DIPARTIMENTO DI INFORMATICA SISTEMISTICA E COMUNICAZIONE

Linguaggi di Programmazione
A.A. 2022-2023

Introduzione al corso

Marco Antoniotti
Gabriella Pasi
Fabio Sartori

Programma del corso

- Introduzione a diversi paradigmi di programmazione

- **Logica Matematica e linguaggi logici (Prolog)**
  - Termini, fatti (predicati), regole, unificazione, procedura di risoluzione

- **Linguaggi funzionali e Lisp (et al.)**
  - Atomi, liste, funzioni e ricorsione

- **Linguaggi imperativi (C/C++)**
  - Memoria, stato, assegnamenti, puntatori

- **Nuove (ok, non così nuove) tendenze**
  - Javascript
  - Inferenza automatica di tipi e linguaggi correlate (e.g., Haskell)
  - Julia

Sito WEB, risorse ed annunci

- Il sito WEB del corso si trova sulla piattaforma Moodle all'indirizzo http://elearning.unimib.it

- Per accedere al sito del corso di Linguaggi di Programmazione (tutti i moduli) **è necessario iscriversi sulla piattaforma**, usando il vostro login CAMPUS

- Durante il periodo di lezione, sono garantiti gli accessi a tutte le funzionalità della piattaforma e la risposta alle domande poste nei vari forum

- Vi preghiamo di controllare periodicamente anche il sito principale del Dipartimento http://www.disco.unimib.it per annunci urgenti e variazioni di orari ecc.

Modalità di Esame

- **Prove**

  Le prove d'esame sono costituite da uno scritto di 5-10 domande e da un progetto da consegnare entro una data prefissata

  - **Prove intermedie/Primo appello**

    - **Prima prova**
      X Novembre 2022, settimana di interruzione delle lezioni (YY novembre), ora e aula da definirsi
      Argomenti: Argomenti introduttivi, Prolog e programmazione logica

    - **Seconda prova**
      Fine Gennaio/Inizio Febbraio 2023, giorno ora e aula da definirsi
      Argomenti: Lisp e programmazione funzionale, C e programmazione imperativa

    - **Progetti**
      Un progetto (o due separati) da realizzare in **Prolog** e **Common Lisp** da consegnarsi entro l'**8 gennaio 2023**

Modalità di Esame

- **Appelli regolari (dopo il 1°)**
  - Gli appelli regolari sono costituiti da un progetto e da uno scritto e possibilmente da un orale a discrezione del docente su tutti i temi trattati durante il corso
  - Scritto, potenziale orale ed il progetto devono essere sostenuti **nello stesso appello**
  - Il progetto e lo scritto (e eventuale orale) sono corretti separatamente ed il voto complessivo finale viene pubblicato per la **verbalizzazione**

- **Calendario**
  - **Appello 2, fine Febbraio - Marzo 2023**
    aula e data da definirsi
  - **Appello 3, Giugno 2023**
    aula e data da definirsi
  - **Appello 4, Luglio 2023**
    aula e data da definirsi
  - **Appello 5, Settembre 2023**
    aula da definirsi

- **Verbalizzazioni**
  - Dovete registrarvi per gli esami scritti; le date a cui potete iscrivervi su S3 sono solo quelle degli esami scritti. Il resto dovrebbe (!) essere fatto tutto in automatico.

Votazioni

- La correzione del progetto dipende da quella dello scritto (ovvero dei compitini); qualora alcune domande chiave (a discrezione dei docenti) non abbiano avuto risposte corrette, il progetto non sarà corretto e l'esame risulterà insufficiente
- Il voto finale sarà una media (**pesata**) dei voti conseguiti nell'esame relativo alla parte "teorica" e nell'esame "progettuale"
  - Anche i voti dei compitini (e dei progetti) saranno mediati.
  - *Il peso è a discrezione dei docenti; ovvero, un voto troppo basso in una delle parti può causare un voto complessivo insufficiente nonostante una media sufficiente*
  - Dall'Anno Accademico 2018-2019 verbalizziamo correttamente i vari casi

    Voto (18-30L)
    Respinto
    Ritirato
    Assente

Studenti anni precedenti

Alcune note per gli studenti degli anni precedenti…

LE MODALITÀ DI ESAME SONO QUELLE DELL'ANNO IN CORSO (2022-2023)

- Le istruzioni e le modalità in vigore negli anni precedenti al presente **NON** SONO PIÙ VALIDE!

Per gli studenti che devono ancora fare l'esame da 12 crediti:

- I compitini di "Linguaggi e Computabilità" valgono per il modulo di LC da 4 crediti
- Durante gli appelli regolari di LP sarà possibile fare lo scritto di LC da 4 crediti
- Gli orali di LC vanno concordati con i docenti del corso di LC corrente
- I moduli di LP e LC devono essere superati nello stesso appello

Studenti con blocchi

- Gli studenti con blocchi su Inglese o altre materie **DEVONO** risolverli **PRIMA DI ISCRIVERSI AD UN APPELLO**

- Ovvero: se avete un blocco, assicuratevi di non averlo al momento in cui vi iscrivete ad un appello, in caso contrario il vostro voto sarà trattato come un **PARZIALE**
  - Vedasi slides successive

NESSUNA ECCEZIONE!

- Ciò significa che se durante il semestre risultate insufficienti in una delle prove intermedie **O** nel progetto, **DOVRETE RIFARE L'INTERO ESAME** negli appelli successivi

- **NON SI TERRÀ BUONO ALCUN PARZIALE!**

- Ripetiamo!

  NON SI TERRÀ BUONO ALCUN PARZIALE!

Tanto per chiarire: NESSUNA ECCEZIONE!

- Ebbene si, questa slide è una ripetizione di quella precedente: **LEGGETE CON CURA**

- Ciò significa che se durante il semestre risultate insufficienti in una delle prove intermedie **O** nel progetto, **DOVRETE RIFARE L'INTERO ESAME** negli appelli successivi

- **NON SI TERRÀ BUONO ALCUN PARZIALE!**

- Ripetiamo!

  NON SI TERRÀ BUONO ALCUN PARZIALE!

Testi consigliati

- Leon Sterling and Ehud Shapiro, *The Art of Prolog Advanced Programming Techniques* - 2nd Edition
- I. Bratko, *Prolog Programming for Artificial Intelligence*
- Paul Brna, *Prolog Programming A First Course*, http://computing.unn.ac.uk/staff/cgpb4/prologbook/book.html
- Abelson, Sussman e Sussman, *Structure and Interpretation of Computer Programs* (SICP), http://mitpress.mit.edu/sicp
- Peter Seibel, *Practical Common Lisp* (PCL), http://www.gigamonkeys.com/book
- Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, *How To Design Programs*, The MIT Press (anche su Web)
- Brian W. Kernigham & Dennis M. Ritchie, *C Programming Language* (2nd Edition), Prentice Hall, 1988
- Samuel P. Harbison & Guy L. Steele, *C: A Reference Manual* (5th Edition), Prentice Hall, 2002
- Robert Sedgewick, *Algorithms in C, Parts 1-5* (Bundle): *Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms* (3rd Edition), Addison Wesley, 2001

Ambienti di programmazione consigliati

- Common Lisp
  - Lispworks Personal Edition (www.lispworks.com) on OSX or Windows
    - Per i veri programmatori Linux, CMUCL o SCBL o ECL o ABCL o CLisp con Emacs+SLIME

- Prolog
  - SWI-prolog (www.swi-prolog.org)
    - Per i veri programmatori con Emacs come editor

- C (e C++)
  - GCC, clang, Visual Studio oppure uno qualsiasi dei compilatori elencati qui: http://en.wikipedia.org/wiki/List_of_compilers#C_compilers
    - Per i veri programmatori con Emacs come editor

Compiti a casa

- Trovate ed installate **Emacs** sul vostro computer
- Trovate ed installate **LispWorks Personal Edition** sul vostro computer
- Trovate ed installate **SWI Prolog** sul vostro computer
- Trovate e, se già non c'è, installate un **compilatore C/C++** sul vostro computer
  - Se è già installato, imparate come invocare il compilatore dalla **command line** (Yes Virginia! The **command line**: `cmd.exe` or Power Shell or the UN*X shell)
  - Eclipse, Xcode, MS Visual Studio, IntelliJ, Netbeans etc., sono per Jedi Masters (che hanno già padroneggiato la spada laser: Emacs); umili padawans appena voi siete!

Come si scrivono i programmi: Le Regole!

- I programmi si scrivono con un editor di testo. **Esiste un solo editor di testo: Emacs!**
  - No *vi*, no *vim*, no *Nano*, no *Atom*, no *Sublime Text*, no niente che non sia **Emacs**! *vi* è (a malapena) accettabile se e solo se si usa uno degli emulatori vi di **Emacs**.

- Il carattere che userete **NON è proporzionale**.

- Emacs **INDENTERÀ** il codice per voi.
  Fidatevi di lui (ed imparate come si comporta)

Come si scrivono i programmi: Le Regole!

- Così come i “coding standards” GNU, Google, Node.js, R e molti, molti altri (specialmente quelli del vostro professore) ci insegnano: **le linee NON DEVONO** (ripetiamo: **NON DEVONO**!) **essere più lunghe di 80 colonne**!

- https://www.gnu.org/prep/standards/

- https://google.github.io/styleguide/cppguide.html

- http://www.w3schools.com/js/js_conventions.asp

Come si scrivono i programmi: Le Regole!

- Tu METTERAI SPAZI attorno agli operatori (+, -, *, /, =, ==, !=, etc) ed UNO SPAZIO dopo le virgole (e punti e virgola).

```
/* VERY VERY VERY BAD! */
area=pi*radius^2

/* GOOD */
area = pi * radius ^ 2

/* VERY VERY VERY VERY BAD! */
colors=[red,green,blue]

/* GOOD */
colors = [red, green, blue]
```

Come si scrivono i programmi: Le Regole!

- Tu NON METTERAI uno spazio tra il nome di una funzione/predicato e la parentesi.

```
factorial (42, 1)        /* NOT SO GOOD! */
factorial(42,1)          /* VERY BAD! */
factorial (42,1)         /* VERY VERY BAD! */
factorial(42, 1)         /* GOOD */
```

Come si scrivono i programmi: Le Regole!

- Tu METTERAI uno spazio tra la parola chiave di uno statement di controllo e la parentesi.

```
/* VERY VERY VERY VERY VERY VERY BAD! */
while(itsFalse||itsTrue) …
for(int i=0;i<n;i++) …
if(something>42) …

/* GOOD */
while (itsFalse || itsTrue) …
for (int i = 0; i < n; i++) …
if (something > 42) …
```

Come si scrivono i programmi: Le Regole!

- Tu NON USERAI i “commenti scatoletta”.

```
/************************/
/* VERY VERY VERY VERY */
/* VERY VERY BAD!      */
/************************/

##########################
# VERY VERY VERY VERY    #
# VERY BAD!!!!           #
##########################

%%% GOOD

/* VERY
 * GOOD!
 */
```

Correzione Progetti

I progetti saranno pre-processati con i seguenti strumenti UN*X

1. Il file `yourfile.c` (o `.lisp`, o `.pl`) sarà reindentato da **Emacs**
2. Il comando **`diff`** sarà invocato sui due file

   ```
   $ diff yourfile.c yourfile-ind.c
   ```

   Se ci saranno delle differenze il vostro progetto risulterà insufficiente.
3. Il comando **`cut`**; in particolare i vostri files saranno riscritti così:

   ```
   $ cut -c -80 yourfile-ind.c > yourfile-processed.c
   ```

4. Il file `yourfile-processed.c` è quello che sarà usato per la valutazione; ovviamente, i sistemi Lisp, Prolog e C non amano codice incompleto e sintatticamente scorretto. Se il vostro progetto genera un errore sarà considerato insufficiente.

…and now, for something completely different…

Monty Python 1971

Ma prima, un po' di convivialità

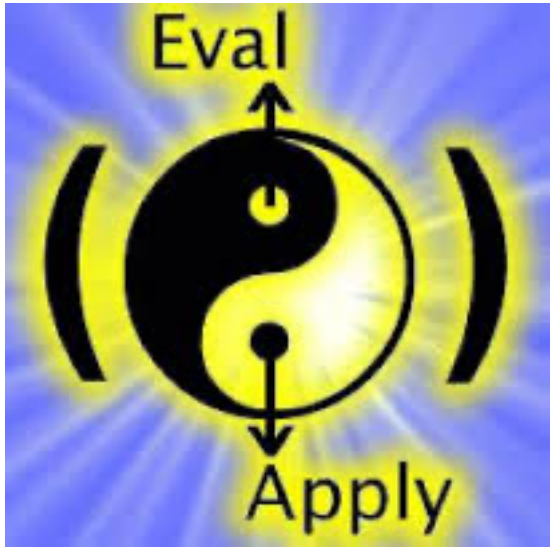
1. 99 bottles of beer on the wall… 99 bottles of beer
2. Take 1 down, pass it around
3. Repeat while there are bottles

https://www.99-bottles-of-beer.net

Potete anche cercare un linguaggio di programmazione speciale di origini italiane: **Monicelli**

(Google "Monicelli Programming Language")

LINGUAGGI IMPERATIVI



LANGUAGGI FUNZIONALI

LANGUAGGI LOGICI

DIPARTIMENTO
DI INFORMATICA
SISTEMISTICA
E COMUNICAZIONE

OBJECT ORIENTED?

Una classificazione dei linguaggi di programmazione

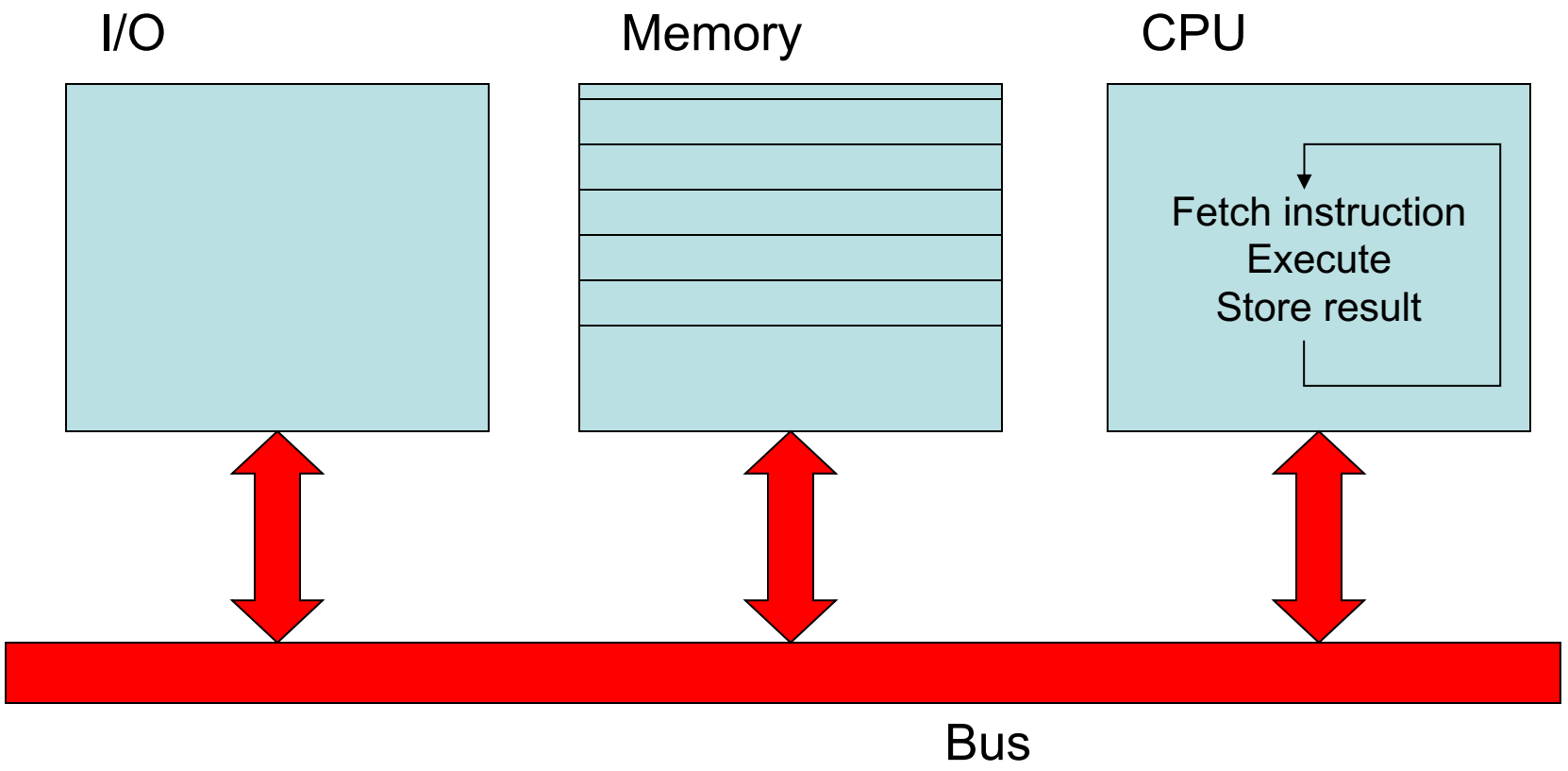
- **Linguaggi imperativi**
  - Vari assemblers, Fortran, Pascal, C, Ada95, Bliss, PL/1, C++, Python, …
- **Linguaggi logici**
  - Prolog, …
- **Linguaggi funzionali**
  - FP, Lisp, Common Lisp, Scheme, ML, Ocaml, Haskell, …

- Ognuna delle categorie citate può contenere dei **linguaggi ad oggetti**
  - Il paradigma ad oggetti è quindi ortogonale alla classificazione data anche se presume alcune caratteristiche tipiche dei linguaggi imperativi
    - Memoria e "stato" di un'istanza
  - Linguaggi ad oggetti: Smalltalk, Ada95, C++, Java, Simula, Common Lisp, Ocaml, Python, Ruby, Julia …

Paradigma imperativo (procedurale)

- Le caratteristiche essenziali dei linguaggi imperativi sono strettamente legate all'architettura di von Neumann.
- L'architettura di von Neumann è costituita da due componenti fondamentali
  - memoria (componente passiva)
  - processore (componente attiva)
- La principale attività del processore consiste nell'eseguire calcoli e assegnare valori a celle di memoria $\Rightarrow$ il che implica che il concetto di **variabile** è un'astrazione di cella di memoria (fisica).
- I linguaggi Assembler, Fortran, Pascal, e C (ad esempio) sono basati su distinti livelli di astrazione di questa architettura.

La macchina di Von Neuman



Paradigma imperativo (procedurale)

- I linguaggi imperativi adottano uno **stile prescrittivo**
  - Un programma scritto in un linguaggio imperativo prescrive le operazioni che il processore deve eseguire per modificare lo stato del sistema(ad es. assegnamento)
  - Esecuzione delle istruzioni nell'ordine in cui appaiono nel programma (eccezione: **strutture di controllo**)

- Realizzati sia attraverso interpretazione (Basic, Tcl, Python, Ruby, Javascript, ...) sia mediante compilazione (C, Pascal, Fortran, C++, Ada, COBOL, PL/1, ...)

- Nati più per manipolazione numerica che simbolica

Paradigma imperativo (procedurale)

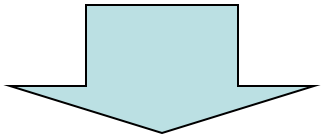
- **Programma = Algoritmi + Strutture Dati** (N. Wirth, circa 1976)

- La struttura del programma consiste in:
  - Una parte di **dichiarazione** in cui si dichiarano tutte le variabili del programma e il loro tipo
  - Una parte che descrive l'algoritmo risolutivo utilizzato, mediante **istruzioni** del linguaggio

- Le istruzioni si dividono in:
  - istruzioni di **lettura e scrittura**
  - istruzioni di **assegnamento** (astrazione di cella di memoria)
  - istruzioni di **controllo**

Definizione di paradigmi di programmazione alternativi

- **Motivazioni**
  - Necessità di gestire applicazioni a più alto livello di astrazione
  - Tentativo di sviluppare programmi più concisi, più semplici da scrivere, più vicini alla logica del problema, la cui correttezza sia più "semplice" da verificare



PROGRAMMAZIONE **FUNZIONALE**
PROGRAMMAZIONE **LOGICA**
PROGRAMMAZIONE A **OGGETTI**

Paradigmi funzionale e logico: caratteristiche comuni

- Linguaggi ad “altissimo livello” (utilizzabili – in teoria – anche da non-programmatori)
- Generati per manipolazione **simbolica** non numerica
- Concetti di programma e di struttura dati non nettamente separati (un programma è specificato per mezzo di una struttura dati) a seconda del linguaggio
- Basati su **concetti matematici** e non come astrazioni della macchina di Von Neumann
- I linguaggi funzionali e logici adottano uno stile essenzialmente **dichiarativo**

Paradigma Logico

- Concetto primitivo: **deduzione logica**
  - **Base:** logica formale
  - **Obiettivo**: formalizzare il ragionamento

- «Programmare» significa:
  - Descrivere il problema con frasi (formule logiche) del linguaggio
  - **Interrogare** il sistema, che effettua deduzioni sulla base della “conoscenza rappresentata”

Paradigma logico

- **Programma = conoscenza + controllo**
  - Stile dichiarativo: la conoscenza del problema è espressa indipendentemente dal suo utilizzo (COSA non COME);
  - Alta modularità e flessibilità
  - Definire un linguaggio logico significa definire come il programmatore può esprimere la conoscenza e quale tipo di controllo si può utilizzare nel processo di deduzione
    - Problematiche di **rappresentazione della conoscenza**

Linguaggio Prolog

- Un programma Prolog è costituito da:
  - Asserzioni incondizionate (fatti):

```
A.
```

  - Asserzioni condizionate (o regole):

```
A :- B, C, D, …, Z.
```

    - `A`: è la conclusione (conseguente)
    - `B,C,D, …, Z`: sono le premesse (o antecedenti)

- Una **interrogazione** (**query**) ha la forma:

```
:-  K, L, M, …, P.
```

Linguaggio Prolog

- **Esempio**: due individui sono colleghi se lavorano per la stessa ditta

REGOLA

```
collega(X, Y) :-
     lavora(X, Z),
     lavora(Y, Z),
     diverso(X, Y).
```

FATTI

```
lavora(ciro, ibm).
lavora(ugo, ibm).
lavora(olivia, samsung).
lavora(ernesto, olivetti).
lavora(enrica, samsung).
```

INTERROGAZIONE

```
:- collega(X, Y).
```

A Comparison between Prescriptive Style and Declarative Style

- **Problema: ordinare una lista**

- **STILE PRESCRITTIVO**:
  - Controlla prima se la lista *L* è vuota; se sì dai come risultato la lista vuota. Altrimenti calcola una permutazione L_1 di *L* e controlla se è ordinata; se sì termina dando come risultato L_1, altrimenti calcola un'altra permutazione di *L* etc.…

- **STILE DICHIARATIVO**:
  - Il risultato dell'ordinamento di una lista vuota è la lista vuota.
  - Il risultato dell'ordinamento di una lista *L* è L_1 se la lista L_1 è ordinata ed è la permutazione di *L*.

Confronto tra programmazione prescrittiva e programmazione dichiarativa

- **Problema: ordinare una lista**

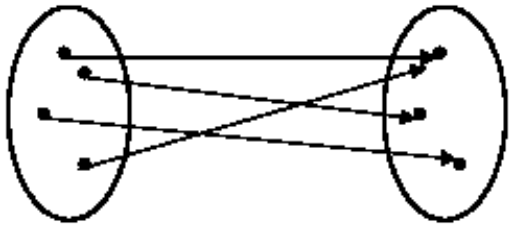
- **STILE PRESCRITTIVO**:
  - Il programmatore deve specificare la sequenza di istruzioni che servono a **generare** la sequenza di permutazioni della lista L

- **STILE DICHIARATIVO**:
  - L'ambiente (Prolog o di *theorem proving*) si fa farico di **generare** le possibili permutazioni della lista L secondo un processo di deduzione matematica

Paradigma funzionale

- **Concetto primitivo: funzione**
- Una **funzione** è una regola di associazione tra due insiemi, che associa un elemento del primo insieme (*dominio*) ad un elemento del secondo insieme (*codominio* o «*range*»)
- La definizione di una funzione ne specifica il **dominio**, il **codominio** e la **regola di associazione**
- Esempio
  (in una "sintassi concreta" a caso):

```
incr: N → N
incr(x) = x + 1.
```

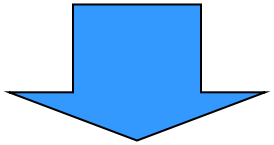


- Dopo averne dato definizione, una funzione può essere **applicata** a un elemento del dominio (argomento) per restituire l'elemento del codominio ad esso associato (**valutazione**):

```
incr(3) ⟹ 4
```

Paradigma funzionale

- **L'unica operazione** utilizzata nel paradigma funzionale (puro) è l'**applicazione** di funzioni.
- Il ruolo dell'esecutore di un linguaggio funzionale si esaurisce nel **valutare** l'applicazione di una funzione (il programma) e produrre un **valore**
- Nel paradigma funzionale "puro" il valore di una funzione è determinato soltanto dal valore degli argomenti che riceve al momento della sua applicazione, e non dallo stato del sistema rappresentato dall'insieme complessivo dei valori associati a variabili (e/o locazioni di memoria) in quel momento
  - **Assenza di effetti collaterali**



- Il concetto di variabile utilizzato è quello di **"costante" matematica**, i cui valori **non sono mutabili** (nessun assegnamento)

- L'essenza della programmazione funzionale consiste nel **combinare funzioni** mediante **composizione** e utilizzo del concetto di **ricorsione**

Paradigma funzionale

- **Programma = Composizione di Funzioni + Ricorsione**

- La struttura del programma consiste nella definizione di un insieme di funzioni (mutualmente ricorsive)

- L'esecuzione del programma consiste nella valutazione dell'applicazione di una funzione principale a un insieme di argomenti

Linguaggio LISP (LISt Processing)

- Proposto nel 1958 da John McCarthy
- Il progetto originale era per un linguaggio funzionale puro

- Nel corso degli anni sono stati sviluppati molti ambienti di programmazione Lisp
  - PSL, Zetalisp, InterLisp, MacLisp, Vlisp, LeLisp, Emacs Lisp
  - I due standard correnti più diffusi sono
    - **Common Lisp**
    - **Scheme**

Linguaggio LISP

- **Esempio**

  Controllare se un elemento (item) appartiene ad un insieme (rappresentato con una lista)

```
(defun member (item list)
     (cond ((null list) nil)
           ((equal item (first list)) T)
           (T (member item (rest list)))))

(member 42 (list 12 34 42))
```

- Il programma è rappresentato da una struttura fondamentale dati del linguaggio (*omoiconicità*)
- Non c'è assegnamento
  - Ok, quasi…

Ambienti “Run Time” di Linguaggi Logici, Funzionali (e non)

- Richiami di nozioni di architettura e programmazione…

- Per eseguire un programma in un qualsiasi linguaggio il sistema (ovvero il sistema operativo) deve mettere a disposizione un **ambiente “run time”**, che fornisca almeno due funzionalità
  - Mantenimento dello stato della computazione (program counter, limiti di memoria etc)
  - Gestione della memoria disponibile (fisica e virtuale)

- L’ambiente run time può essere una vera e propria macchina virtuale quale quella di Java (la Java Virtual Machine - JVM) o .Net Microsoft.

Ambienti "Run Time" di Linguaggi Logici, Funzionali (e non)

- In particolare la gestione della memoria avviene usando due aree concettualmente ben distinte con funzioni diverse
  - Lo **Stack** dell'ambiente run time serve per la gestione delle chiamate - anche e soprattutto ricorsive! - a procedure (o metodi, o funzioni, o subroutines etc. etc.)
  - Lo **Heap** dell'ambiente run time serve per la gestione di strutture dati dinamiche
    - Esempio tipico: liste ed alberi
    - Cfr., operatore `new` di Java e C++, libreria `malloc` per il C
- I linguaggi logici e funzionali (ma anche Java) utilizzano pesantemente lo **Heap** dato che forniscono come strutture dati "built-in" liste e, spesso, vettori di dimensione variabile.

Stack e valutazione di procedure: richiami

- La valutazione di **procedure** avviene mediante la costruzione (sullo stack di sistema) di activation frames
- I **parametri formali** di una procedura vengono associati ai valori (nb: si passa tutto per **valore**; ribadiamo che *non esistono effetti collaterali*)
- Il **corpo** della procedura viene valutato (ricorsivamente) tenendo conto di questi legami in maniera "**statica**"
  - Ovvero bisogna tener presente cosa accade con variabili che risultano "libere" in una sotto-espressione
- Ad ogni sotto-espressione del corpo si sostituisce il valore che essa denota (computa)
- Il **valore** (valori) restituito dalla procedura in un'espressione `return` o con meccanismi analoghi è il valore del corpo della procedura (che non è altro che una sotto-espressione)
  - Quando il valore finale viene ritornato i legami temporanei ai paramentri formali spariscono (lo stack di sistema subisce una "pop" e l'activation frame viene rimosso)

Stack e valutazione di procedure: richiami

Esempio

```
int doppio(int x) { return 2 * x; }
```

Definizione del legame tra **`doppio`** e la sua definizione

La chiamata

```
int d = doppio(3);
```

- estende l'ambiente corrente, dove è stata dichiarata la variabile **`d`**, con quello locale che contiene i legami tra parametri formali e valori dei parametri attuali
  - Un activation frame viene inserito in cima allo stack di valutazione
- valuta il corpo della procedura
- ripristina l'ambiente di partenza
  - Il risultato della chiamata viene salvato nella variabile **`d`**
  - l'activation frame viene rimosso dalla cima dello stack di valutazione

Stack e chiamate di procedure: riassunto

- Per l'esecuzione di una procedura **F**, un programma deve eseguire i seguenti sei passi:
  - mettere i parametri in un posto dove la procedura possa recuperarli
  - trasferire il controllo alla procedura
  - allocare le risorse (di memorizzazione dei dati) necessarie alla procedura
  - effettuare la computazione della procedura
  - mettere i risultati in un posto accessibile al chiamante
  - restituire il controllo al chiamante

- Queste operazioni agiscono sui registri a disposizione e sullo "stack" utilizzato dal runtime (esecutore) del linguaggio

- Lo spazio richiesto per salvare (sullo stack) tutte le informazioni necessarie all'esecuzione di una procedura **F** ed al ripristino dello stato precedente alla chiamata è quindi costituito da
  - Spazio per i registri da salvare prima della chiamata di una sotto procedura
  - Spazio per l'indirizzo di ritorno (nel codice del corpo della procedura)
  - Spazio per le variabili, definizioni locali, e valori di ritorno
  - Spazio per i valori degli argomenti
  - Spazio per il riferimento statico (*static link*)
  - Spazio per il riferimento dinamico (*dynamic link*)
  - Altro spazio dipendente dal particolare linguaggio e/o politiche di allocazione del compilatore

Stack e chiamate di procedure: riassunto

Activation Frame di una procedura

- Spazio per i registri da salvare prima della chiamata di una sotto procedura
- Spazio per l'indirizzo di ritorno (*return address* nel codice del corpo della procedura)
- Spazio per le variabili, definizioni locali e spazio per valori di ritorno
- Spazio per i valori degli argomenti
- Spazio per il riferimento statico (*static link*)
- Spazio per il riferimento dinamico (*dynamic link*)
- Altro spazio dipendente dal particolare linguaggio e/o politiche di allocazione del compilatore

Return address
Registri . .
Static link
Dynamic link
Argomenti . .
Variabili/definizioni locali, valori di ritorno . .

Stack e chiamate di procedure: riassunto

- **Esempio**

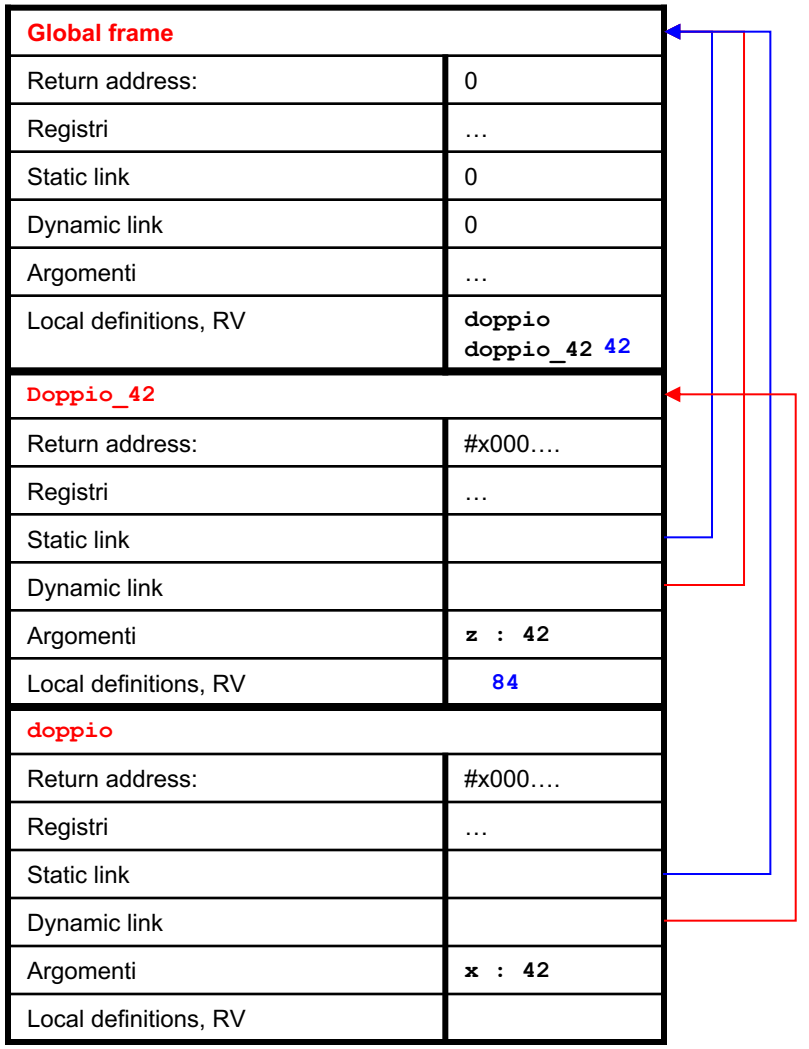
```
int doppio(x) { return 2 * x; }

int doppio_42(z) { return doppio(z) - 42; }
```

La chiamata `doppio_42(42)` ha il seguente effetto:

- **Static link** e **dynamic link** sono molto importanti perchè servono a mantenere informazioni circa il
  - Dove una procedura è definita
  - Quando una procedura è chiamata

- Il contenuto effettivo di un activation frame dipende da diverse scelte implementative

- L'esempio riportato non è necessariamente completo



Heap e Garbage Collection

- L'area di memoria per la manipolazione di strutture dati dinamiche verrà spiegata meglio al momento dell'introduzione delle primitive per la costruzione delle liste in Prolog e Lisp

  - Nel frattempo potete ripassare questi concetti nel contesto di Java (operatore **`new`**), C++ (operatore **`new`**) e C (libreria **`malloc`**, disponibile anche in C++)

  - Il componente software che si occupa della gestione automatica della memoria in Lisp, Prolog, Java, C#, ed altri linguaggi è il **garbage collector** (il servizio di raccolta rifiuti)
    - C e C++ (ed altri linguaggi) non hanno un garbage collector standard

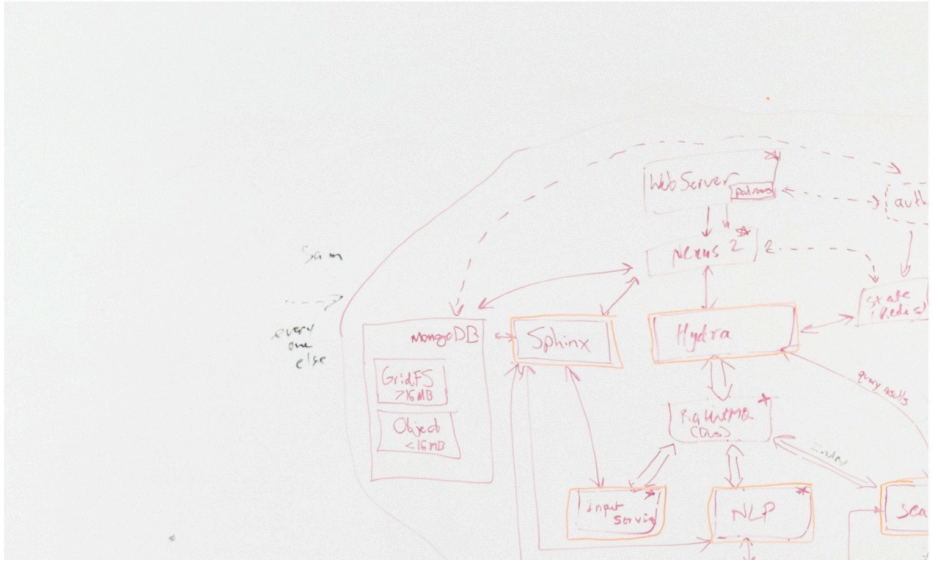
Qualche motivazione…

La startup della Silicon Valley Kindy (kindy.com) usa Prolog

https://www.nytimes.com/2018/06/20/technology/deep-learning-artificial-intelligence.html

The New York Times

Is There a Smarter Path to Artificial Intelligence? Some Experts Hope So



Sommario (1/4)

- I linguaggi si possono raggruppare in tre grandi gruppi
  - **Linguaggi imperativi**
    - Vari assemblers, Fortran, Pascal, C, Ada95, Bliss, PL/1, C++, Python, Ruby…
  - **Linguaggi funzionali**
    - FP, Lisp, Common Lisp, Scheme, ML, Ocaml, Haskell, …
  - **Linguaggi logici**
    - Prolog, …
  - Il paradigma ad oggetti è trasversale a questi tre gruppi, sebbene presuma caratteristiche imperative derivate dall'astrazione dell'architettura di von Neuman

Sommario (2/4)

- Tutti i linguaggi operano in un **ambiente runtime** che può avere caratteristiche molto diverse a seconda dei casi

- Ci sono essenzialmente tre funzionalità che l'ambiente runtime fornisce
  - **Stack** per la manipolazione delle chiamate a procedure
    - Manipolazione degli activation frames
  - **Heap** per l'allocazione di strutture di dati dinamiche
    - Liste, vettori, componenti di alberi etc. etc.
    - Il garbage collector (se presente) si preoccupa di liberare la memoria non utilizzata nello Heap.
  - Threading models
  - Interazione con il Sistema Operativo sottostante (cross-cutting)

Sommario (3/4)

- Nello svolgersi del corso noi vedremo
  - **Prolog** come esempio di linguaggio logico
  - **Lisp** come esempio di linguaggio funzionale
  - **C** (C++) come esempio di linguaggio imperativo

- Cose che non vedremo
  - Estensibilità dei linguaggi mediante macro e “parser” integrati
  - Estensioni “Object Oriented” e concorrenti di Lisp e Prolog
  - Inferenza automatica di tipi (Haskell, ML, Mercury etc)

- **Andate a recuperare i libri e gli ambienti su Internet stasera!**

- **Buon Divertimento**

Sommario (4/4)

- Non c'è un '**4**'; ma se non avessi preparato le slides in questo modo, a questo punto sareste già tutti andati
- **Andate a recuperare i libri e gli ambienti su Internet stasera (Emacs!!!)**
- **Buon Divertimento**