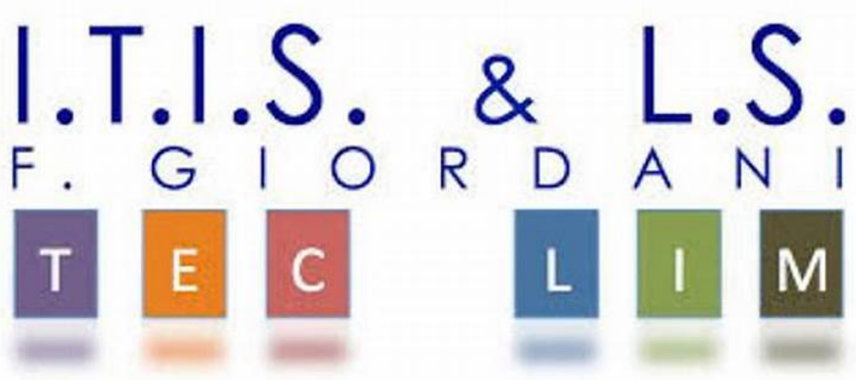



**ITIS-LS “Francesco Giordani” Caserta**

**prof. Ennio Ranucci**

**a.s. 2019-2020**

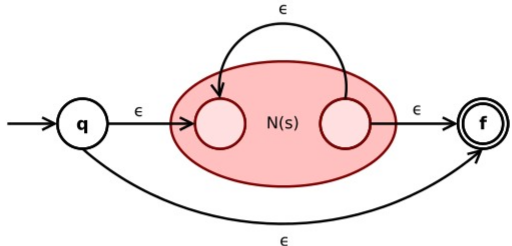
# *Linguaggi ed espressioni regolari*

*Javascript*

*Linguaggi formali*

*Espressione regolare un esempio di grammatica*

*Automi ed espressioni regolari*



***ITIS-LS "Francesco Giordani" Caserta***
***Anno scolastico:*** *2019/2020*
***Classe 3^ sez.B spec. Informatica e telecomunicazioni***
***Data:***
***Numero progressivo dell'esercizio:*** *es1 (espressioneRegolareData.html)*
***Versione:*** *1.0*
***Programmatore/i:***
***Sistema Operativo:*** *Windows 10*
***Compilatore/Interprete:*** *Browser - Javascript*
***Obiettivo didattico:***
*Espressioni regolari*
***Obiettivo del programma:***
*Validare la data*

***File espressioneRegolareData.html***

```
<script>

 function validate(val)

   { var expr = /^\d{2}\/\d{2}\/\d{4}$/;

     if (! expr.test(val)) {

                       window.alert("Formato data non valida");

                       return false;

                     }

     return true;

   }

</script>

<form onsubmit="return validate(usrdate.value)">

 Date (dd/mm/yyyy): <input name="usrdate">

 <input type="submit">

</form>
```

***ITIS-LS "Francesco Giordani" Caserta***
***Anno scolastico:*** *2019/2020*
***Classe 3^ sez.B spec. Informatica e telecomunicazioni***
***Data:***
***Numero progressivo dell'esercizio:*** *es2 (espressioneRegolareMail.html)*
***Versione:*** *1.0*
***Programmatore/i:***
***Sistema Operativo:*** *Windows 10*
***Compilatore/Interprete:*** *Browser - Javascript*
***Obiettivo didattico:***
*Espressioni regolari*
***Obiettivo del programma:***
*Validare la mail*

***File espressioneRegolareMail.html***

```
<script>

 function validate(val)

  { var expr =/^[\w\-\.]+@[\w\-\.]+\.[a-z]+$/;

   if (! expr.test(val)) {

                    window.alert("Formato mail non valida");

                    return false;

                   }

   return true;

  }

</script>

<form onsubmit="return validate(usrdate.value)">

 Mail (prova@dominioII.dominioI): <input name="usrdate">

 <input type="submit">

</form>
```

***Un tipo diverso di algebra: le espressioni regolari. (fonte lidimatematici)***

*Immaginiamo di avere a disposizione un insieme di simboli a nostra scelta, ad esempio:*

*A = {a, b, c}*

*lo chiameremo alfabeto.*

L'idea è di costruire un modello formale che ci consenta di generare sequenze di questi simboli, come ad esempio "aaa" oppure "aabb" o ancora, "acac".
Le sequenze di simboli sono dette stringhe.
Per fare ciò, abbiamo bisogno di definire un insieme di operazioni ammesse, in modo tale da costruire stringhe a partire dai singoli simboli dell'insieme A dell'alfabeto. Queste operazioni saranno anch'esse rappresentate da simboli. Simboli speciali, però, perché indicano azioni.
I simboli delle operazioni ammesse, che chiameremo operatori, sono quindi diversi dai simboli degli elementi dell'alfabeto.

E' una distinzione importante, perché il nostro processo di costruzione non ha alcun vincolo, siamo perfettamente liberi di inventare i simboli e di attribuirli ai vari operatori o elementi dell'alfabeto. E' una nostra scelta.

La prima operazione che vogliamo è sicuramente di poter scrivere serie di simboli, la chiameremo concatenazione, e la indichiamo con un punto '.':

a.a = aa

decidiamo anche che, per comodità, questo operatore può essere omesso, in modo da rendere più leggibile l'operazione.
La seconda operazione che vogliamo è la possibilità di scegliere alternativamente un simbolo oppure l'altro, chiameremo questo operatore unione, e lo indicheremo con il +. Quindi, ad esempio, usando concatenazione ed unione otteniamo:

a(a+b) = aa oppure ab

Infine, definiamo un terzo operatore che consente di concatenare zero o più volte, a nostra scelta, una stringa, lo indichiamo con *:

a* = aaaaaaaa …

E' un operatore molto particolare, che va sotto il nome di Stella di Kleene.
Manca ancora un elemento importantissimo, la stringa vuota, che di norma viene indicata con la lettera greca ε (epsilon). La stella di Kleene ammette infatti anche nessuna ripetizione del simbolo, quindi la stringa vuota ε è un risultato ammesso dell'espressione a*.

a* = ε, a, aa, aaa, …

Soffermiamoci per ora a giocare con le espressioni regolari, per costruire insiemi di stringhe a partire dai simboli dell'alfabeto e gli operatori che abbiamo definito.

Ad esempio, l'espressione regolare:

a(b+c)b*

ammette come soluzione questo insieme di stringhe:

{ab, ac, abb, abbb…., acb, acbbb …}

Le espressioni regolari sono una vera e propria algebra, detta Algebra di Kleene.
E' grazie al matematico americano Stephen Kleene se, oggi, abbiamo gli strumenti matematici che ci consentono di comunicare con i calcolatori elettronici attraverso strutture di linguaggio specifiche, dette linguaggi formali.

Le espressioni regolari sono strumenti che consentono di rappresentare, in modo estremamente sintetico, intere famiglie di stringhe dalla struttura anche piuttosto complessa.

E' possibile costruire un'automa in grado di riconoscere stringhe aventi struttura specifica. Teorema:

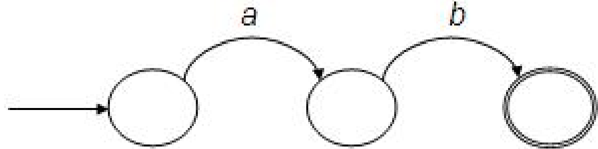
Per ogni espressione regolare esiste un automa a stati finiti in grado di riconoscere il linguaggio generato.

Un automa è un insieme di stati, archi e simboli che etichettano gli archi. Gli automi hanno due stati speciali, lo stato iniziale e lo stato finale. A partire da una espressione regolare, si costruisce un automa a stati finiti che ne riconosce il linguaggio applicando pochissime regole.

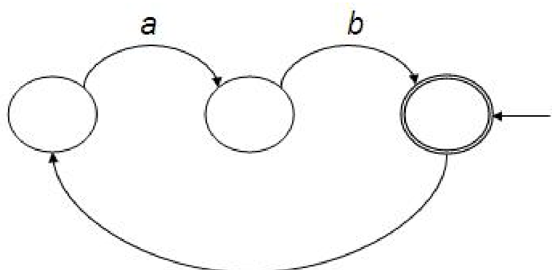
Lo stato iniziale si distingue per essere l'una con una freccia entrante e nessuno stato precedente, e quello finale per una doppia bordatura.

La stringa vuota è riconosciuta da uno stato che è, contemporaneamente, iniziale e finale. Quindi ε è riconosciuta da:

La concatenazione di due porzioni di espressione regolare si costruisce connettendo gli automi che le riconoscono con un arco, su cui è stato posto l'elemento corrispondente del vocabolario. Quindi, ad esempio, la stringa ab è riconosciuta da:

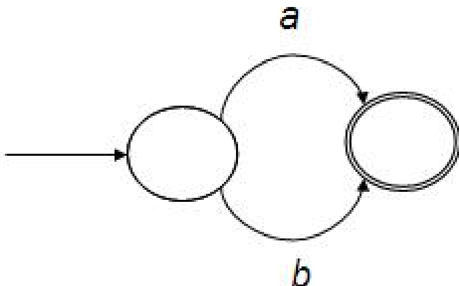



La Stella di Kleene si realizza ponendo un arco che rientra all'ingresso stesso della porzione di espressione da ripetere indefinitamente, ad esempio, questo automa riconosce la stringa (ab)*:

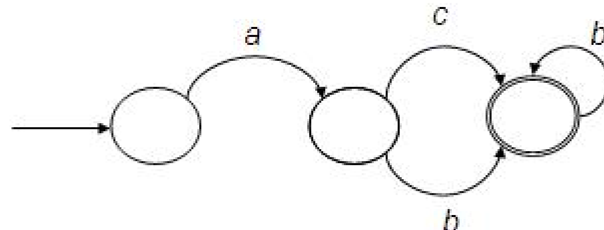


Notate che, siccome l'espressione (ab)* è soddisfatta anche dalla stringa vuota ε, l'automa ha uno stato iniziale che è anche finale.

Infine, l'unione si riconosce ponendo due archi uscenti con il corrispondente elemento dell'alfabeto. Ad esempio, a+b è riconosciuta da questo automa:




L'automa che riconosce l'espressione regolare a(b+c)b* è quindi:




L'insieme di stringhe generato dalle espressioni regolari e riconosciuto dai relativi automi a stati finiti è anche detto linguaggio. Automi ed espressioni regolari sono oggetti matematici dotati di un formalismo ben preciso e di potenzialità espressive caratteristiche.

Il matematico inglese Alan Turing, oggi è ritenuto a buon diritto uno dei padri, se non il padre e fondatore della scienza dell'informazione.

Come tanti geni è stato incompreso in vita e, per il fatto di essere gay, addirittura osteggiato. Turing nasce il 23 giugno 1912 a Paddington nei dintorni di Londra.

Si laurea nel 1934 a Cambridge, distinguendosi già per i propri solidissimi contributi in meccanica quantistica, probabilità e logica.
Nel 1936 inventa una astrazione matematica del moderno calcolatore: la Macchina di Turing.
Il genio inglese supera le limitazioni degli automi a stati finiti intuendo che il problema sta nella mancanza di memoria.
Immagina una macchina astratta che, come gli automi a stati finiti, è dotata di un alfabeto di simboli, un insieme di stati e una funzione di transizione ma, per la prima volta in assoluto, a questi elementi aggiunge una memoria interna, un nastro, da cui legge e scrive simboli dell'alfabeto. Le operazioni condotte dalla macchina di Turing sono assimilabili, a tutti gli effetti, alle istruzioni dei microprocessori moderni.

Nel suo lavoro di dottorato (1938) in Logica, Teoria dei numeri e Algebra, Turing dimostra che la sua macchina è universale, cioé equivalente a tutti i possibili tipi di macchina calcolatrice. In altri termini, qualsiasi programma di computer scritto in passato, presente o futuro, è convertibile in un suo equivalente eseguibile con la Macchina di Turing.
Turing si distingue durante la seconda guerra mondiale per aver saputo svelare il segreto di Enigma, il codice di crittografia usato dai nazisti per le proprie comunicazioni. Il suo contributo si rivela determinante per la vittoria degli alleati. Alla fine della seconda guerra mondiale Turing si dedica a tempo pieno allo sviluppo della moderna scienza dei calcolatori, e svolge un ruolo di

primissimo piano nella definizione dei principi teorici sottostanti la programmazione, le reti neurali, e l'intelligenza artificiale.

***ITIS-LS "Francesco Giordani" Caserta***
***Anno scolastico:*** *2019/2020*
***Classe 3^ sez.B spec. Informatica e telecomunicazioni***
***Data:***
***Numero progressivo dell'esercizio:*** *es3 (GeneraStringhe.html)*
***Versione:*** *1.0*
***Programmatore/i:***
***Sistema Operativo:*** *Windows 10*
***Compilatore/Interprete:*** *Browser - Javascript*
***Obiettivo didattico:***
*Espressioni regolari, generare stringhe in modo casuale e validarle*
***Obiettivo del programma:***
*genera e valida stringa*

***File GeneraStringhe.html***

```
<html>

<head>

 <script>

/*

   Funzione in puro Javascript  per generare una stringa random

*/

var stringaDavalidare;

function generateRandomString(iLen)

 {

   var stringaRandom = '';

   var sChrs = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXTZabcdefghiklmnopqrstuvwxyz";

   for (var i = 0; i < iLen; i++)

    {

      var randomPoz = Math.floor(Math.random() * sChrs.length);

      stringaRandom += sChrs.substring(randomPoz, randomPoz + 1);

    }

   document.getElementById("result").innerHTML = stringaRandom;

   stringaDaValidare=stringaRandom

  }

function validate()

   { var expr = /a+\w*/i;
```

```
    if (! expr.test(stringaDaValidare))
     {
       window.alert("Formato stringa non valida");
     }
    else
    {
      window.alert("Formato stringa valida");
    }
  }
</script>
</head>
<body>
 <input type="button" value="Genera stringa" onClick="generateRandomString(8)" />
 <input type="button" value="Valida stringa" onClick="validate()" />
 <br /><br />
 Stringa random generata:
 <div id="result"></div>


</body>
</html>
```

Nota:

```
var expr = /a(b+c)b*/;
```

ammette questo insieme di stringhe valide:

{ab, ac, abb, abbb…., acb, acbbb …}

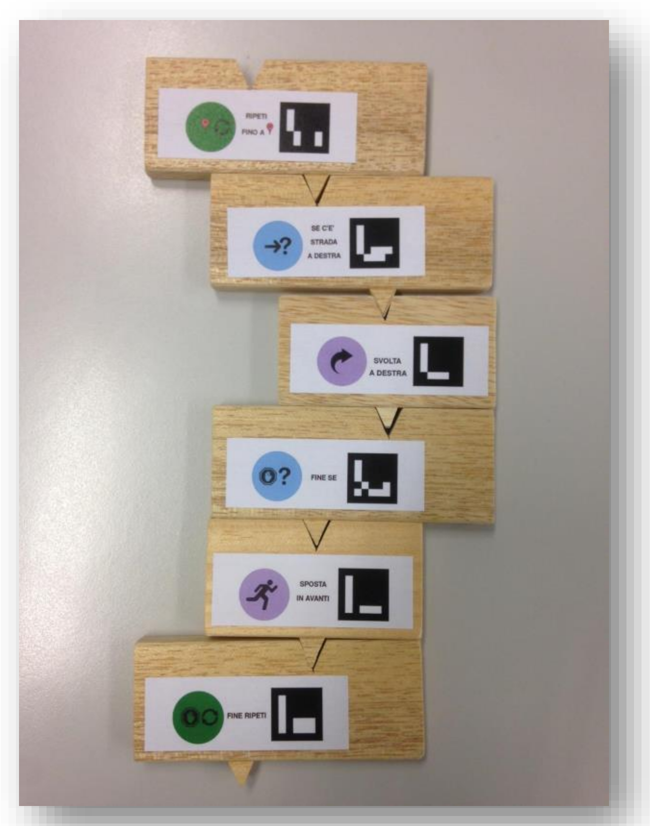
# Informatica e Laboratorio di Programmazione

# linguaggi

*Alberto Ferrari*

# linguaggi formali

- linguaggi ***formali***
  - linguaggi di ***programmazione***
  - linguaggi di ***marcatura*** (es. HTML, Latex)
  - ***interazione*** uomo macchina (es. ricerca Google)
  - linguaggi nel software di sistema
    - ***compilatori***
    - ***interpreti*** …




*http://www.ce.unipr.it/~aferrari/codowood*

# descrizione di un linguaggio

- ***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

# alfabeti e stringhe

- ***alfabeto* Σ**: insieme di simboli
- stringa **s**: sequenza di simboli di **Σ**
  - **s ∈ Σ***, insieme di tutte le stringhe
  - **ε**: stringa vuota
  - **|s|**: lunghezza della stringa s
- linguaggio **L ⊆ Σ***
  - ***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 ...

# concatenazione di stringhe



- operazione di concatenazione •
  - proprietà associativa: **(x • y) • z = x • (y • z)**
  - non commutativa: **x • y ≠ y • x**
  - **Σ*** chiuso rispetto alla concatenazione: **Σ* • Σ* → Σ***
- potenza
  - $x^n = x \cdot x \cdot x \cdot x \ldots$ (n volte)
- elemento neutro **ε**
  - stringa vuota, **∀ x ∈ Σ*, ε • x = x • ε = x**

- $L_1$ ed $L_2$ linguaggi su **Σ*** (due insiemi di stringhe)
- ***unione***: $L_1 \cup L_2 = \{x \in \Sigma^* : x \in L_1 \lor x \in L_2\}$
- ***intersezione***: $L_1 \cap L_2 = \{x \in \Sigma^* : x \in L_1 \land x \in L_2\}$
- ***complementazione***: $\overline{L_1} = \{x \in \Sigma^* : x \notin L_1\}$
- concatenazione o ***prodotto***: $L_1 \bullet L_2 = \{x \in \Sigma^* : x = x1 \bullet x2, x1 \in L_1, x2 \in L_2\}$
- ***potenza***: $L^n = L \bullet L^{n-1}, n \geq 1; L^0 = \{\varepsilon\}$ *per convenzione*
  - concatenazione di n stringhe qualsiasi di L
- *stella di Kleene*: $L^* = \cup L^n, n = 0..\infty$
  - concatenazione arbitraria di stringhe di L

***L****: *chiusura riflessiva e transitiva di* ***L*** *rispetto a •*

# definizione dei linguaggi

- 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

# espressioni regolari

- dato un alfabeto **Σ**, chiamiamo ***espressione regolare*** una stringa **r** sull'alfabeto **Σ ∪ {+, *, (, ), •, Ø}** tale che:
  - **r = Ø**: linguaggio vuoto; oppure
  - **r ∈ Σ**: linguaggio con un solo simbolo; oppure
  - **r = s + t**: unione dei linguaggi **L(s), L(t)**; oppure
  - **r = s • t**: concatenazione dei linguaggi **L(s), L(t)**; oppure
  - **r = s*:** chiusura del linguaggio **L(s)**
  - *(con s e t espressioni regolari; simbolo • spesso implicito)*

*esempio* ***Σ = {a, b}: a • (a + b)* • b***

# espressioni regolari



- 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

# regex – classi di caratteri

- **[…]** per includere uno qualsiasi dei caratteri in parentesi
- singoli caratteri o intervalli di caratteri adiacenti
  - **[A-Z]** = qualsiasi lettera maiuscola
  - **[a-zABC]** = 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]

# regex – caratteri speciali

- **.** 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

# regex - ripetizioni

- **{…}** 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

# regex - esempi

- 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>
```

# regex in Python

- modulo standard re
- le espressioni regolari vengono ***compilate*** 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*

# regex - esempio



```
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)
```

# regex - esempio

```
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 ')
```

# regex - esempi



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

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

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

recursively enumerable
context-sensitive
context-free
regular

approccio generativo

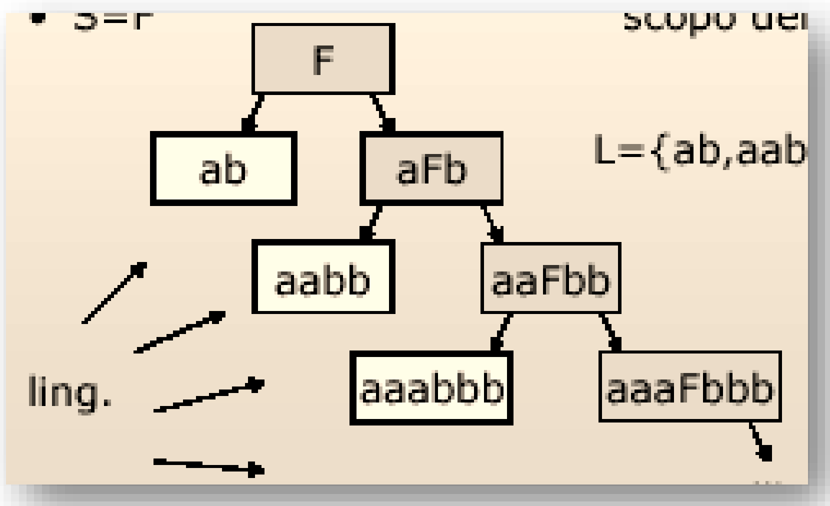
# grammatiche di Chomsky

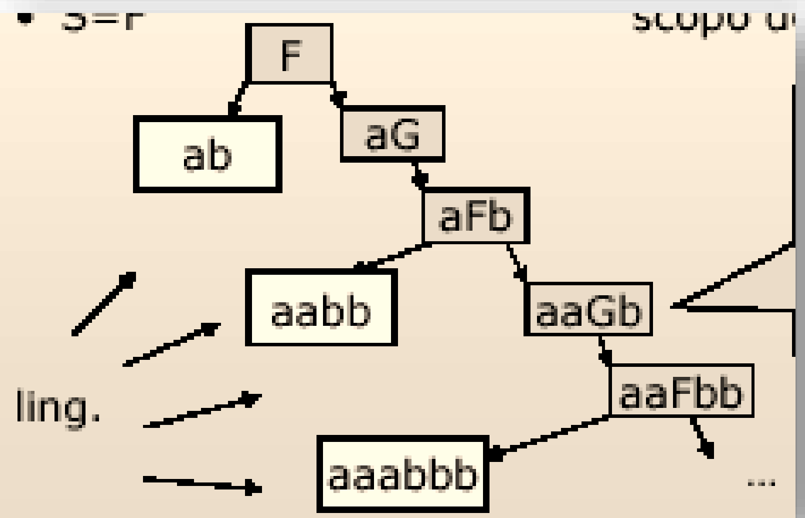
- grammatica **G = < $V_T$, $V_N$, P, S >**
- **$V_T$** : alfabeto finito di simboli ***terminali***
- **$V_N$** : alfabeto finito di simboli ***non terminali***
  - (variabili, categorie sintattiche)
- **$V = V_T \cup V_N$**
- **P**: insieme di produzioni
  - **$\langle\alpha, \beta\rangle \in P$** 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*

# albero delle frasi



- 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:
  - $G_1$ = <{a,b}, {F}, $P_1$, F>
  - $P_1$ = {F → ab, F → aFb}
- secondo esempio:
  - $G_2$ = <{a,b}, {F,G}, $P_2$, F>
  - $P_2$ = {F → ab, F → aG, G → Fb}



# esempio grammatica context-free

- ```
  G = <{a, b, +, *, -, (, )}, {E, I}, P, E>
  ```
  ```
  (1) E → I
  (2) E → E + E
  (3) E → E * E
  (4) E → E - E
  (5) E → -E
  (6) E → (E)
  (7) I → a
  (8) I → b
  (9) I → Ia
  (10)I → Ia
  ```
- … genera espressioni aritmetiche con operatori +,*,- unario e binario e simboli a b
  ```

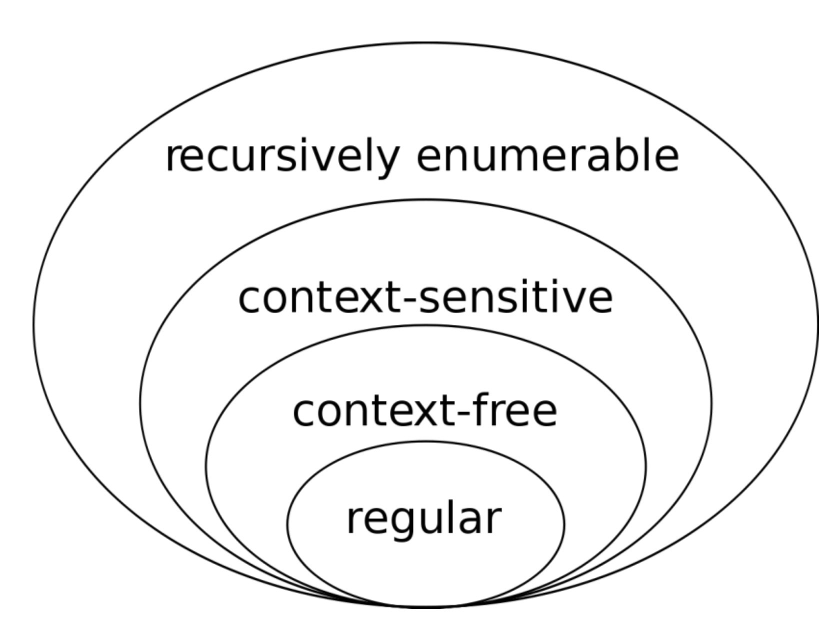
# esempio grammatica context-sensitive



- **G = <{a, b, c}, {S, B, C}, P, S>**
  - **(1) S → aSBC**
  - **(2) S → aBC**
  - **(3) CB → BC**
  - **(4) aB → ab**
  - **(5) bB → bb**
  - **(6) bC → bc**
  - **(7) cC → 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

# classificazione di Chomsky

- ***tipo 0***: grammatiche ***ricorsivamente enumerabili*** (***RE***)
  - **αAβ → γ** (non limitate)
- ***tipo 1***: grammatiche ***contestuali*** (***CS***)
  - **αAβ → αγβ**
- ***tipo 2***: grammatiche ***non contestuali*** (***CF***)
  - **A → γ**
- ***tipo 3***: grammatiche ***regolari*** (***REG***)
  - **A → aB**, oppure **A → b**, oppure A → **ε**
  - coincide con classe dei linguaggi definiti da ***regex***




***A, B ∈ $V_N$; a, b ∈ $V_T$; α, β, γ ∈ V****

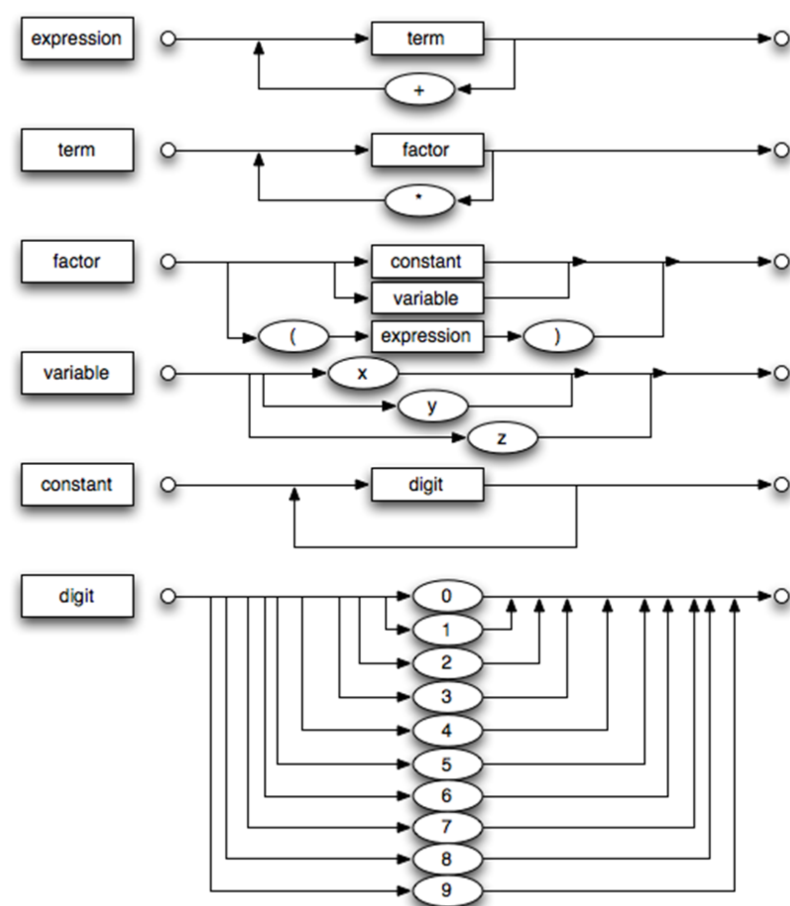
# linguaggi di programmazione

# linguaggio 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***

# sintassi

- 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***



# semantica

- 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

# semantica
# nei linguaggi di programmazione



- 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 operazionale***
  - 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

# linguaggi di alto livello

- 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

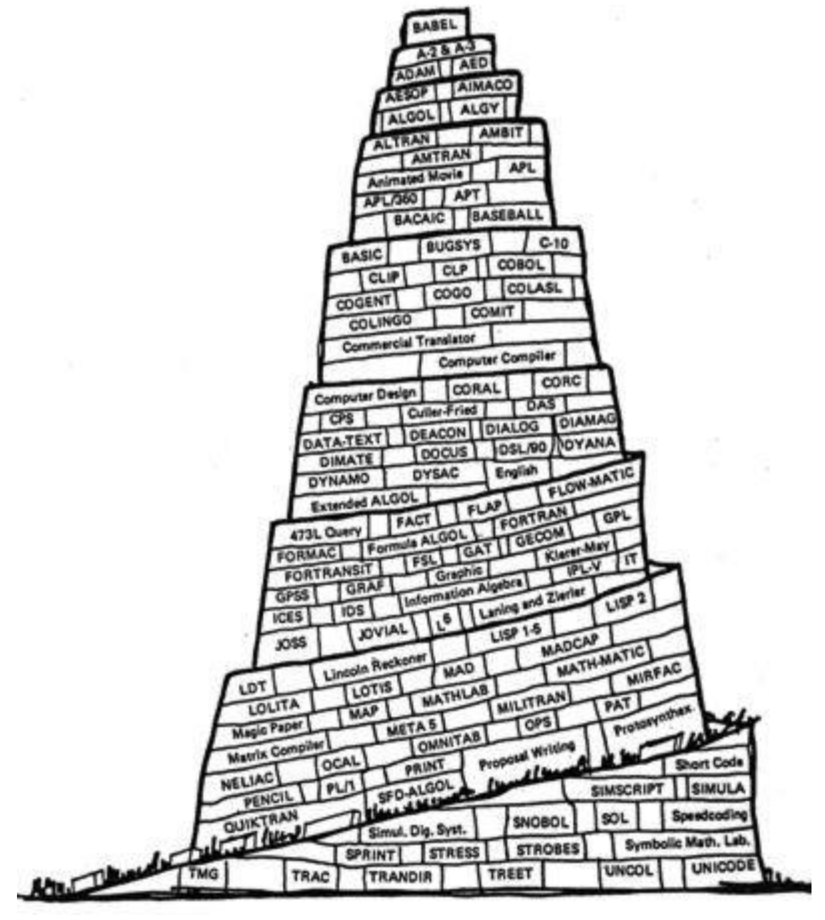
Python 29.8%
Java 25.8%
C++ 12.6%
Ruby 9.6%
PHP 7.3%
C 4.9%
Javascript 3.9%
C# 2.5%
Perl 2%
Clojure .8%
Scala .6%
Objective C .1%
TCL .02%

| 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



# paradigmi di programmazione

- 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*)

# linguaggi e paradigmi



- ***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...

# paradigma logico - prolog



- il ***paradigma logico*** utilizza la ***logica del primo ordine*** *(clausole di Horn)* sia per rappresentare sia per elaborare l'informazione
  - ***assiomi*** e ***regole di inferenza***
- si descrive la struttura logica del problema piuttosto che il modo di risolverlo
  - focus sugli aspetti logici del problema e sul modo migliore per rappresentarli
- ***prolog*** *(**pro**grammation en **log**ique)*
  - ideato da Alain Colmerauer negli anni '70
  - linguaggio di programmazione di tipo ***dichiarativo***
    - il programmatore dice ***cosa*** fare e ***non come*** farlo

- un programma prolog è composto da
  - ***fatti*** (affermazioni sempre vere)

```
padre(aldo,giovanni).
padre(aldo,lucia).
madre(giulia,giovanni).
madre(giulia,lucia).
```

  - ***regole***

```
genitore(X,Y) :- padre(X,Y).
genitore(X,Y) :- madre(X,Y).
fratello_sorella(X,Y) :-
padre(Z,X), padre(Z,Y), madre(W,X), madre(W,Y), X \= Y.
```

  - ***domande (goal)***

```
?- padre(luca,pippo)                    -- YES
?- fratello_sorella(X,giovanni)         -- X=lucia
```

- per soddisfare un ***goal*** (rispondere a una domanda)
  - cerca un ***fatto*** che unifica il goal
    - identificare una sostituzione delle variabili che rende identici il fatto e il goal
    - se un fatto viene trovato, il goal è ***soddisfatto***
  - cerca una ***regola*** la cui testa (parte sinistra) unifica il goal
    - se la trova, cerca di soddisfare tutti i sotto-goal di cui è composta la regola (parte destra)
    - usa una stessa sostituzione delle variabili per tutti i sotto-goal

*https://swish.swi-prolog.org/*

# example: mortal philosophers



*consider the following argument:*

```
all men are mortal.
Socrates is a man.
-------------------------
hence, Socrates is mortal.
```

*it has two premises and a conclusion*

*the two premises can be expressed as a little prolog program:*

```
mortal(X) :- man(X).
man(socrates).
```

the conclusion can then be formulated as a query:

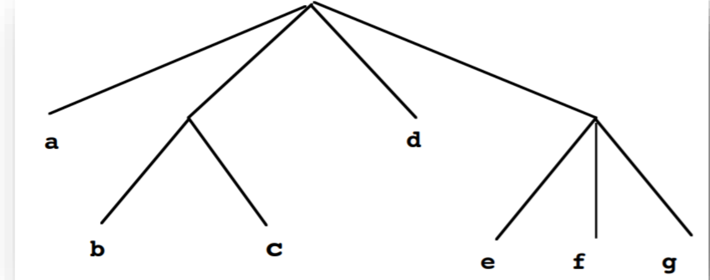
```
?- mortal(socrates).
Yes
```

# paradigma funzionale

- paradigma di programmazione in cui l'operazione fondamentale è l'applicazione di ***funzioni*** ai loro argomenti
- un programma è una funzione, definita in termini di ***altre funzioni***, che elaborano l'input e restituiscono il risultato finale
- le funzioni sono prive di effetti collaterali (no side effects)
  - ⇒ ***funzioni pure***
- le funzioni sono trattate come un qualsiasi altro tipo di dato
  - ⇒ ***first class objects***
- la ***ricorsione*** è la principale struttura di controllo
  - non esistono costrutti iterativi
- ***high order functions***: funzioni che operano su funzioni, che operano su funzioni, …

# paradigma funzionale - lisp

- ***lisp*** *(**list** **p**rocessing)* è un linguaggio di programmazione (*John McCarthy 1958*) per il trattamento di ***liste*** e, in generale, di informazione non numerica
  - utilizzato in progetti di intelligenza artificiale
- le liste in lisp sono in realtà alberi con l'informazione (i simboli atomici) memorizzata nelle foglie
  - es. la lista (a (b c) d (e f g)) corrisponde all'albero in figura



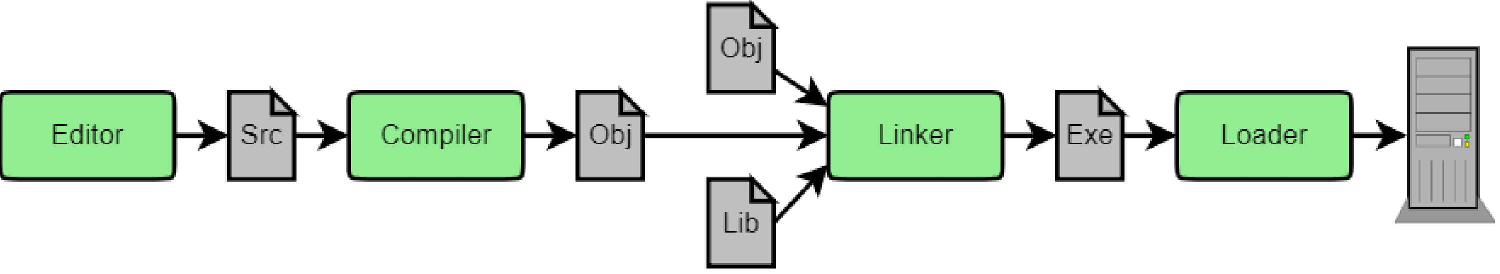

- un ***programma*** lisp è una ***lista***
  - è possibile scrivere programmi che ricevono in input altri programmi

```
(defun mylength (lst)
    (cond ((null lst) 0)
          ((atom lst) nil)
          (t (+ 1 (mylength (cdr lst))))))
```

*funzione mylength che, data una lista lst, ne calcola il numero di elementi*

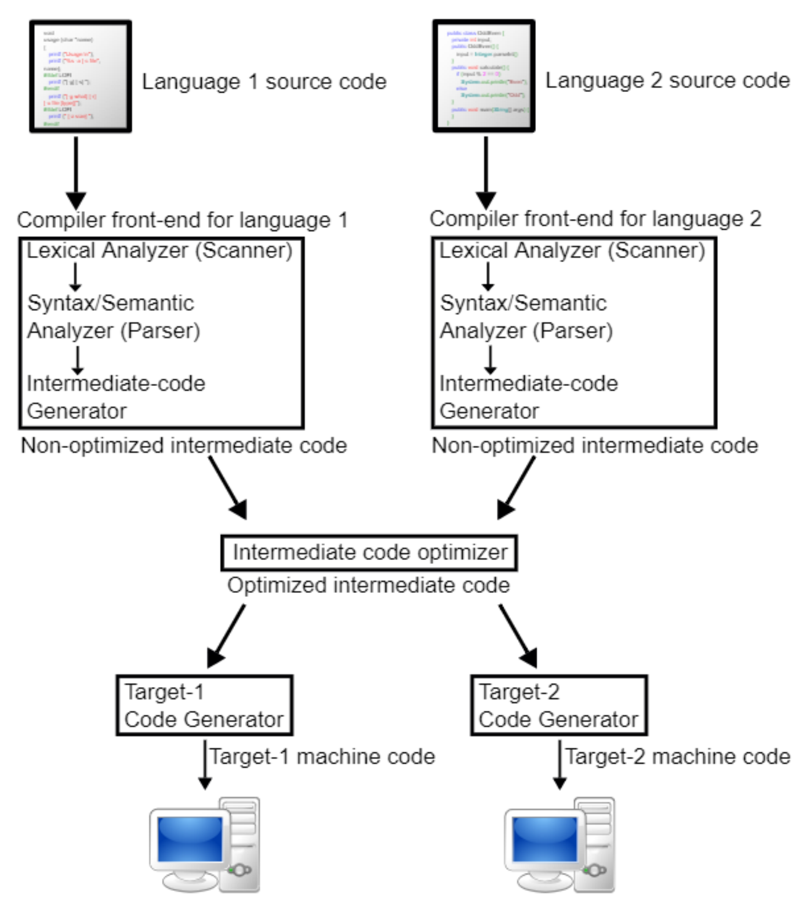
# esecuzione dei programmi

- 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...
  - a ***tutto il codice***; prima dell'esecuzione
- programmi ***interpretati***: applicati i 3 passi...
  - in sequenza, ***su ogni istruzione***; a tempo di esecuzione



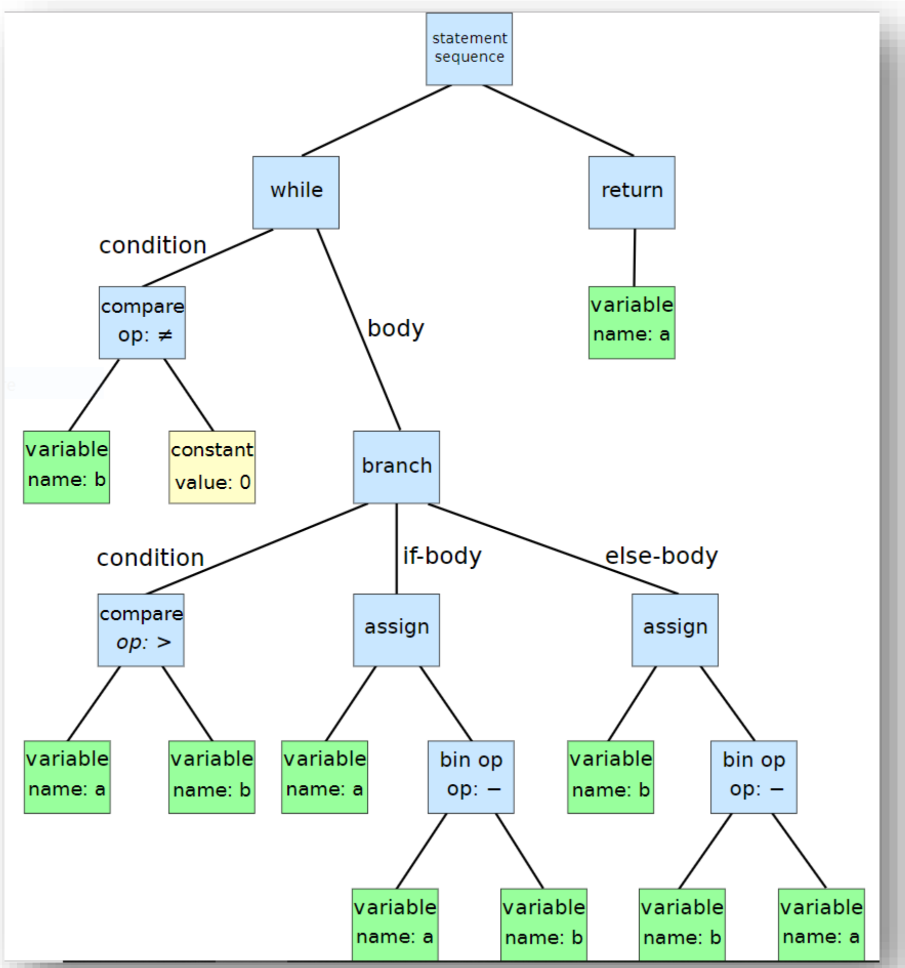
# processo di compilazione

- 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*



# collegamento

- 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
- rilocazione ***statica***
  - indirizzi logici trasformati in ***indirizzi assoluti***
- rilocazione ***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

# codice gestito

- 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