Unione europea
Fondo sociale europeo

RegioneEmilia-Romagna

RETE POLITECNICA
La tecnica per crescere

FONDAZIONE ISTITUTO TECNICO SUPERIORE
MECCANICA, MECCATRONICA, MOTORISTICA E PACKAGING
Sede di BOLOGNA

"TECNICO SUPERIORE DEI SISTEMI DI CONTROLLO NELLA FABBRICA DIGITALE"

Rif.PA. 2017-7220/RER
Biennio 2017/2019
Progetto 1 Edizione 1

Modulo: *Programmazione Informatica*

Riallineamento – Linguaggi IEC61131

A cura di: *Matteo Sartini – Consorzio LIAM*

In collaborazione con:

LIAMLAB
RESEARCH - DEVELOPMENT - TECHNOLOGY

ITSMAKER
ISTITUTO TECNICO SUPERIORE

Indipendenza dalla piattaforma

Piattaforme per il controllo macchina

STANDARD

Costruttori di macchina (Emilia Romagna)

Linguaggi indipendenti dalla piattaforma

- Astrazione: un concetto o un'idea non associata a nessuna istanza specifica
  - Indica quanto il codice scritto in un linguaggio di programmazione si distacca dalle istruzioni in linguaggio macchina che ad esso corrisponderanno
  - **Meno complessità** nell'esprimersi e **maggiore** possibilità di **riutilizzo** dello stesso software

- **Linguaggio Macchina:**

```
0100 0000 0000 1000
0100 0000 0000 1001
0000 0000 0000 1000
```

Difficile leggere e capire un programma scritto in forma binaria

- **Linguaggio Assembler:**

```
… LOADA H
  LOADB Z
  ADD
  …
```

Le istruzioni corrispondono univocamente a quelle macchina, ma vengono espresse tramite nomi simbolici (parole chiave)

- **Linguaggi di Alto Livello:**

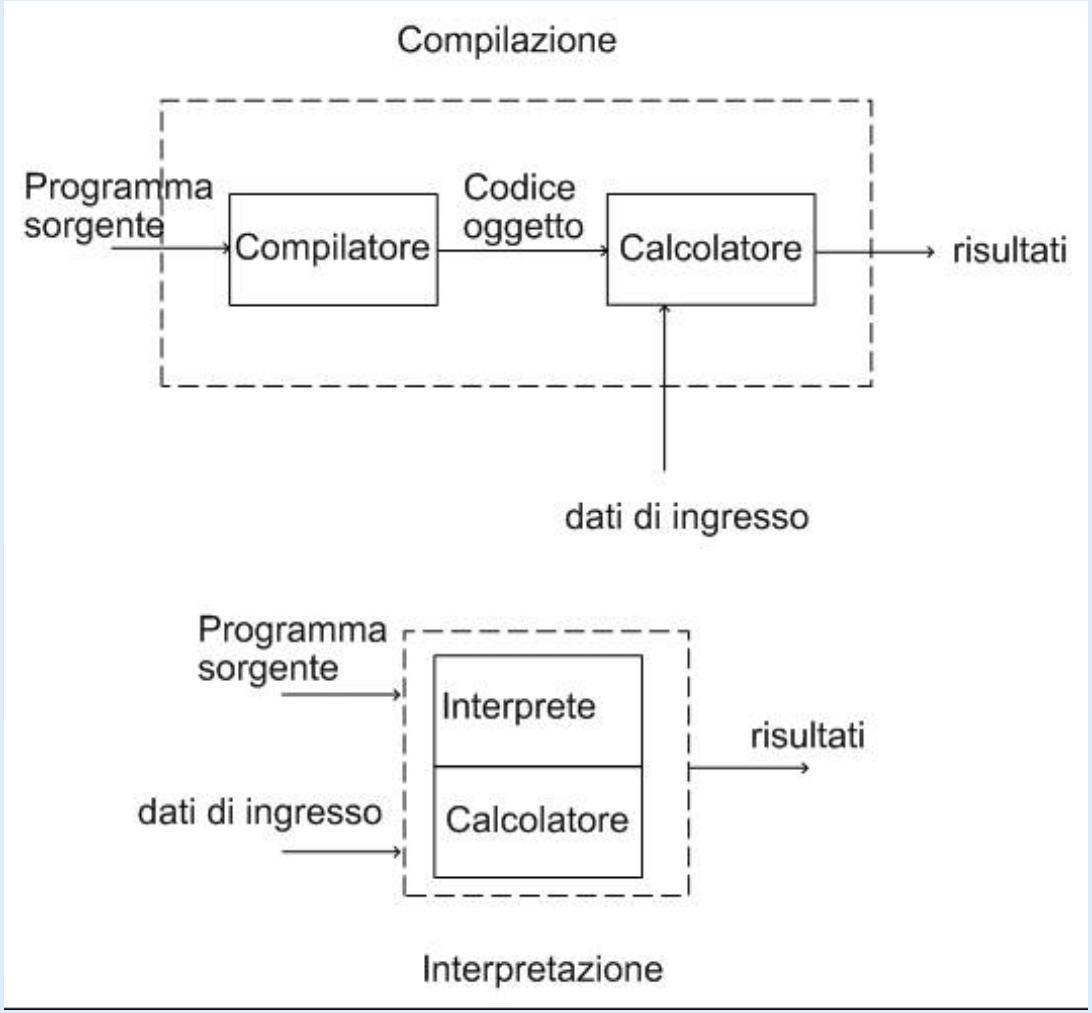
```
main()
{ int A;
  scanf("%d",&A);
  if (A==0) {…}
…}
```

Sono indipendenti dalla macchina

Linguaggi indipendenti dalla piattaforma

Per eseguire sulla macchina hardware un programma scritto in un linguaggio di alto livello è necessario tradurre il programma in sequenze di istruzioni di basso livello, direttamente eseguite dal calcolatore, attraverso:

- interpretazione (ad es. BASIC, Java)
  - Meno efficiente perché eseguito dall'interprete
- compilazione (ad es. C, FORTRAN, Pascal, … ST)
  - Più efficiente perché esegue direttamente sulla macchina



ESECUZIONE

- Per eseguire sulla macchina hardware un programma scritto in un linguaggio di alto livello è necessario tradurre il programma in sequenze di istruzioni di basso livello, direttamente eseguite dal processore, attraverso:
  - interpretazione (ad es. BASIC, Java)
    - Meno efficiente perché eseguito dall'interprete
  - compilazione (ad es. C, FORTRAN, Pascal, … ST)
    - Più efficiente perché esegue direttamente sulla macchina

LINGUAGGI DI PROGRAMMAZIONE

- Il "potere espressivo" di un linguaggio, quindi il suo livello di astrazione, è caratterizzato da:
  - **quali tipi di dati** consente di rappresentare (direttamente o tramite definizione dell'utente)
  - **quali istruzioni di controllo** mette a disposizione (quali operazioni e in quale ordine di esecuzione)

PROGRAMMA = DATI + CONTROLLO

Lo Standard IEC 61131-3

- Considerazioni
  - Parte della norma IEC61131 (1993, recepita in Italia nel 1996, revisione in 2003): standardizzazione PLC
    - IEC61131-1: Definizione PLC
    - IEC61131-2: Architettura HW/SW dei PLC (Real-time Time Driven)
    - IEC61131-3: Linguaggi di programmazione per implementare su PLC i controlli di sequenze
- **SCOPO**
  - Definire linguaggi standard per la programmazione di PLC
    - Stimolare una normalizzazione della sintassi dei linguaggi di programmazione per l'automazione
- **RISULTATO**
  - Vengono proposti 5 linguaggi (tanti?) molto diversi tra loro per questioni di retro compatibilità delle competenze
  - Linguaggi che tengono conto in modo diverso delle caratteristiche del PLC:
    - Esecuzione sequenziale
    - S.O. Real Time, Time Driven: “ciclo while intrinseco”

Lo Standard IEC 61131-3

- 5 linguaggi proposti
  - Grafici
    - Sequential Function Chart (**SFC**): Linguaggio grafico di alto livello dove è esplicito il concetto di stato
    - **Ladder** Diagram (LD): Linguaggio grafico di basso livello per la rappresentazione di schemi a contatti (relè)
    - Function Block Diagram (**FBD**): Linguaggio grafico per la rappresentazione di schemi a blocchi funzionali
  - Testuali:
    - **Instruction List** (IL, o AWL): Linguaggio testuale di basso livello simile all'Assembler
    - **Structured Test** (ST): Linguaggio testuale di alto livello simile al C

Ladder diagram

- Più vecchio e diffuso linguaggio di programmazione per PLC
  - facilmente comprensibile dai tecnici dell'epoca
    - Simboli di provenienza elettrica
    - Istruzioni che rappresentano schemi a contatti
    - Scarse o nulle conoscenze informatiche
- La forma deriva dalla **logica a Relais**
  - Due linee verticali laterali rappresentanti l'alimentazione
  - Linee orizzontali che alimentano la bobina se i contatti permettono il flusso di energia
  - I contatti sono associati agli ingressi digitali (variabili input) oppure a condizioni interne del dispositivo (variabili interne)
  - La bobina è associata ad un bit di memoria che può comandare una uscita digitale (variabile di output) oppure variare una condizione interna (variabile interna)

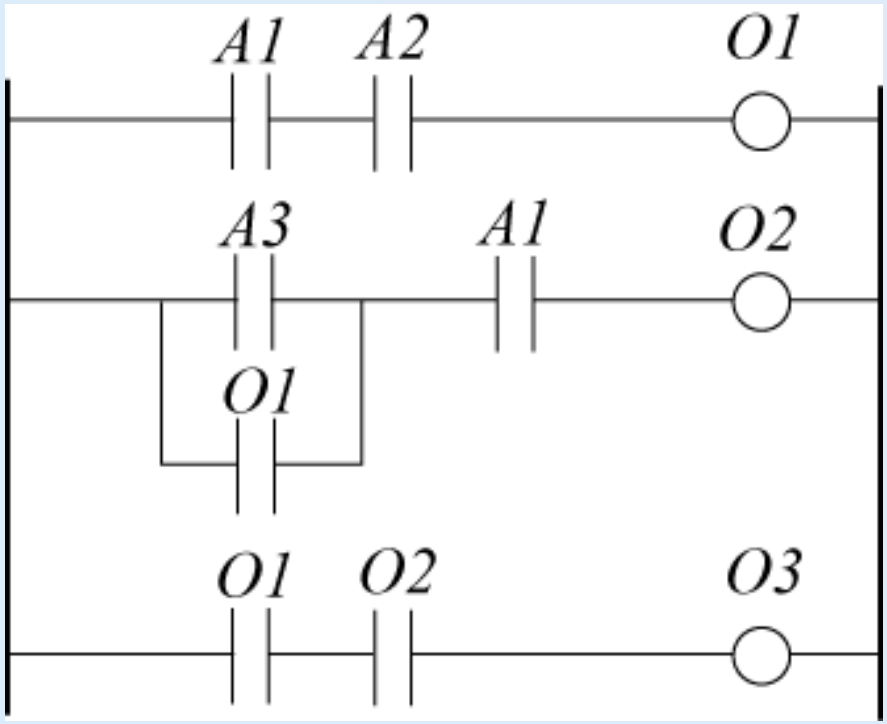
Esecuzione di un LD

- Un LD viene eseguito secondo una modalità ciclica composta dalle seguenti fasi:
  - Lettura degli input e scrittura del loro stato in locazioni di memoria particolari
  - Esecuzione del programma un rung dopo l'altro, procedendo **dall'alto verso il basso, da sinistra verso destra**
  - Scrittura delle uscite, prelevando il loro stato da locazioni di memoria particolari

Esempio ladder

- Regole principali
  - Il flusso di energia è diretto **sempre da sinistra a destra**
  - Le istruzioni sono eseguite dall'alto verso il basso
    - Ad ogni ciclo di scansione sono valutati tutti i rung

- Bobine e contatti sono le istruzioni di base del linguaggio a contatti.
- I contatti in un rung rappresentano le condizioni logiche da valutare per poter determinare lo stato che deve assumere l'uscita rappresentata dalla bobina.
- I contatti e le bobine devono essere sempre associati a bit di memoria oppure a nomi simbolici corrispondenti.



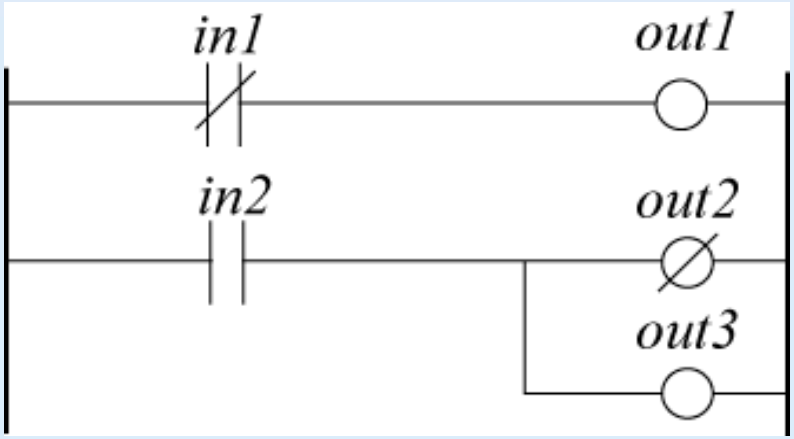
Esempio:

A1,A2,A3 input (Contatti)

O3 output (Bobina)

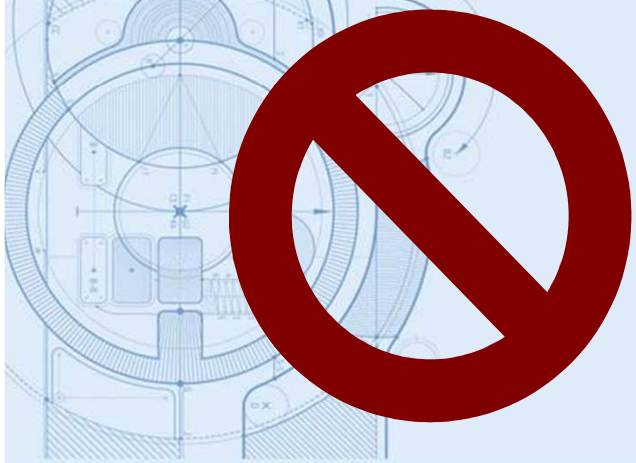
Contatti e bobine

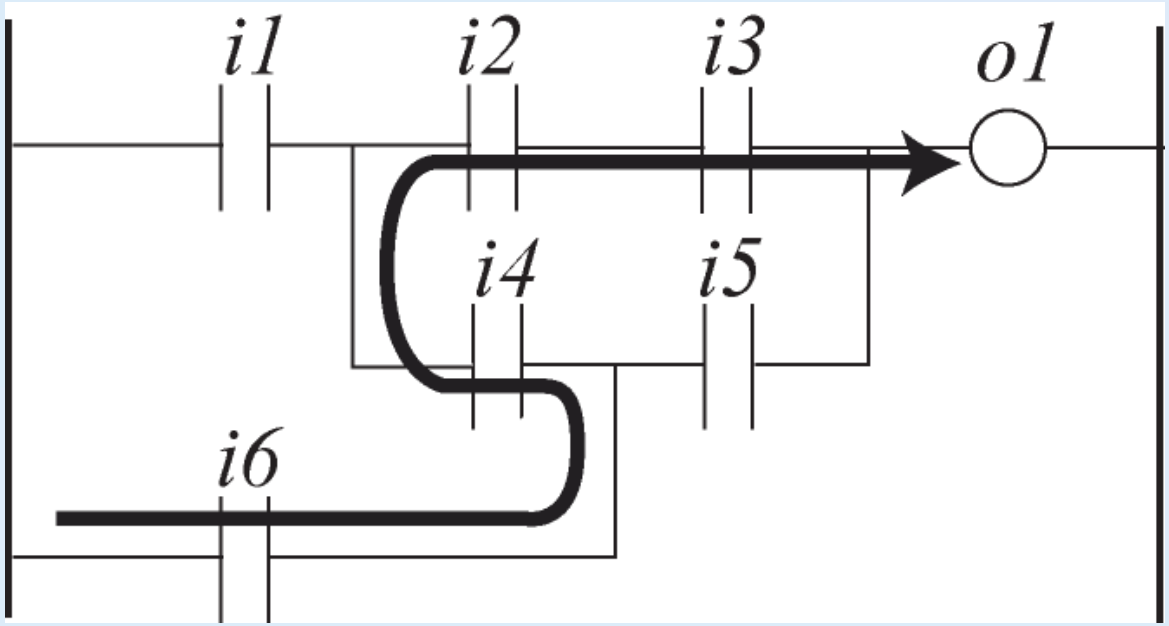
- **Contatto NA -| |-** può essere associato a un bit di input, di uscita, interno.
  - Se il bit associato vale 1 (ON) il processore chiuderà il contatto assicurando la continuità logica (elettrica) nel rung dove si trova. Se il bit vale 0 (OFF) il contatto rimarrà aperto.
- **Contatto NC -|/|-** è il duale del precedente.
- **Bobina –( )–** serve per controllare lo stato del bit a essa associato che può rappresentare un'uscita o un marker interno.
- L'istruzione deve essere sempre inserita sulla destra alla fine del rung: se le condizioni logiche alla sua sinistra sono verificate (esiste cioè una continuità logica/elettrica) il suo stato viene portato a 1 (ON), altrimenti è posto a 0 (OFF).



Ladder esempio

- Attenzione al flusso di “corrente”:

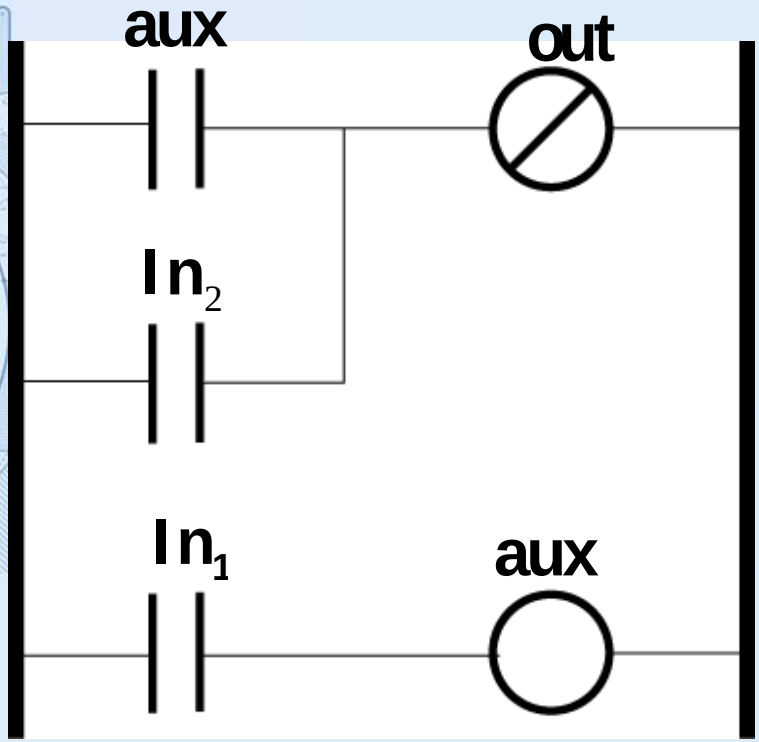




- Controllo sequenziale?

$$\mathrm{out}(k) = \mathrm{not}\left[\mathrm{In}_1(k-1)\ \mathrm{OR}\ \mathrm{In}_2(k)\right]$$

IMPLEMENTAZIONE di un controllo sequenziale

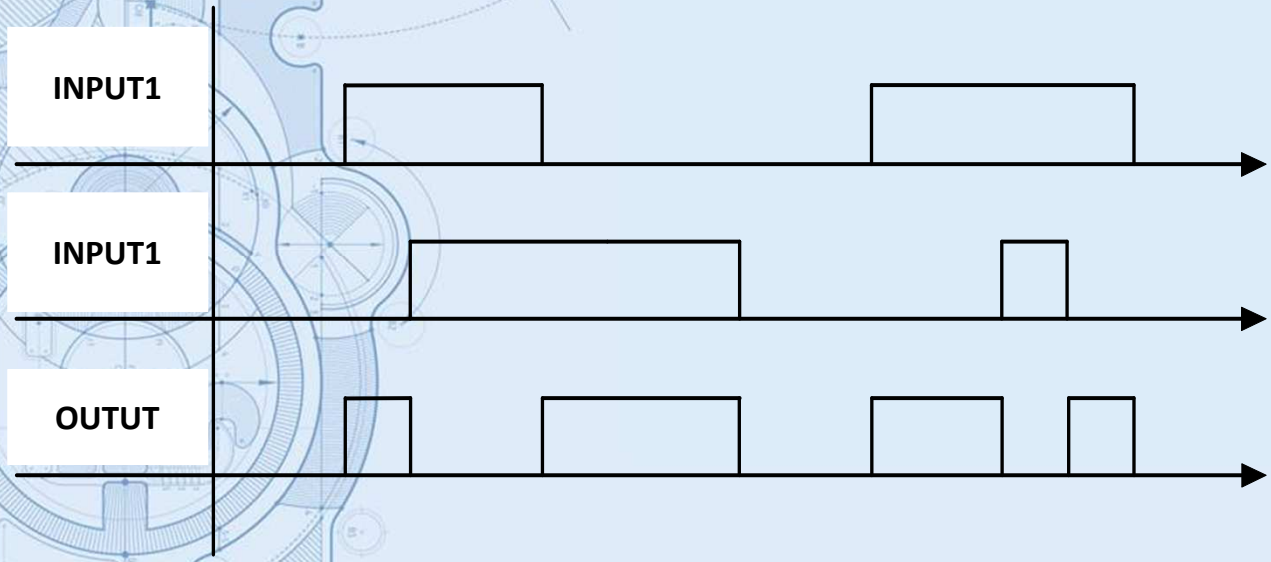


$$\text{not}\left(\text{out}\left(k\right)\right) = \text{aux}\left(k\text{-}1\right)\ \text{OR}\ \text{In}_2\left(k\right)$$

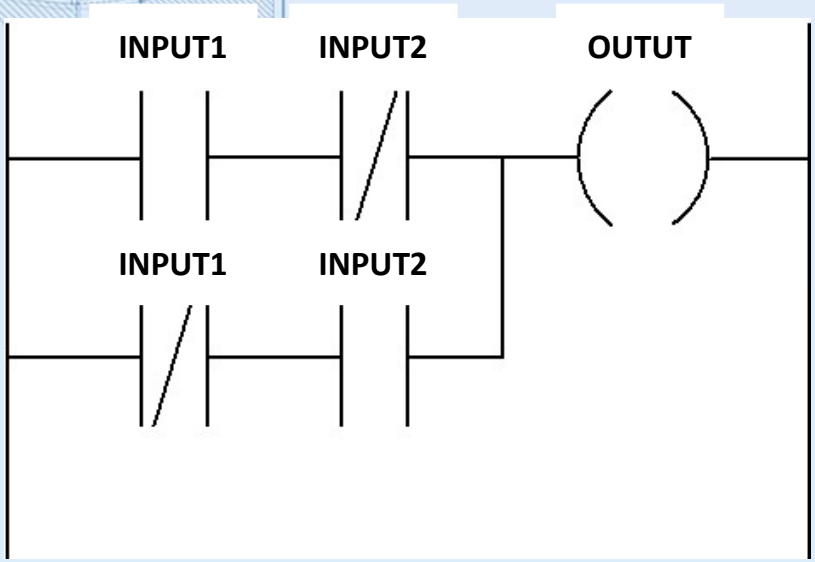
$$\text{aux}(k) = \text{In}_1(k)$$

$$\text{out}\left(k\right) = \text{not}\left[\text{In}_1\left(k\text{-}1\right)\ \text{OR}\ \text{In}_2\left(k\right)\right]$$

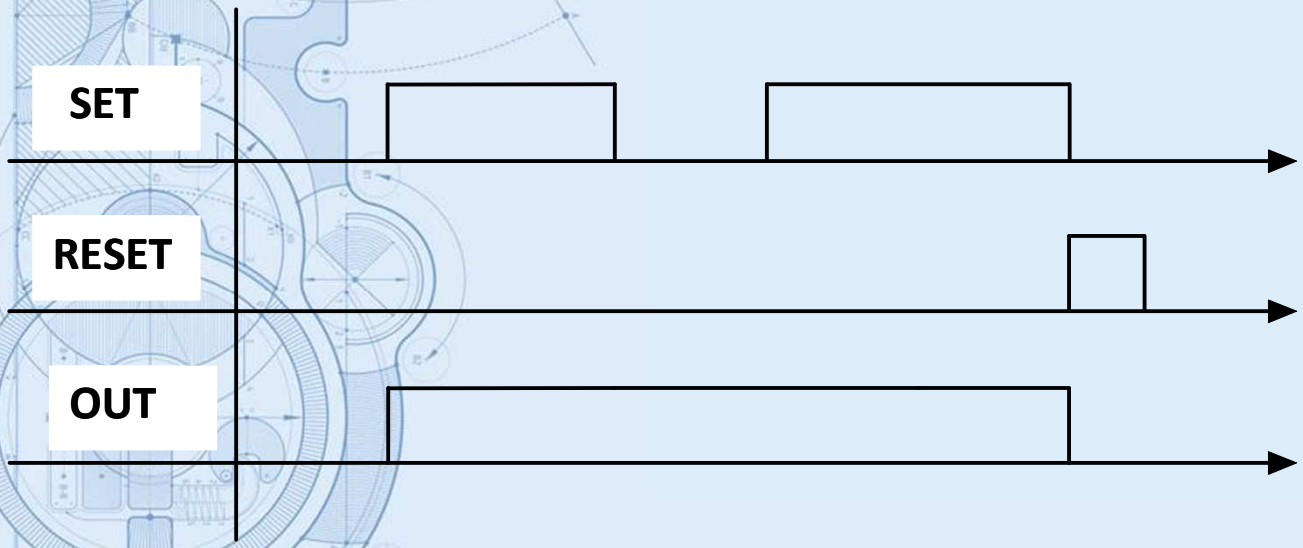
IMPLEMENTAZIONE di una XOR

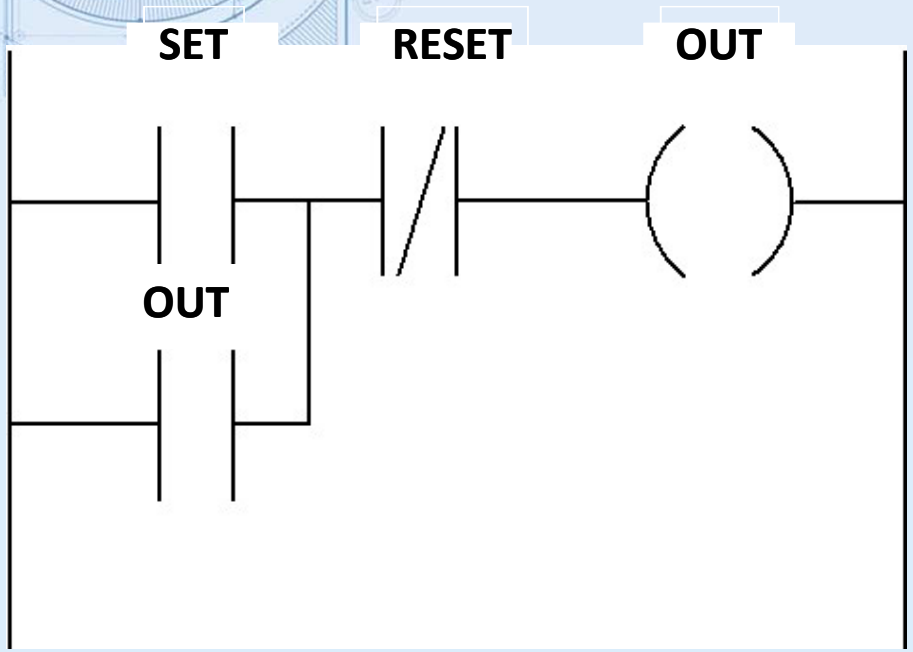


L’uscita è vera se uno dei due ingressi è vero ma non entrambi



IMPLEMENTAZIONE di un flip flop S/R





OUT = NOT RESET AND (SET OR OUT)

programma

- Introduzione ai linguaggi di programmazione
- Ladder
- **ST**
  - **Variabili e tipi di dato**
  - Dati strutturati
  - Operatori e istruzioni
  - Errori di programmazione
  - Strutture di controllo
    - Selezione
    - Iterazione
    - Funzioni (Function)
    - Componenti SW con memoria (Function Block)
- SFC

17

VARIABILI

- Una variabile è un'astrazione della cella di memoria
  - è un simbolo, detto identificatore, associato a un **indirizzo** fisico fisso e immutabile
  - che denota un **valore** che può cambiare

Identificatore	Indirizzo	Valore
AnniLuca	1328	28

VARIABILI

- Per usare una variabile (leggerla o modificarla) la si deve prima **dichiarare**
  - Nome della variabile (identificatore)
  - **Tipo di dato** che stabilisce la classe di valori che può assumere (necessario per l'allocazione di memoria che deve essergli dedicata)

```
AnniLuca    : INT; (* ES.1 *)
AnniStefano : INT; (* ES.2 *)
```

VARIABILI: UTILIZZO

- Una volta dichiarata la variabile può essere usata
  - nella parte di programma in cui è visibile (scope)
- Compatibilmente al suo tipo, essa può:
  - assumere il valore risultante dalla valutazione di una espressione (**assegnamento**: solo il valore cambia!)
  - far parte di una espressione

```
AnniLuca    := 28;      (* ES.1 *)
AnniStefano := Luca - 2; (* ES.2
*)
```

ELABORATORE come manipolatore di simboli

- L'architettura fisica di ogni elaboratore è intrinsecamente capace di trattare vari domini di dati, detti tipi primitivi (o elementari)
  - dominio dei numeri naturali e interi
  - dominio dei numeri reali (con qualche approssimazione)
  - dominio dei caratteri
  - dominio delle stringhe di caratteri

Come esprimere valori numerici

No.	Feature description	Examples
1	Integer literals	-12 0 123_456 +986
2	Real literals	-12.0 0.0 0.4560 3.14159_26
3	Real literals with exponents	-1.34E-12 or -1.34e-12 1.0E+6 or 1.0e+6 1.234E6 or 1.234e6
4	Base 2 literals	2#1111_1111 (255 decimal) 2#1110_0000 (224 decimal)
5	Base 8 literals	8#377 (255 decimal) 8#340 (224 decimal)
6	Base 16 literals	16#FF or 16#ff (255 decimal) 16#E0 or 16#e0 (224 decimal)
7	Boolean zero and one	0 1
8	Boolean FALSE and TRUE	FALSE TRUE
9	Typed literals	DINT#5 (DINT representation of 5) UINT#16#9AF (UINT representation of the hexadecimal value 9AF) BOOL#0 BOOL#1 BOOL#TRUE BOOL#FALSE
NOTE The keywords FALSE and TRUE correspond to Boolean values of 0 and 1, respectively.		

Come esprimere CARATTERI

No.	Example	Explanation
1	**Single-byte character strings**	
	`''`	Empty string (length zero)
	`'A'`	String of length one containing the single character A
	`' '`	String of length one containing the “space” character
	`'$''`	String of length one containing the “single quote” character
	`'"'`	String of length one containing the “double quote” character
	`'$R$L'`	String of length two containing CR and LF characters
	`'$0A'`	String of length one containing the LF character
	`'$$1.00'`	String of length five which would print as “$1.00”
	`'ÄË'` `'$C4$CB'`	Equivalent strings of length two

Come esprimere durata e ora

Table 7 - Duration literal features

No.	Feature description	Examples
1a	Duration literals without underlines: short prefix	`T#14ms T#-14ms T#14.7s T#14.7m T#14.7h t#14.7d t#25h15m t#5d14h12m18s3.5ms`
1b	long prefix	`TIME#14ms TIME#-14ms time#14.7s`
2a	Duration literals with underlines: short prefix	`t#25h_15m t#5d_14h_12m_18s_3.5ms`
2b	long prefix	`TIME#25h_15m` `time#5d_14h_12m_18s_3.5ms`

Table 9 - Examples of date and time of day literals

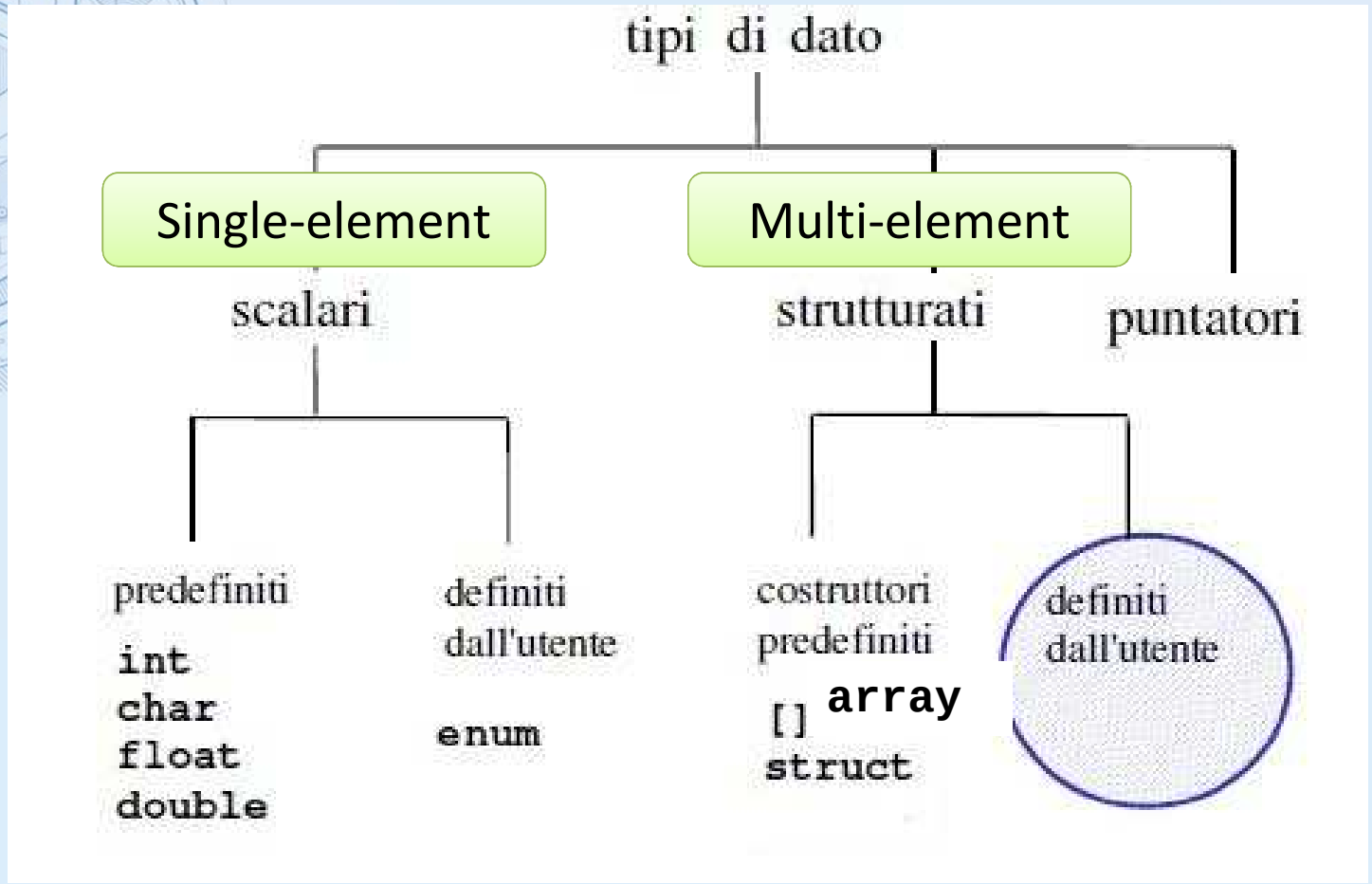
Long prefix notation	Short prefix notation
`DATE#1984-06-25` `date#1984-06-25`	`D#1984-06-25` `d#1984-06-25`
`TIME_OF_DAY#15:36:55.36` `time_of_day#15:36:55.36`	`TOD#15:36:55.36` `tod#15:36:55.36`
`DATE_AND_TIME#1984-06-25-15:36:55.36` `date_and_time#1984-06-25-15:36:55.36`	`DT#1984-06-25-15:36:55.36` `dt#1984-06-25-15:36:55.36`

TIPO di dato

- Come si fa per dire che una variabile è un intero o un reale, etc… ?
  - TIPO DI DATO
- Il concetto di tipo di dato viene introdotto per raggiungere due obiettivi:
  - esprimere in modo sintetico la loro rappresentazione in memoria, e un insieme di operazioni ammissibili
  - permettere di effettuare controlli statici (al momento della compilazione) sulla correttezza del programma

TIPO di dato

- I tipi di dato si differenziano in *scalari* e *strutturati*



TiPI di dato elementari (o predefiniti)

Tipo di dato	C		Structured Text (ST)	
Caratteri (solitamente codifica ASCII a 8 bit)	CHAR	8 bit	STRING	8 bit
	UNSIGNED CHAR	8 bit	WSTRING	16 bit
Interi con segno *Intervallo di valori ammessi:* *[-2^Nbit, +(2^Nbit)-1]*	SHORT	16 bit	SINT	8 bit
	INT	32 bit	INT	16 bit
	LONG	32 bit o 64 bit a seconda del processore e del compilatore	DINT	32 bit
			LINT	64 bit
Naturali (interi senza segno) *Intervallo di valori ammessi:* *[0, +(2^Nbit)-1]*	UNSIGNED SHORT	16 bit	USINT	8 bit
	UNSIGNED	32 bit	UINT	16 bit
	UNSIGNED LONG	32 bit o 64 bit a seconda del processore e del compilatore	UDINT	32 bit
			ULINT	64 bit
Reali	Float	32 bit (*10^-38 … 10^38 circa)*	REAL	32 bit
	Double	64 bit (*10^-308 a 10^308 circa*)	LREAL	64 bit
Binario	NO (uso di INT)		BOOL	1 bit

TiPI di dato elementari (o predefiniti)

Tipo di dato	C		Structured Text (ST)	
Stringhe di bit	NO		**BYTE**	8 bit
			WORD	16 bit
			DWORD	32 bit
			LWORD	64 bit
Tempo	NO		**TIME** per esprimere una durata Ad esempio: TIME#5s	Dipende dalla piattaforma
			DATE Per esprimere una data Ad esempio: D#1984-06-25	Dipende dalla piattaforma
			TOD (o TIME_OF_DAY) Per esprimere un orario del giorno Ad esempio: TOD#15:05:34.10	Dipende dalla piattaforma
			DT (o DAY_AND_TIME) Per esprimere un orario in un dato giorno Ad esempio: DT#1984-06-25-15:36:55.36	Dipende dalla piattaforma

Caso particolare: la stringa

- In C non esiste il concetto di stringa
  - Si costruisce mettendo uno dietro l'altro N caratteri
- In ST esiste il concetto di stringa e non quello di carattere
  - Di default la stringa è lunga 80 caratteri

```
Nome    : STRING;       (* 80 caratteri
*)
Cognome : STRING[100]; (* 100 caratteri
*)
```

VALORI DI DEFAULT dei tipi di dato elementari

Data type(s)	Initial value
BOOL, SINT, INT, DINT, LINT	0
USINT, UINT, UDINT, ULINT	0
BYTE, WORD, DWORD, LWORD	0
REAL, LREAL	0.0
TIME	T#0S
DATE	D#0001-01-01
TIME_OF_DAY	TOD#00:00:00
DATE_AND_TIME	DT#0001-01-01-00:00:00
STRING	'' (the empty string)
WSTRING	"" (the empty string)

Tipi di dato derivati

- Sono definiti dall'utente
- Diverse tipologie per diversi utilizzi
  - Scalari (Single-element)
    - Derivazione diretta
    - Definizione di intervallo
    - Enumerativi
  - Strutturati (Multi-element)
    - Struttura
    - Array

Tipi di dato derivati

- Definito con il costrutto **TYPE…END_TYPE**

```
TYPE Messaggio:
   (ERRORE,
    ANOMALIA,
    INFO)
END_TYPE
```

- Una variabile assume questo tipo nella dichiarazione

```
MessaggioOperatore : Messaggio := INFO;
```

- e viene usata come le altre variabili

```
MessaggioOperatore := ERRORE;
```

Derivazione diretta ed enumerativi

- Una derivazione diretta può essere usata per cambiare nome al tipo oppure definire degli intervalli di validità dei valori
- Gli *enumerativi* definiscono un insieme finito di identificatori, utili per una maggiore leggibilità del codice

Di default è un elenco di interi da 0 a seguire con incremento di 1

No.	Feature/textual example
1	Direct derivation from elementary types, e.g.: `TYPE RU_REAL : REAL ; END_TYPE`
2	Enumerated data types, e.g.: `TYPE ANALOG_SIGNAL_TYPE : (SINGLE_ENDED, DIFFERENTIAL) ; END_TYPE`
3	Subrange data types, e.g.: `TYPE ANALOG_DATA : INT (-4095..4095) ; END_TYPE`
4	Array data types, e.g.: `TYPE ANALOG_16_INPUT_DATA : ARRAY [1..16] OF ANALOG_DATA ; END_TYPE`

DERIVAZIONE DIRETTA ED ENUMERATIVI: UTILIZZO

- **Enumerativo** (già visto anche l'esempio del messaggio…)

```
TYPE LCD_TYPE :
   (LED,
   PLASMA);
```

LED = 0

PLASMA = 1

```
T : LCD_TYPE ;
```

```
T := PLASMA;
```

- **Derivazione diretta**

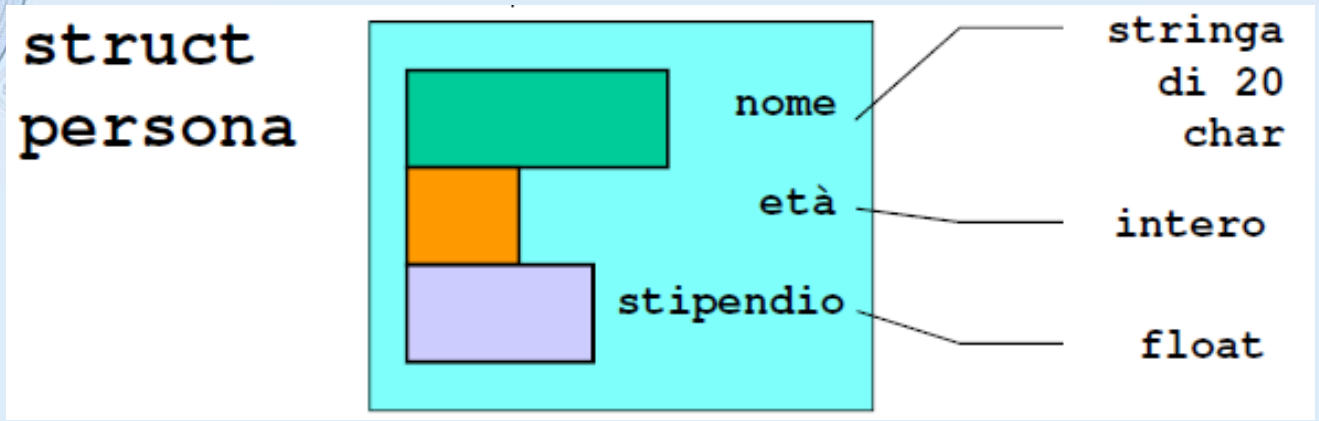
```
TYPE FREQ : REAL := 50.0 ; END_TYPE
```

```
F,G : FREQ;
```

```
F := 10;
G := F + 20;
```

STRUTTURA

- Una struttura è una collezione finita di variabili non necessariamente dello stesso tipo, ognuna identificata da un nome



STRUTTURA

- Definite con il costrutto **STRUCT...END_STRUCT**, possono contenere più tipi di dato alla volta
- Sono lo strumento che permette al programmatore di modellare concetti complessi presenti nel mondo reale

5 Structured data types, e.g.:

```
TYPE
  ANALOG_CHANNEL_CONFIGURATION :
    STRUCT
      RANGE : ANALOG_SIGNAL_RANGE ;
      MIN_SCALE : ANALOG_DATA ;
      MAX_SCALE : ANALOG_DATA ;
    END_STRUCT ;
  ANALOG_16_INPUT_CONFIGURATION :
    STRUCT
      SIGNAL_TYPE : ANALOG_SIGNAL_TYPE ;
      FILTER_PARAMETER : SINT (0..99) ;
      CHANNEL : ARRAY [1..16] OF  ANALOG_CHANNEL_CONFIGURATION ;
    END_STRUCT ;
END_TYPE
```

Struttura: utilizzo

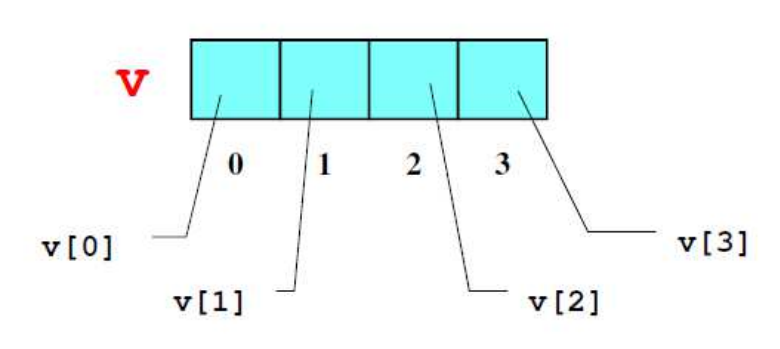
```
TYPE TV :
  STRUCT
    Frequenza  : FREQ;
    Dimensione : REAL;
    LcdType  : LCD_TYPE;
  END_STRUCT
END_TYPE
```

```
MioTV : TV;
```

```
MioTV.Frequenza  := 100;
MioTV.Dimensione := 80.2;
MioTV.LcdType    := LED;
```

ARRAY

- Un array è una collezione finita di N variabili dello stesso tipo, ognuna identificata da un indice compreso fra 0 e N-1



- La dimensione fisica N è decisa staticamente all'atto della definizione della variabile di tipo array

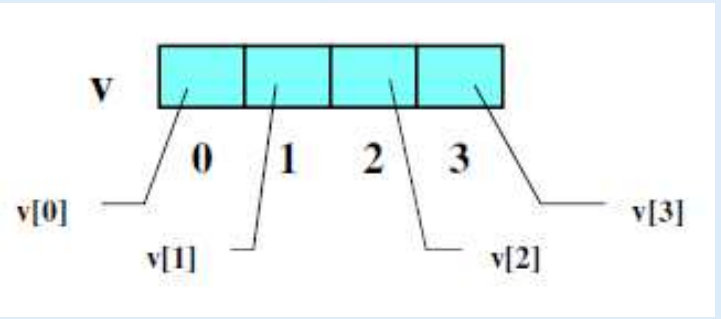
ARRAY

- Definizione di una variabile di tipo array:

```
NomeVariabile : ARRAY [Costante1..Costante2] OF
TipoDato


dove Costante1 <= Costante2
```

```
V : ARRAY [0..3] OF INT;
```

```
W : ARRAY [-5..5] OF INT;
```




```
N : INT;
V : ARRAY [0.. N] OF INT;
```

NO! il compilatore non saprebbe quanta

memoria allocare per l'array

ARRAY MULTI-DIMENSIONALE

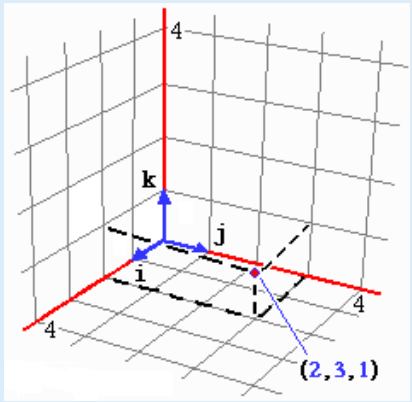
- Un array può essere bidimensionale (matrice), tridimensionale, … N-dimensionale.
  - Attenzione all'esplosione della dimensione e alla conseguente occupazione della memoria!

```
TWO : ARRAY [1..3, 1..4] OF
INT;
```

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} 9 & 13 & 5 \\ 1 & 11 & 7 \\ 3 & 7 & 2 \\ 6 & 0 & 7 \end{bmatrix}$$

```
THREE : ARRAY[1..5,1..10,1..8] OF
INT;
```



OPERATORI IN STRUCTURED TEXT

In Codesys non esiste l'operatore **

Operatore	Simbolo	Precedenza	Esempio
Parentesi	**(** *espressione* **)**	**PIU' ALTA**	A + (B – C)
Valutazione di una funzione			MAX(X,Y))
Complemento (negazione)	**NOT** (-)		NOT A
Esponente	** (oppure **EXPT(A,B)**)		A**B
Prodotto	*		X*Y
Divisione	/		X/Y
Modulo	**MOD**		X MOD Y
Somma	+		X + Y
Sottrazione	-		X – Y
Comparazione	< , > , <= , >=		X >= Y
Uguaglianza	=		X = Y
Disuguaglianza	<>		X <> Y
Prodotto logico	**AND**		A AND B
OR esclusivo	**XOR**		A XOR B
Somma logica	**OR**	**PIU' BASSA**	A OR B

Aritmetici

Relazionali

Logici

TIPOLOGIE DI ESPRESSIONI

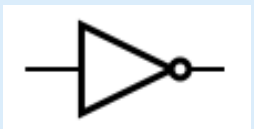
- Aritmetiche

```
Z := 8*X-Y;     (* Lineari *)
Z := 7*X**2+4*X-10;   (* Quadratica *)
Z := (X-1)/(Y+12); (* Razionale *)
```

- Logiche
  - Algebra booleana
    - 1 = TRUE
    - 0 = FALSE

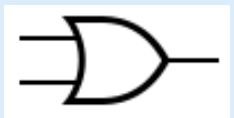
```
D := A OR B AND C;
D := (A OR B) AND C;
D := A XOR NOT B;
```

A	NOT A
0	1
1	0



A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1



A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

TIPOLOGIE DI ESPRESSIONI

- Relazionale
  - esprimere condizioni

```
D := A >= B;
D := A <> B;
D := A = B;
```

- Miste

```
Z1 := (8*X)>(5*Y);                    (* Es.1 … tipo di Z1? *)
Z2 := (A AND B) OR (X+1 = Y);          * Es.2 … tipo di Z2? *)
Z3 := 1.2 * ABS(B) + 5;               (* Es.3 … tipo di Z3? *)


Z4 := SIN(A) * MiaFunzione(C:=1,B:=5); (* Es.4 … tipo di
Z4? *)


Z5 := (A > B) XOR (C < D);                (* Es.5 … tipo di
Z5? *)
```

errori

- Un programma può contenere varie tipologie di errori riscontrabili in fasi diversi dello sviluppo:
  - In fase di scrittura
  - In fase di compilazione (compile time)
    - Durante la traduzione in linguaggio macchina si verifica che la sintassi sia corretta (controllo statico) ovvero che sia scritto giusto con rispetto dei tipi
  - In fase di esecuzione (runtime)
    - Il controllo in questo caso è dinamico, perché non si può prevedere a priori che ci si presenti e deve prevedere un gestione dello stesso
- Dipende anche dal supporto del compilatore e dell'ambiente di sviluppo

Esempi di ERRORI

- Divisione per zero

```
A, B, C, D : LREAL;
C := A / B; (* se B = 0, errore runtime*)
D := D / 0; (* errore di compilazione *)
```

- Conversione di tipo

```
A : INT;
B : LREAL;
A := B; (* errore di compilazione*)
```

- Il valore di una variabile non è compreso nell'intervallo dei valori ammessi

Codesys non dà errore!!!

```
TYPE Length : INT (1..50); END_TYPE
L : Length;
A : INT;

L := A;  (* se A > 50, errore runtime*)
L := 70; (* errore di compilazione *)
```

ASSEGNAMENTO

- Sostituisce il valore corrente di una variabile elementare o strutturata con il risultato della valutazione di una espressione

```
A, B, C, D, R : INT;

R := A+B-C*ABS(D);
```

```
MioTV, NuovoTV : TV;

MioTV := NuovoTV;
```

- La valutazione di una espressione restituisce un valore … che viene assegnato alla variabile avente indirizzo specificato dall'identificatore alla sinistra dell'operatore assegnamento (:=)

Overloading in assegnamento

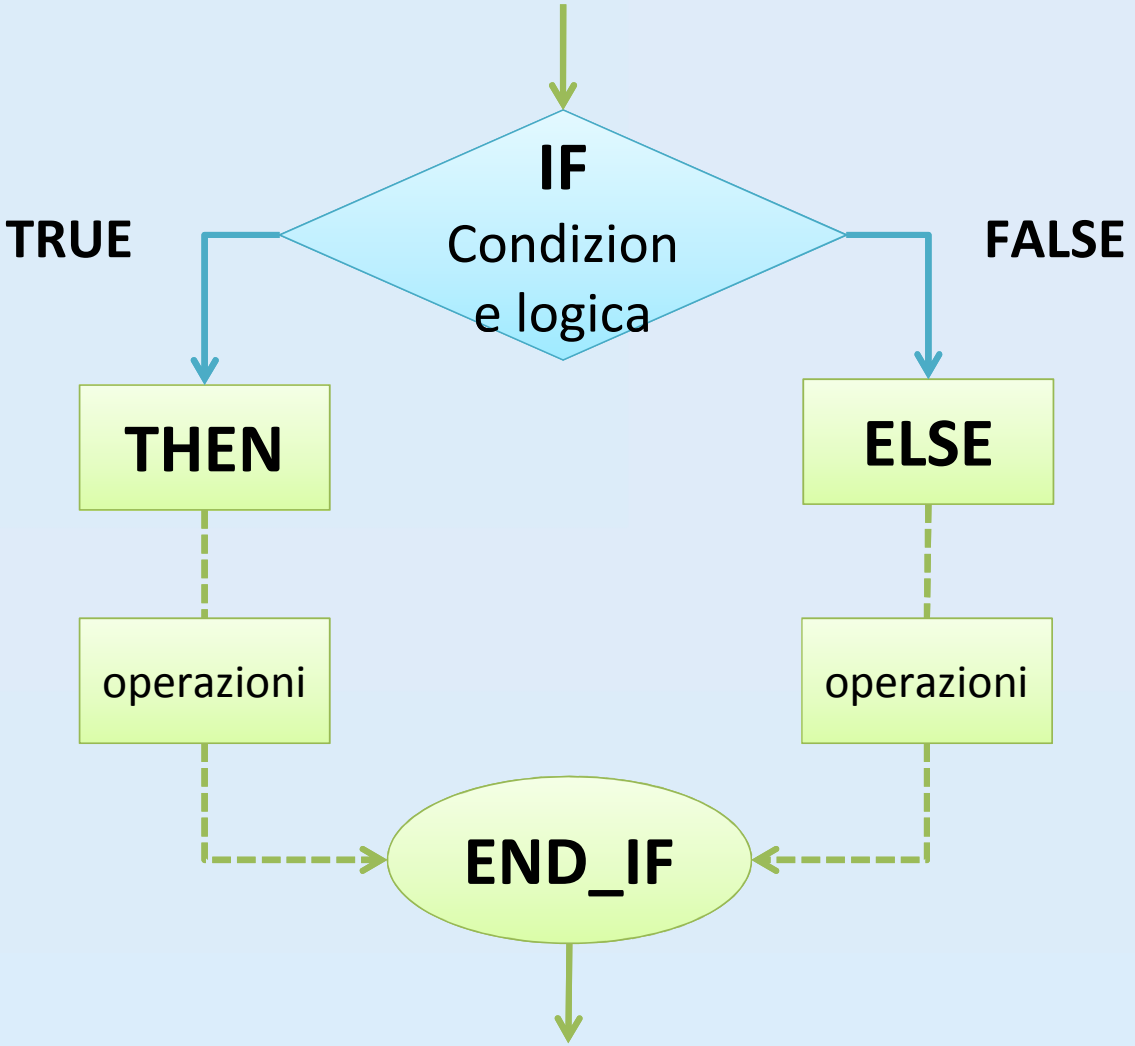
- In un assegnamento, l'identificatore di variabile e l’espressione devono essere dello stesso tipo
  - Nel caso di tipi diversi, se possibile il compilatore effettua automaticamente la conversione implicita
- Altrimenti, anche se l’espressione non è illegale, si ottiene un errore di compilazione, per evitare che si verifichino perdite di informazione
  - Queste regole dipendono dal compilatore e dalle specifiche del linguaggio di programmazione…
    - In generale occhio alla compatibilità dei tipi!

FLUSSO DI CONTROLLO

- Un problema di calcolo può essere risolto eseguendo una serie di operazioni in un ordine opportuno
  - ***Algoritmo***: procedimento di soluzione in termini di
    - azioni che devono essere eseguite
    - Ordine in cui queste azioni devono essere eseguite
- **Controllo del programma** (o del flusso di esecuzione)
  - Specifica l'ordine con cui le azioni devono essere eseguite

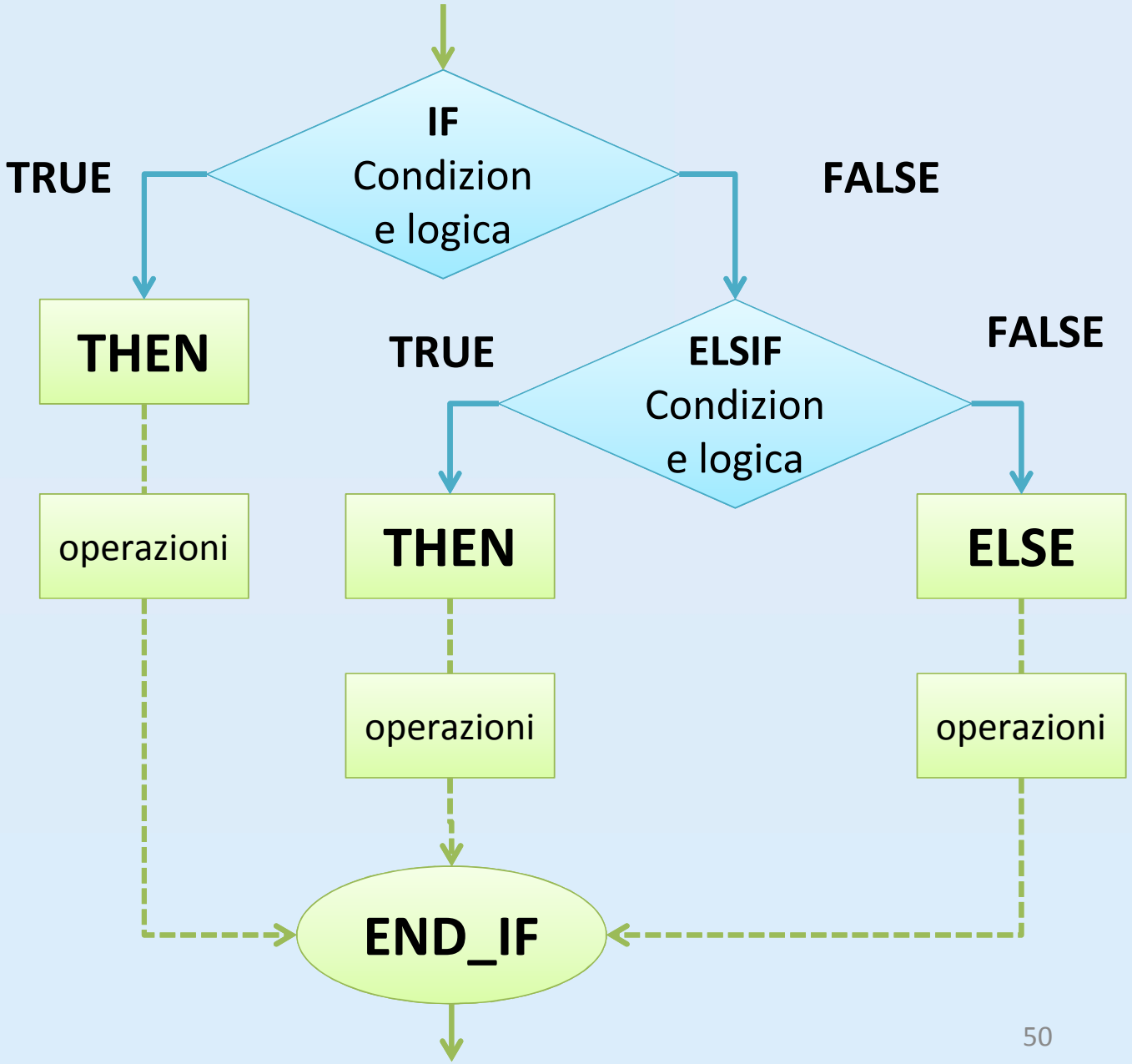
SELEZIONE: IF THEN ELSE

```
IF ( A > 20)
THEN
   B := 1;
ELSE
   C := A + B;
END_IF;
```



SELEZIONE: IF annidati

```
IF ( A > 20)
THEN
   B := 1;
ELSIF ( A < 0 )
THEN
   C := A + B;
ELSE
   C := A - B;
END_IF;
```

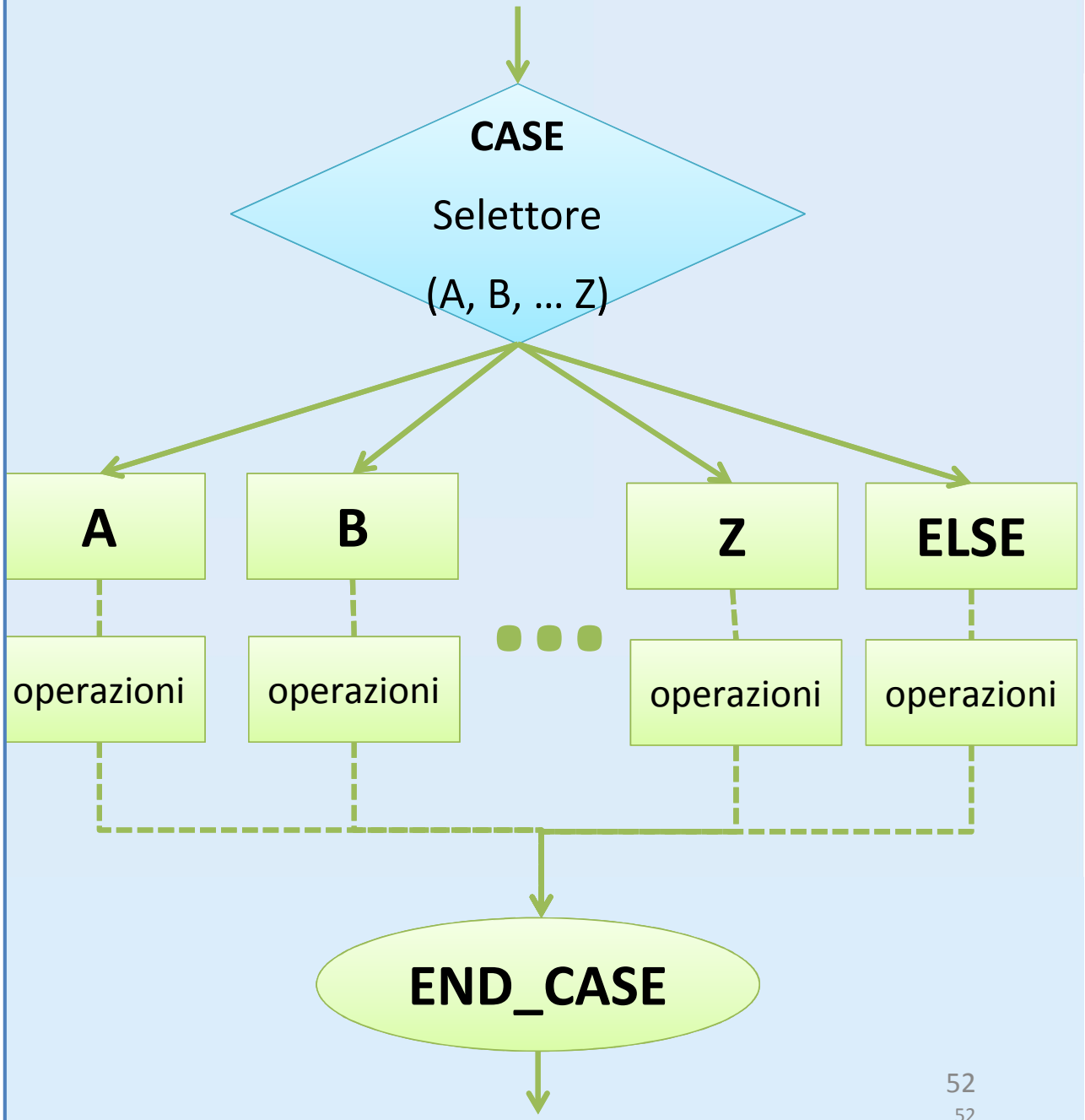


SELEZIONE MULTIPLA: CASE

- Utile quando una variabile o espressione deve dar luogo a diverse azioni per i diversi valori assunti
- E' una sorta di ''Menù'':
  - dà all'utente la ***possibilità di scegliere*** quale azione compiere

SELEZIONE MULTIPLA: CASE

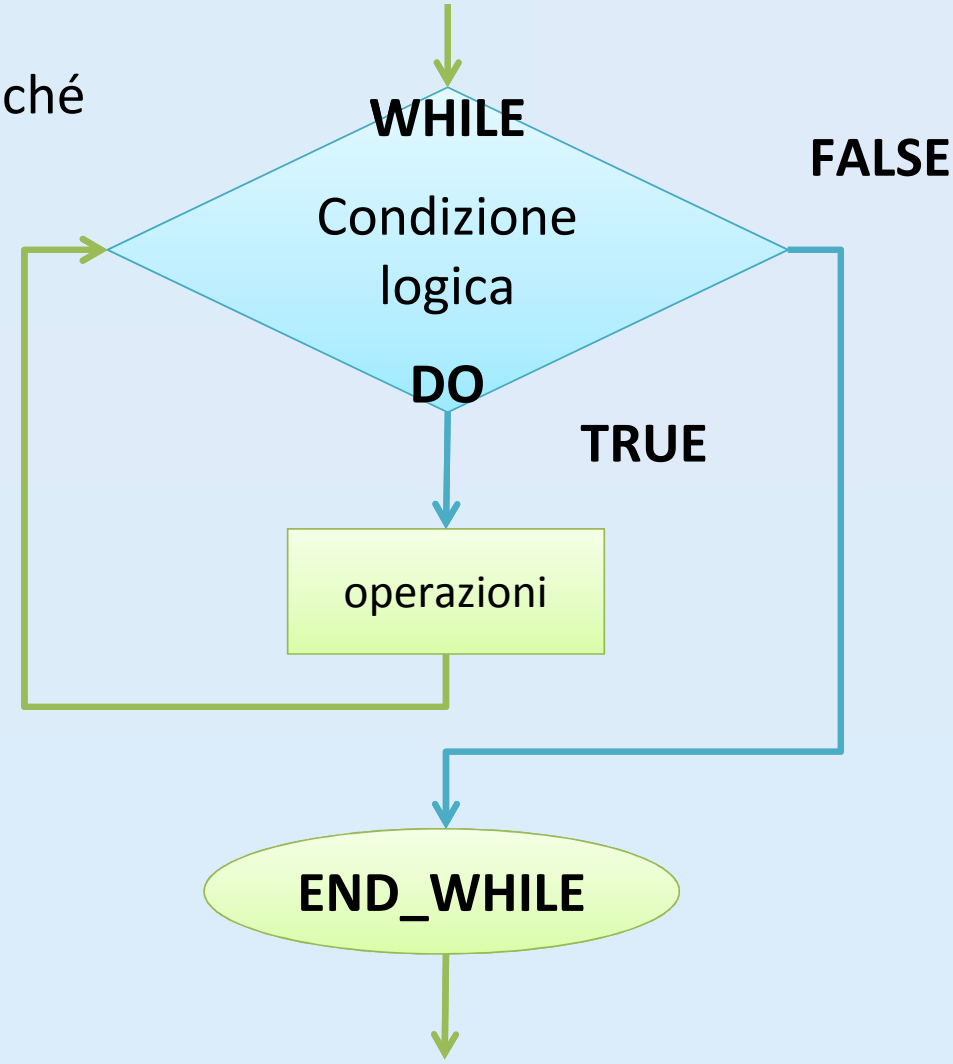
```
CASE Operazione OF
  SOMMA: C := A+B;

  SOTTRAZIONE: C := A-B;

  PRODOTTO: C := A*B;

  DIVISIONE: C := A/B;

  ELSE C := A;

END_CASE
```



RIPETIZIONE: WHILE

- Il programmatore specifica una azione che deve essere ripetuta mentre (WHILE) una certa condizione logica rimane TRUE
  - Il loop while viene ripetuto finché la condizione diventa false

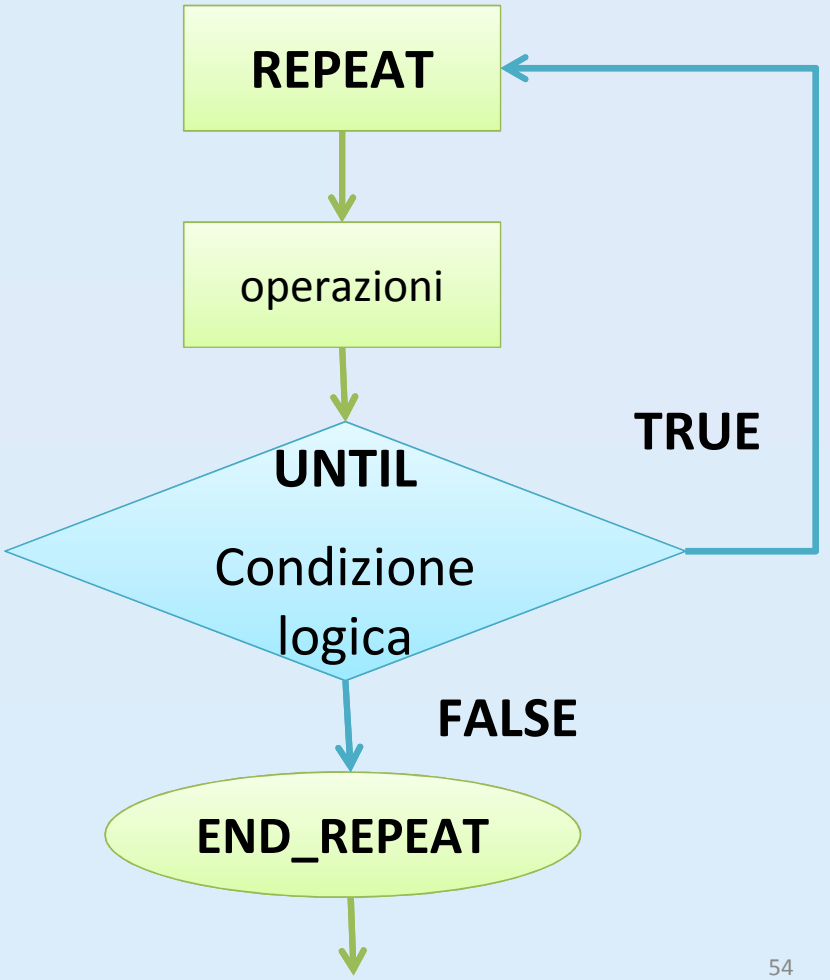
```
I := 1;
WHILE I <= 100 DO
   C := A + B;
   I := I + 1;
END_WHILE ;
```



RIPETIZIONE: UNTIL

- Il programmatore specifica una azione che deve essere ripetuta mentre una certa condizione logica rimane TRUE
  - A differenza del WHILE, la condizione è in fondo quindi il corpo del ciclo viene eseguito almeno una volta
  - Il ciclo viene ripetuto finché la condizione diventa FALSE

```
I := 1
REPEAT
   C := A + B;
   I := I + 1;
UNTIL I <= 100
END_REPEAT;
```



Utilizzo dei cicli in automazione

- Il programma di controllo di una macchina automatica in esecuzione su un PLC prevede già una ciclicità
- Le istruzioni WHILE e UNTIL sono raramente utilizzate
- Più diffuso è l'uso del **FOR** per ripetere delle operazioni su ARRAY di dati

RIPETIZIONE: FOR

- E’ equivalente al WHILE ma è più semplice da usare se si deve ripetere una operazione N volte con N predefinito
  - Non c’è la condizione ma c’è un indice che determina il numero dei cicli

```
FOR I := 1 TO 100 BY 1
DO
    C := A + B;
END_FOR ;
```

Strutturazione del programma

- Quando affrontiamo un problema complesso come lo affrontiamo?
  - Il problema può essere scomposto di tanti sottoproblemi, alcuni uguali fra loro
- Il programma è formato da un unico grande pezzo di codice o da tanti pezzi in relazione tra loro?

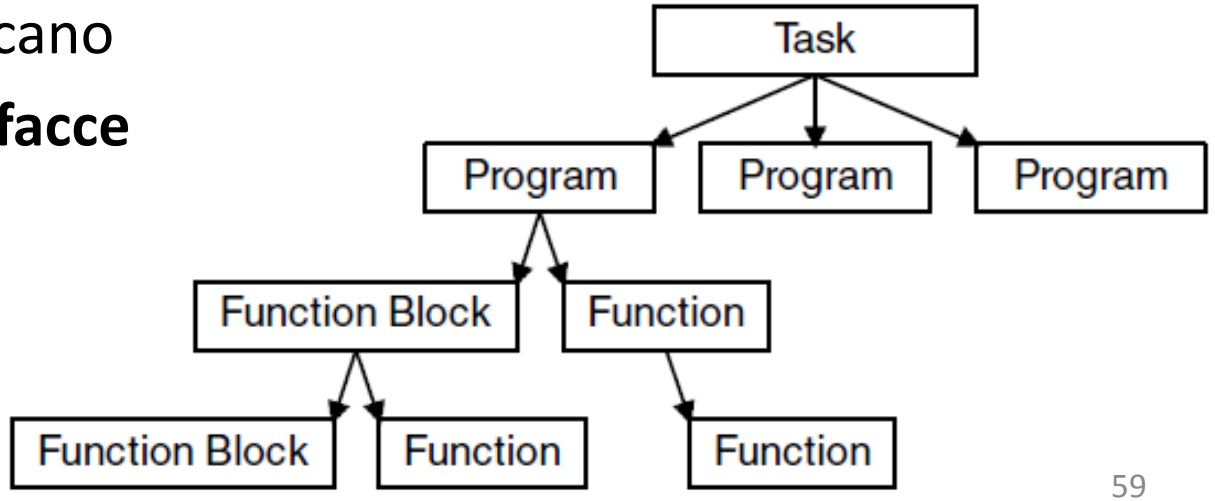
In spaghetti code, the relations between the pieces of code are so tangled that it is nearly impossible to add or change something without unpredictably breaking something somewhere else.

Modularità

- Quando un linguaggio facilita la scrittura di parti di programma indipendenti (moduli) viene definito ***modulare***.
- I moduli semplificano la ricerca e la correzione degli errori, permettendo di isolare rapidamente la parte di programma che mostra il comportamento errato e modificarla senza timore di introdurre conseguenze in altre parti del programma stesso.
- La modularità si ripercuote positivamente sulla **manutenibilità** del codice; inoltre permette di riutilizzare il codice scritto in passato per nuovi programmi, apportando poche modifiche.
- In genere la modularità si ottiene con l'uso di **sottoprogrammi** (subroutine, procedure, funzioni) e con la programmazione ad oggetti.

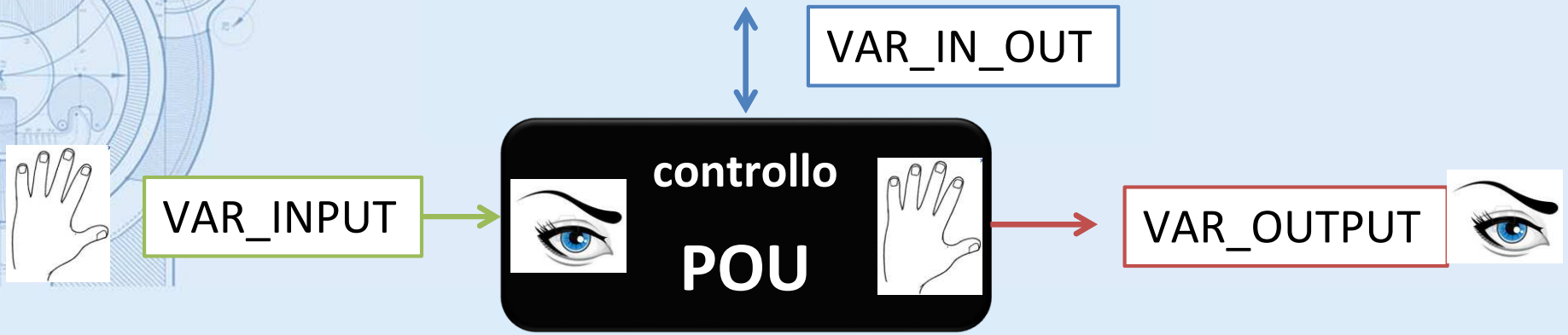
Program organization unit (POU)

- Lo standard IEC61131-3 definisce i *Program*, i *Function Block*, e le *Function*, come unità di programma (POU) che vengono definite e richiamate dall'utente nel suo programma
- L'utente può così organizzare il proprio software di controllo in una gerarchia di moduli (ovvero le POU)
  - In cima alla gerarchia c'è sempre una Task che esegue uno o più Program
  - I vari blocchi comunicano attraverso delle **interfacce**



INTERFACCIA di COMUNICAZIONE TRA POU

- Una POU è una *black box* che svolge una o più funzionalità
  - può contenere delle proprie variabili interne
  - ha un proprio programma di controllo
  - comunica (scambia dati) con il mondo esterno tramite interfaccia ben definita



- **VAR_INPUT**: le variabili di ingresso vengono modificate da chi utilizza la POU e possono solo essere lette dal programma di controllo della POU
- **VAR_OUTPUT**: le variabili di uscita possono essere modificate dalla POU. Chi utilizza la POU non può modificare queste variabili, ma solo leggerle.
- **VAR_IN_OUT**: queste variabili possono essere lette e modificate sia dalla POU che da chi la utilizza

FLUSSO di controllo

Invocazione: all'atto della chiamata, l'esecuzione del PROGRAM viene sospesa e il controllo passa alla POU invocata

```
PROGRAM PLC_PRG
…
MioFunctionBlock(I:=X);
Risultato :=
MioFunctionBlock.O;
…

K := SOMMA(A:=Y, B:=Z);
…
END_PROGRAM
```

```
FUNCTION_BLOCK
PIPPO
…
IF(I=1)
THEN
   O := 'PIPPO'
END_IF
```

```
FUNCTION SOMMA
…
SOMMA := A + B;
…
RETURN;
END_FUNCTION
```

Il controllo ritorna al Program chiamante quando la POU chiamata termina o quando si incontra un RETURN

FUNCTION (FUNZIoNI)

- Spesso può essere utile avere la possibilità di costruire “nuove istruzioni” che risolvano parti specifiche di un problema
- Una ***funzione*** è un ***componente (modulo) software*** che cattura l’idea matematica di funzione, e che possiede
  - un **nome** che identifica una espressione
  - molti possibili **parametri di ingresso** (che *non vengono modificati)*
  - *una sola **uscita*** (il risultato)
    - In ST è possibile fornire altri output addizionali oltre il risultato

FUNZIONE

- Riceve dati di ingresso in corrispondenza ai ***parametri (VAR_INPUT)***
- ha come corpo una ***espressione*** che utilizza i parametri e la cui valutazione fornisce un risultato
  - Lo standard IEC61131 permette di restituire anche altre uscite in aggiunta al risultato (**VAR_OUTPUT** e **VAR_IN_OUT**)
- viene invocata da chi la usa tramite il suo ***nome***
- **non ha memoria** delle esecuzioni precedenti (*comportamento combinatorio*)

FUNZIONE: DEFINIZIONE

```
FUNCTION nomeFunzione: TipoDiDatoInUscita
VAR_INPUT
   Variabili di ingresso
END_VAR
VAR
   Variabili interne di appoggio
END_VAR
VAR_OUTPUT
   Variabili di uscita
END_VAR
```

IN CODESYS questa è la parte delle variabili ovvero il riquadro superiore

```
   Operazioni


RETURN;


END_FUNCTION
```

IN CODESYS questa è la parte del controllo ovvero il riquadro inferiore

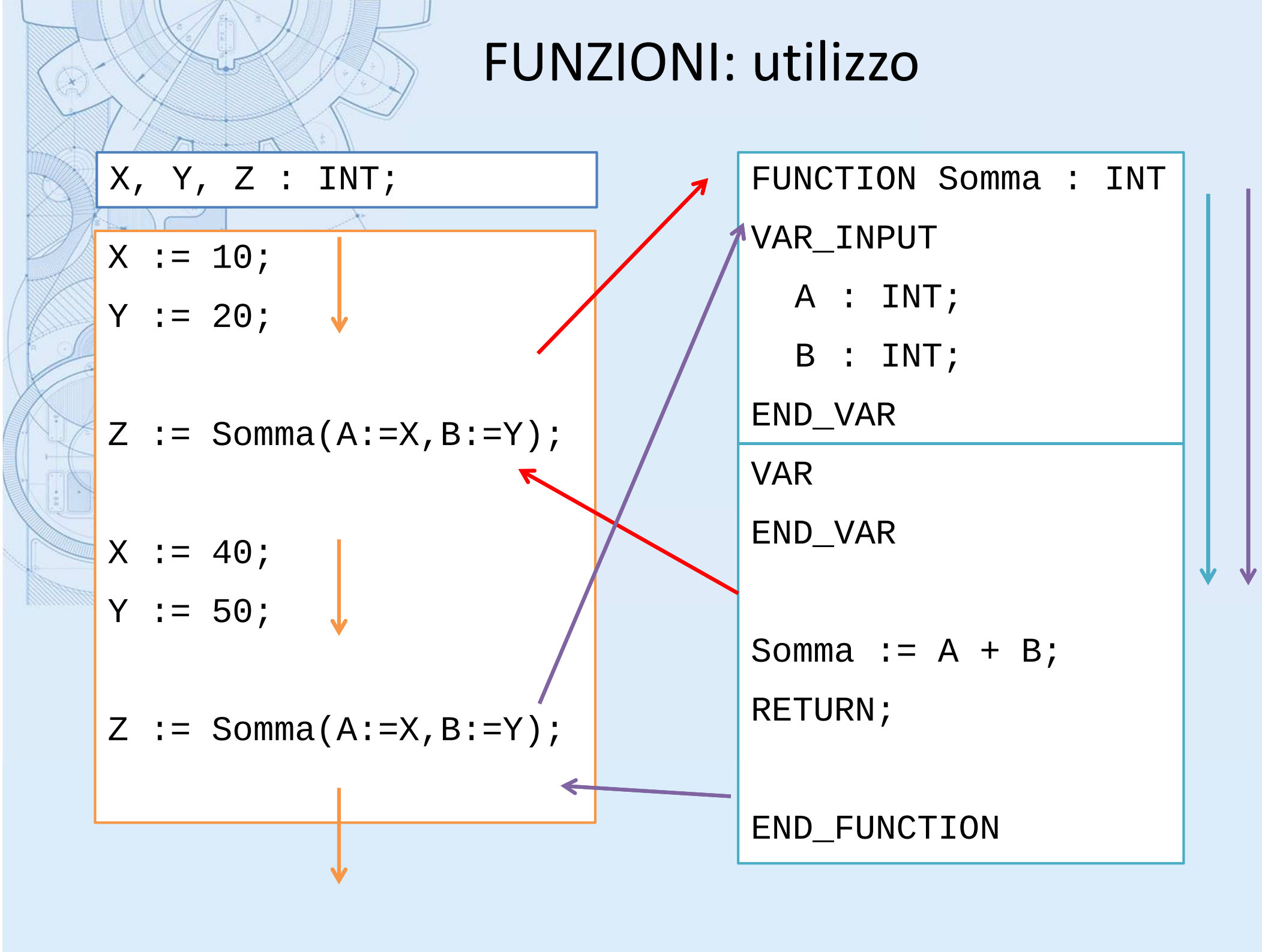
FUNZIONI: utilizzo

```
X, Y, Z : INT;
```

```
X := 10;
Y := 20;

Z := Somma(A:=X,B:=Y);

X := 40;
Y := 50;

Z := Somma(A:=X,B:=Y);
```

```
FUNCTION Somma : INT
VAR_INPUT
   A : INT;
   B : INT;
END_VAR
```

```
VAR
END_VAR


Somma := A + B;
RETURN;


END_FUNCTION
```



FUNZIONI STANDARD

- Il linguaggio strutturato (ST) mette a disposizione delle funzioni standard (elementari)
  - ABS: calcola il valore assoluto di un numero, X := ABS(Y);
  - SQRT: calcola la radice quadrata di un numero, X := SQRT(Y);
  - LN: calcola il logaritmo in base naturale di un numero, X := LN(Y);
  - LOG: calcola il logaritmo in base 10 di un numero, X := LOG(Y);
  - EXP: calcola l'esponenziale di un numero, X := EXP(Y);
  - SIN: calcola il seno di un numero, X := SIN(Y);
  - COS: calcola il coseno di un numero, X := COS(Y);
  - ADD: calcola la somma di numeri, A := ADD(B,C,D) ;
  - MUL : calcola il prodotto di numeri, A := MUL(B,C,D) ;
  - SUB : calcola la sottrazione di numeri, A := SUB(B,C) ;
  - DIV : calcola la somma di numeri, A := DIV(B,C) ;
  - MOD: calcola il modulo, A := MOD(B,C) ;
  - Altre…

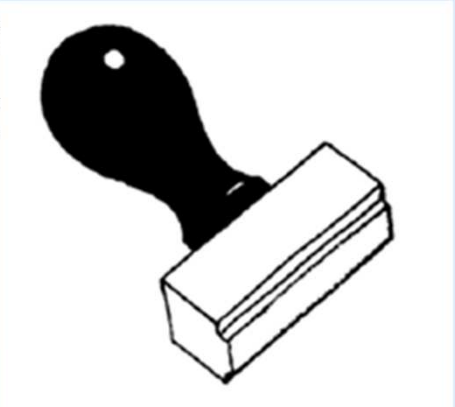
FUNCTION BLOCK (Blocco funzione)

- Un ***Function Block*** è un ***componente (modulo) software*** che cattura l'idea di dispositivo fisico che svolge una specifica funzionalità, e che possiede
  - un **nome** che descrive il tipo di componente,
  - un **identificatore** che identifica la specifica istanza del componente
  - uno o più **parametri di ingresso**
  - una o più **variabili di uscita**
  - una o più **variabili di stato**, ovvero possiede memoria riguardo le esecuzioni precedenti

Classi di dato e oggetti nei linguaggi moderni

- Il Function Block è vicino all'idea informatica di classe di oggetti, disponibile in linguaggi come C++ e Java

- Un **OGGETTO** è un componente software creato sulla base di un "modello" (la **CLAS** che può essere paragonato a un timbro, c definisce le caratteristiche degli oggetti creati a sua immagine.



- Gli oggetti creati a immagine e somiglianza di un certo "timbro" condividono:
  - la stessa struttura interna dei dati
  - lo stesso funzionamento (logica)
- ma ciascuno ha la propria identità
  - I valori delle variabili possono cambiare in maniera autonoma!

FUNCTION BLOCK

- Quindi… un Function Block:
  - Deve essere creato con **un nome che è un identificatore univoco** (uno solo in tutto il programma)
  - Viene eseguito dal programma, che lo richiama attraverso il suo identificatore
    - Riceve dall'esterno dei segnali in ingresso (**VAR_INPUT**)
    - Esegue il programma di controllo definito nel suo ''corpo''
    - L'elaborazione del programma di controllo dipende dalla storia precedente di quel Function Block con quell'identificatore (storia rappresentata dallo stato interno)
    - Restituisce uno o più segnali di uscita all'esterno (**VAR_OUTPUT**)
- L'istanza di un Function Block **ha memoria** delle sue esecuzioni precedenti (*comportamento sequenziale*), ovvero della storia passata.

FUNCTION_BLOCK: DEFINIZIONE

```
FUNCTION_BLOCK NomeFunctionBlock

VAR_INPUT
   Variabili di ingresso
END_VAR
VAR
   Variabili interne di stato
   (sono la memoria dell'oggetto)
END_VAR
VAR_OUTPUT
   Variabili di uscita
END_VAR
```

IN CODESYS questa è la parte delle variabili ovvero il riquadro superiore

```
   logica di controllo

END_FUNCTION_BLOCK
```

IN CODESYS questa è la parte del controllo ovvero il riquadro inferiore

FUNCTION_BLOCK: utilizzo

```
segnale      : BOOL;
MioflipFlop  : FF_T;
risultato    : BOOL;
```

```
...


MioflipFlop.T := segnale;
MioflipFlop;
Risultato :=
MioflipFlop.Q;


...
```

```
FUNCTION_BLOCK FF_T
VAR_INPUT
 T    : BOOL;
END_VAR
VAR_OUTPUT
 Q : BOOL;
END_VAR
VAR
 stato  : BOOL;
END_VAR
```

```
Q  := NOT stato;
stato   := T;


END_FUNCTION_BLOCK
```

FUNCTION BLOCK STANDARD

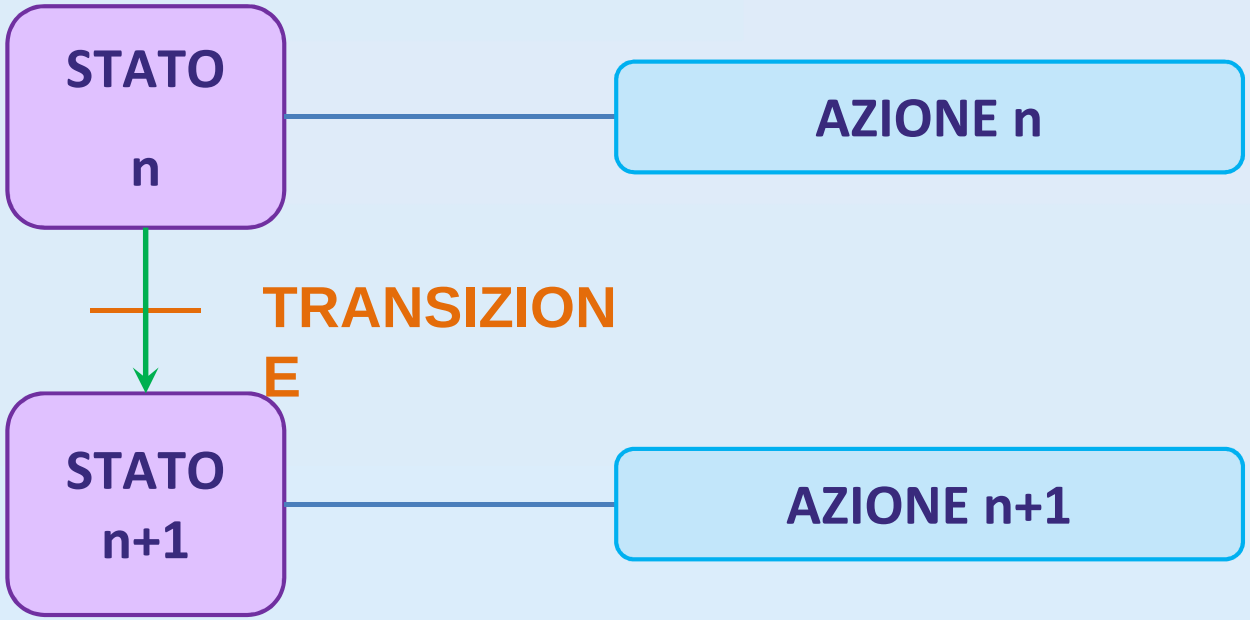
- Lo standard IEC 61131 mette a disposizione dei function block standard
- Nel manuale di Codesys che si apre premendo F1, si trovano i FunctionBlock messi a disposizione dalla libreria standard, nel percorso:
  - Contenuto → CoDeSys Libraries → Standard.lib library
    - Divisi in
      - Flip Flop (bistabili)
      - Trigger
      - Counter
      - Timer

Sequential Function Chart (SFC)

- E’ un potente strumento grafico per descrivere il comportamento logico/sequenziale di un sistema di automazione industriale.
- Non può essere considerato un linguaggio di programmazione a tutti gli effetti: necessita dell’utilizzo di altri linguaggi di programmazione specifici per l’implementazione vera e propria.

Elementi base

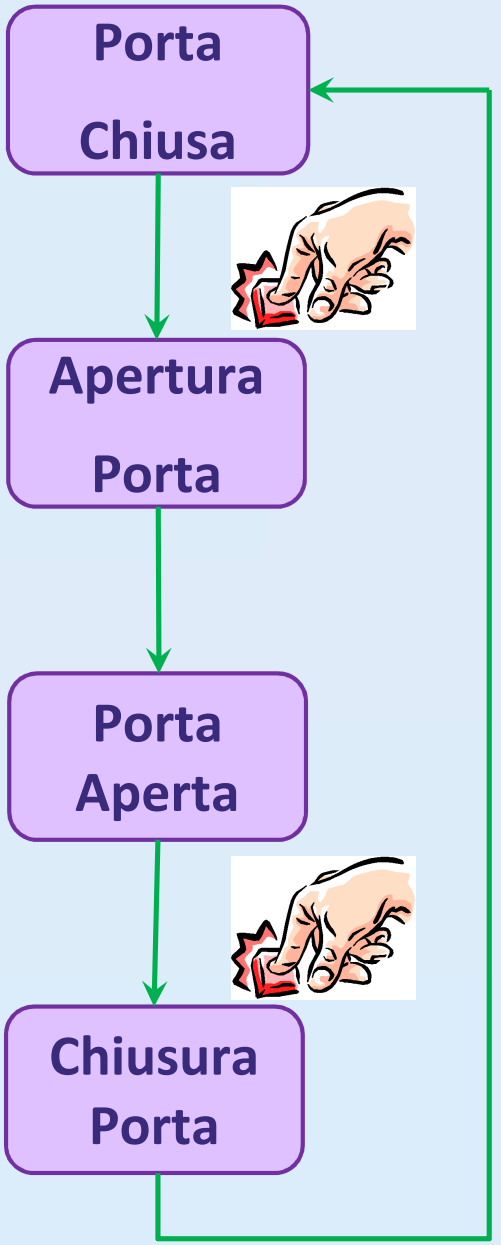
1. **STATO**
2. **AZIONE**
3. **TRANSIZIONE**
4. **ARCHI DI COLLEGAMENTO**



Elementi Di Base: Stato

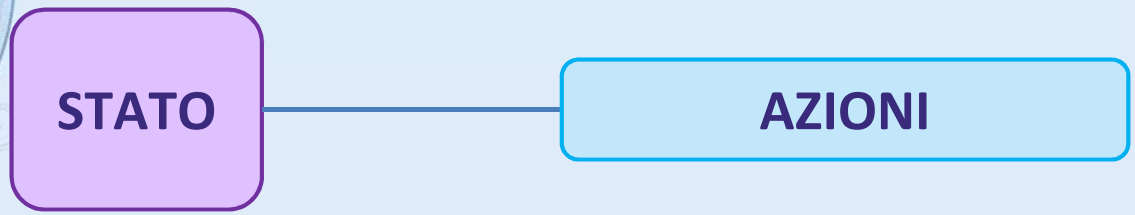
- Lo **STATO** (Fase, Step, Passo) è una situazione del sistema che può modificarsi in base al verificarsi di eventi (interni o esterni). In altre parole è una **condizione operativa** della macchina che ne riflette il **comportamento** in un determinato istante.
- L'evoluzione temporale del funzionamento di un impianto è descrivibile mediante una successione temporale di situazioni operative più semplici (stati).
- Uno stato, in un determinato istante, può essere:
  - Attivo
  - Disattivo

Concetto di stato: la porta



ELEMENTI DI BASE: AZIONI

- A ogni stato sono associate nessuna, una o più **AZIONI**, ovvero istruzioni o algoritmi di controllo che vengono eseguite dal controllore quando lo **stato** è **attivo**.



Elementi di base: transizioni

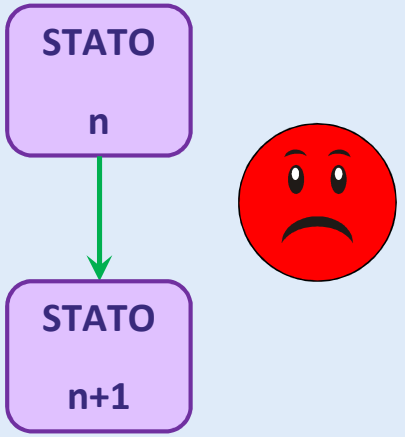
- Le **TRANSIZIONI** sono le possibilità di **evolvere** da uno **stato** ad un altro.
- Ad ogni transizione è associata una condizione che deve essere verificata affinché l'evoluzione da uno stato ad un altro possa avvenire.
- Descrivono gli eventi che possono modificare la condizione operativa del sistema.
- In genere sono funzioni booleane.

Elementi di base: archi orientati

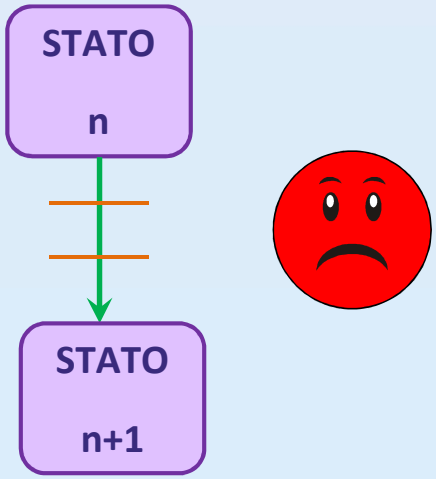
- Gli **ARCHI ORIENTATI di collegamento** uniscono due stati indicandone la sequenza evolutiva: l'evoluzione da uno stato ad un altro è univocamente determinato dall'arco orientato che li unisce e dalla **transizione** (con la relativa condizione).

Regole di base 1/2

1. Due **stati** **non** possono essere connessi **direttamente**: deve esistere una ed una sola transizione tra di loro.

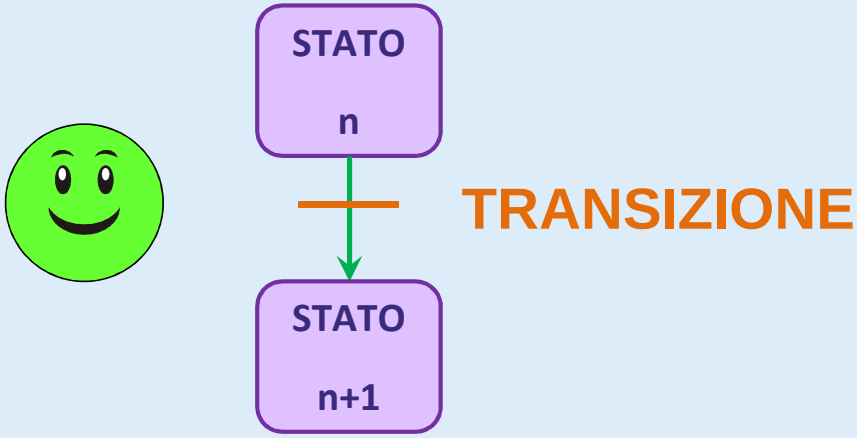


2. Due **transizioni** **non** possono essere connesse **direttamente**: deve esistere uno stato tra di loro.



Regole di base 2/2

3. La sequenza **stato** – **transizione** – **stato** deve essere rispettata.



4. Lo **stato** **iniziale** è rappresentato con un doppio rettangolo/quadrato ed è unico.

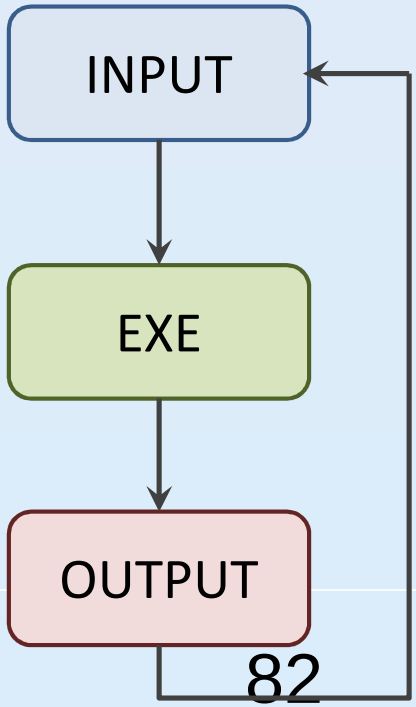
5. Gli **identificatori** (nomi) degli **stati** devono essere **univoci**.

Regole di evoluzione 1/2

- Una transizione è abilitata se tutti gli stati a monte sono attivi: una transizione non abilitata non viene testata.
- Una transizione diviene attiva (o superabile) se è abilitata e la condizione ad essa associata risulta essere vera.
- La transizione attiva determina il cambio di stato (le azioni associate al nuovo stato attivo vengono eseguite).

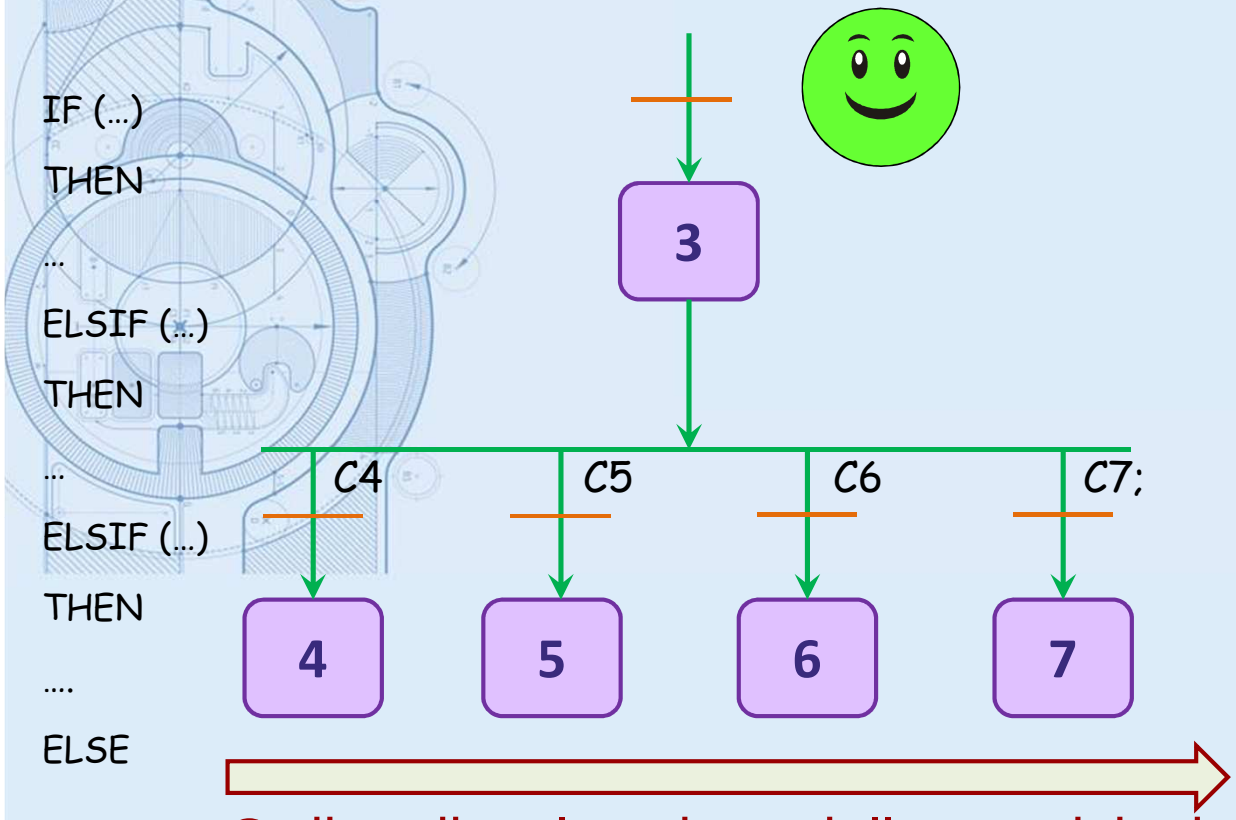
Regole di evoluzione 2/2

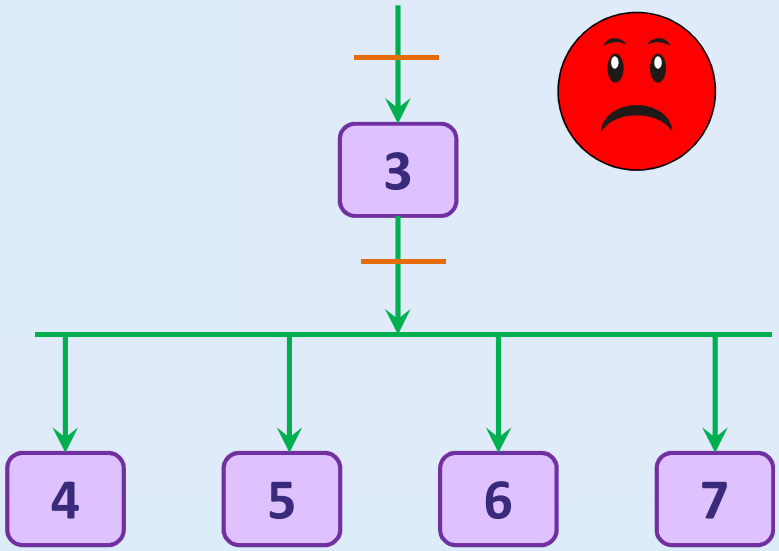
- L'SFC viene eseguito da un PLC.
- L'**esecuzione** del programma avviene in modalità **CICLICA**:
  - Lettura dei sensori (INPUT):
    - Si testano le transizioni abilitate per determinare quelle superabili;
    - Si definiscono gli stati attivi attuali.
  - Esecuzione degli algoritmi di controllo (EXE):
    - Si eseguono le azioni associate agli stati attivi.
  - Invio dei comandi agli attuatori (OUTPUT).
- Occorre definite lo stato attivo all'avvio.



Strutture di collegamento: scelta

Una **scelta** permette di scegliere e attivare stati (e quindi sequenze di azioni) tra loro **mutuamente esclusivi** (ovvero solo uno di essi può essere attivo dopo la scelta) a partire da un unico stato.

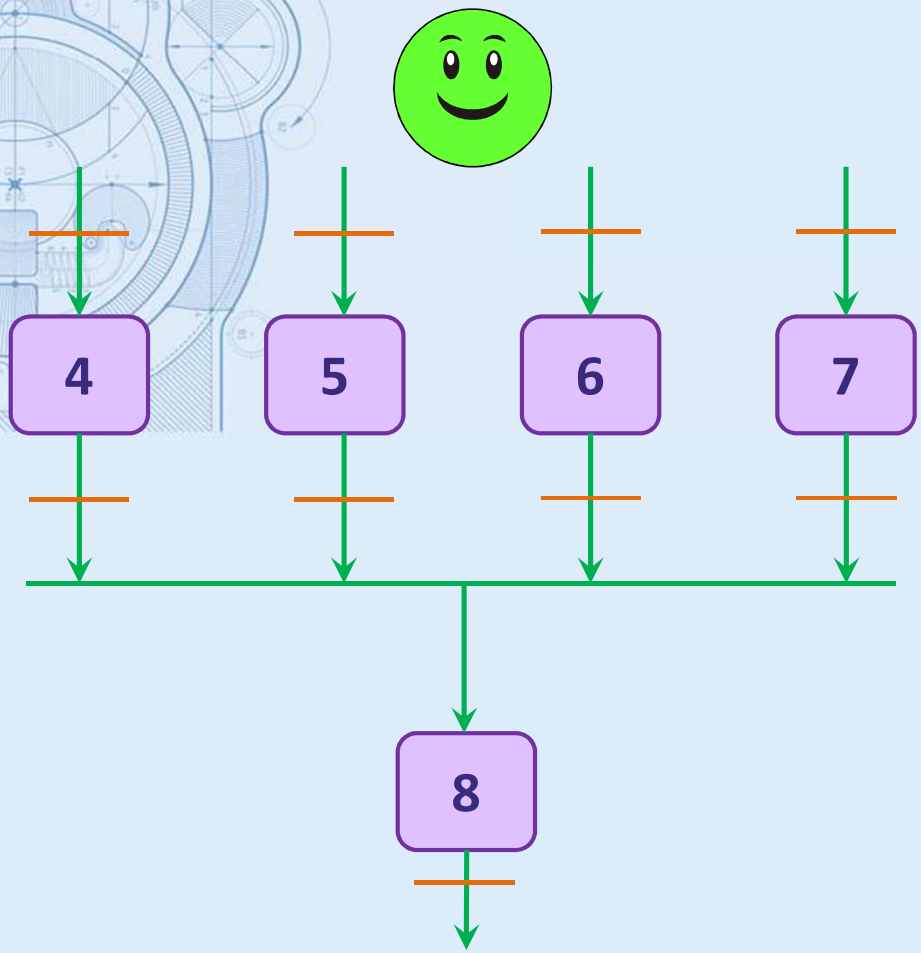


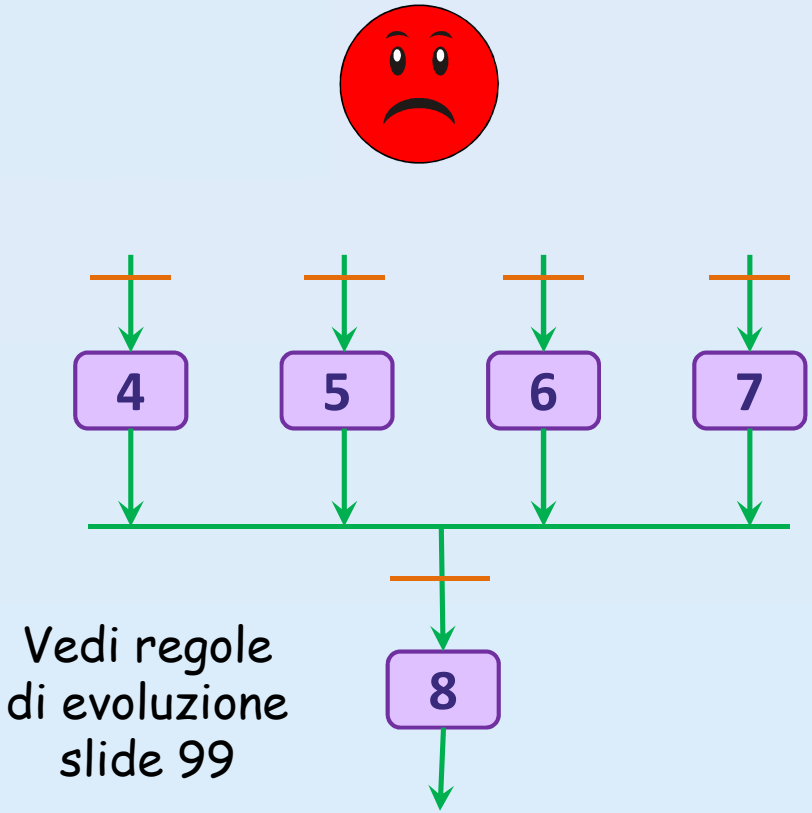



```
C4 := cond4;
C5 := cond5 AND NOT cond4;
C6 := cond6 AND NOT cond5 AND NOT cond4;
C7 := cond7 AND NOT cond6 AND NOT cond5 AND NOT cond4;
```

Strutture di collegamento: convergenza

Quando più sequenze (solitamente mutuamente esclusive tra loro) terminano in un medesimo stato attraverso diverse transizioni, deve essere utilizzata una struttura chiamata **convergenza**. Questa struttura è la logica terminazione di una scelta.





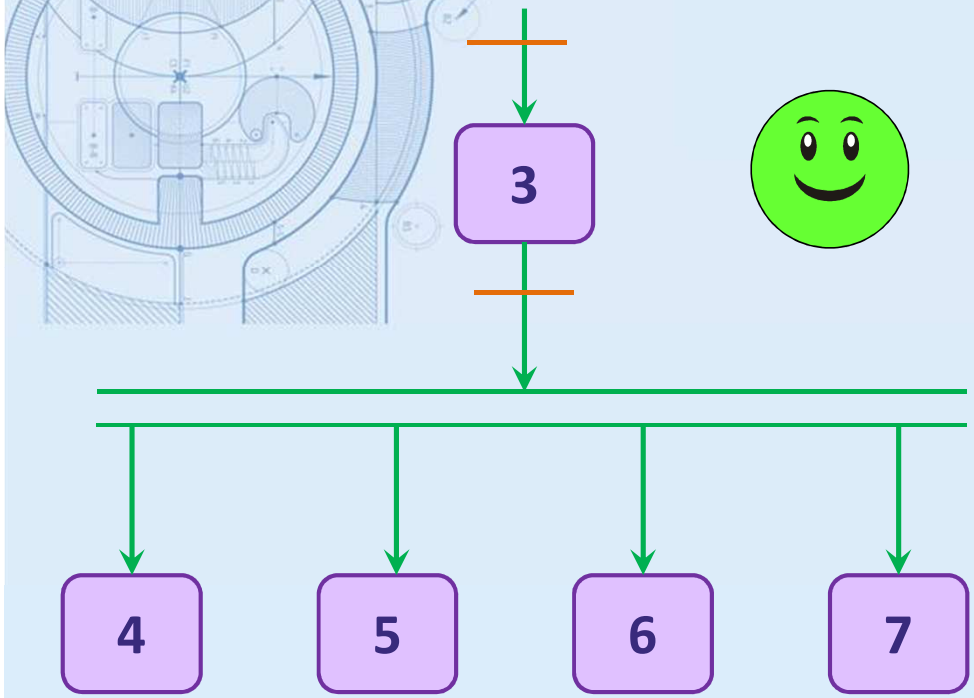
Vedi regole di evoluzione slide 99

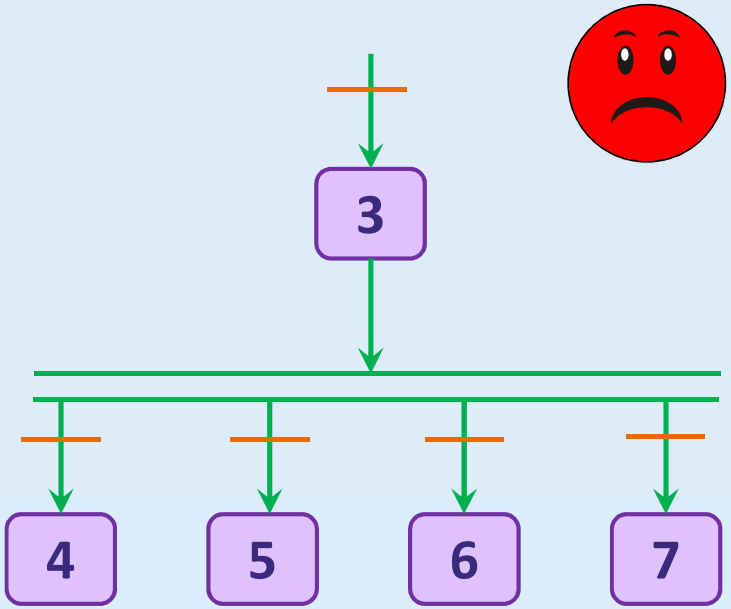
Strutture di collegamento: parallelismo

Il **parallelismo** è una struttura di collegamento che permette la definizione di sequenze di azioni parallele che evolvono in maniera indipendente.

La doppia linea rende immediatamente identificabile questa struttura rispetto alla scelta.

Possono esistere più stati attivi contemporaneamente.

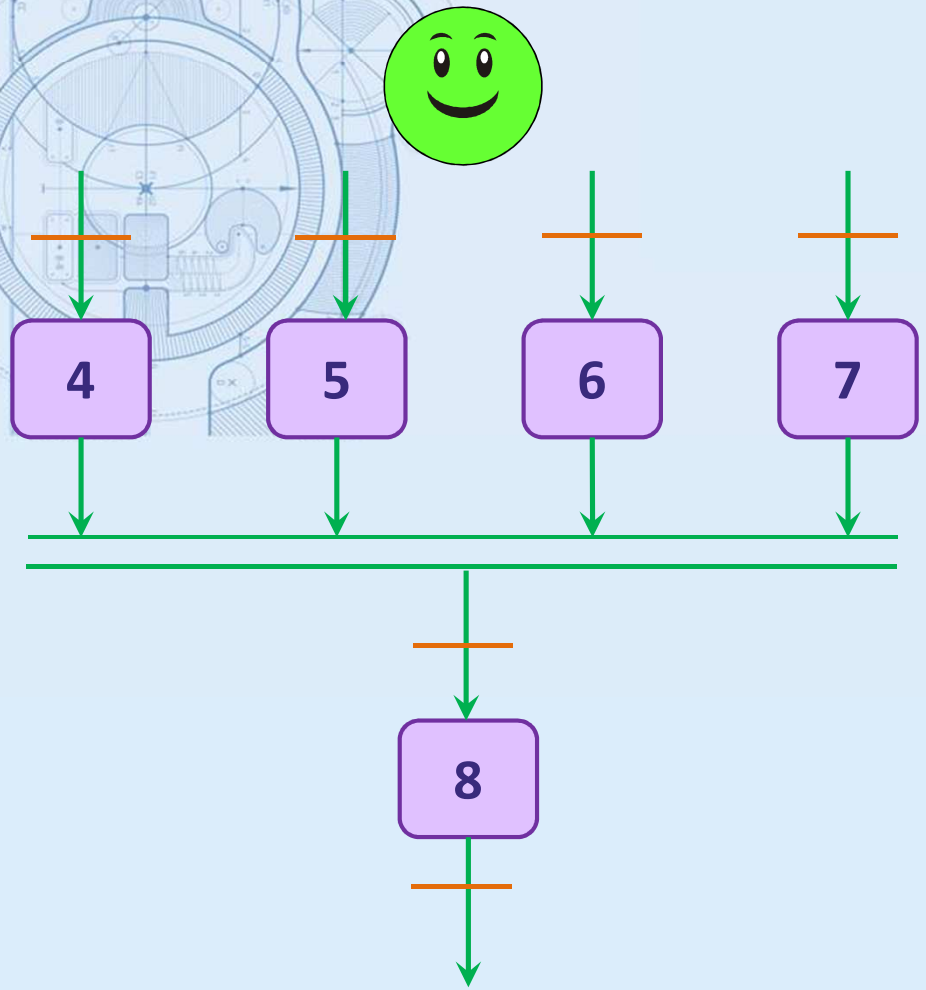


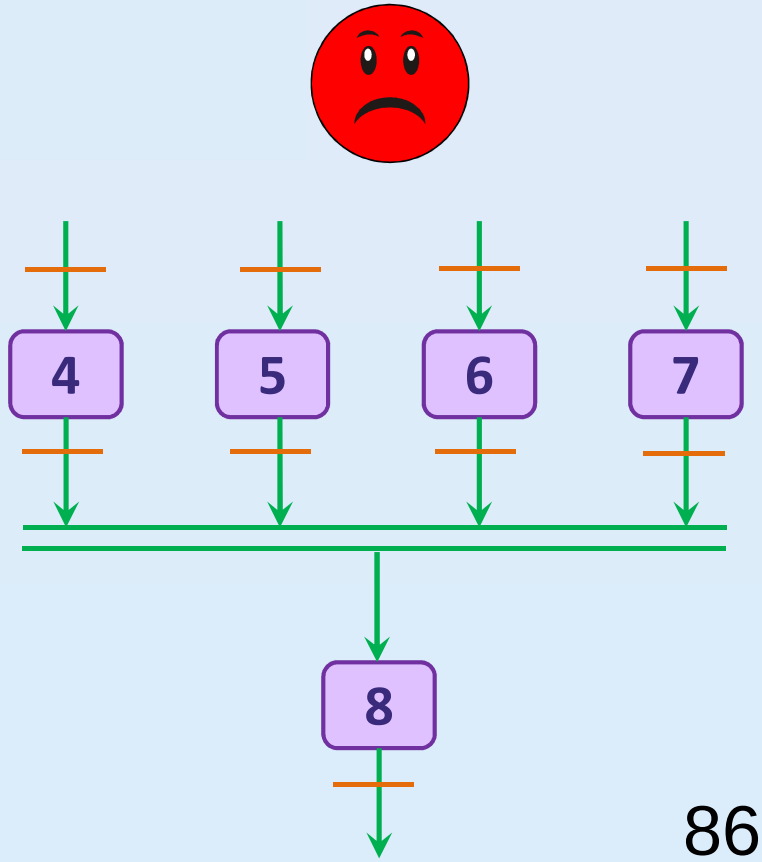


Strutture di collegamento: sincronizzazione

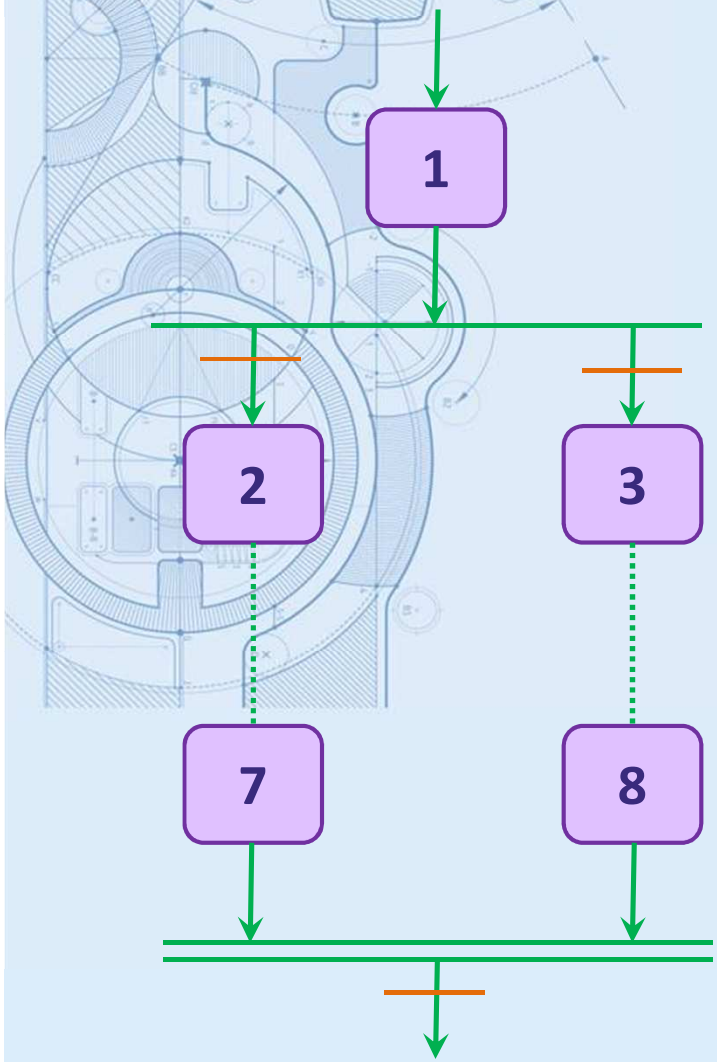
La **sincronizzazione** è la struttura duale al parallelismo: permette di sincronizzare diverse sequenze parallele indipendenti.

N. B. Affinché la transizione sia abilitata, è necessario che tutti gli stati precedenti siano abilitati!

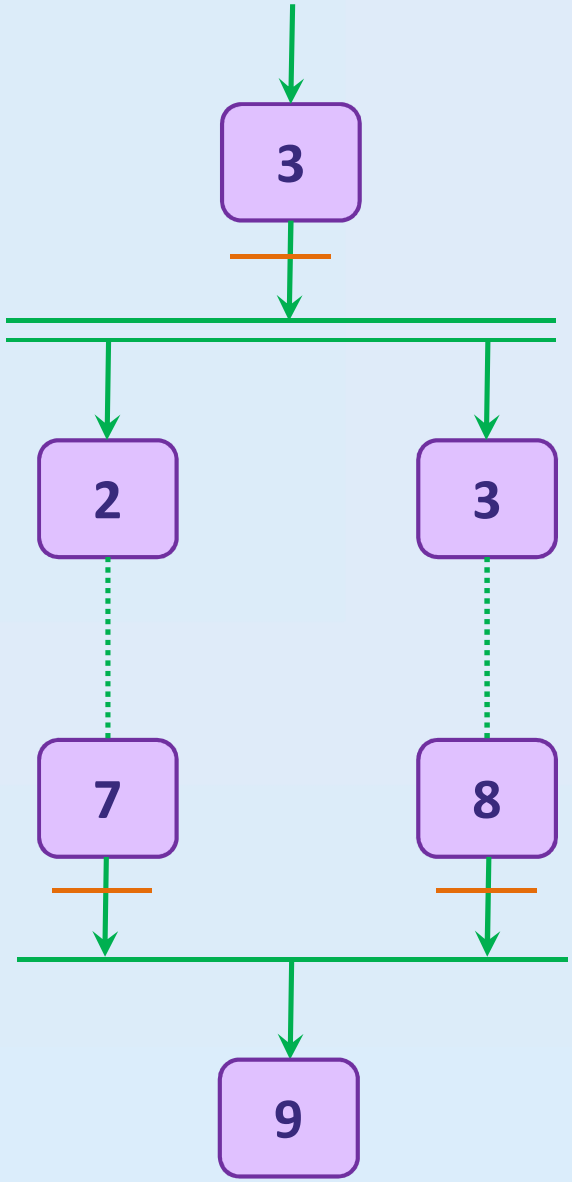




Strutture di collegamento: altri errori



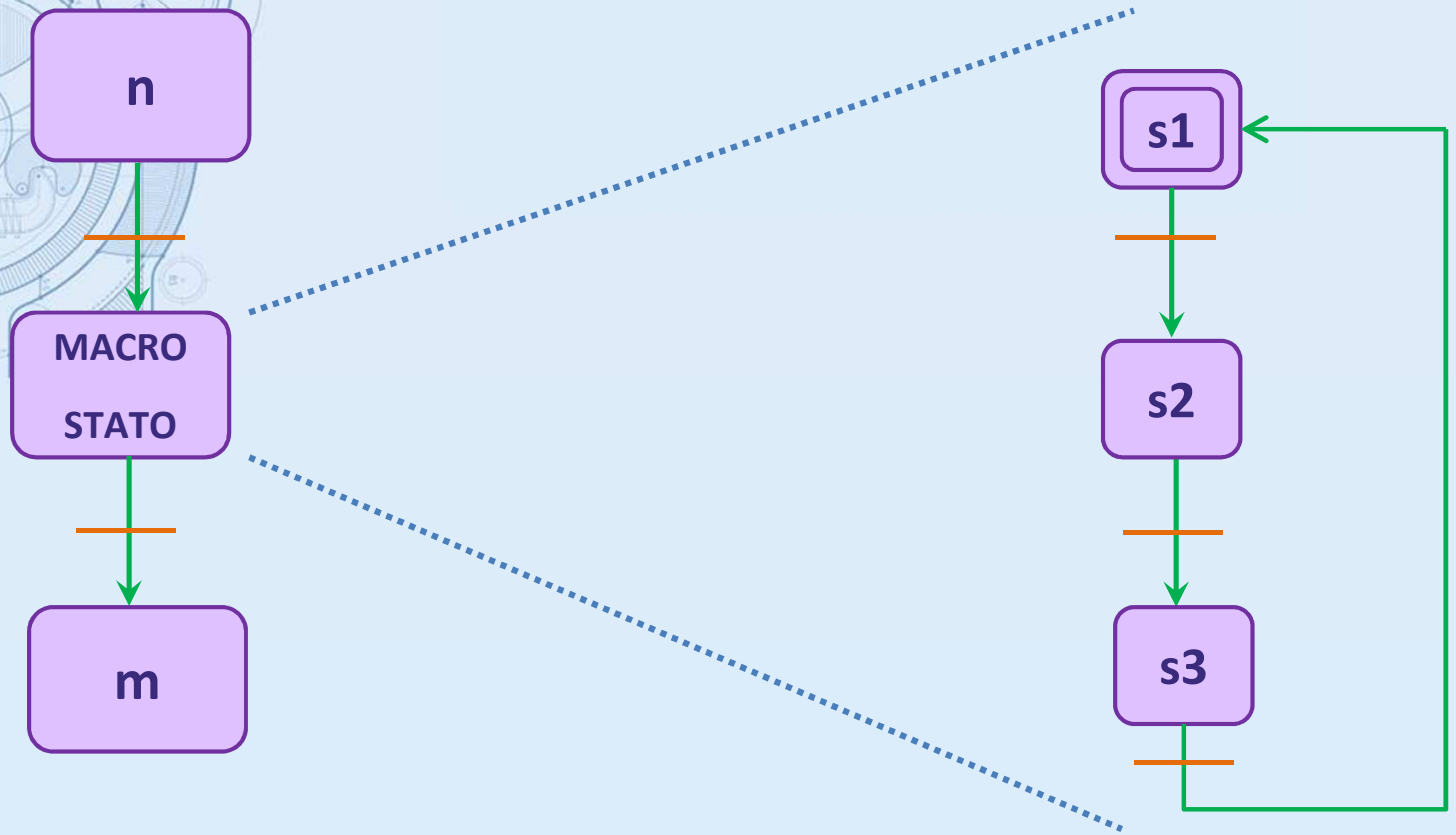
Scelta con sincronizzazione



Ambiguità: parallelismo con convergenza

Macrostati

- I **macrostati** permettono di implementare una rappresentazione grafica più sintetica.
- Ogni **macrostato** è esploso nei suoi componenti a tempo di esecuzione.
- Viene utilizzato per racchiudere una sequenza in uno stato.



Sintassi

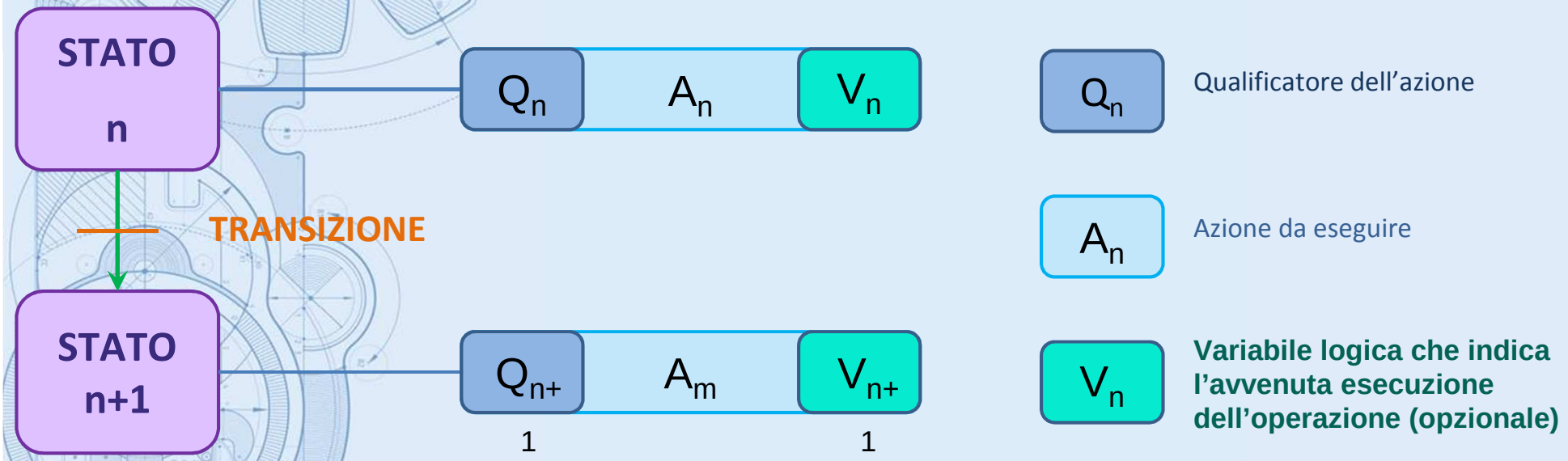
- Variabili associate ad uno stato:
  - Nome dello stato univoco: Name_State
  - Step flag (Marker): Name_State.X
    - indica che lo stato Name_State è attivo
      - variabile logica che assume il valore “vero” (TRUE, 1) per tutto il tempo di permanenza nello stato.
  - Step time: Name_State.t
    - indica il tempo trascorso dall’attivazione dello stato Name_State
      - variabile timer
        - » inizializzata a zero all'entrata nello stato, contiene l'indicazione del tempo trascorso dall'entrata nello stato.
        - » all'uscita dallo stato rimane costante (al valore pari al tempo di permanenza nello stato).

Sintassi

- **ENTRY ACTION**:
  - Definibile per ogni stato (tranne per lo stato iniziale);
  - Sequenza di istruzioni eseguite una sola volta quando lo stato diventa attivo;
  - Implementabile con uno dei linguaggi di programmazione previsti dallo standard IEC 61131;

- **EXIT ACTION**:
  - Definibile per ogni stato (tranne che per lo stato iniziale);
  - Sequenza di istruzioni eseguite una sola volta prima che lo stato da attivo diventi inattivo;
  - Implementabile con uno dei linguaggi di programmazione previsti dallo standard IEC 61131;

Qualificatori

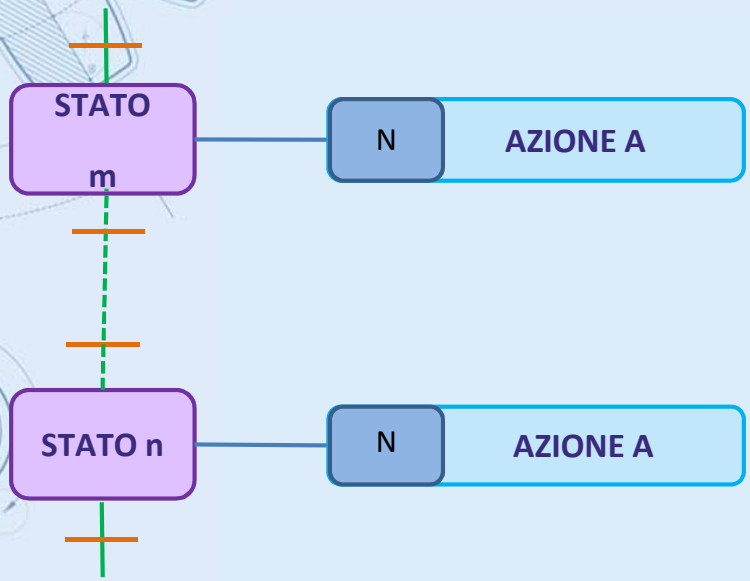


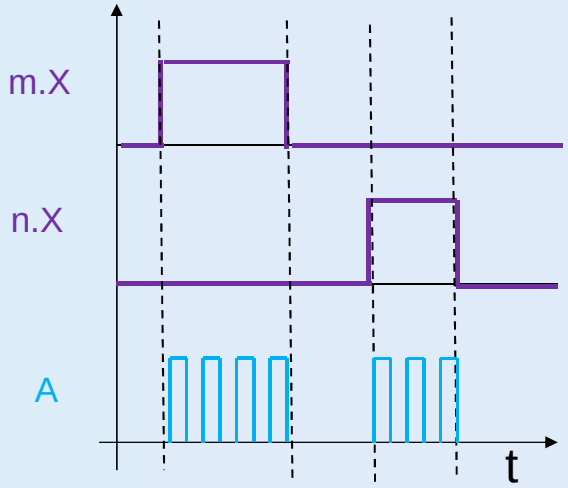
- Ad ogni **stato** possono essere associati uno, nessuno o più blocchi di **azione**;
- Il **Qualificatore** determina le regole di esecuzione dell' **azione**;
- Si può scegliere un qualsiasi linguaggio definito dallo standard IEC 61131.

Qualificatori: N e P

Azione N

Non Stored

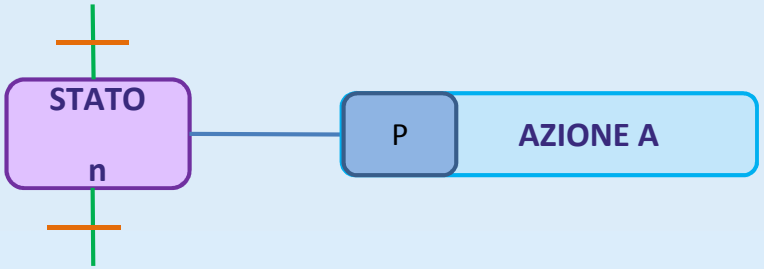


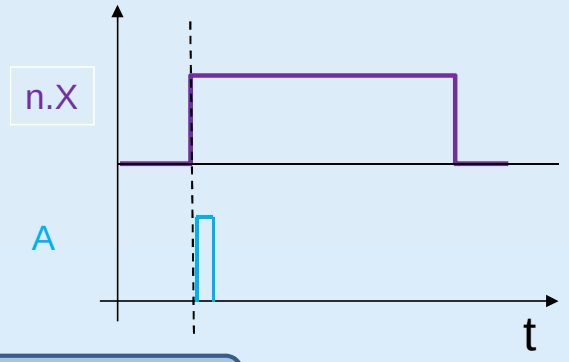


L'azione è eseguita in tutti gli stati attivi a cui è associata

Azione P

Pulse

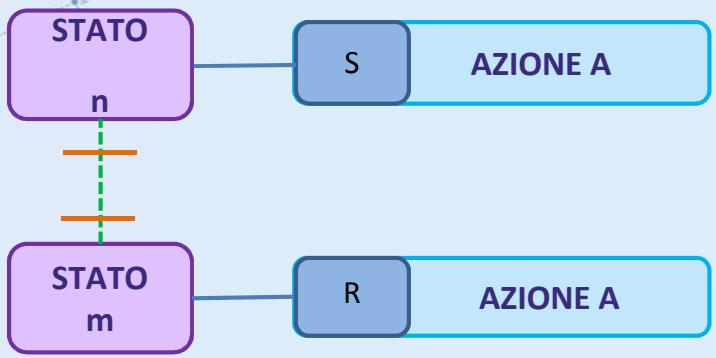


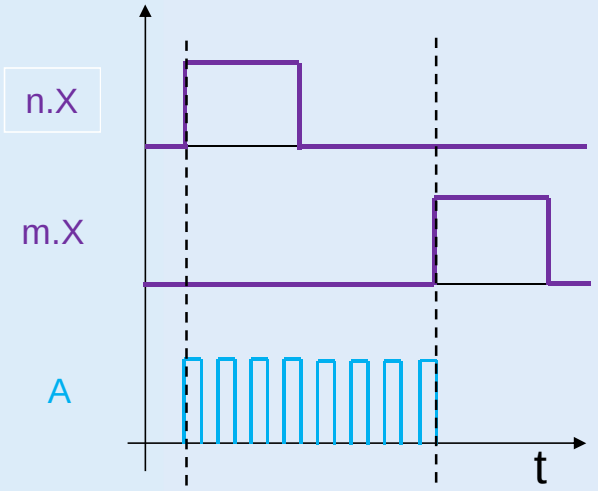


L'azione è eseguita solo una volta quando lo stato è attivato

Qualificatori: S/R e L

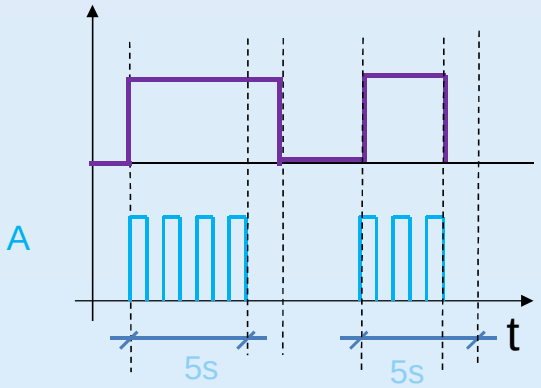
Azione S/R
Set/Reset





L'azione viene attuata sino a quando non viene eseguita la stessa azione con il qualificatore R

Azione L
Time Limited

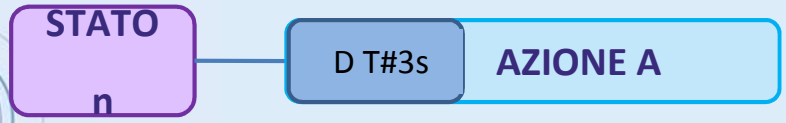


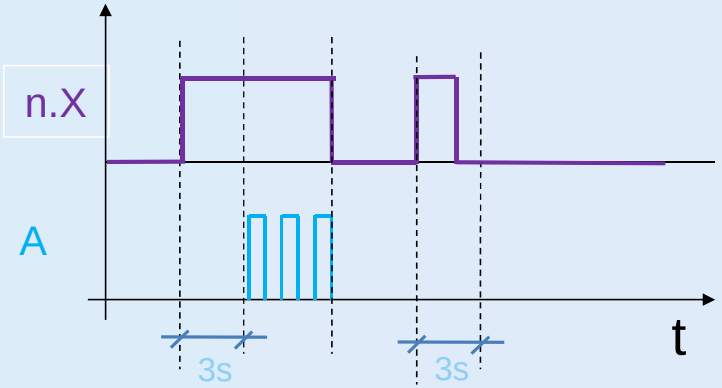
L' azione inizia all'attivazione dello stato e termina dopo l'intervallo di tempo specificato o quando si esce dallo stato a cui essa è associata.

Qualificatori: D

Azione D

Time Delayed

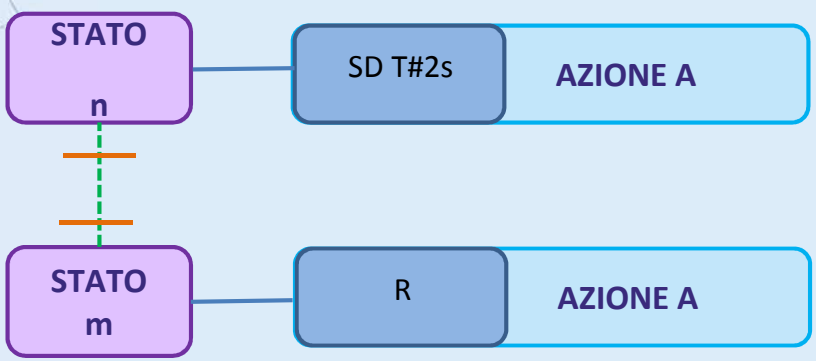


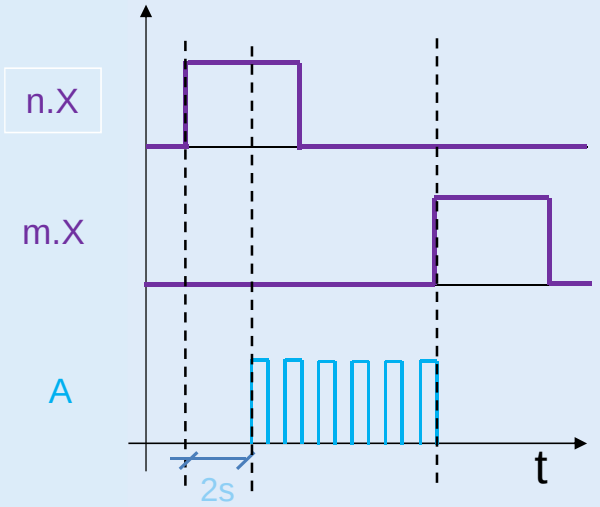


L’azione viene eseguita dopo un intervallo di tempo a partire dall’attivazione dello stato a cui è associata e termina all’uscita dallo stato stesso. Nel caso in cui si esca dallo stato prima dell’intervallo specificato, l’azione non viene intrapresa.

Qualificatori: SD

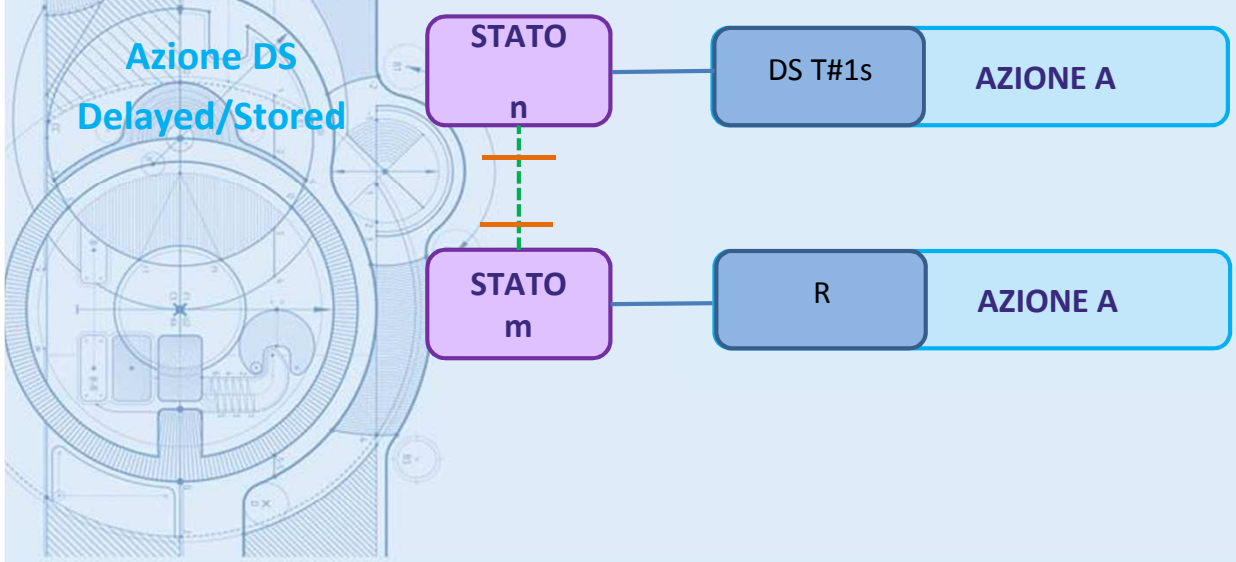
Azione SD
Stored/Delayed

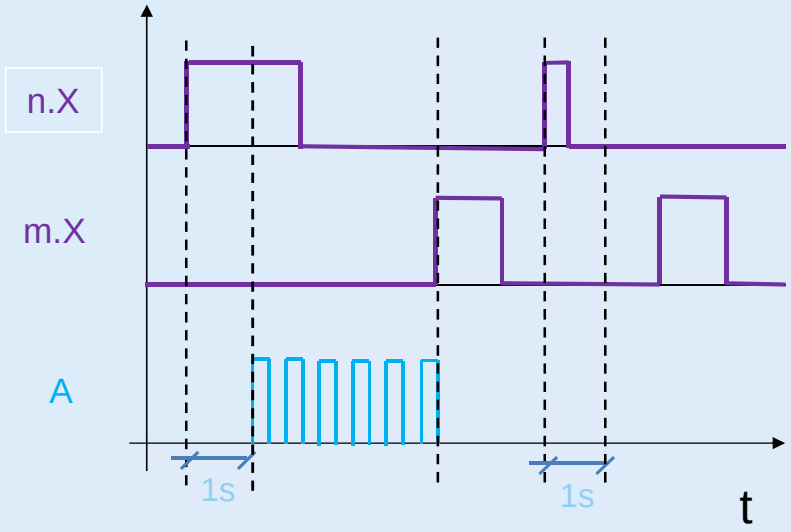




L'azione viene eseguita dopo un intervallo di tempo pari al tempo specificato dall'attivazione dello stato a cui essa è associata, fino a quando non viene eseguita la stessa azione con il qualificatore R.

Qualificatori: DS

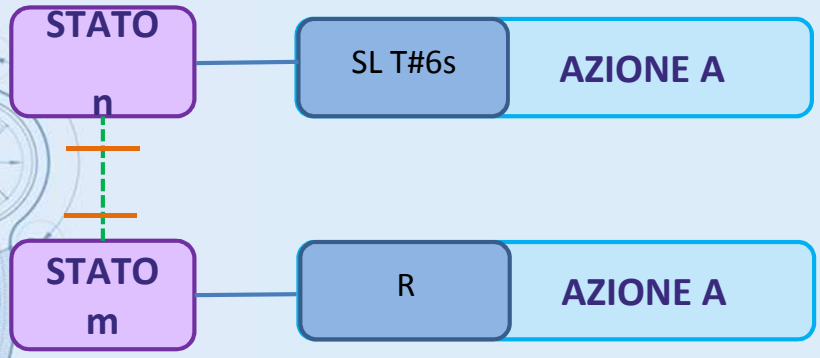


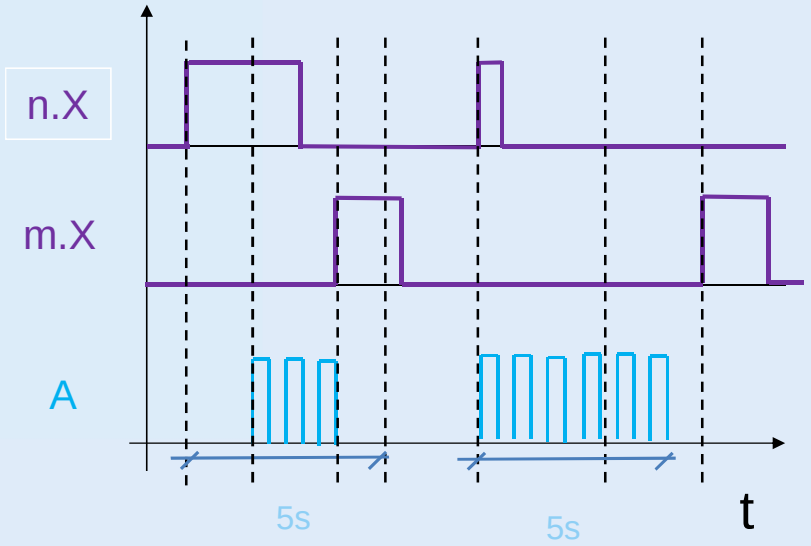


L’azione viene eseguita come azione set se lo stato a cui è associata rimane attivo per un intervallo di tempo maggiore di quello specifico per l’attivazione dell’azione.

Qualificatori: SL

Azione SL
Stored/
Time Limited





L’azione viene eseguita come azione set e viene terminata dopo un intervallo di tempo definito oppure tramite un’azione reset.