

# Appunti linguaggi formali

## Compilatori e interpreti

Un **programma** scritto in un linguaggio di programmazione per poter essere **eseguito dev'essere tradotto in linguaggio macchina** (codice oggetto) e questa traduzione viene effettuata dal **compilatore**. In **alternativa** c'è l'**interprete** che esegue la **traduzione** riga per riga **in simultanea con l'esecuzione**

**Albero sintattico astratto**: il **compilatore con i codici costruisce un albero sintattico** che ricostruisce l'ordine dei passaggi ed il modo in cui costruisce l'albero è uno standard. Prima la definizione della funzione, poi l'header e il body, ecc.

La **compilazione** si divide in:

- **Fase analitica**: **sorgente esaminato per verificare se soddisfa regole sintattiche e semantiche**
  - **Analisi lessicale**: lo **scanner raggruppa i caratteri in lessemi** e ad ognuno associa un token e un attributo che indicano la classe e il riferimento alla tabella dei simboli
  - **Analisi sintattica**: il **parser crea l'albero sintattico astratto** coi token dello scanner
  - **Analisi semantica statica**: utilizzando l'albero del parser e la tabella dei simboli **si esegue il type checking in cui si verifica che ogni operatore abbia gli operandi di tipo corretto** e può eseguire anche la coercizione che sarebbe la conversione implicita
- **Fase sintetica**: viene **generato il codice oggetto**
  - **Generazione codice intermedio**: dall'albero sintattico otteniamo un **codice fatto d'istruzioni elementari ed indipendente dall'architettura** perché non vengono specificati registri
    - **Ottimizzatore**: riduce il tempo d'esecuzione del codice intermedio
  - **Generazione del codice oggetto**: **dipende dall'architettura** perché è necessario fissare le locazioni di memoria per i dati e va generato anche il codice per accedervi. È riutilizzabile per tutti i linguaggi disponibili per l'architettura su cui lavoriamo

**Organizzazione del compilatore**: il nucleo è l'**analizzatore sintattico che legge ogni token e attiva la generazione del codice**

## Parole e linguaggi

Un **linguaggio** è un **insieme di parole** che può essere **finito o infinito**

**Teorema di Cantor**: è **impossibile associare** a ciascun **linguaggio** una **parola** che lo **caratterizzi univocamente**

**Alfabeto**: un **insieme finito non vuoto**  $\Sigma$  di simboli ed i suoi **elementi** sono detti **lettere**. L' **Insieme** delle **parole** sull'alfabeto è  $\Sigma^*$

**Parola**: ogni **sequenza finita di lettere** prese su un certo  $\Sigma$  (insieme) sarà detta **parola sull'alfabeto**  $\Sigma$ . L'intero  $k$  si dice lunghezza della parola  $u$ , la parola di lunghezza 0 si denota con  $\epsilon$  (epsilon)

**Concatenazione**: di due parole  $u$  e  $v$  è la parola  $uv$ , è un'**operazione binaria su  $\Sigma^*$**  con le seguenti proprietà:

- **Associativa**
- **Elemento neutro**
- **Cancellatività**

**Potenza n-esima**: si ottiene **concatenando n coppie** della **parola**  $w$

**Fattore**: una **parola**  $v$  è fattore di una  $w$  **se**  $w = xvy$  **appare come un blocco unico e contiguo**

**Linguaggi formali**: ogni **sottoinsieme**  $\Sigma^*$  di **parole** sull'alfabeto  $\Sigma$  si dice **linguaggio formale** sull'alfabeto  $\Sigma$

**Grammatiche:** una grammatica a struttura di frase è una **quadrupla**  $G = (V, \Sigma, P, S)$

- $V$  **alfabeto finito** (vocabolario totale)
- $\Sigma$  (sottoinsieme di  $V$ ) **alfabeto** dei simboli **terminali**
- $P$  insieme finito di espressioni della forma  $\alpha \rightarrow \beta$  (**insieme delle produzioni**)
- $S \in N = V \setminus \Sigma$  **simbolo iniziale** o assioma (le lettere in  $N$  si dicono variabili)
  - **Variabili** lettere **maiuscole**
  - **Terminali** lettere **minuscole**

**Linguaggio generato:** siano  $\alpha, \beta \in V^*$

- $\beta$  **conseguenza diretta**  $\alpha$  se  $\gamma_1, \gamma_2 \in V^*$  e una produzione  $\gamma \rightarrow \gamma'$  tale che  $\alpha = \gamma_1 \gamma \gamma_2, \beta = \gamma_1 \gamma' \gamma_2$  cioè **da una stringa  $\alpha$  ottieni  $\beta$  applicando una sola produzione**
- $\beta$  **deriva**  $\alpha$  se esistono  $n$  produzioni tali che  $\alpha = \alpha_0 \Rightarrow \dots \Rightarrow \alpha_n = \beta$  cioè come prima solo **applicando più produzioni** una dopo l'altra

**Forme sentenziali:** sono le **frasi** che otteniamo **partendo dal simbolo iniziale applicando la grammatica**  
Il **linguaggio generato** è l'**insieme di tutte le frasi senza variabili ottenibili partendo da  $S$**

### La gerarchia di Chomsky

Distinzione tra freccette:  $\rightarrow$  indica la produzione,  $\Rightarrow$  indica una conseguenza diretta,  $\Rightarrow^*$  indica una conseguenza non diretta

**Grammatiche equivalenti:** equivalenti se **generano lo stesso linguaggio**

**Problema di ricognizione & parsing:** quello di **ricognizione** riguarda il riconoscere se una **parola appartiene ad un linguaggio** di una specifica grammatica, mentre quello di **parsing** **trovare una o più derivazioni di una parola data la grammatica**. Per risolvere questi problemi bisogna restringersi a classi particolari

**Linguaggi di tipo 0:** le **grammatiche a struttura di frase** si chiamano anche grammatiche di tipo 0 e i linguaggi generati sono detti linguaggi di tipo 0 o ricorsivamente enumerabili (**sono il tipo di default essendo senza alcuna restrizione**)

**Grammatiche sensibili al contesto** (sottoclasse propria del tipo 0): sensibile al contesto se le produzioni hanno la forma  $\alpha_1 X \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$  con  $X \in N, \alpha_1, \alpha_2, \beta \in V^*, \beta \neq \epsilon$ . I linguaggi generati si dicono linguaggi di **tipo 1** o sensibili al contesto infatti **una variabile può essere sostituita solo se si trova in un determinato contesto (simboli che ha intorno) e la lunghezza della parola generata non diminuisce (monotonicità)**

**Grammatiche monotone:** i **lati  $dx$  delle parole hanno lunghezza  $\geq$  rispetto ai lati  $sx$** . Non tutte le grammatiche monotone sono sensibili al contesto

**Grammatiche non contestuali** (sottoclasse propria del tipo 1): non contestuale o **tipo 2** se le produzioni hanno la forma  $X \rightarrow \beta$  con  $X \in N, \beta \in V^*$  i linguaggi generati si dicono di tipo 2 o non contestuali quindi sostituisco qualcosa **indipendentemente** da ciò che ha **intorno (sx sempre una singola variabile a dx qualsiasi combinazione di variabili e terminali)**

**Grammatiche regolari** (sottoclasse propria del tipo 2): regolare o **tipo 3** se le produzioni hanno la forma  $X \rightarrow aY$  oppure  $X \rightarrow a$  con  $X, Y \in N, a \in \Sigma$  i linguaggi generati si dicono di tipo 3 o regolari (**dx almeno un terminale, una variabile produce un terminale seguito da un'altra variabile o una variabile produce un singolo terminale**)

## Automati a stati finiti

Dispositivo con un numero finito di configurazioni interne. Ad ogni passo legge una cella, passa in nuovo stato e esamina il contenuto della cella successiva, terminata la lettura in base allo stato finale accetta o rifiuta la parola letta

**Automa deterministico:** automa a stati finiti deterministico è una quintupla  $A=(Q,\Sigma,\delta,q_0,F)$

- $Q$  insieme degli stati
- $\Sigma$  alfabeto di input
- $\delta: Q \times \Sigma \rightarrow Q$  funzione di transizione
- $q_0 \in Q$  stato iniziale
- $F \subseteq Q$  stati finali

**Funzione di transizione estesa:** si estende a  $\hat{\delta}: Q \times \Sigma^* \rightarrow Q$  ponendo  $\hat{\delta}(q, \varepsilon) = q$  per ogni  $q \in Q$ ,  $\hat{\delta}(q, va) = \delta(\hat{\delta}(q, v), a)$  per ogni  $q \in Q, v \in \Sigma^*, a \in \Sigma$ . Ti dice lo stato in cui si trova un automa dopo aver letto un'intera parola invece che un solo carattere. Applica la funzione normale un carattere alla volta fornendoti poi direttamente lo stato alla fine

**Linguaggio accettato deterministico:** una parola  $w \in \Sigma^*$  è accettata da  $A$  se  $\hat{\delta}(q_0, w) \in F$  (partendo dallo stato iniziale e seguendo l'unico percorso guidato dai caratteri della parola si arriva in uno stato finale). L'insieme delle parole accettate da  $A$  si dice linguaggio riconosciuto e si denota  $L(A)$   $L(A)=\{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$

**Grafo di un automa deterministico:** ogni automa ha associato un grafo diretto in cui i vertici sono gli stati, ci sono una serie di frecce che vanno da stato a stato ed ognuno con un'etichetta con una lettera dell'alfabeto (rappresentano le transizioni), lo stato iniziale è identificato da una freccia entrante mentre quelli finali dai doppi bordi. Le parole accettate dall'automa sono le etichette dei cammini dallo stato iniziale ad uno finale

## Automati a stati finiti non deterministici

per gli automi deterministici esiste esattamente un arco uscente da uno stato se si eliminasse questa regola otterremo un automa non deterministico. Funziona che ad ogni passo viene eseguita una fra le transizioni possibili, poi l'input è accettato se almeno una delle computazioni possibili termina altrimenti è rifiutato.

**Automa non deterministico:** è una quintupla  $A=(Q,\Sigma,\delta,q_0,F)$  la cosa che cambia è la funzione di transizione infatti  $\delta: Q \times \Sigma \rightarrow p(Q)$  (restituisce un insieme di stati)

**Linguaggio accettato non deterministico:** una parola  $w$  è accettata se esistono stati  $q_i \in \delta(q_{i-1}, a_i), 1 \leq i \leq n, q_n \in F$  (se esiste almeno un percorso che partendo dallo stato iniziale arriva in uno finale e le etichette formano la parola)

La differenza con quello deterministico è che potendoci essere più scelte potrebbero esserci più percorsi ecco perché scriviamo almeno un percorso

**Grafo di un automa non deterministico:** ogni automa ha associato un grafo come per gli automi deterministici cambiano le frecce, in un grafo di un automa deterministico da uno stato ci sarà sempre un'unica transizione possibile per ogni simbolo mentre se è non deterministico ci potrebbero essere nessuna o più frecce. La parola è accettata solo se c'è un cammino dallo stato iniziale ad uno finale con quell'etichetta

**Determinizzazione:** sia  $A$  un automa a stati finiti non deterministico esiste effettivamente un automa a stati finiti deterministico  $A'$  tale che  $L(A)=L(A')$  (che accetta lo stesso linguaggio)

- **Dimostrazione:** gli stati dell'automa deterministico registreranno l'insieme degli stati raggiungibili dalle computazioni di quello non deterministico con il medesimo input accettando se fra questi ce

ne sta uno finale. Dato l'automa non deterministico  $A=(Q,\Sigma,\delta,q_0,F)$  costruisco quello deterministico in cui l'insieme degli stati sono i sottoinsiemi di  $Q$ , lo stato iniziale prendo l'insieme con solo  $q_0$  e gli stati finali sono tutti i sottoinsiemi di  $Q$  che contengono almeno un elemento di  $F$ , infine la funzione di transizione è definita così partendo da un insieme di stati e leggendo un simbolo calcolo l'unione di tutte le possibili destinazioni. Definito  $A'$  risulta che per ogni  $w \in \Sigma^*$  risulta  $\hat{\delta}(s_0, w) = \{q \in Q \mid \text{nel grafo } A \text{ c'è un cammino da } q_0 \text{ a } q \text{ con etichetta } w\}$  ( $s_0$  è lo stato di destinazione del nuovo automa deterministico,  $dx$  è l'insieme di tutti gli stati dell'automa non deterministico che sono raggiungibili partendo da  $q_0$  seguendo la parola  $w$ ). In breve si costruisce un automa deterministico dove ogni stato è un insieme di stati di quello non deterministico, così facendo il primo tiene traccia di tutti i percorsi che il secondo potrebbe seguire contemporaneamente e la parola è accettata da entrambi se lo stato-insieme in cui si trova contiene uno stato finale dell'automa non deterministico.

**Stati inaccessibili:** uno stato  $q \in Q$  di un automa deterministico si dice accessibile se  $q = \hat{\delta}(q_0, w)$  per qualche  $w \in \Sigma^*$  cioè se può essere raggiunto da  $q_0$  seguendo il simbolo  $w$ . Eliminando gli stati inaccessibili si ottiene un automa equivalente per esempio l'automa determinizzato se  $Q=n$  allora  $Q'=2^n$  ma non tutti saranno accessibili

**Algoritmo di determinizzazione:** inserisco nella lista degli stati l'insieme con solo  $q_0$ , per ogni stato  $r$  della lista ed ogni lettera  $a \in \Sigma$  faccio che: calcolo la funzione di transizione  $s$ , se non è nella lista degli stati allora l'appendo e se contiene uno stato finale l'aggiungo alla lista dei finali. Ad un certo punto non si aggiungeranno più stato quindi avrò terminato

### Epsilon-transizioni

**E-transizioni:** automa a stati finiti non deterministico con  $\epsilon$ -transizioni è una quintupla  $A=(Q,\Sigma,\delta,q_0,F)$  in cui la funzione di transizione diventa  $\delta:Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$  (è compresa la parola vuota) ed una parola  $w$  è accettata da  $A$  se esiste almeno un percorso nel grafo dell'automa che parte dallo stato iniziale, finisce in uno finale e le etichette sulle frecce ignorate le  $\epsilon$ -transizioni formano la parola che sto leggendo. Vanno ignorate perché permettono di passare da uno stato all'altro senza leggere nessun simbolo

**Grafo dell'automa  $\epsilon$ -transizioni:** l'unica cosa che cambia rispetto a prima è che le frecce ora possono essere etichettate con  $\epsilon$

**Determinizzazione:** sia  $A$  un automa a stati finiti non deterministico con  $\epsilon$ -transizioni esiste effettivamente un automa a stati finiti deterministico  $A'$  tale che  $L(A)=L(A')$

- **Dimostrazione determinizzazione:** sia  $A$  l'automa non deterministico con  $\epsilon$ -transizioni e  $A'$  l'automa deterministico, l'insieme degli stati  $Q'$  è costituito dai sottoinsiemi di  $Q$ , lo stato iniziale è composto dal  $\epsilon$ -chiusura di  $q_0$ , gli stati finali sono tutti i sottoinsiemi di  $Q$  con almeno un elemento di  $F$  e la funzione di transizione è l' $\epsilon$ -chiusura di tutti gli stati raggiungibili leggendo un determinato carattere (prima vanno calcolate le normali transizioni)
- **Algoritmo di determinizzazione:** l'algoritmo aggiunge il calcolo dell' $\epsilon$ -chiusura all'inizio e dopo ogni transizione. In sostanza all'inizio calcolo lo stato iniziale che sarebbe l' $\epsilon$ -chiusura di quello originale, poi fino a che ci sono stati che non ho controllato calcolo la transizione per ogni simbolo dell'alfabeto e faccio l'unione di tutti gli stati raggiungibili leggendo quel carattere e per finire faccio l' $\epsilon$ -chiusura di ciò

**$\epsilon$ -chiusura:** è l'insieme degli stati che da questo si possono raggiungere utilizzando solo  $\epsilon$ -transizioni perciò l' $\epsilon$ -chiusura di un insieme di stati è l'unione delle  $\epsilon$ -chiusure dei singoli stati

- **Calcolo della  $\epsilon$ -chiusura:** come input abbiamo un automa non deterministico con  $\epsilon$ -transizioni e uno stato  $q$ . La procedura è: nell'insieme degli stati  $R$  metto lo stato  $q$  e in una variabile  $p$  metto il primo elemento dell'insieme. Finché  $p$  non sarà nulla appendo all'insieme tutti gli elementi

raggiungibili da  $p$  con transizione  $\epsilon$  che non siano già stati presi, poi  $p$  diventa il successivo elemento dell'insieme. L'output sarà l' $\epsilon$ -chiusura di  $q$

## Espressioni regolari

**Operazioni regolari:** **unione, concatenazione e chiusura di Kleene** (tutte le parole che puoi formare concatenando un numero qualsiasi di parole prese dal linguaggio, inclusa quella vuota) sono dette **operazioni regolari**

**Espressioni regolari:** sia  $\Sigma$  l'alfabeto ottenuto aggiungendo a quello di partenza le lettere  $\emptyset, +, *, (, )$  si dicono espressioni regolari sull'alfabeto  $\Sigma$  le parole sull'alfabeto  $\Sigma$  che si ottengono applicando per un numero finito di volte le regole: ogni lettera dell'alfabeto è un'espressione regolare e anche  $\emptyset$  lo è, se  $E$  e  $F$  sono espressioni regolari allora  $(E+F)$  unione,  $(EF)$  concatenazione e  $E^*$  chiusura di Kleene, sono espressioni regolari

In breve è una **sequenza di simboli costruita seguendo queste regole**: sono **espressioni regolari** **qualsiasi singola lettera dell'alfabeto** e il linguaggio vuoto, se ho **due espressioni regolari** allora **lo** sono anche l'unione, la concatenazione e la chiusura di Kleene

**Linguaggi regolari:** ad ogni **espressione regolare** è **associato un linguaggio** e definito dalle seguenti regole: per ogni lettera dell'alfabeto l'espressione regolare denota il linguaggio tra graffe mentre l'espressione regolare  $\emptyset$  significa **linguaggio vuoto**, poi le operazioni che si fanno sulle espressioni regolari si traducono nelle operazioni corrispondenti sui linguaggi che esse descrivono

**Priorità operazioni:** chiusura di Kleene, concatenazione, somma

**Teorema di Kleene:** un **linguaggio** è **regolare se** e soltanto se è **riconosciuto da un automa a stati finiti**

- **Dall'espressione regolare all'automa:** utilizzeremo automi non deterministici con  $\epsilon$ -transizioni ed un **unico stato finale**. Iniziando dalle espressioni regolari base, **per una lettera facciamo un automa con uno stato iniziale ed uno finale collegati da una freccia con etichetta il simbolo dell'espressione**, per il linguaggio vuoto non ci sarà nessuna freccia mentre **per la parola vuota avremo l'etichetta  $\epsilon$** . Adesso **per combinare i due automi:** per l'**unione** creiamo un **nuovo stato iniziale e uno finale**, il primo si collega con  $\epsilon$ -transizioni ai due precedenti iniziali e il secondo si collega allo stesso modo, per la **concatenazione** si **collega lo stato finale di uno allo stato iniziale dell'altro** con un  $\epsilon$ -transizione mentre per la **chiusura di Kleene** **creiamo un nuovo stato iniziale ed uno finale aggiungendo 4  $\epsilon$ -transizioni**: dal nuovo iniziale al nuovo finale, dal nuovo iniziale al vecchio iniziale, dal vecchio finale al nuovo finale e dal vecchio finale al vecchio iniziale

In breve **smontiamo l'espressione regolare in un albero sintattico e poi applichiamo queste costruzioni partendo dalle foglie** che saranno le lettere fino ad arrivare alla radice arrivando ad un unico grande automa

- **Dall'automa all'espressione regolare:**
  - **Metodo induttivo:** definiamo  $L_{ijk}$  come l'insieme di tutte le parole che ti portano dallo stato  $q_i$  allo stato  $q_j$  usando come stati intermedi solo quelli con indice  $\leq k$  e l'obiettivo è trovare l'espressione regolare per  $L_{1jn}$  aumentando  $k$  da 0 fino a  $n$ . Il **caso base** sarà per  $k=0$  in cui **si guardano solo le etichette delle frecce dirette tra due stati**, il **passo induttivo** **utilizzeremo le espressioni calcolate al passo precedente  $k-1$** , considerando due tipi di percorsi da  $q_i$  a  $q_j$  quelli che non passano per  $q_k$  ma per stati con indice minore e quelli sì che fanno  $q_i \rightarrow q_k$  poi zero o più cammini con origine e termine in  $q_k$  e  $q_k \rightarrow q_j$

$$E_{ijk} = E_{ijk-1} + (E_{ikk-1})(E_{kkk-1})^* (E_{kjk-1}), 1 \leq i, j, k \leq n$$

**Costruisco l'espressione regolare calcolando i percorsi tra due stati che usano stati intermedi fino ad un indice  $k$ , considerando solo frecce dirette come caso base mentre come passo induttivo aggiungo i percorsi che passano attraverso lo stato di indice  $k$**

- **Regola pratica:** ci riduciamo al caso in cui abbiamo uno **stato iniziale privo di frecce entranti** ed uno **finale privo di frecce uscenti** e le etichette multiple diventano la loro somma. **Selezioniamo uno stato intermedio** e lo **si elimina** ed i suoi **percorsi** vengono **assorbiti dalle frecce degli altri stati**, ripetiamo questa operazione **fino a quando non rimane solo l'iniziale e il finale** perciò una sola freccia e la sua etichetta sarà l'espressione regolare

**Pattern matching:** se denoto un pattern che voglio trovare come un'espressione regolare mi basterà fare un automa non deterministico con  $\epsilon$ -transizioni che lo riconosca e poi posso passare ad un automa deterministico

**Operazioni booleane:** la **classe dei linguaggi regolari è un'algebra booleana** cioè chiusa rispetto alle operazioni di unione, complemento e intersezione

- **Dimostrazione:**
  - **Unione:** l'unione dei linguaggi è denotata dall'espressione  $E+F$
  - **Complemento:** per il teorema di Kleene un linguaggio regolare ha un automa a stati finito deterministico che lo riconosce, per far riconoscere il complemento è sufficiente **scambiare gli stati finali con quelli non** e riconoscerà tutte le parole che non erano nel linguaggio
  - **Intersezione:** poiché abbiamo **due linguaggi regolari anche i loro complementi** lo sono e l'unione di quest'ultimi è regolare e il **complemento** di quest'ultima (intersezione) è **di nuovo** regolare

## Automa minimo

**Equivalenze:** un'equivalenza è una **relazione riflessiva, simmetrica e transitiva**

**Congruenze:** sia  $\Sigma$  un alfabeto, una relazione di equivalenza  $\sim$  su  $\Sigma^*$  si dice una congruenza  $dx$  se per ogni  $u, v \in \Sigma^*$  tali che  $u \sim v$  e per ogni lettera  $a \in \Sigma$  si ha  $ua \sim va$  (quella  $sx$  è l'opposto). Una relazione che è sia congruenza  $dx$  che  $sx$  si dice congruenza

in breve **se due parole sono considerate equivalenti devono rimanere tali anche dopo aver attaccato la stessa lettera alla fine di entrambe (congruenza  $dx$ )**

**Equivalenza di Nerode:** sia  $L$  un linguaggio sull'alfabeto  $\Sigma$  l'equivalenza di Nerode è la relazione  $N_L$  definita in  $\Sigma^*$  da  $uN_Lv \forall y \in \Sigma^*, uy \in L \leftrightarrow vy \in L$  cioè  $uN_Lv$  se  $u$  e  $v$  hanno gli stessi completamenti a  $dx$  in  $L$

In breve **due parole sono equivalenti secondo Nerode se aggiungendo ad entrambe una parola  $y$  le nuove parole o sono entrambe nel linguaggio o entrambe fuori**

$N_L$  (**classe di equivalenza di Nerode**) è una **congruenza  $dx$**

- **Dimostrazione:** dobbiamo verificare la regola della congruenza  $dx$ , sappiamo che **da definizione** dell'equivalenza di **Nerode**  **$uy$  e  $vy$  sono intercambiabili sostituendo quindi  $y$  con  $ay$  ottengo  $uay$  e  $vay$  e rimangono gli stessi completamenti a  $dx$**  perciò varrà anche con  $ua$  e  $va$  e quindi è verificato

**$L$  è unione di classi di equivalenza di  $N_L$**

- **Dimostrazione:** assumendo che  **$u$  e  $v$  siano equivalenti** prendendo **come  $y$  la parola vuota** otterrei l'equivalenza di Nerode tra  $u$  e  $v$  quindi **ogni classe di equivalenza o sta dentro  $L$  o no**

**Teorema di Nerode:** sia  **$L$  un linguaggio sull'alfabeto  $\Sigma$  le seguenti proposizioni sono equivalenti:  $L$  è regolare** (può essere rappresentata con un'espressione regolare),  **$L$  è unione di classi di una congruenza destra,  $N_L$  ha indice finito** (ogni gruppo corrisponde ad uno stato nell'automa minimo)

- **Dimostrazione:** sia  **$L$  regolare quindi accettato da un automa deterministico** per il teorema di Kleene, **uso quest'automato per dividere in gruppi le parole in base allo stato in cui terminano**, il linguaggio sarà l'unione dei gruppi che terminano in uno stato finale. Ora dimostriamo che questo raggruppamento è una congruenza  $dx$ , **se due parole  $u$  e  $v$  sono nello stesso gruppo significa che portano allo stesso stato** di conseguenza **se gli attacchiamo una lettera l'automato essendo**

deterministico le porterà in un nuovo stato uguale ad entrambe. Essendo quindi una congruenza  $\sim$  ha indice finito devo dimostrare che vale anche per la classe di equivalenza, prendendo due parole che si trovano nello stesso gruppo, essendo una congruenza  $\sim$  attaccando qualcosa rimarranno nello stesso gruppo e sapendo che il linguaggio è formato da alcuni di questi gruppi le parole o sono entrambe dentro o entrambe fuori perciò i gruppi di Nerode non potrebbero essere di più. Sapendo quindi che la classe di equivalenza ha indice finito per dimostrare che il linguaggio è regolare basta fare un automa che lo riconosca secondo Kleene, gli stati saranno le classi di equivalenza, quello iniziale è la classe con la parola vuota, quelli finali sono le classi che sono dentro il linguaggio e la funzione di transizione è definita che leggendo un carattere porta allo stato che contiene tutte le nuove parole che si possono formare aggiungendo quel carattere ed essendo deterministico è lo stesso gruppo per tutti. Abbiamo costruito un automa perciò  $L$  è regolare

**Automa minimo:** sia  $L$  un linguaggio regolare l'automa di Nerode associato è un automa deterministico che accetta il linguaggio col minimo numero di stati possibile

- **Dimostrazione:** sia  $A$  l'automa deterministico che accetta il linguaggio col minimo numero di stati possibili  $n$ , sappiamo che il linguaggio è un'unione di classi di una congruenza  $\sim$  di dimensione  $n$  e sappiamo anche che la classe di equivalenza è  $\leq n$ , sapendo tutto questo abbiamo costruito un automa partendo dalla classe di equivalenza che quindi ha un numero di stati pari al suo indice
- **Costruzione dell'automa di Nerode:** l'idea è di raggruppare i vecchi stati in gruppi che si comportano in modo identico tramite classi di equivalenza, per trovare questi gruppi bisogna rispettare 3 regole: stati finali e non finali non possono stare in un unico gruppo, se due stati sono nello stesso gruppo devono essere equivalenti (cioè per ogni carattere devono portare a stati dello stesso gruppo) e dobbiamo fare la suddivisione con il minor numero di gruppi

**Algoritmo di minimizzazione:** il funzionamento è per approssimazioni successive cioè partendo da una suddivisione generica ad una specifica infatti partiamo da  $\Pi_0$  in cui creiamo due gruppi principali quello degli stati finali e quello dei non. Poi ad ogni passo creiamo gruppi sempre più precisi utilizzando il fatto che due stati possono stare nello stesso gruppo solo se per ogni simbolo le loro transizioni portano a stati che stanno nello stesso gruppo, se non la rispettano il gruppo viene spezzato. Questo processo si arresta quando la partizione trovata è identica alla precedente  $\Pi_n$

## Grammatiche regolari

Una grammatica a struttura di frase  $G = (V, \Sigma, P, S)$  è di tipo 3 se tutte le produzioni hanno le forme: sempre e solo una variabile a sx e a dx o un terminale ed una variabile o solo un terminale (la parola vuota è ammessa solo se la variabile che la produce non compaio nei lati dx delle produzioni)

**Teorema:** un linguaggio è regolare se e soltanto se è generato da una grammatica di tipo 3

Data la grammatica si può costruire un automa che accetti il linguaggio generato e viceversa

**Dalla grammatica all'automa:** considerando una grammatica di tipo 3 costruiamo un automa non deterministico che come stati ha le variabili della grammatica ed epsilon, per ogni produzione variabile  $\rightarrow$  terminale | variabile mettiamo una freccia con come etichetta il terminale mentre per l'altro tipo di produzioni facciamo lo stesso solo che lo stato di arrivo sarà un nuovo stato epsilon che farà anche da stato finale mentre l'iniziale sarà  $S$ .

- **Se l'automa accetta una parola allora la grammatica la genera:** parto da una parola che so che viene accettata perciò esiste un percorso dallo stato iniziale al finale, visto che ogni freccia corrisponde ad una regola della grammatica se le mettiamo tutte insieme ottengo una derivazione grammaticale che genera la parola
- **Se la grammatica genera una parola allora l'automa l'accetta:** parto da una parola generata cioè partendo da  $S$  ho aggiunto un terminale a sx ad ogni passo passando ad una nuova variabile, queste regole applicate hanno una controparte nell'automa e portano dallo stato iniziale al finale

**Dall'automa alla grammatica:** considerando un automa a stati finiti privo di  $\epsilon$ -transizioni **costruisco la grammatica** mettendo come **variabili gli stati dell'automa**, per **ogni freccia tra due stati X Y** aggiungo la relativa **produzione** con davanti ad Y l'etichetta della freccia, **se Y fosse anche finale aggiungo anche la produzione con solo l'etichetta a dx**, infine il simbolo iniziale sarà lo stato iniziale

**Grammatiche lineari:** si dice **lineare** se tutte le produzioni hanno la forma:  $X \rightarrow uYv$  o  $X \rightarrow u$  (**solo una variabile a sx e a dx o una sequenza di terminali con al centro una variabile oppure solo una stringa di terminali**)

- **Grammatiche lineari destre:** si dice lineare destra se tutte le produzioni hanno la forma:  $X \rightarrow uY$  o  $X \rightarrow u$  (cambia che stavolta a **dx dopo la variabile non ci dev'essere nessun terminale**)

**Teorema:** un **linguaggio** è **accettato da un automa a stati finiti** se e solo se è **generato da una grammatica lineare dx**

### **Lemma di iterazione per i linguaggi regolari**

**Lemma:** sia L un linguaggio regolare esiste un intero n per cui ogni parola di lunghezza  $\geq n$  si può fattorizzare  $w = xyz$  con  $y \neq \epsilon$  e  $xy^mz \in L$  per ogni  $m \geq 0$

In breve in **ogni parola del linguaggio abbastanza lunga si può trovare un fattore non vuoto che può essere iterato senza uscire dal linguaggio** (questo perché ci sarà una parte dopo che permette alla parola di essere accettata e ci sarà anche una parte prima che porta la parola all'inizio del ciclo)

- **Dimostrazione:** per il teorema di Kleene il linguaggio se regolare è accettato da un automa a stati finiti A con n stati. Abbiamo una **parola del linguaggio che è  $\geq n$** , visto che è nel linguaggio **ci sarà un percorso** che dallo stato **iniziale** arriva ad uno **finale** e **dato che è più lunga del numero degli stati almeno uno dovrà essere visitato due volte**, perciò spezziamo la parola in 3 parti quella iniziale sarà quella che va **dallo stato iniziale allo stato ripetuto**, quella **in mezzo sarà il ciclo** perciò non può essere vuota e quella finale sarà quella che va **dallo stato ripetuto al finale**. Perciò indipendentemente da quante volte si iteri la parola avrà sempre x e z che porta ad uno stato finale

### **Analisi lessicale**

Lessemi, token e pattern:

- **Token:** **simbolo astratto che rappresenta un'unità lessicale** come un identificatore ed è processato dal parser
  - **Attributi del token:** se più di un lessema è associato allo stesso pattern dobbiamo avere più informazioni al riguardo che sono registrate nella **tabella dei simboli** che contiene per **esempio puntatori al lessema specifico**
  - **Classi di token:**
    - Token per ogni **parola chiave**
    - Token per gli **operatori**
    - Unico token per gli **identificatori**
    - Uno o più token per le **costanti**
    - Token per ogni segno di **punteggiatura**
- **Lessema:** **sequenza di caratteri nel codice associata a un token** e l'analizzatore lessicale identifica i lessemi come istanze del token
- **Pattern:** **espressione regolare che descrive i lessemi**

**Analisi lessicale:** i token sono identificati dai pattern, hanno una priorità e **l'analizzatore lessicale identifica il lessema più lungo** con il pattern e **in ordine di priorità restituisce il primo token che da una corrispondenza** e continuerà a leggere

- **Automa per l'analisi lessicale:** **per ognuno dei pattern costruiamo un automa** a stati finiti, poi costruiremo un **automa unico** in cui avremo gli stati, le transizioni e gli stati finali di **ognuno dei singoli**, **nuovo stato iniziale** e delle  $\epsilon$ -transizioni di questo ai vecchi iniziali. **Per rendere poi questo**

automa deterministico sfruttiamo la costruzione dei sottoinsiemi in cui ogni suo stato è un insieme di stati del vecchio automa e quindi una parola corrisponde ad un pattern se lo stato in cui si trova contiene lo stato finale dell'automa precedente fatto ad hoc per quel pattern

- o **Algoritmo**: come input abbiamo l'automa deterministico e un puntatore al testo in esame, quindi inizieremo da lì partendo dallo stato iniziale finché esso non sarà nullo passeremo allo stato successivo e se è finale aggiorniamo la lunghezza di ciò che finora ho letto e passiamo alla lettera successiva. Quando lo stato sarà nullo ripartiremo dal punto dopo la lunghezza di ciò che ho letto

**Lex**: è un generatore di analizzatori lessicali e un programma è strutturato con: dichiarazioni, regole di traduzione, funzioni ausiliarie tutte sono separate con il simbolo %

### Linguaggi non contestuali

**Linguaggi non contestuali**: una grammatica si dice non contestuale o di **tipo 2** se tutte le produzioni sono della forma:  $X \rightarrow \alpha$  a sx una variabile e a dx una combinazione di variabili e terminali

Ogni grammatica lineare dx è non contestuale quindi ogni linguaggio regolare anche però non vale l'inverso

**Linguaggio di Dick**: data una grammatica con un'unica variabile S e produzioni:  $S \rightarrow a_i S b_i$ ,  $S \rightarrow SS$ ,  $S \rightarrow \varepsilon$  il linguaggio generato è detto linguaggio di Dick, viene chiamato il linguaggio delle parentesi ben bilanciate usando a e b come parentesi aperta e chiusa infatti la prima produzione dice che se ho una sequenza di parentesi posso metterne altre due come a e b e sarà ancora valida la sequenza, la seconda che se ho due sequenze valide mettendole di seguito sarà una sequenza valida ed infine l'ultima sarebbe la regola base

**Alberi di derivazione**: un albero è di derivazione della grammatica G se verifica le seguenti regole: l'etichetta della radice è S, le etichette dei nodi interni sono degli elementi di N, le etichette delle foglie sono elementi dell'alfabeto inclusa la parola vuota (le foglie con quest'etichetta sono figli unici) ed infine se un nodo ha figli ordinati da sx a dx allora ci sono le produzioni con l'etichetta del nodo a sx e le etichette delle foglie a dx. La parola associata all'albero la si ottiene leggendo da sx a dx le etichette delle foglie

- **Proposizione**: sia G una grammatica non contestuale si ha che una parola appartiene al linguaggio se e solo se esiste un albero di derivazione associato

**Grammatiche ambigue**: una grammatica è non ambigua se ad ogni parola del linguaggio generato corrisponde un solo albero di derivazione

**Linguaggi interamente ambigui**: un linguaggio non contestuale si dice interamente ambiguo se non è generato da nessuna grammatica non contestuale non ambigua

### Semplificazioni nelle grammatiche non contestuali

**Produzioni fastidiose**: andrebbero evitate le  $\varepsilon$ -produzioni, le produzioni 1-arie e le variabili che non compaiono nelle forme sentenziali perciò risultano inaccessibili e quelle da cui non si derivano parole prive di variabili

- **E-produzioni**: le produzioni con a dx la parola vuota
- **Produzioni 1-arie**: le produzioni con a dx un'unica variabile

**Motivazione**: in assenza di quelle produzioni, avremmo una garanzia di progresso infatti se passassimo da una stringa ad un'altra la stringa di arrivo potrebbe essere più lunga oppure migliorare nel senso che avrà meno variabili e più terminali. Se avessi quelle produzioni il progresso non sarebbe garantito infatti con la parola vuota la stringa si accorcia e basta e con la produzione 1-aria non ci sarebbe nessun cambiamento

**Eliminazione delle  $\varepsilon$ -produzioni**: ogni grammatica potrebbe avere una  $\varepsilon$ -produzione solo se prodotta dal simbolo iniziale che poi non dovrà comparire nei lati dx delle produzioni

- S non è annullabile (**il linguaggio non contiene la parola vuota**)
  - **Costruzione della grammatica:** una **variabile di una grammatica si dice annullabile se produce  $\epsilon$** , cancellandole verrà una grammatica equivalente
    - **Dimostrazione:** **ogni nuova produzione creata per la nuova grammatica è una semplificazione di una derivazione che era già possibile nella grammatica originale** ne consegue che ogni parola generata da questa grammatica è generata anche dall'originale
- S è annullabile (**il linguaggio contiene la parola vuota quindi dovrà rimanere**)
  - **Costruzione della grammatica:** **procedo eliminando dai lati dx tutte le variabili annullabili** poi se
    - **S non compare nei lati dx:** **aggiungo la  $\epsilon$ -produzione non darà problemi** visto che non potrà essere usata a dx
    - **S compare nei lati dx:** **aggiungo una nuova variabile che diverrà il mio simbolo iniziale e produrrà lei la  $\epsilon$ -produzione ed S.** Se mettesi la parola vuota ad S tornerei alla situazione di prima perciò aggiungo un nuovo simbolo che isoli la parola vuota e al contempo genera le altre parole

**Ricerca delle variabili annullabili:** **costruisco un insieme di variabili annullabili partendo dalle più ovvie fino alle più specifiche**, parto da quelle che hanno come produzione diretta la parola vuota per poi aggiungere tutte le variabili di cui verifico se esiste una sua produzione il cui lato dx è composto interamente da variabili che abbiamo già scoperto al passo precedente. Ci fermeremo quando l'insieme smetterà di crescere essendo finito il numero di variabili

**Eliminazione delle produzioni unarie:** a partire dalla grammatica che abbiamo ne costruiamo una nuova in cui eliminiamo tutte le  $\epsilon$ -produzioni e **per ogni coppia di variabili che possono derivarsi tra loro aggiungiamo i lati dx delle produzioni di quella derivabile a quelli delle produzioni della prima così facendo possiamo eliminare tutte le produzioni unarie** visto che ora non abbiamo più lunghe catene di derivazione ma se una poteva derivare un'altra allora potrà fare tutto quello che quella può fare. Alla fine del processo avrò una grammatica equivalente

- **Dimostrazione:** faremo vedere che **a ogni albero di derivazione della precedente grammatica ce n'è uno associato alla stessa parola nella nuova grammatica.**
  - **S ha più di un figlio (non inizia con una produzione unaria):** **essendo che la produzione di S non è unaria la regola verrà mantenuta anche nella nuova grammatica**, poi guardiamo ai **sotto alberi** che questa produzione forma e **diamo per scontato che siano già semplificati visto che sono più piccoli dell'albero** che parte da S e quindi più vicini ai terminali. Alla fine assembliamo tutto e il risultato sarà un albero di derivazione completo
  - **S ha un solo figlio (catena di produzioni unarie):** poiché nella grammatica originale **S può derivare un'altra variabile allora facciamo ereditare ad S tutte le produzioni non unarie dell'altra** così viene creata una produzione diretta. A questo punto assumo che i sotto alberi che partono da questa produzione siano già costruibili nella nuova grammatica perciò assemblo tutto insieme

**Ricerca delle derivazioni unarie:** **costruisco un insieme di coppie di variabili partendo dalle derivazioni più brevi fino alle più lunghe.** All'inizio conterrò solo le derivazioni dirette poi tramite una **regola ricorsiva che aggiunge all'insieme le nuove coppie che trovo combinando quelle scoperte al passo prima**, cioè se io già so che posso andare da X a Z e da Z a Y allora X e Y si possono derivare. Ad ogni passo aggiungeremo ogni volta una coppia e ci fermeremo quando non ne verranno più aggiunte

Pensando ad un grafo in cui le variabili sono i nodi e le produzioni sono le frecce l'algoritmo cerca tutti i percorsi tra tutti i nodi

**Variabili improduttive:** una variabile si dice **produttiva se** da essa si può **derivare una stringa composta solo da terminali** altrimenti è improduttiva

- **Ricerca delle variabili produttive:** parto da un insieme che conterrà solo le variabili che hanno già una produzione esclusivamente composta da terminali, poi aggiungerò variabili se hanno una produzione il cui lato dx è composto da qualsiasi combinazione di terminali/variabili che abbiamo già preso al passo precedente. Quando l'insieme smetterà di crescere avremo finito

**Variabili inaccessibili:** una variabile si dice accessibile se compare in qualche forma sentenziale altrimenti è inaccessibile

- **Ricerca delle variabili accessibili:** parto da un insieme che conterrà l'unica variabile che so essere accessibile cioè il simbolo iniziale poi ricorsivamente se una variabile è già stata considerata accessibile allora tutte le variabili che compaiono nella parte dx di qualsiasi sua produzione sono a loro volta accessibili. Ci arresteremo quando l'insieme smetterà di crescere

Va fatta sempre prima la rimozione delle variabili improduttive e poi quelle inaccessibili perché eliminando variabili improduttive se ne formeranno nuove inaccessibili

### Forma normale di Chomsky

**Forma normale di Chomsky:** una grammatica non contestuale si dice in forma normale di Chomsky se ha solo produzioni dei tipi:  $X \rightarrow YZ$  (una variabile che produce due variabili),  $X \rightarrow a$  (una variabile che produce un terminale) e  $S \rightarrow \epsilon$  (parola vuota) ma solo a condizione che S non compaia nei lati dx

**Alberi di derivazione Chomsky:** gli alberi di derivazione di una grammatica in forma normale di Chomsky hanno che le foglie sono figli unici e i nodi interni costituiscono un albero binario completo

**Riduzione in forma normale di Chomsky:** ogni linguaggio non contestuale è effettivamente generato da una grammatica non contestuale in forma normale di Chomsky

- **Procedura:**
  - Eliminare le  $\epsilon$ -produzioni e quelle unarie così da rimanere con quelle che producono un solo terminale e quelle che anno due o più simboli
  - Per ogni terminale a dx che non compare solo introduco una nuova variabile che produce quel terminale e vado a sostituire tutte le occorrenze con la variabile. Ripeto questo passo finché non avrò tutte le produzioni in forma normale di Chomsky

**Forma normale di Greibach:** una grammatica non contestuale si dice in forma normale di Greibach se ha solo produzioni dei tipi:  $X \rightarrow ay$  (una variabile che produce un terminale e una combinazione di variabili) e  $S \rightarrow \epsilon$  (parola vuota)

- **Riduzione in forma normale di Greibach:** in una grammatica in forma normale di Greibach ogni derivazione di una parola dell'linguaggio di lunghezza n richiede esattamente n passi

### Lemma di iterazione per i linguaggi non contestuali

**Lemma (di iterazione per i linguaggi regolari):** dato un linguaggio regolare esiste un intero n tale che ogni parola di lunghezza  $\geq n$  può essere fattorizzata come xyz con y non nullo e y può essere iterata zero o più volte

**Lemma (di iterazione per i linguaggi non contestuali):** dato un linguaggio non contestuale esiste un intero n tale che ogni parola di lunghezza  $> n$  si può fattorizzare come xuyvz con u e v non nulli e  $xu^k yv^k z \in L$  per ogni  $k \geq 0$

In breve qualsiasi parola dell'linguaggio più lunga di n può essere spezzata in cinque parti in cui x y z sono le parti fisse mentre u e v le parti che possono essere iterate zero o più volte. La differenza da quello regolare è che lì viene iterata una sola sottostringa

- **Dimostrazione:** prendiamo un linguaggio con una grammatica priva di  $\epsilon$ -produzioni e produzioni unarie, stabiliamo n come la massima lunghezza delle parole che hanno un albero di derivazione di

altezza  $\leq$  al numero di variabili. Poi prendiamo una parola che ha lunghezza superiore ad  $n$  quindi avrà un cammino di lunghezza maggiore al numero di variabili quindi ci sarà una variabile che si ripete perciò dividiamo la parola in 5 parti in cui  $x$  e  $z$  sono le parti esterne,  $u$  e  $v$  sono le parti da iterare e  $y$  la parte fissa \*\*\*

### Algoritmo di Cocke-Kasami-Younger

**Algoritmo CYK:** è una soluzione al problema di ricognizione (se una parola appartiene ad un linguaggio) e di parsing (se una parola è derivabile nel linguaggio) per le grammatiche non contestuali

- **Idea:** data una grammatica in forma normale di Chomsky e una parola, la stessa la scriviamo come stringa di lettere e cerchiamo di trovare quali variabili possono generare i singoli caratteri poi ripeto questo passo per sottostringhe sempre più lunghe utilizzando i risultati del passo precedente. Il passaggio finale sarà quando arrivo alla stringa che equivale all'intera parola verifico che nell'insieme di variabili che ho costruito è presente anche il simbolo iniziale
- Algoritmo: l'algoritmo occupa  $O(n^3)$

**Parsing:** una volta che sappiamo che la parola data all'algoritmo CYK è nel linguaggio ricostruiamo i passaggi per capire la derivazione. L'algoritmo controlla le sottostringhe, per quelle di lunghezza 1 significa che la derivazione è la produzione di un solo terminale quindi possiamo fermarci, per quelle più lunghe avremo una produzione fatta da due variabili perciò l'algoritmo cerca nella tabella il punto in cui le due variabili sono state spezzate per produrre quella sottostringa per poi richiamarsi sulle due metà così da verificare se a loro volta sono state generate da due variabili o hanno una produzione diretta con un terminale

### Parsing top-down

**Derivazioni  $sx$ :** data una grammatica non contestuale e  $\alpha, \beta \in V^*$  (combinazione di variabili) scriveremo che alfa deriva beta se questa si può ottenere da alfa sostituendo la variabile più a  $sx$  con il lato  $dx$  di una sua produzione (in breve se abbiamo più variabili in una derivazione a  $sx$  dovremmo espandere quella che incontro a  $sx$  così garantiamo a ogni albero di derivazione una sola derivazione  $sx$ )

C'è una corrispondenza biunivoca tra le derivazioni  $sx$  di una parola e gli alberi di derivazione ad essa associati

Data una grammatica non contestuale ogni parola del linguaggio di quest'ultima si ottiene dal simbolo iniziale con una derivazione  $sx$

**Endmarker:** data una grammatica non contestuale per motivi di efficienza aggiungiamo un nuovo simbolo iniziale, un nuovo terminale ed una produzione del nuovo simbolo iniziale che deriva il vecchio iniziale con il nuovo terminale, poi per il parsing all'insieme di terminali aggiungiamo anche il simbolo terminale

In breve aggiungiamo questo nuovo simbolo finale che comparirà in ogni derivazione così ad indicare la fine della stessa

**Parsing predittivo:** è una procedura non deterministica in cui si parte dal simbolo iniziale con l'obiettivo di generare la parola data e si guarda sempre il primo simbolo a  $sx$  che se è una variabile il parser dovrà predire quale sua produzione usare se invece è un terminale si fa un confronto

Ha una gestione LIFO

- **In ampiezza:** esaminiamo in parallelo tutti i possibili sviluppi
- **In profondità:** esaminiamo uno sviluppo alla volta ed in caso di errore si torna indietro

**Recursione  $dx$ :** situazione in cui il simbolo più a  $sx$  della produzione è lo stesso simbolo che la genera e questo fa fallire il parsing predittivo perché genera un ciclo infinito infatti il parser potrebbe scegliere la stessa regola

**Backtracking:** quando nella ricerca in profondità arriviamo ad un fallimento e quindi torniamo indietro cioè annulliamo l'ultima scelta di produzione

**Parsing a discesa ricorsiva:** interpretiamo la produzione  $S \rightarrow aB \mid bA$  come S ha successo se a ha successo e poi B oppure b ha successo e poi A

In breve creiamo una funzione per ogni variabile, partendo dal simbolo iniziale la funzione ad essa dedicata da vero se almeno una delle sue produzioni restituisce vero quindi se c'è un terminale deve fare match e se c'è una variabile allora la sua funzione si comporterà allo stesso modo

## Automi a pila

**Funzionamento informale:** legge l'input dal nastro in input e dalla cima della pila tramite pop, in base allo stato e i due simboli letti decide lo stato in cui andare poi scrive sulla pila con push e sposta la testina sul nastro a dx

**Automi a pila:** è una settupla  $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  dove Q è l'insieme degli stati,  $\Sigma$  è l'alfabeto in input,  $\Gamma$  è l'alfabeto relativo alla pila,  $\delta$  è la funzione di transizione,  $q_0$  è lo stato iniziale,  $Z_0$  è il simbolo iniziale della pila ed F è l'insieme degli stati finali

In breve come un normale automa ha un alfabeto, degli stati uno iniziale e un insieme di finali, in aggiunta ha poi la pila che sarebbe una memoria con un alfabeto che può essere diverso da quello in input ed in più la funzione di transizione stavolta dipenderà anche dal simbolo in cima alla pila

**Funzionamento:** l'automata parte dallo stato iniziale e la pila contiene il proprio simbolo iniziale. Per procedere in un altro stato bisogna tenere conto dello stato attuale, del simbolo in input e della cima della pila e se è presente una transizione (è un automa non deterministico) avanziamo e scriviamo in cima alla pila (push)

**Configurazioni interne:** relativamente al automa a Pila è una coppia  $(q, \gamma) \in Q \times \Gamma^*$  istante per istante salva lo stato in cui si trova e l'intero contenuto della pila in quel momento facendo come una istantanea

**Computazioni:** è una notazione che indica la partenza di un automa da una configurazione iniziale arrivando ad una finale dopo aver letto l'intera parola data

**Metodi di accettazione:** dato un automa a pila ed una parola diremo che essa è accettata per stato finale se la configurazione finale restituisce uno stato finale. È accettata per pila vuota se invece la pila è appunto vuota, se lo stato fosse finale e la pila vuota è accettata per stato finale e pila vuota

L'insieme delle parole accettate per stato finale formerà il linguaggio accettato dall'automata per questa tipologia di accettazione

**Automi a pila deterministici:** un automa a pila è deterministico se per ogni stato, lettera dell'alfabeto e della pila la transizione contiene al più un elemento perciò esisterà al più una configurazione interna relativa a quella lettera

**Pila vuota e stato finale a tutti i metodi:** costruiamo un automa a pila con un unico stato finale che come primo passo scrive un marker all'inizio della pila poi simula il funzionamento dell'automata che accetta il linguaggio pila vuota e stato finale e quando si troverà in uno stato finale della simulazione con il marker sulla pila, un  $\epsilon$ -transizione permetterà di estrarlo così da svuotarla e passare al unico stato finale che ha

**Pila vuota a tutti i metodi:** si considerano tutti gli stati come finali in questo modo l'unica condizione che conta è la pila vuota

**Stato finale a tutti i metodi:** si aggiunge un nuovo stato finale con il compito di svuotare progressivamente la pila per ricondursi ad un caso già risolto \*\*\*

## Teorema di caratterizzazione dei linguaggi non contestuali

**Teorema:** un linguaggio è non contestuale se e solo se è riconosciuto da un automa a pila. Questo implica che data una grammatica non contestuale esiste un automa a pila che accetta per pila vuota il linguaggio generato e viceversa

**Dalla grammatica all'automa:** esaminiamo il simbolo più a sx della predizione e se è una **variabile** faremo una **sostituzione con uno dei suoi lati dx**, se è un **terminale** deve **coincidere con la prima lettera dell'input**, il risultato sarà che **avrò un solo stato** che accetterà il mio linguaggio questo perché lo stato **si limiterà solamente a controllare cosa c'è sulla pila** che terrà traccia dei simboli

- **Dimostrazione:** dimostreremo che **il nostro linguaggio corrisponde al linguaggio accettato dall'automa a pila con un solo stato**. Se una parola è **accettata** dall'automa allora **avremo una computazione che restituirà la parola vuota dopo averla letta tutta**, se andassimo **a ritroso** possiamo ricostruire la derivazione sx della parola nella grammatica. Ragionando **al contrario** **ogni parola del linguaggio si ottiene partendo dal simbolo iniziale e procedendo con una derivazione sinistra**, questo può essere **emulato con il nostro automa** a pila perciò può seguire la derivazione arrivando alla parola vuota

**Dall'automa alla grammatica:** dato un automa a pila con un solo stato **esiste** effettivamente una **grammatica non contestuale** che genera il linguaggio accettato

- **Dimostrazione:** **sostituiamo ad ogni transizione la coppia di predizione e match** ciò non cambia il linguaggio accettato dall'automa perché in fin dei conti fanno la stessa cosa di prima uno push l'altro pop. **Un automa fatto così è esattamente quello che viene generato dalla grammatica**

**Eliminazione degli stati:** dato un automa a pila **esiste effettivamente un automa a pila con un solo stato** che accetta il medesimo linguaggio

- **Dimostrazione:** dato un automa a pila bisogna realizzarne uno che abbia un solo stato e simuli il comportamento del primo. Allora **faremo un solo stato, l'alfabeto della pila saranno le triplette stato simbolo stato** ed il **simbolo iniziale** partirà dallo **stato iniziale**. Riguardo alla **funzione di transizione** una mossa **push spingerà la tripletta** mentre una mossa **pop** accadrà **se ci sarà match** tra la tripletta che sto simulando e quella sulla pila

### Parsing top-down deterministico

**Funzione FIRST:** data una grammatica non contestuale priva di  $\epsilon$ -produzioni la funzione FIRST è **l'insieme dei terminali che possono comparire come prima lettera di una conseguenza di una stringa** (insieme terminali e variabili). Se la prima lettera è un terminale mi fermo se è una variabile vedo cosa può generare

- **Importanza:** permette di **agire in modo deterministico sulla previsione** infatti **basta scegliere sempre le produzioni che iniziano con il prossimo carattere di input** e per scoprire quali sono usiamo FIRST

**Tabella di parsing:** per ottimizzare il parsing **calcoliamo FIRST per tutti i lati dx delle produzioni** e costruiamo una tabella in cui a **ogni coppia variabile e lettera corrispondono le produzioni per cui quella lettera appartiene al first del lato dx**

- **Generazione:** **se il lato dx inizia con un terminale** sarà **l'unico elemento nel FIRST**, **se inizia con una variabile** devo svilupparla e **tutti i terminali saranno nel FIRST**. Devo applicare queste due regole in maniera iterativa sfruttando le informazioni nell'iterata precedente

**Grammatica di classe LL(1):** una grammatica non contestuale priva di  $\epsilon$ -produzioni si dice di classe LL(1) **se per ogni coppia di produzioni distinte si ha che l'intersezione tra i FIRST di ognuna è vuota**, cioè **la tabella di parsing contiene al più una produzione per coppia assicurando così un parsing deterministico**

- **Algoritmo:** data una tabella di parsing, **inizio mettendo sulla pila un endmarker** e il **simbolo iniziale S**, **ripeto poi le azioni di predizione e match finché non arrivo a fare pop sull'endmarker**. La predizione qua sarà deterministica visto che controllerà la tabella di parsing e troverà una sola entrata per ogni coppia visto che ci troviamo con una grammatica LL(1)

**Funzione FOLLOW:** è **l'insieme dei terminali che possono comparire dopo una variabile**. **Aiuta a gestire le  $\epsilon$ -produzioni** infatti se follow ci restituisce il carattere successivo nell'input allora possiamo scegliere la regola altrimenti il parser avrebbe fatto scomparire una variabile non sapendo se avesse fatto la mossa giusta

- **Tabella di parsing:** visto che abbiamo introdotto le  $\epsilon$ -produzioni le righe della tabella dovranno rispettare anche la **condizione che la parola vuota appartenga a FIRST e che il simbolo successivo nell'input stia in FOLLOW**
- **Grammatica di classe LL(1):** si aggiunge una **regola** ossia che se ho una  $\epsilon$ -produzione allora il FOLLOW di una variabile non deve avere elementi in comune con il FIRST di una stringa
- **Generazione:** partendo al **simbolo iniziale seguito dall'endmarker**, se una **variabile** è seguita da una stringa allora il suo FIRST è contenuto nel FOLLOW della variabile e se quella stringa può sparire allora anche il FOLLOW della variabile che la produce è contenuta nel FOLLOW

### Proprietà di chiusura dei linguaggi non contestuali

**Operazioni regolari:** la **classe dei linguaggi non contestuali è chiusa per le operazioni regolari** quali unione, concatenazione e chiusura di Kleene

- **Dimostrazione:** vogliamo dimostrare che **facendo operazioni regolari il risultato sarà sempre una grammatica non contestuale**. Partiamo da **due grammatiche** senza variabili in comune, **facendo l'unione** aggiungerò una **produzione con le regole di entrambe** ciò mi permetterà di generare le parole di entrambe. Per la **concatenazione** aggiungerò la **produzione in cui prima ci sarà una regola di una e poi di un'altra** così da dover derivare in ordine ed ottenere due parole complete da entrambi. Infine per la **chiusura di Kleene** aggiungo la **produzione in cui richiamo ricorsivamente una regola e metto anche  $\epsilon$  per potermi fermare**. In tutti e 3 i casi le produzioni sono di tipo non contestuale

**Intersezione con linguaggi regolari:** **l'intersezione di un linguaggio non contestuale con un linguaggio regolare è un linguaggio non contestuale**

- **Dimostrazione:** un **linguaggio regolare** è accettato da un **automa a stati finiti** mentre uno **non contestuale** da un **automa a pila con un unico stato**. **L'intersezione** tra i due linguaggi è **accettata per stato finale e pila vuota da un automa a pila che simula i due automi in parallelo**

**Linguaggio di Dick:** è il linguaggio sull'alfabeto  $\Sigma_n = \{... a_n, ... b_n\}$  generato dalla grammatica con le produzioni  $S \rightarrow a_i S b_i S, S \rightarrow \epsilon$

In breve **rappresenta tutte le sequenze di parentesi ben bilanciate**

**Morfismi:** un morfismo è una funzione che **trasforma le parole di un alfabeto in parole di un altro alfabeto** rispettando la concatenazione (la trasformazione di una parola concatenata equivale alla concatenazione delle trasformazioni singole)

**Teorema Chomsky, Schutzenberger:** un **linguaggio** è **non contestuale** se e soltanto se **esiste** un intero **k** positivo, un **linguaggio regolare** e un **morfismo** tali che il **linguaggio è uguale al morfismo dell'intersezione tra il linguaggio di Dick su k e il linguaggio regolare**

In breve il fulcro di ogni linguaggio non contestuale è una **struttura di parentesi bilanciate a cui vengono applicate restrizioni** tramite il linguaggio regolare che fa da filtro selezionando solo le parole che rispettano quell'espressione regolare e **poi i simboli vengono riscritti applicando il morfismo** che traduce nei simboli dell'alfabeto di partenza

### Domande & risposte

1. **Grammatica a Struttura di Frase: Cos'è una conseguenza diretta e una derivazione?** Una **grammatica a struttura di frase** è una **quadrupla** formata da un **vocabolario** totale finito, un **alfabeto** dei terminali, un insieme delle **produzioni** variabile-terminale ed un **simbolo iniziale**. Ogni grammatica poi genera un linguaggio in cui sono presenti **conseguenze dirette** ossia quando da una stringa ne otteniamo un'altra **applicando una sola regola** oppure **derivazioni** quando per ottenere una stringa bisogna **applicare più produzioni** di seguito

2. **Differenza tra  $V$  e  $\Sigma$ :** Qual è la differenza tra il vocabolario totale e l'alfabeto dei terminali? La differenza tra vocabolario totale e alfabeto dei terminali è che nel vocabolario totale sono presenti anche le variabili rappresentate per notazione con la lettera maiuscola mentre nell'alfabeto dei terminali sono presenti solo appunti i terminali rappresentati con la lettera minuscola
3. **Linguaggio Generato:** Come si definisce il linguaggio generato da una grammatica a struttura di frase? Il linguaggio generato da una grammatica a struttura di frase è l'insieme di tutte le frasi senza variabili ottenibili partendo dal simbolo iniziale ed applicando tutte le regole
4. **Gerarchia di Chomsky: Com'è strutturata?** S'inizia con le grammatiche di tipo 0 altro nome di quelle a struttura di frase ed è il tipo di default visto che è priva di restrizioni. Poi ci sono le grammatiche di tipo 1 o sensibili al contesto in cui una variabile può essere sostituita in una produzione solo se si trova in un determinato contesto e la lunghezza della parola sia maggiore o uguale rispetto al lato sx (questa proprietà si chiama monotonicità), poi ci sono le grammatiche di tipo 2 o non contestuali che è l'opposto del tipo 1 infatti a dx ci può stare qualsiasi combinazione di variabili e/o terminali, infine ci sono quelle di tipo 3 o regolari in cui a dx ci dev'essere almeno un terminale o da solo o seguito da un'altra variabile. Ricapitolando abbiamo tipo 0 struttura di frase poi tipo 1 sensibili al contesto poi tipo 2 non contestuali poi tipo 3 regolari
5. **Grammatiche Regolari (Tipo 3): Come sono definite?** Una grammatica è regolare se le sue produzioni sono nella forma  $a \rightarrow b$  con  $a$  singolo terminale o con  $a$  terminale e variabile
6. **Grammatiche Ambigue:** Cosa si intende per grammatica ambigua e non ambigua? una grammatica è ambigua quando per ogni parola del linguaggio c'è associato più di un albero di derivazione mentre è non ambigua se vale l'inverso
7. **Derivazione Sinistra: Cos'è e qual è la sua importanza?** Una derivazione sx è quando nel lato dx di una produzione andiamo a derivare la variabile più a sx, la sua importanza è che garantisce un'unica derivazione sx ad ogni albero di derivazione stabilendo una sequenza di derivazione
8. **Grafo di un Automa a Stati Finiti: Come si associa un grafo a un automa?** Ogni automa a stati finiti ha associato un grafo in cui i vertici sono gli stati dell'automa, le frecce da stato a stato rappresentano le transizioni ognuna avrà un'etichetta con una lettera dell'alfabeto dei terminali e poi lo stato iniziale sarà identificato da una freccia entrante mentre quelli finali dai doppi bordi
9. **Funzione Delta Estesa: A cosa serve e come funziona?** La funzione di transizione estesa semplicemente invece che restituire lo stato lettera per lettera restituisce direttamente lo stato che si ottiene alla fine della lettura dell'intera parola
10. **Automa a Stati Finiti Non Deterministico: Qual è la differenza con quello deterministico?** La differenza è che negli automi deterministici per ogni stato esiste esattamente una freccia con quella etichetta mentre in quelli non deterministici potrebbero esserci zero o più frecce con la stessa etichetta
11. **Passaggio da Grammatica Regolare ad Automa: Come si passa da una grammatica di tipo 3 a un automa a stati finiti?** Per il teorema di Kleene un linguaggio è regolare se è riconosciuto da un automa a stati finiti perciò se ho un linguaggio regolare c'è un automa associato. La struttura sarà un automa non deterministico con  $\epsilon$ -transizioni ed un solo stato finale, per crearlo analizzeremo pezzo per pezzo l'espressione regolare applicando determinate costruzioni ad ogni pezzo, per una lettera faremo due stati collegati tramite freccia la cui etichetta sarà quella lettera passando poi alle operazioni per l'unione bisognerà creare un nuovo stato iniziale ed un nuovo finale che si collegheranno ai vecchi, ogni operazione avrà la sua regola di costruzione
12. **Lemma di Iterazione per i Linguaggi Regolari: Cosa afferma e a cosa serve?** Dice che in ogni parola del linguaggio regolare superiore di lunghezza ad un interno  $n$  si può trovare un fattore non vuoto che può essere iterato rimanendo nel linguaggio, questo perché la parola potrà essere spezzata in 3 parti la prima che va dallo stato iniziale all'inizio del ciclo, la seconda che sarà la parte iterata mentre la terza andrà dalla fine del ciclo allo stato finale.
13. **Espressione Regolare: Cos'è?** Un'espressione regolare è una sequenza di simboli costruita seguendo le seguenti regole quali sono espressioni regolari qualsiasi singola lettera dell'alfabeto e il

linguaggio vuoto e poi se ho **due espressioni regolari** se facessi **operazioni regolari** come unione concatenazione e chiusura di Kleene (tutte le parole che si possono formare concatenando un numero qualsiasi di parole di quel linguaggio) il **risultato** rimarrebbe un'espressione regolare

14. **Parola Vuota nei Linguaggi Regolari: Come viene gestita?** Nei **linguaggi regolari** la **parola vuota** è **ammessa** solo **se prodotta** dal **simbolo iniziale** che poi **non dovrà comparire** nei **lati dx** delle produzioni
15. **Automa Minimo: Cos'è e come si costruisce?** L'**automa minimo** è un **automa deterministico** che accetta il linguaggio col minimo numero di stati possibili, **per costruirlo si raggruppano i vecchi stati per stesso comportamento** tramite classi di equivalenza (la regola che usiamo per raggruppare è l'equivalenza di Nerode cioè che due stringhe si comportano allo stesso modo per qualsiasi suffisso che aggiungiamo ad entrambe). la suddivisione deve stare anche ad altre regole cioè **stati finali e non finali non devono stare in un unico gruppo** e la suddivisione va fatta con il minor numero di gruppi, finita la suddivisione **questi gruppi diverranno i nuovi stati** dell'automa e le nuove frecce verranno messe in base alle destinazioni originali
16. **Equivalenza di Nerode: Cos'è?** L'**equivalenza di Nerode** dice che **due parole sono equivalenti se aggiungendo ad entrambe la stessa stringa sono intercambiabili** perciò avranno **stesso comportamento** e se una va fuori dal linguaggio l'altra fa lo stesso
17. **Teorema di Nerode: Cosa afferma e come si dimostra?** Il **teorema di Nerode** afferma che **dato un linguaggio sull'alfabeto  $\Sigma$  sigma sono proposizioni equivalenti che il linguaggio è regolare, che è unione di classi di una congruenza destra e che ogni classe d'equivalenza corrisponde ad uno stato nell'automa minimo**. Per dimostrarlo partiamo dal fatto che il **linguaggio essendo regolare per Kleene ha un automa deterministico** che lo accetta, **lo uso per raggruppare le parole per stato di terminazione** così facendo il risultato sarà l'unione di tutti quei gruppi che terminano in uno stato finale. Questo **raggruppamento sarà una congruenza dx** (se due parole sono considerate equivalenti devono rimanerlo anche attaccando la stessa lettera alla dx di entrambe) infatti **e due parole si trovano sullo stesso gruppo è perché hanno stesso comportamento** perciò **se attacchiamo una lettera ad ognuno l'automa essendo deterministico le porterà allo stesso stato** rispettando il principio di congruenza dx, per di più **ha indice finito** visto che mettendo che **gli stati siano n**, i gruppi non potranno essere più di n visto che **son formati da stati** e questa cosa **vale** anche per le classi d'equivalenza visto che il linguaggio è formato da alcuni gruppi creati in precedenza ed ognuno abbiamo visto che è una congruenza dx perciò **due parole equivalenti o sono entrambe dentro il linguaggio o entrambe fuori** e questa è la definizione d'equivalenza per Nerode
18. **Automi a Pila: Qual è il modello astratto di macchina su cui si basano?** Un **automa a pila** è un **automa a stati finiti con aggiunta una pila** che sarebbe una memoria che ha un alfabeto che potrebbe differire da quello di input e che è **determinante insieme a stato e input** per la funzione di transizione
19. **Linguaggi e Automi a Pila: Quale tipo di linguaggio della gerarchia di Chomsky è accettato dagli automi a pila?** Un **automa a pila accetta i linguaggi di tipo 2 o non contestuali** e questo implica che una grammatica non contestuale abbia associato un automa a pila. Questo automa userà la **pila per tenere traccia della predizione** cioè prendendo il simbolo più a sx di quest'ultima se questo è una variabile sostituisco con uno dei suoi lati dx (push) mentre se è un terminale deve fare match con l'input (pop). Ci sarà **un solo stato che si occuperà di scrivere sulla pila** (push) e leggere la cima (pop)
20. **Analisi Lessicale: In cosa consiste?** L'**analisi lessicale** dato un codice **si occupa di identificare il lessema più lungo tramite un pattern** ed in **ordine di priorità restituisce il primo token** che da una corrispondenza. Partiamo dai **token** che sono simboli astratti che **rappresentano un'unità lessicale** tipo identificatore, un **lessema** invece è una **sequenza di caratteri** che si trova nel codice e ogni lessema **ha associato un token** specifico come il token per la punteggiatura o per gli operatori, infine i **pattern** sono **espressioni regolari** che **descrivono i lessemi**

- 21. Costruzione di un Albero (Parsing): Come fa un calcolatore a costruire un albero di derivazione?** ci stanno due modi o utilizzando l'algoritmo CYK in cui data una grammatica in forma normale di Chomsky (perciò solo due tipi di produzione o un solo terminale o due variabili) e una parola, la scriviamo come stringa di lettere e cerchiamo di trovare le variabili che possano generare i singoli caratteri, itero questo passaggio per sottostringhe sempre più lunghe utilizzando i risultati ottenuti prima. Arriverò alla fine quando la sottostringa equivarrà alla parola quindi verificherò che nell'insieme delle variabili ci sia il simbolo iniziale. Il secondo metodo è il parsing predittivo in cui data una parola e una grammatica non contestuale, partiamo dal simbolo iniziale con l'obiettivo di voler generare la parola, quindi prendo una produzione e guarderò sempre il primo simbolo a sx che se è una variabile dovrò sostituirla con uno dei suoi lati dx a scelta (ecco la predizione) se invece è un terminale dovrà matchare con il carattere che sto leggendo della parola. Entrambi i metodi restituiscono un albero di derivazione ma lo fanno partendo da direzioni diverse chi dalle foglie chi dalla radice
- 22. Parser Top-Down vs Bottom-Up: Qual è la differenza?** La differenza sta nella partenza, top down parte dal simbolo iniziale a scendere fino alle foglie mentre bottom up parte dalle foglie e risale di nodo in nodo fino al simbolo iniziale
- 23. Parsing in Ampiezza e in Profondità: Cosa significano queste due strategie?** Il parsing predittivo deve fare una scelta su quale produzione usare di una variabile ed una scelta sbagliata potrebbe bloccare il processo perciò ci stanno due modi per applicare l'algoritmo o in ampiezza in cui vengono esaminati in parallelo tutti i possibili sviluppi anche se è molto dispendioso oppure in profondità in cui si esamina uno sviluppo alla volta ed in caso di errore si può tornare indietro
- 24. Parsing Predittivo: Come funziona (nei casi deterministico e non deterministico)?** Il parsing predittivo data una parola ed una grammatica non contestuale partendo dal simbolo iniziale controllerà sempre il simbolo più a sx che se sarà una variabile andrà sostituita con uno dei suoi lati dx se invece è un terminale si fa il confronto con l'input. Nel caso non deterministico il momento della scelta del lato dx è una vera e propria predizione perché non sappiamo a priori quale sarà la scelta giusta nel caso deterministico invece abbiamo la tabella di parsing ossia una tabella in cui utilizzando due funzioni FIRST (insieme dei terminali che possono comparire come prima lettera nel lato dx di una stringa) e FOLLOW (insieme dei terminali che possono comparire dopo una variabile) per ogni coppia di variabile-lettera abbiamo la produzione che deve verificare una di queste condizioni: la prima la lettera corrisponde al first del lato dx, la seconda la parola vuota appartiene al first del lato dx e il simbolo successivo in input è nel follow della variabile, queste due condizioni garantiscono che ci sia al massimo una produzione per riga creando il determinismo
- 25. Algoritmo CYK: Come funziona?** L'algoritmo CYK funziona che data una grammatica in forma normale di Chomsky (produzioni o solo terminale o due variabili) e una parola scrive la parola come una sequenza di lettere e cerchiamo di trovare quali variabili possono generare i singoli caratteri ed iteriamo questo passaggio per sottostringhe sempre più lunghe utilizzando i risultati precedenti. Alla fine quando la sottostringa equivarrà alla parola controlla che nell'insieme delle variabili ci sia il simbolo iniziale
- 26. Linguaggio di Dick: Cos'è?** Il linguaggio di dick rappresenta tutte le sequenze di parentesi ben bilanciate, ha una sola variabile e tre tipi di produzione la prima dice che se ho una sequenza di parentesi posso metterla all'interno di altre, la seconda che se ho due sequenze valide concatenandole rimarrà valida e poi ho la parola vuota
- 27. Qual è la differenza tra la fase analitica e la fase sintetica della compilazione?** Nella fase analitica viene analizzato il sorgente per verificare che soddisfi le regole sintattiche e semantiche mentre nella fase sintetica viene generato il codice oggetto senza eseguire controlli perché si basa sull'analisi già fatta
- 28. Cosa sono le forme sentenziali e in cosa si differenziano dal linguaggio generato da una grammatica?** Una forma sentenziale è una frase che ottengo partendo dal simbolo iniziale ed

applicando le regole della grammatica mentre il linguaggio generato è l'insieme di tutte le forme sentenziali che non contengono variabili

29. Qual è la proprietà di monotonicità e a quale tipo di grammatica della gerarchia di Chomsky si applica? la monotonicità è una proprietà per cui ogni produzione ha il lato dx maggiore o uguale rispetto al sx ed è una proprietà delle grammatiche di tipo 1 o sensibili al contesto ed è una condizione necessaria
30. Cos'è l' $\epsilon$ -chiusura di uno stato e qual è il suo ruolo nel processo di determinizzazione? l' $\epsilon$ -chiusura di uno stato restituisce l'insieme di stati che da questo possiamo raggiungere con delle  $\epsilon$  transizioni ed il suo ruolo nel passaggio da automa non deterministico a deterministico è necessario perché individua le  $\epsilon$  transizioni e viene messo all'inizio calcolandola per lo stato iniziale e poi lo calcolo dopo ogni transizione
31. Qual è l'idea dietro il metodo di eliminazione degli stati per passare da un automa a un'espressione regolare? Per passare da un automa a un'espressione regolare abbiamo due metodi: quello induttivo in cui costruisco l'espressione regolare calcolando i percorsi tra due stati che usano stati intermedi fino ad un indice  $k$  e come caso base avrò solo frecce dirette mentre come passo induttivo aggiungo i percorsi che passano attraverso lo stato con indice  $k$ , alla fine avrò l'espressione regolare di tutte le etichette dei cammini tra i due punti. Il secondo metodo consiste nel selezionare uno stato intermedio dall'automa, eliminarlo e i suoi percorsi verranno assorbiti dalle frecce degli altri stati e questo processo lo stoppiamo quando resteremo con iniziale e finale e l'etichetta della freccia che li collega sarà l'espressione regolare
32. Qual è il primo passo dell'algoritmo di minimizzazione di un automa e perché è logicamente necessario? Il primo passo dell'algoritmo di minimizzazione consiste nel creare due gruppi principali: quello degli stati finali e quello dei non ed è logicamente necessario perché nello stesso gruppo ci stanno gli stati che per ogni simbolo portano a stati all'interno del gruppo, se prendiamo la parola vuota per uno stato finale è accettata mentre per uno stato non è rifiutata questo perché la parola vuota fa rimanere nello stato
33. Quali sono le due regole principali che un analizzatore lessicale segue per risolvere le ambiguità quando un pezzo di codice corrisponde a più pattern? Un analizzatore lessicale per risolvere le ambiguità innanzitutto prende il prefisso più lungo poi sceglie la classe di token con più alta priorità che da una corrispondenza e la più alta sono le parole chiave come per esempio if
34. Perché, nella procedura di semplificazione di una grammatica, è importante eliminare le variabili improduttive prima di quelle inaccessibili? Nella procedura di semplificazione di una grammatica è importante eliminare le variabili improduttive prima di quelle inaccessibili perché eliminando le improduttive potrebbero venire fuori nuove inaccessibili visto che ci potrebbe essere qualche variabile che compare nei lati dx delle improduttive
35. Qual è il vantaggio pratico di avere una grammatica in Forma Normale di Chomsky? Il vantaggio pratico di avere una grammatica in forma normale di Chomsky è che la struttura rigida sulle produzioni crea un albero binario completo vantaggio sfruttato dall'algoritmo CYK
36. Quali sono i tre diversi metodi di accettazione per un automa a pila? I tre diversi metodi di accettazione per un automa a pila sono: accettazione per stato finale ovvero quando lo stato in si trova l'automa è uno stato finale, accettazione per pila vuota e accettazione per stato finale e pila vuota ovvero si verificano entrambe le condizioni precedenti
37. A cosa serve la funzione FOLLOW e in quale caso è indispensabile per il parsing predittivo? La funzione follow data una variabile restituisce l'insieme di terminali dopo di essa in un lato dx ed è utile perché protegge il parser da una scelta sbagliata in caso di presenza di una  $\epsilon$  produzione che comporterebbe l'eliminazione di una variabile, infatti facendo follow possiamo vedere se dopo la variabile che si eliminerebbe c'è il simbolo in input
38. I linguaggi non contestuali sono chiusi rispetto all'intersezione? I linguaggi non contestuali sono chiusi rispetto all'intersezione solo con un linguaggio regolare infatti l'intersezione con un

linguaggio regolare restituisce un linguaggio non contestuale questo perché uno è accettato da un automa a stati finiti l'altro da uno a pila il risultato sarà un automa a pila che simulerà i due automi in parallelo

39. Qual è la differenza tra una variabile improduttiva e una variabile inaccessibile? La differenza tra una variabile improduttiva e una inaccessibile è che quella inaccessibile non compare in nessuna forma sentenziale mentre quelle improduttive sì ma non derivano nessuna stringa di terminali
40. Come fa un automa a pila con un solo stato a simulare una grammatica non contestuale? Un automa a pila con un solo stato riesce a simulare una grammatica non contestuale perché la pila si occupa di tenere memoria delle predizioni mentre lo stato si occupa di valutare il simbolo più a sx se variabile o terminale e questo alla fine è il comportamento della grammatica ovvero che ogni parola del linguaggio la si ottiene con una serie di derivazioni sx
41. Qual è il ruolo della Tabella dei Simboli e da quali fasi della compilazione viene utilizzata? La tabella dei simboli è appunto una tabella che contiene i riferimenti e i token dei lessemi e viene utilizzata una prima volta nell'analisi lessicale in cui viene riempita e poi nell'analisi semantica in cui si controlla che ogni operatore abbia gli operandi corretti
42. Perché la ricorsione a sinistra è un problema specifico per i parser top-down? La ricorsione a sx è un problema specifico per i parser top down perché il parser quando va a fare la predizione potrebbe scegliere il lato dx con di nuovo a sx la variabile e potrebbe generare un loop
43. Qual è l'obiettivo della riduzione in Forma Normale di Chomsky e quali sono i passaggi principali per ottenerla? L'obiettivo della riduzione in forma normale di Chomsky è ottenere una grammatica con una struttura rigida che dia un albero binario e i passaggi sono: eliminare le epsilon produzioni e quelle unarie e poi per ogni terminale a dx che non compare solo introduco una variabile che lo produca e la vado a sostituire
44. Qual è la differenza pratica tra il metodo induttivo e il metodo di eliminazione degli stati per derivare un'espressione regolare da un automa? La differenza pratica tra i due metodi per derivare un'espressione regolare da un automa è che con il metodo di eliminazione vado ad eliminare gli stati dell'automata per raggiungere l'espressione mentre con il metodo induttivo li utilizzo controllando i cammini
45. Come fa un automa a pila che simula una grammatica a gestire le produzioni e il match con l'input pur avendo un solo stato? Un automa a pila con un solo stato riesce a gestire le produzioni perché la pila tiene traccia delle predizioni mentre lo stato a decidere quando fare push o pop e partendo dal simbolo iniziale deriva ogni parola del linguaggio procedendo con una derivazione a sx proprio come fa la grammatica
46. In un parser predittivo deterministico, in quale situazione specifica è indispensabile consultare l'insieme FOLLOW di una variabile? È indispensabile quando abbiamo a che fare con la parola vuota questo perché il parser potrebbe andare ad eliminare una variabile commettendo un errore, il follow ci permette di vedere l'insieme di terminali dopo una specifica variabile perciò se il simbolo in input appartiene possiamo guidare il parser
47. Come si dimostra che la classe dei linguaggi non contestuali è chiusa rispetto alla concatenazione? La classe dei linguaggi non contestuali è chiusa rispetto alla concatenazione infatti prendendo due grammatiche non contestuali senza variabili comune aggiungendo una produzione in cui prima metto una regola di una e poi una dell'altra otterrò due parole complete da entrambi che appariranno sempre ad una grammatica non contestuale
48. Qual è la differenza fondamentale tra il Lemma di Iterazione per i linguaggi regolari e quello per i linguaggi non contestuali, e perché questa differenza è significativa? La differenza fondamentale tra il lemma di iterazione per i linguaggi regolari e quello per i non contestuali è che nel primo viene iterata una sola stringa nel secondo due

49. Qual è il vantaggio pratico principale di una grammatica in Forma Normale di Greibach rispetto a una in Forma Normale di Chomsky? Il vantaggio principale nell'avere una grammatica in forma normale di Greibach è che la lunghezza di una parola equivale ai passi per ottenerla
50. Perché l'algoritmo CYK ha una complessità temporale di  $O(n^3)$ ? Da quali cicli annidati deriva? L'algoritmo CYK ha complessità di  $n$  alla terza perché abbiamo il ciclo in cui prendiamo sottostringhe piano piano più grandi, quello con cui scorriamo le varie sottostringhe che si vengono a creare e quello in cui cerchiamo per ognuna la produzione
51. Spiega brevemente come il Teorema di Kleene collega tre concetti fondamentali: le espressioni regolari, gli automi a stati finiti e le grammatiche di tipo 3. Il teorema di Kleene dice che un linguaggio è regolare se riconosciuto da un automa a stati finiti, un linguaggio regolare deriva da una grammatica di tipo 3 e seguendo il teorema ne deriva che un linguaggio regolare abbia un automa a stati finiti associato e visto che un linguaggio ha un'espressione regolare associata possiamo dire che un'espressione regolare ha un automa associato