Linguaggi di programmazione

Prof. Roberto Gorrieri (I modulo)

e

Prof. Maurizio Gabbrielli e

Dr. Saverio Giallorenzo (II modulo)

Corso annuale (12 crediti) diviso in due/tre moduli

facciamo le presentazioni…

- Roberto Gorrieri
  - Professore di Informatica
  - Laurea in Scienze dell'Informazione, Pisa, 1986
  - Dottore di Ricerca in Informatica, Pisa, 1991
  - Insegnato sempre a Bologna e Cesena, prima come ricercatore (dal 1991), poi come Prof. Associato (dal 1992), quindi come Prof. Ordinario (dal 2000)

- Interessi di ricerca:
  - Metodi Formali per la specifica, l'analisi e la verifica di sistemi concorrenti e distribuiti (corso di Modelli e Sistemi Concorrenti alla Laurea Magistrale): automi, linguaggio CCS, logiche modali e temporali, reti di Petri --- *naturale prosecuzione di alcuni temi di questo modulo*
  - Metodi Formali per la Sicurezza: flusso d'informazioni, analisi di protocolli di sicurezza.
  - Teoria dei linguaggi di programmazione: semantica.

facciamo le presentazioni…

- Maurizio Gabbrielli
  - Professore di Informatica
  - Laurea in Scienze dell'Informazione, Pisa, 1988
  - Dottore di Ricerca in Informatica, Pisa, 1993
  - Ricercatore al CWI Amsterdam 1993-95
  - Ricercatore Pisa (1995-98), associato a Udine (1998-2001), ordinario a Bologna (2001-)
  - Attualmente direttore del Dipartimento di Informatica

- Interessi di ricerca:
  - Teoria dei linguaggi di programmazione
  - Linguaggi logici e con vincoli
  - Intelligenza Artificiale (in particolare, metodi di machine learning per i linguaggi di programmazione)

facciamo le presentazioni…

- Saverio Giallorenzo
  - Ricercatore a Tempo Determinato, Univ. di Bologna, 2020
  - Assegnista di Ricerca, Univ. of Southern Denmark, 2018-20
  - Assegnista di Ricerca, Univ. di Bologna, 2016-17,
  - Dottore di Ricerca in Informatica, Univ. di Bologna, 2016
  - Laurea Magistrale in Science di Internet, Univ. di Bologna, 2012

- Interessi di ricerca:
  - Linguaggi di Programmazione Orientati ai Servizi (microservizi, serverless)
  - Teoria di Linguaggi di Programmazione (specifiche formali, invarianti, analisi)
  - Design e Sviluppo di Linguaggi (esempi: Jolie, Choral, APP, AIOCJ)
  - Sicurezza (architetture software e contrasto malware

Info

- Email: **roberto.gorrieri@unibo.it**
  maurizio.gabbrielli@unibo.it
  saverio.giallorenzo2@unibo.it
- Ricevimento:
  - Dopo lezione (o, meglio, per appuntamento via email)
- Web: **https://virtuale.unibo.it/course/view.php?id=46127**
- Ma anche (per i compiti vecchi)
  - **www.cs.unibo.it/~gorrieri/LP/lp.html**

    login: esame

    password: linguaggi

  https://saveriogiallorenzo.com/teaching/#pl
- Altri Web

  www.cs.unibo.it/~gabbri/corsi/linguaggi.html

  www.cs.unibo.it/~martini/PP/LP-index.html

  www.unibo.it/sitoweb/gianluigi.zavattaro

Modalità d'esame

- Appelli regolari: (giugno, luglio, settembre, gennaio, febbraio)
  - scritto + orale (da svolgersi *nella stessa sessione*)
  - Allo scritto non è consentita la consultazione di materiale.
  - Ci si presenta ad entrambe le prove muniti del tesserino universitario (e possibilmente di un documento d'identità)
  - Sono ammessi all'orale i candidati con voto allo scritto >= 18.

- Iscrizione obbligatoria agli appelli attraverso almaesami.

- Per agevolare l'orale: lista di domande possibili, in un file pdf sul sito del corso (solo per la mia parte).

Modalità d'esame (2)

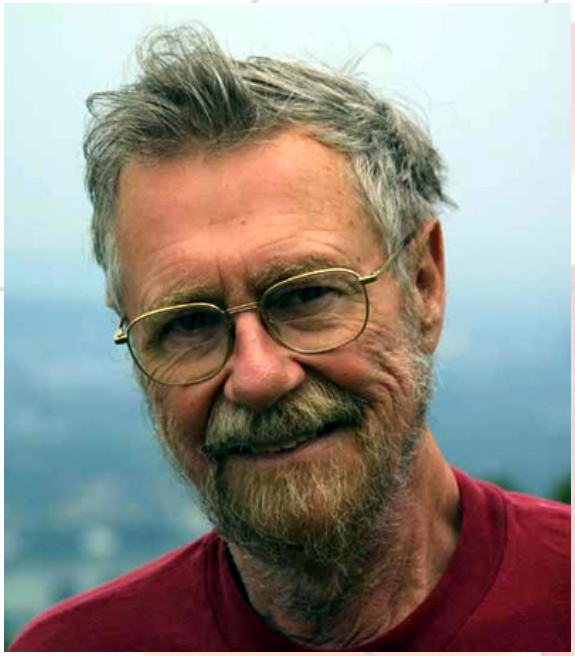
- Parziali
  - Compitini al termine di ciascuna delle due parti. Se in entrambi i parziali viene ottenuta una valutazione >= 16 e la somma delle due valutazioni è >= 36 non e' necessario fare lo scritto. Orale conseguente da fare nella sessione estiva.

- Parziale 1 di fine modulo 1:
- Mercoledì 20 dicembre 2023, ore 10:00-12:30, 12:45-15:15 in Ercolani 1 (due turni, se siete in molti)

- Iscrizione obbligatoria ai parziali attraverso almaesami.
- Ricordo che sono ammessi al parziale 2 di fine maggio 2024 solo coloro i quali abbiano superato il parziale 1 (con voto >=16)

Obiettivi, 1

Computer Science is no more about computers than Astronomy is about telescopes. -- E.W. Dijkstra (1930 – 2002); Premio Turing 1972.



A Good Programming Language is a Conceptual Universe for thinking about Programming. -- Alan Perlis (1922 – 1990); Primo premio Turing 1966; primo presidente ACM; primo direttore CS Dpt a CMU

Obiettivi, 2

- Conoscere decine di linguaggi diversi?
  - Conoscere meccanismi comuni a centinaia di linguaggi!
- Per evitare l'invecchiamento della tecnologia...
  - Competenza trasversale sui LP
    - Imparare cosa è importante riguardo a vari linguaggi
    - Comprendere idee e metodologie di programmazione
    - Comprendere i linguaggi che usate (C, C++, Java) per confronto con altri linguaggi
    - Comprendere la storia dei concetti e loro evoluzione
    - Conoscere il costo dei diversi meccanismi
  - Pensiero critico
    - Proprietà dei linguaggi, non documentazione

Gli attrezzi dell'informatico

C

Altri linguaggi

Se hai solo un martello, tutto ti sembra un chiodo

Programma del *corso* -- *Gorrieri*

- Evoluzione dei linguaggi di programmazione: Dai linguaggi assembler ai linguaggi ad alto livello.
- Macchine astratte, interpreti, compilatori.
- Compilatori: struttura generale (fasi/moduli in cui è organizzato)
- Come descrivere un linguaggio: sintassi, semantica, pragmatica
- Sintassi (BNF) e semantica (tecnica SOS).
- Grammatiche regolari, automi a stati finiti, espressioni regolari: equivalenze e risultati principali (ad esempio, minimizzazione).
- Scanner: costruzione di analizzatori lessicali (cenni a lex).
- Grammatiche libere da contesto, automi a pila: equivalenze e risultati principali (ad esempio, pumping theorem).
- Grammatiche libere deterministiche: algoritmi di riconoscimento e costruzione dell'albero di derivazione; grammatiche LL(1), LR(0), SLR(1), LR(1), LALR(1).
- Parser: costruzione di analizzatori sintattici (cenni a yacc).
- Fondamenti: esistono vincoli che un compilatore non può verificare; proprietà indecidibili; Macchine di Turing (cenni).

Prosecuzione:

- Compilatori e Interpreti (LM: Laneve)
- Informatica Teorica (LT: Asperti)

Programma del *corso -- Gabbrielli e Giallorenzo*

- Nomi: blocchi e ambiente; regole di scope (statico vs dinamico)
- Gestione della memoria: statica, dinamica; pila e heap; regole di scoping; garbage collection.
- Strutturare il controllo: espressioni e comandi
- Astrarre sul controllo: funzioni e procedure.
- Parametri e modalità di passaggio: per valore, per riferimento, per risultato, per nome. Parametri funzionali. Chiusure. Eccezioni
- Strutturare i dati: tipi di dato, sistemi di tipo, controlli.
- Astrarre sui dati: tipi di dato astratti
- Paradigma orientato agli oggetti: classi e oggetti, ereditarietà, selezione dinamica dei metodi, astrazione.
- *Cenni al paradigma funzionale: funzioni di ordine superiore, strategie di valutazione, tipi, lambda-calcolo, cenni a Scala*
- *Cenni al paradigma logico: unificazione, pattern-matching, risoluzione, semantica operazionale, backtracking, cenni a Prolog*
- *Cenni alla programmazione concorrente e service oriented: Jolie.*

Prosecuzione:

- *Emerging Programming Paradigms (LM: Sacerdoti Coen) Si studiano linguaggi quali Go, Scala, Erlang, Rust*
- *Scalable and Cloud Programming (LM: Zavattaro) Scala ed altri linguaggi*

Libro di testo di riferimento (per i due moduli)

In Italiano (ed. 2010):
Copre il modulo 1 e parte del modulo 2 (non la parte del Prof. Giallorenzo)

Approfondimenti sul libro di testo

Materiale didattico aggiuntivo (approfondimenti nel libro) non stampato nel libro:

sulla pagina web

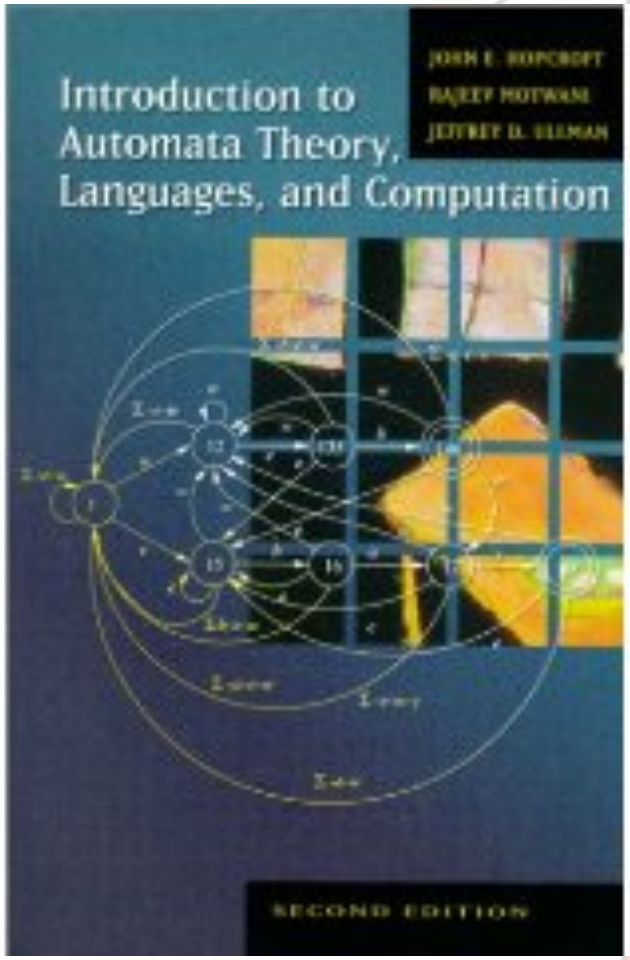
http://www.ateneonline.it/gabbrielli/

Capitoli rilevanti per Modulo 1

- Capitolo 1: Macchine Astratte
- Capitolo 2: Descrivere un linguaggio di programmazione (parte non sul libro: semantica operazionale – sul sito ci sono i miei appunti a mano)
- Capitolo 3: Analisi Lessicale: Linguaggi regolari
- Capitolo 4: Analisi Sintattica: Linguaggi liberi
- Capitolo 5: Fondamenti (lezione conclusiva, come introduzione al corso di Informatica Teorica)

- Seguite le lezioni perché non tutto è sul libro, e in più a lezione faccio molti esempi ed esercizi in più di quanto ci sia sul libro.

Libri di testo complementari per il Modulo 1



Altri libri di testo complementari per il Modulo 1

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

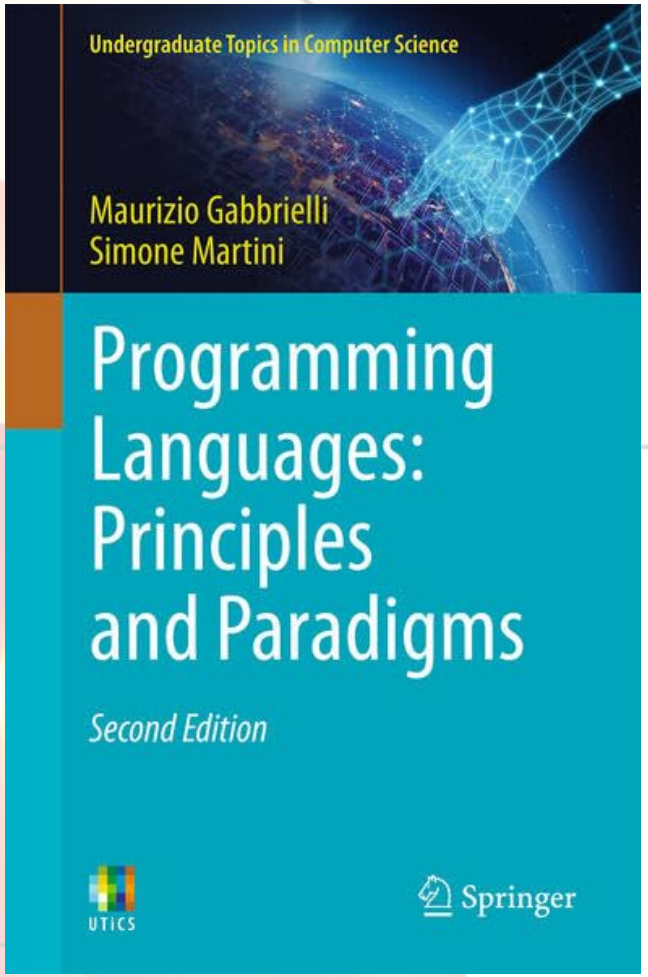
Compilatori Principi, tecniche e strumenti - 2/Ed. Pearson

Libro di testo (modulo 2 in inglese –ed. 2023)

Nuovo libro di testo del corso:

M. Gabbrielli, S. Martini, S. Giallorenzo

Lo trovate anche su Amazon



- Non contiene materiale per il modulo 1 su grammatiche, ma copre tutto il modulo 2 (inclusa la parte del prof. Giallorenzo)

Orario delle lezioni

Da Martedì 19 Settembre 2023 a Mercoledì13 Dicembre 2023

Ora inizio	lun	mar	mer	gio	ven
11:00-12:00		M2			
12:00-13:00		M2			
13:00-14:00		M2			
14:00-15:00					
15:00-16:00					
16:00-17:00			Magna		
17:00-18:00			Magna		

Cosa dovete sapere

- aspetti propriamente linguistici (quale paradigma, quali costrutti e perchè);
- come i costrutti linguistici possono essere implementati ed il relativo costo;
- aspetti architetturali che influenzano l'implementazione;
- tecniche di traduzione (compilazione).

Perché studiare i linguaggi in generale?

- Progettare un nuovo linguaggio?
  - non così raro: linguaggi specifici per classi di applicazioni
  - una GUI è un linguaggio per la sua applicazione
- Imparare più facilmente un nuovo linguaggio
  - costrutti già noti e meccanismi semantici di base
- Riassumendo: non imparare un linguaggio specifico ma imparare i principi generali dei linguaggi

Competenze da acquisire

- Competenza tecnica (scritto)
  - Esercizi
- Competenza linguistica (*orale*!)
  - correttezza terminologica
  - completezza di esposizione
  - Per aiutarvi, metterò sul sito del corso un elenco di domande frequenti che vengono fatte all'orale (per la mia parte)
- Competenza implementativa
  - come è implementato il meccanismo X?
  - collegamenti con i corsi di Architetture, Programmazione e Sistemi Operativi.
- Competenza critica
  - A language that doesn't affect the way you think about programming, is not worth knowing. (A.Perlis)

Come studiare

- Seguire *attivamente* le lezioni (domande, commenti, critiche)
- Studiare (subito) sul manuale
  - competenza tecnica
- Esercizi scritti
  - per verificare la competenza tecnica
- Ripetere (da soli o con altri)
  - per memorizzare
  - per migliorare la competenza linguistica
- Ri-studiare il manuale da capo
  - chiedersi:
    - come implemento questo meccanismo?
    - di quali informazioni ho bisogno per tale implementazione?
    - quanto costa questo meccanismo?
- Andare avanti e indietro sul manuale
  - quali collegamenti?
  - quali problemi?
  - competenza critica... *A language that doesn't affect the way you think about programming, is not worth knowing. (A.Perlis)*

Dai linguaggi hw ai meccanismi di astrazione

- Anni 1950--60:
  - Compilazione dei programmi per ottenere efficienza
  - Connessione diretta fra hw e linguaggi: integers, reals, goto
  - *Programmers cheap, machines expensive –> Keep the machine busy !*

- Oggi:
  - Compilazione di programmi costruiti in modo efficiente
  - Connessione diretta fra sw design e linguaggio: encapsulation, records, ereditarietà, dichiaratività
  - *Programmers expensive, machine cheap –> Keep the programmer busy!*

Evoluzione dei linguaggi

- Anni 40-50 Preistoria: (linguaggio macchina e assembler)
- Anni 50-60: Primi linguaggi alto livello:
  - FORTRAN (Backus 1957) per calcolo numerico-scientifico (su IBM704): uso di notazione matematica in espressioni (es. i+2*j)
  - ALGOL (58, 60, W) (Naur 1958) come linguaggio universale, per esprimere algoritmi: Indipendenza dalla macchina, vicinanza con la notazione matematica, call by name, funzioni ricorsive, type systems, data structures (anche dinamiche)
  - LISP (McCarthy 59-60) per intelligenza artificiale (List Processor): funzioni su una certa classe di espressioni simboliche (S-Expressions), ordine superiore (antesignano del paradigma funzionale)
  - COBOL (59-60) per applicazioni commerciali (record, archivi)
  - ALGOL 68 linguaggio barocco, complessa progettazione, ma contiene anche idee importanti (struttura a blocchi)
  - SIMULA (Nygaard & Dahl): da ALGOL, primo linguaggio con classi e oggetti (antesignano del paradigma object-oriented)

Evoluzione dei linguaggi II

- Anni 70:
  - PL/I (IBM): Fortran + COBOL, primo esempio di linguaggio multi-uso (PL deriva da Multi-Purpose Prog. Lang.) Scarso successo
  - Pascal (Wirth): Evoluzione (e semplificazione) di Algol W. Essenzialmente linguaggio didattico fino agli anni 80 (poi rimpiazzato da C e Java). Portabilità grazie a codice intermedio (P-Code)
  - C (Ritchie): Programmazione di sistema (Unix). Ha avuto notevole successo fino ai nostri giorni.
  - Prolog: (Colmerauer, Kowalski): uso esplicito della logica come base di un linguaggio di programmazione (primo linguaggio logico)
  - SmallTalk: Incapsulamento, oggetti e classi. O-O.
  - ML (Milner): Funzionale. Nato come Meta Linguaggio per un sistema semi-automatico di prova di proprietà dei programmi. Sistema di tipi evoluto.

Evoluzione dei linguaggi III

- Anni 80:
  - ADA: Linguaggio real-time di US Dep. Of Defense. Linguaggio per sviluppo di programmi molto grandi da parte di team. Include quasi tutto quello che esisteva al tempo, inclusi costrutti per il controllo della concorrenza
  - Postscript: page description language nell'area di desktop publishing elettronico
  - C++ Estensione Object Oriented di C
  - Standard ML
  - CLP (linguaggi logici con vincoli). Linguaggi che permettono di manipolare relazioni su opportuni domini. Usati per problemi combinatori, ottimizzazione, intelligenza artificiale etc.

Evoluzione dei linguaggi IV

- Anni 90:
  - Java (Goslin): Altamente portabile; object-oriented; programmi spediti sulla rete (applet). Elevata portabilità (Write once, run anywhere!) --> Java Virtual Machine e ByteCode
  - PERL: Linguaggio per applicazioni di text-processing nato come linguaggio di shell-scripting in UNIX
  - HTML linguaggi di marcatura per ipertesti (usato per creare pagine web)
  - XML linguaggio per la rappresentazione di dati semi-strutturati (ad esempio, pagine web).

Evoluzione dei linguaggi V

- Anni dal 2000 in poi:
  - service-oriented computing: paradigma che usa i "servizi" (di solito web services), come unità di base di computazione, per progettare ed implementare applicazioni integrate di business.
  - WSDL: web services description lamguage
  - WS-BPEL: Web Services Business Process Execution Language
  - **Jolie *(sviluppato a Bologna!)***
  - **Agent Based Programming** (vedi corso di Sacerdoti Coen in LM) Go, Erlang, Rust, ecc..
  - **Scalable Cloud Programming** (vedi corso di Zavattaro) – Scala e Spark

Linguaggi imperativi e linguaggi dichiarativi

- Linguaggi imperativi
- Basati sulla nozione di stato (insieme di locazioni di memoria contenenti dei valori)
- Le istruzioni sono **comandi** che cambiano lo stato.
- Es. X:= X+1; X:= X+2 …
- C, Pascal, FORTRAN, COBOL …
- Analogia con le frasi *imperative* dove il soggetto è implicito (e.g. taglia quella mela)

- Linguaggi dichiarativi
- Basati sulla nozione di funzioni o relazione.
- Le istruzioni sono **dichiarazioni** di nuovi valori, in modo diretto oppure per composizione di funzioni o relazioni.
- Es. (fun(X). X+2) 3;
- Analogia con le frasi *dichiarative* (e.g., quella mela è tagliata)

Linguaggi dichiarativi

- Linguaggi funzionali: Lisp, Scheme, ML, Haskell
  - Basati sulla nozione di funzione: risultato del programma = valore esplicito di una espressione
  - applicazione e definizione di funzioni, ricorsione
  - Programmare = costruire la funzione che calcola il risultato

- Linguaggi logici (e con vincoli): Prolog, CLP
  - Basati su relazioni: risultato = valore di alcune variabili determinato da alcune relazioni
  - Istruzioni = implicazioni logiche fra opportune formule, che possono essere viste come regole di riscrittura
  - Programmare = specificare la relazione che definisce il valore delle variabili di interesse (sostanzialmente ``dire come è fatta la soluzione'')
-

Un semplice esempio: calcolo del Fattoriale

PASCAL *(imperativo)*

```
function Fatt (n: integer): integer;
    var
        i, acc: integer;
    begin
        acc:=1;
        for i:=1 to n do
                acc:=acc*i;
        Fatt:=acc;
    end;
```

Un semplice esempio: calcolo del Fattoriale (2)

SCHEME *(funzionale)*

```
(define (fatt n)
    (cond [(= n 0)  1]
          [else  (*  n  (fatt (- n 1)))]
    ))
```

Un semplice esempio: calcolo del Fattoriale (3)

PROLOG *(logico)*

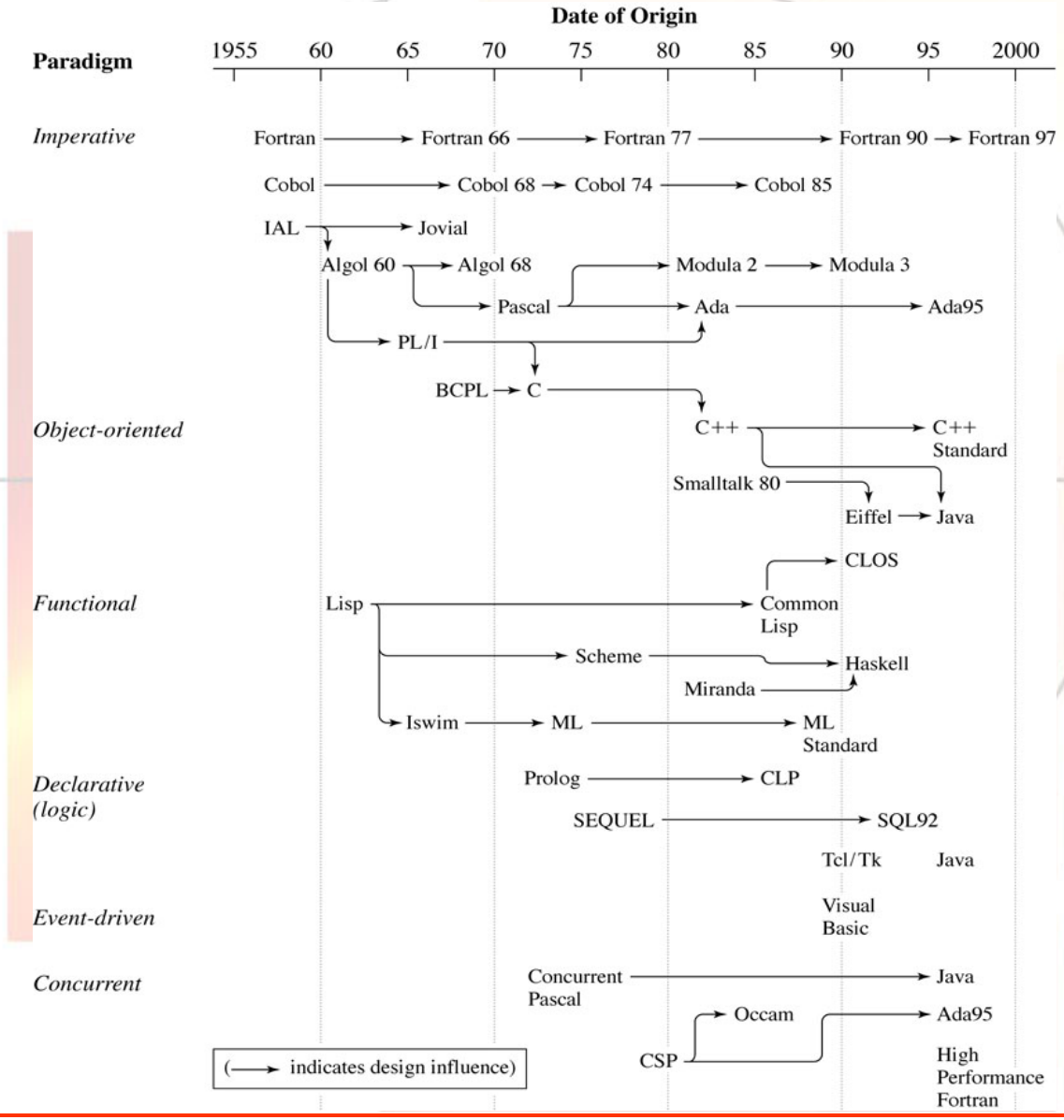
```
Fatt(0, 1).
Fatt(N, Z) :- Fatt(N-1, Y), Z is Y*N.
                                Prod(Y, N, Z).

Prod(0, X, 0).
Prod(N+1, X, Z) :- Prod(N, X, Y), Sum(Y, X, Z).
Sum(0, X, X).
Sum(Y+1, X, Z+1) :- Sum(Y, X, Z).
```

Linguaggi orientati agli oggetti

- Linguaggi orientati agli oggetti: Java, C++, Smalltalk
  - linguaggi imperativi con alcune metodologie di progettazione dichiarative
  - Oggetti (istanze di opportune classi) che contengono i dati (concetti imperativi) e metodi come funzioni per operare su tali oggetti (concetti dichiarativi)
  - Incapsulamento
  - Ereditarietà
  - … molto altro più avanti

Storia dei linguaggi e dei paradigmi -- Timeline



Link utili – da Wikipedia English

- Per un elenco di (quasi) tutti i linguaggi di programmazione esistenti

  https://en.wikipedia.org/wiki/

  List_of_programming_languages

  (non è possibile conoscerli tutti!)

- Per una timeline piu' articolata e completa

  https://en.wikipedia.org/wiki/

  Timeline_of_programming_languages

- Per un confronto tra le dozzine di linguaggi di programmazione …..

  https://en.wikipedia.org/wiki/

  Comparison_of_programming_languages

Come confrontare linguaggi?

Caratteristiche intrinseche (dipendenti dalla sua sintassi e classe di primitive che mette a disposizione):

- Espressività: cosa/quanto può calcolare? Turing-completezza come criterio assoluto per i linguaggi sequenziali
- Didattica: quanto sono semplici le primitive del linguaggio e quanto velocemente lo si può apprendere
- Leggibilità: quanto è facile, leggendo un programma, capire cosa fa.
- Robustezza: la capacità del linguaggio di prevenire, nei limiti del possibile, gli errori di programmazione (ad esempio, perché è tipato).

Come confrontare linguaggi? (2)

Altre caratteristiche intrinseche :

- Generalità: la facilità con cui il linguaggio si presta a codificare algoritmi e soluzioni di problemi in campi diversi. Linguaggi domain-specific vs general purpose.
- Efficienza: la velocità di esecuzione e l'uso oculato delle risorse del sistema su cui il programma gira. In genere i programmi scritti in linguaggi molto astratti tendono ad essere lenti e voraci di risorse; in compenso facilitano molto la vita del programmatore, accelerando lo sviluppo di nuovi programmi ed eliminando intere classi di errori di programmazione possibili. Viceversa un linguaggio meno astratto ma più vicino alla reale struttura di un computer genererà programmi molto piccoli e veloci ma a costo di uno sviluppo più lungo e difficoltoso.

Come confrontare linguaggi? (3)

Caratteristiche esterne (dipendenti dall'ambiente di lavoro)

- Diffusione: più è numerosa la comunità per un linguaggio, tanto più è facile trovare materiale, aiuto, librerie di funzioni, documentazione, consigli. Inoltre ci sono un maggior numero di software house che producono strumenti di sviluppo per quel linguaggio, e di qualità migliore.
- Standardizzazione: un produttore sente spesso la tentazione di introdurre delle variazioni sintattiche o delle migliorie ad un linguaggio, originando un *dialetto* del linguaggio e fidelizzando così i programmatori al suo prodotto: ma più dialetti esistono, più la comunità di programmatori si frammenta in sottocomunità più piccole e quindi meno utili. Per questo è importante l'esistenza di uno standard ufficiale, di solito definito dall'ANSI (American National Standards Institute) o dall'ISO (International Organization for Standardization).

Come confrontare linguaggi? (4)

Altre caratteristiche esterne

- Integrabilità: dovendo scrivere programmi grandi, è facile trovarsi a dover integrare parti di codice precedente scritte in altri linguaggi: se un dato linguaggio consente di farlo facilmente, questo è decisamente un punto a suo favore.
- Portabilità: la possibilità che portando il codice scritto per una certa **piattaforma** (CPU + architettura + sistema operativo) su un'altra piattaforma, questo funzioni subito, senza doverlo modificare. Questa è, ad esempio, una delle motivazioni principali per l'introduzione del linguaggio Java (negli anni '90), che è altamente portabile grazie alla Java Virtual Machine (JVM).