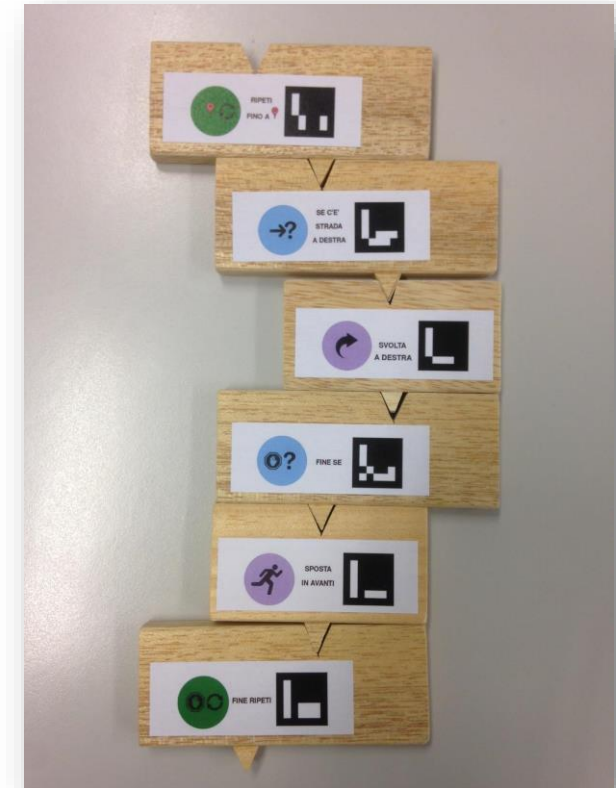




UNIVERSITÀ
DI PARMA

Informatica e Laboratorio di Programmazione
linguaggi
Alberto Ferrari

- linguaggi *formali*
 - linguaggi di *programmazione*
 - linguaggi di *marcatura* (es. HTML, Latex)
 - *interazione* uomo macchina (es. ricerca [Google](http://www.google.com))
 - linguaggi nel software di sistema
 - *compilatori*
 - *interpreti* ...

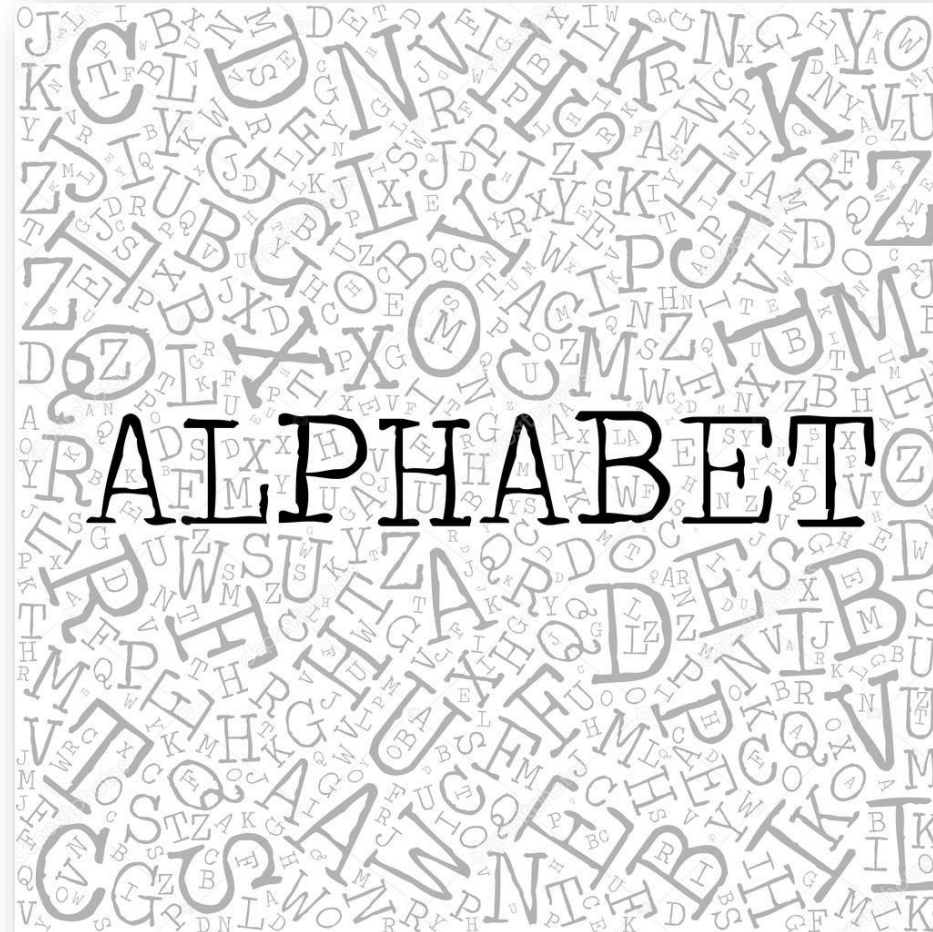


<http://www.ce.unipr.it/~aferrari/codowood>

- **grammatica**
 - quali frasi sono corrette?
 - **lessico**
 - analisi lessicale (sequenze corrette di simboli dell'alfabeto (token))
 - **sintassi**
 - analisi sintattica (sequenze corrette di token)
- **semantica**
 - cosa significa una frase corretta?
 - attribuzione del significato
- **pragmatica**
 - come usare una frase corretta?
- **implementazione**
 - come eseguire una frase corretta rispettando la semantica?

linguaggi formali

GRAMMATICA



- **alfabeto** Σ : insieme di simboli
- stringa s : sequenza di simboli di Σ
 - $s \in \Sigma^*$, insieme di tutte le stringhe
 - ϵ : stringa vuota
 - $|s|$: lunghezza della stringa s
- linguaggio $L \subseteq \Sigma^*$
 - **sottoinsieme** di tutte le stringhe possibili
 - **grammatica**: regole formali per definire le “*stringhe ben formate*” di L
- esempio: numeri romani da 1 a 1000
 - alfabeto $\{I, V, X, L, C, D, M\}$ + regole ...

- operazione di concatenazione •
 - proprietà associativa: $(\mathbf{x} \cdot \mathbf{y}) \cdot \mathbf{z} = \mathbf{x} \cdot (\mathbf{y} \cdot \mathbf{z})$
 - non commutativa: $\mathbf{x} \cdot \mathbf{y} \neq \mathbf{y} \cdot \mathbf{x}$
 - Σ^* chiuso rispetto alla concatenazione: $\Sigma^* \cdot \Sigma^* \rightarrow \Sigma^*$
- potenza
 - $\mathbf{x}^n = \mathbf{x} \cdot \mathbf{x} \cdot \mathbf{x} \cdot \mathbf{x} \dots$ (n volte)
- elemento neutro ϵ
 - stringa vuota, $\forall \mathbf{x} \in \Sigma^*, \epsilon \cdot \mathbf{x} = \mathbf{x} \cdot \epsilon = \mathbf{x}$

- L_1 ed L_2 linguaggi su Σ^* (due insiemi di stringhe)
- **unione**: $L_1 \cup L_2 = \{x \in \Sigma^* : x \in L_1 \vee x \in L_2\}$
- **intersezione**: $L_1 \cap L_2 = \{x \in \Sigma^* : x \in L_1 \wedge x \in L_2\}$
- **complementazione**: $\overline{L_1} = \{x \in \Sigma^* : x \notin L_1\}$
- concatenazione o **prodotto**: $L_1 \cdot L_2 = \{x \in \Sigma^* : x = x_1 \cdot x_2, x_1 \in L_1, x_2 \in L_2\}$
- **potenza**: $L^n = L \cdot L^{n-1}, n \geq 1; L^0 = \{\epsilon\}$ per convenzione
 - concatenazione di n stringhe qualsiasi di L
- **stella di Kleene**: $L^* = \bigcup L^n, n = 0.. \infty$
 - concatenazione arbitraria di stringhe di L

L^* : chiusura riflessiva e transitiva di L rispetto a \cdot

- approccio *algebrico*
 - linguaggio costruito a partire da linguaggi più elementari, con *operazioni* su linguaggi
- approccio *generativo*
 - *grammatica*, regole per la *generazione* di stringhe appartenenti al linguaggio
- approccio *riconoscitivo*
 - *macchina* astratta o *algoritmo* di *riconoscimento*, per decidere se una stringa appartiene o no al linguaggio

un esempio di grammatica

espressioni regolari - regex

- dato un alfabeto Σ , chiamiamo *espressione regolare* una stringa r sull'alfabeto $\Sigma \cup \{+, *, (,), \cdot, \emptyset\}$ tale che:
 - $r = \emptyset$: linguaggio vuoto; oppure
 - $r \in \Sigma$: linguaggio con un solo simbolo; oppure
 - $r = s + t$: unione dei linguaggi $L(s)$, $L(t)$; oppure
 - $r = s \cdot t$: concatenazione dei linguaggi $L(s)$, $L(t)$; oppure
 - $r = s^*$: chiusura del linguaggio $L(s)$
 - (*con s e t espressioni regolari; simbolo \cdot spesso implicito*)

esempio $\Sigma = \{a, b\}$: $a \cdot (a + b)^* \cdot b$

- una **espressione regolare** (*regular expression*) è una sequenza di simboli che identifica un insieme di stringhe
- esistono **regole condivise** fra i vari linguaggi relative alle regex 😊
- i vari linguaggi utilizzano notazioni diverse per rappresentare le stesse espressioni regolari 😞
- le espressioni regolari possono definire tutti e soli i **linguaggi regolari**
 - nella gerarchia di Chomsky la classe dei linguaggi generati da grammatiche di tipo 3
 - riconosciuti da automi a stati finiti
- espressioni regolari nei linguaggi di programmazione
 - prima implementazione **PERL** '80

- [...] per includere uno qualsiasi dei caratteri in parentesi
- singoli caratteri o intervalli di caratteri adiacenti
 - [A-Z] = qualsiasi lettera maiuscola
 - [a-zA-Z] = qualsiasi lettera minuscola oppure A, B, o C
- [^...] per escludere uno qualsiasi dei caratteri in parentesi
 - [^0-9] = qualsiasi carattere non numerico
- simboli speciali per identificare classi di caratteri
 - \d = numerico, equivale a [0-9]
 - \s = [\t\r\n\f]
 - \w = [0-9a-zA-Z_]
 - \D = non numerico, equivale a [^0-9]

- . per un carattere qualsiasi
 - A.B riconosce la stringa AoB, AwB, AOB ecc.
- \ *escape*, per sequenze speciali o caratteri speciali
 - \? cerca il ?
- ^ corrisponde all'inizio del testo
- \$ corrisponde alla fine del testo
- | per alternativa tra due espressioni (unione)
 - A|B = carattere A o carattere B
- (...) per raggruppare sotto-espressioni
 - ga(zz|tt)a trova sia gazza che gatta

- `{...}` per specificare il numero di ripetizioni
 - `\d{3,5}` sequenze di almeno tre cifre, al massimo cinque
- `*` zero o più occorrenze di un'espressione
 - `(ab)*` riconosce `ab`, `abab`, la stringa vuota, ma non riconosce `abba`
- `+` una o più occorrenze
 - `(ab)+` non riconosce la stringa vuota
- `?` zero o al più una occorrenza (parte opzionale)
 - `(ab)?` riconosce `ab` ma non `abab`



- codice fiscale:
 - `^[A-Z]{6}[0-9]{2}[A-Z][0-9]{2}[A-Z][0-9]{3}[A-Z]$`
 - dominio:
 - `^[\\w\\-]+\\. (it|com|org|net|eu|mobi)$`
- e-mail:
 - `^[\\w\\-\\.]+@[\\w\\-\\.]+\\. [a-z]+$`
- file: `^.+\\.zip$`
- data: `^\\d{2}/\\d{2}/\\d{4}$`

```
<script>
function validate(val) {
    var expr = /^\\d{2}\\/\\d{2}\\/\\d{4}$/;
    if (! expr.test(val)) {
        window.alert("Wrong date format");
        return false;
    }
    return true;
}
</script>

<form onsubmit="return validate(usrdate.value)">
    Date (dd/mm/yyyy): <input name="usrdate">
    <input type="submit">
</form>
```


- modulo standard re
- le espressioni regolari vengono ***compile*** in oggetti RegexObject
 - gli oggetti RegexObject contengono metodi per varie operazioni
 - ***match()***
 - determina se la RE corrisponde all'inizio della stringa (**None** se non trovata)
 - ***search()***
 - ricerca tutte le posizioni corrispondenti alla RE all'interno di una stringa (**None** se non trovata)
 - ***findall()***
 - trova tutte le sottostringhe corrispondenti alla RE, e le restituisce in una lista
 - ...

```
>>> import re  
>>> p = re.compile('ab*')
```

<https://docs.python.it/html/lib/re-syntax.html>

```
import re
ro = re.compile('[a-z]+')    # uno o più caratteri alfabetici minuscolo

#f = ro.match('sistemi09informativi')
f = ro.search('09sistemi09informativi')

if f:
    #group()    restituisce la stringa corrispondente alla RE
    print('f.group()', f.group())
    #start()    restituisce la posizione iniziale della corrispondenza
    print('f.start()', f.start())
    #end()    restituisce la posizione finale della corrispondenza
    print('f.end()', f.end())
    #span() restituisce una tupla contenente la (start, end) posizione della corrispondenza
    print('f.span()', f.span())
else:
    print('no match')

l = ro.findall('09sistemi09informativi')
print(l)
```

```
import re

# Remove anything other than digits
phone = 'Phone: +39 0521 905708'
num = re.sub(r'\D', "", phone)
print("Phone Num : ", num)

# Hide password
string = 'Password: mypwd'
hidden = re.sub(r'^(Password:\s*).*$', r'\1(*****)', string)
print(hidden)

# part of a match
address = 'Please mail it to alberto.ferrari@unipr.it.com'
match = re.search(r'([\w.-]+)@([\w-]+).([\w]+)', address)
print(match.group(0))
print(match.group(1))
print(match.group(2))
print(match.group(3))
```

<https://docs.python.org/3/library/re.html>

```
import re

print('match only upper and lowercase letters, numbers, and underscores')
patterns = r'^[a-zA-Z0-9_]*$'
text1 = 'sistemi_informativi'
if re.search(patterns, text1):
    print('ok')

print('minimum eight characters, at least one uppercase letter, one lowercase letter and one number')
patterns = r'^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-Z\d]{8,}$'
pwd = input('password ')
while not re.search(patterns, pwd):
    pwd = input('password ')
```

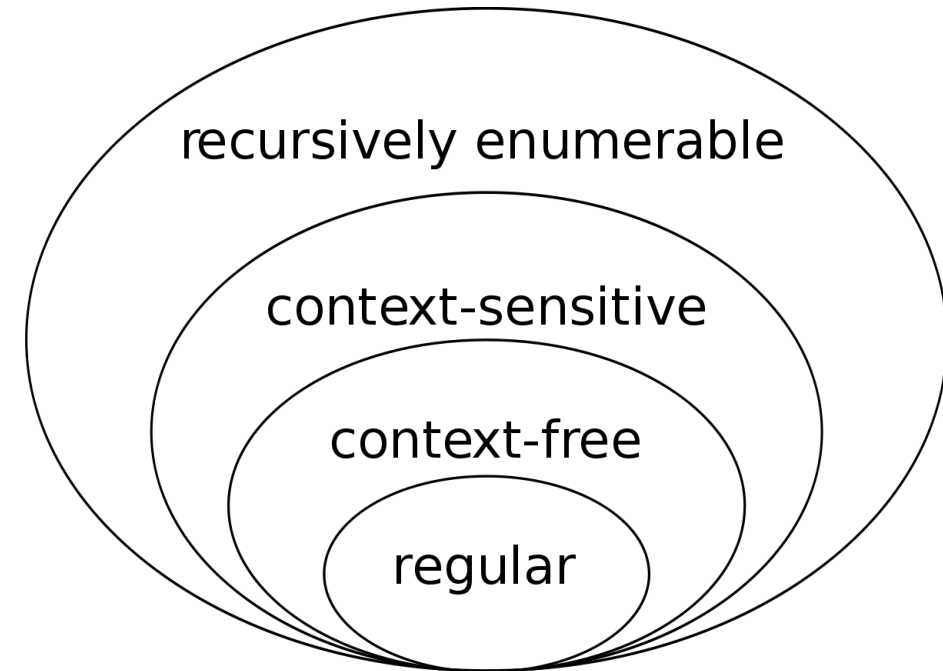
```
def is_alphanumeric(string):
    characterRegex = re.compile(r'^a-zA-Z0-9.'])
    string = characterRegex.search(string)
    return bool(string)

def is_mailAddress(string):
    characterRegex = re.compile(r'^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})$')
    string = characterRegex.search(string)
    return bool(string)

def is_CF(string):
    # characterRegex = re.compile(r'^[a-z]{6}[0-9]{2}[a-z][0-9]{2}[a-z][0-9]{3}[a-z]$')
    characterRegex = re.compile(r'^[A-Z]{6}'
                                r'[0-9LMNPQRSTUVWXYZ]{2}[ABCDEHLMRST]{1}[0-9LMNPQRSTUVWXYZ]{2}'
                                r'[A-Z]{1}[0-9LMNPQRSTUVWXYZ]{3}[A-Z]{1}$')
    string = characterRegex.search(string)
    return bool(string)
```

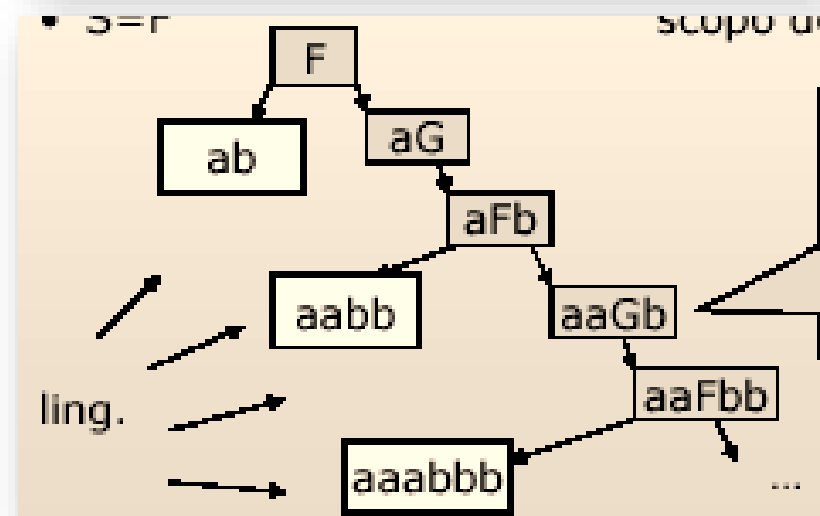
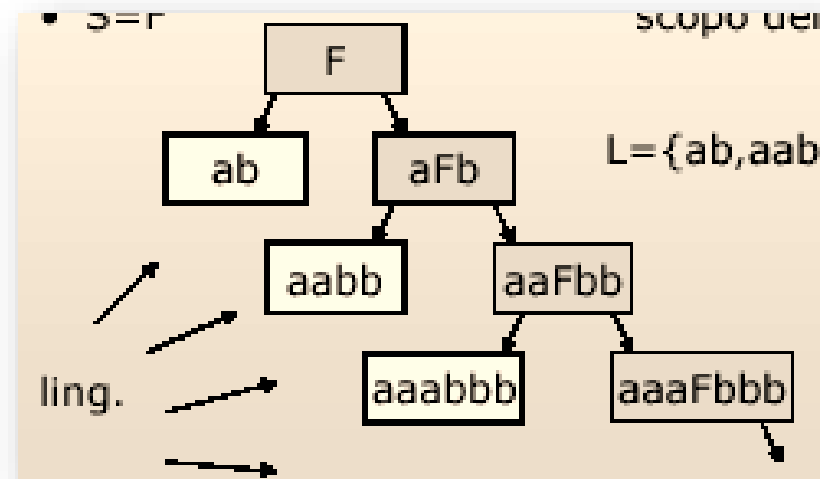
approccio generativo

grammatiche di Chomsky



- grammatica $G = \langle V_T, V_N, P, S \rangle$
- V_T : alfabeto finito di simboli *terminali*
- V_N : alfabeto finito di simboli *non terminali*
 - (variabili, categorie sintattiche)
- $V = V_T \cup V_N$
- P : insieme di produzioni
 - $\langle \alpha, \beta \rangle \in P$ si indica con $\alpha \rightarrow \beta$
 - $\alpha \in V$
 - $\beta \in V$
- $S \in V_N$: assioma
- $L(G)$: *insieme delle stringhe di simboli terminali ottenibili con finite operazioni di riscrittura*

- la radice dell'albero è l'**assioma**
- ogni nodo ha tanti **figli**, quante sono le **produzioni** applicabili
- ogni **figlio** è una **forma di frase**
- le **foglie** sono **stringhe del linguaggio**
- primo esempio:
 - $G_1 = \langle \{a,b\}, \{F\}, P_1, F \rangle$
 - $P_1 = \{F \rightarrow ab, F \rightarrow aFb\}$
- secondo esempio:
 - $G_2 = \langle \{a,b\}, \{F,G\}, P_2, F \rangle$
 - $P_2 = \{F \rightarrow ab, F \rightarrow aG, G \rightarrow Fb\}$



- $G = \langle \{a, b, +, *, -, (,)\}, \{E, I\}, P, E \rangle$
 - (1) $E \rightarrow I$
 - (2) $E \rightarrow E + E$
 - (3) $E \rightarrow E * E$
 - (4) $E \rightarrow E - E$
 - (5) $E \rightarrow -E$
 - (6) $E \rightarrow (E)$
 - (7) $I \rightarrow a$
 - (8) $I \rightarrow b$
 - (9) $I \rightarrow Ia$
 - (10) $I \rightarrow Ia$
- ... genera espressioni aritmetiche con operatori +,*, - unario e binario e simboli a b

○ $G = \langle \{a, b, c\}, \{S, B, C\}, P, S \rangle$

(1) $S \rightarrow aSBC$

(2) $S \rightarrow aBC$

(3) $CB \rightarrow BC$

(4) $aB \rightarrow ab$

(5) $bB \rightarrow bb$

(6) $bC \rightarrow bc$

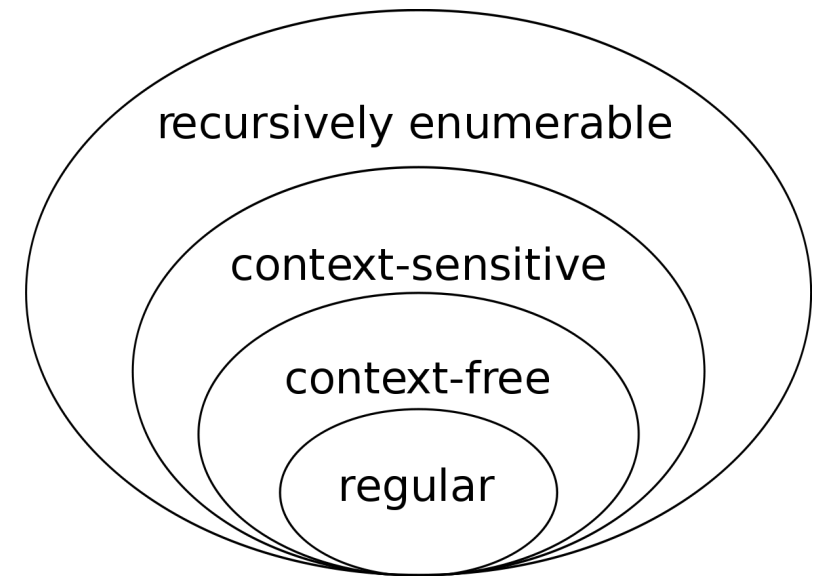
(7) $cC \rightarrow cc$

○ ... genera il linguaggio $\{a^n b^n c^n : n \geq 1\}$

○ per generare aaabbbccc:

○ applicare 1-1-2-3-3-3-4-5-5-6-7-7

- **tipo 0**: grammatiche *ricorsivamente enumerabili (RE)*
 - $\alpha A \beta \rightarrow \gamma$ (non limitate)
- **tipo 1**: grammatiche *contestuali (CS)*
 - $\alpha A \beta \rightarrow \alpha \gamma \beta$
- **tipo 2**: grammatiche *non contestuali (CF)*
 - $A \rightarrow \gamma$
- **tipo 3**: grammatiche *regolari (REG)*
 - $A \rightarrow aB$, oppure $A \rightarrow b$, oppure $A \rightarrow \varepsilon$
 - coincide con classe dei linguaggi definiti da *regex*



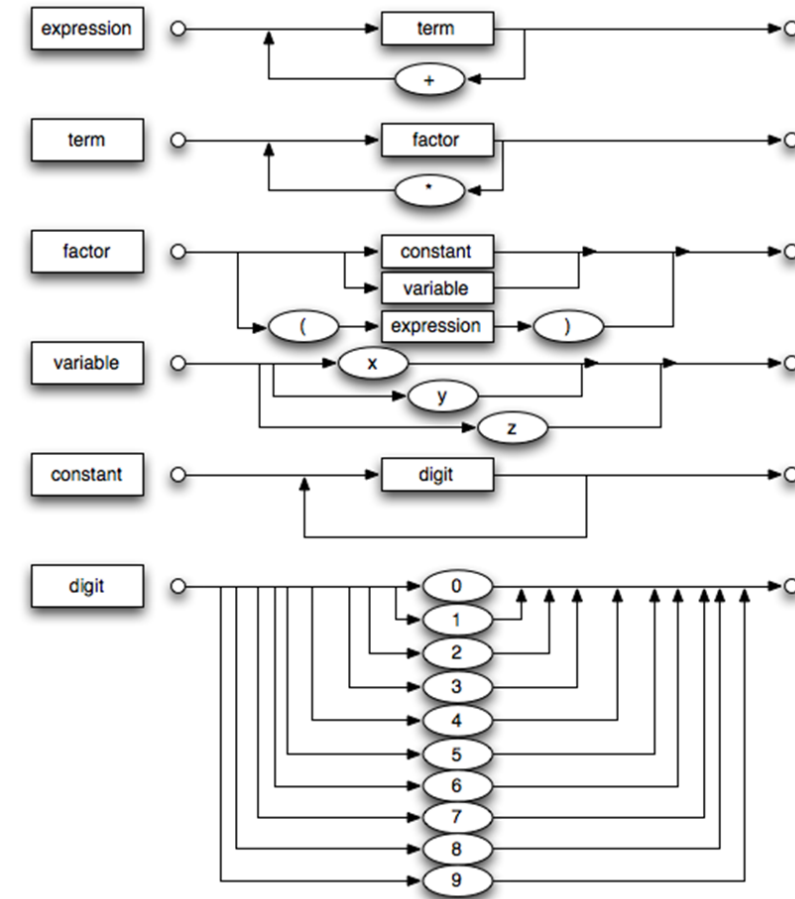
$A, B \in V_N; a, b \in V_T; \alpha, \beta, \gamma \in V^*$



linguaggi di programmazione

- notazione **formale** per definire algoritmi
 - *algoritmo: sequenza di istruzioni per risolvere un dato problema in un tempo finito*
- ogni linguaggio è caratterizzato da:
 - **sintassi**
 - **semantica**

- insieme di *regole formali* per scrivere *frasi ben formate* (*programmi*)
- lessico*
 - parole riservate, operatori, variabili, costanti ecc. (*token*)
- grammatiche non contestuali (...) espresse con notazioni formali:
 - Backus-Naur Form*
 - Extended BNF*
 - Diagrammi sintattici*



- attribuisce un ***significato*** alle frasi (*sintatticamente corrette*) costruite nel linguaggio
- una frase può essere sintatticamente corretta ma non aver alcun significato
 - soggetto – predicato – complemento
 - “La mela mangia il bambino”
 - “Il bambino mangia la mela”
- avere un significato diverso da quello previsto...
 - `GREEK_PI = 345`

- correttezza sui ***tipi***
 - quali tipi di dato possono essere elaborati?
 - quali operatori applicabili ad ogni dato?
 - quali regole per definire nuovi tipi e operatori?
- ***semantica operativa***
 - qual è l'effetto di ogni azione elementare?
 - qual è l'effetto dell'aggregazione delle azioni?
 - qual è l'effetto dell'esecuzione di un certo programma?

- più *orientati alla macchina* che ai problemi da trattare
- *linguaggi macchina*
 - solo operazioni eseguibili direttamente dall'elaboratore
 - Op. molto elementari, diverse per ogni processore, in formato binario
- *linguaggi assembly*
 - prima evoluzione
 - codici binari → mnemonici

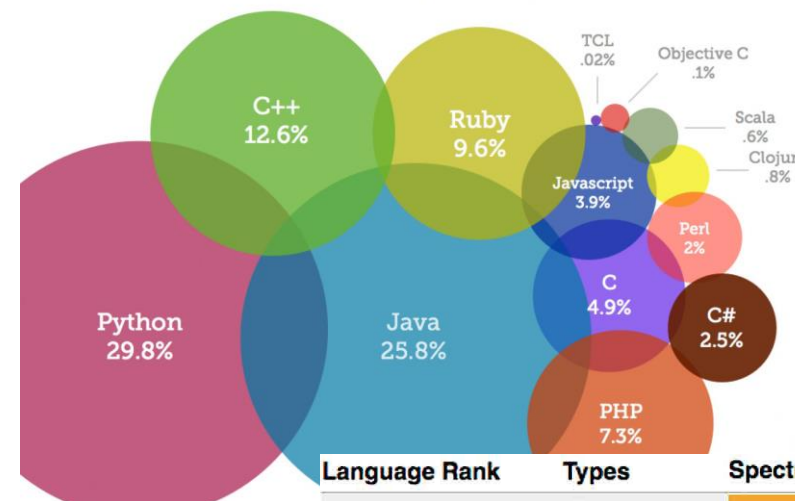
```
; Example of IBM PC assembly language
; Accepts a number in register AX;
; subtracts 32 if it is in the range 97-122;
; otherwise leaves it unchanged.























SUB32  PROC           ; procedure begins here
        CMP  AX,97     ; compare AX to 97
        JL   DONE      ; if less, jump to DONE
        CMP  AX,122    ; compare AX to 122
        JG   DONE      ; if greater, jump to DONE
        SUB  AX,32     ; subtract 32 from AX
DONE:   RET            ; return to main program
SUB32   ENDP          ; procedure ends here
```

FIGURE 17. Assembly language

- introdotti per *facilitare* la scrittura dei programmi
- definizione della soluzione in modo intuitivo
- *astrazione* rispetto al calcolatore su cui verranno eseguiti
- necessaria *traduzione* in linguaggio macchina

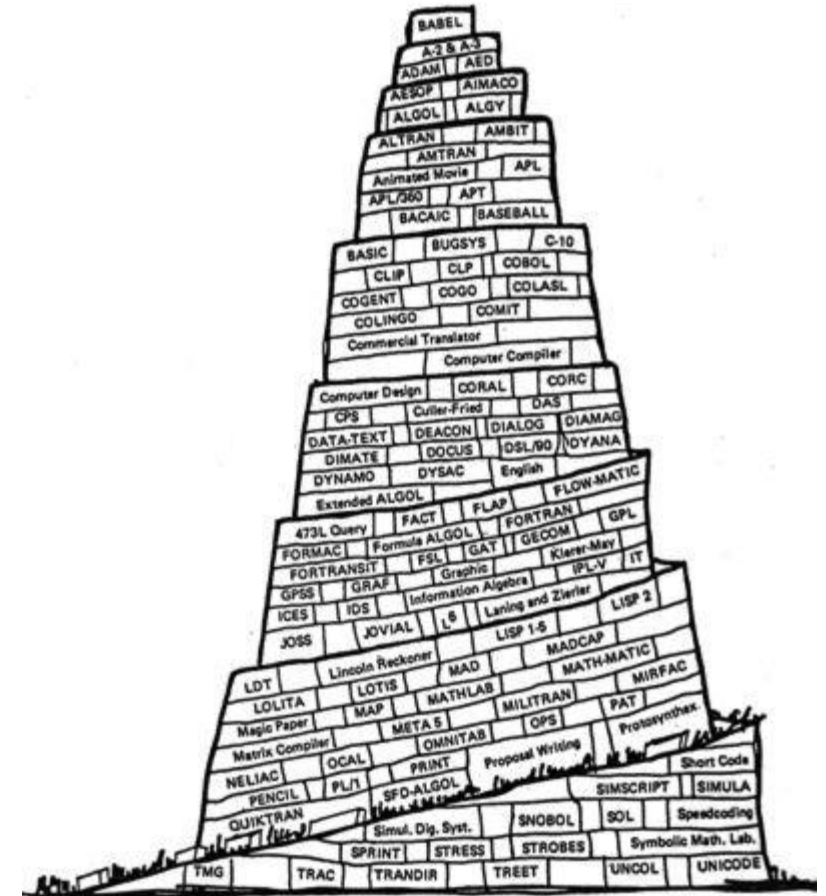
Most Popular Coding Languages of 2018



Language Rank	Types	Spectrum Ranking
1. Python	  	100.0
2. C++	  	99.7
3. Java	  	97.5
4. C	  	96.7
5. C#	  	89.4
6. PHP		84.9
7. R		82.9
8. JavaScript	 	82.6
9. Go	 	76.4
10. Assembly		74.1

storia ed evoluzione dei linguaggi di programmazione

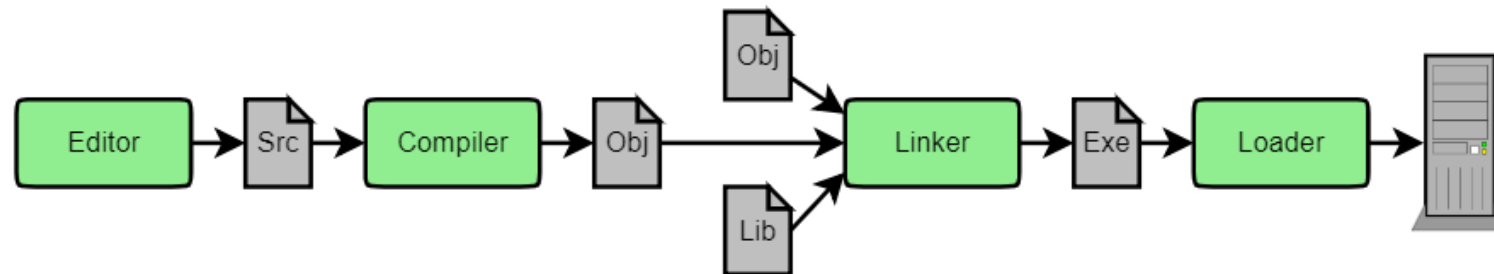
<https://www.cs.toronto.edu/~gpenn/csc324/PLhistory.pdf>
<http://www.levenez.com/lang/history.html>
http://www.cs.brown.edu/~adf/programming_languages.html



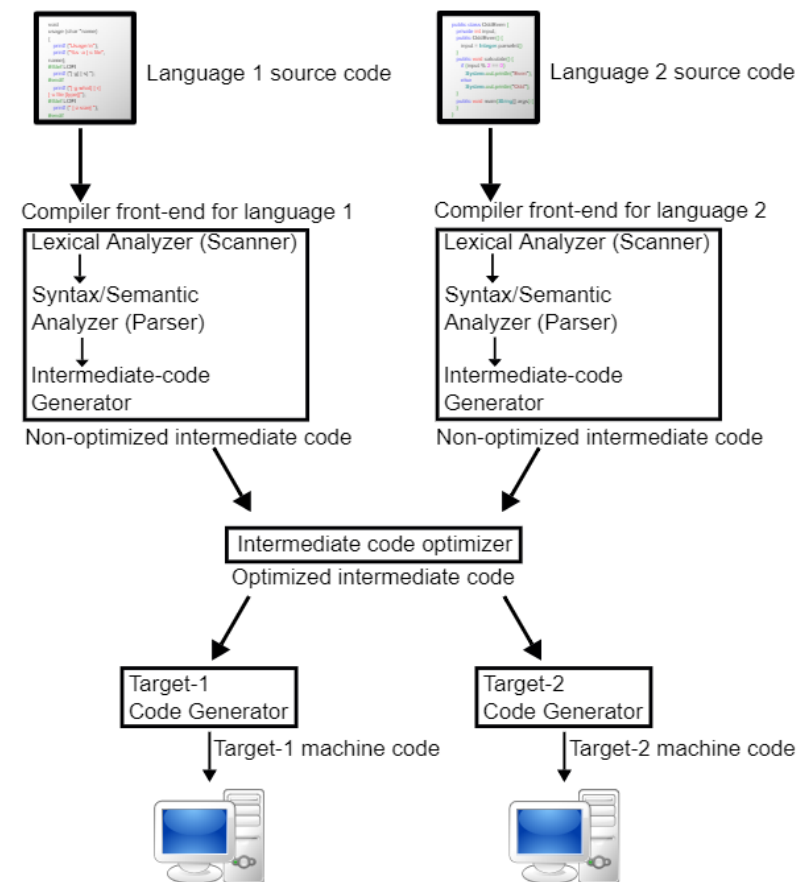
- definiscono la filosofia e la metodologia con cui si scrivono i programmi
- definiscono il concetto (*astratto*) di **computazione**
- ogni linguaggio consente (o spinge verso) l'adozione di un particolare paradigma
 - **imperativo / procedurale**
 - **orientato agli oggetti**
 - **scripting** (*tipizzazione dinamica, principio DRY - Don't Repeat Yourself*)
 - **funzionale** (*funzioni come “cittadini di prima classe”*)
 - **logico** (*base di conoscenza + regole di inferenza*)

- ***imperativi / procedurali***
 - Cobol, Fortran, Algol, C, Pascal
- ***orientati agli oggetti***
 - Simula, Smalltalk, Eiffel, C++, Delphi, Java, C#, VB.NET
- ***scripting***
 - Basic, Perl, PHP, Javascript, Python
- ***funzionali***
 - Lisp, Scheme, ML, Haskell, Erlang
- ***logici***
 - Prolog...

- linguaggio ad alto livello → passi necessari:
 - **compilazione**, traduzione in linguaggio macchina
 - **collegamento** (*linking*) con librerie di supporto
 - **caricamento** (*loading*) in memoria
 - programmi **compilati**: applicati i 3 passi...
 - programmi **interpretati**: applicati i 3 passi...
- a **tutto il codice**; prima dell'esecuzione
 - in sequenza, **su ogni istruzione**; a tempo di esecuzione

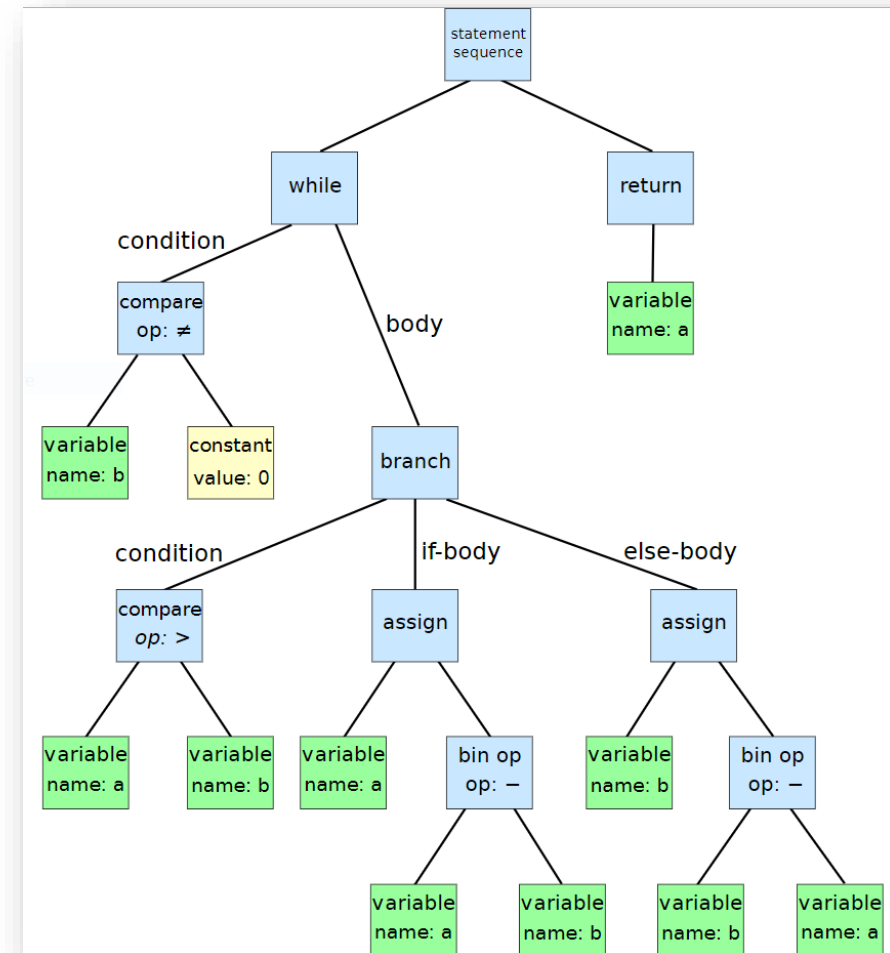


- traduzione da linguaggio alto livello a linguaggio macchina
- **analisi**: lessicale, grammaticale, contestuale
- rappresentazione **intermedia**:
albero sintattico annotato (AST)
- generazione **codice oggetto**
 - non ancora eseguibile
 - linker, loader



```
while b != 0:  
    if a > b:  
        a = a - b  
    else:  
        b = b - a  
return a
```

Algoritmo di Euclide per MCD



- il ***linker*** collega diversi moduli oggetto
 - simboli irrisolti → riferimenti esterni
- il collegamento può essere ***statico*** o ***dinamico***
 - collegamento ***statico***
 - libreria inclusa nel file oggetto, eseguibile stand-alone
 - dimensioni maggiori, ma possibile includere solo funzionalità utilizzate
 - collegamento ***dinamico***
 - librerie condivise da diverse applicazioni
 - installazione ed aggiornamento unici
 - caricate in memoria una sola volta

- il **loader** carica in memoria un programma *rilocabile*
 - risolti tutti gli **indirizzi** relativi (variabili, salti ecc.)
 - caricati eventuali programmi di supporto
- rilocalizzazione **statica**
 - indirizzi logici trasformati in **indirizzi assoluti**
- rilocalizzazione **dinamica**
 - **indirizzi logici** mantenuti nel programma in esecuzione
 - programma compilato: indirizzamento relativo
 - tramite registro base: locazione in memoria del codice, dei dati e dello stack
 - Memory Management Unit

- compilazione in *codice intermedio*
 - Bytecode (Java), Common Intermediate Lang. (.NET), ...
 - Python: compilato per una macchina virtuale (file .pyc), ma in modo trasparente
- esecuzione su una *macchina virtuale*, che gestisce la memoria (garbage collection)
 - Java Virtual Machine, Common Language Runtime, ...
 - spesso compilazione “al volo” (Just In Time) in codice nativo