CAPITOLO 1:

Macchina fisica = consente di eseguire algoritmi formalizzati(tradotti) perché siano “comprensibili” dall’esecutore.

- Formalizzazione: decodifica di algoritmi in un certo linguaggio (L)definito da una specifica sintassi.
- La sintassi di L permette di utilizzare i costrutti per comporre programmi in L.
- Un programma in L è una sequenza di istruzioni del linguaggio L.

-Macchina astratta per L(ML) = Un insieme di algoritmi e strutture dati che permettono di memorizzare ed eseguire programmi scritti in L.
Composta da: Una memoria per immagazzinare dati e programmi e Un interprete che esegue le istruzioni contenute nei programmi.

-Linguaggio Macchina = Data una macchina astratta ML, il linguaggio L "€ " dall'interprete di ML è detto linguaggio macchina di ML.

-Interprete = è un programma che si occupa del controllo della sequenza delle istruzioni del linguaggio, controllo dei dati e gestire la memoria, es: va a prendere i valori delle variabili dalla memoria.

+Operazioni interprete:

+ Elaborazione dei dati primitivi(int,real,aritm);
+ Operazioni e strutture dati per il controllo della sequenza di esecuzione;
+ Operazioni e strutture dati per il controllo del trasferimento dei dati;
+ Operazioni e strutture dati per la gestione della memoria;

-Funzione parziale = Una funzione parziale f: A->B è una corrispondenza tra elementi dell'insieme A e quelli dell'insieme B

-Compilatore= Un compilatore da L a L0 è un programma che realizza la funzione:
Cl,L0 : Prog^L -> Prog^L0
Per eseguire PL su Input, bisogna eseguire CL,L0 con PL come input.
Si avrà come risultato un programma compilato PcL0 scritto in L0, che sarà eseguito su M0L0 con il dato in ingresso Input

-Funzione parziale Calcolabile= Una funzione parziale f: A ⍰ B è calcolabile nel linguaggio L se esiste un programma P scritto in L tale che:

+ Se f(a) = b allora P(a) termina e produce come output b;
+ Se f(a) non è definita allora P con input a va in ciclo (si, esistono funzioni non calcolabili);

-Macchine di turing (Mdt)= Permettono di verificare se la funzione è calcolabile per tutti i linguaggi.
Una MdT è definita da una quintupla: M = (X, Q, fm, fd, δ)
X = insieme finito di simboli (comprende il blank ovvero cella vuota)
Q = insieme finito di stati (comprende HALT che definisce la terminazione)
fm (funzione macchina) :Q x X-> X (Determina il simbolo da scrivere sul nastro)
fd (funzione direzione): Q x X -> {S,D,F} (spostamento testina, S=Sinistra, D=Destra, F=Ferma)
Funzione di transizione di stato δ: Q x X -> Q (Definisce lo stato successivo della computazione)

-La MdT Universale (MdTU)= Legge dal nastro i dati e il programma (anche le 5-ple).
è una macchina programmabile. (fetch->decode->execute) è un interprete

-Un linguaggio di programmazione L = è un formalismo per portare al livello di macchina fisica gli algoritmi, implementare L significa realizzarne l'interprete ovvero il programma che esegue la traduzione di L per la macchina ospite.

-BNF = Notazione usata per descrivere la sintassi dei linguaggi di programmazione.

-Semantica= corretto significato.

-Sintassi= corretta struttura grammaticale. (per i linguaggi importante)

-Grammatiche libere da contesto (C.F.) = Una classe importante di regole che generano linguaggi formali è un LINGUAGGIO DALLE PARENTESI BEN FORMATE che comprende tutte le stringhe di parentesi aperte e chiuse bilanciate correttamente:
i) La stringa è ben formata;
ii) se la stringa di simboli A è ben formata, allora lo è anche A ;
iii) se le stringhe A e B sono ben formate, allora lo è anche la stringa AB.
Regole di produzione / produzioni =
1: S -> ()
2: S -> (S)
3: S -> SS
Stabiliscono che "data una stringa si può formare una nuova stringa sostituendo
una S o con () o con (S) o con SS".
-Derivazione = Sequenza di applicazioni di regole di produzione: 3 -> 2 -> 3 -> 1 -> 1 -> 2 -> 1
=> Con il numero sotto si dice produce direttamente per applicazione della regola di riscrittura n.
=> Con * sopra si dice es : S produce (())(()()).
-Simboli non terminali / variabili = caratteri che non possono apparire nelle stringhe finali.
-Simboli terminali = caratteri che possono apparire nelle stringhe finali.

CAPITOLO 2

- Abeto= Insieme X finito e non vuoto di simboli.
- Parola= Sequenza finita di simboli x1x2...xnm dove ogni xi è preso da uno stesso αbeto X è una parola su X.
 + La lunghezza di una stringa w è denotata con |w|.
 + La parola vuota denotata con λ = stringa priva di simboli e ha lunghezza 0.
- Uguaglianza tra stringhe= Due stringhe sono uguali se i loro caratteri, letti ordinatamente da sinistra a destra, coincidono.
- X*= L'insieme di tutte le stringhe di lunghezza finita sull'αbeto X si denota con X*.
(Ha un numero di elementi che è un infinito numerabile).
- Concatenazione o prodotto= Sia α € X* una stringa di lunghezza m e β € X* una stringa di lunghezza n, la concatenazione di α e β, denotata con αβ o α per β, è definita come la stringa di lunghezza m+n, i cui primi m simboli costituiscono una stringa uguale ad α ed i cui ultimi n simboli costituiscono una stringa uguale a β.
 + Quindi se α = x1x2...xm e β = x1'x2'..xn' si ha : αβ= x1x2..xmx1'x2'...xn'.
-Operazione di concatenazione = La concatenazione di stringhe su X è una operazione binaria su X*:
 * : X* x X* -> X*.
 + è associativa: (αβ)δ= α(β δ) = αβ δ compresi tutti in X.
 + non è commutativa: Esiste α,β € X* : αβ ≠ βα
 + ha elemento neutro lamba: λα = αλ = α α € X*.
Quindi (X*,.) è un monoide.
- Prefisso/suffisso= Sia δ € X*. se δ = α β, con α, β € X*, allora α è detto prefisso di δ e β è detto suffisso di δ.
- Sottostringa= Sia δ € X*, se δ = α β γ, con α, β, γ € X*, allora β è detta sottostringa di δ.
- Potenza di una stringa= data una stringa α su X, la potenza h-esima di α è λ se h= 0 oppure αα^h-1 con h ≠ 0.

-Linguaggio formale= Un linguaggio formale L su un alfabeto X è un sottoinsieme proprio di X*.
L ⊆ X*
- Una grammatica generativa è una quadrupla G = (X, V, S, P) con:
 + X alfabeto di simboli terminali;
 + V insieme finito di simboli nonterminali/variabili;
 + S simbolo di partenza per la grammatica;
 + P insieme di produzioni della grammatica dove valgono:
 + X ∩ V = Ø (X e V disgiunti), S € V.

-Una produzione= è una coppia (v, w) indicata con v -> w dove v € $(X \cup V)^+$ e w € $(X \cup V)^*$.

-Derivazione/produzione diretta= diciamo che y produce direttamente z se : α -> β € P
ossia se esiste in G una produzione α -> β.
-Derivazione/produzione= y=>*z y produce z se y = z o esiste una sequenza di stringhe.
- =>* = è la chiusura riflessiva e transitiva della relazione di derivazione diretta.
- =>+ = è la chiusura transitiva della stessa relazione.

-Linguaggio generato da G= denotato con L(G), è l'insieme delle stringhe di terminali derivabili dal simbolo di partenza S. L(G) = { w € X* | S =>* w}

-Forma di frase= Sia G=(X,V,S,P) una grammatica, una stringa w € (X ∪ V)* è una forma di frase se: S=>*w.
-Grammatiche equivalenti= Due grammatiche G e G' sono dette equivalenti se L(G) = L(G').

CAPITOLO 3:

- Grammatica context-free= Una grammatica G = (X,V,S,P) è libera da contesto se per ogni produzione v -> w, v è un non terminale.

-Linguaggio context-free= Un linguaggio L su un alfabeto X è libero da contesto se può essere generato da una grammatica libera da contesto.

-Grammatica context-sensitive= Una grammatica G = (X,V,S,P) è dipendente da contesto se ogni produzione è in una delle seguenti forme:
+ yAz -> ywz, A può essere sostituita con w nel contesto sinistro y e destro z.
+ S -> λ purchè S non compaia nella parte destra di alcuna produzione.

-Linguaggio context-sensitive= Un linguaggio L è dipendente da contesto se può essere generato da una grammatica dipendente da contesto.

-Grammatica monotona= Una grammatica G = (X,V,S,P) è monotona se ogni sua produzione è monotona cioè se: Per ogni v -> w € P : |v| <= |w|

-Linguaggio monotono= Un linguaggio L è monotono se può essere generato da una grammatica monotona.

CAPITOLO 4:

-Albero di derivazione= Albero di derivazione per w l'albero Tw avente le seguenti proprietà:
+ la radice è etichettata con il simbolo iniziale S;
+ ogni nodo interno è etichettato con un simbolo di V (un nonterminale);
+ ogni nodo foglia è etichettato con un simbolo di X (un terminale) o λ;
+ Un nodo N etichettato A, N ha k discendenti (N1,N2,..Nk) etichettati con i simboli (A1,A2,..,Ak) allora la produzione A -> A1 A2 ...Ak deve appartenere a P;
+ La stringa w è ottenuta leggendo e concatenando le foglie dell'albero da sinistra a destra;

-Lunghezza di un cammino= Dato un albero di derivazione, la lunghezza di un cammino dalla radice ad una foglia è data dal numero di nonterminali su quel cammino.

-Altezza= L'altezza di un albero è data dalla lunghezza del suo cammino più lungo.

-Derivazione destra (sinistra) = Data una grammatica G = (X,V,S,P) diremo che una derivazione S=>w, ove:
S=>w1=>w2=>..=>wn = w; wi= yiAzi, wi+1 = yiwi'zi; i=1,2,..,n-1; è destra (sinistra) se, per ogni i, i=1,2,..,n-1 risulta : zi € X* (yi € X*)

-Principio sostituzione sottoalbero=Supponiamo di avere un albero di derivazione Tz per una stringa z di terminali generata da una grammatica C.F. e supponiamo che il simbolo non terminale A compaia 2 volte sullo stesso cammino.
Il sottoalbero più in basso con radice nel nodo etichettato con una A genera la sottostringa w, mentre quello più in alto genera la sottostringa vwx.
Poiché la G è C.F., sostituendo il sottoalbero più in alto con quello più in basso, si ottiene ancora una derivazione valida.
Il nuovo albero genera la stringa uwy.
Se effettuiamo la sostituzione inversa (il sottoalbero più in basso viene rimpiazzato da quello più in alto), otteniamo un albero di derivazione lecito per la stringa uvvwxxy, ossia uv2wx2y.
Ripetendo questa sostituzione un numero finito di volte, si ottiene l'insieme di stringhe:
{uv^nwx^ny | n ≥ 0}

-Lemma= Sia G= (X, V, S, P) una grammatica C.F. e supponiamo che m= max {|v| | A->v € P}
Sia Tw un albero di derivazione per una stringa w di L(G), se l'altezza di Tw è al più uguale ad un intero j allora: |w|<=m^j
+in formule: height(Tw)<= J => |w|<=m^j

-Pumping lemma=Sia L un linguaggio C.F. allora esiste p,che dipende da L, t.c. se z € L di lunghezza |z|>p, allora z può essere scritta come uvwxy t.c.:
+ |vwx| <= p
+ vx ≠ λ
+ Per ogni i>=0: uv^iwx^iy € L

-Grammatica Ambigua= Una grammatica G libera da contesto è ambigua se esiste una stringa x in L(G) cha ha due alberi di derivazione differenti.
Oppure se esiste una stringa x in L(G) che ha due derivazioni sinistre(destre) distinte.

-Linguaggio inerentemente ambiguo= Un linguaggio L è inerentemente ambiguo se ogni grammatica che lo genera è ambigua.

N.B: Se una grammatica genera parole la cui lunghezza cresce in maniera esponenziale (o lineare o quadratica o cubica o…), allora il linguaggio generato non è libero

CAPITOLO 5:

- Gerarchia di chomsky (1956-1959) = Sia G= (X,V,S,P) una grammatica, a seconda delle restrizioni imposte sulle regole di produzione si distinguono le varie classi di grammatiche:
 + Tipo ‘0’ - Quando le stringhe che appaiono nella produzione v->w non sono soggette ad alcuna limitazione.
 + Tipo ‘1’ - Dipendente da contesto - quando le produzioni sono limitate alla forma:
  (1) yAz->ywz, con A € V, y,z € (XuV)*, w € (XuV)^+
  (2) S->λ, purché S non compaia nella parte destra di alcuna produzione
 + Tipo '2'- libera da contesto - quando le produzioni sono limitate alla forma: v->w con v € V
 + Tipo '3'- lineare destra - quando le produzioni sono limitate alla forma:
  (1) A->bC con A,C € V e b € X;
  (2) A->b con A € V e b € Xu{λ}

-Teorema della gerarchia= Li = {L c X* | L=L(G), G di tipo i} ovvero : L3 ⊂ L2 ⊂ L1 ⊂ L0

-Lemma della stringa vuota= Sia G=(X,V,S,P) una grammatica C.F. con almeno una λ-produzione, allora esiste una grammatica C.F. G' t.c:
 + L(G)=L(G')
 + Se λ non € L(G) allora in G' non esistono produzioni del tipo A->Λ
 + Se λ € L(G) allora in G' esiste un'unica produzione S'->λ, ove S' è il simbolo iniziale di G' ed S' non compare nella parte destra di alcuna produzione di G'.

-Operazioni sui linguaggi= Siano L1 ed L2 due linguaggi definiti su uno stesso alfabeto X (L1,L2 ⊆ X*).
 + L'unione di L1 ed L2 è: L1 U L2 = {w|w € L1 v w € L2}
 + La concatenazione di L1 ed L2 è: L1 * L2 = {w|w=w1w2, w1 € L1, w2 € L2}
 + L'iterazione di L1 (chiusura riflessiva transitiva di L1 rispetto concatenazione) è:
  L1* = {w | w = w1w2..wn, n>=0 e per ogni i: wi € Li}
 + Complemento di L1 è: NOT L1 = X* - L1
 + ∩ di L1 ed L2 è: L1 ∩ L2 = {w|w € L1 ^ w € L2 }

- Potenza di un linguaggio= Sia L un linguaggio definito su un alfabeto X, dicesi potenza n-esima di L, si denota con L^n, n>=0, il seguente linguaggio:
 + L^n = {λ} con n=0
 + L^n = L^n-1 * L altrimenti posto L^+ = ∪ di i>=1 L^i
Si ha; L* = {λ} ∪ L^+ = L^0 u L^+ = U L^i, i>=0

-Proprietà di chiusura delle classi di linguaggi=
 + L linguaggio definito su X <=> L ⊆ X* <=> L € 2^X*
 + Lgotico classe di linguaggi su X <=> Lgotico ⊆ 2^X* <=> Lgotico € 2^2^X*

-Definizione di chiusura= Sia Lgotico una classe di linguaggi su X.
+ Sia α un'operazione binaria sui linguaggi di Lgotico: A : 2^X* X 2^X* -> 2^X*,
(L1,L2) |-> α(L1,L2)
+ Sia β un'operazione unaria sui linguaggi di Lgotico: B: 2^X* -> 2^X*, L |-> B(L)

Lgotico è chiusa rispetto ad α <=> per ogni L1,L2 € Lgotico : α(L1,L2) € lgotico
Lgotico è chiusa rispetto a β <=> per ogni L1 € Lgotico : B(L1) comrpeso lgotico

-Teorema di chiusura= La classe dei linguaggi di tipo i = 0,1,2,3 è chiusa rispetto alle operazioni di unione, concatenazione ed iterazione.

-Stringa riflessa= Sia w una parola su un alfabeto X = {x1,x2,...,xk}, w= xi1xi2...xin-1 xin. Dicesi stringa riflessa di w la stringa: w^R = xin xin-1 .... xi2 xi1

-Operazione di riflessione= Sia w una parola su un αbeto X = {x1,x2,...,xk} e sia w^R la stringa riflessa di w. L'operazione che trasforma w in w^R è detta operazione di riflessione.

-Parola palindromica= Un palindromo è una parola la cui lettura a ritroso riproduce la parola di partenza: w palindromo <=> w = w^R.

-Teorema= sia w una parola su un alfabeto X. w è palindromo se e solo se w= α x α^R, x € X u {λ}

CAPITOLO 6

- Automa/Accettatore a stati finiti (FSA)= Sia X un alfabeto, un automa a stati finiti è una quadrupla:
M=(Q,δ,q0,F) ove:
+ X è detto alfabeto di ingresso;
+ Q è un insieme finito e non vuoto di stati;
+ δ è una funzione da in Q, detta funzione di transizione: δ: Q x X -> Q
+ q0 è lo stato iniziale;
+ F ⊆ Q è l'insieme degli stati di accettazione o finali.

-Grafo degli stati/diagramma di transizione/diagramma di stato= rappresentazione grafica di un FSA in cui:
+ ogni stato q € Q è rappr. da un cerchio (nodo) con etichetta q;
+ lo stato iniziale (nodo q0) ha un arco orientato entrante libero (non proviene da nessun altro nodo);
+ per ogni stato q € Q e per ogni simbolo x € X, se δ (q,x) = q' esiste un arco orientato etichettato con x uscente dal nodo q ed entrante nel nodo q'.

-Tavola di transizione= può rappresentare la FSA, una tabella in cui ci sono gl istati sulle righe e i simboli dell'alfabeto in ingresso sulle colonne.

- δ* per FSA= Dato un FSA M = (Q,δ,q0,F) con alfabeto di ingresso X, definiamo per induzione la funzione: δ*:Q x X* -> Q, dove δ*(q,w) per q € Q e w € X*, sia lo stato in cui M si porta avendo in ingresso la parola w su X e partendo dallo stato q.
δ*(q,lamda) = q;
δ* (q,wx) = δ (δ* (q,w),x) per ogni q € Q, x € X, w € X*.

-Parola accettata FSA= Sia M = (Q,δ,q0,F) un FSA con alfabeto in ingresso X una parola w € X* è accettata da M se partendo da q0, lo stato q in cui l'automa si porta alla fine della sequenza di ingresso w è uno stato finale.

-Linguaggio accettato FSA= Sia M = (Q,δ,q0,F) un FSA con alfabeto di ingresso X, il linguaggio accettato da M è il seguente sotto insieme di X*:
+ T(M) = {w € X* | δ* (q0,w) € F} - insieme delle parole accettate da M.

-FSA equivalenti= Sia M1 = (Q1,δ1,q1,F1) ed M2 = (Q2,δ2,q2,F2) due FSA di alfabeto di ingresso X. M1 ed M2 si dicono equivalenti se: T(M1) = T(M2)

-Linguaggi a stati finiti= Dato un alfabeto X, un linguaggio L su X è un linguaggio a stati finiti se esiste un FSA M con alfabeto di ingresso X t.c. L= T(M)

-Classe dei linguaggi FSA= Lgotico FSL = {L € 2^X* | ∃M, M è un FSA : L = T(M)}

-Automa/accettatore a stati finiti non deterministico(NDA) = Un NDA con alfabeto di ingresso X è una quadrupla M=(Q,δ,q0,F) ove:
+ Per Q,q0 e F valgono le stesse cose dell' FSA;
+ δ: Q x X -> 2^Q è la funzione di transizione che assegna ad ogni coppia (q,x) un insieme δ (q,x) ⊆ Q di possibili stati successivi.

-δ* per NDA= Dato un NDA M=(Q,δ,q0,F) con alfabeto X, definiamo per induzione la funzione δ*: 2^Q x X*-> 2^Q
+ δ*(p,λ) = p
+ δ*(p,wx) = $\bigcup_{q \in \delta^*(p,w)} \delta(q,x)$ per ogni p € 2^Q (p sott. Q) x € X w € X*

-Parola accettata NDA = Sia M = (Q,δ,q0,F) un NDA con alfabeto di ingresso X una parola w € X* è accettata da M se, partendo dallo stato iniziale q0, esiste almeno un modo per M di portarsi in uno stato di accettazione alla fine della sequenza di ingresso w.

-Linguaggio accettato NDA= Sia M = (Q,δ,q0,F) un NDA con alfabeto di ingresso X, il linguaggio accettato da M è l'insieme delle parole su X accettate da M: T(M) = { x € X* | δ* ({q0},w) n F ≠ 0}

-NDA Equivalenti = Siano M1 = (Q1,δ1,q1,F1) ed M2 = (Q2,δ2,q2,F2) due NDA di alfabeto di ingresso X, M1 ed M2 si dicono equivalenti se: T(M1) = T(M2)

-Classe dei linguaggi NDA= LgoticoNDL = { L € 2^X* | Esiste M, M è un NDA : L=T(M) }

-Teorema= Le classi dei linguaggi Lgotico FSL e Lgotico NDL sono equivalenti.

CAPITOLO 7

-Linguaggio regolare= Sia X un alfabeto, un linguaggio L ⊆ X* è regolare se:
+L è finito: |L| = k , k intero;
+L può essere ottenuto per induzione utilizzando una delle seguenti op:
+ L = L1 u L2 con L1 ,L2 regolari;
+ L = L1 per L2 con L1,L2 regolari;
+ L = L1*, con L1 regolare; 0 e {λ} sono regolari

-Espressioni regolari= Sia X un alfabeto una stringa R di alfabeto X u {λ,+,*,., 0, (,)} con (X n {λ,+,*,., 0, (,)}= 0) è una espressione regolare di alfabeto X se e solo se vale una delle seguenti condizioni:
+ R = 0
+ R = λ
+ R= a per ogni a € X
+ R = (R1 + R2) con R1,R2 espressioni regolari di alfabeto X
+ R = (R1 per R2) con R1,R2 espressioni regolari di alfabeto X
+ R = (R1)* con R1espressioni regolari di alfabeto X
Ad ogni espressione regolare corrisponde un linguaggio regolare S(R) definito

-Classe dei linguaggio REG= Lgotico REG = {L € 2^X* | Esiste R € R gotico; L = S(R)}

-Espressioni regolari equivalenti= Due espressioni regolari R1 e R2 su X sono equivalenti se e solo se S(R1) = S(R2)

-Proprietà delle espressioni regolari=
+ Ass: (R1+R2) + R3 = R1 + (R2 +R3) = R1 + R2 + R3
+ Comm: R1 + R2 = R2 + R1
+ 0 El. neutro +: R1 + 0 = 0 + R1 = R1
+ Idempotenza: R1 + R1 = R1
+ Prop.ass per: (R1 * R2) * R3 = R1 * (R2 * R3) = R1 * R2 * R3 + R1 * R2 ≠ R2 * R1
+ λ El. neutro *: R1 * λ = λ * R1 = R1
+ 0 El. assorbente *: R1 * 0 = 0 * R1 = 0
+ distributiva * sinistra : R1 * (R2+R3) = (R1*R2) + (R1 * R3) + distributiva * destra : (R1 + R2) *R3 = (R1*R3) + (R2 * R3)
+ (R1)* = (R1)* * (R1)* = ((R1)*)* = (λ +R1)*
+ (0)* = (λ)* = λ.
+ (R1)* = λ + R1 + R1^2 + ... + R1^n + (R1^n+1 * R1*) caso particolare:
(R1)* = λ + R1 * R1* = λ R1* * R1
+ (R1 + R2)* = (R1* + R2*)* = (R1* * R2*)* = (R1* * R2)* * R1* = R1* * (R2 * R1*)*
+ (R1 + R2)* ≠ R1* + R2*
+ R1* * R1 = R1 * R1*
+ R1 * (R2 * R1)* = (R1 * R2)* * R1
+ (R1* * R2)* = λ + (R1 + R2)* * R2
+ (R1 * R2*)* = λ + R1* (R1 + R2)*
Importante!
+ R1 = R2 * R1 + R3 se e solo se R1 = R2* * R3
+ R1 = R1 * R2 + R3 se e solo se R1 = R3 * R2*

-Teorema di kleene= Equivalenza tra Linguaggi lineari destri, linguaggi a stati finiti e linguaggi regolari Abbiamo fatto Lgotico3 = LgoticoFSL (e viceversa soltanto)

-Pumping lemma per linguaggi regolari= Sia L= T(M) un linguaggio regolare con M = (Q,δ,q0,F) un automa a stati finiti, allora Esiste n = |Q| t.c. per ogniz € L, |z| >= n; z = uvw e:
+ |uv| <= n
+ v ≠ λ
+ uv^iw € L, per ogni i>=0

CAPITOLO 8

-Compilatore= Traduce il programma sorgente in programma oggetto, Esegue:
+Analisi del programma sorgente; (diviso in analizzatore lessicale, sintattico e semantico)
+Sintesi del programma oggetto; (diviso in generatore di codice, ottimizzatore di codice)

-Analizzatore lessicale(scanner)= Prende in input il programma sorgente, esamina il programma per individuare i token classificando le parole chiave, identificatori, operatori, cost,ecc..
Ogni classe di token è dato da un numero unico che la identifica,
Il token è una stringa di caratteri, ed è memorizzato in una tabella.
In output produce una lista di token.

-Analizzatore sintattico (parser)= Prende in input la lista di token, Individua la struttura sintattica della stringa in esame a partire dal programma sorgente sotto forma di token.
Identifica quindi espressioni, istruzioni, procedure.
Il controllo sintattico si basa sulle regole grammaticali utilizzate per definire il linguaggio, durante il controllo si genera l'albero sintattico quest'ultimo dato in output

-Analizzatore semantico= Riceve in input l'albero sintattico generato dal parser,
Si compone di due fasi principali:
+ Controlli statici: quando riconosce gli operatori aritmetici, invoca una routine semantica che indica le azioni da svolgere facendo un controllo se hanno stesso tipo / valore ecc..
+ Generazione di una rappresentazione intermedia (IR), Produce una forma intermedia di codice sorgente dove:
Rimuove dall'albero alcune delle categorie intermedie e mantiene solo la struttura essenziale (albero sintattico astratto): i nodi sono i token, le foglie sono gli operandi, i nodi intermedi operatori.
Qui può esserci un ottimizzatore del codice: per la propagazione di costanti o eliminazione di sotto-espressioni comuni/ripetute.
In output produce un Albero arricchito con informazioni sui vincoli sintattici contestuali.

-L'analisi è svolta in fase di compilazione, verifica che i simboli utilizzati siano legali ovvero appartengono all'alfabeto , le regole grammaticali siano rispettate e i vincoli imposti dal contesto siano rispettati.

-Generatore di codice=Riceve in input l' Albero arricchito dell'analizzatore semantico e trasla la forma intermedia in linguaggio assembler o macchina.
Prima però ci sono due fasi di preparazione:
+ Allocazione della memoria: può essere allocata staticamente oppure è uno stack o heap la cui dimensione cambia durante l'esecuzione;
+ Allocazione dei registri: poiché l'accesso ai registri è più rapido dell'accesso alle locazioni di memoria, i valori cui si accede più spesso andrebbero mantenuti nei registri.
-(S), 1(S), 2(S): accede al contenuto al top dello stack, ad una posizione successiva, a due etc..
-@A: accede alla locazione il cui valore è puntato da A.

-Ottimizzatore di codice= In input riceve il codice intermedio creato dal generatore di codice e ci sono due modi di ottimizzarlo:
+ Ottimizzazioni indipendenti dalla macchina: ad esempio la rimozione di istruzioni invarianti all'interno di un loop, fuori dal loop, etc.
+ Ottimizzazioni dipendenti dalla macchina: ad esempio ottimizzazione dell'uso dei registri.

-Altri aspetti importanti della compilazione:
1. Error Detection e Recovery;
2. Le Tabelle dei Simboli prodotte dai vari moduli;
3. La Gestione della Memoria implicata da alcuni costrutti del linguaggio di alto livello.

-Linking e caricamento= Il programma oggetto prodotto dal compilatore contiene una serie di riferimenti esterni, sono risolti dal LINKER.
+ Il programma è rilocabile: può essere allocato in diverse zone di memoria cambiando indirizzo ind (indirizzamento relativo).
+ Fase di caricamento compiuta dal LOADER che assegna un valore numerico all'indirizzo ind, trasformando gli indirizzi relativi in assoluti.

CAPITOLO 9

Lo scanner è un'interfaccia fra il programma sorgente e analizzatore sintattico(parser);
Il parser potrebbe fare direttamente anche l'analisi lessicale, ma non è conveniente in quanto la grammatica per i token è una grammatica di tipo 3, più semplice quindi di quella che tratta il parser

Lo scanner può interagire con il parser lavorando in passo separato producendo i token in una grossa tabella in memoria di massa oppure interagisce direttamente con il parser quando è chiamato da quest'ultimo e quindi è necessario il prossimo token nell'analisi sintattica(preferibile).

Il token è una stringa di caratteri memorizzato in una tabella.

-costant table= i valori delle costanti dello scanner sono memorizzati nella costant table.

-Symbol table= i nomi delle variabili dello scanner sono memorizzati nella symbol table.

-Compiti scanner=
+ Eliminare spazi bianchi, commenti, ecc;
+ Isolare il prossimo token dalla sequenza di caratteri in input;
+ Isolare identificatori e parole-chiave;
+ Generare la symbol-table
+ I token possono essere descritti in diversi modi. Spesso si utilizzano le grammatiche regolari o con gli automi a stati finiti.

-Algoritmo
Procedure SCAN (PROGRAM, LOOKAHEAD, CHAR, TOKEN, POS)
Data la stringa sorgente PROGRAM, questo algoritmo restituisce il numero di rappresentazione interna del TOKEN successivo nella stringa.
Se TOKEN rappresenta un identificatore, stringa o costante, la procedura restituisce anche la sua posizione numerica nella tabella POS.
CHAR rappresenta il carattere corrente su cui si sta facendo l'analisi lessicale.
LOOKAHEAD è una variabile logica che ci dice se il simbolo in CHAR è stato usato nella chiamata precedente a SCAN.
Un valore "false" denota che non è stato ancora controllato.

-L'algoritmo utilizza le seguenti funzioni:
+ GET_CHAR(PROGRAM) che restituisce il prossimo carattere del programma sorgente;
+ INSERT(STRING,type) che inserisce un dato token STRING (se necessario) ed il suo tipo (cioè costante, stringa o variabile) nella tabella dei simboli;
+ KEYWORD(STRING) che restituisce il numero di rappresentazione interna del suo argomento se è una keyword, 0 altrimenti.
+ STRING contiene il token corrente (nome di variabile, costante, stringa).
Le variabili DIVISION, LEQ, NEQ, LTN, GEQ, GTN, EQ, LEFT, RIGHT, ADDITION, SUBTRACTION, MULIPLICATION, ASSIGNMENT, SEMICOLON,LITERAL, IDENTIFIER e CONSTANT
contengono i numeri interni di rappresentazione dei token /, <=, <>,<,>=, >, =, (,), +, -,*, :=, ;, stringhe, identificatori e costanti rispettivamente.

-Lex = un generatore di scanner utilizza espressioni regolari per specificare lo scanner, genera una tabella delle transizioni a stati finiti e interpreta tale tabella.
Il programma, quindi, simula l'automa a stati finiti ed esegue le funzioni associate con le azioni.
Lex può essere usato in congiunzione con un parser (YACC) per eseguire sia analisi lessicale che sintattica.

CAPITOLO 10

- Tabelle dei simboli= Servono a:
 +Controllo della correttezza semantica (cs);
 + aiuto nella generazione del codice.
Contiene gli attributi ovvero: il tipo, il nome, la dimensione, l'indirizzo i quali sono determinati analizzando il codice sorgente, è inserita in memoria centrale e viene modificata dinamicamente.

-Una tavola o tabella: è un tipo di dato astratto per rappresentare insiemi di coppie <chiave, attributi> , ciascuna coppia referenza un un'unica entità logica.

-Operazioni sulle tavole=
 +inserimento di un elemento <Chiave, Attributi>, inserisci: tavola x chiave x attributi → tavola
 + cancellazione di un elemento (nota la chiave),cancella: tavola x chiave → tavola
 + verifica di appartenenza di un elemento, esiste: tavola x chiave → boolean
 + ricerca di un elemento nella tavola, ricerca: tavola x chiave → attributi
(più frequente inserimento e ricerca , obiettivo ottimizzarli).

-Creazione della tabella dei simboli= in un compilatore a molti passi, la TS è creata durante l'analisi lessicale (scanner).

-Attributi della TS:
1. Nome della variabile, un problema è la dimensione della stringa del nome, inserito dallo scanner
2. Indirizzo nel codice oggetto, la locazione relativa delle variabili a run-time, inserito nella TS quando la variabile è dichiarata o incontrata per la prima volta, per i linguaggi a blocchi l'indirizzo è rappresentato da una coppia <BL , ON> a run-time ON rappresenta l'offset rispetto al blocco.
3. Tipo: implicito, esplicito o non esiste, fondamentale per il controllo semantico e per sapere quanta memoria è allocata alla variabile, inserito con una codifica.
4. Numero dei parametri di una procedura (dimensione): importante per il controllo semantico, in una procedura indichiamo il numero dei parametri per procedere al controllo durante le chiamate.
5. Linea sorgente in cui la variabile è dichiarata;
6. Linee sorgenti in cui la variabile è referenziata;
7. Pointer: Puntatori per esempio per listarli in ordine alfabetico.

-Per linguaggi a blocchi sono necessarie due operazioni addizionali che chiamiamo set e reset, set si invoca quando si entra in un blocco reset quando si esce.

-TS non ordinate= vengono aggiunte degli entries alla tabella nell'ordine in cui le variabili sono dichiarate.
In questo modo per un inserimento non e' richiesto nessun confronto, mentre una ricerca richiede, nel caso peggiore, il confronto con gli N elementi all'interno della tabella.
Poiché' ciò e' inefficiente, questa organizzazione dovrebbe essere adottata solo se la dimensione della TS e' piccola.

-TS ordinate= La posizione dell'elemento nella TS e' determinato dal nome della variabile (ordinamento lessicale).
In questo caso un inserimento e' sempre accompagnato ad una procedura di ricerca.
L'inserimento può inoltre richiedere uno spostamento di altri elementi già inseriti nella tabella (e' la maggiore fonte di inefficienza).
Ottimizzazione della ricerca: si ottiene se gli elementi sono memorizzati in modo ordinato nella tavola.
Deve esistere un ordinamento sul campo Chiave

-Miglioramento ulteriore= ricerca binaria.

-Tavole: rappresentazione collegata e ad albero II tempo per inserire un elemento in una TS ordinata si può ridurre utilizzando una struttura collegata od ad albero.

-La rappresentazione collegata: memorizzare gli elementi associando a ciascuno una particolare informazione (riferimento) che permetta di individuare la locazione in cui e' inserito l'elemento successivo.
La sequenzialità degli elementi della lista non e' rappresentata mediante l'adiacenza delle locazioni di memoria in cui sono memorizzati.
Non c'è un limite massimo alla dimensione della tavola.
Lo spazio di memoria occupato dipende solo dal numero di elementi della tavola.
Se la lista é ordinata sulla chiave, si può applicare la ricerca ordinata.

-Tabelle dei simboli strutturate ad albero= Ogni record rappresentato da un nodo ha tre campi: uno per l'informazione associata al nodo, uno per il puntatore al sottoalbero sinistro ed uno per il puntatore al sottoalbero destro.
non è necessario che l'albero sia completo; si ha un'occupazione di memoria efficiente; operare modifiche è agevole

- Algoritmi di visita per alberi binari=
+ Preordine o ordine anticipato: Analizza radice, albero sinistro, albero destro.
+ Postordine o ordine ritardato: Analizza albero sinistro, albero destro, radice.
+ Simmetrica: Analizza albero sinistro, radice, albero destro.

- Alberi binari di ricerca: Sono alberi binari (ordinati) utilizzati per memorizzare grosse quantità di dati su cui si esegue spesso un'operazione di ricerca di un dato.
In un albero binario di ricerca, ogni nodo N ha la seguente proprietà:
• tutti i nodi del sottoalbero sinistro di N hanno un valore minore o uguale a quello di N
• tutti i nodi del sottoalbero destro di N hanno un valore maggiore di quello di N.
N.B: Inserimento e cancellazioni devono essere fatte in modo da mantenere l'albero bilanciato.

Alberi binari (di ricerca): inserzione con bilanciamento=
-Nodo perfettamente bilanciato: l'altezza dei suoi due sottoalberi è la stessa
-Nodo bilanciato a sinistra: height(left(n)) = height(right(n))+l
-Nodo bilanciato a destra: height(left(n)) = height(right(n))-l
-Albero bilanciato (in altezza): quando l'altezza del sottoalbero sinistro e destro di ogni nodo differisce al
più di 1.

-Tavole: rappresentazione a funzione di accesso=
Funzione di accesso biunivoca: che data la chiave k restituisce il numero corrispondente a k(accesso, inserimento e cancellazione realizzati in modo efficiente);
L'utilizzo di una funzione di accesso biunivoca è possibile solo se il numero di elementi della tavola è circa uguale al numero di chiavi possibili, in alternativa si possono verificare collisioni (più chiavi per lo stesso indirizzo).

IN CASO DI COLLISIONE:
1. scegliere una funzione di accesso che riduca il più possibile le collisioni;
2. in caso di collisione, determinare un metodo di scansione della tavola (per ricerca o inserimento).

-Tabelle Hash: Funzioni hash (tritare, mescolare), perchè l'indirizzo calcolato dipende da tutta la chiave, eventualmente spezzata in parti "rimescolate".
La funzione hash si dice perfetta se e solo se non produce collisioni, ma non è facile. Evitarli.

-Gestione delle collisioni= Metodi di scansione, nel caso si verifichino collisioni:
+ Scansione lineare: Si passa a considerare per k2 l'indirizzo i+h (h prefissato), poi i+2*h, i+3*h;
+ Scansione con aree di trabocco: Area di trabocco con rappresentazione collegata (lista), ogni volta che si verifica una collisione l'elemento viene inserito in una classe di equivalenza di i oppure tutti in un'unica area di trabocco.

-Vantaggi e svantaggi dell'approccio hash:
+ basso costo medio di ricerca, elevate prestazioni, organizzazione dinamica;
+ semplicità di realizzazione;
+ non si prestano a una visita dei dati in ordine diverso da quello fisico;
+ non sono adatte a reperire un sottoinsieme dei dati con chiave primaria che soddisfi una relazione specificata;
+ costo medio delle principali operazioni basso, ma costo di caso peggiore nettamente più alto;

- Struttura a blocchi, regole di visibilità degli identificatori e tempo di vita= II significato della dichiarazione è quello di associare – durante l'attivazione di un'unità di programma (main o sottoprogramma) - varie informazioni all'identificatore.
II tempo di vita di una di queste associazioni è la durata dell'attivazione dell'unità di programma in cui compare la dichiarazione dell'identificatore.
L'effetto di una dichiarazione perdura per tutto il tempo di attivazione dell'unità di programma in cui tale dichiarazione si trova.

-Regole di visibilità degli identificatori= In pascal il campo di azione per gli identificatori segue le seguenti regole:
(1) il campo di azione della dichiarazione di un identificatore è il blocco (unità di programma) in cui essa compare e tutti i blocchi in esso contenuti, a meno della regola (2);
(2) quando un identificatore dichiarato in un blocco P è ridichiarato in un blocco Q, racchiuso da P, allora il blocco Q, e tutti i blocchi innestati in Q, sono esclusi dal campo di azione della dichiarazione dell'identificatore in P.
II campo di azione è determinato staticamente, dalla struttura del testo del programma (regole di visibilità lessicali).

-TS a Stack= E' l'organizzazione piu' semplice per un linguaggio a blocchi.
I record che contengono gli attributi delle variabili di un blocco B1 vengono messi nello stack quando si incontrano le corrispondenti dichiarazioni (push) ed eliminati al termine del blocco B1 (pop).
L'operazione di ricerca per un simbolo parte dal top dello stack, e quindi garantisce che i simboli piu' innestati siano trovati per primi.
L'operazione di set, salva il contenuto di top nello stack degli indici, mentre l'operazione di reset lo ripristina, cancellando implicitamente tutti i valori non piu' referenziati
L'organizzazione si può rendere piu' efficiente introducendo nello stack rappresentazioni piu' sofisticate quali quelle ad albero o con funzione di accesso hash.

CAPITOLO 11

-Linguaggio macchina= binario e fortemente legato all'architettura;
-Linguaggi Assembler, istruzioni del linguaggio macchina e riferimenti alle celle di memoria espressi mediante simboli.
-Linguaggi di alto livello, tentativo di astrazione dell'architettura sottostante.
Una possibile classificazione dei linguaggi di programmazione:
+ IMPERATIVI;
+ FUNZIONALI;
+ LOGICI;
+ AD OGGETTI;
A loro volta possono essere SEQUENZIALI o CONCORRENTI.

-**Linguaggi Imperativi**= Le caratteristiche essenziali sono dettate dalla architettura di Von Neumann.
Tutti i linguaggi tradizionali sono livelli di astrazione costruiti sopra tale architettura.
I linguaggi imperativi adottano uno stile prescrittivo e il programma imperativo prescrive le operazioni che il processore deve compiere.
Esecuzione delle istruzioni nell'ordine in cui appaiono nel programma, realizzati sia con interpretazione che compilazione.
Sono la classe di linguaggi più matura; più per manipolazione numerica che simbolica, evoluti verso: blocchi, moduli, tipi di dato astratto e parallelismo.
Il programma è formato da algoritmo + dati, istruzioni di lettura e scrittura, assegnamento e controllo.

- Programmazione modulare= Tecnica di suddividere un progetto software in parti il più possibile indipendenti, le cui modalità di interazione siano ben definite(interfacce standard).
Supporto a progettazione sia **top-down** che **bottom-up**.

Programmazione modulare: requisiti
1. Moduli corrispondenti ad unità sintattiche nel linguaggio usato;
2. Ciascun modulo deve "comunicare" con il minor numero di moduli;
3. Se due moduli si interfacciano tra loro, le loro interfacce devono scambiarsi il minor numero di informazioni;
4. Se due moduli comunicano tra loro, questo deve essere evidente dal loro codice;
5. Le informazioni di un modulo sono private, a meno che il modulo non le dichiari esplicitamente pubbliche;
Un modulo in genere conterrà dati ed operazioni, definisce e confina un preciso ambiente di visibilità.

-Le entità esportate sono definite in una parte dichiarativa dell'unità detta interfaccia.
-Le entità importate sono determinate mediante la parola chiave **uses**.
Il modulo (unit) è una unità sintattica con regole di visibilità dei nomi.

-Librerie= Il modulo rende visibili procedure e funzioni che non fanno uso di variabili non locali. Il modulo è una collezione di operazioni.

-Astrazioni di dato= Il modulo ha dati locali e rende visibili all'esterno le operazioni invocabili su questi dati locali, ma non gli identificatori dei dati.
La parte di inizializzazione del modulo può assegnare un valore iniziale ai dati locali nascosti.

-Tipo di dato astratto= Il modulo esporta un identificatore di tipo T e le operazioni eseguibili su dati dichiarati di questo tipo. I "clienti" del modulo dichiarano e controllano quindi il tempo di vita delle variabili di tipo T.

-**Linguaggi funzionali**= chiariamo alcuni concetti primitivi:
- Una funzione e' una regola di corrispondenza che associa ad ogni elemento del suo dominio un unico elemento nel codominio.
- Una definizione di funzione specifica il dominio, il codominio e la regola di associazione.
Esecuzione del programma = valutazione della funzione;
Ovvero Programma = funzione;
La sola OPERAZIONE del modello funzionale e' l'APPLICAZIONE di FUNZIONI ad OPERANDI;
- Il ruolo della macchina FUNZIONALE (interprete) e' VALUTARE il programma e produrre un valore;
- Il valore di una funzione e' determinato solo dai suoi argomenti (no effetti collaterali, puro);
- L'essenza della programmazione funzionale e' COMBINARE funzioni (utilizzo della ricorsione);
- Le VARIABILI sono variabili MATEMATICHE che denotano un valore fisso nel tempo e non valori mutabili (nessun assegnamento)

-Linguaggi LISP= Il programma e' una struttura dati;
Non c'e' dichiarazione di tipo; Non c'è assegnamento;

-Quando usare i linguaggi funzionali:
+ Manipolazione simbolica e non numerica;
+ Applicazioni di Intelligenza Artificiale;
+ Programmazione esplorativa;
+ Modifica dinamica dei programmi;
+ Problemi naturalmente ricorsivi;
+ Non per problemi di controllo di processo o real-time.

-Linguaggi Logici: Programmazione dichiarativa= Programma = conoscenza + controllo;
La conoscenza sul problema è espressa indipendentemente dal suo utilizzo, Alta modularità e flessibilità;
I moderni linguaggi di programmazione per Intelligenza Artificiale tendono a riprodurre tale schema;
Problematiche di RAPPRESENTAZIONE della conoscenza, è sia DICHIARATIVO che PROCEDURALE:

-DICHIARATIVO= Il risultato dell'ordinamento di una 1 lista vuota e' la lista vuota" ;
"Il risultato dell'ordinamento di una lista L è L' se la lista L' è ordinata ed è la permutazione di L";
-PROCEDURALE= "Controlla prima se la lista e' vuota; se si dai come risultato la lista vuota";
"Altrimenti calcola una permutazione L' di L e controlla se e' ordinata; se si, termina dando come risultato L', altrimenti calcola un'altra permutazione di L etc…"

-Linguaggio Porlog= E' il piu' noto linguaggio di programmazione dichiarativo:
Algoritmo = logica + controllo
Si basa sulla logica dei predicati del I ordine -
Le strutture dati su cui lavora sono alberi e anche i programmi sono strutture dati manipolabili;
Utilizzo della ricorsione, assenza di assegnamento.

-Un programma PROLOG e' un insieme di clausole di Horn (formule logiche), che rappresentano:
- fatti, riguardanti gli oggetti del dominio e le loro relazioni;
- regole sugli oggetti e sulle relazioni;
- goal o interrogazione sulla base di conoscenza (programma) definita.

-Quando utilizzare un linguaggio logico:
+ Manipolazione simbolica e non numerica;
+ Applicazioni di A.I.;
+ Modifica dinamica dei programmi;
+ Problemi naturalmente ricorsivi. (condividono caratteristiche coi linguaggi funzionali)
+ Metodologicamente ci si concentra piu' sulla specifica del problema che non sulla strategia di soluzione (efficienza).
-base di molti concetti e discipline.
Ma sono linguaggi relativamente giovani: mancanza di ambienti di programmazione evoluti, efficienza paragonabile a quella del LISP.

-**Programmazione ad oggetti**= Linguaggio ed ambiente di programmazione si confondono, icone ed interfaccia a finestre, il più noto c++.

- Linguaggi ad oggetti si sono evoluti a partire dal concetto di astrazione di dato, tipo di dato astratto e classe introdotti in linguaggi imperativi;
- Punto di unione fra programmazione tradizionale e rappresentazione della conoscenza;
- Adatti per la programmazione "in largo";
- RIUSABILITA';
- Incrementalità: ciascuna classe estende le definizioni delle sue superclassi;
- Interfaccia ed ambiente gradevole.