Corso di Linguaggi di Programmazione

AA 2025 2026

Fabio Sartori, Claudio Ferretti, Marco Antoniotti

Generalità

Docenti:

T1 Lez: Fabio Sartori, fabio.sartori@unimib.it

T2 Lez: Claudio Ferretti, claudio.ferretti@unimib.it

T1 Lab / T2 Lab: Marco Antoniotti, marco.antoniotti@unimib.it

Orari del corso:

T1 Lez: Lunedì 16.30-18.30 e Mercoledì 8.30-11.30

T2 Lez: Lunedì 16.30-18.30 e Venerdì 8.30-11.30

T1 Lab: Martedì 15.30-18.30, **T2 Lab:** Venerdì 15.30-18.30

Ricevimento: TDB su appuntamento

Generalità/2

- **Libro di riferimento**: Sebesta R. W. Concepts of Programming Languages. Pearson/Addison Wesley (12th Edition)
- **Edizione digitale consigliata**
- **Modalità d'esame**: prova scritta + progetto
  - due prove in itinere (compitini) durante il corso (novembre e gennaio)
  - sei appelli totali (compresi compitini)

Perchè studiare i Linguaggi di Programmazione?

La Nascita dei Linguaggi di Programmazione

- **Domanda...**chi ha progettato il primo calcolatore (computer)?

La Nascita dei Linguaggi di Programmazione

- **Domanda...**chi ha progettato il primo calcolatore (computer)?
- Immagino che potreste rispondere indicando uno di questi illustri nomi:

La Nascita dei Linguaggi di Programmazione

- **Domanda...**chi ha progettato il primo calcolatore (computer)?
- Immagino che potreste rispondere indicando uno di questi illustri nomi:

John Von Neumann

La Nascita dei Linguaggi di Programmazione

- **Domanda...**chi ha progettato il primo calcolatore (computer)?
- Immagino che potreste rispondere indicando uno di questi illustri nomi:

John Von Neumann

Alan Turing

La Nascita dei Linguaggi di Programmazione

- **Domanda...**chi ha progettato il primo calcolatore (computer)?
- Immagino che potreste rispondere indicando uno di questi illustri nomi:

John Von Neumann

Alan Turing

Claude Shannon

...E Invece...A Sorpresa...

Il Linguaggio di Programmazione Plankalkül

La generazione di calcolatori Z è programmabile in un linguaggio abbastanza sofisticato, con caratteristiche di alto livello già abbastanza elevate rispetto ai suoi successori

- **Tipi di dati**: bit, interi, float con "hidden bit"
- **Strutture dati,** array e record, incluse definizioni ricorsive
- **niente goto!,** ciclo for e un'istruzione **Fin con apice** e inserimento di break/continue per cicli annidati
- **costrutto di selezione a una via if,** senza else
- Il linguaggio permetteva di effettuare calcoli floating point, ordinamenti di array, e problemi su grafi, analisi sintattica di formule logiche
- Nella tesi di dottorato di Zuse sono riportate 49 pagine di algoritmi per il goco degli scacchi!

Un esempio di Codice

Porzione di programma in Plankalkül, che assegna A[4] + 1 ad A[5] :

```
        |   A + 1 => A
    V   |   4         5          (subscripts)
    S   |   1.n       1.n        (data types)
```

Equivalente a A[5] := A[4] + 1; **in forma tabellare:**

V	0	1	2		variabile di lavoro temporanea
A	5	4	1		variabili e costanti in uso (array A e costante numerica 1)
S	+				operatori e istruzioni

Tanti linguaggi di programmazione...perchè?

Il modello di riferimento

L'architettura di Von Neumann

Un semplice programma in linguaggio macchina (somma delle locazioni di memoria 10 e 11 nella posizione 12):

Indirizzo	Codice	Assembly MIPS	Descrizione
0000	8C08000A	lw $t0, 10($zero)	Carica da locazione 10
0002	8C09000B	lw $t1, 11($zero)	Carica da locazione 11
0004	010A5020	add $t2, $t0, $t1	Somma i valori
0006	AC0A000C	sw $t2, 12($zero)	Memorizza

```
10001100000001000000000000001010

10001100000001001000000000001011

00000001000010010101000000100000

10101100000001010000000000001100
```

Più in dettaglio

Istruzione 1: Formato I-type: [opcode][rs][rt][immediate]

```
100011 00000 01000 0000000000001010
 ↑           ↑       ↑         ↑
opcode  $zero  $t0   offset=10
(35)        (0)    (8)    (10)
```

Istruzione 4: Formato I-type: [opcode][rs][rt][imm.]

```
101011 00000 01010 0000000000001100
    ↑         ↑       ↑         ↑
 opcode  $zero  $t2   offset=12
   (43)     (0)  (10)    (12)
```

Istruzione 2: Formato I-type: [opcode][rs][rt][immediate]

```
100011 00000 01001 0000000000001011
    ↑         ↑       ↑         ↑
opcode  $zero  $t1   offset=11
  (35)      (0)    (9)    (11)
```

Istruzione 3: Formato R-type: [opcode][rs][rt][rd][shamt][funct]

```
000000 01000 01001 01010 00000 100000
    ↑         ↑       ↑         ↑       ↑     ↑
  opcode   $t0   $t1     $t2    shamt  add
     (0)     (8)   (9)     (10)    (0)    (32)
```

Lo stesso programma in Python

- **Prima versione, dettata dalla comodità**: MIPSSim.py
- **Seconda versione, dettata dalle possibilità**: MIPSSimCompleto.py

- **Volete provare in altri linguaggi?**: http://www.dangermouse.net/esoteric

Cosa ne pensa Dijkstra: il ruolo dei linguaggi di programmazione

Edsger Wybe Dijkstra

- "the art of programming is the art of organising complexity"
- "we must organise the computations in such a way that our limited powers are sufficient to guarantee that the computation will establish the desired effect."
- "program testing can be used to show the presence of bugs, but never to show their absence!"

That is...non è possibile stabilire la correttezza di un programma tramite testing, a meno che non si tenga conto anche della sua struttura interna (ma per questo vi rimando ai corsi di Ingegneria del Software e Analisi del Software :-))

Motivazione del corso

- Apprendere strumenti metodologici per scegliere i linguaggi più opportuni
- Apprendere strumenti metodologici per imparare nuovi linguaggi
- Approfondire il significato dell'implementazione
- Organizzare una visione generale dello sviluppo del calcolo

Motivazione del corso

- Apprendere strumenti metodologici per scegliere i linguaggi più opportuni
- Apprendere strumenti metodologici per imparare nuovi linguaggi
- Approfondire il significato dell'implementazione
- Organizzare una visione generale dello sviluppo del calcolo
- **Migliorare la capacità di esprimere e mettere in pratica idee**

Ovvero?

Proviamo a collegare quanto detto ai possibili domini applicativi che conosciamo:

- Calcolo scientifico, caratterizzato da **grandi numeri, operazioni floating point, massiccio uso di array (n-dimensionali).** Linguaggi adatti **Fortran, Matlab e, oggi, Python, Julia.**
- Business Application, caratterizzato dalla necessità di produrre **report, usare numeri decimali e caratteri.** Linguaggi adatti **Cobol**, **Julia**
- Intelligenza Artificiale, caratterizzata dalla manipolazione di **simboli,** piuttosto che **numeri.** Liguaggi adatti **Lisp e dialetti, Scheme**
- Programmazione di sistemi operativi, caratterizzata dalla necessità di efficienza per tutte le operazioni in background. Liguaggio adatto: **C**
- Sviluppo in ambiente di rete: caratterizzato da **markup** (**HTML e derivati),** scripting (**PHP, Javascript),** general-purpose (**Java, Android**)

Programma del Corso

- Paradigmi di programmazione, sintassi e semantica di un linguaggio, classificazione dei linguaggi
- Il paradigma logico
- Il paradigma funzionale
- evoluzione dei linguaggi dal punto di vista dei costrutti:
  - strutturazione dei dati
  - strutturazione della computazione
  - strutturazione dei programmi
- oggetti, abstraction e generic programming
- gestione della concorrenza e degli eventi

Classificazione del Linguaggi di Programmazione: Criteri

- **Leggibilità**: la facilità con cui i programmi possono essere letti e compresi
- **Facilità di sviluppo**: la facilità con cui un linguaggio può essere utilizzato per creare programmi
- **Affidabilità**: conformità alle specifiche (ovvero, funziona secondo le sue specifiche)
- **Costo**: il costo totale finale

Leggibilità: approfondimento

- **Semplicità complessiva** – Un insieme gestibile di caratteristiche e costrutti – Molteplicità minima delle funzionalità (**es incrementi in C o Java**) – Overloading minimo degli operatori (**+ in Java**)
- **Ortogonalità** – Un insieme relativamente piccolo di costrutti primitivi può essere combinato in un numero relativamente piccolo di modi, per definire le istruzioni di controllo e le strutture dati del linguaggio – Ogni combinazione possibile di primitive è **legale,** con la conseguenza di minor necessità di **controllo sintattico** - Più un linguaggio è ortogonale, minore sarà la probabilità che si presentino eccezioni - Un linguaggio sicuramente **non ortogonale** è C - un linguaggio sicuramente **ortogonale** è LISP
- **Tipi di dati** – Tipi di dati predefiniti adeguati
- **Considerazioni sintattiche** – Forma degli identificatori: composizione flessibile – Parole speciali e metodi per formare istruzioni composte (**es begin/end** o **{/}**, **if/elseif/endif**) – Forma e significato: costrutti auto-descrittivi, parole chiave significative (**es static in C**)

Facilità di sviluppo: approfondimento

- **Semplicità e ortogonalità:** Pochi costrutti, un numero ridotto di primitive, un piccolo insieme di regole per combinarle
- **Supporto per l'astrazione** – La capacità di definire e utilizzare strutture complesse o operazioni in modi che consentano di ignorare i dettagli (implementativi) (**es process abstraction per mezzo di sottoprogrammi o data abstraction per mezzo di record**)
- **Espressività** – Operatori potenti per specificare le operazioni in modo compatto (**es for invece che while in Java**) – Quantità e potenza di operatori e funzioni predefinite (**es operatori di incremento in C e Java**)

Affidabilità: approfondimento

- **Controllo dei tipi** – Test per errori di tipo
- **Gestione delle eccezioni** – Intercettare errori di esecuzione e specificare opportune misure correttive (a runtime)
- **Aliasing** – Presenza di due o più metodi di riferimento distinti per la stessa posizione di memoria (**es puntatori in C o operatore di assegnamento fra oggetti in Java**)
- **Leggibilità e facilità di sviluppo** – Un linguaggio che non supporta modi "naturali" di esprimere un algoritmo richiederà l'uso di approcci "innaturali", e quindi una ridotta affidabilità. **Factoring,** inteso come la flessibilità data dal suddividere un programma in unità separate (es **sottoprogrammi o oggetti in Java, la definizione di macro in C)** e **locality,** inteso come la restrizione dell'effetto di un'istruzione a una porzione circostritta di codice di un programma.

Costo: Approfondimento

- **Formare i programmatori all'uso del linguaggio:** semplicità e ortogonalità, esperienza del programmatore, potenza del linguaggio
- **Scrivere programmi:** dipendenza da applicazioni particolari)
- **Eseguire programmi:** esistono ambienti di sviluppo evoluti (**es IDE?**) per il linguaggio? Realizzazione/ disponibilità di compilatori/interpreti a basso costo, compilatori ottimizzanti/"just in time", necessità di macchine particolari per l'esecuzione del compilatore/programma
- **Affidabilità**: scarsa affidabilità porta a costi elevati, sistemi critici, supporto clienti
- **Mantenere programmi:** costi manutenzione enormemente maggiori dei costi di sviluppo

Riassumendo

	Criteri		
Caratteristiche	**Leggibilità**	**Sviluppo**	**Affidabilità**
Semplicità/ortogonalità	X	X	X
Strutture di controllo	X	X	X
Tipi e strutture dati	X	X	X
Design e sintassi	X	X	X
Factoring	X	X	X
Locality	X	X	X
Supporto all'astrazione		X	X
Espressività		X	X
Type checking			X
Gestione eccezioni			X
Restrizione aliasing			X

Architettura di Von Neumann e Paradigmi di Programmazione

Influenza sulla progettazione dei linguaggi

- **Architettura dei Computer** – I linguaggi vengono sviluppati attorno all'architettura informatica prevalente, nota come architettura *von Neumann*
- **Metodologie di Progettazione dei Programmi** – Le nuove metodologie di sviluppo software (ad esempio, lo sviluppo software **orientato agli oggetti**) hanno portato a nuovi paradigmi di programmazione e, di conseguenza, a nuovi linguaggi di programmazione

Influenza dell'Architettura dei Computer

- **Architettura informatica nota:** Von Neumann
- **Linguaggi imperativi,** i più dominanti, a causa dei computer von Neumann – Dati e programmi memorizzati in memoria – La memoria è separata dalla CPU – Istruzioni e dati vengono trasferiti dalla memoria alla CPU – Nei Linguaggi imperativi, di base, **le variabili sono modellate come locazioni di memoria** (celle), l'operatore di **assegnamento** definisce il trasferimento di valori tra celle di memoria, **l'iterazione è efficiente**.

Influenza dell'Architettura dei Computer

- **Architettura informatica nota:** Von Neumann
- **Linguaggi imperativi,** i più dominanti, a causa dei computer von Neumann – Dati e programmi memorizzati in memoria – La memoria è separata dalla CPU – Istruzioni e dati vengono trasferiti dalla memoria alla CPU – Nei Linguaggi imperativi, di base, **le variabili sono modellate come locazioni di memoria** (celle), l'operatore di **assegnamento** definisce il trasferimento di valori tra celle di memoria, **l'iterazione è efficiente**.
- I programmi sono governati dal ciclo **Fetch-Decode-Execute:**

```
initialize the program counter (PC)
repeat forever
  fetch the instruction pointed by
  increment the counter
  decode the instruction
  execute the instruction
end repeat
```

Paradigmi di Programmazione

- **Imperativo:** Il programma è composto da istruzioni che realizzano trasformazioni di stato. Uno stato è identificato da tutti i valori di un certo insieme di variabili ad un certo stadio dell'esecuzione. Le caratteristiche centrali sono variabili, istruzioni di assegnazione e iterazione – Include linguaggi che supportano la programmazione orientata agli oggetti – Include linguaggi di scripting – Include linguaggi visivi – Esempi: C, Java, Perl, JavaScript, Visual BASIC .NET, C++
- **Funzionale:** Un programma è un'espressione che viene valutata per ottenere un risultato. Il mezzo principale per effettuare calcoli è l'applicazione di funzioni a parametri dati (che possono essere funzioni a loro volta) – Esempi: LISP, Scheme, ML, F#, Haskell, Julia
- **Logico:** Un programma è un insieme di fatti e regole specificate senza un ordine particolare, la sua esecuzione equivale alla realizzazione di una dimostrazione – Esempio: Prolog
- **Ibrido markup/programmazione** – Linguaggi di markup estesi per supportare alcune funzionalità di programmazione – Esempi: JSTL, XSLT
- **A oggetti:** Un programma è un insieme di oggetti (astrazioni della realtà) che comunicano fra loro tramite scambio di messaggi – Esempi C++, Java, Python

Compromessi nella Progettazione dei Linguaggi

- **Affidabilità vs. costo di esecuzione** – Esempio: Java richiede che tutti i riferimenti agli elementi dell'array vengano controllati per un'indicizzazione corretta, il che porta a un aumento del costo di esecuzione
- **Leggibilità vs. sviluppo** – Esempio: APL fornisce molti operatori potenti (e un gran numero di nuovi simboli), consentendo di scrivere calcoli complessi in un programma compatto ma a costo di una scarsa leggibilità
- **Sviluppo (flessibilità) vs. affidabilità** – Esempio: I puntatori di C/C++ sono potenti e molto flessibili ma sono inaffidabili (aliasing)

Tipi di Applicazione ed Evoluzione dei Linguaggi

- **Anni '50 e primi anni '60:** Applicazioni semplici; preoccupazione per l'efficienza della macchina
- **Fine anni '60:** L'efficienza delle persone divenne importante; leggibilità, migliori strutture di controllo – programmazione strutturata – progettazione top-down e raffinamento graduale
- **Fine anni '70:** Da orientato ai processi a orientato ai dati – astrazione dei dati
- **Metà anni '80:** Programmazione orientata agli oggetti – Astrazione dei dati + ereditarietà + polimorfismo

Metodi di Implementazione

Vista stratificata del computer

- **Compilazione** – I programmi vengono tradotti in linguaggio macchina; include sistemi JIT – Uso: Grandi applicazioni commerciali
- **Interpretazione Pura** – I programmi vengono interpretati da un altro programma noto come interprete – Uso: Programmi piccoli o quando l'efficienza non è un problema
- **Sistemi di Implementazione Ibridi** – Un compromesso tra compilatori e interpreti puri – Uso: Sistemi piccoli e medi quando l'efficienza non è la preoccupazione

Confronto e Approfondimento: Compilazione

- Traduzione lenta, esecuzione veloce
- Il processo di compilazione ha diverse fasi: – **analisi lessicale**: converte i caratteri nel programma sorgente in unità lessicali – **analisi sintattica**: trasforma le unità lessicali in *alberi di parsing* che rappresentano la struttura sintattica del programma – **analisi semantica**: genera codice intermedio – **generazione del codice**: viene generato il codice macchina
- **Linking and loading**: processo di analisi e indicizzazione delle funzioni di libreria (e chiamate di sistema) necessarie e loro collegamento/caricamento a livello di programma/codice

Il Processo di Compilazione

Confronto e Approfondimento: Interpretazione

- Implementazione più semplice dei programmi (gli errori di runtime possono essere facilmente e immediatamente visualizzati)
- Esecuzione più lenta (da 10 a 100 volte più lenta dei programmi compilati)
- Spesso richiede più spazio
- Ora rara per i linguaggi tradizionali ad alto livello
- Significativa risposta con alcuni linguaggi di scripting Web (ad es., JavaScript, PHP)

Il processo di Intepretazione

Confronto e Approfondimento: Interpretazione

Sistemi di Implementazione Ibrida

- Un compromesso tra compilatori e interpreti puri
- Un programma in linguaggio ad alto livello viene tradotto in un linguaggio intermedio che permette una facile interpretazione
- Più veloce dell'interpretazione pura
- Esempi: – I programmi Perl sono parzialmente compilati per rilevare errori prima dell'interpretazione – Le implementazioni iniziali di Java erano ibride; la forma intermedia, *byte code*, fornisce portabilità a qualsiasi macchina che abbia un interprete di byte code e un sistema di runtime (insieme, questi sono chiamati *Java Virtual Machine*)

Sistemi di Implementazione Just-in-Time

- Inizialmente traducono i programmi in un linguaggio intermedio
- Poi compilano il linguaggio intermedio dei sottoprogrammi in codice macchina quando vengono chiamati
- La versione in codice macchina viene mantenuta per le chiamate successive
- I sistemi JIT sono ampiamente utilizzati per i programmi Java
- I linguaggi .NET sono implementati con un sistema JIT
- In sostanza, i sistemi JIT sono **compilatori ritardati**, evoluzione del sistema ibrido che elimina il collo di bottiglia dell'interpretazione ripetuta, compilando "al volo" solo le parti di codice effettivamente utilizzate; attenzione a non confondere con sistemi ibridi:
  - **Ibrido**: Sorgente → Linguaggio Intermedio → **Interpretazione** (ogni volta)
  - **JIT**: Sorgente → Linguaggio Intermedio → **Compilazione a Codice Macchina** (solo quando necessario) → Esecuzione nativa

Preprocessori

- Le macro del preprocessore (istruzioni) sono comunemente usate per specificare che il codice di un altro file deve essere incluso
- Un preprocessore elabora un programma immediatamente prima che il programma venga compilato per espandere le macro del preprocessore incorporate
- Un esempio ben noto: il preprocessore C – espande #include, #define, e macro simili

Ambienti di Programmazione

- Una raccolta di strumenti utilizzati nello sviluppo software
- UNIX – Un sistema operativo e raccolta di strumenti più vecchi – Oggi spesso utilizzato attraverso una GUI (ad es., CDE, KDE, o GNOME) che funziona sopra UNIX
- Microsoft Visual Studio.NET – Un ambiente visuale grande e complesso – Usato per costruire applicazioni Web e non-Web in qualsiasi linguaggio .NET
- NetBeans – Correlato a Visual Studio .NET, eccetto per applicazioni in Java

Riassumendo

- Lo studio dei linguaggi di programmazione è prezioso per diverse ragioni: – Aumenta la nostra capacità di utilizzare costrutti diversi – Ci permette di scegliere i linguaggi più intelligentemente – Rende più facile l'apprendimento di nuovi linguaggi
- I criteri più importanti per valutare i linguaggi di programmazione includono: – Leggibilità, scrivibilità, affidabilità, costo
- Le principali influenze sulla progettazione dei linguaggi sono state l'architettura delle macchine e le metodologie di sviluppo software
- I principali metodi di implementazione dei linguaggi di programmazione sono: compilazione, interpretazione pura, e implementazione ibrida