Paradigmi e linguaggi di programmazione

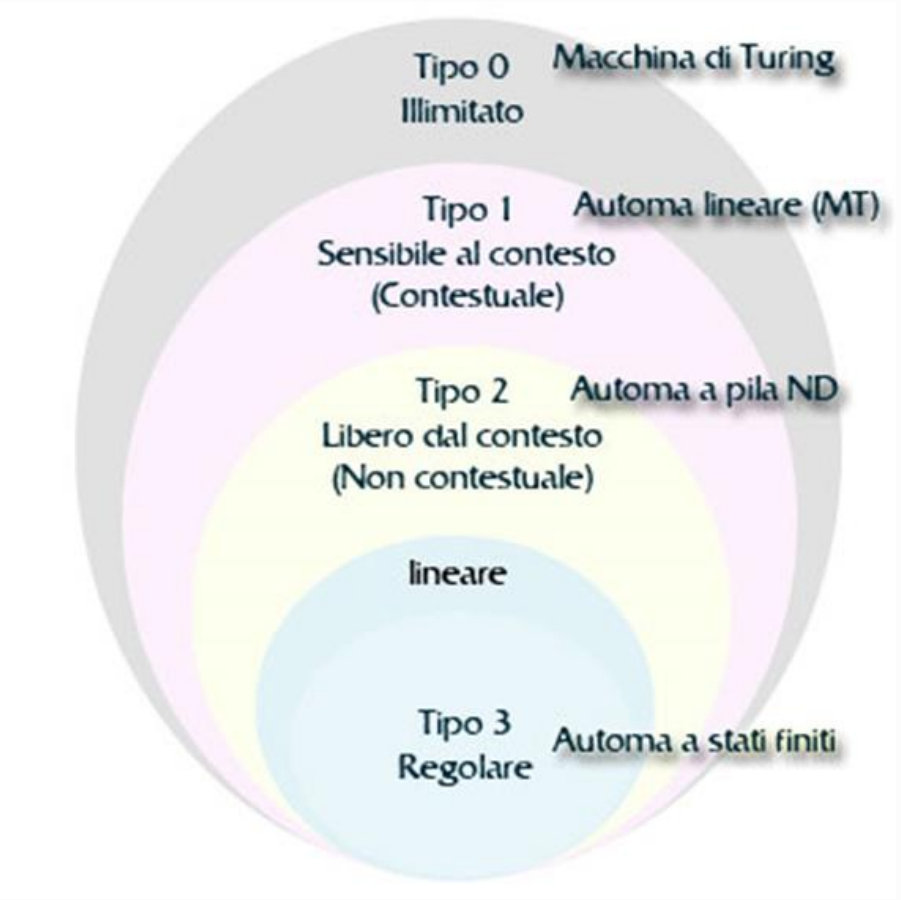
Linguaggi formali e modelli per l'informatica

Linguaggi formali

- **Automi**
  - Approccio riconoscitivo

Automi

- Sono l'altro principale approccio per generare linguaggi formali
- Se le grammatiche sono ***modelli dichiarativi***
- Gli automi sono i corrispondenti ***modelli procedurali***



Oss. Gli automi, originariamente, furono proposti per creare un modello matematico che riproducesse il funzionamento del cervello

Automi

- La distinzione tra procedurale e dichiarativo è una distinzione introdotta negli anni 70 a proposito
- a) dei sistemi di rappresentazione della conoscenza
- e (b) dei linguaggi di programmazione

(a) Secondo lo psicologo cognitivo John Robert Anderson (*Language, memory, and thought*, Hillsdale, N.J., Erlbaum Associates, 1976) – che si richiama all'impostazione simbolica proposta da Allen Newell (*Human problem solving*, con Herbert A. Simon, Englewood Cliffs, N.J., Prentice-Hall, 1972), uno dei padri riconosciuti dell'intelligenza artificiale – possiamo ripartire le conoscenze umane in due tipi di rappresentazioni tra loro irriducibili: le rappresentazioni *dichiarative* e le rappresentazioni *procedurali*. Le conoscenze dichiarative possono essere espresse con asserzioni vere o false, mentre le conoscenze procedurali sono quelle su cui si basa l'esecuzione di un certo compito: una conoscenza *dichiarativa* è conoscere 'che,' mentre una conoscenza *procedurale* è conoscere 'come.' I modelli di rappresentazione dichiarativi fanno riferimento a rappresentazioni di oggetti ed eventi e alle relazioni tra tali oggetti ed eventi ed altri oggetti ed eventi; essi si concentrano sul 'perché' piuttosto che sul 'come.' I modelli di rappresentazione procedurali, invece, si concentrano sui compiti che debbono essere eseguiti per raggiungere un particolare obiettivo; essi considerano il 'come' piuttosto che il 'perché.'

http://web.dfc.unibo.it/buzzetti/IUcorso2007-08/mdidattici/dichiarativo-procedurale.pdf

Automi

- La distinzione tra procedurale e dichiarativo è una distinzione introdotta a proposito
- a) dei sistemi di rappresentazione della conoscenza
- e (b) dei linguaggi di programmazione

(b) La programmazione *procedurale* è spesso assimilata alla programmazione *imperativa* che, a sua volta, si distingue nettamente dalla programmazione *dichiarativa*:

> i programmi *imperativi* [o *procedurali*] forniscono una specificazione esplicita dell'algoritmo che porta ad un certo risultato, mentre i programmi *dichiarativi* presentano una specificazione esplicita del risultato, affidando l'implementazione dell'algoritmo al software d'appoggio [imperative programs explicitly specify an algorithm to achieve a goal, while declarative programs explicitly specify the goal and leave the implementation of the algorithm to the support software], *Wikipedia* (English), *sub voce* Declarative programming, http://en.wikipedia.org/wiki/Declarative_programming.

Si parla quindi, rispettivamente, di linguaggi di programmazione *dichiarativi* e di linguaggi di programmazione imperativi o *procedurali*: un linguaggio dichiarativo specifica 'che cosa' si vuole ottenere, un linguaggio procedurale specifica 'come' lo si può ottenere; un linguaggio dichiarativo formula asserzioni vere o false, un linguaggio procedurale formula istruzioni o comandi.

http://web.dfc.unibo.it/buzzetti/IUcorso2007-08/mdidattici/dichiarativo-procedurale.pdf

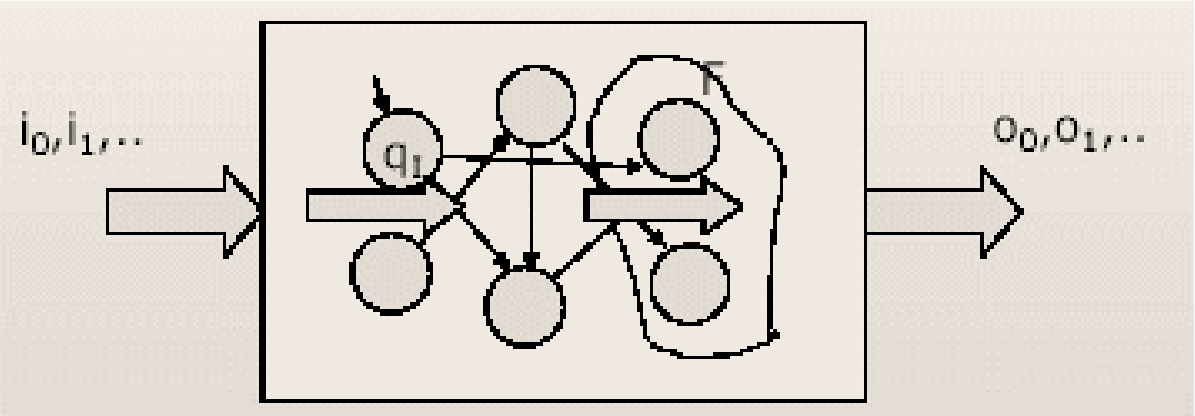
Automi

- Un automa è un dispositivo, o un sistema in forma di macchina sequenziale, che, ad ogni istante, può trovarsi in un determinato “stato”
- Lo scopo dello stato è quello di ricordare la parte rilavante della storia del sistema.

- Schema di computazione
  - input: in In con I insieme finito
  - output: in Om con O insieme finito
  - algoritmo: consuma gli input e produce gli output

Automi a stati finiti (ASF)

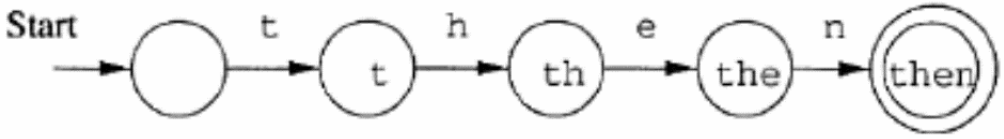
- ASF può trovarsi in un numero finito di stati diversi come conseguenza di qualche ingresso, che può variare su un insieme finito di valori.
- Esegue una transizione da uno stato all'altro attraverso una funzione di cambiamento di stato



Automi a stati finiti (ASF)

- Schematizzando:
  - Input: sequenza (finita) di simboli
  - Output: sequenza (finita) di simboli
  - Computazione:
    - si parte da uno stato iniziale
    - si consuma un simbolo di input
    - sulla base di input e stato corrente si produce un simbolo di output e ci si muove su un nuovo stato
    - si procede finché non si raggiunge uno stato finale

Es.: Un automa finito che riconosce la parola **then**



Automi a stati finiti

- Si riescono a rappresentare
  - semplici trasformazioni di sequenze
  - riconoscitori di linguaggi regolari (Tipo 3)
- Non si riesce a rappresentare
  - se in una espressione, le parentesi sono bilanciate
  - riconoscitori di linguaggi context-free (Tipo 2)

- *Infatti il numero di stati è fissato a priori (memoria finita) e quindi non consente di trattare quei problemi che richiedono un "conteggio" di elementi senza limite fissato*

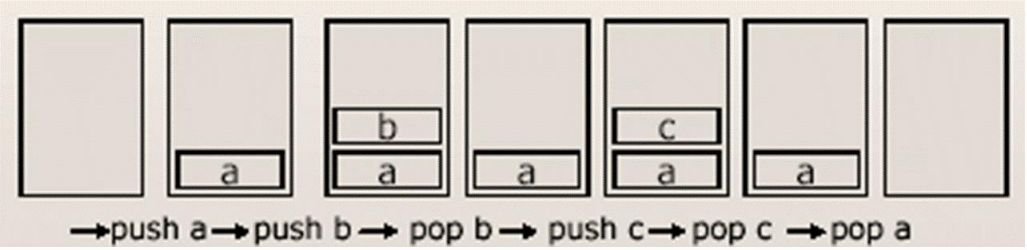
  - es.: così un ASF può riconoscere
    - L={c,acb,aacbb,aaacbbb}
    - ma poi non riconosce "aaaacbbbb"...
  - rendere il numero degli stati infinito non è praticabile

Automi a pila (PDA:Push Down Automata)

- E' un elaboratore con
  - un insieme finito di stati (possibili configurazioni interne)
  - input ed output: sequenze (finita) di simboli
  - ***una struttura dati di supporto*** dove scrivere e leggere informazioni temporanee, detta a ***pila***

Struttura a pila o stack o lista FILO (First in Last Out)

- si inseriscono dei simboli (operazione PUSH)
- si leggono in ordine inverso (operazione POP)



Automi a pila (PDA:Push Down Automata)

Funzionamento:

- si parte da uno stato iniziale
- si accetta un simbolo di input
- sulla base di
  - (input, stato, elementi in testa della pila)
- si ottiene
  - (output, nuovo stato, nuova testa della pila)
- usando una funzione di transizione di stato
- si procede finché non si raggiunge uno stato finale

Automi a pila (PDA:Push Down Automata)

- *La pila come struttura d'appoggio per memorizzare informazioni parziali*
  - es.: per contare il numero dei simboli
- Si riesce a rappresentare
  - trasformazioni di sequenze tenendo traccia dei simboli
    - invertire una sequenza di simboli
  - riconoscitori di linguaggi context-free (Tipo 2)
- Non si riesce a rappresentare
  - riconoscitori di linguaggi context-dependent (Tipo 1)
    - la pila, ha un accesso limitato che si limita alla testa della pila
      - es.: come accedere al primo degli elementi inseriti?
      - bisogna disporre di una struttura dati più flessibile

Pila (stack) e programmazione

Utilizzo non solo in ***informatica teorica*** …:

1. la struttura dati a stack è un ***tipo di struttura dati*** che un programma può implementare e utilizzare per il proprio funzionamento;
2. lo stack è un ***elemento dell'architettura dei moderni processori***, e fornisce il supporto fondamentale per l'implementazione del concetto di subroutine
3. le macchine virtuali di quasi tutti i linguaggi di programmazione ad alto livello usano uno ***stack dei record di attivazione per implementare il concetto di subroutine*** (generalmente, ma non necessariamente, basandosi sullo stack del processore);
4. la ***gestione di più versioni dello stesso software nel medesimo sistema operativo***. Per installare, occorre partire in ordine cronologico, dalla più vecchia alla più recente; per disinstallare, occorrerà seguire l'ordine opposto, dalla più recente alla più vecchia.

Pila (stack) e overflow

uno stack overflow avviene quando è richiesto l'uso di una quantità troppo elevata di memoria nello stack*

- La causa più comune di uno stack overflow è una ricorsione con profondità eccessiva o infinita.

- I linguaggi che implementano la tecnica ***tail recursion***, come ad esempio i linguaggio funzionali, permettono una particolare ricorsione infinita che può essere eseguita senza stack overflow.
  - Questo avviene poiché le chiamate che fanno uso di tail-recursion non richiedono uno spazio aggiuntivo nello stack

* *Errore comune tra i programmatori se ha dato il nome a*

a community of 6.9 million programmers, just like you, helping each other.

Pila (stack) e RPN (Reverse Polish Notation - notazione polacca inversa)

Utilizza la notazione post fissa (cioè gli operatori si trovano tutti a sinistra degli argomenti) 3 + 2 ⇔ 3 2 +

- Si usa uno stack su cui si accumulano gli operandi: prima si impila il 3, poi il 2.
- Un operatore invece preleva dalla cima della pila tutti gli operandi di cui ha bisogno, esegue l'operazione, e vi ri-deposita il risultato.
- Se l'espressione completa è corretta, alla fine di tutte le operazioni sulla pila si avrà un solo elemento, il risultato finale.

- Deve conoscere in anticipo il numero di argomenti (operandi) che richiede la funzione (o operazione).
- Ha il vantaggio di eliminare i problemi dovuti alle parentesi e alla precedenza degli operatori .

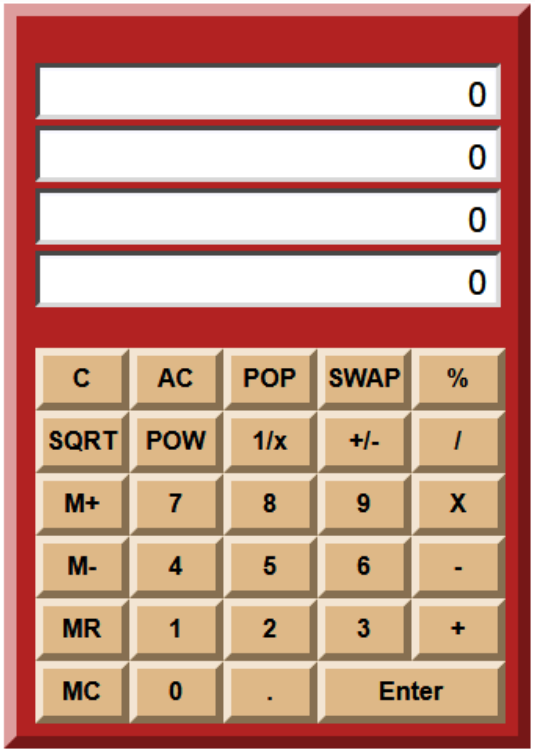
- Alcune calcolatrici scientifiche (Hewlett Packard negli anni 80-90) utilizzano (utilizzavano) la RPN in quanto evita l'annotazione di risultati intermedi durante le operazioni.
- La RPN fu così chiamata per analogia con la notazione polacca, inventata da Łukasiewicz (filosofo logico polacco 1878-1956)

Pila (stack) e RPN (reverse polish notation - notazione polacca inversa)

Utilizza la notazione post fissa e fa uso di stack

- Esempi:
  - 5 + (10 * 2) → 5 10 2 * +
  - ((10 * 2) + (4 - 5)) / 2 →
  - 10 2 * 4 5 - + 2 /
  - (7 / 3) / ((1 - 4) * 2) + 1 →
  - 1 7 3 / 1 4 - 2 * / +

oppure 7 3 / 1 4 - 2 * / 1 +



http://www.alcula.com/calculators/rpn/

Pila (stack) e RPN (reverse polish notation - notazione polacca inversa)

Algoritmo per la trasformazione da notazione infissa a RPN

- Strutture dati
- Stringa di input = formula in notazione infissa.
- Stringa di output = formula convertita in notazione polacca inversa (postfissa).
- Stack = Pila temporanea (da usarsi solo per gli operatori). Risulterà vuota a fine elaborazione.

- Si effettua **l'analisi lessicale** dell'espressione in forma infissa allo scopo di individuare i singoli componenti (operandi, operatori e parentesi), procedendo sequenzialmente da sinistra a destra.
- Si aggiunge ogni operando incontrato alla fine della stringa di output.
- Ogni volta che incontriamo un operatore, lo confrontiamo con quello sulla testa della pila. Se la sua priorità è maggiore lo inseriamo nella pila (push), altrimenti togliamo dalla pila (pop) tutti gli operatori di priorità minore (o uguale) e li accodiamo nell'espressione di ritorno.

Pila (stack) e RPN (reverse polish notation - notazione polacca inversa)

Algoritmo per la trasformazione da notazione infissa a RPN

- Ogni volta che incontriamo una parentesi aperta la inseriamo nella pila.
- Ogni volta che incontriamo una parentesi chiusa accodiamo in output espressione tutto quello che c'è sulla pila fino alla prima parentesi aperta

Infix	Stack	Postfix
A*B-(C+D)+E	#	
*B-(C+D)+E	#	A
B-(C+D)+E	# *	A
-(C+D)+E	# *	A B
(C+D)+E	# -	A B *
C+D)+E	# - (	A B *
+D)+E	# - (	A B * C
D)+E	# - (+	A B * C
)+E	# - (+	A B * C D
+E	# -	A B * C D +
E	# +	A B * C D + -
	# +	A B * C D + - E
	#	A B * C D + - E +

Pila (stack) e RPN (reverse polish notation - notazione polacca inversa)

- RPN ha dei vantaggi?

- a parte considerazioni che, con l'evolversi della tecnologia, sono divenute irrilevanti, il vero vantaggio è visibile quando si eseguono dei calcoli non partendo da una forma scritta, ma costruendo un ragionamento mano a mano che li si esegue

http://sarmaivero.blogspot.it/2012/02/rpn-la-notazione-polacca-inversa.html

Pila (stack) e RPN (reverse polish notation - notazione polacca inversa)

- Esempio: Sono in macchina e mi viene la curiosità di calcolare l'energia cinetica accumulata dal veicolo.

1. Guardo il tachimetro e scrivo **125** (siamo in autostrada) **1000 * 3600 /** per avere la velocità in m/sec.
2. Butto un occhio sul display, quasi trentacinque metri al secondo, non male.
3. Pigio **X²** per elevare al quadrato, poi prendo il libretto della macchina e scrivo 1850,
4. chiedo a mia moglie che sta guidando, quanto pesa, scrivo **54** e **+** per sommare,
5. scrivo il mio peso **95** e **+** per sommare,
6. stimo il peso degli abiti che indossiamo, scrivo **5** e **+** per sommare.
7. Smanetto un po' col computer di bordo e scopro che ci sono 35 litri di gasolio nel serbatoio. Scrivo **35**. Un litro di gasolio peserà meno di un chilogrammo, quanto esattamente non lo so, per cui tiro ad indovinare. Scrivo **0.8**, ***** per moltiplicare
8. e scopro che ci stiamo portando dietro 28 Kg di gasolio; pigio **+** per sommare.
9. A questo punto sul display dovrei avere il peso totale del veicolo e penso: "cavoli, più di due tonnellate!" Però non è il peso che mi interessa, è l'energia. scrivo * per moltiplicare ed ottenere il doppio dell'energia,
10. **2 /** per averla in Joule ed ecco fatto.

http://sarmaivero.blogspot.it/2012/02/rpn-la-notazione-polacca-inversa.html

Pila (stack) e RPN (reverse polish notation - notazione polacca inversa)

Se avete seguito con attenzione, avrete notato due cose:

a. se anche avessi deciso di sommare al peso del veicolo quello del contenuto dopo aver scritto il primo, non sarebbe cambiato nulla;
b. ho sempre avuto sott'occhio i valori intermedi e questi sono sempre stati valori logici, se ci fosse stato un errore marchiano, quindi, me ne sarei accorto.

in notazione algebrica invece:

http://sarmaivero.blogspot.it/2012/02/rpn-la-notazione-polacca-inversa.html

Pila (stack) e RPN (reverse polish notation - notazione polacca inversa)

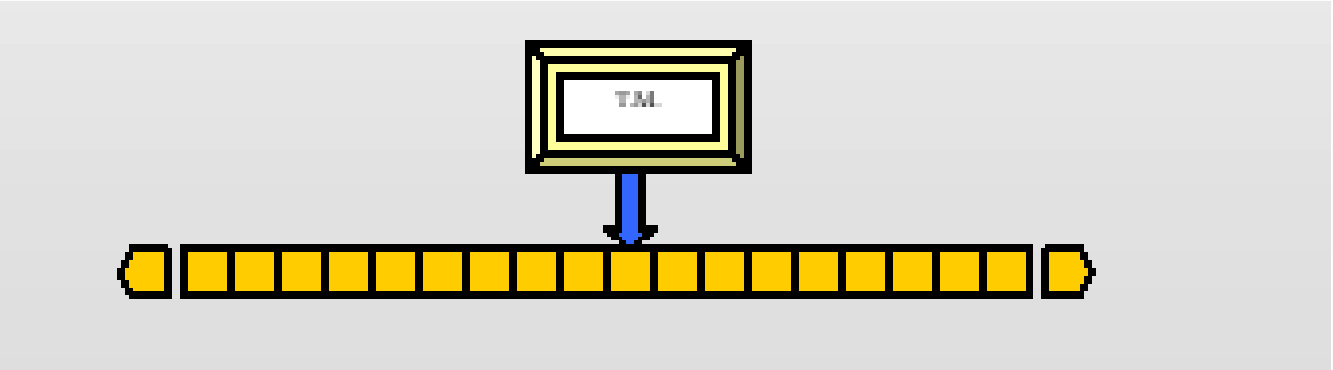
1. Avrei scritto 125/1000*3600=, per avere la velocità in m/sec.
2. Elevo al quadrato ed inizio a moltiplicare per il peso, scrivendo X^2*.
3. A questo punto, o ho già chiaro che devo calcolare il peso dell'auto come una somma, e apro una parentesi, o inserisco il peso del libretto e, a questo punto, non posso più tornare indietro.
4. Diciamo che sono una persona previdente, apro una parentesi ed inizio a sommare, fino a che non arrivo al gasolio, cioè (1850+54+95+5+.
5. Decido, prima di scrivere i litri, che per convertirli in Kg servirà una moltiplicazione, quindi, vista la precedenza degli operatori, non ho bisogno di ulteriori parentesi, digito 35*0,8;
6. chiudo la parentesi della somma e divido per due, cioè)/2=

Più semplice? Più complicato? No, nè più semplice nè più complicato, solo, in questo secondo caso, per decidere cosa digitare, ho dovuto fare delle ***considerazioni sulle operazioni successive****, nel primo caso no. Irrilevante quando effettuiamo dei calcoli partendo da una equazione scritta ma, come già detto, significativo quando eseguiamo i calcoli mano a mano che costruiamo un ragionamento.*

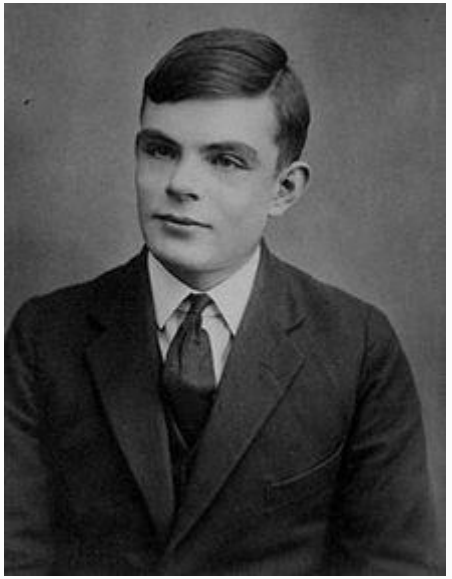
http://sarmaivero.blogspot.it/2012/02/rpn-la-notazione-polacca-inversa.html

La macchina di Turing

- E’ un elaboratore con
  - un insieme finito di stati (possibili configurazioni interne)
  - un nastro illimitato con testina:
    - read/write dell’elemento sotto la testina,
    - spostamento testina LEFT/RIGHT

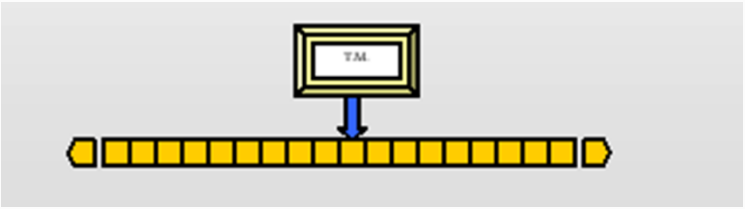


La macchina di Turing



- Alan Mathison Turing (Londra, 23 giugno 1912 – Wilmslow, 7 giugno 1954) è stato un matematico, logico e crittografo britannico
- lavorò a Bletchley Park, il principale centro di crittoanalisi del Regno Unito, dove ideò una serie di tecniche per violare i cifrari tedeschi, incluso il metodo della Bomba, una macchina elettromeccanica in grado di decodificare codici creati mediante la macchina Enigma.

La macchina di Turing



Funzionamento

- si parte da uno stato iniziale
- sul nastro è scritto il valore dell'input
- si legge il simbolo sotto la testina
- sulla base di un input (stato) si ottiene un valore da scrivere sul nastro, nuovo stato
- si procede finché non si raggiunge uno stato finale

- Il nastro come luogo dove
  - trasformare l'input in output
  - scrivere valori d'appoggio - risultati intermedi

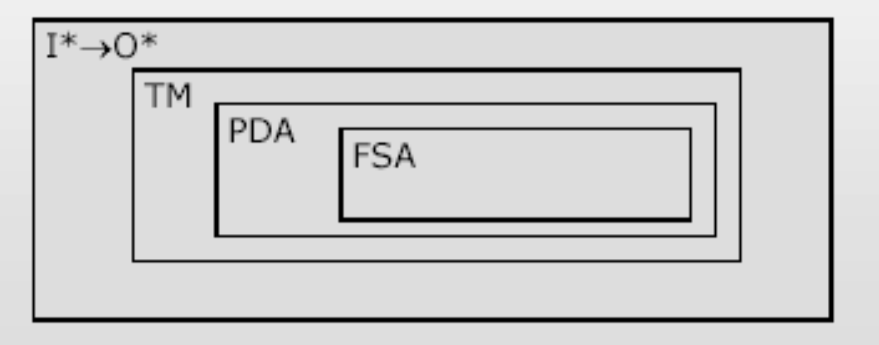
La macchina di Turing

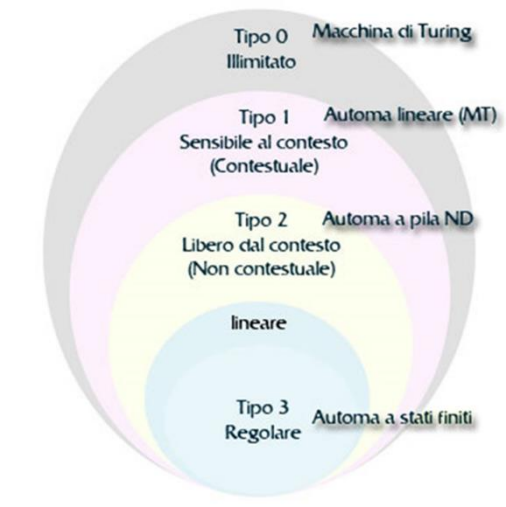
- Si ha sufficiente espressività per agire sul nastro in ogni modo
  - copiare pezzi di nastro da una parte all'altra
  - sovrascrivere pezzi di nastro
  - "cercare" sotto-sequenze di simboli su e giù per il nastro
- tuttavia la funzione di cambiamento di stato diventa brevemente molto complessa!
  - ma in ***linea di principio è ottenibile***...

La macchina di Turing è un formalismo potente (si riconoscono frasi di linguaggio di tipo 1) ma non semplice da utilizzare per algoritmi complessi

Gerarchia fra gli automi

- Ogni automa realizza una funzione F che parte da un dato input per arrivare ad un output finale
- Esiste una gerarchia tra gli automi
  - ogni FSA è anche un PDA (che non usa la pila)
  - ogni PDA è anche una TM (usa il nastro come pila)





Automi: commenti

- Gli ***automi a stati finiti*** sono ragionevolmente semplici da gestire
  - produrre, leggere, analizzare
- Sono in effetti strumenti usati nell'ingegneria del S/W
  - per specificare e progettare semplici sistemi come interfacce grafiche, piccoli agenti, etc..

i PDA sono in genere un pò complicati ma sono importanti per la loro rilevanza concettuale

Turing ha notevoli relazioni con gli H/W moderni

- lettura/scrittura simboli vs. I/O da RAM/ROM
- transitare stato vs. cambiare stato CPU
- agire sul nastro vs. accedere alla memoria

La risoluzione meccanica di problemi

- **La risoluzione meccanica di problemi**
  - Tesi di Church e MT universali
    - ***Problemi risolubili e irrisolubili***

La risoluzione meccanica di problemi

- Gli automi, le grammatiche e gli altri formalismi si possono considerare dispositivi "***meccanici***" per la risoluzione di un problema

- Spesso un problema matematico costituisce la formalizzazione di un qualche problema pratico non matematico

  - Es: verifica appartenenza di una stringa ad un linguaggio=> riformulazione formale di problemi pratici quali il riconoscimento di istruzioni in programmi scritti in un linguaggio di programmazione

La risoluzione meccanica di problemi

- Alcuni formalismi (modelli matematici) sono più potenti di altri
  - Es. Riconoscono linguaggi che altri non riescono ad accettare

- Se neanche la macchina “più potente” riesce a risolvere un problema, potrebbero esserci ***problemi non risolubili***

La risoluzione meccanica di problemi

- La teoria della computabilità
  - Tutti i formalismi esaminati finora sono *discreti*
    - *Dotati di domini matematici numerabili, definiti in modo finito.*
  - Concetto di algoritmo
    - Procedura di risoluzione di un problema mediante dispositivo automatico di calcolo
    - Metodo astratto di descrizione dei programmi per calcolatore: una sequenza di comandi che, se eseguita, risolve un determinato problema (dipende dal linguaggio e dalla macchina)

La risoluzione meccanica di problemi

- Computazione algoritmica
  - trasformazione di un input a rappresentazione finita
  - in un output a rappresentazione finita
  - mediante un algoritmo (sequenza finita di azioni elementari)

- **Problema**
  - **quali funzioni sono esprimibili tramite algoritmi?**

La risoluzione meccanica di problemi

- ***Tesi di Church-Turing (1936)***
  - Non esiste alcun formalismo, per modellare una determinata computazione meccanica, che sia più potente della Macchina di Turing e dei formalismi ad essi equivalenti
  - Ogni algoritmo può essere codificato in termini di MT (o di formalismo equivalente)

  - *Tesi intrinsecamente non dimostrabile, poichè basata sulla notazione intuitiva di calcolo meccanico: essa è sostenuta solo dall'esperienza precedente e dall'evidenza intuitiva*

La risoluzione meccanica di problemi

- ***Storia***
  - Il lavoro dei due sull'argomento prese avvio per risolvere problema di decisione sollevato da David Hilbert.
    - In sostanza, Hilbert si chiedeva se potesse esistere un algoritmo che potesse decidere circa la verità o la falsità di qualsiasi enunciato matematico.
  - Indipendentemente, e per mezzo di strade molto diverse, **nel 1936** essi arrivarono agli stessi risultati.
  - Turing attraverso il concetto di macchina di Turing appunto
  - Church attraverso al definizione di lambda calcolo

La risoluzione meccanica di problemi

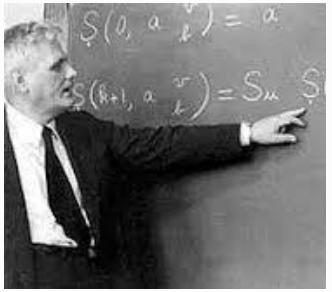
- ***Hilbert e il teorema di finitezza***
  - David Hilbert (1862 – 1943) è stato un matematico tedesco
  - Iniziatore del ***concetto della metateoria*** in campo matematico.
  - Dimostrò il *teorema di finitezza di Hilbert*: un metodo per dimostrare che esiste un insieme di generatori finito per un numero di variabili qualsiasi, ma in forma totalmente astratta: ***pur dimostrandone l'esistenza, non si fornisce nessun procedimento che permetta di costruirlo***.
  - Paul Gordan, l'esperto sulla teoria degli invarianti per i *Mathematische Annalen*, non riuscì ad apprezzare il rivoluzionario teorema di Hilbert e rifiutò un suo articolo, criticandone l'esposizione, a suo dire poco esaustiva. Il suo commento fu:
    - Questa è Teologia, non Matematica!
  - Più tardi, dopo che l'utilità del metodo di Hilbert fu universalmente riconosciuta, lo stesso Gordan ebbe a dire:
    - Debbo ammettere che anche la teologia ha i suoi pregi.

Da wikipedia

La risoluzione meccanica di problemi

- ***Church e λ-calcolo***

  - Alonzo Church (1903 – 1995) è stato un matematico e logico statunitense.



  - λ-calcolo è un sistema formale da lui definito e sviluppato per analizzare formalmente le funzioni e il loro calcolo. Le prime sono espresse per mezzo di un linguaggio formale, che stabilisce quali siano le regole per formare un termine, il secondo con un sistema di riscrittura, che definisce come i termini possano essere ridotti e semplificati.
  - nella teoria della calcolabilità fu sviluppato per lo studio delle funzioni ricorsive come effettivamente calcolabili.
  - nella teoria dei linguaggi di programmazione è il principale prototipo dei linguaggi di programmazione di tipo funzionale e della ricorsione.

Tesi Church e TM

se un problema è intuitivamente calcolabile, allora esisterà una macchina di Turing (o un dispositivo equivalente, come il computer) in grado di risolverlo (cioè di calcolarlo)

Più formalmente possiamo dire che la classe delle funzioni calcolabili coincide con quella delle funzioni calcolabili da una macchina di Turing.

- **Quale completezza?**
  - Quale rapporto fra TM e funzioni?
  - Esistono funzioni non realizzabili da TM?

Tesi Church e TM

- **Incompletezza xchè**
- Le TM, che operano su insiemi discreti, sono una infinità numerabile (aritmetizzazione di Goedel)
  - Ma fissato un dominio D e un codominio C numerabili, ad esempio entrambi N (numeri naturali)
  - L'insieme delle funzioni da N a N non è numerabile, è una infinità non numerabile!!
- *Quindi*
  - ***Esistono delle funzioni da N a N che non è possibile realizzare tramite un algoritmo della TM***
    - ***Problema:* Quali si possono realizzare e quali no?**

Tesi Church e TM

Si dimostra che le TM realizzano tutte e sole le funzioni ricorsive parziali (FRP)

Esse sono date da:

- Funzioni base:
  - fz(x)=0, fs(x)=x+1, fpi(x1,..,xn)=xi
  - sono le più semplici funzioni aritmetiche
- Quelle ottenibili da queste tramite:
  - *Composizione*
  - *Ricorsione*
  - *Parzialità*

tenere conto di computazioni fallite: con un certo input una TM potrebbe non terminare mai perché entra in loop: parzialità ***trucco formale*** che consente di costruire una funzione che per uno o più elementi non ritorna alcun valore

Tesi Church e TM

- Quali sono le Funzioni Ricorsive Parziali?
  - la totalità delle funzioni matematiche “discrete” (su N,Q,..)
    - (es.: quelle dell’analisi, dell’aritmetica, probabilità...)
  - le funzioni d’accesso e manipolazione sequenze
    - (ricerca, inserimento, ordinamento,...)
  - il riconoscimento di linguaggi (di tipo 1)

Tesi Church e TM

- Tuttavia:
  - sappiamo che le funzioni delle TM sono numerabili (tante quanti sono i naturali N)
  - la totalità delle funzioni esistenti sono però non numerabili (tante quanti sono i reali, R)
  - ***quali funzioni restano fuori, non sono realizzabili?***

Tesi Church e TM

- ***Quali funzioni non sono FRP?***
  - Molto difficili da trovare fra quelle “classiche”
  - E’ stato uno dei problemi della matematica più cruciali del secolo scorso
- ***Idea:***
  - *costruire TM che prendono in input la specifica di una altra TM e cercano di elaborarla per capire come funziona...*
- **Problema generale**:
  - può un formalismo predicare su se stesso?
    - es.: potrà un essere umano predire al 100% il funzionamento di un cervello umano?

UTM: macchine di Turing universali

- accetta due interi
  - un numero intero *y* che rappresenta un'altra TM (la descrizione di una certa MT Zy)
  - un input *x*
- ritorna l'output ottenuto applicando *x* alla TM identificata da *y*
- la funzione che realizza si denota con *fy(x)*

➢ **Questa macchina è realizzabile**

- è la macchina che realizza l'algoritmo delle TM
- è un simulatore delle TM
- prende gli input da nastro, applica la funzione di transizione di stato, produce gli output

UTM: macchine di Turing universali

- Una Macchina di Turing la cui parte di controllo (cioè il cui algoritmo “cablato”) consiste nel leggere dal nastro una descrizione dell’algoritmo richiesto è una ***UMT***
- E’ una macchina ***UNIVERSALE:***
  - senza modifiche alla sua struttura, può essere istruita per risolvere un qualunque problema (risolubile)
- E’ una macchina programmabile

Essa è la formalizzazione più corretta di un ordinario calcolatore, infatti introduce la possibilità di avere il programma memorizzato.

UTM: macchine di Turing universali

- Leggere *dal nastro* una *descrizione dell'algoritmo* richiesto richiede di:
  - saper descrivere tale algoritmo
  - il che richiede un qualche *linguaggio*
  - e una macchina che lo *interpreti !*
- Dunque la ***UTM*** è *l'interprete di un linguaggio*

Una UTM modella il concetto di elaboratore di uso generale ("general purpose")

- *Una macchina che va a cercare le "istruzioni" da compiere**…(fetch)***
- *.. le interpreta**…(decode)***
- *.. e le esegue**…(execute)***

UTM: macchine di Turing universali

UTM e macchina di Von Neuman

- UTM è in grado di *elaborare*
  - prendendo *dati* e *algoritmo* dal nastro
  - e scrivendo sul nastro i *risultati*
- Dunque, una UTM opera solo da/verso il nastro (astrazione della memoria): non esiste il concetto di "*mondo esterno*"
- Quindi la macchina di Von Neumann è modellata dalla UTM per ciò che attiene alla *computazione* ma prevede anche la dimensione dell'*interazione*

Problemi risolubili e non risolubili

- ***Il problema della terminazione***
  - può la UTM accorgersi che la TM (y) in input non terminerà la computazione di x?
- Turing ha dimostrato che
  - nessuna TM può calcolare la funzione *che* calcola la terminazione della TM *y* con input *x*

- *Dimostrazione per assurdo*

Problemi risolubili e non risolubili

- ***Ma facciamo un esempio…***

```
main()
{
    printf("hello, world\n");
}
```

Modifichiamo questo semplice programma

→

```
main ()
{
    int n, total, x, y, z;
    scanf("%d", &n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; x++)
            for (y=1; y<=total-x-1; y++) {
                z = total - x - y;
                if (exp(x,n) + exp(y,n) == exp(z,n))
                    printf("hello, world\n");
            }
        total++;
    }
}
```

Riusciamo ad accorgersi che siamo in loop, se sto simulando di essere l'automa che esegue le istruzioni?

Problemi risolubili e non risolubili

- Come sopperire a questa mancanza delle TM?
- Definire un formalismo più potente delle TM?

  - Tesi di Church:

  “***Nessun formalismo computazionale è più potente delle TM nello specificare funzioni discrete***”

  - Nessun procedimento automatico (algoritmo) può calcolare funzioni che le TM non calcolano

Problemi risolubili e non risolubili

- *Risultato generale*
  - un formalismo non riesce a predicare su se stesso in modo completo...

Problemi risolubili e non risolubili

Quale impatto?

- In generale, limita la potenza dei compilatori
  - le applicazione SW che accettano il listato di un programma e producono il codice eseguibile dall'elaboratore
- Non è possibile verificare:
  - se l'esecuzione termina con un certo input
  - qual è l'output corrispondente ad un certo input
  - testare automaticamente se una applicazione fa quello per cui è stata progettata
  - In alcuni casi è possibile farlo, ma in generale no

Problemi risolubili e non risolubili

Dato che un problema lo risolviamo in termini di computazione algoritmica possiamo dire che equivalentemente un problema risolubile è un problema decidibile

- ***Problemi decidibili***:
  - questi sono meccanicamente risolvibili da una TM
- ***Problemi indecidibili***:
  - Es: Il problema dell'arresto
    - può un calcolatore determinare se un dato programma, scritto in un dato linguaggio di programmazione, con determinati dati in ingresso, terminerà prima o poi la sua computazione?
      - *mancanza concetto infinito*

Problemi risolubili e non risolubili

- ***Problemi semidecidibili***:
  - Es: Si vuole determinare meccanicamente se un dato programma, scritto in un determinato linguaggio di programmazione, appartiene o meno all'insieme dei programmi che contengono errori.
    - il problema si può risolvere meccanicamente solo se la risposta al quesito è "Sì", cioè se il programma contiene errori
    - E.W.Dijkstra: "Il testing del software è funzionale solo ad affermare la presenza di errori, ma non la loro assenza".

Problemi risolubili e non risolubili

Tra i problemi teoricamente risolubili troviamo

Problemi trattabili e non trattabili

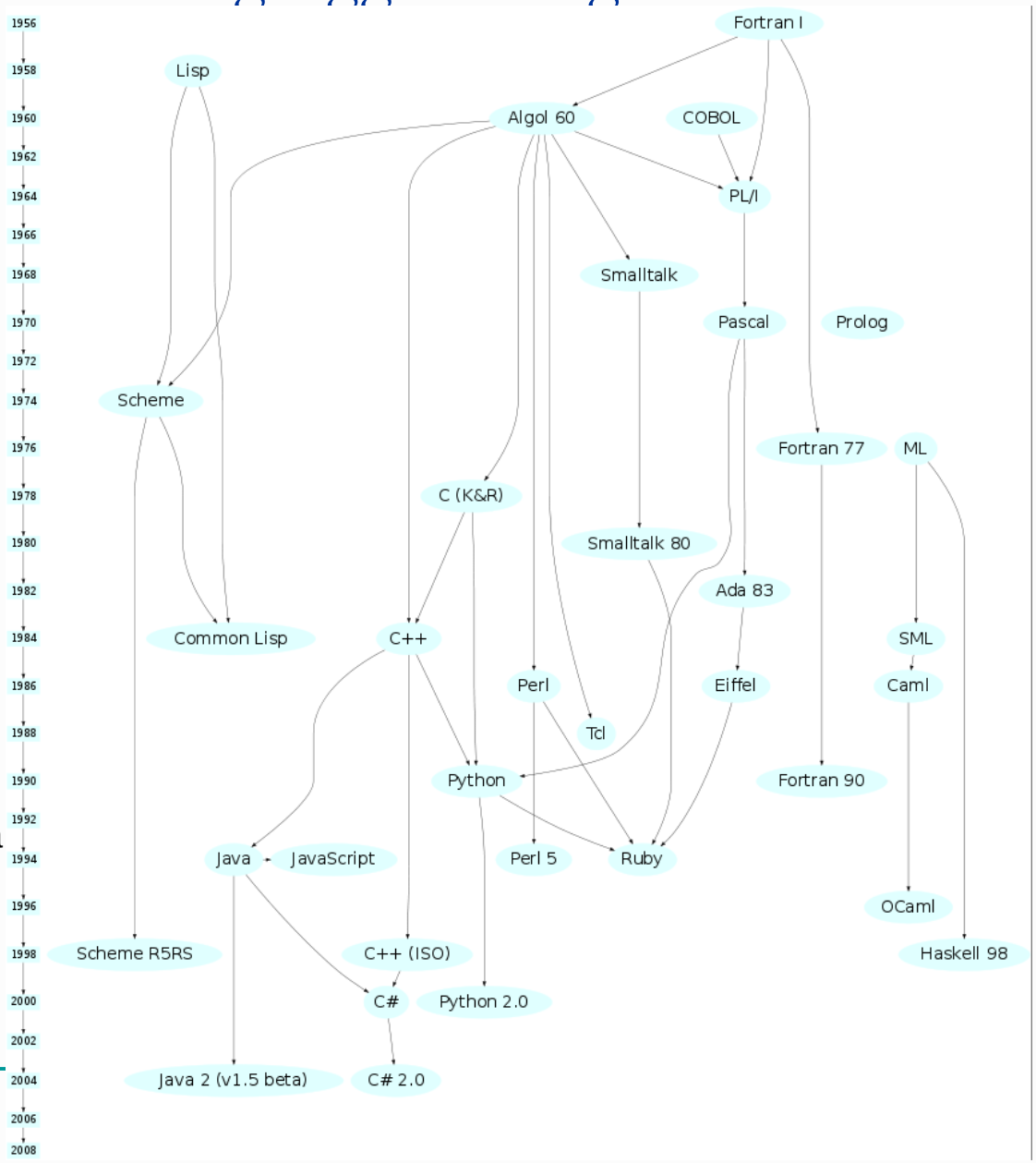
- Strettamente correlato alla complessità del calcolo

- *Se non è possibile ottenere una soluzione ad un problema entro un "ragionevole tempo" il problema diviene intrattabile anche se teoricamente risolubile*

- Es. risultato di una perfetta partita a scacchi in cui entrambi i giocatori giocano al meglio
  - Problema risolubile (numero finito di stati sulla scacchiera)
  - Si può dimostrare che un calcolatore anche molto potente ci metterebbe circa il tempo di vita stimato dell'universo: *problema intrattabile*

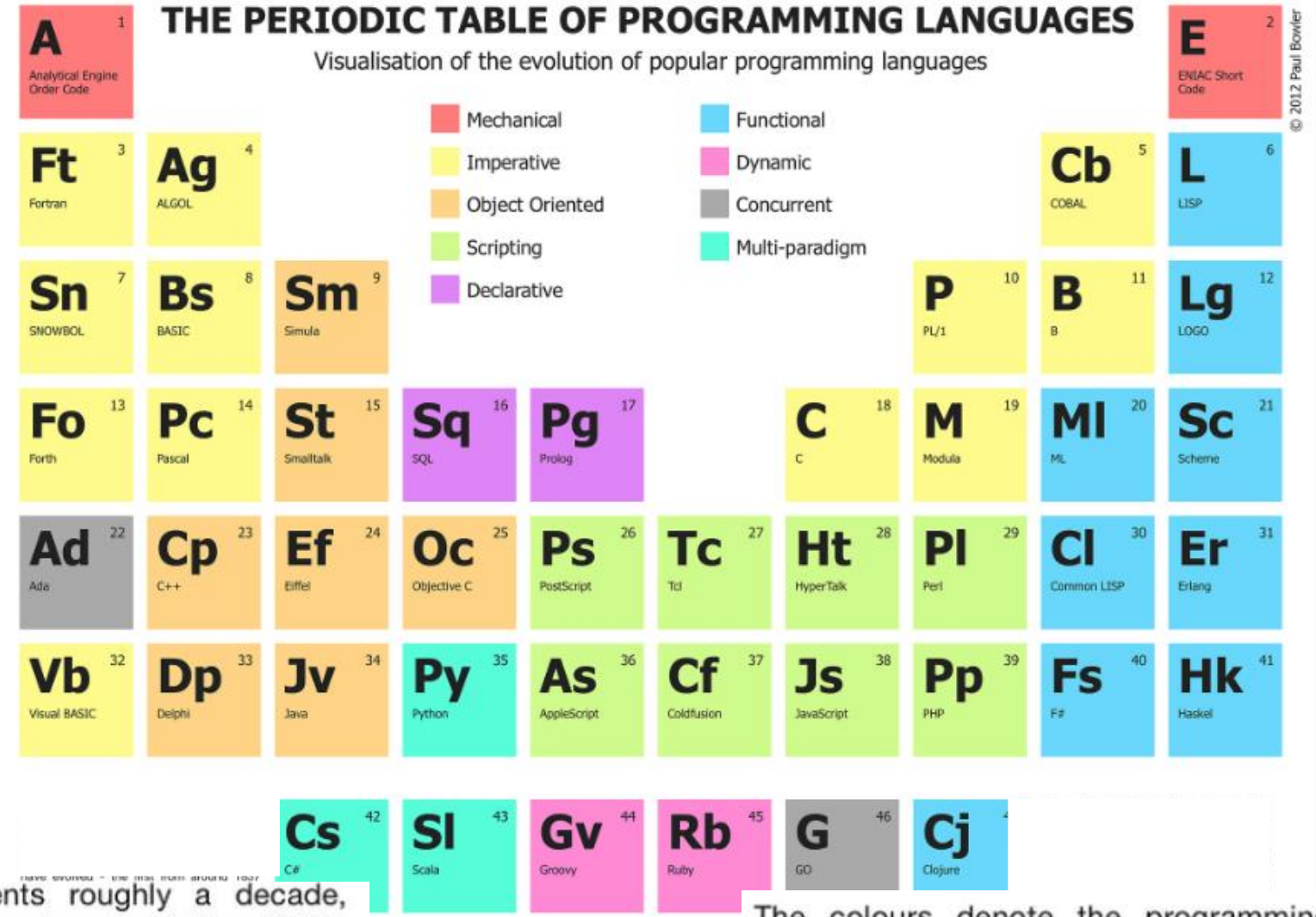
La teoria e l'evoluzione dei linguaggi di Programmazione

- ***Anni'30***
  - Teoria sistemi formali
  - Lambda calcolo
- ***Anni'40***
- UMT (Universal Machine Turing)
- 1947 macchina di Von Neumann
- ***Anni'50***
- 1956 nascita del Fortran (Formula Translator) (spesso si fa coincidere con la nascita dei L.di P.)
- 1957 Chomsky grammatiche generative
- ***Anni '60-80***
- Linguaggi ad alto livello
- '70-79 focalizzazione sulla differenza fra modelli dichiarativi e procedurali
- ***Anni '90***
- Paradigma a d oggetti



https://www.flickr.com/photos/68618833@N00/7637578338

https://www.slideshare.net/paulbowler/periodic-table-of-programming-languages-13738602



Each row represents roughly a decade, starting on the second row with the 1950's up to the 2000's on the final row. The first row is pre-1950 with the two mechanical programming systems from which all others have evolved - the first from around 1837 created by Charles Babbage and Ada Lovelace.

The colours denote the programming paradigm that the language in question originally supported or the primary paradigm for which it is known. Some languages may have evolved to support other paradigms over time which are not shown.

https://www.howtogeek.com/94917/the-evolution-of-computer-programming-languages-infographic/