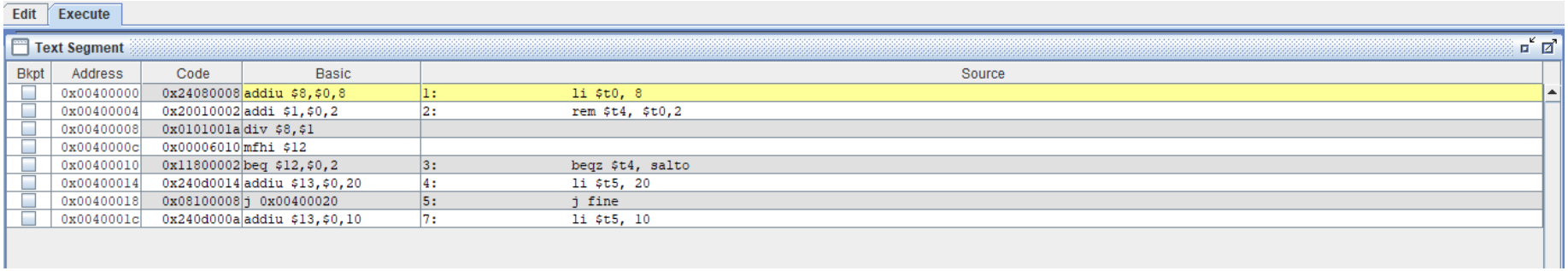
Modulo C	
000	x
001	y
...	...
250	Ret B

Eseguibile	
000	...
001	x
...	...
150	Jsr 201
...	...
200	...
201	...
202	y
...	...
321	Loc (001)
...	...
376	Jsr 502
...	...
501	Ret 151
502	Loc (001)
503	Loc (202)
...	...
752	Ret 377

PROGRAMMA

Collegatore (Linker)

- Il linker produce un **file eseguibile** che di norma ha la stessa struttura di un file oggetto, ma non contiene riferimenti non risolti



Bkpt	Address	Code	Basic		Source
	0x00400000	0x24080008	addiu $8,$0,8	1:	li $t0, 8
	0x00400004	0x20010002	addi $1,$0,2	2:	rem $t4, $t0,2
	0x00400008	0x0101001a	div $8,$1		
	0x0040000c	0x00006010	mfhi $12		
	0x00400010	0x11800002	beq $12,$0,2	3:	beqz $t4, salto
	0x00400014	0x240d0014	addiu $13,$0,20	4:	li $t5, 20
	0x00400018	0x08100008	j 0x00400020	5:	j fine
	0x0040001c	0x240d000a	addiu $13,$0,10	7:	li $t5, 10

PROGRAMMA

Eseguibile

Linguaggio assembly (MIPS)

```
.text
.globl main
main:
lw $a0,base
lw $a1,espo
jal pow
sw $a2,ris
li $v0,10
syscall
pow:
li $t0,0
li $t1,1
move $t3,$a0
ciclo:
                bge $t0,$a1,fine
                mul $t1,$t1,$t3
                j ciclo
fine:
move $a2,$t1
jr $ra
.data
base: .word 2
espo: .word 3
ris: .word 0
```

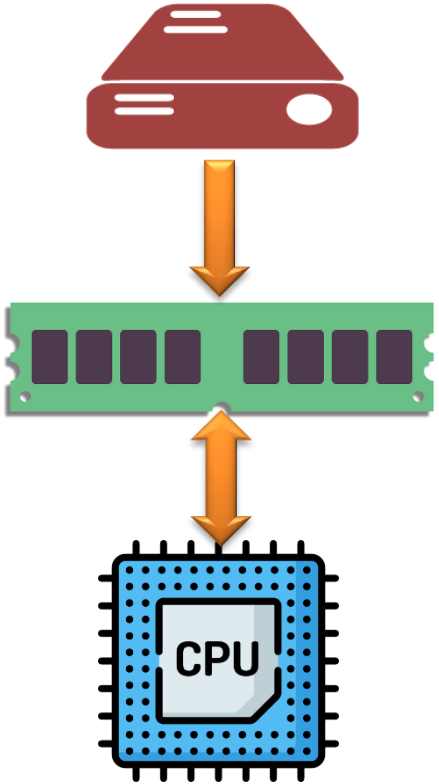
Linguaggio macchina (MIPS)
Area istruzioni

```
0x00000000  00000000000001001001100000101100
0x00000001  10001100001001000000000000000000
0x00000002  00111100000000010001000000000001
0x00000003  10001100001001010000000000000100
0x00000004  00001100000100000000000000001001
0x00000005  00111100000000010001000000000001
0x00000006  10101100001001100000000000001000
0x00000007  00110100000000100000000000001010
0x00000008  00000000000000000000000000001100
0x00000009  00000010000010000000010100000000
0x0000000a  00110100000010010000000000000001
0x0000000b  00000000000001000101100000100001
0x0000000c  00000001000001010000100000101010
0x0000000d  00010000001000000000000000000100
0x0000000e  01110001001010110100100000000010
0x0000000f  00100001000010000000000000000001
0x00000010  00001000000100000000000000001100
0x00000011  00000000000001001001100000100001
0x00000012  00000011111000000000000000001000
```

PROGRAMMA

Caricatore (loader)

- Una volta che il file eseguibile è memorizzato sul supporto di massa (generalmente il disco magnetico), il loader (un programma del Sistema operativo) può caricarlo in memoria per l'esecuzione



PROGRAMMA

Dal codice scritto all'eseguibile

- Un programma scritto in un linguaggio ad alto livello può essere editato usando un normale editor testuale (NOTEPAD, VI,Brackets, VIM, Komodo,…)
- Affinché questo diventi un file eseguibile necessita di essere letto da un programma compilatore/interprete
- Dopo la compilazione o l'interprete si genera un codice eseguibile

Ad esempio il comando (Linux) ./ nomefile si attiva il caricatore che trasferisce l'eseguibile in memoria e lo avvia

Creazione programma MAIN.c

main.c

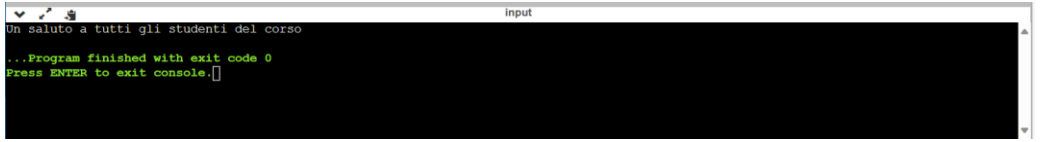
```
/*****************************************************************************
Franco Liberati
Esempio programma in C
02/10/2023 ore 12:00
*****************************************************************************/

#include <stdio.h>

int main()
{
    printf("Un saluto a tutti gli studenti del corso");

    return 0;
}

```

Compilazione di un programma C sotto Linux

```
franco@ubuntu: gcc main.c –o SalutoStudenti
```

Esecuzione di un programma C sotto Linux

```
franco@ubuntu: ./SalutoStudenti
```




Fine