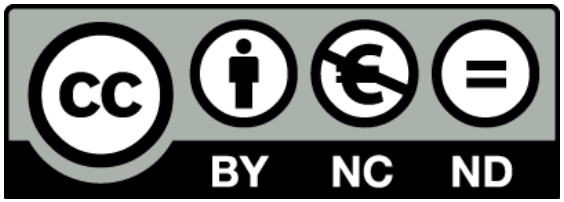
Laboratorio di Programmazione di Sistema

Introduzione al Laboratorio di Programmazione di Sistema

Luca Forlizzi, Ph.D.

Versione 20.2



Luca Forlizzi, 2020

Cosa si studia in LPS ?

- La risposta molto approssimativa (e con un errore) che spesso viene data:
  Il C e l'assembler
- Il termine *assembler* indica un traduttore non una tipologia di linguaggi
- Il termine corretto è *assembly*
- In questa lezione cerchiamo di rispondere in maniera più precisa

Cosa si studia in LPS ?

- L'obiettivo principale di LPS è l'acquisizione di:
  Conoscenza, comprensione e capacità di operare con i sistemi di base per la programmazione
- Le conoscenze trasmesse in LPS, in particolare quelle relative alle tecnologie utilizzate, *non* sono indispensabili per programmare un computer o per svolgere un'attività lavorativa nel campo dell'informatica
- Al giorno d'oggi, molti linguaggi e strumenti per la programmazione possono essere usati con profitto senza avere tali conoscenze
- In un'ottica utilitaristica dello studio (studiare tecnologie usate in ambito lavorativo), si tratta di conoscenze anacronistiche

Cosa si studia in LPS ?

- Vi sono invece ragioni molto importanti per cui è opportuno conoscere in modo approfondito i sistemi di base per la programmazione
  1. rende evidente il fatto che i modelli di calcolo dell'informatica teorica sono aderenti al reale funzionamento dei computer
  2. permette di diventare programmatori più efficienti e competenti, sopratutto in relazione a
     - tecniche di debugging
     - prestazioni
     - portabilità
     - robustezza
- L'auspicio è che al termine delle lezioni voi siate convinti di aver acquisito conoscenze e capacità che sono da un lato interessanti e dall'altro utili ai fini del prosieguo degli studi e dell'attività lavorativa

Dispositivi, Interfacce, Architetture

- Chiamiamo *dispositivo* una qualunque entità che fornisce dei servizi rispondendo a dei comandi
- Un dispositivo che può essere combinato con altri, per realizzare ulteriori servizi, viene detto *modulo*
- È possibile usufruire dei servizi di un dispositivo anche se non si conosce in quale modo il dispositivo riesca a fornirli
- Esempio: un oggetto Java per la gestione di un calendario
  - Permette di scrivere e leggere la data corrente
  - Permette di visualizzare le date precedenti e successive rispetto alla data corrente
  - Permette di visualizzare un calendario annuale, mensile, settimanale

Dispositivi, Interfacce, Architetture

- L'insieme di comandi che permettono di utilizzare i servizi di un dispositivo, viene detto *interfaccia* del dispositivo
- La descrizione di sintassi e semantica dell'interfaccia viene detta *architettura* del dispositivo
- L'architettura può non essere una descrizione della reale struttura o del funzionamento interno del dispositivo
- L'architettura è una descrizione astratta, ovvero essa descrive
  - Tutto ciò che è necessario conoscere per usare il dispositivo
  - Solo ciò che è necessario conoscere per usare il dispositivo

Dispositivi, Interfacce, Architetture

- Nell'esempio dell'oggetto Java per la gestione di un calendario
  - L'interfaccia è costituita dai metodi che realizzano i servizi
  - L'architettura comprende la nozione che l'oggetto memorizza, in quale modo, la data corrente e descrive come usare i metodi
  - L'architettura non dice esattamente in che modo l'oggetto memorizza la data
    - Ad esempio la data potrebbe essere memorizzata in una singola variabile di tipo `Date`, oppure in un insieme di `int`, o in una variabile di tipo `String`
- Se A è l'architettura di un dispositivo D, diciamo anche che D è una *implementazione* di A
- Un'implementazione può dirsi *corretta* se i comandi dell'interfaccia producono su di essa gli effetti descritti dall'architettura

Compatibilità e Famiglie

- Possono esistere dispositivi diversi che hanno interfacce uguali e che si comportano allo stesso modo in risposta ai comandi
- Ovvero possono esistere diverse implementazioni della stessa architettura
- Due implementazioni della stessa architettura possono essere molto diverse tra loro, ma, se entrambe sono corrette, i comandi dell'interfaccia producono su ciascuna di esse i medesimi effetti
- Due (o più) implementazioni di una stessa architettura, pur essendo diverse tra loro vengono quindi usate allo stesso modo, e per questo vengono dette *compatibili (tra loro)*

Compatibilità e Famiglie

- A volte vi sono dispositivi simili tra di loro, che hanno interfacce e architetture per buona parte identiche, sia sintatticamente che semanticamente, e poche differenze in alcuni dei servizi
- Dispositivi simili possono essere definiti
  - contestualmente, quando si vogliono differenziare i servizi offerti agli utenti
  - nel corso del tempo, come conseguenza "naturale" dell'evoluzione tecnologica

Compatibilità e Famiglie

- Ad esempio pensiamo a modelli di automobile
- Un'automobile può essere messa in produzione in una versione base e una versione più accessoriata
- In corrispondenza vi sono due architetture, che si differenziano per i comandi relativi agli accessori che cambiano tra una versione e l'altra dell'automobile
- Poi, l'anno seguente, la linea di automobili viene aggiornata, aggiungendo nuovi accessori e introducendo una terza versione più adatta ai terreni accidentati; in corrispondenza ci saranno 3 nuove architetture, simili tra di loro e anche a quelle definite l'anno precedente

Compatibilità e Famiglie

- Un insieme di architetture simili tra loro sia nella sintassi che nella semantica delle relative interfacce viene detto *famiglia di architetture*
- Se F è il nome di una famiglia di architetture, ciascuna architettura della famiglia è una *versione* di F
- In modo analogo, una *famiglia di interfacce I* è un insieme di interfacce simili tra loro, e ciascuna delle interfacce della famiglia viene chiamata *versione di I*

Compatibilità e Famiglie

- Poiché due architetture (o interfacce) appartenenti alla stessa famiglia sono simili, molti comandi e molte sequenze di comandi producono i medesimi effetti
- Si dice pertanto che le due architetture (o interfacce) sono *parzialmente compatibili*
- Analogamente, le implementazioni di un'architettura sono *parzialmente compatibili* con quelle di un altra architettura che appartiene alla stessa famiglia della prima

Compatibilità e Famiglie

- Un'architettura A_{new} appartenente ad una famiglia F si dice *backward compatible* con una versione A_{old} meno recente di F, se A_{new} accetta ed esegue, con cambiamenti nulli o minimi, sequenze di comandi valide per A_{old}
- La definizione si estende in modo analogo alle interfacce e ai dispositivi
- Concretamente, vedremo, la *backward compatibility* è la proprietà che consente a un calcolatore, sistema operativo o linguaggio di poter riutilizzare con cambiamenti nulli o minimi software prodotto nel passato
- Si tratta di una proprietà desiderabile per motivi economici, ma che costituisce un limite alle possibilità di migliorare un prodotto e un freno all'innovazione

Interfaccia e Architettura come Contratto

- L'interfaccia e l'architettura possono essere viste come un contratto tra gli utenti e i realizzatori di un dispositivo
  - Gli utenti si impegnano ad usare il dispositivo solo attraverso l'interfaccia, secondo quanto specificato dall'architettura
  - Gli implementatori si impegnano a realizzare un dispositivo che si comporta come descritto dall'architettura
- Se rispettato, il contratto porta vantaggi a entrambe le parti
  - Gli utenti hanno il vantaggio di poter usare allo stesso modo tutte le implementazioni di un'architettura
  - Il vantaggio per gli implementatori è che qualunque utente che conosca l'architettura, sa come utilizzare il dispositivo

Interfaccia e Architettura come Contratto

- Dispositivo, interfaccia, architettura e implementazione sono tra i concetti fondamentali delle discipline scientifiche e, sopratutto, ingegneristiche
- Nell'informatica giocano un ruolo chiave in molti settori
- Alcuni esempi
  - programmazione
  - ingegneria del software
  - progettazione dei computer

Dispositivi di Calcolo, Linguaggi, Architetture

- Un *dispositivo di calcolo* è un tipo di dispositivo che ha come interfaccia un linguaggio di programmazione
- L'architettura di un dispositivo di calcolo è quindi la descrizione delle regole sintattiche e semantiche di un linguaggio di programmazione, e viene anche chiamata *programming model* o *modello computazionale*
- Quindi, specializzando la definizione data in precedenza, se A è il programming model di un dispositivo D, diciamo anche che D è una *implementazione* di A

Dispositivi di Calcolo, Linguaggi, Architetture

- L'architettura A di un dispositivo di calcolo D, per descrivere la sintassi e la semantica del linguaggio di programmazione L del dispositivo, definisce (a volte in modo esplicito, altre in modo implicito) un *dispositivo di calcolo astratto* in grado di eseguire i programmi scritti in L, chiamato *abstract machine* di A
- La abstract machine di A è quindi un modello astratto di D, che tiene conto solo degli aspetti di D che sono rilevanti per poterne utilizzare i servizi tramite L
- Estendiamo il significato di implementazione: se un dispositivo D è una *implementazione* di un'architettura A, diciamo anche che D è una *implementazione* della abstract machine di A

Dispositivi di Calcolo, Linguaggi, Architetture

- Un linguaggio L e il relativo programming model A possono essere visti come un contratto tra i programmatori e i realizzatori delle implementazioni di A
  - I programmatori si impegnano a seguire le regole di L
  - Gli implementatori si impegnano a realizzare un dispositivo che si comporta come la abstract machine di A
- Se rispettato, il contratto porta vantaggi a entrambe le parti
  - Per i programmatori il vantaggio è che i loro programmi funzionano correttamente su tutte le implementazioni della abstract machine di A
  - Per gli implementatori il vantaggio è che la loro implementazione è in grado di eseguire qualunque programma che rispetti le regole di L

Organizzazione Strutturata

- Definiamo un *modello concettuale* che rappresenta il modo in cui pensiamo ad un computer in LPS
- In relazione al modo in cui si usano, ci sono molti modi di pensare e descrivere un computer
- In LPS descriviamo i computer dal punto di vista di un particolare tipo di utente: il *programmatore*
- Un programmatore usa un computer mediante un linguaggio di programmazione, quindi il suo modello concettuale di computer è l'architettura del computer

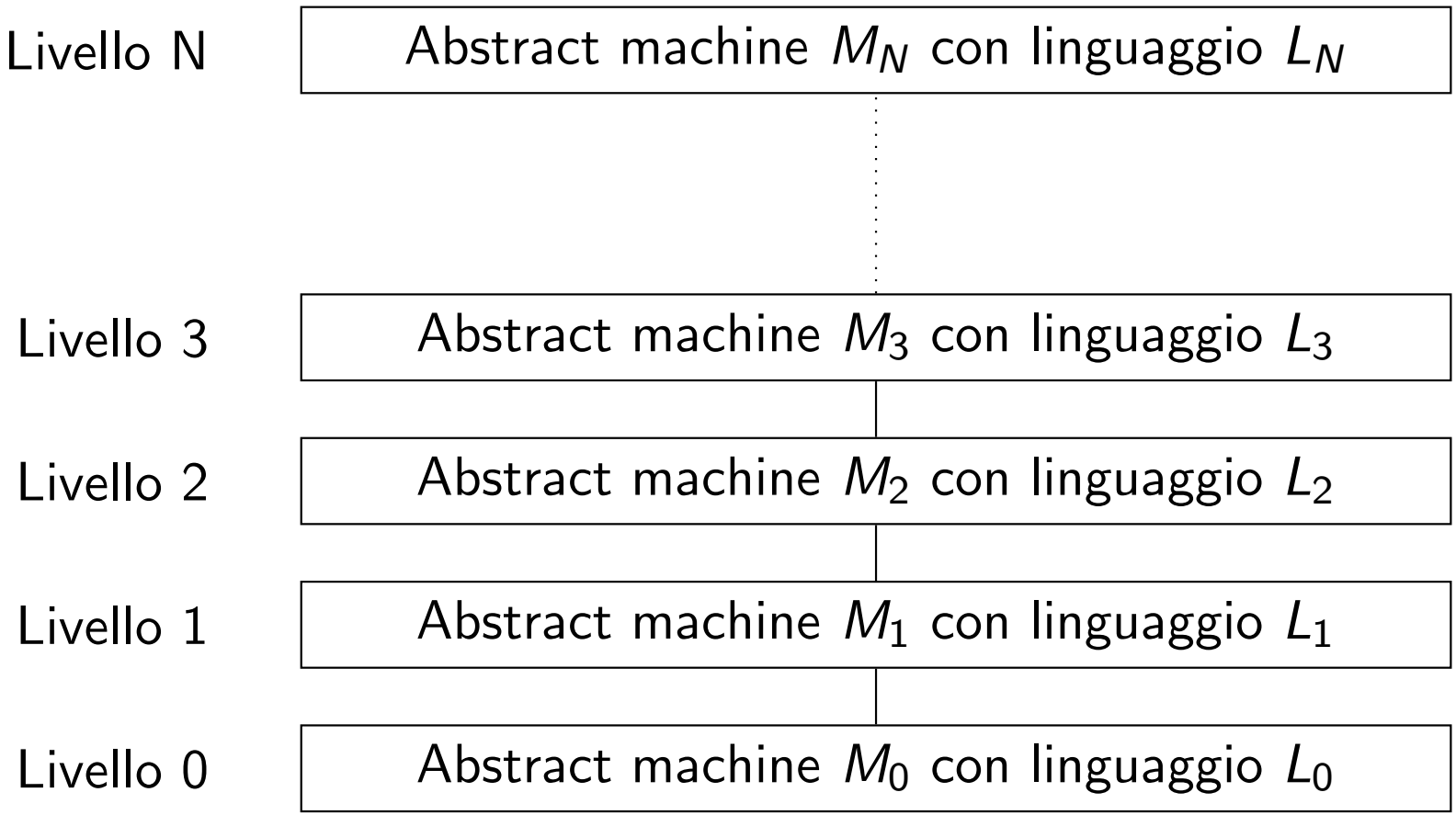
Organizzazione Strutturata

- La abstract machine di un computer è un sistema complesso che può essere visto come la composizione parti più semplici, ciascuna delle quali è una abstract machine
- Più precisamente, la abstract machine di un computer è composta da una successione di dispositivi di calcolo con potenza e complessità crescenti
- A partire da alcune abstract machine molto semplici, se ne costruiscono altre sempre più potenti e complesse

Organizzazione Strutturata

- Vediamo quindi un computer come un dispositivo di calcolo multi-livello
- Ogni *livello* è costituito da una o più abstract machine che offrono servizi ai livelli superiori
- I livelli vengono numerati a partire da 0, che indica il livello delle abstract machine più semplici
- L'interfaccia offerta da una abstract machine M_i di livello i, è un linguaggio di programmazione L_i per M_i
- Procedendo dal livello 0 verso i livelli più alti
  - Cresce la potenza e la complessità dei servizi offerti
  - I concetti definiti dal programming model e i comandi forniti dal linguaggio di programmazione sono più simili al modo di pensare ed esprimersi degli umani

Organizzazione Strutturata



Organizzazione Strutturata

- Una abstract machine di livello 0 viene implementata direttamente
- Una abstract machine di livello $i > 0$ viene implementata combinando implementazioni di abstract machine dei livelli sottostanti, ovvero utilizzando uno o più linguaggi L_h, dove $h < i$
- Di norma un livello i è una "black box", ovvero un'architettura di livello i non descrive se e come la corrispondente abstract machine venga implementata mediante implementazioni di abstract machine di livelli sottostanti
- Il linguaggio L_i e la abstract machine M_i sono quindi completamente diversi da linguaggi e dispositivi dei livelli inferiori mediante i quali vengono implementati

Organizzazione Strutturata

- Quando un programmatore utilizza un determinato linguaggio di programmazione per descrivere un algoritmo, pensa in termini di concetti e costrutti del linguaggio
- Ad esempio, per ripetere 10 volte una certa operazione in linguaggio Java, si può utilizzare un'istruzione `for`
- Non pensa a come *davvero* tale istruzione verrà eseguita da un computer reale, ovvero non pensa ai livelli diversi da quello del linguaggio in uso

Organizzazione Strutturata

- In generale, un programmatore che scrive un programma in un linguaggio L_i, lo fa immaginando che la abstract machine M_i esegua direttamente il suo programma
- Ovvero, *astrae* (cioè trascura deliberatamente) il fatto che in realtà il programma da lui scritto viene eseguito da abstract machine di livello inferiore, dopo essere stato opportunamente trasformato
- Per questo motivo, i livelli che formano il modello di computer di riferimento vengono detti *livelli di astrazione* e i dispositivi vengono detti abstract machine

Organizzazione Strutturata

- L'operazione mentale di astrazione che il programmatore mette in atto, gli consente di scrivere software in maniera più semplice, più rapida e più efficace
- Ma, in determinate situazioni, "rompere" l'astrazione, ovvero andare oltre l'astrazione e pensare a come davvero il software viene eseguito può portare notevoli vantaggi, ad esempio in relazione a
  - miglioramento delle prestazioni
  - ricerca delle cause di malfunzionamenti
- Un programmatore che è in grado, quando opportuno, di rompere l'astrazione, riesce a produrre software di qualità superiore

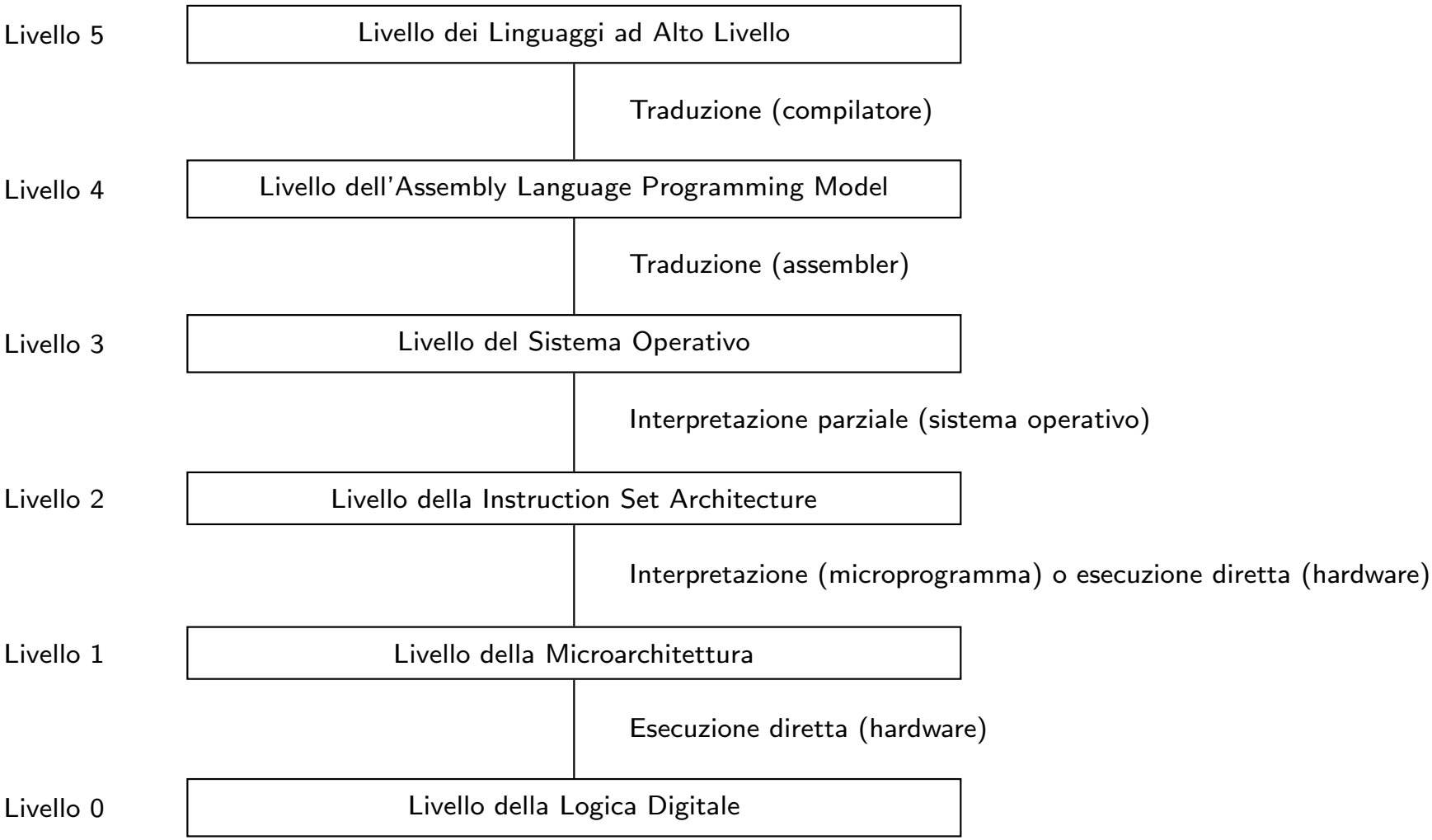
Traduzione e Interpretazione di Programmi

- Per eseguire un programma P_i, scritto in un linguaggio L_i di livello i, mediante una abstract machine di livello j la cui interfaccia è il linguaggio L_j, con $j < i$, vi sono 2 strategie fondamentali
  - Traduzione: è una strategia che prevede due macro-fasi successive
    1. costruire un programma P_j in L_j che produce lo stesso effetto di P_i
    2. eseguire P_j mediante M_j
  - Interpretazione: utilizzare un programma in L_j, detto *interprete* che prende in input P_i e simula l'architettura di L_i
    - L'interprete prende in esame, una alla volta le istruzioni di P_i, a partire da quella iniziale
    - Per ciascuna di esse esegue una sequenza di istruzioni di L_j che producono l'effetto dell'istruzione nella simulazione dell'architettura di L_i

Traduzione e Interpretazione di Programmi

- In un computer multi-livello, le abstract machine M_0 di livello 0 sono in grado di eseguire direttamente i programmi scritti in L_0
- L'esecuzione di un programma P_i scritto in un linguaggio L_i di livello i, con $i > 0$, avviene attraverso una successione di operazioni di traduzione o interpretazione
  - Si sceglie un livello $h < i$ e, detto L_h il linguaggio di livello h:
    - si traduce P_i in un programma equivalente scritto in L_h
    - oppure si interpreta P_i mediante un interprete scritto in L_h
  - Se $h > 0$, si ripete per il programma in L_h si ripete quanto fatto per quello scritto in L_i
- Dopo alcune traduzioni o interpretazioni, si giunge ad eseguire in M_0 un programma equivalente a quello scritto usando L_i

Modello a 6 livelli



Modello a 6 livelli

- Il livello 0 è chiamato *Livello della Logica Digitale*
- Le abstract machine del livello 0 sono le *porte logiche*, che vengono connesse le une alle altre; le porte logiche e le loro interconnessioni sono implementate come hardware
- I linguaggi di livello 0 permettono di indicare come interconnettere porte logiche
- Le architetture di livello 0 sono tabelle di verità che definiscono la semantica di porte logiche interconnesse
- Esempi di linguaggi di livello 0
  - Algebra booleana
  - Diagrammi di circuiti
  - Netlist
  - Automi a stati finiti
  - Verilog
  - VHDL

Modello a 6 livelli

- Le abstract machine del livello 1, detto *Livello della Microarchitettura*, sono dispositivi in grado di memorizzare dati elementari ed effettuare semplici operazioni su di essi; di norma tali dispositivi vengono implementati come hardware, interconnettendo dispositivi di livello 0
- Esempi di abstract machine di livello 1 sono bit, registri, multiplexer, reti combinatorie per operazioni aritmetiche
- I linguaggi di livello 1 permettono di connettere abstract machine di livello 1 e di livello 0, specificando in che modo esse comunicano tra loro
- Le architetture di livello 1 descrivono come utilizzare e combinare abstract machine di livello 1
- Esempi di linguaggi di livello 1
  - linguaggi di livello 0
  - Microcodice

Modello a 6 livelli

- Le abstract machine di livello 2, dette *Processing Unit*, sono in grado di eseguire programmi formati da sequenze di istruzioni che memorizzano ed elaborano dati, per effettuare generiche computazioni
- Tra le abstract machine di livello 2, ci concentriamo in prevalenza sulle *Central Processing Unit* (CPU) presenti nei computer di utilizzo generale
- Le abstract machine di livello 2 sono di solito implementate come hardware

Modello a 6 livelli

- Un linguaggio di livello 2 è l'insieme dei comandi che una abstract machine di livello 2 può eseguire
- Un'architettura di livello 2 descrive quindi il funzionamento di una abstract machine di livello 2 che esegue comandi di un linguaggio di livello 2
- Le architetture di livello 2 sono chiamate *Instruction Set Architecture* (*ISA*)
- Il livello 2 viene chiamato *Livello della ISA*
- I comandi di un linguaggio di livello 2 sono implementati come software

Modello a 6 livelli

- Dunque al livello 2 troviamo abstract machine implementate come hardware che vengono programmate mediante un linguaggio i cui comandi sono implementati come software
- Quindi un linguaggio per una abstract machine di livello 2, ovvero un'interfaccia di livello 2, è un'interfaccia tra hardware e software
- Di solito, le abstract machine di livello 2 sono quelle a più alto livello ad essere implementate completamente come hardware
- Per questo motivo, essendo uso comune denotare l'hardware con il termine "macchina", i linguaggi di livello 2 vengono chiamati *linguaggi macchina* (*LM*)

Modello a 6 livelli

- Il livello 3, chiamato *Livello del Sistema Operativo*, è un livello ibrido, in quanto tra i servizi che esso offre troviamo anche gran parte dei servizi forniti dal livello 2
- Il Livello del Sistema Operativo aggiunge concetti e servizi a quelli presenti al livello 2
- In altre parole, il livello 3, diversamente dagli altri, non è una "black box" ma "espone" anche il livello 2
- Una abstract machine M_3 di livello 3 è composta da
  - una abstract machine M_2^* per i servizi base che è uguale o molto simile alla abstract machine del livello 2
  - del software per M_2^*, detto *Sistema Operativo* che realizza i servizi aggiuntivi offerti da M_3

Modello a 6 livelli

- Il linguaggio L_3 di livello 3 per M_3 è formato quindi da due componenti
  1. il linguaggio LM^* di M_2^*, uguale o molto simile al LM della abstract machine di livello 2
  2. i comandi, detti *system call*, che consentono di eseguire programmi che fanno parte del sistema operativo
- In modo simile, l'architettura di una M_3 è formata da due componenti
  1. l'architettura ISA^* di M_2^*, uguale o molto simile alla ISA della abstract machine di livello 2
  2. la descrizione dei servizi aggiunti dal sistema operativo, chiamata *operating system architecture*

Modello a 6 livelli

- Al livello 4 troviamo abstract machine, linguaggi e architetture che, pur essendo più astratte di quelli di livello 3, sono molto simili a questi ultimi
- Infatti, nella maggior parte dei casi, ogni linguaggio di livello 4 è legato ad uno specifico linguaggio di livello 3 e, in modo analogo, ogni architettura di livello 4 è legata ad una specifica architettura di livello 3
- Le abstract machine di livello 4 sono di solito implementate come software

Modello a 6 livelli

- Un linguaggio L_4 di livello 4 è a sua volta formato da 2 componenti, ciascuna delle quali corrisponde, a grandi linee a una delle componenti di un linguaggio L_3 di livello 3
  1. il *linguaggio assembly* (*ASM*) che corrisponde alla componente LM^* del L_3
  2. la *assembly language Application Program Interface* (*ASM-API*) che corrisponde alla componente *system call* del L_3
- Poiché la componente LM^* di un L_3 è molto simile ad un *LM*, per transitività a ciascun *ASM* corrisponde un *LM*
- Un *ASM* e il corrispondente *LM* hanno una semantica sostanzialmente equivalente
- In alcuni casi c'è una perfetta corrispondenza biunivoca tra istruzioni *LM* e istruzioni *ASM*
- Negli altri casi, c'è una corrispondenza "quasi" biunivoca

Modello a 6 livelli

- Un *ASM* e il corrispondente *LM* differiscono invece in modo notevole nella sintassi
  - le istruzioni di un *ASM* sono formate da una o più stringhe di caratteri alfanumerici e speciali
  - le istruzioni di un *LM* sono *stringhe binarie*, ovvero sequenze di cifre binarie
- Semplificando un po', possiamo dire che le istruzioni di un *ASM* sono quelle di un *LM* "rivestite" da una sintassi più comoda e semplice da usare per i programmatori umani
- Poiché la semantica degli *ASM* è simile (in molti casi identica) a quella degli *LM*, gli *ASM* vengono detti *linguaggi a basso livello* (di astrazione)

Modello a 6 livelli

- In modo analogo le architetture di livello 4 sono formate da 2 componenti, ciascuna delle quali corrisponde, a grandi linee, a una delle componenti di un'architettura di livello 3
  1. l'*Assembly Language Programming Model* (*ASM-PM*) che corrisponde a ISA^*
  2. una descrizione del funzionamento di *ASM-API* che corrisponde alla *operating system architecture*
- Poiché ISA^* è a sua volta molto simile ad una *ISA*, per transitività a ciascun *ASM-PM* corrisponde una *ISA*

Modello a 6 livelli

- Al livello 5, o a livelli ancora superiori, troviamo una moltitudine di linguaggi di programmazione, con le relative abstract machine e architetture
- Rispetto a quelli dei livelli inferiori, i linguaggi e le architetture di livello 5 presentano strutture e concetti molto più simili al modo di esprimersi, pensare e operare degli umani
- Per questo si è soliti chiamare i linguaggi di livello 5, o dei livelli superiori, *linguaggi ad alto livello* (di astrazione), abbreviato in *HLL* (dall'inglese *high level language*)
- Le abstract machine di livello 5, o dei livelli superiori, sono quasi sempre implementate come software

Modello a 6 livelli

- Tra gli *HLL*, in LPS interessano in modo particolare i *linguaggi ad alto livello imperativi*
- Una abstract machine che esegue un linguaggio imperativo possiede uno *stato*, ovvero una memoria modificabile che può contenere dati
- Un programma scritto in un linguaggio imperativo consiste essenzialmente in una successione di modifiche dello stato
- Un linguaggio imperativo fornisce un insieme di costrutti che consentono di modificare lo stato della abstract machine e di strutturare una successione di modifiche dello stato

Modello a 6 livelli

- Le abstract machine di linguaggi ad alto livello imperativi sono più simili al modello di Von Neumann rispetto a quelle di altri linguaggi (es. HTML) proprio per la presenza di uno stato
  - Un programma è un insieme di istruzioni
  - Un'istruzione indica una operazione da effettuare su dati memorizzati
  - Tutti i dati sono memorizzati
  - Le istruzioni vengono eseguite in sequenza
- Ma, in genere, si discostano dal modello di Von Neumann per alcuni aspetti
  - Programma e dati non condividono la memoria
  - Il tipo dei dati è associato ai costrutti che contengono i dati (variabili, aggregazioni) e non alle istruzioni

Modello a 6 livelli

- Caratteristiche tipiche di un linguaggio ad alto livello imperativo
  - uso di variabili per memorizzare lo stato
  - tipi di dato associati alle variabili
  - aggregazioni di dati elementari
  - operazioni specificate mediante espressioni algebriche
  - costrutti per l'esecuzione selettiva di istruzioni
  - costrutti per l'esecuzione ripetuta di istruzioni
  - programma diviso in funzioni e procedure (Programmazione Procedurale)
  - struttura a blocchi innestati del programma

Cosa esattamente si studia in LPS

- Abbiamo iniziato la lezione dicendo che l'obiettivo principale di LPS è l'acquisizione di:
  Conoscenza, comprensione e capacità di operare con i sistemi di base per la programmazione
- Come ci mostra il modello a 6 livelli, un programma viene eseguito mediante una serie di traduzioni/interpretazioni in linguaggi di livello inferiore
- Principalmente ci interessa come un programma venga eseguito da una abstract machine di livello 2, perché:
  - le istruzioni dei *LM*, i linguaggi di livello 2, eseguono operazioni elementari sui dati
  - i programmi eseguiti dalle abstract machine di livello 2 sono quelli a più basso livello di astrazione ad essere realizzati in software, e quindi possono essere modificati, a differenza di quelli dei livelli 0 e 1

Cosa esattamente si studia in LPS

- Dunque, con *sistemi di base per la programmazione* intendiamo quei sistemi, presenti in un computer, che consentono l'esecuzione di programmi realizzati in software
  - sistemi per la produzione ed esecuzione di programmi in *LM*
  - sistemi che consentono alle abstract machine di livello 2 di eseguire programmi scritti in linguaggi di livello 5
- Purtroppo è molto scomodo per un essere umano non solo scrivere programmi in un *LM*, ma anche più semplicemente descrivere come un *LM* funziona e come viene usato per eseguire programmi scritti in un linguaggio di livello 5

Cosa esattamente si studia in LPS

- Le difficoltà di utilizzo e comprensione dei *LM* dipendono in primo luogo dalla loro sintassi, che è concepita per rendere efficiente l'esecuzione dei comandi da una abstract machine ma non per agevolare l'uso dei *LM* da parte degli umani
- Infatti ciascun costrutto di un *LM* è una stringa binaria, ovvero una sequenza di cifre binarie
- È molto scomodo per un essere umano ricordare, distinguere e comporre stringhe binarie
- Quindi, per lavorare più comodamente, utilizziamo linguaggi di livello 4 al posto dei corrispondenti linguaggi di livello 2

Cosa esattamente si studia in LPS

- In LPS lavoreremo quindi con i seguenti linguaggi
  - *LM*, ovvero i linguaggi delle abstract machine di più alto livello tra quelle realizzate in hardware
  - *ASM*, allo scopo di parlare di *LM* con una sintassi più comoda
  - un linguaggio ad alto livello imperativo per fare esempi di programmi da eseguire
- Quale linguaggio ad alto livello imperativo conviene usare in LPS?
- Poiché lo conoscete bene, il primo candidato da prendere in considerazione è Java

Cosa esattamente si studia in LPS

- Usare Java sarebbe possibile, ma
  - Java può essere considerato come linguaggio ad un livello di astrazione superiore al 5: infatti viene tradotto in bytecode ed eseguito da una JVM
  - esistono abstract machine di livello 2 che eseguono bytecode, ma è difficile programmarle
  - Java ha astrazioni complesse e una abstract machine molto diversa da una *ISA*: i programmi di livello 2 che traducono o interpretano programmi Java sono molto complessi
- Inoltre c'è una motivazione ulteriore che spinge ad usare un linguaggio diverso: studiare un linguaggio in più!
- Infatti conoscere più linguaggi rende programmatori migliori

Cosa esattamente si studia in LPS

- Vogliamo quindi un linguaggio ad alto livello imperativo diverso da Java, che abbia una abstract machine più simile a una *ISA*
  C Language
  - è molto conosciuto, *lingua franca* dell'ICT
  - influenza linguaggi più moderni
  - aiuta a costruire una prospettiva storica dell'ICT
  - è tuttora molto usato
- In futuro forse si userà un diverso linguaggio ad alto livello imperativo in LPS (Go ?)

A proposito del Linguaggio C

Its types and operations are well grounded in those provided by real machines, and for people used to how computers work, learning the idioms for generating time and space efficient programs is not difficult. At the same time the language is sufficiently abstracted from machine details that program portability can be achieved

D. M. Ritchie (HOPL-II, 1996)

A proposito dell'obiettivo principale di LPS

The best computer scientists are thoroughly grounded in basic concepts of how computers actually work, and indeed the essence of computer science is an ability to understand many levels of abstraction simultaneously

D. E. Knuth (ITiCSE'03)