# Capitolo 1 – slide 1

***Numero 15 - Importante***

La descrizione di un linguaggio avviene su 3 livelli:

1. **Sintassi**: consiste nelle regole di formulazione
2. **Semantica**: consiste nel significato di quello che si scrive
3. **Pragramatica**: consiste nel stabilire in quale modo frasi corrette e sensate vengono usate.
4. **Implementazione (SOLO IN LINGUAGGI ESEGUIBILI):** spiega come eseguire il codice scritto rispettando la semantica.

*Numero 16 – Poco imp*

Nella dimensione sintattica, dobbiamo considerare l'aspetto:

- **Lessicale**: è costituito dall'insieme delle parole che possiamo usare. Viene descritto attraverso dizionari (nei linguaggi naturali) o strutture complesse nel caso dei linguaggi artificiali. Un errore lessicale è dato dall'uso di una parola che non è compresa nel vocabolario.
- **Grammaticale**: è costituito dall'insieme delle frasi corrette che si possono costruire attraverso il lessico. Questo aspetto viene descritto attraverso le regole grammaticali. Le frasi generabili sono infinite.

***Numero 17 – Importante***

Un **alfabeto** è un insieme finito di simboli, mentre una **parola o stringa** definita su un alfabeto A è una sequenza finita di simboli in A. **A* è un insieme infinito contabile, ed è costituito dalla stringa epsilon (stringa vuota) e da tutte le stringhe di lunghezza da 1 a infinito.** Questo è dimostrato attraverso la tecnica di Dove Tailing.

*Numero 18 – Poco imp*

- Con potenza di una stringa si intende una stringa ripetuta/concatenata con sé stessa tante volte quanto il valore della potenza. Se il suo valore è 0, allora si intende la stringa epsilon.
- Con potenza di un linguaggio, si intende: $L^0=\{\varepsilon\}$, $L^{n+1} = L \bullet L^n$
- Con contatenazione di un linguaggio si intende: $L_1 \bullet L_2 = \{w_1 w_2 \mid w_1 \in L_1 \text{ e } w_2 \in L_2\}$
- Con stella di Klenee di un linguaggio si intende: $L^* = \bigcup_{n \geq 0} L^n$

# Capitolo 1 – slide 2

***Numero 19 – MOLTO IMPORTANTE***

**19) Una grammatica libera dal contesto è una quadrupla (NT, T, R, S), dove NT è l'insieme dei nonTerminali, T è l'insieme dei terminali, R è l'insieme delle produzioni (nella forma $V \to \omega$), ed è S il simbolo nonTerminale iniziale. Derivare una stringa vuol dire scrivere v in una sequenza di passi in w, ovvero, più in modo schematico:**

$$\frac{v = xAy \quad (A \to z) \in R \quad w = xzy}{v \Rightarrow w} \qquad x, y, z \in (T \cup NT)^*$$

**Il linguaggio generato da una grammatica libera è: $L(G) = \{w \in T^* \mid S \to^* w\}$,** ovvero a partire del simbolo iniziale S riusciamo in una serie di passi a derivare la stringa w.

**20) Un albero di derivazione è un modo di riassumere tante derivazioni diverse, ma tutte equivalenti. Rappresenta quindi uno schema associato a una derivazione. Può essere rightmost o leftmost (a seconda della derivazione se è rmost o lmost).** Esiste una corrispondenza biunivoca fra alberi di derivazione e derivazioni, siccome data una derivazione leftmost (o rightmost),

Def: Data una grammatica libera $G=(NT, T, S, R)$, un albero di derivazione (o di parsing) è un albero ordinato in cui:

- ogni nodo è etichettato con un simbolo in $NT \cup \{\varepsilon\} \cup T$
- la radice è etichettata con S
- ogni nodo interno è etichettato con un simbolo in NT

ad essa viene associata un solo albero di derivazione, e per ogni albero di derivazione esiste una sola derivazione rightmost che lo genera.

**21) Una grammatica è ambigua quando abbiamo più di una derivazione per la stessa stringa/produzione.** Un esempio è la grammatica S → a | b | c | S + S | S*S per la produzione a*b+c.
**Un linguaggio è ambiguo quando tutte le grammatiche G tali che L(G) = L sono ambigue (**ovvero è generato solo da grammatiche ambigue**). Un esempio può essere L1 e L2 (PAG 39 SLIDES 3),** dove tuttle le stringhe hanno doppia derivazione: una derivazione che fa uso delle produzioni che generano L2, e l'altra verso le produzioni che generano L3.

22) Si può rimuovere l'ambiguità dalla grammatica delle espressioni aritmetiche aggiungendo zucchero sintattico come parantesi, oppure aggiungendo delle nuove produzioni in modo che l'associatività sia sempre a sinistra e facendo in modo che un segno abbia la precendenza rispetto ad un altro (se una produzione è più interna nell'albero generato, allora ha la precedenza). Se aggiungiamo le produzioni però, il linguaggio ottenuto risulta diverso da quello di partenza, siccome potrebbe non avere lo stesso albero di derivazione per le stesse produzioni.

**23) L'albero di sintassi astratta è un albero di derivazione costruito a partire da quello di sintassi "concreta". Quest'ultimo contiene anche zucchero sintattico e terminali ausiliari, che sono semanticamente irrilevanti.** L'albero di sintassi astratta, invece, possiede esclusivamente terminali.

**24) Esempi di vincoli sintattici contestuali:**

- il numero e il tipo di parametri di una chiamata di procedura **deve combaciare** con il numero ed il tipo di parametri della dichiarazione delle procedura.
- Devi dichiarare una variabile prima di usarla
- il valore che si assegna ad una variabile deve essere compatibile con il tipo di variabile stessa.

**Questi vincoli non possono essere catturati da una grammatica libera**, in quanto non sono in grado di descrivere dei vincoli dipendenti dal contesto. Per risolvere, possiamo fare uso di grammatiche dipendenti dal contesto, oppure di controlli ad hoc.

**25) Con *semantica statica* si intende l'insieme di quei controlli che possono essere fatti sul testo del programma senza avviarlo. Con *semantica dinamica*, invece, si intendono quei controlli che si fanno "simulando" l'esecuzione del programma, ovvero una rappresentazione formale del programma che può mostrare errori durante l'esecuzione.**

**26) Fasi di un compilatore:**

1. **Analisi lessicale:** in questa fase si fa uso di uno **scanner**, e si spezza il programma nei componenti sintattici primitivi, detti *tokens*. Si controllano che il lessico sia ammissibile, e si riempie parzialmente la tabella dei simboli.
2. **Analisi sintattica:** a partire dalla lista dei tokens, il parser produce l'albero di derivazione del programma, riconoscendo se le frasi sono sintatticamente corrette. In questa fase, si fa uso di un **Parser**.
3. **Analisi semantica:** Si eseguono controlli sulla semantica statica, per rilevare eventuali errori semantici. In questa fase: si arricchisce l'albero di derivazione, verifica i tipi durante gli assegnamenti…
4. **Generazione della forma intermedia**, che sarà facilmente traducibile nel linguaggio macchina della target machine. Questo codice inoltre è composto da istruzioni molto semplici. Durante la generazione della forma intermedia, inoltre, si genera anche l'albero sintattico, ricavato dall'albero di derivazione.
5. **Ottimizzazione**: Si ottimizza il codice intermedio in modo da renderlo più efficiente:
   Si espandono le funzioni, si elimina il dead code, si mettono fuori dai cicli sottoespressioni che non variano…
6. **Generazione del codice:** viene generato codice per una specifica architettura (include anche l'assegnazione dei registri e ottimizzazioni specifiche per l'architettura).

Ogni errore rilevato nelle prime 3 fasi non genera un'interruzione della compilazione, bensì genera un messaggio d'errore.

27) definire la semantica serve al:
- programmatore, siccome deve sapere cosa deve fare il suo programma, dimostrandone le sue proprietà
- Progettista del linguaggio, siccome deve dimostrare le proprietà del linguaggio (come la turing-completezza).
- All'implementatore del linguaggio, che deve dimostrare la correttezza della sua implementazione.

28) per definire la semantica di un linguaggio di programmazione, si usano due tecniche principali:
- **operazionale**: si costruisce una specie di automa che, passo-passo, mostra l'effetto dell'esecuzione delle varie istruzioni. Si ha quindi una enfasi su come si calcola.
- **Denotazionale**: si associa ad ogni programma una funzione da un input a un output (incluse strutture ausiliarie come ambiente e memoria). Si ha quindi una enfasi su cosa si calcola.

**29) Ecco le regole di semantica operazionale: (VEDERE SLIDES 5 pag. 5)**

30) Pragmatica ed implementazione sono altri due aspetti della descrizione di un linguaggio.
- Con pragmatica si intende un insieme di regole/consigli sul modo in cui è meglio fare uso delle isturzioni a propria disposizione (es. evitare il goto).
- L'implementazione consiste nella costruzione di un compilatore per una macchina ospite già realizzata, costruendo così una macchina astratta per il linguaggio.

# Capitolo 2 – slide 6

**31) Un analizzatore lessicale riconosce** nella stringa in ingresso gruppi/sequenze di simboli che corrispondono a delle categorie sintattiche specifiche. Dunque, trasforma il programma/stringa in input in una sequenza di simboli detti token.

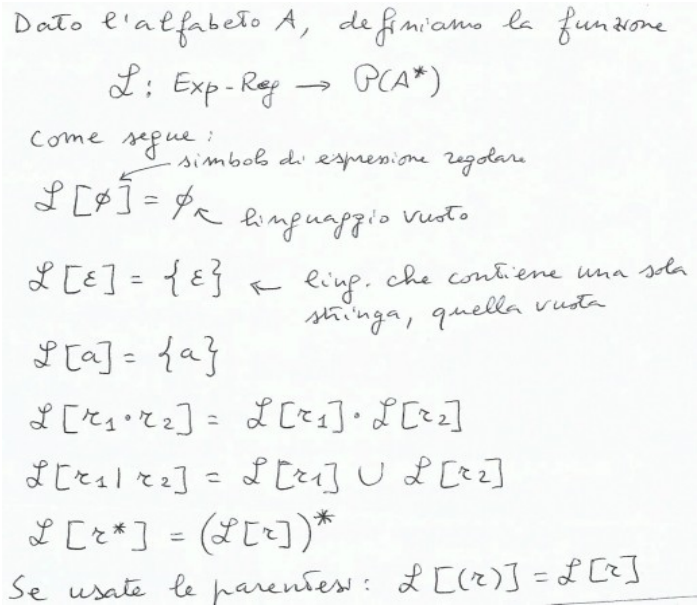
32) Un token è una coppia (nome, valore) dove:
- per nome si intende l'informazione che identifica una classe di token (e quindi rapprensenta una categoria sintattica).
- per valore si intende l'informazione che indentifica uno specifico token, e corrisponde a una sequenza di simboli del programma in ingresso.

33) Un *pattern* rappresenta la descrizione generale della forma dei valori di una classe di token.
Un *lessema* è una stringa istanza di un pattern (es. il valore è un lessema istanza del pattern).
[In generale, ad ogni nome di una data categoria sintattica è associato un pattern che specifica che i possibili valori che possono essere presi per quel nome.]

**34) Una espressione regolare**, fissato un alfabeto A = {a1, a2 … an}, è definita su A tramite la seguente BNF:

$$r ::= \varnothing \mid \varepsilon \mid a \mid r \bullet r \mid r|r \mid r^*$$

Questa rappresenta la sintassi delle espressioni regolari.
Il linguaggio associato ad una espressione regolare è, invece:



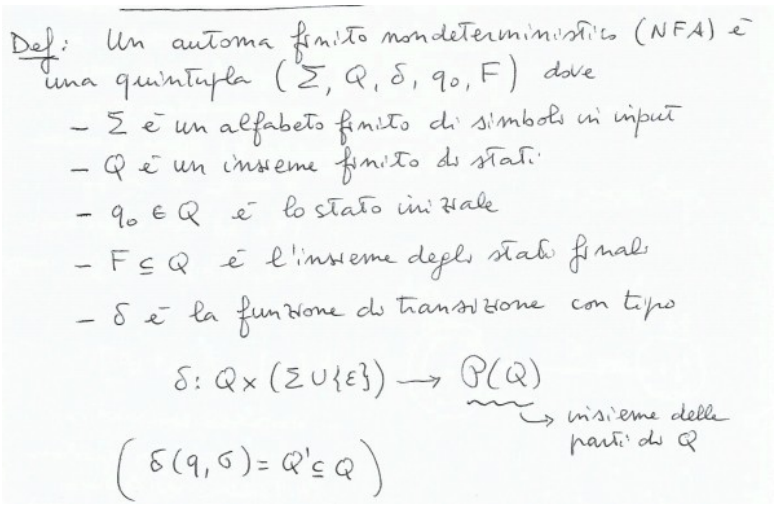
**35) Un linguaggio L è detto regolare** sse esiste una espressione regolare r tale per cui L = L[r] (ovvero il linguaggio L è generato da una espressione regolare).
Ogni linguaggio finito è anche regolare, ed esistono linguaggi regolari infiniti (es. $\{a^n b \mid n \geq 0\}$).

36) Due espressioni regolari r e s sono equivalenti sse L[r] = L[s] (ovvero le due espressioni regolari denotano lo stesso linguaggio).
Alcune leggi di equivalenza sono:

| | | | | |
|---|---|---|---|---|
| $r \mid s \cong s \mid r$ | $(r^*)^* \cong r^*$ | $r \mid r \cong r$ | $\varepsilon \bullet r \cong r$ | $\varnothing^* \cong \varepsilon$ |

37) Le definizioni regolari rappresentano un altro modo di definire i linguaggi regolari, e una definizione regolare è definita a partire da una lista di definizioni.

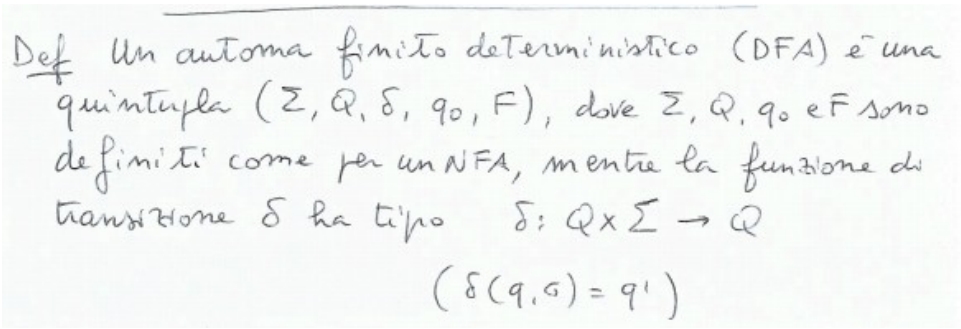
**38) Ecco la definizione di NFA:**




Con non-determinismo si intende che per ogni coppia (q, s) [stato, simbolo] ci sono più mosse possibili, e non una e una sola.

39) Un linguaggio è accettato da un NFA se esiste un cammino dallo stato iniziale ad uno finale per ogni stringa.
Due NFA sono equivalenti se e solo se accettano lo stesso linguaggio.

**40) Definizione di NFA:**




Un DFA è un particolare tipo di DFA, tale che non contiene mosse epsilon e per cui l'insieme delle mosse possibili per ciascuno stato è sempre un singoletto.

41) Si fa uso della costruzione per sott'insiemi, con cui, attraverso le ε-closure si permette di calcolare gli stati del DFA e delle loro relazioni (con archi).
L'algoritmo delle e-closure è simile a una visita in ampiezza, in cui si scelgono gli archi epsilon e si aggiunge il nodo adiacente all'insieme dei nodi.
Il numero di stati nel caso pessimo è $2^n$, ovvero l'insieme delle parti del NFA.

**42) Sia N un NFA e Mn l'automa ottenuto attraverso costruzione per sott'insiemi. Allora Mn è un DFA e si ha che il linguaggio riconosciuto da N è pari al linguagigo ricosciuto da Mn. (L[N] = L[Mn]).**

**43) Vedere pag 31 (ovvero la prima facciata) del pacco di slides 7.**

**44) Una grammatica libera è regolare se ogni sua produzione è della forma V → aW, oppure se è V → a, dove V, W sono NonTerminali, mentre a è un Terminale. Per il simbolo iniziale S, valgono anche le produzioni epsilon (S → ε).**

45) Per associare una grammatica regolare ad un equivalente NFA, si associa ad ogni stato dell'NFA un NonTerminale, e per ogni produzione V → aW si costruisce un arco "a" dallo stato V a quello W , mentre per ogni produzione V → a si crea un arco "a" da uno stato V ad uno stato epsilon.

**46) Per costruire una grammatica regolare Gm da un DFA M, la grammatica Gm che costruiamo dovrà avere:**

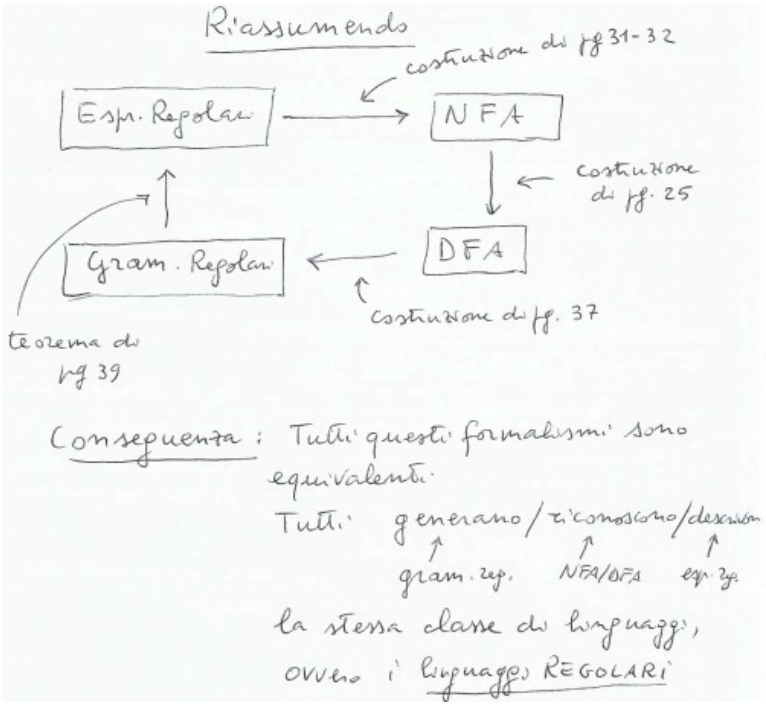
- **Per terminali, gli l'alfabeto di M.**
- **Per NonTerminali, gli stati di M.**
- **Per simbolo iniziale, lo stato iniziale di M.**
- **Per produzioni R:**
  - **per ogni δ(qi, a) = qj, la produzione qi → a qj (se qj ∈ F, allora qi → a)**
  - **se q0 (ovvero lo stato iniziale) è uno degli stati finali, allora aggiungo anche q0 → ε**

**47) Il linguaggio definito da una grammatica regolare è un linguaggio regolare, dunque è possibile costruire una espressione regolare Sg tale che L(G) = *L*[Sg].**
**Per costruire una grammatica regolare partendo da una espressione regolare, si fa così:**

1. **Ricavo il linguaggio dall'ultima produzione.**
2. **Sostituisco il linguaggio ricavato dentro le produzione subito dopo.**
3. **E così via, fino ad arrivare alla prima produzione**

**NB. Dove abbiamo la testa della produzione tra i nonTerminali della produzione stessa, sostiuiamo con la stella di kleene.**

48)




**49) Due stati q1 e q2 di un DFA N sono indistinguibili (o equivalenti) quando:**
**L[N, q1] = L[N, q2], ovvero quando il linguaggio definito a partire da q1 è lo stesso linguaggio definito a partire da q2.**

**Oppure, quando $\forall x \in \Sigma^*$ (ovvero per ogni stringa x):**
**$\hat{\delta}(q1, x) \in F$ se e solo se $\hat{\delta}(q2, x) \in F$, dove $\hat{\delta}$ è la funzione di transizione che prende una stringa e uno stato e ritorna lo stato in cui si arriva.**
Sono distinguibili se $\hat{\delta}(q1, x) \in F$ ma $\hat{\delta}(q2, x) \notin F$.

50) Si usa l'algoritmo a scala per permettere di minimizzare facilmente i DFA.
Questo funziona in questo modo:

1. **Si crea una tabella a scala, senza coppie identiche {quindi niente (A,A), (B,B)…}, dunque nelle righe si elencano gli stati in ordine alfabetico (partendo da B), mentre nelle colonne si elencano gli stati in ordine (finendo con la penultima lettera degli stati).**
2. **Al passo 0, marco con X0 tutte le coppie distinte tali per cui sono composti da (Finali, NonFinali).**
3. **Al passo 1, marco con X1 tutte le coppie (ovviamente non ancora marcate) per cui, ad un passo per la stessa lettera dell'alfabeto, raggiungo una casella/coppia marchiata con X0.**
4. **Al passo 2, faccio la stessa cosa del passo 1, ma per le coppie marcate con X1.**
5. **Continuo finché non ho più coppie da marcate.**

51) A sto punto, costruiamo il DFA minimizzato. Se tutte le caselle sono state marchiate, allora il DFA è minimo, altrimenti le caselle non marchiate rappresentano gli stati equivalenti (le due coordinate rappresentano gli stati che sono equivalenti).

**52) Lex è un software Unix che permette di creare degli analizzatori lessicali. Prende in input un file .l che contiene un insieme di definizioni regolari e una serie di azioni corrispondenti, e da in output un programma C che realizza l'automa riconoscitore e che associa ad ogni istanza di una definizione la relativa azione.** (file con estensione .yy.c)

53) Un file .l è diviso in 3 parti:

1. Le dichiarazioni, che sono delle definizioni regolari.
2. Le regole, che sono un insieme di coppie espressione regolare-azione
3. Le funzioni ausiliarie, es. funzioni complesse per le parti azione (chiamate nella parte azione quindi).

L'analizzatore lessicale consegnato in output implementa il DFA riconoscitore delle espressioni regolari contenuti nella parte "regole".
Dunque, scandisce il testo per vedere se una parte del testo corrisponde ad una delle espr. Regolari della zona regole, e quando lo riconosce esegue l'azione specificata per tale lessema.
Quando l'input non corrisponde a nessun pattern, allora lo lascia inalterato e segnala il fatto al gestore degli errori.

54) Il programma generato da Lex viene usato come una subroutine per YACC, che è un generatore di analizzatori sintattici. Collaborano in questo modo:

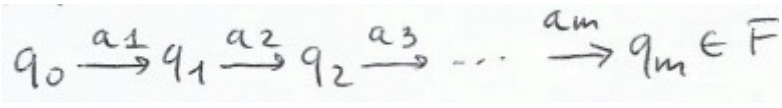
- lex.yy.c (ovvero l'analizzatore lessicale) viene usato da YACC on demand, e richiede il token successivo su richiesta.
- Yylex() restituisce il nome del token, mentre il valore del token è condiviso nella variabile yyval (che è una variabile globale).

**55) Intestazione e dimostrazione del Pumping Lemma:**
**Il pumping lemma dice che, se L è un linguaggio regolare, allora $\exists N>0$ tale per cui $\forall z \in L$, con $|z| \geq N$, $\exists u,v,w$ tale che**

- **$z = uvw$**
- **$|uv| \leq N$**
- **$|v| \geq 1$**
- **$\forall k \geq 0$ $uv^kw \in L$**

**(Inoltre N è minore o uguale del numero di stati del DFA minimo che accetta L.)**

**Dimostrazione:**
**Sia $N = |Q_M|$ , dove M è il DFA minimo che accetta L. Sia $z = a1a2...am \in L$ con $m \geq N$.**
**Quindi,**



**Il percorso da 0 a m è dato da m+1 stati, con m+1 > N (per le proposizioni precedenti).**
**Quindi, ∃i,j tali che (i ≠ j) e tali per cui qi = qj (e quindi, c'è un ciclo!).**

**Il percorso da qi a qj è pari a v, ed è |v|≥1, siccome i≠j.**
**La parte prima di questa serie di stati sarà u, mentre la parte dopo sarà w.**

**La condizione |uv| ≤ 1 ci dice che dobbiamo prendere il primo ciclo.**
**∀k≥0 siccome il ciclo può essere percorso un numero arbitrario di volte (anche 0).**

**56) Possiamo dimostrare che un linguaggio non è regolare attraverso la negazione del Pumping Lemma. Ovvero:**
**Se ∀N>0, ∃z ∈ L con |z|≥N, allora ∀uvw se**

- **z = uvw**
- **|uv| ≤ N**
- **|v| ≥ 1**

**Allora $uv^kw \notin L$, allora L non è regolare.**

**57) Le proprietà di chiusura dei linguaggi regolari sono:**

- **Unione**
- **Concatenazione**
- **stella di Kleene**
- **complementazione**
- **intersezione**

**Ovvero, per ciascuna di queste operazioni fra 2 linguaggi regolari, ottengo un linguaggio regolare.**

**58) L'analisi sintattica è quella fase della compilazione in cui una lista di token (prodotto dall'analizzatore lessicale) viene trasformata da un parser in un albero di derivazione relativo alla grammatica usata per costruire il parser.**

**59) Per i linguaggi liberi (che sono una classe più generale dei linguaggi regolari), fare uso di dei NFA o DFA non basta, e bisogna quindi fare uso di degli automi a pila (PDA).**
**Un automa a pila Non deterministico (PDA) è una 7upla (Σ, Q, Γ, δ, q0, ⊥, F) dove:**

- **Σ è un alfabeto finito**
- **Q è l'insieme degli stati**
- **Γ è l'insieme dei simboli della pila**
- **δ è la funzione di tranzione, che prende 3 input.**
  - **δ: Q x (Σ ∪ {ε}) x Γ**
- **q0 è lo stato iniziale**
- **⊥ ∈ Γ è il simbolo iniziale sulla pila**
- **F ⊆ Q è l'insieme degli stati finali**

Una configurazione è una tripla (q, w, β).
Il resto si trova a pag 7 delle slides 10.

**60) Possiamo riconoscere un linugaggio con un PDA in due modi diversi:**

- **Per stato finale, ovvero riconosco il linguaggio solo se raggiungo lo stato finale**
  **$L[N] = \{w \in \Sigma^* \mid (q0, w, \perp) \vdash^* (q, \varepsilon, \alpha) \text{ con } q \in F\}$**
- **Per pila vuota, ovvero riconosco un linguaggio solamente se la pila è vuota:**
  **$P[N] = \{w \in \Sigma^* \mid (q0, w, \perp) \vdash^* (q, \varepsilon, \varepsilon) \text{ con } q \in F\}$**

**Per un PDA, normalmente, L[N] ≠ P[N] (**ovvero il linguaggio riconosciuto per pila vuota non coincide con il linguaggio riconosciuto per stato finale**).**
La classe dei linguaggi riconosciuti dal PDA però non cambia: infatti posso costruire un N' che conosce per stato finale a partire da un N che riconosce per pila vuota (e viceversa).
[Rispettivamente: Puntando gli stati che riconoscono per pila vuota a un singolo stato finale f, e puntando gli stati finali a uno stato d'errore f con un cappio per pila vuota]

61) Come ottenere un PDA da una grammatica libera?
Supponiamo di avere una produzione in stile S → aSb | ε .
Allora, un possibile metodo sarebbe creare un PDA che riconosce per pila vuota. S è il simbolo iniziale della pila.
Il nostro automa avrà un solo stato. Ogni arco sarà un cappio, e sarà del tipo (a, a/ε) per i terminali e per il terminale con produzione epsilon. Altrimenti, per ogni produzione A→ α, si avrà un arco (ε, A/aAa).

Un linguaggio L è libero se e solo se è riconoscuito/accettato da un PDA.

**62) Le proprietà di chiusura per un linguaggio libero sono:**
- **Intersezione**
- **concatenazione**
- **Ripetizione (stella di Kleene)**

**L'intersezione di un linguaggio libero con un linguaggio regolare è un linguaggio libero.**

**63) Il pumping theorem dice che, se L è un linguaggio regolare, allora ∃N>0 tale per cui ∀z ∈ L, con |z|≥N, ∃u,v,w,x,y tale che**
- **z = uvwxy**
- **|vwx| ≤ N**
- **|vx| ≥ 1**
- **∀k ≥ 0 $uv^kwx^ky$ ∈ L**

**Dimostrazione:**
**Sia b il massimo fattore di ramificazione in un albero di derivazione (ovvero il masssimo numero di simboli che compaiono nella parte destra di una produzione in R), ovvero b = max {|α| | A→α ∈ R}.**

**Un albero ha altezza h (con 0 a livello della radice) e fattore di ramificazione b, ha al massimo $b^h$ foglie. Fissiamo N = $b^{|NT|+1}$, allora per ogni albero di derivazione per la stringa z, con |z|≥N, deve avere altezza almeno |NT|+1.**

**Prendiamo un qualunque z ∈ L, con |z|≥N. Consideriamo il suo albero di derivazione (se ne possiede più di uno, allora G è ambigua, e allora prendiamo quello con il minor numero di nodi).**

**Dunque, considerando che |z|≥N, avremo che:**
- **avremo un albero di derivazione con altezza maggiore di |NT|+1.**
- **Esiste un cammino dalla radice S ad una foglia con almeno |NT|+2 nodi.**
- **Quel cammino attraversa |NT|+1 nodi interni etichettati con un nonTerminale (la foglia è etichettata da un terminale).**
- **Almeno un nonTerminale si ripete in quel cammino.**

**Allora S =>* z (ovvero la derivazione da S a z) può essere divisa come**

**S =>* uAy =>* uvAxy =>* uvwxy**

Questa parte della derivazione può essere ripetuta più volte.

**Quindi**
**S =>* uAy =>* uwy con k = 0**
**S =>* uAy =>* uvAxy =>* uvwxy con k = 1**
**S =>* uAy =>* uvAxy =>* $uv^2Ax^2y$ ... con k = 2**

**Devo verificare i vincoli.**
- **|vx| ≥ 1**
  - **se sia v che x fossero epsilon, allora l'albero per k=0 genererebbe ancora z ed avrebbe meno nodi, contraddicendo l'ipotesi di aver scelto il più piccolo albero.**
- **|vwx| ≤ N**
  - **il cammino da A alla foglia è di lunghezza ≤ |NT|+1 (cioè usa |NT|+2 nodi al massimo, di cui uno è il terminale foglia) => la A in alto non può generare parole più lunghe di $b^{|NT|+1}$ = N.**

**64) Possiamo usare il pumping theorem all'inverso!**
**$\forall N>0$, $\exists z \in L$ con $|z| \geq N$, tale che $\forall u,v,w,x,y$ e**

- **$z = uvwxy$**
- **$|vwx| \leq N$**
- **$|vx| \geq 1$**

**E detto ciò abbiamo che $\forall k \geq 0$ $uv^kwx^ky \notin L$ , allora L non è libero!**

**65) Oltre ai linguaggi liberi, abbiamo le classificazioni di Chomsky, secondo cui abbiamo:**

1. **grammatiche regolari - NFA/DFA**
2. **grammatiche libere dal contesto - PDA**
3. **grammatiche dipendenti dal contesto – Automi Limitati** (misto di terminali e non terminali in testa e coda delle produzioni, e parte di queste stringhe di NT ∪ T si trovano sia a destra che a sinistra della produzione, proprio lo stesso identico, e almeno un nonTerminale a destra).
4. **grammatiche monotone – Automi di Turing** (abbiamo di tutto, ma la testa deve essere di lunghezza minore o uguale della coda).
5. **grammatiche generali** (abbiamo di tutto).

Per capire per bene ste cagate, vai a pag. 30 del pacco 11 delle slides.

66) **Il DPDA è una automa a pila deterministico. Un PDA $N = (\Sigma, Q, \Gamma, \delta, q0, \perp, F)$ è deterministico se e solo se:**

- **Per ogni stato q e per ogni simbolo z della pila, se $\delta(q, \varepsilon, z) \neq \varnothing$, allora $\delta(q, a, z) = \varnothing$ per ogni simbolo *a* dell'input, ovvero se ho una mossa epsilon per quello stato q, allora non potrò avere altre mosse per qualsiasi altro simbolo.**
- **Per ogni stato q e per ogni simbolo z della pila, allora la lunghezza della transizione è di massimo 1, per ogni simbolo dell'alfabeto Σ ed epsilon.**
  **(ovvero $|\delta(q, a, z)| \leq 1$ $\forall a \in \Sigma \cup \{\varepsilon\}$)**

**Un linguaggio è libero deterministico se è accettato per stato finale da un DPDA.**

67) La classe dei linguaggi liberi deterministici è strettamente inclusa in quella dei linguaggi liberi, e contiene strettamente la classe dei linguaggi regolari.

**68) La prefix property ci dice che non esistono due stringhe x e y appartenenti al linguaggio tali che tali che x è prefisso di y.**
**es. $L=\{wcw^R \mid w \in \{a,b\}^*\}$ gode della prefix property.**

**La prefix property è utile siccome:**
**"Un linguaggio libero deterministico L è riconosciuto da un DPDA per pila vuota se e solo se gode della prefix property".**

69) Si, se si aggiunge l'end marker $ allora $L\$ = \{w\$ \mid w \in L\}$ gode della prefix property.
"Questo si può forzare" aggiungendo uno stato finale con transizione $, ovvero uno stato finale per pila vuota, e aggiungendo un arco ($, z/ε) per ciascuno stato finale.

70) No, i linguaggi liberi deterministici NON sono ambigui, in quanto generabili da una grammatica libera NON ambigua.

71) I linguaggi liberi deterministici sono:

- Chiusi per complementazione, ovvero se esiste un DPDA N tale che L = L[N], allora esiste un DPDA N' tale che L' = L[N'] e L' = Σ* \ L (ovvero è complementare).
- Non sono chiusi per l'intersezione.
- Non sono chiusi per l'unione.

**72) Per costruire un parser, si parte da una grammatica libera che viene data ad un algoritmo di costruzione (es. YACC) che crea un Parser, ovvero un DPDA con un output.**

**73) Un parser (come già visto mille volte) prende in input una lista di token e ritorna un albero di derivazione.**

74) I parser possono essere:

- Non deterministici: se durante la ricerca di una derivazione, si scopre che una scelta è improduttiva e non porta a riconoscere l'input, e allora il parser dovrà fare backtracking, disfando parte della derivazione costruita, e scegliendo un'altra produzione.
- Deterministici: leggono l'input una volta sola, e la loro scelta è quindi definitiva.

75)?

**76)** Possiamo cotruire i parser:

- **Top-down**: **Ricostruiscono una derivazione leftmost per una stringa w, per a partire dal simbolo iniziale S (che si trova all'inizio sulla pila).**
- **Bottom-up**: **Ricostruiscono una derivazione rightmost (quindi a rovescio), partendo da una stringa w e cercando di ridurla al simbolo iniziale S (che si trova alla fine sulla pila).**

77)
Per il top down parsing, NON sono adatte le grammatiche LR(k) **e con grammatiche aventi ticorsione sinistra**, mentre per il bottom-up parsing, NON sono adatte le LL(k) e **con grammatiche aventi produzioni epsilon**.

78) Una produzione epsilon è una produzione stile $A \to \varepsilon$, e un simbolo annullabile è un NonTerminale A tale per cui $A =>^* \varepsilon$, ovvero è derivabile in epsilon.

**79) Per creare una grammatica senza produzioni ε, a partire da una grammatica con produzioni ε, si calcola prima l'insieme dei simboli annullabili in modo induttivo.**
**Poi, per ogni produzione in cui occorrono i simboli annullabili, procediamo per casi e consideriamo i casi in cui ogni simbolo viene annullato (uno alla volta e poi gradualmente tutti assieme).**
**Ciascuna produzione che otteniamo in ciascuno dei casi verrà aggiunta alle produzioni.**

80) Le produzioni unitarie sono tutte quelle produzioni del tipo $A \to B$ con $A,B \in NT$.
Le coppie unitarie sono tutte quelle coppie (A, B) tale per cui $A =>^* B$ (ovvero A è derivabile in B) usando solo produzioni unitarie.

**81) Per creare una grammatica senza produzioni unitarie:**

1. **Calcolo l'insieme delle coppie unitarie in modo induttivo:**

$$U_0(G) = \{ (A,A) \mid A \in NT \}$$

$$U_{i+1}(G) = \{ (A,C) \mid \text{con } (A, B) \in U_i(G) \text{ e } B \to C \in R \}$$

2. **Elimino le produzioni unitarie nella mia grammatica.**
3. **Per ogni coppia (A,B) presente in U(G), aggiungo tutte lo produzioni di B in A.**

82) Un simbolo $X \in \{T \cup NT\}$ è:

- Un generatore, se e solo se esiste una stringa di terminali w con $X =>^* w$, ovvero esiste una derivazione da X a w.
- Raggiungibile, se e solo se esiste una derivazione da S a una stringa di terminali e nonTerminali contenente X (quindi $S=>^* aXb$ con $a,b \in (T \cup NT)^*$).
- Un simbolo è utile se e solo se è sia generatore che raggiungibile, quindi se X compare in almeno una derivazione di una stringa $z \in L(G)$.

83) I generatori si calcolano in questo modo (sempre in modo induttivo):

- $G_0(G) = T$
- $G_{i+1}(G) = G_i(G) \cup \{ B \in NT \mid B \to c1...ck \in R \text{ e } c1...ck \in G_i(G) \}$

I raggiungibili si calcolano in questo modo:

- $R_0(G) = \{S\}$
- $R_{i+1}(G) = R_i(G) \cup \bigcup_{B \in Ri(G),\, B \to x1..xk \in R} \{x1...xk\}$

**84) I simboli inutili in una grammatica si eliminano in questo ordine (che è importante):**

1. **Prima si elimano i simboli non-generatori e le produzioni che contengono almeno uno di questi simboli.**
2. **Poi si eliminano i simnoli non-raggiunbili e le produzioni che usano almeno uno di questi simboli.**

**85) Una produzione è ricorsiva sinistra quando ha una struttura $A \rightarrow A\alpha$.**
**Una grammatica si dice ricorsiva sinistra quando ha almeno un terminale tale per cui $A =>^* A\alpha$.**

**85_2)**

- **Per rimuovere la ricorsione sinistra immediata:**
  **Dobbiamo avere una produzione stile $A \rightarrow A\alpha 1 \mid \ldots \mid A\alpha k \mid \beta 1 \mid \ldots \mid \beta k$.**
  **La sostituiamo con le due produzioni:**
  **$A \rightarrow \beta 1A' \mid \ldots \mid \beta kA'$**
  **$A' \rightarrow \alpha 1A' \mid \ldots \mid \alpha kA' \mid \varepsilon$**
- Per rimuovere la ricorsione sinistra non immediata:
  Ordiniamo i NT in modo {A1,…,An}
  for i, 1...n
  {
  for j, 1...i-1
  {
  Se la produzione è di forma $Ai \rightarrow Aj\alpha$, allora sostituiamo la produzione con
  $Ai \rightarrow \beta 1\alpha \mid \ldots \mid \beta k\alpha$ dove era $Aj \rightarrow \beta 1 \mid \ldots \mid \beta k$
  }
  }

**La rimozione della ricorsione sinistra serve per rendere la grammatica adatta al top-down parsing (LL(1))**

**86) Fattorizzare a sinistra permette la compatibilità col top-down parsing (LL(1)), in particolare per ridurre il non determinismo.**
**Per fattorizzare una grammatica, raccolgo le parti che sono in comune di due produzioni con la stessa testa, e introduco un nuovo nonTerminale per rappresentare la parte non in comune (che è anche la parte rimanente).**

87) Un parser a discesa ricorsiva è un parser top down non deterministico, che usa implicitamente una pila per gestire le chiamate ricorsive (in un parser top down non ricorsivo, la pila è esplicita).
In pratica controlla se il simbolo è un terminale o un non terminale, e nel caso il simbolo non terminale combaci, allora è tutto ok, altrimenti si fa backtracking.
(Testa ogni produzione sull'input, e se non combacia fa backtracking cambiando produzione).

**88) Definizione First e Follow:**

- **First: data una stringa $\alpha$ di terminali e nonterminali, diciamo che il First($\alpha$) è l'insieme dei TERMINALI che possono stare in prima posizione in una stringa che si deriva da $\alpha$.**
  **In modo simbolico,**
  - **$a \in T$, $a \in$ First($\alpha$) se e solo se $\alpha =>^* a\beta$ con $\beta \in (T \cup NT)^*$**
  - **inoltre, se $\alpha =>^* \varepsilon$, allora $\varepsilon \in$ First($\alpha$)**

- **Follow: data una grammatica libera G e $A \in NT$, diciamo che il Follow(A) è l'insieme dei TERMINALI che possono comparire immediatamente a destra di A in una forma sentenziale.**
  **In modo simbolico,**
  - **per $a \in T$, $a \in$ Follow(A) se e solo se $S =>^* \alpha Aa\beta$ con $\alpha,\beta \in (T \cup NT)^*$**
  - **$\$ \in$ Follow(A) se $S =>^* \alpha A$**

**89) Algoritmo per calcolare il First:**
**Per ogni x ∈ T, First(x) = {x}**
**Per ogni X ∈ NT, inizializziamo First(X) = ∅**
**Ripeti finché nessun First(X) non viene più modificato:**
**Per ogni produzione X → Y1...Yk**
**Per ogni i da 1 a k**
**se (Y1,…,Yi-1 ∈ N(G))**
**allora First(X):=First(X) ∪ (First(Yi)\{ε})**
**Per ogni X ∈ N(G)First(X):=First(X) ∪ {ε}**

Insieme dei simboli annullabili

**Algoritmo per calcolare il Follow:**
**Follow(X):={$}**
**Per ogni X ∈ NT, inizializziamo Follow(X) := ∅**
**Ripeti finché nessun Follow(X) non viene più modificato:**
**Per ogni produzione X → αYβ**
**Follow(Y):=Follow(Y) ∪ First(β)\{ε}**
**Per ogni produzione X → αY e per ogni produzione X → αYβ con ε ∈ First(β)**
**Follow(Y):=Follow(Y) ∪ Follow(X)**

Aggiungo I follow di X a Y

**90) Una tabella di Parsing LL(1) ha**
- **Per righe, i NonTerminali**
- **Per colonne, i terminali**
- **Ogni casella M[A, a] contiene le produzioni che possono essere scelte dal parser mentre tenta di espandere A e l'input corrente a (solo se ha al massimo una produzione per casella è deterministico).**

**Per riempire una tabella di parsing LL(1), dobbiamo tenere a mente le seguenti regole:**
**Per ogni produzione A → α**
- **per ogni a ∈ T tale che a ∈ First(α), allora inseriamo la produzione nella casella M[A, a]**
- **se ε ∈ First(α), allora inseriamo la produzione in tuelle caselle M[A, x] con x ∈ Follow(A).**

**91) Una grammatica è di classe LL(1) se e solo se ogni casella della tabella di parsing contiene al massimo una produzione.**

**G è una grammatica LL(1) se e solo se, per ogni coppia di produzioni distinte avente stessa testa:**
**A → α | β**

1. **First(α) ∩ First(β) = ∅**
2. **a) se ε ∈ First(α), allora First(β) ∩ Follow(A) =∅**
   **b) se ε ∈ First(β), allora First(α) ∩ Follow(A) =∅**

**92) Questo parser si chiama LL(1) perché ha un solo simbolo di look-ahead, legge l'input left-to-right e ha una derivazione left-most.**

93) Si sostituisce ogni NT con la derivazione di una delle produzioni, finché il carattere dell'input non combacia con il First() della derivazione sulla pila. A questo punto, si "mangia" (pop) il carattere e si continua con la derivazione, finché non si raggiunge il simbolo di end of input ($) e non si ha **ε** sulla pila.

**94) Ogni linguaggio regolare è generabile da una grammatica di classe LL(1).**

95)
- $w \in First_k(\alpha)$ se e solo se $\alpha =>^* w\beta$, con $|w| = k$, $w \in T^*$ e $\beta \in (T \cup NT)^*$, oppure se $\alpha =>^* w\beta$, con $|w| \leq k$, $w \in T^*$
- $w \in Follow_k(\alpha)$ se e solo se $S =>^* \alpha A w\beta$, con $|w| = k$, $w \in T^*$, oppure se $S =>^* \alpha A w$, con $|w| \leq k$, $w \in T^*$

96) Un linguaggio L è di classe LL(k) se esiste G di classe LL(k) tale che L=L(G).

97) Una grammatica ricorsiva sinistra non è LL(k) per nessun k, stessa cosa per le grammatiche ambigue.

98) Si, esistono L liberi deterministici tali che NON esiste G di classe LL(k) per nessun k.
es. L = $\{a^i b^j \mid i \geq j\}$

**99) Un parser bottom-up (o shift-reduce) è un PDA con un solo stato che riconosce per pila vuota un linguaggio L$.**
**Prende in input una grammatica libera G con un simbolo iniziale S e una stringa w ∈ T*.**
**In output, invece, se la stringa appartiene al linguaggio, ovvero w ∈ L(G), allora restituisce la sua derivazione rightmost a rovescio.**
**Sono chiamati parser LR perché leggono l'input left-to-right e ritornato una derivazione rightmost.**

100) Si possono verifare dei conflitti shift-reduce e reduce-shift. Per risolvere i conflitti bisogna scegliere l'azione giusta in modo che sulla pila ci sia un prefisso viabile.

**101) Un prefisso viabile è una sequenza di terminali e nonTerminali che può apparire sulla pila di un parser bottom-up per una computazione che accetta un input (in pratica la parte top della pila deve essere un prefisso di una parte dx di una produzione).**
**In termini di una grammatica libera: una stringa w ∈ (NT ∪ T)* è un prefisso viabile per G se e solo se esiste una derivazione rightmost (es. S=>*Say=>Sαβy=wβy) per una qualche stringa di terminali, stringa di terminali e non terminali e per una produzione A → αβ.**

**102) Un item LR(0) è una produzione con indicata, con un punto, una posizione della sua parte destra. Questo punto indica quale parte della produzione è già stata analizzata.** Gli item di una grammatica si ottengono esaminando lettera per lettera una produzione.

**103)** (Risposta alla prima parte a pag.14 pacco 15). **Attraverso le funzioni di Clos e di Goto partendo dall'NFA dei prefissi viabili.**

**104) Una tabella di parsing LR(0) ha per righe l'indice degli stati dell'automa canonico, e per colonne l'insieme dei terminali + {$} e non terminali.**
**Si riempie in questo modo:**
**Per ogni stato s dell'automa canonico LR(0):**

- **se x è un terminale e ho una transizione x da s a uno stato t nell'automa LR(0), inserisce il comando shift t in posizione M[s, x]**
- **se A → α. ∈ s e A≠S' , allora inserisci reduce A → α in M[s, x] per tutti gli x ∈ T ∪ {$}**
- **se S' → S. ∈ s, allora inserisci Accept in M[s, $]**
- **se A ∈ NT e c'è una transizione A da s a t nell'automa LR(0), allora inserisci Goto t in M[s, A]**

**Una grammatica è di classe LR(0) solo se ogni casella della tabella di parsing LR(0) contiene al massimo un elemento.**

**105) Il parser è formato da 2 stack (uno degli stati e uno dei simboli).**

106) Una grammatica libera può non essere di classe LR(0), un esempio può essere S → a | ab, che ha un conflitto shift-reduce.
In generale, una grammatica è LR(0) quando L è libero deterministico e gode della prefix property.

**107) Una tabella di parsing SLR(1) si riempe allo stesso modo di una tabella LR(0), con l'unica differenza che:**

- **se A → α. ∈ s e A≠S' , allora inserisci reduce A → α in M[s, x] per TUTTI gli x ∈ Follow(A).**

**La S sta per simple, e l'1 è perché abbiamo un simbolo di look ahead (anche se non è un simbolo di lookhead esplicito, ma è perché fa lookahead grazie al follow dei NT).**

108) Esistono grammatiche non di classe SLR(1), siccome potremmo comunque avere un conflitto shift-reduce.

**109) Un item LR(1) è una coppia formata da un item LR(0) e un simbolo di look-ahead in T ∪ {$}.**
L'NFA LR(1) si costruisce in questo modo:

- [S' → .S, $] è lo stato iniziale
- dallo stato [A → α.Xβ, a] c'è una transizione X allo stato [A → αX.β, a] per ogni X ∈ T ∪ NT.
- dallo stato [A → α.Xβ, a], per X ∈ NT e per ogni produzione X → y, c'è una transizione ε verso lo stato [X → .y, b] **per ogni b ∈ First(βa)**.

**L'automa canonico si costruisce attraverso il clos() e goto() partendo dallo stato iniziale clos([S' → .S, $]).**

**110) La tabella di parsing LR(1) ha la stessa formazione della tabella LR(0) e SLR(1), e si costruisce in questo modo:**
**Per ogni stato s dell'automa canonico LR(1):**

- **se x è un terminale e ho una transizione x da s a uno stato t nell'automa LR(1), inserisce il comando shift t in posizione M[s, x]**
- **se [A → α., x] ∈ s e A≠S' , allora inserisci reduce A → α in M[s, x] solo per l'x del look ahead!**
- **se [S' → S., $] ∈ s, allora inserisci Accept in M[s, $]**
- **se A ∈ NT e c'è una transizione A da s a t nell'automa LR(1), allora inserisci Goto t in M[s, A]**

**111) La tabella di parsing LALR(1) si ottiene a partire dalla LR(1) fondendo assieme gli stati con lo stesso nucleo, dove un nucleo di stato LR(1) è l'insieme degli item LR(0) ottenuto dimenticando i look ahead degli item LR(1).** In questo modo abbiamo tante righe quanti gli stati dell'automa LR(0), e meno reduce di una tabella SLR(1).
Per gli stati duplici, bisogna inserire nella pila degli stati entrambi gli indici (es. se lo stato 4 si è fuso con lo stato 7, allora devo mettere S47 per fare uno shift, e inserire i due stati come se fossero un singolo stato.)

Un grammatica è LARL(1) quando la sua tabella di parsing si ottiene a partire da quella di LR(1) senza che vi siano conflitti.

112) Esistono grammatiche che sono LR(1) ma non LALR(1), in quanto possono formarsi dei conflitti reduce-reduce nella tabella di parsing LALR(1).
Un esempio è la grammatica:
S → aAa | bAb | aBb | bBa
A → c
B → c

113) Una grammatica LR(k) ha:

- un item formato da [item LR(0), β] con |β| ≤ k
- le colonne della tabella espresse su stringhe $T^k$ (ovvero di k terminali).
- Quando un [A → α.Xy, β] ∈ s (ovvero a uno stato dell'automa canonico LR(k)), allora si aggiunge nel DFA l'item [X → .δ, w] se X → δ è una produzione e w ∈ $First_k(y\beta)$

Se la tabella di parsing LR(k) non presenta conflitti, allora la grammatica G è LR(k).

Una grammatica SLR(k) si ottiene partendo dall'automa canonico LR(0) e si riempie la tabella di parsing SLR(k) secondo la legge

- se A → α. ∈ s e A≠S' , allora inserisci reduce A → α in M[s, w] per tutti i w ∈ $Follow_k(A)$

Per creare una grammatica LALR(k), si parte dall'automa canonico LR(k) e si fondono gli stati con lo stesso nucleo. Se la tabella risultante non presenta conflitti, allora la grammatica è LALR(k).

**114)**
**SLR(k), LALR(k) e LR(k) si relazionano, al variare di k, in questo modo:**

- **SLR(k) ⊂ LALR(k) ⊂ LR(k) per ogni k ≥ 1**
- **SLR(1) ⊂ SLR(2)⊂ SLR(3) ⊂ … ⊂ SLR(k)**
- **LALR(1) ⊂ LALR(2)⊂ LALR(3) ⊂ … ⊂ LALR(k)**
- **LR(0)⊂ LR(1) ⊂ LR(2) ⊂ … ⊂ SLR(k)**

**Curiosamente, la relazione fra grammatiche LL(k) e LR(k) è questa:**

- **LL(k) $\subset$ LR(k) per ogni k≥0**
- **LL(k) $\not\subset$ LR(k-1) per ogni k≥1**

**Se una grammatica G è LL(k) o LR(k), allora G non è ambigua e L(G) è deterministico.**

115) I linguaggi generati da grammatiche LL(k) sono strettamente contenuti nei linguaggi generati da grammatiche SLR(1). Inoltre, un linguaggio è libero deterministico se e solo se è generato da una grammatica SLR(1) (oppure se e solo se è generato da una grammatica LR(k) per qualche k≥0). Quindi, esistono linguaggi liberi deterministici che sono SLR(1), ma non LL(k).

Un esempio di linguaggio libero deterministico (quindi SLR(1)) ma non LL(k) è $L=\{a^ib^j \mid i \geq j \geq 0\}$

I linguaggi LL(0) sono inoltre strettamente contenuti dentro LR(0).

**116) ?**

**117) Si.**

**118) YACC è un generatore di analizzatori sintattici. Questo prende in input un file .y che contiene la descrizione della grammatica libera, e da in output un programma C dal nome y.tab.c, che realizza il parser LALR(1)** per costruzione diretta (non fatta a lezione). Quando questo file viene compilato, si ottiene un eseguibile a.out.

L'analizzatore lessicale Lex è usato da YACC come una subroutine, invocandolo con yylex() per richiedere il token successivo su richiesta. Ci sono poi alcune variabili comuni (es. yyval) che permettono di scambiare delle informazioni.

**119) Un file YACC è costituito da:**

- **Un prologo,** una parte opzionale che contiene le definizioni di macro e di altre dichiarazioni di variabili o funzioni.
- **Un area definizioni,** che contiene delle dichiarazioni di simboli usati nella descrizione della grammatica (es. nomi dei token, simboli condivisi con Lex..). In questa sezione è inoltre possibile dichiarare la precedenza e l'associatività di alcuni terminali/operazioni.
- **Un area regole,** che contiene le varie produzioni espresse come nonterminale: corpo1 {azione semantrica}. L'azione semantica è codice C eseguito al momento in cui il parser riduce mediante quella produzione: in generale, calcola il "valore semantico" della testa della produzione in funzione dei valori semantici dei simboli che ne compongono il corpo. Potrebbe essere l'albero di derivazione oppure codice intermedio.
- **Una zona delle funzioni ausiliarie.** Contiene le funzioni di supporto per la generazione del parser, tra questi yylex(), che invoca l'analizzatore lessicale e restituisce il nome del token.

120) Attraverso le informazioni su associatività e precedenza degli operatori, è possibile risolvere tutti i conflitti. In ogni caso, in assenza di indicazioni, risolve i conflitti shift-reduce in favore dello shift, e conflitti di tipo reduce-reduce in favore della prima produzione elencata.

121) Per rispondere a questa domanda, dobbiamo consinderare in particolare in caso in cui l'errore sta nella nonTerminazione del programma. Per questo, costruiamo una funzione H(P, x) che ritorna 1 se P(x) termina, e 0 altrimenti. Ma non sappiamo se si può costruire un tale programma.

**122) Il problema dell'halting problem sta nel chiedersi se la funzione H esista veramente.**
**Possiamo dimostrare che il problema non può essere risolto:**
**Costruiamo una applicazione/funzione K(P) tale che K(P) = 1 se P(P) converge (↓) e K(P) = 0 se P(P) diverge (↑). In poche parole, K(P) = H(P, P). Se esiste H, quindi, esiste anche K.**

**Se esiste K, allora possiamo scrivere un programma G che prende in input un programma P e calcola**

**G(P) = 1 se K(P) = 0 (se il programma diverge), e G(P) diverge se K(P) = 1 (quindi se il programma converge).**

**Dunque, se K esiste, allora anche G è facilemente calcolabile!**
**Tuttavia, abbiamo che se diamo input G a G, otteniamo che**

- **G(G) = 1 se e solo se K(G) = 0 e quindi se e solo G(G) diverge (↑)**
- **G(G) diverge (↑) se e solo se K(G) = 1 e quindi se e solo se G(G) converge (↓)Q**

**Questo però è un assurdo! G quindi non può esistere =>K non può esistere => H non può esistere!**
**H è quindi il primo esempio di funzione non calcolabile.**

123) Un problema è decidibile quando esiste una procedura che può risolverla.
Più formalmente, una procedura di decisione è una procedura che funziona per argomenti arbitrari, e risponde SI o NO in tempo finito (una procedura di semi-decisione non può rispondere NO in tempo finito).

**124) Alcuni tipi esempi di proprietà indecibili per i linguaggi di programmazione sono:**

- **Terminazione**
- **Divergenza**
- **Equivalenza fra programmi**
- **Calcolo di una funzione costante**
- **Generazione di errori a run-time**

125) Una macchina di Turing è una 6-upla (Q, A, B, δ, q0, qf) dove

- Q è un insieme finito di stati
- A è l'alfabeto finito dell'input
- B è l'alfabeto finito del nastro (A ⊂ B)
- q0 è lo stato iniziale
- qf lo stato finale
- δ: Q x B -o-> Q x B x {dx, sx}

Le mosse della macchina di Turing sono:

- αqsβ ⊢ αs'q'β se δ(q, s) = (q', s', dx)
- αpqsβ ⊢ αqps'β se δ(q, s) = (q', s', sx)

(p è la casella a fianco a quella di s)
In pratica si ha una punta che può muoversi a destra e a sinistra su un nastro infinito.
La macchina di Turing può non raggiungere né uno stato d'errore, né qf (lo stato finale).

In generale, una macchina di Turing deterministica può essere vista come una macchina che calcola funzioni parziali binarie.

**126) Un linguaggio è Turing-completo quando ha la stessa potenza espressiva delle MdT, e quindi calcola le stesse funzioni calcolabili con MdT. Il linguaggio usato per spiegare la semantica SOS è Turing completo.**
**La tesi di Church-Turing afferma che se una funzione può essere calcolata algoritmicamente in un qualche formalismo, allora è calcolabile con una Macchina Di Turing.**
**Questa non è dimostrabile siccome non abbiamo una definizione formale di "algoritmicamente calcolabile" e siccome fa una quantificazione universale su tutti i possibili formalismi.**

127) I normali linguaggi di programmazione sequenziali sono Turing-completi, e se due linguaggi sono turing-completi, allora sono equamente espressivi.
Nel caso del linguaggi concorrenti, però, la situazione è diversa.

Secondo la tesi di Jacopini-Bohm, ogni un linguaggio è Turing completo se contiene istruzioni:

- condizionali
- iterazione indeterminata
- composizione sequenziale

e istruzione di assegnamento.

128)

- Le macchine di Turing esprimono grammatiche generali, e la proprietà $w \in L(M)$ è solo semidecidibile
- I PDA esprimono il formalismo delle grammatiche libere, $w \in L[N]$ è decibile, mentre $L(G1)=L(G2)$ è indecidibile.
- I DFA esprimono il formalismo delle grammatiche regolari, $L(G1)=L(G2)$ è decidibile.