

# Linguaggi di Programmazione

<b>1</b>	<b>Introduzione</b>
<b>2</b>	<b>Specifica</b>
<b>3</b>	<b>Variabili</b>
<b>4</b>	<b>Espressioni</b>
<b>5</b>	<b>Sottoprogrammi</b>
<b>6</b>	<b>Programmazione funzionale</b>
<b>7</b>	<b>Programmazione logica</b>

- Materiale didattico in rete: <http://www.ing.unibs.it/lamperti>
- R.W. Sebesta “*Concepts of Programming Languages*”, 8<sup>th</sup> edition, Addison-Wesley, 2009.
- R.K. Dibvig “*The Scheme Programming Language*”, 3<sup>rd</sup> edition, The MIT Press, 2003.
- S. Thompson “*Haskell – The Craft of Functional Programming*”, 3<sup>rd</sup> edition, Pearson, 2011.
- G. Hutton “*Programming in Haskell*”, Cambridge University Press, 2007.
- W.F. Clocksin, C.S. Mellish “*Programming in Prolog*”, 5<sup>th</sup> edition, Springer, 2003.

# Perché studiare i concetti dei LP?

1. Accrescimento della capacità di esprimere idee
2. Miglioramento della capacità di scelta di LP appropriati (a fronte di un progetto)
3. Accrescimento della capacità di imparare nuovi linguaggi
4. Accrescimento della capacità di progettare nuovi linguaggi
5. Contributo all'avanzamento della tecnologia del software (LP) nella “giusta” direzione

# Che cosa è un LP?


**Def:** “*Un LP è uno strumento di astrazione che permette di specificare computazioni tali da poter essere eseguite su un elaboratore*”.

- Esistono migliaia di LP, ognuno progettato in modo da soddisfare certi requisiti.
- Progettista di un LP deve bilanciare due requisiti fondamentali:
  1. Computazione espressa convenientemente per la persona;
  2. Uso efficiente degli elaboratori.

# Verso LP di alto livello

- LP inventati per rendere l'uso degli elaboratori (macchine) facile.
- Termine informale di **livello** utile per una distinzione di massima dei LP.
- Linguaggio macchina: basso livello (pieno di dettagli che hanno a che fare più con il modo con cui funziona la macchina che con l'oggetto della computazione).
- LP progettati in modo da essere
  - alto livello** → indipendente dalla macchina
  - general-purpose** → applicabile ad un ampio dominio
- Estremi:
  - L naturale
    - verboso
    - informale
    - ambiguo
  - macchina: incomprensibile (solo 0 e 1)

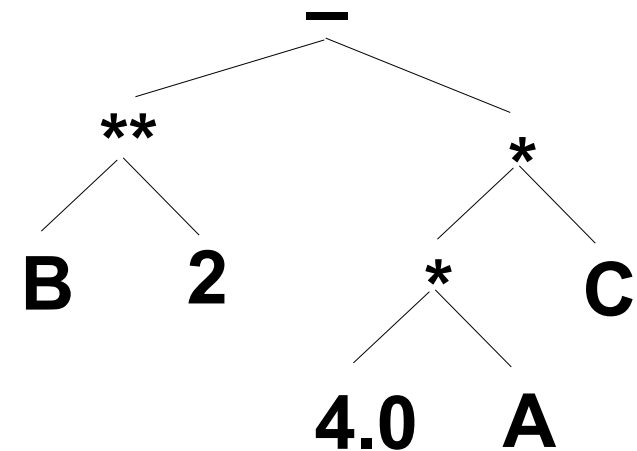
# Verso LP di alto livello (ii)

- Inizio della storia evolutiva dei LP verso l'alto: definizione di un linguaggio simbolico (mnemonico) da tradurre manualmente.
- Linguaggio assembly: tradotto in L macchina automaticamente (“*programmazione automatica*”).
- LP di alto livello hanno sostituito il linguaggio assembly virtualmente in tutte le aree della programmazione, poiché:
  1. Notazione  familiare  
leggibile
  2. Indipendenti dalla macchina (portabilità)
  3. Disponibilità di librerie di programmi
  4. Permettono analisi del programma → supporta l'individuazione di errori

# Verso LP di alto livello (iii): Programmazione scientifica

**FORTRAN** (FORmula TRANslation): permetteva di scrivere espressioni matematiche in modo naturale (IBM 704)

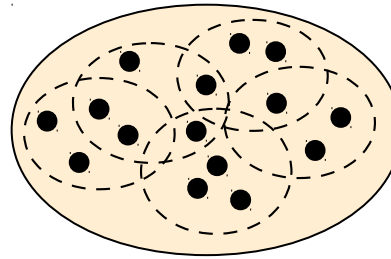
<i>Matematicamente</i>	<i>Pragmaticamente</i>
<b><math>b^2 - 4ac</math></b>	<b><math>B^{**}2 - 4.0*A*C</math></b>
Esprime un risultato	Espressione trattata come un algoritmo per computare il risultato → traducibile in linguaggio macchina!



# Paradigmi

- Ogni LP supporta uno stile di programmazione  $\equiv$  **paradigma di programmazione**
- **Def:** “LP che suggerisce un particolare paradigma si dice **orientato al paradigma**”

- Un LP può avere diversi paradigmi:



- **Def:** “Un LP che supporta diversi paradigmi si dice **ibrido**” (C++)

- Quando  $\left\langle \begin{array}{l} \text{metodo di design} \\ \text{LP} \end{array} \right\rangle$  con stesso paradigma



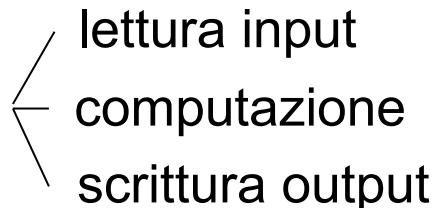
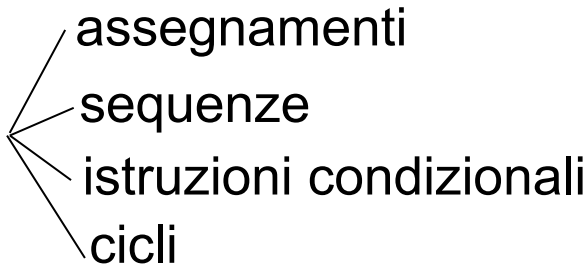
Astrazioni del design direttamente mappabili sui componenti del programma

- Altrimenti: scollamento  $\rightarrow$  aumento del costo della codifica:

Programma deve implementare  $\left\langle \begin{array}{l} \text{soluzione del problema} \\ \text{concetti del paradigma} \end{array} \right\rangle$  (OO in FORTRAN)

# Paradigmi (ii)

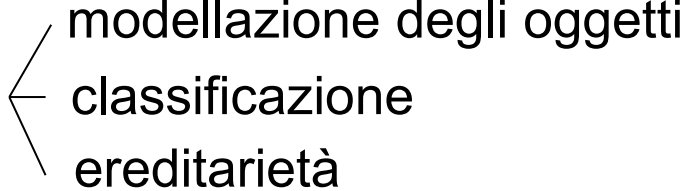
## 1. Programmazione imperativa

- Programma = sequenza di passi
- Ad ogni passo 
  - lettura input
  - computazione
  - scrittura output
- Meccanismo di astrazione: procedura (istruzione "complessa") → riuso
- Costrutti fondamentali: 
  - assegnamenti
  - sequenze
  - istruzioni condizionali
  - cicli
- FORTRAN, Cobol, C, Pascal



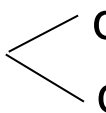
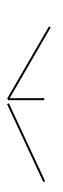
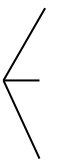
# Paradigmi (iii)

## 2. Programmazione orientata agli oggetti

- Programma = collezione di oggetti che interagiscono passandosi messaggi che trasformano il loro stato.
- Costrutti fondamentali: 
  - modellazione degli oggetti
  - classificazione
  - ereditarietà
- Smalltalk, C++, Java, C#, Ruby

# Paradigmi (iv)

## 3. Programmazione funzionale

- Programma = collezione di funzioni matematiche
- Ogni funzione ha 
  - dominio
  - codominio
- Costrutti fondamentali 
  - composizione
  - condizionali
  - ricorsione
- Non esistono: 
  - variabili
  - assegnamenti
  - istruzioni di controllo
- Lisp, Scheme, ML, Haskell

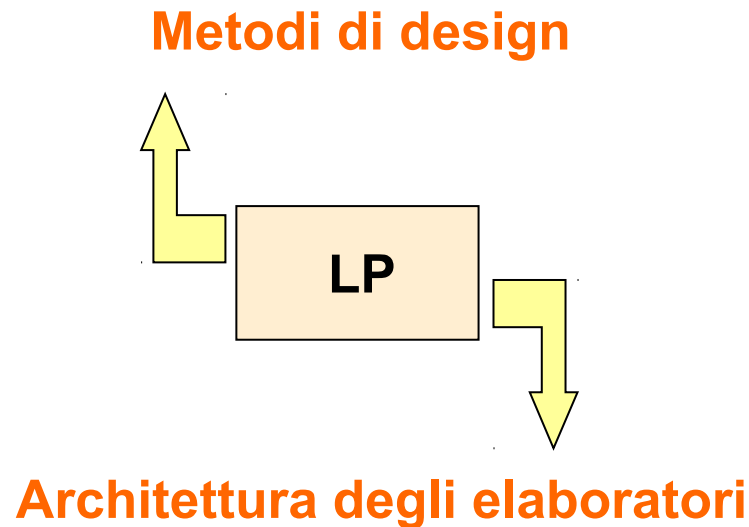
# Paradigmi (v)

## 4. Programmazione logica

- Programma = collezione di dichiarazioni logiche su cosa una certa funzione deve computare piuttosto che sul come
- Esecuzione: applica le dichiarazioni per trovare possibili soluzioni al problema
- Tipicamente: problemi risolvibili mediante "tentativi"
- Backtracking: ritorno sui propri passi per percorrere una strada alternativa
- Nondeterminismo: soluzione del problema non unica
- Prolog

# LP ed architettura degli elaboratori

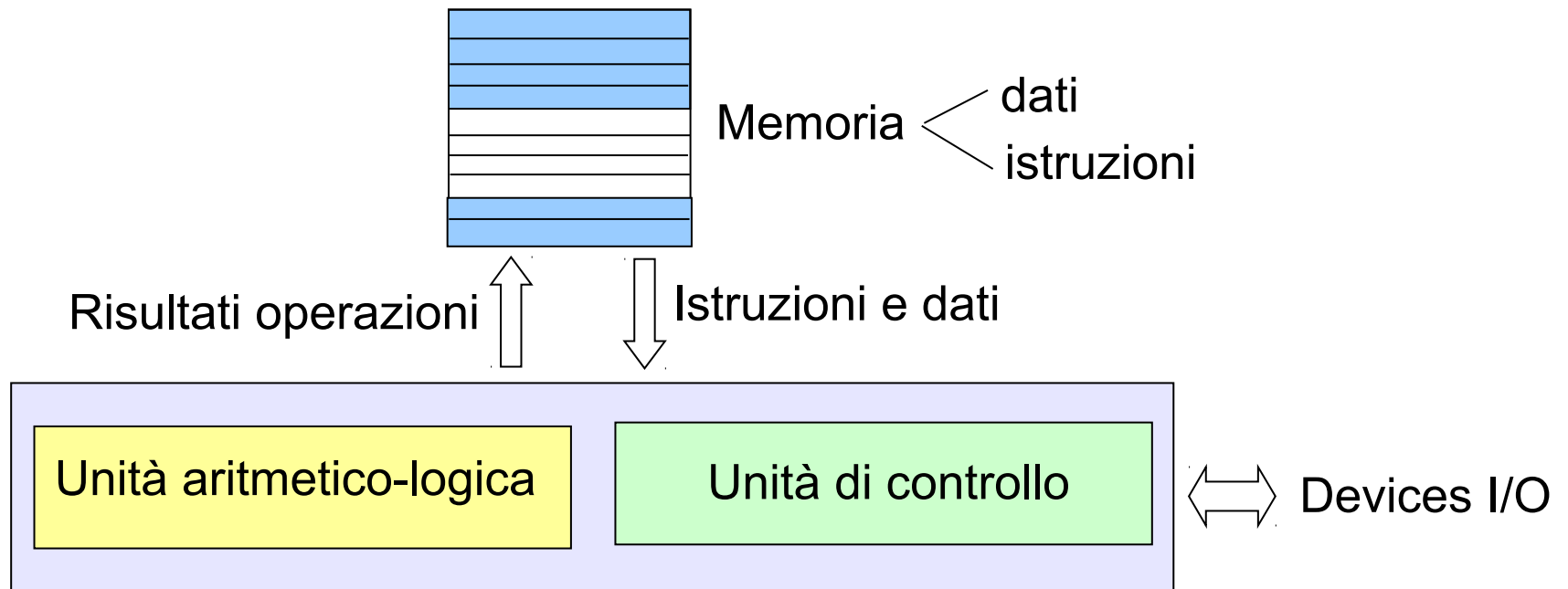
- Doppia influenza sui LP:



- Metodi di design → requisiti sul LP in modo da supportare meglio lo sviluppo (design) del sw
- Architettura degli elaboratori → requisiti sul LP in modo che possa essere implementato efficientemente sulle macchine correnti (architettura di Von Neumann)

# LP ed architettura degli elaboratori (ii)

Architettura della macchina di Von Neumann:



- CPU: preleva una istruzione alla volta dalla memoria
- Esecuzione di una istruzione → prelievo di dati dalla memoria + manipolazione dei dati + copiatura dei risultati nella memoria

**Modello  
computazionale**

**Transizione di stato della macchina**

# LP ed architettura degli elaboratori (iii)

- LP convenzionali (imperativi): visti come astrazione di una architettura di Von Neumann
- Astrazione  $\begin{cases} \text{evidenzia gli aspetti rilevanti} \\ \text{ignora i dettagli} \end{cases}$
- Modello computazionale di un LP imperativo = Esecuzione sequenziale di istruzioni, ognuna delle quali cambia lo stato della computazione mediante la modifica dei valori di un insieme di variabili

LP	Architettura Von Neumann
Esecuzione sequenziale delle istruzioni	Prelievo sequenziale della CPU + esecuzione
Variabile (nome, valore)	Cella di memoria (indirizzo, contenuto)
Stato = valore delle variabili	Stato = contenuto della memoria

- Storicamente: LP si sono evoluti verso livelli di astrazione crescenti

# LP ed architettura degli elaboratori (v)

- Abbandono del modello computazionale di Von Neumann:



- Fondamenti concettuali non definiti in relazione alla arch. di Von Neumann



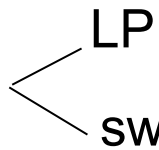
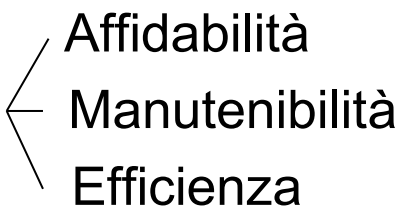
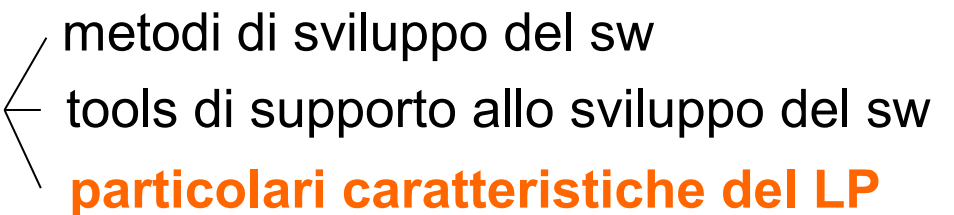
Conflitto con efficienza di esecuzione



(compromesso)

Miglioramento dell'efficienza mediante l'introduzione di costrutti imperativi

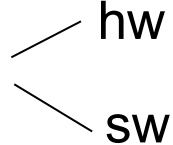
# Qualità dei LP

- LP = strumento per lo sviluppo del sw → correlazione qualità 
  - LP
  - sw
- Requisiti di qualità del sw 
  - Affidabilità
  - Manutenibilità
  - Efficienza
- Requisiti di qualità soddisfacenti mediante 
  - metodi di sviluppo del sw
  - tools di supporto allo sviluppo del sw
  - particolari caratteristiche del LP**



# Requisiti di qualità del sw

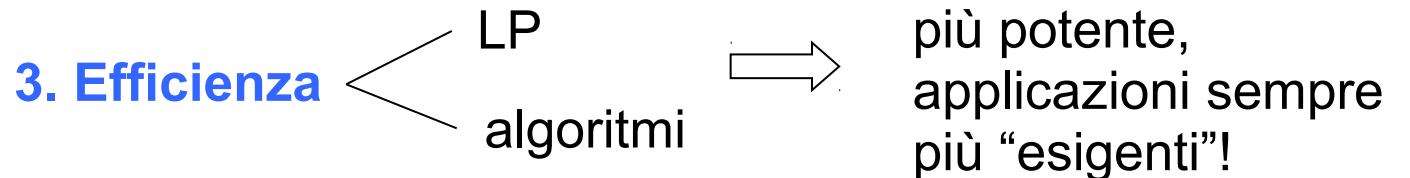
## 1. Affidabilità

- Sw deve rispettare i suoi requisiti in ogni circostanza
- Idealmente: tollerante ai guasti 
  - hw
  - sw
- Importanza crescente, perché sw sempre più usato in ambienti critici:
  - Impianti nucleari
  - Navicelle spaziali
  - Strumentazione chirurgica

# Requisiti di qualità del sw (ii)

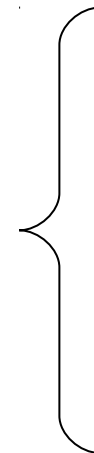
## 2. Manutenibilità

- Possibilità di intervenire sul sw esistente per soddisfare nuovi requisiti, poiché:
  - Sw sempre più costoso e sistemi sempre più complessi (non si può gettare)
  - Impossibile catturare tutti i requisiti reali “al primo colpo”





# LP e affidabilità

- Affidabilità supportata da svariate caratteristiche del LP



- Scrivibilità
- Leggibilità
- Semplicità
- Sicurezza
- Robustezza

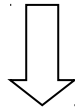
- Però:

- Soggettive
  - Qualitative
  - ▪ Non indipendenti (correlate)
  - In conflitto
- 
- alcune

# LP e affidabilità (ii)

## 1. Scrivibilità

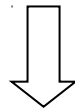
**Def:** “Misura di quanto facilmente un LP può essere usato per scrivere un programma relativo ad un certo dominio applicativo”



- Non ha senso confrontare LP1 e LP2 in un dominio per il quale LP1 è stato progettato, mentre LP2 no!

Esempio:  $\left\{ \begin{array}{ll} \text{FORTRAN} & \rightarrow \text{scientifico} \\ \text{COBOL} & \rightarrow \text{gestionale} \end{array} \right.$

- **Principio di scrivibilità:** Programmatore concentrato sul problem solving, senza essere distratto dai dettagli/trucchi del LP

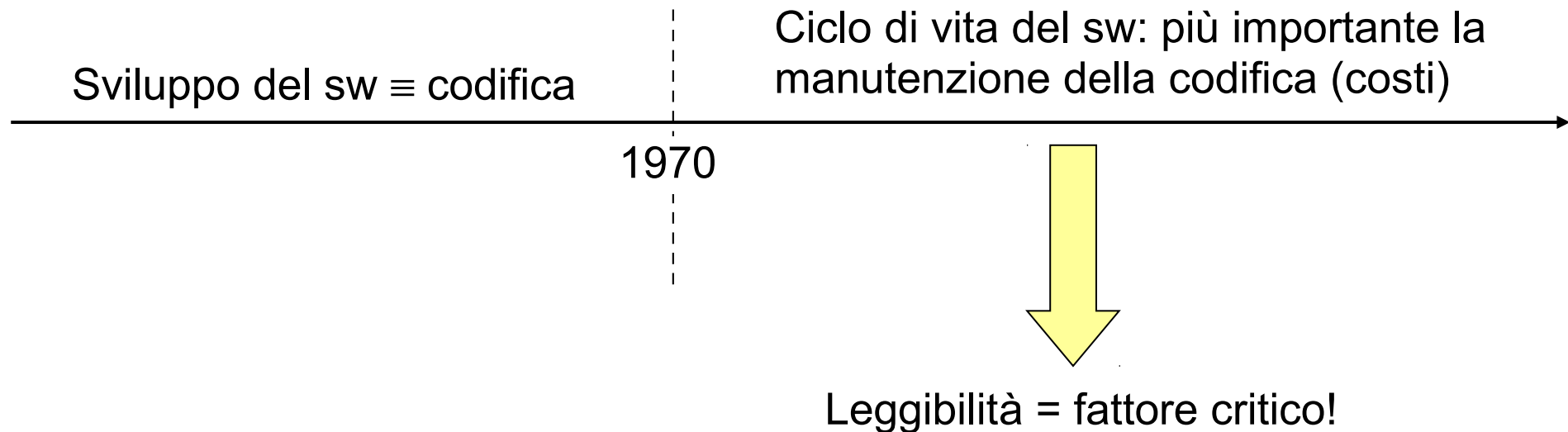


LP di alto livello più scrivibili dell'assembly

# LP e affidabilità (iii)

## 2. Leggibilità

**Def:** “Misura della possibilità di seguire la logica del programma leggendolo”



# LP e affidabilità (iv)

**Fatto storico** (22 luglio 1962): Razzo contenente la sonda *Mariner I* diretta verso Venere viene distrutto dopo 290 secondi dal lancio.

mancante!

Frammento di codice  
del computer di terra:

```
if not in contatto radar con il razzo then  
    non correggere il suo cammino di volo.
```

Nota: Programma (bacato) usato precedentemente con successo in 4 lanci lunari!

- Scienziati NASA hanno cercato di spiegare l'errore ad una commissione d'indagine
- Tecniche di verifica/convalida {
  - Ispezione del codice fatta da altri ← **leggibilità!**
  - Codice testato (300 esecuzioni di test)

Oss: Supporto alla leggibilità = def {

- operazioni
- tipi di dati

} separati dal punto d'uso

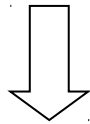
# LP e affidabilità (v)

## 3. Semplicità

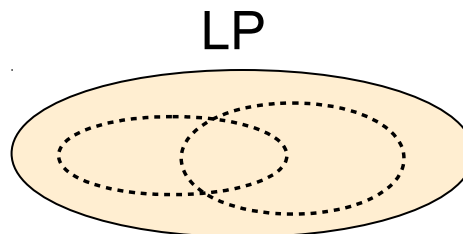
**Def:** “Grado di riduzione del numero di costrutti del LP”

Problemi di un LP complicato:

- a) LP con molti costrutti di base: difficile da imparare
- b) Programmatore: tende ad imparare un sottoinsieme del LP



Possibilità di scollamento tra i sottoinsiemi del { programmatore  
lettore (verifica/convalida)



# LP e affidabilità (vi)

- Contrario della semplicità: **molteplicità**  $\equiv$  esistenza di costrutti alternativi per esprimere la stessa operazione

Incremento di `cont` in C:

```
cont = cont + 1  
cont += 1  
cont++  
++cont
```

$\Rightarrow$  stesso significato se usate stand-alone

- Altro pb: **overloading** operatori a  $\left\{ \begin{array}{l} \text{proposito: '+' per interi/reali} \\ \text{sproposito: } V_1 + V_2 \equiv \sum_{v \in V_1} v + \sum_{v \in V_2} v \end{array} \right.$

- Pb opposto = LP troppo semplice (*Assembly*: nonostante istruzioni semplici, programma poco leggibile!)



# LP e affidabilità (vii)

## 4. Sicurezza

**Def:** “Mancanza di costrutti che permettono la scrittura di programmi pericolosi”

Esempi:  $\left\{ \begin{array}{l} \text{goto} \\ \text{puntatori} \end{array} \right\}$  sorgenti di pericolo di errori nascosti che emergono solo dopo la consegna del sw

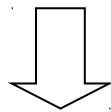
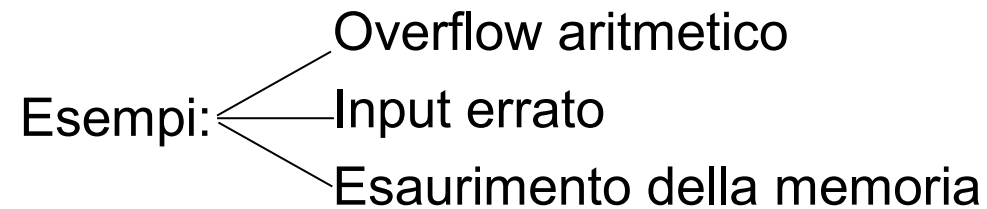
- Similitudine  $\left\{ \begin{array}{l} \text{goto: amplia l'insieme delle istruzioni successive} \\ \text{puntatore: amplia l'insieme delle celle di memoria referenziate da var} \end{array} \right.$
- Hoare (1973): *“La loro comparsa nei LP è stato un passo indietro da cui non ci si potrà più risanare”*

Altra faccia della medaglia: riduzione dei pericoli → riduzione di  $\left\{ \begin{array}{l} \text{flessibilità} \\ \text{efficienza} \end{array} \right.$

# LP e affidabilità (viii)

## 5. Robustezza

**Def:** “Grado di capacità di reazione ad eventi indesiderati”

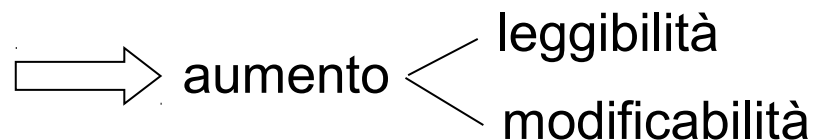
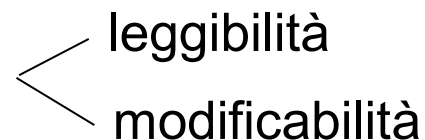


**Gestione delle eccezioni** = Possibilità di intrappolare tali eventi e definire una risposta appropriata alla loro manifestazione

# LP e manutenibilità

Requisito: LP devono supportare lo sviluppo di programmi facilmente modificabili

- **Fattorizzazione** = possibilità di fattorizzare duplicazioni in singole unità      Es:  sottoprogramma  
costanti simboliche


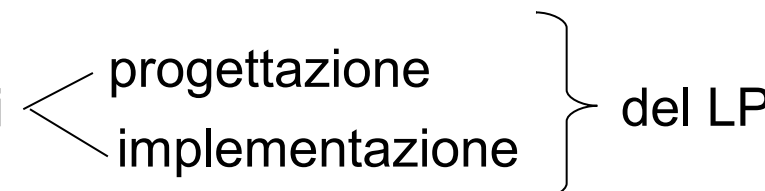
 aumento  leggibilità  
modificabilità

- **Localizzazione** = possibilità di restringere l'effetto di un costrutto ad una piccola porzione del programma

Es: ADT → possibilità di modificare la struttura dati incapsulata in una classe senza toccare il resto del programma (mantenendo invariate le chiamate delle operazioni che manipolano le strutture dati)

Oss: fattorizzazione promuove localizzazione (Es: IVA = 20)

# LP ed efficienza

- Diverse accezioni 
- Efficienza = risultato della combinazione di 
- Esempi di influenza negativa sulla efficienza:
  1. **Progettazione** del LP influenza negativamente l'efficienza quando non permette certe ottimizzazioni al compilatore

Funzione C:  $f(y)$  che modifica  $z$

$$x = f(y) + z + f(y) + z \neq x = 2*f(y) + 2*z$$

2. **Implementazione** del LP influenza negativamente l'efficienza (spazio) quando non riusa la memoria rilasciata dal programma