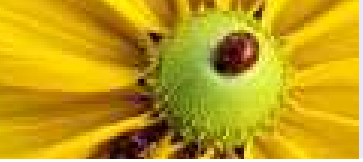
Linguaggi di Programmazione I – Lezione 1

Prof. P. A. Bonatti
mailto://bonatti@na.infn.it
http://people.na.infn.it/~bonatti

Si ringrazia il Prof. Marcello Sette per il materiale didattico

6 marzo 2012



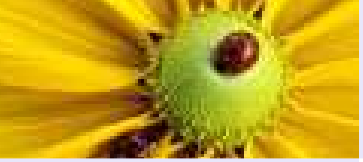
Panoramica della lezione

Introduzione al corso

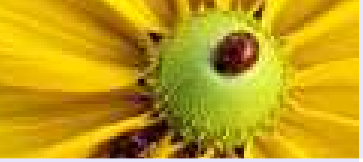
Linguaggi (di programmazione)

Macchine astratte

Paradigmi computazionali

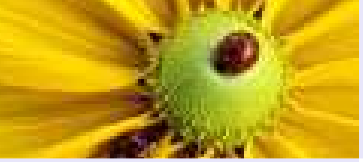


Introduzione al corso



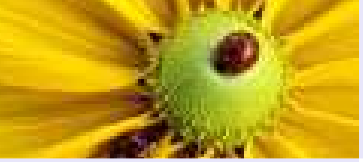
Obbiettivi specifici

- Spiegare le differenze tra i vari paradigmi di programmazione - in particolare dei paradigmi imperativo e ad oggetti - e sul loro impatto sullo stile di soluzione dei problemi.
- Mostrare cenni sui criteri di progetto e le tecniche d'implementazione dei linguaggi di programmazione
- Capacità media di progettare *ad oggetti*.



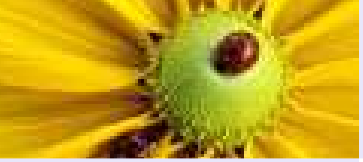
Obbiettivi specifici

- Spiegare le differenze tra i vari paradigmi di programmazione - in particolare dei paradigmi imperativo e ad oggetti - e sul loro impatto sullo stile di soluzione dei problemi.
- Mostrare cenni sui criteri di progetto e le tecniche d'implementazione dei linguaggi di programmazione
- Capacità media di progettare *ad oggetti*.
- Capacità media di programmare in Java.



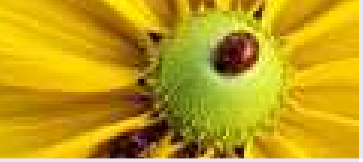
Obbiettivi generali

- Migliorare l'abilità nel risolvere i problemi.



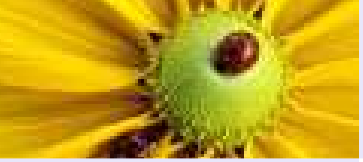
Obbiettivi generali

- Migliorare l'abilità nel risolvere i problemi.
- Imparare a usare meglio i linguaggi di programmazione.



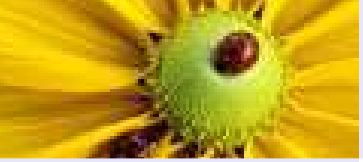
Obbiettivi generali

- Migliorare l'abilità nel risolvere i problemi.
- Imparare a usare meglio i linguaggi di programmazione.
- Imparare a scegliere più intelligentemente, in dipendenza del problema, il linguaggio di programmazione.



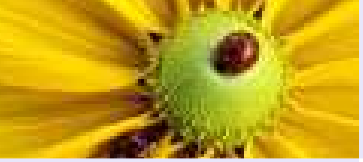
Obbiettivi generali

- Migliorare l'abilità nel risolvere i problemi.
- Imparare a usare meglio i linguaggi di programmazione.
- Imparare a scegliere più intelligentemente, in dipendenza del problema, il linguaggio di programmazione.
- Aumentare la capacità di imparare linguaggi di programmazione (che sono TANTI!).



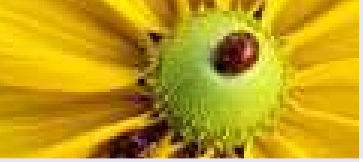
Scheduling

- Parte prima – 8 lezioni
  1. Paradigmi dei linguaggi di programmazione.
  2. Il modello imperativo.
  3. Il modello ad oggetti.
  4. Progettazione orientata ad oggetti e UML.



Scheduling

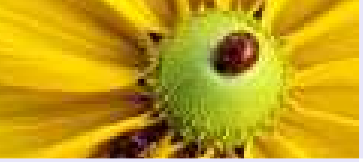
- Parte prima – 8 lezioni
  1. Paradigmi dei linguaggi di programmazione.
  2. Il modello imperativo.
  3. Il modello ad oggetti.
  4. Progettazione orientata ad oggetti e UML.
- Parte seconda (Java) – 15 lezioni
  1. Studio dei costrutti fondamentali: identificatori, parole chiave, tipi; espressioni e controllo di flusso; array; ereditarietà; overloading e overriding; polimorfismo.
  2. Studio dei costrutti più avanzati: qualificatori di classi, metodi, attributi; classi astratte; interfacce; classi interne; gestione degli errori: eccezioni.



Esame

L'esame consisterà in due scritti:

1. sulla prima parte del corso
2. su Java



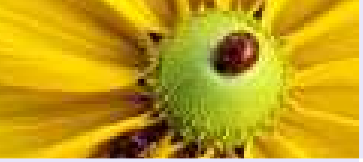
Esame

L'esame consisterà in due scritti:

1. sulla prima parte del corso
2. su Java

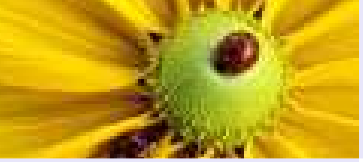
Solo in casi per noi dubbi, un orale. Inoltre:

- In caso di grave insufficienza non sarà possibile sostenere altre prove scritte fino alla *sessione* successiva
- E' possibile mantenere il voto di una prova scritta fino alla fine dell'anno accademico
- L'ultima prova sostenuta cancella le precedenti (anche se è andata peggio)



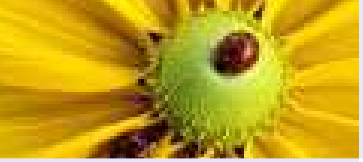
Altre informazioni

- *Libri di testo*:
  - Dershem - Jipping. *Programming languages: structures and models*.
  - Wampler. *The essence on object oriented programming with Java and UML*.
  - Fowler. *UML distilled*.
  - Eckel. *Thinking in Java*.
  - Gabbrielli - Martini. *Linguaggi di programmazione: Principi e paradigmi*

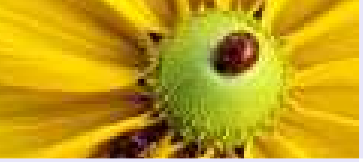


Altre informazioni

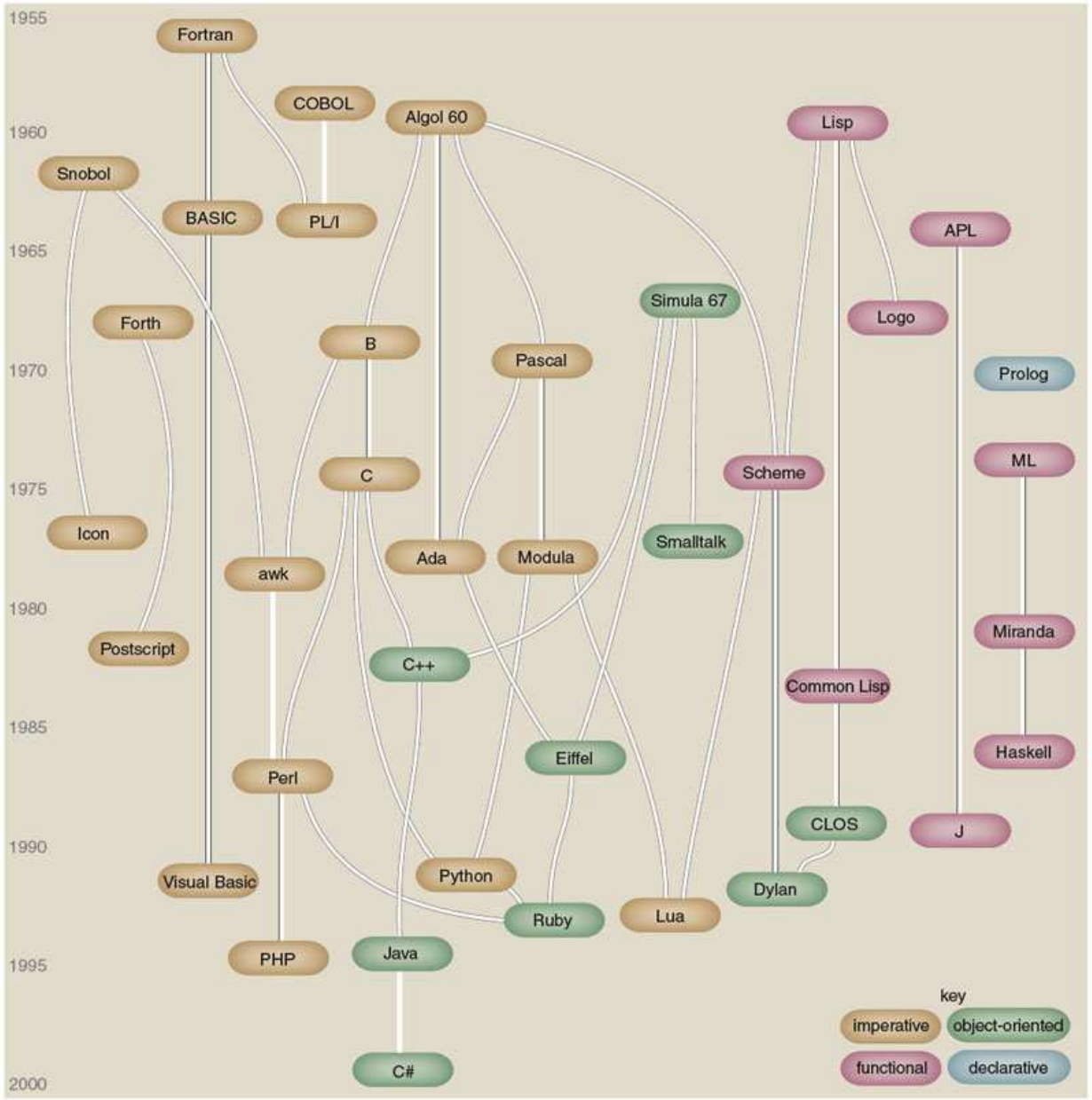
- *Libri di testo*:
  - Dershem - Jipping. *Programming languages: structures and models*.
  - Wampler. *The essence on object oriented programming with Java and UML*.
  - Fowler. *UML distilled*.
  - Eckel. *Thinking in Java*.
  - Gabbrielli - Martini. *Linguaggi di programmazione: Principi e paradigmi*
- *Materiali*: http://people.na.infn.it/~bonatti/didattica/
  - Accesso con `ldprog-1` e `algol-60`
- *Ricevimento studenti*: per appuntamento con email al docente

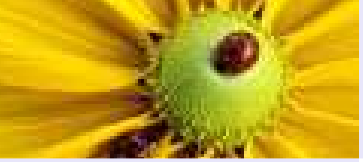


Linguaggi (di programmazione)



Sono ~ 40?





Sono ~ 80?

Mother Tongues

Tracing the roots of computer languages through the ages

Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java, and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers – electronic lexicographers, if you will – aim to save, or at least document, the lingo of classic software. They're combining the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua francas. Among the most endangered are Ada, APL, B (the predecessor of C), Lisp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at *www.informatik.uni-freiburg.de/Java/misc/lang_list.html*. – **Michael Menduno**

Key

1954 Year Introduced

Active: thousands of users

Protected: taught at universities; compilers available

Endangered: usage dropping off

Extinct: no known active users or up-to-date compilers

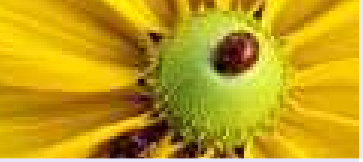
Lineage continues

Survival of the Fittest

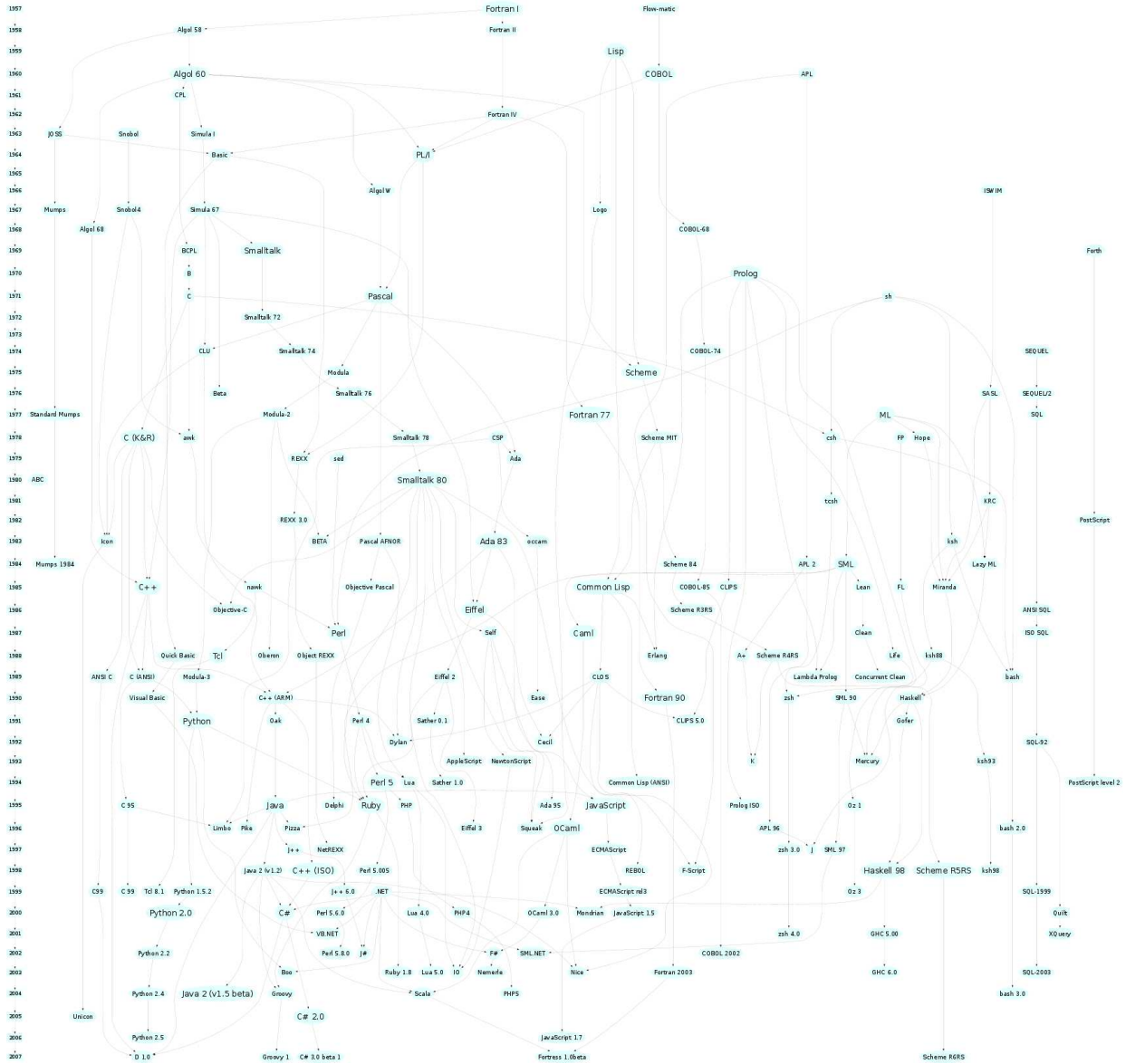
Reasons a language endures, with examples of some classic tongues

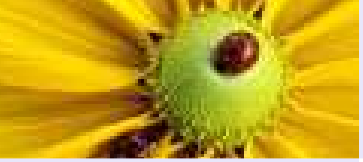
- **Appeals to a wide audience** C (bolstered by the popularity of Unix)
- **Gets a job done** Cobol (designed for business-report writing)
- **Delivers new functionality** Java (runs on any hardware platform)
- **Fills a niche** Mathematica (speeds up complex computations)
- **Offers a modicum of elegance** Icon (has friendly, line-oriented syntax)
- **Has a powerful user base or backer** C# (developed by Microsoft for .Net)
- **Has a charismatic leader** Perl (programmer-author Larry Wall)

Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University



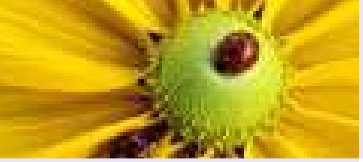
Quanti sono?





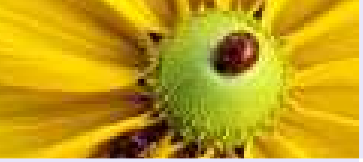
Quanti sono?

- C'è chi dice addirittura qualche migliaio
  - Come orientarsi?
  - Come usarli *bene*?
  - Come apprendere in fretta quelli nuovi?
- Occorre comprensione astratta delle caratteristiche dei linguaggi per coglierne somiglianze/differenze
- e per comprendere lo scopo di ciascun costrutto (ovvero i principi del *language design*)



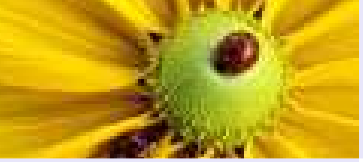
Breve storia

1954	FORTRAN (FORmula TRANslation)



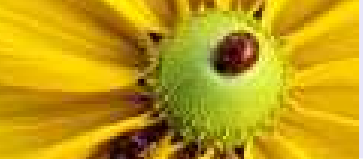
Breve storia

1954	FORTRAN (FORmula TRANslation)
1960	COBOL (Common Business Oriented Language)
	ALGOL 60 (Algorithmic Oriented Language)
	PL/1 (Programming Language 1)
	Simula 67
	ALGOL 68
	PASCAL
	LISP (LISt Processing)
	APL
	BASIC



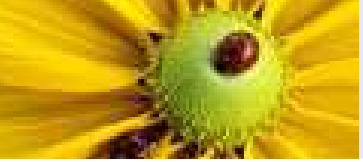
Breve storia

1954	FORTRAN (FORmula TRANslation)
1960	COBOL (Common Business Oriented Language) ALGOL 60 (Algorithmic Oriented Language) PL/1 (Programming Language 1) Simula 67 ALGOL 68 PASCAL LISP (LISt Processing) APL BASIC
1970/80	PROLOG SMALLTALK C MODULA/2 ADA



Storia dei concetti introdotti dai progenitori dell'oggi

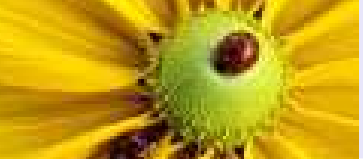
Fortran: nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output. Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.



Storia dei concetti introdotti dai progenitori dell'oggi

Fortran: nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output. Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.

Cobol: indipendenza dalla macchina e statement "English like". Orientato ai database. Introduce il record.

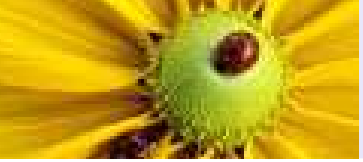


Storia dei concetti introdotti dai progenitori dell'oggi

Fortran: nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output. Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.

Cobol: indipendenza dalla macchina e statement "English like". Orientato ai database. Introduce il record.

Algol60: indipendenza dalla macchina e definizione mediante grammatica(bakus-naur form), strutture a blocco, supporto generale dell'iterazione e ricorsione.



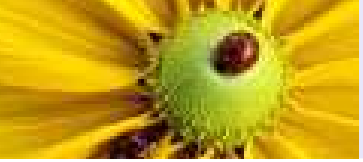
Storia dei concetti introdotti dai progenitori dell'oggi

Fortran: nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output. Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.

Cobol: indipendenza dalla macchina e statement "English like". Orientato ai database. Introduce il record.

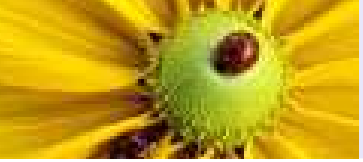
Algol60: indipendenza dalla macchina e definizione mediante grammatica(bakus-naur form), strutture a blocco, supporto generale dell'iterazione e ricorsione.

Lisp: primo vero linguaggio di manipolazione simbolica, paradigma funzionale, non c'è lo statement di assegnazione, e quindi concettualmente non c'è "il valore" ovvero l'idea di cambiare lo stato della memoria. Non c'è differenza concettuale fra funzione e dato: dipende dall'uso. Prima versione essenzialmente non tipata.



Storia dei concetti introdotti dai progenitori dell'oggi

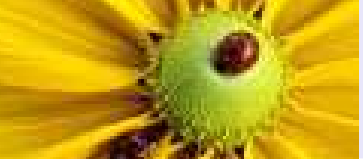
Prolog: primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (generate and test). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.



Storia dei concetti introdotti dai progenitori dell'oggi

Prolog: primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (generate and test). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.

Simula 67: classe come incapsulamento di dati e procedure, istanze delle classi (oggetti): anticipatorio del concetto di tipo di dato astratto implementati in Ada e Modula2, e del concetto di classe di Smalltalk e C++.

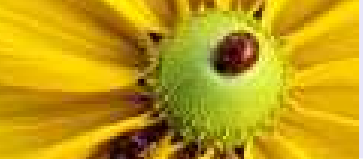


Storia dei concetti introdotti dai progenitori dell'oggi

Prolog: primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (generate and test). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.

Simula 67: classe come incapsulamento di dati e procedure, istanze delle classi (oggetti): anticipatorio del concetto di tipo di dato astratto implementati in Ada e Modula2, e del concetto di classe di Smalltalk e C++.

PL/1: abilità ad eseguire procedure specificate quando si verifica una condizione eccezionale; "multitasking", cioè specificazione di tasks che possono essere eseguiti in concorrenza.



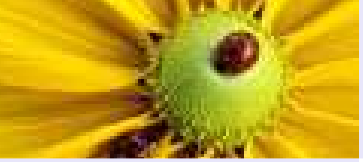
Storia dei concetti introdotti dai progenitori dell'oggi

Prolog: primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (generate and test). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.

Simula 67: classe come incapsulamento di dati e procedure, istanze delle classi (oggetti): anticipatorio del concetto di tipo di dato astratto implementati in Ada e Modula2, e del concetto di classe di Smalltalk e C++.

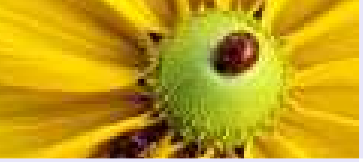
PL/1: abilità ad eseguire procedure specificate quando si verifica una condizione eccezionale; "multitasking", cioè specificazione di tasks che possono essere eseguiti in concorrenza.

Pascal: programmazione strutturata, tipi di dato definiti da utente,ricchezza di strutture dati. Ma ancora niente encapsulation; si dovrà aspettare Modula.



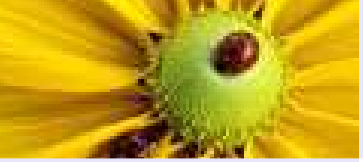
Terminologia

- **Linguaggio di programmazione**: è un linguaggio che è usato per esprimere (mediante un programma) un processo con il quale un processore può risolvere un problema.



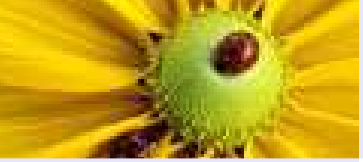
Terminologia

- **Linguaggio di programmazione**: è un linguaggio che è usato per esprimere (mediante un programma) un processo con il quale un processore può risolvere un problema.
- **Processore**: è la macchina che eseguirà il processo descritto dal programma; non si deve intendere come un singolo oggetto, ma come una *architettura di elaborazione*.



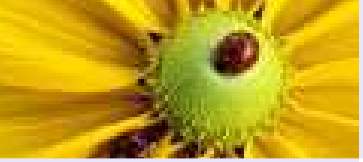
Terminologia

- **Linguaggio di programmazione**: è un linguaggio che è usato per esprimere (mediante un programma) un processo con il quale un processore può risolvere un problema.
- **Processore**: è la macchina che eseguirà il processo descritto dal programma; non si deve intendere come un singolo oggetto, ma come una *architettura di elaborazione*.
- **Programma**: è l'espressione codificata di un processo.



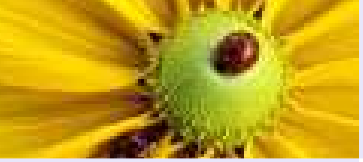
Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli equivalenti in potere espressivo al linguaggio della macchina di Turing.



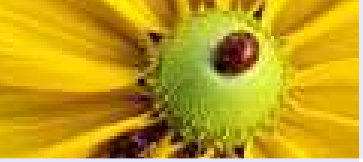
Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli equivalenti in potere espressivo al linguaggio della macchina di Turing.
- Sono completi solo quelli che riescono ad esprimere anche programmi di cui non è decidibile la terminazione



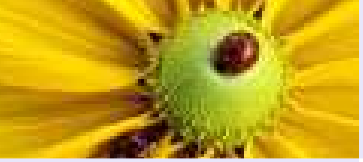
Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli equivalenti in potere espressivo al linguaggio della macchina di Turing.
- Sono completi solo quelli che riescono ad esprimere anche programmi di cui non è decidibile la terminazione
- SQL non è un linguaggio completo (perché la terminazione dei programmi è sempre decidibile).



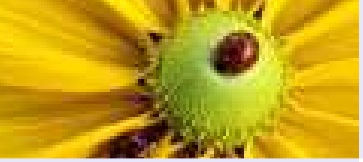
Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli equivalenti in potere espressivo al linguaggio della macchina di Turing.
- Sono completi solo quelli che riescono ad esprimere anche programmi di cui non è decidibile la terminazione
- SQL non è un linguaggio completo (perché la terminazione dei programmi è sempre decidibile). Attenzione alle ultime incarnazioni dei linguaggi.

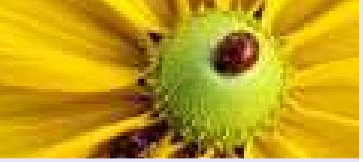


Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli equivalenti in potere espressivo al linguaggio della macchina di Turing.
- Sono completi solo quelli che riescono ad esprimere anche programmi di cui non è decidibile la terminazione
- SQL non è un linguaggio completo (perché la terminazione dei programmi è sempre decidibile). Attenzione alle ultime incarnazioni dei linguaggi.
- HTML non è un linguaggio completo (idem).
- Per mostrare la completezza di un linguaggio: usarlo per simulare arbitrarie macchine di Turing

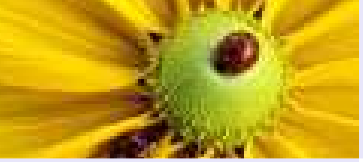


Macchine astratte



Definizione

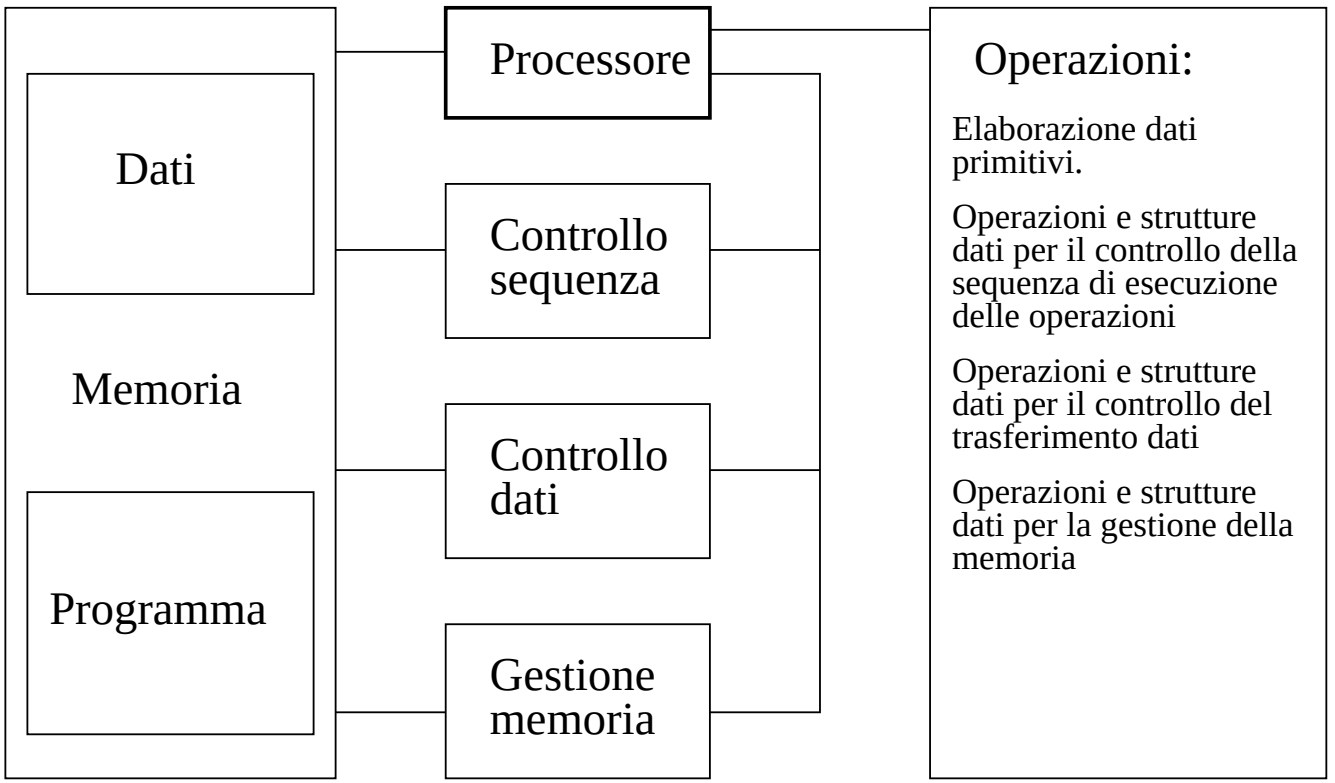
Dato un linguaggio di programmazione L, una macchina astratta per L (in simboli, M_L) è un qualsiasi insieme di strutture dati e algoritmi che permettano di memorizzare ed eseguire programmi scritti in L.



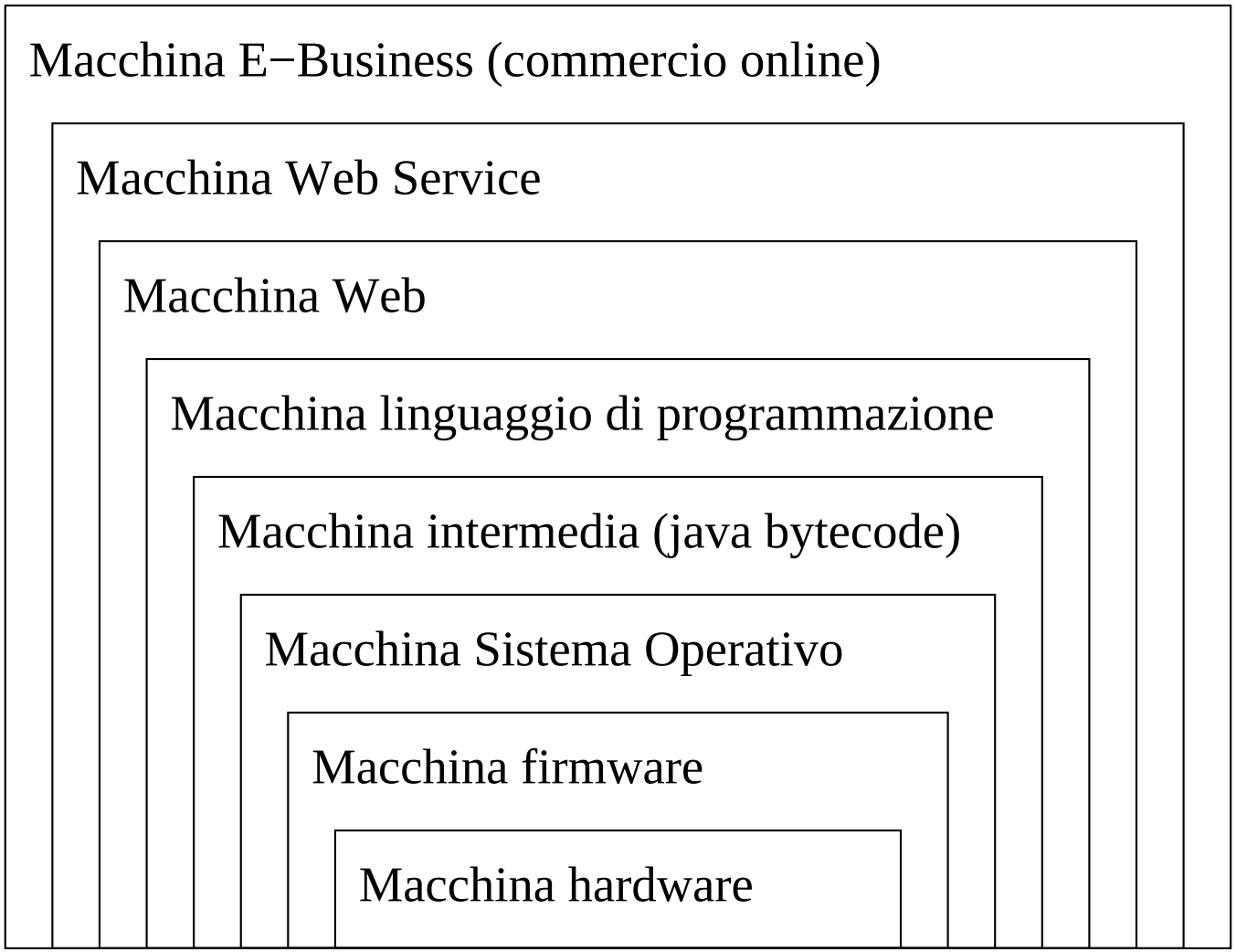
Definizione

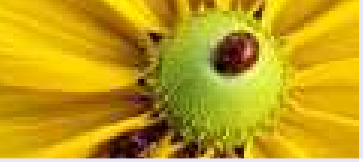
Dato un linguaggio di programmazione L, una macchina astratta per L (in simboli, M_L) è un qualsiasi insieme di strutture dati e algoritmi che permettano di memorizzare ed eseguire programmi scritti in L.

La struttura di una macchina astratta è (essenzialmente memoria e processore):

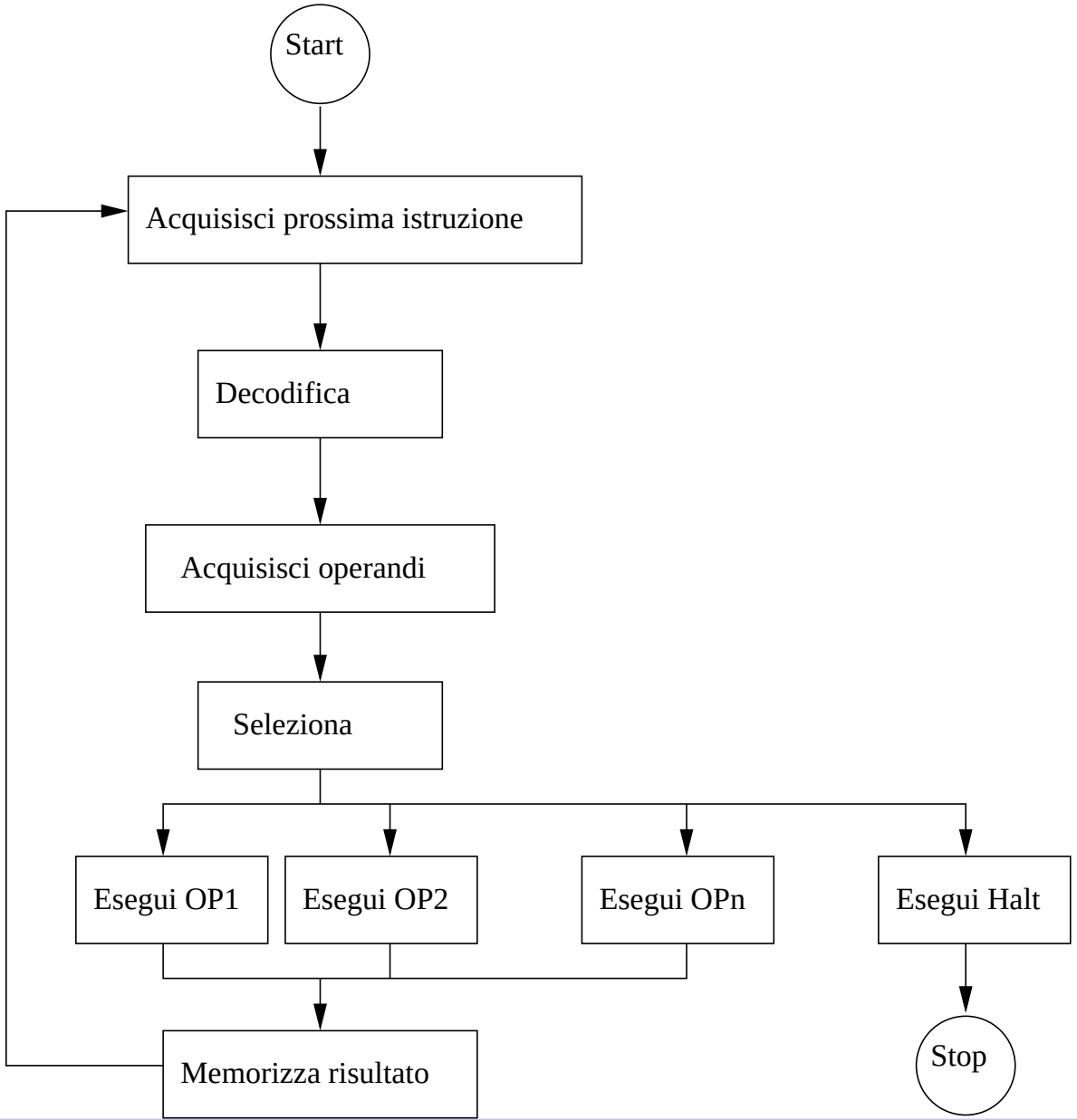


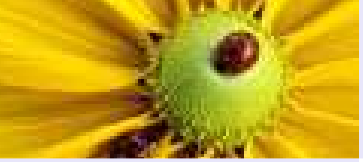
Esempio





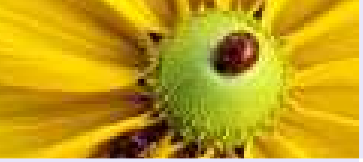
Processore della macchina astratta





Tecnologie di realizzazione di macchina astratta

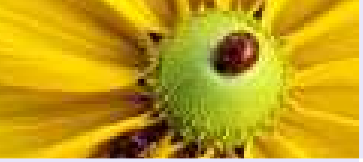
Hardware: Si era pensato per Lisp e Prolog



Tecnologie di realizzazione di macchina astratta

Hardware: Si era pensato per Lisp e Prolog

Firmware: Soluzione spesso adottata per flessibilità o semplicità/economicità di progetto

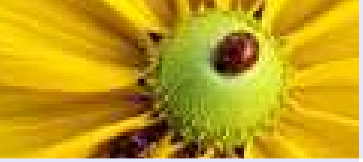


Tecnologie di realizzazione di macchina astratta

Hardware: Si era pensato per Lisp e Prolog

Firmware: Soluzione spesso adottata per flessibilità o semplicità/economicità di progetto

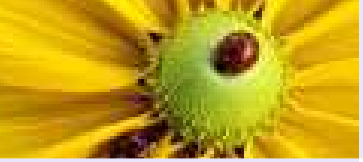
Software: Ad es. macchina astratta Java, o Warren Abstract Machine (Prolog)



Traduttori

- **Interpreti**: traducono ed eseguono un costrutto alla volta.

  PRO: debug - fase di sviluppo: interazione più snella.



Traduttori

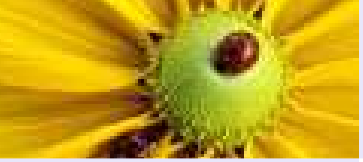
- **Interpreti**: traducono ed eseguono un costrutto alla volta.

  PRO: debug - fase di sviluppo: interazione più snella.

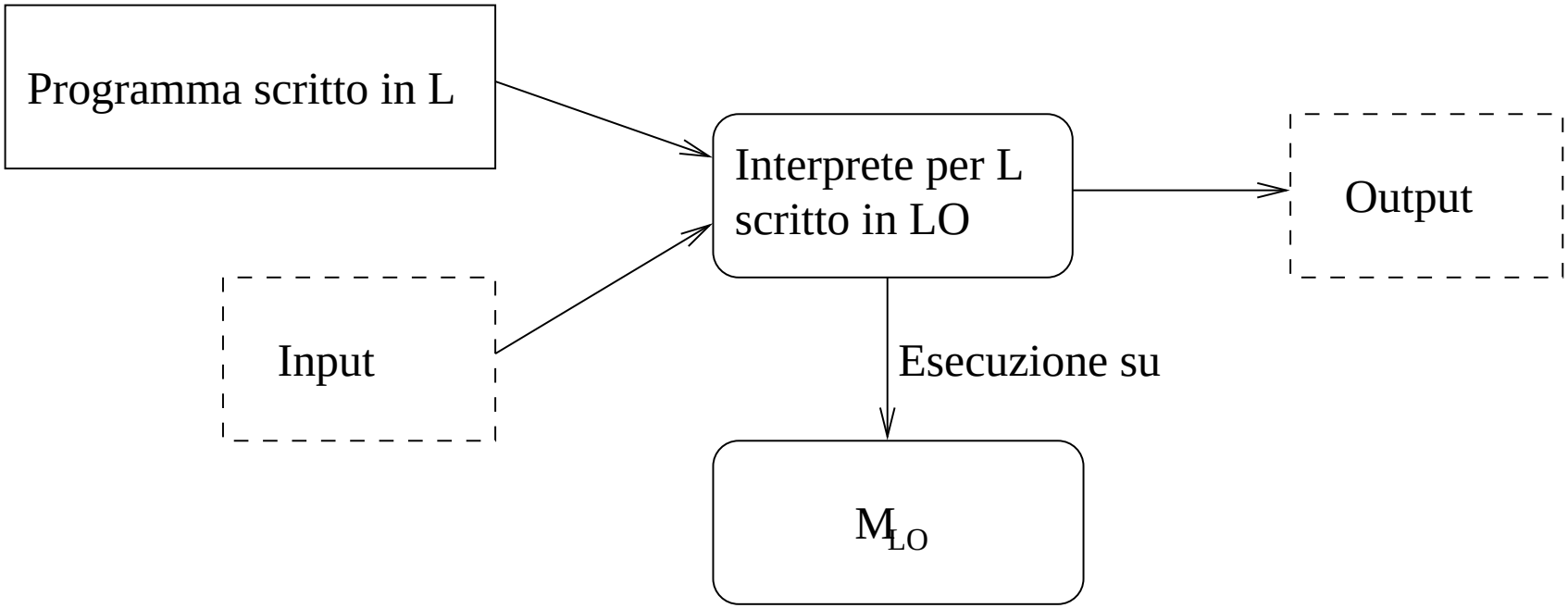
- **Compilatori**: prima traducono l'intero programma; poi la traduzione può essere eseguita (anche più volte).

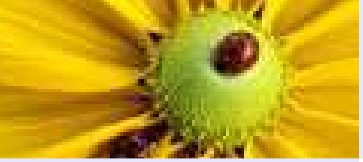
  PRO: velocità di esecuzione finale - fase di rilascio.

  PRO: più controlli e in anticipo

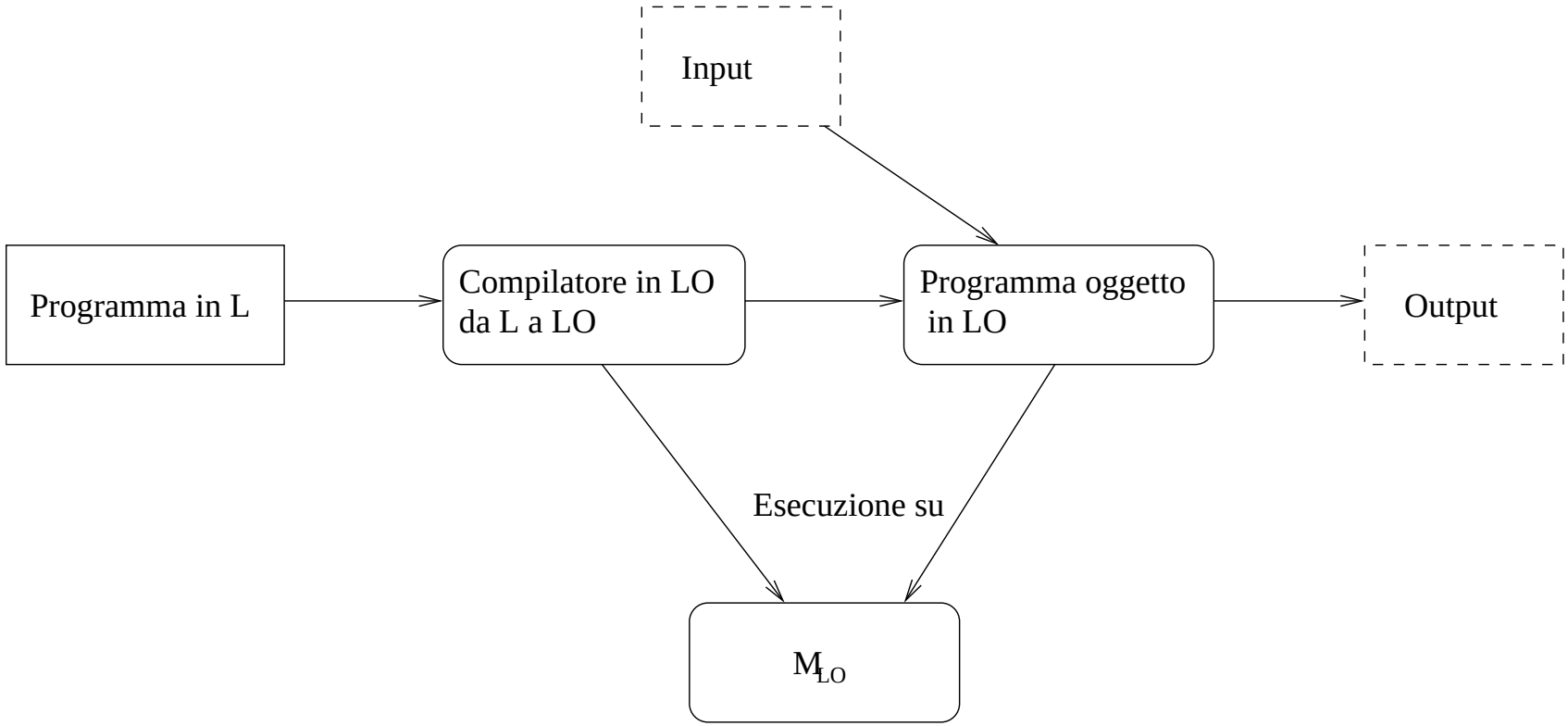


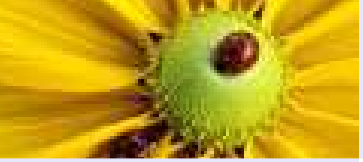
Interpretazione pura



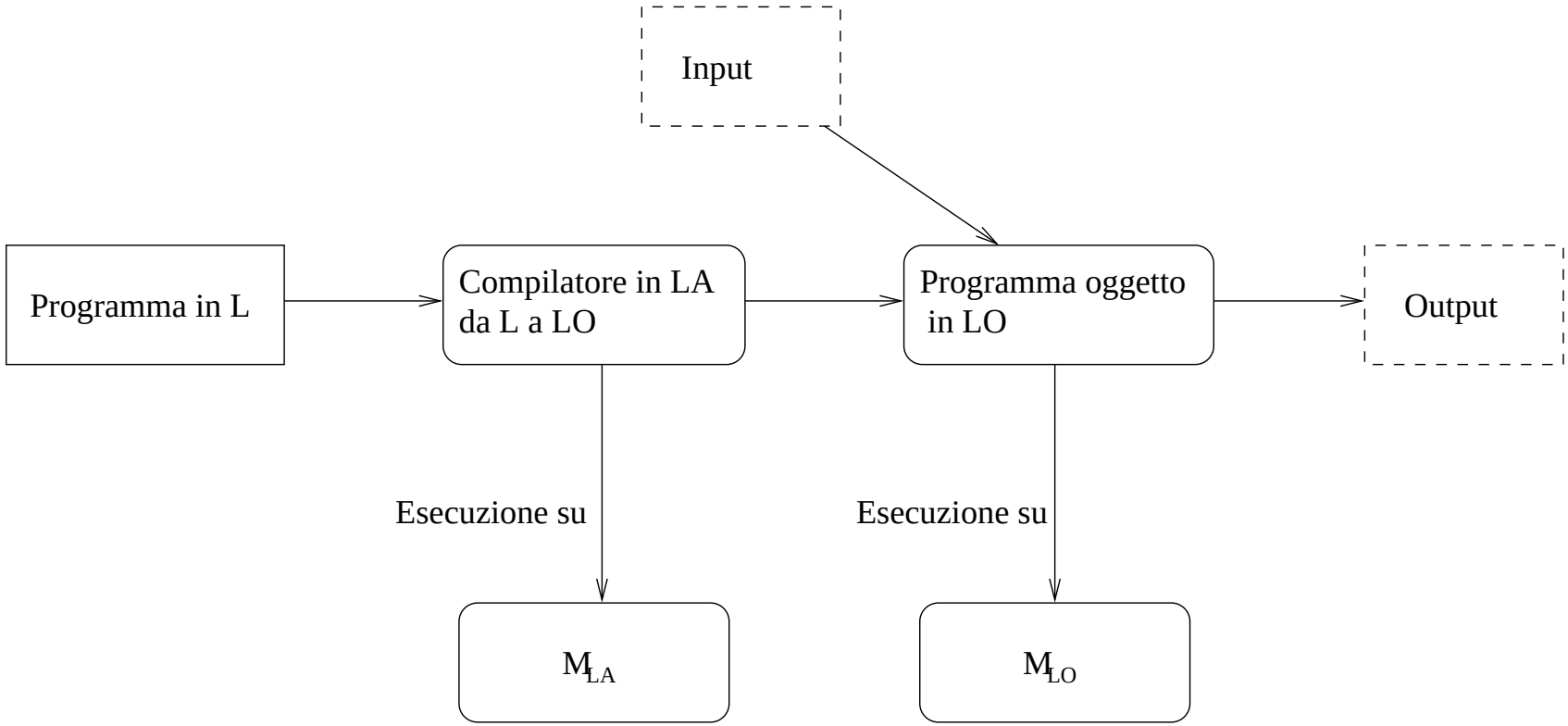


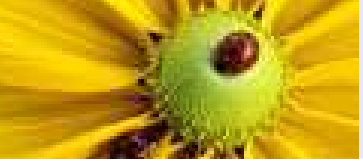
Compilazione pura (caso semplice)



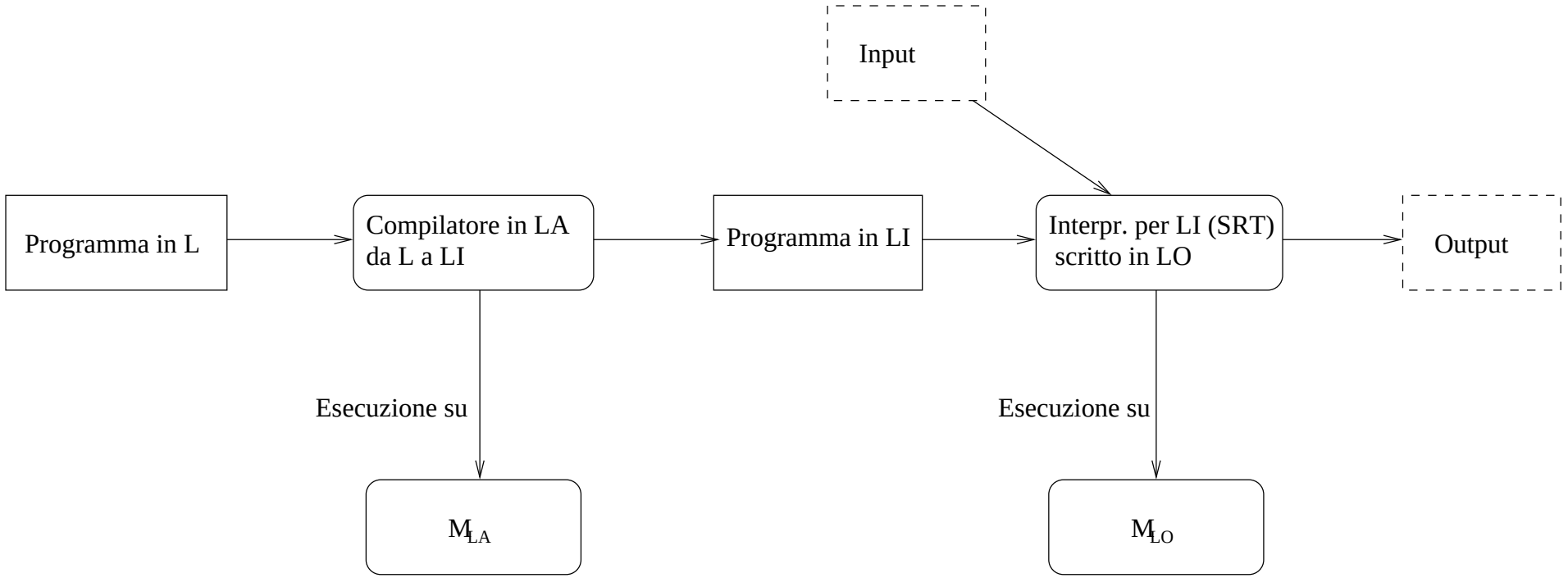


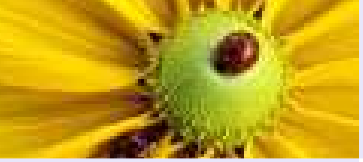
Compilazione pura (caso più generale)





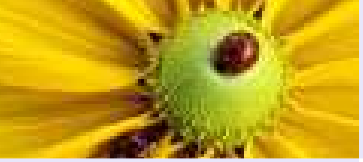
Compilazione per macchina intermedia





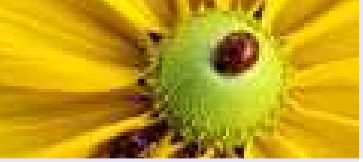
Supporto a Run Time (SRT)

- Funzionalità aggiuntive (rispetto a M_{LO})
  - Funzioni di basso livello / interfacce col S.O.; ad es. funzioni di I/O



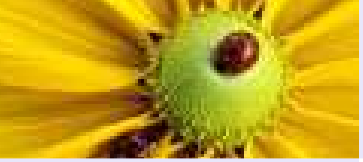
Supporto a Run Time (SRT)

- Funzionalità aggiuntive (rispetto a M_{LO})
  - Funzioni di basso livello / interfacce col S.O.; ad es. funzioni di I/O
  - Gestione della memoria; garbage collection; gestione dell'heap; gestione dello stack

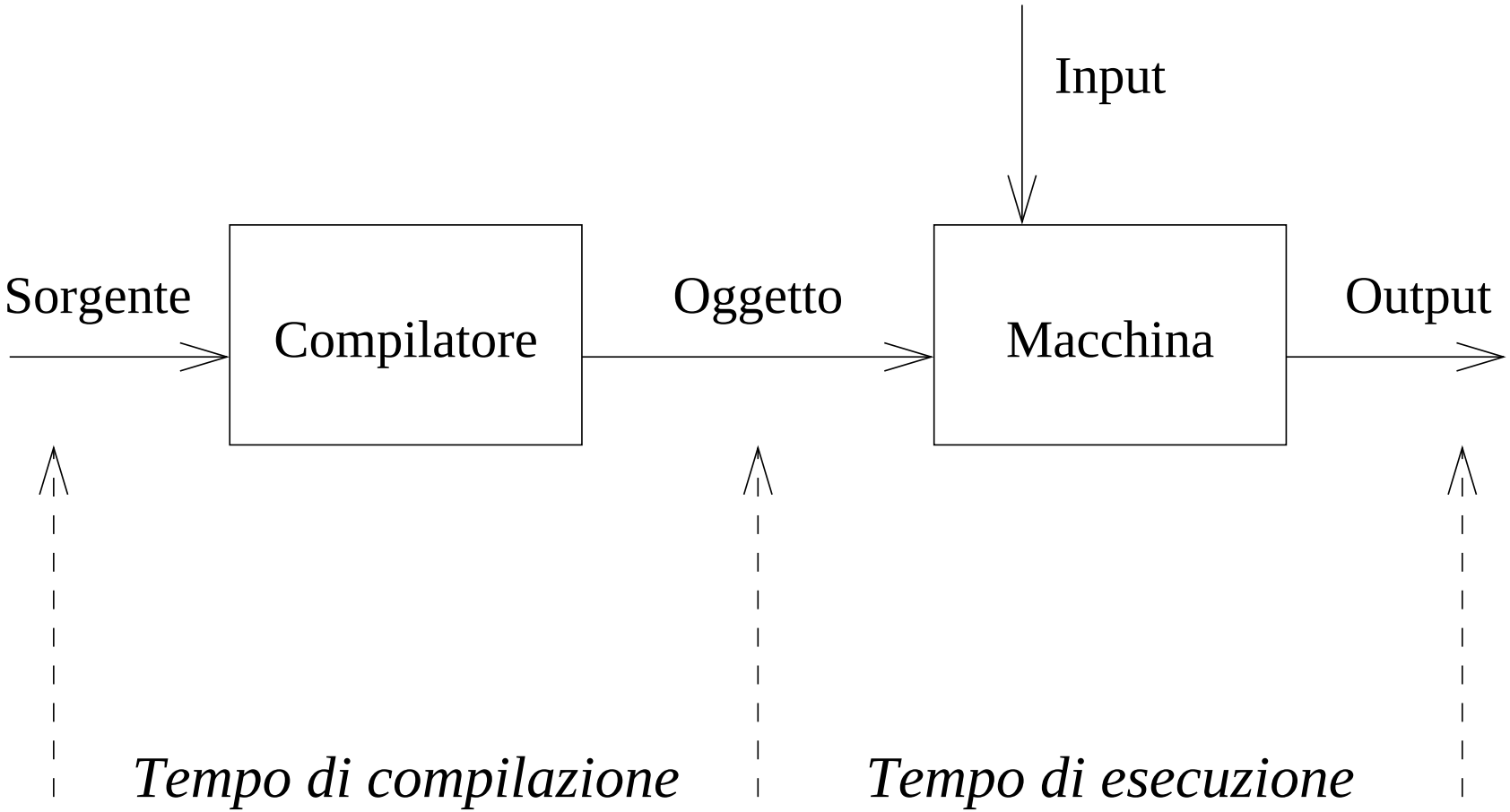


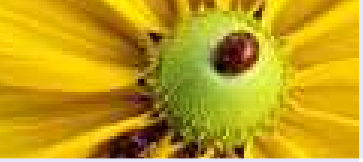
Supporto a Run Time (SRT)

- Funzionalità aggiuntive (rispetto a M_{LO})
  - Funzioni di basso livello / interfacce col S.O.; ad es. funzioni di I/O
  - Gestione della memoria; garbage collection; gestione dell'heap; gestione dello stack
- Non necessariamente una macchina astratta radicalmente diversa
  - A volte solo un pacchetto di funzioni o librerie aggiunte automaticamente al codice oggetto

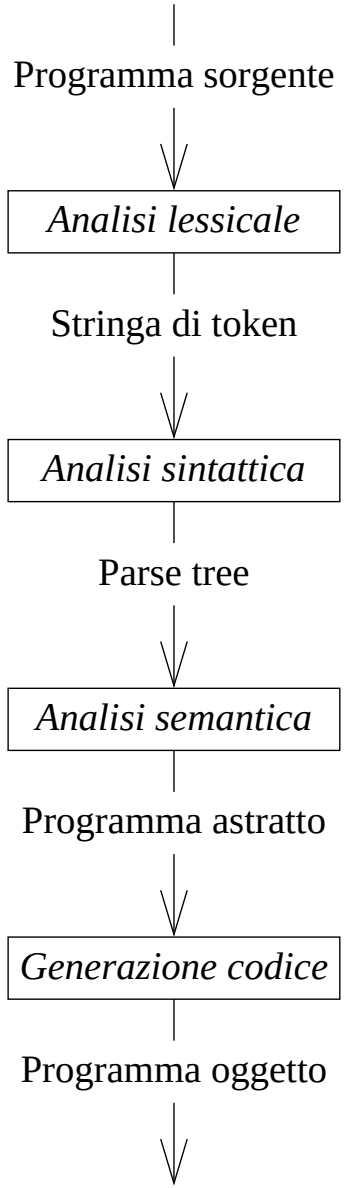


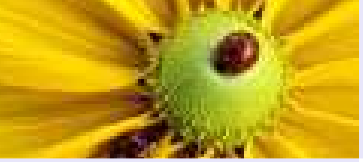
Compilazione ed esecuzione





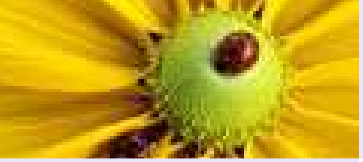
Compilatore





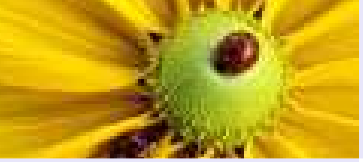
Proprietà dei linguaggi

- Semplicità – (concisione) VS (leggibilità)



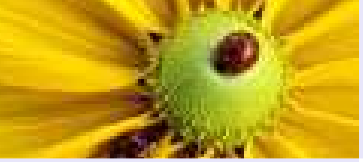
Proprietà dei linguaggi

- Semplicità – (concisione) VS (leggibilità)
  - Semantica: minimo numero di concetti e strutture.



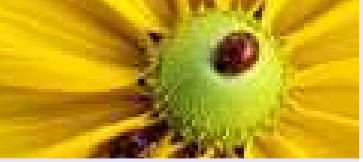
Proprietà dei linguaggi

- Semplicità – (concisione) VS (leggibilità)
  - Semantica: minimo numero di concetti e strutture.
  - Sintattica: unica rappresentabilità di ogni concetto.



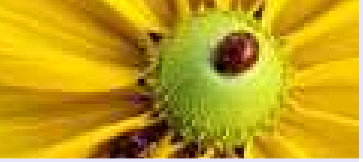
Proprietà dei linguaggi

- Semplicità – (concisione) VS (leggibilità)
  - Semantica: minimo numero di concetti e strutture.
  - Sintattica: unica rappresentabilità di ogni concetto.
- Astrazione – (rappresentare solo attributi essenziali)



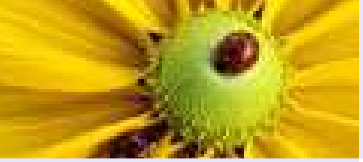
Proprietà dei linguaggi

- Semplicità – (concisione) VS (leggibilità)
  - Semantica: minimo numero di concetti e strutture.
  - Sintattica: unica rappresentabilità di ogni concetto.
- Astrazione – (rappresentare solo attributi essenziali)
  - Dati: nascondere i dettagli di oggetti.



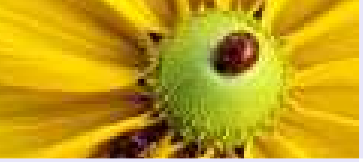
Proprietà dei linguaggi

- Semplicità – (concisione) VS (leggibilità)
  - Semantica: minimo numero di concetti e strutture.
  - Sintattica: unica rappresentabilità di ogni concetto.
- Astrazione – (rappresentare solo attributi essenziali)
  - Dati: nascondere i dettagli di oggetti.
  - Procedure: facilitare la modularità del progetto.



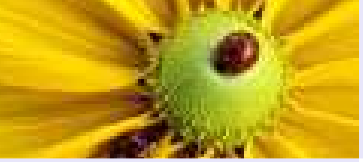
Proprietà dei linguaggi

- Semplicità – (concisione) VS (leggibilità)
  - Semantica: minimo numero di concetti e strutture.
  - Sintattica: unica rappresentabilità di ogni concetto.
- Astrazione – (rappresentare solo attributi essenziali)
  - Dati: nascondere i dettagli di oggetti.
  - Procedure: facilitare la modularità del progetto.
- Espressività – (facilità di rappresentazione di oggetti) VS (semplicità)



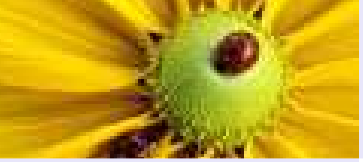
Proprietà dei linguaggi

- Semplicità – (concisione) VS (leggibilità)
  - Semantica: minimo numero di concetti e strutture.
  - Sintattica: unica rappresentabilità di ogni concetto.
- Astrazione – (rappresentare solo attributi essenziali)
  - Dati: nascondere i dettagli di oggetti.
  - Procedure: facilitare la modularità del progetto.
- Espressività – (facilità di rappresentazione di oggetti) VS (semplicità)
- Ortogonalità – (meno eccezioni alle regole del linguaggio)



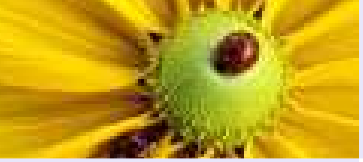
Proprietà dei linguaggi

- Semplicità – (concisione) VS (leggibilità)
  - Semantica: minimo numero di concetti e strutture.
  - Sintattica: unica rappresentabilità di ogni concetto.
- Astrazione – (rappresentare solo attributi essenziali)
  - Dati: nascondere i dettagli di oggetti.
  - Procedure: facilitare la modularità del progetto.
- Espressività – (facilità di rappresentazione di oggetti) VS (semplicità)
- Ortogonalità – (meno eccezioni alle regole del linguaggio)
- Portabilità



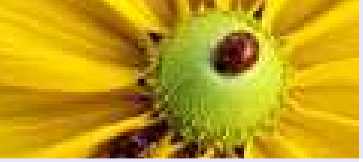
Criteri di scelta del linguaggio

- Disponibilità dei traduttori



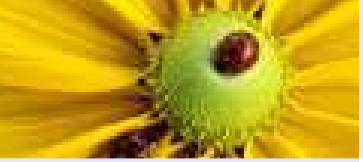
Criteri di scelta del linguaggio

- Disponibilità dei traduttori
- Maggiore conoscenza da parte del programmatore



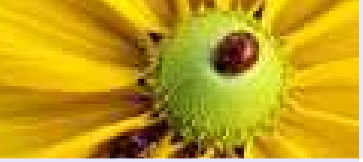
Criteri di scelta del linguaggio

- Disponibilità dei traduttori
- Maggiore conoscenza da parte del programmatore
- Esistenza di standard di portabilità



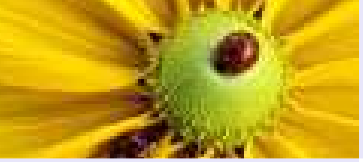
Criteri di scelta del linguaggio

- Disponibilità dei traduttori
- Maggiore conoscenza da parte del programmatore
- Esistenza di standard di portabilità
- Comodità dell'ambiente di programmazione
- Sintassi aderente al problema



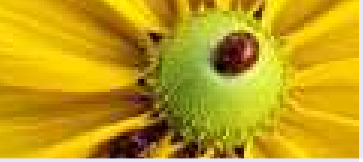
Criteri di scelta del linguaggio

- Disponibilità dei traduttori
- Maggiore conoscenza da parte del programmatore
- Esistenza di standard di portabilità
- Comodità dell'ambiente di programmazione
- Sintassi aderente al problema
- Semantica aderente alla architettura fisica

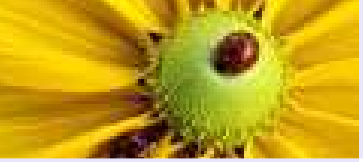


Criteri di scelta del linguaggio

- Disponibilità dei traduttori
- Maggiore conoscenza da parte del programmatore
- Esistenza di standard di portabilità
- Comodità dell'ambiente di programmazione
- Sintassi aderente al problema
- Semantica aderente alla architettura fisica

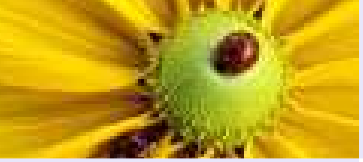


Paradigmi computazionali



Paradigmi

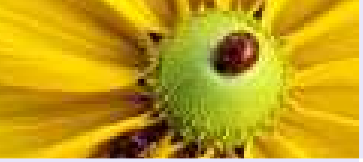
Imperativo: Un programma specifica sequenze di modifiche da apportare allo *stato della macchina* (memoria).



Paradigmi

Imperativo: Un programma specifica sequenze di modifiche da apportare allo *stato della macchina* (memoria).

Funzionale: Il programma e le sue componenti sono *funzioni*. Esecuzione come valutazione di funzioni.

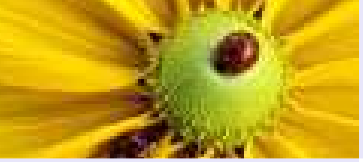


Paradigmi

Imperativo: Un programma specifica sequenze di modifiche da apportare allo *stato della macchina* (memoria).

Funzionale: Il programma e le sue componenti sono *funzioni*. Esecuzione come valutazione di funzioni.

Logico: Programma come descrizione logica di un problema. Esecuzione analoga a processi di dimostrazione di teoremi.



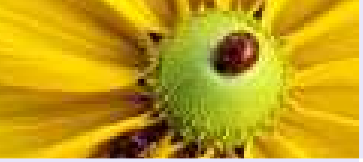
Paradigmi

Imperativo: Un programma specifica sequenze di modifiche da apportare allo *stato della macchina* (memoria).

Funzionale: Il programma e le sue componenti sono *funzioni*. Esecuzione come valutazione di funzioni.

Logico: Programma come descrizione logica di un problema. Esecuzione analoga a processi di dimostrazione di teoremi.

Orientato ad oggetti: Programma costituito da oggetti che scambiano messaggi.



Paradigmi

Imperativo: Un programma specifica sequenze di modifiche da apportare allo *stato della macchina* (memoria).

Funzionale: Il programma e le sue componenti sono *funzioni*. Esecuzione come valutazione di funzioni.

Logico: Programma come descrizione logica di un problema. Esecuzione analoga a processi di dimostrazione di teoremi.

Orientato ad oggetti: Programma costituito da oggetti che scambiano messaggi.

Parallelo: Programmi che descrivono entità distribuite che sono eseguite contemporaneamente ed in modo asincrono.

Gli ultimi due sono ortogonali rispetto ai primi tre...

Esempio 1.0 Imperativo vs. Funzionale

function definition

```
function factI (n)
  local accumulator = 1
  for i = 1,n do
    accum = accumulator*i
  end
  return accum
end
```

trace of execution

```
factI(4):
          accumulator = 1
i = 1     accumulator = 1 * 1
i = 2     accumulator = 1 * 2
i = 3     accumulator = 2 * 3
i = 4     accumulator = 6 * 4
return 24
```

function definition

```
function factR (n)
  if n == 1 then
    return 1
  else
    return n*factR(n-1)
  end
end
```

trace of execution

```
factR(4) =
4 * factR(3) =
    3 * factR(2) =
        2 * factR(1) =
                1
                1
            1
        2
    6
24
```

Notare eliminazione assegnamenti: n rimpiazza i, ricorsione invece di cicli

Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `membro(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `membro(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Es.:

```
membro(2, [1, 2, 3]) = true;
membro(4, [1, 2, 3]) = false.
```

Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `membro(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Es.:

`membro(2, [1, 2, 3]) = true;`

`membro(4, [1, 2, 3]) = false.`

A questo scopo, si suppongano già esistenti le funzioni:

- `vuota(L)`, che restituisce `true` se `L` è vuota, altrimenti `false`.

Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `membro(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Es.:

`membro(2, [1, 2, 3]) = true`;

`membro(4, [1, 2, 3]) = false`.

A questo scopo, si suppongano già esistenti le funzioni:

- `vuota(L)`, che restituisce `true` se `L` è vuota, altrimenti `false`.
- `testa(L)`, che restituisce il primo elemento della lista `L`.

  Es.: `testa([1, 2, 3]) = 1`.

Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `membro(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Es.:

`membro(2, [1, 2, 3]) = true;`

`membro(4, [1, 2, 3]) = false.`

A questo scopo, si suppongano già esistenti le funzioni:

- `vuota(L)`, che restituisce `true` se `L` è vuota, altrimenti `false`.
- `testa(L)`, che restituisce il primo elemento della lista `L`.

  Es.: `testa([1, 2, 3]) = 1`.
- `coda(L)`, che restituisce una sottolista ottenuta rimuovendo il primo elemento di `L`.

  Es.: `coda([1, 2, 3]) = [2, 3]`.

Esempio 1.2

La funzione `membro` nel paradigma imperativo:

```
procedure membro(X,L)
  local L1 = L
  while not vuota(L1) and not X=testa(L1)
    do L1 = coda(L1)
  return not vuota(L1)
```

Esempio 1.2

La funzione `membro` nel paradigma imperativo:

```
procedure membro(X,L)
  local L1 = L
  while not vuota(L1) and not X=testa(L1)
    do L1 = coda(L1)
  return not vuota(L1)
```

In C (altro linguaggio imperativo):

```
bool member(X,L) {
  List L1 = L;
  while( ! empty(L1) && ! X=testa(L1))
    L1 = coda(L1);
  return (! vuota(L1));
}
```

NB: nessuna differenza strutturale, solo dettagli sintattici

Esempio 2

La funzione `membro` nel paradigma funzionale (come Lisp):

```
function member(X,L)
  if  vuota(L)  then false
  else if  X == testa(L) then true
  else member(X, coda(L))
```

Esempio 2

La funzione `membro` nel paradigma funzionale (come Lisp):

```
function member(X,L)
  if  vuota(L)  then false
  else if  X == testa(L) then true
  else member(X, coda(L))
```

Nella versione funzionale pura non ci sono variabili, nè assegnazioni. Quindi non si possono usare cicli e bisogna rimpiazzarli con la ricorsione. La sintassi Lisp e Scheme sarebbe un po' particolare:

```
(defun membro (x l)
  (cond ((null l) l)
        ((equal x (first l)) T)
        (T (membro x (rest l)))))
```

Esempio 2

La funzione `membro` nel paradigma funzionale (come Lisp):

```
function member(X,L)
  if  vuota(L)  then false
  else if  X == testa(L) then true
  else member(X, coda(L))
```

Nella versione funzionale pura non ci sono variabili, nè assegnazioni. Quindi non si possono usare cicli e bisogna rimpiazzarli con la ricorsione. La sintassi Lisp e Scheme sarebbe un po' particolare:

```
(defun membro (x l)
  (cond ((null l) l)
        ((equal x (first l)) T)
        (T (membro x (rest l)))))
```

Anche il C si potrebbe usare in stile funzionale se evitassimo di usare i costrutti imperativi:

```
bool member(X,L) {
return (vuota(L)) ?  false :
         (X == testa(L))?  true :
            member(X, coda(L)) }
```

Esempio 3

La funzione `membro` nel paradigma logico (prolog):

```
membro(X, [X|L]).
membro(X, [Y|L]) :- membro(X, L).
```

Esempio 3

La funzione `membro` nel paradigma logico (prolog):

```
membro(X, [X|L]).
membro(X, [Y|L]) :- membro(X, L).
```

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*.

Esempio 3

La funzione `membro` nel paradigma logico (prolog):

```
membro(X, [X|L]).
membro(X, [Y|L]) :- membro(X, L).
```

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce *yes* (true)
- member(0,[1,2,3]) restituisce *no* (false)

Esempio 3

La funzione `membro` nel paradigma logico (prolog):

```
membro(X, [X|L]).
membro(X, [Y|L]) :- membro(X, L).
```

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce *yes* (true)
- member(0,[1,2,3]) restituisce *no* (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - `X=1`

Esempio 3

La funzione `membro` nel paradigma logico (prolog):

```
membro(X, [X|L]).
membro(X, [Y|L]) :- membro(X, L).
```

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce *yes* (true)
- member(0,[1,2,3]) restituisce *no* (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - `X=1; X=2`

Esempio 3

La funzione `membro` nel paradigma logico (prolog):

```
membro(X, [X|L]).
membro(X, [Y|L]) :- membro(X, L).
```

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce *yes* (true)
- member(0,[1,2,3]) restituisce *no* (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - `X=1; X=2; X=3`

Esempio 3

La funzione `membro` nel paradigma logico (prolog):

```
membro(X, [X|L]).
membro(X, [Y|L]) :- membro(X, L).
```

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce *yes* (true)
- member(0,[1,2,3]) restituisce *no* (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - `X=1; X=2; X=3; no` (si comporta come un generatore)

Esempio 3

La funzione `membro` nel paradigma logico (prolog):

```
membro(X, [X|L]).
membro(X, [Y|L]) :- membro(X, L).
```

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce *yes* (true)
- member(0,[1,2,3]) restituisce *no* (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - `X=1; X=2; X=3; no` (si comporta come un generatore)
- risposte con variabili: member(1,L) restituisce
  - `L=[1|`L_0`]`

Esempio 3

La funzione `membro` nel paradigma logico (prolog):

```
membro(X, [X|L]).
membro(X, [Y|L]) :- membro(X, L).
```

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce *yes* (true)
- member(0,[1,2,3]) restituisce *no* (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - `X=1; X=2; X=3; no` (si comporta come un generatore)
- risposte con variabili: member(1,L) restituisce
  - $\texttt{L=[1|L}_0\texttt{]; L=[Y}_0\texttt{,1|L}_1\texttt{]}$

Esempio 3

La funzione `membro` nel paradigma logico (prolog):

```
membro(X, [X|L]).
membro(X, [Y|L]) :- membro(X, L).
```

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce *yes* (true)
- member(0,[1,2,3]) restituisce *no* (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - `X=1; X=2; X=3; no` (si comporta come un generatore)
- risposte con variabili: member(1,L) restituisce
  - $\texttt{L=[1|L}_0\texttt{]; L=[Y}_0\texttt{,1|L}_1\texttt{]; L=[Y}_0\texttt{,Y}_1\texttt{,1|L}_2\texttt{]} \ldots$

Esempio 3

La funzione `membro` nel paradigma logico (prolog):

```
membro(X, [X|L]).
membro(X, [Y|L]) :- membro(X, L).
```

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce *yes* (true)
- member(0,[1,2,3]) restituisce *no* (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - `X=1; X=2; X=3; no` (si comporta come un generatore)
- risposte con variabili: member(1,L) restituisce
  - $\texttt{L=[1|L}_0\texttt{]; L=[Y}_0\texttt{,1|L}_1\texttt{]; L=[Y}_0\texttt{,Y}_1\texttt{,1|L}_2\texttt{]} \ldots$
- *Invertibilità*: nessuna distinzione tra input e output. Un solo predicato (programma), molte funzioni.

Conclusioni

- Il paradigma di appartenenza può influenzare *radicalmente* il modo in cui si risolve il problema
- Non è l'unico aspetto determinante. Altri esempi di aspetti importanti:
  - Il sistema di tipi supportato
  - Eventuale supporto alle eccezioni
  - Modello di concorrenza e sincronizzazione
  - ...