Capitolo 1 – Macchine astratte, interpreti, compilatori

1. Che cosa è una macchina astratta? E in cosa si differenzia da una macchina fisica?
Dato un linguaggio di programmazione L, una macchina astratta M_L è un insieme di strutture dati e algoritmi per memorizzare ed eseguire programmi scritti in L.
La macchina fisica è una specifica implementazione della macchina astratta, fatta di circuiti logici e dispositivi elettronici.

2. Che cos'è un interprete? In cosa consiste il ciclo fetch--decode--execute?
Un interprete per il linguaggio L, scritto nel linguaggio L_0, è un programma che realizza una funzione parziale (ovvero che può essere indefinita per qualche valore del dominio)

$$\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_0} : (\mathcal{P}rog^{\mathcal{L}} \times \mathcal{D}) \to \mathcal{D} \quad tale\ che \quad \mathcal{I}_{\mathcal{L}}^{\mathcal{L}_0}(\mathcal{P}^{\mathcal{L}}, Input) = \mathcal{P}^{\mathcal{L}}(Input).$$

L'interprete è il componente di una macchina astratta che serve per eseguire istruzioni. In particolare contiene sistemi di gestione della memoria, controllo dei dati e controllo di flusso.
Il ciclo FDE è il ciclo eseguito da un interprete che consiste nel prelevare, decodificare ed infine eseguire un'istruzione.

3. Cos'è il linguaggio macchina?
Data una m.a. A, il linguaggio compreso dal suo interprete (L) è il suo linguaggio macchina.

4. Possono esistere macchine diverse con lo stesso linguaggio macchina?
Dato che il linguaggio macchina è il linguaggio compreso dall'interprete si perchè dipende dal modo in cui l'interprete è implementato e dalle strutture dati che utilizza.
Nel caso della macchina hardware per esempio molti processori implementano in modo diverso uno stesso ISA.
5. In quali modi è possibile implementare una macchina astratta? Elencare vantaggi e svantaggi delle varie tecniche.
Hardware, software e firmware.

Realizzazione in **Hardware**:
Concettualmente semplice, permette estrema velocità di esecuzione **MA** è difficile implementare costrutti d'alto livello inoltre è poco flessibile quindi una modifica al linguaggio comporta una grossa riprogettazione della MA.
Viene utilizzata solo per linguaggi di **basso livello**.

Simulazione mediante **Software**:
Ottima flessibilità **ma** velocità inferiore. Comunque si basa su altre M.A. che dovranno essere implementate.

Emulazione mediante **Firmware**:
Via di mezzo tra HW e SW, consente maggiore velocità rispetto a SW (ma non quanto HW) e maggiore flessibilità rispetto a HW (ma non quanto SW).
Possibile grazie a microprogrammi scritti un linguaggio di livello molto basso.

Che cos'è un compilatore?

Un compilatore da L a L_0 è un programma che realizza una funzione

$$\mathcal{C}_{\mathcal{L},\mathcal{L}o} : \mathcal{P}rog^{\mathcal{L}} \rightarrow \mathcal{P}rog^{\mathcal{L}o}$$

tale che, dato un programma $\mathcal{P}^{\mathcal{L}}$, *se*

$$\mathcal{C}_{\mathcal{L},\mathcal{L}o}(\mathcal{P}^{\mathcal{L}}) = \mathcal{P}c^{\mathcal{L}o}$$

allora, per ogni $Input \in \mathcal{D}^S$,

$$\mathcal{P}^{\mathcal{L}}(Input) = \mathcal{P}c^{\mathcal{L}o}(Input).$$

6. Relativamente alla tecnica di implementazione software, descrivere la tecnica di implementazione interpretativa pura e quella compilativa pura.

Dato un linguaggio L_1 e una m.a. ospite per L_O un'implementazione è:

- **compilativa pura** se si fornisce un compilatore da L_1 a L_O (che viene eseguito su una **qualsiasi** m.a. in nostro possesso, non necessariamente M_O). Ogni istruzione di L_1 viene esplicitamente tradotta in un insieme di istruzioni di L_O. In questo caso la fase di traduzione avviene **prima** della fase di esecuzione.

- **interpretativa pura** se si fornisce un interprete per L scritto in L_O (viene eseguito sulla macchina ospite M_O). Non vi è alcuna traduzione del codice solo una decodifica che assegna ad ogni istruzione di L un insieme di istruzioni di L_O che vengono eseguite **direttamente**.

7. Quando un interprete si può dire corretto? Quando un compilatore si può dire corretto?

8. Confrontare l'implementazione di una macchina astratta su una macchina ospite per mezzo di un interprete o di un compilatore.

- Implementazione **interpretativa**

  Il principale **svantaggio** consiste risiede nella scarsa efficienza. Infatti ai tempi di esecuzione del programma bisogna sommare i tempi necessari alla decodifica del codice sorgente. L'interprete **non genera codice**: il codice prodotto della traduzione non viene prodotto dall'interprete ma descrive solamente le operazioni che questo deve effettuare.
  Gli svantaggi in termini di efficienza sono bilanciati dai vantaggi in termini di **flessibilità**, per esempio per poter modificare a run-time il funzionamento del programma

- Implementazione **compilativa**

  La traduzione di un programma avviene **separatamente** rispetto alla sua esecuzione. Trascurando il tempo necessario alla compilazione il programma oggetto **eseguirà più velocemente** della sua versione interpretata. Inoltre ogni istruzione viene tradotta solamente una volta, indipendentemente dal numero di occorrenze all'interno del programma. I principali **svantaggi** risiedono nella perdita di informazioni riguardo alla struttura del programma sorgente, utili in fase di debug.

9. Come vengono implementate nella realtà le macchine astratte? Che cos'è la macchina intermedia?
Nella realtà si tende a usare una versione **intermedia** tra l'implementazione compilativa pura e quella interpretativa pura.
In particolare viene creata una macchina astratta intermedia (chiamiamola M_I e il suo linguaggio L_I), viene creato un compilatore dal linguaggio di partenza (chiamiamolo L_1) al linguaggio intermedio (da L_1 a L_I) e un interprete scritto in L_O (linguaggio della macchina ospite) che interpreta il linguaggio intermedio L_I.

10. Quando si dice che una implementazione è di tipo interpretativo e quando di tipo compilativo? Fare esempi di linguaggi la cui implementazione è di un tipo o dell'altro.

Se l'interprete del linguaggio intermedio è **completamente diverso** dall'interprete della macchina ospite l'implementazione si dice di tipo interpretativo (perché il lavoro grosso lo fa l'interprete).

Se l'interprete del linguaggio intermedio è **sostanzialmente uguale** all'interprete della macchina ospite l'implementazione si dice compilativa (perché il lavoro grosso è fatto dal compilatore).

11. L'interprete e il compilatore si possono sempre realizzare?
Si a patto di avere a disposizione un linguaggio "espressivo" almeno tanto quanto quello da implementare.
Vedi Linguaggi Turing-Completi
12. Che cosa è l'implementazione via kernel?
13. Quando si parla di bootstrapping?

Capitolo 2 – Descrivere un linguaggio di programmazione

14. Quali sono i livelli di descrizione di un linguaggio?

- **Grammatica**
  È quella parte della descrizione di un linguaggio che risponde alla domanda: "Quali frasi sono corrette?".
  Si compone di:
  - Analisi **Lessicale**
    Utilizzando un alfabeto sono riconosciute le sequenze corrette di simboli che costituiscono i **token** (parole) del linguaggio
  - Analisi **Sintattica**
    Descrive quali sequenze di token costituiscono frasi legali.
    La sintassi è dunque una **relazione tra segni**: tra tutte le possibili sequenze di parole seleziona un sottoinsieme di sequenze che costituiscono le frasi del linguaggio.

- **Semantica**
  La semantica attribuisce un **significato** ad ogni frase corretta. È una relazione tra segni (le frasi corrette) e significati (entità autonome che esistono a prescindere dai segni che usiamo per descriverle).

- **Pragmatica**
  È quella parte della descrizione di un linguaggio che stabilisce **come** usare una frase corretta e sensata in un certo **contesto**.

- **Implementazione**
  Nel caso dei linguaggi di programmazione ai tre livelli classici si aggiunge quello dell'implementazione, ovvero come le frasi "operative" del linguaggio realizzano lo stato di cui stanno parlando.

15. Sintassi: qual è l'aspetto lessicale e quale quello grammaticale di un linguaggio?
L'aspetto lessicale è il modo di costruire **token** a partire da caratteri dell'alfabeto.

L'aspetto sintattico invece è il modo di costruire "frasi" legali a partire da token.

16. Cos'è un alfabeto? Cos'è una parola o stringa? Cos'è A*? Tale insieme è enumerabile?
Un alfabeto è un insieme finito di simboli.
Una parola o stringa è una sequenza corretta di simboli appartenenti all'alfabeto.
A^* , che indica la chiusura di Kleene sull'alfabeto A, è numerabile.

17. Definizione di potenza di una stringa. Definizione di potenza di un linguaggio. Definizione di chiusura/iterazione (o stella di Kleene) di un linguaggio.
La chiusura di un linguaggio è costituita da tutte le concatenazioni finite di elementi in L ed è definita come:
$A^* = \cup_{n\geq 0} A^n$ dove:
$A^0 = \{\varepsilon\}$
$A^n = A * A^{n-1} = \{aw | a \in A, w \in A^{n-1}\}$

18. Definizione di grammatica libera da contesto. Come si deriva una stringa? Qual è il linguaggio generato da una grammatica libera?
Una grammatica libera da contesto è una quadrupla (NT, T, R, S) dove:

- NT è l'insieme finito dei simboli **non terminali**
- T è l'insieme finito dei simboli **terminali**
- R è l'insieme delle produzioni, ciascuna delle quali consiste in un espressione nella forma:

$$V \rightarrow w$$

dove $V \in NT$ e $w \in \{T \cup NT\}^*$

- $S \in NT$ è il simbolo iniziale

Fissata una grammatica $G = (NT, T, R, S)$ e assegnate due stringhe **v**, **w** su $\{T \cup NT\}$, diciamo che da v si **deriva immediatamente** w ($v \Rightarrow w$), se w si ottiene da v sostituendo ad un non terminale V in v il corpo di una produzione la cui testa sia V.
Diciamo che da v si **deriva** w ($v \Rightarrow^* w$) se esiste una sequenza finita di derivazioni immediate che porta da v a w.

Il **linguaggio generato** da una grammatica libera $G = (NT, T, R, S)$ è l'insieme $L[G] = \{w \in T^* \mid S \Rightarrow^* w\}$, ovvero l'insieme delle stringhe finite ottenibili dal simbolo iniziale S con una o più derivazioni sui terminali.

19. Cos'è un albero di derivazione? Cos'è una derivazione canonica sinistra/destra? Esiste una corrispondenza biunivoca tra alberi di derivazioni e derivazioni canoniche?
Data una grammatica libera $G = (NT, T, R, S)$, un **albero di derivazione** (o albero di parsing) è un albero ordinato in cui:

- ogni nodo è etichettato con un simbolo in $NT \cup T \cup \{\varepsilon\}$
- la radice è etichettata con S
- ogni nodo interno è etichettato con un simbolo in NT
- se il nodo n
    - ha etichetta A in NT e l'insieme delle etichette dei suoi figli è $\{X_i \in NT \cup T \ \forall i \in [1,k]\}$ allora esiste una produzione $A \rightarrow X_1 ... X_k \mid A \in R$
    - se il nodo n ha etichetta ε, allora n è una foglia, è figlio unico e, detto A suo padre, $A \rightarrow \varepsilon$ è una produzione in R
- se inoltre ogni nodo foglia è etichettato su $T \cup \varepsilon$, allora l'albero di derivazione è detto **completo**.

Una derivazione si dice **sinistra/destra** se ad ogni passo viene riscritto il non terminale più a sinistra (destra) della stringa.
Derivazioni sinistra e destra generano lo stesso albero.

Si può osservare che derivazioni canoniche e alberi sono in **corrispondenza biunivoca:** se due derivazioni sinistre per uno stesso NT sono diverse, allora ad un certo punto al NT più a sinistra si sono applicate due produzioni diverse e questo si traduce in alberi di derivazione distinti.

20. Quando una grammatica è ambigua? (fare un esempio) Quando un linguaggio è ambiguo? (fare un esempio)

Una grammatica è ambigua se e solo se esiste una stringa $v \in L[G]$ che ammette due o più derivazioni sinistre (destre) dal simbolo iniziale. Ad esempio la grammatica delle espressioni.
Un linguaggio è ambiguo quando tutte le grammatiche che lo generano sono ambigue.

21. È possibile rimuovere l'ambiguità dalla grammatica delle espressioni aritmetiche? Come?
Sì, decidendo precedenze delle derivazioni e manipolando la grammatica opportunamente. In base alla notazione utilizzata potrebbe dover essere necessario un uso estensivo delle parentesi.

22. Cos'è l'albero di sintassi astratta? Che differenza c'è tra sintassi concreta e sintassi astratta? Cos'è lo zucchero sintattico?
L'**albero di sintassi astratta** è un albero di derivazione che soddisfa i vincoli contestuali di un linguaggio in cui compaiono solo i terminali.

La sintassi **astratta** di un linguaggio è composta da tutti gli alberi di derivazione che soddisfano anche **i vincoli contestuali** del linguaggio.

Per zucchero sintattico si intendono simboli atti solo a chiarire la derivazione di una stringa.

23. Fare esempi di vincoli sintattici contestuali. Possono essere catturati attraverso grammatiche libere?
- il numero e il tipo dei parametri attuali di una chiamata di procedura deve essere uguale a quello dei parametri formali della dichiarazione.
- un identificatore dev'essere dichiarato prima dell'uso
- prima di usare una variabile dev'esserci stato un assegnamento su di essa

24. Definizione di grammatica dipendente dal contesto e di grammatica monotona.
Grammatica dipendente dal contesto
produzioni del tipo

$$uAv \Rightarrow uvw \text{ con } u,v,w \in \{T \cup NT\},\ A \in NT$$

La produzione $S \Rightarrow \varepsilon$ è ammessa solo se S non compare a destra di nessun altra produzione. (Ricordiamoci che S è il simbolo iniziale).

Grammatiche monotone
basta che la lunghezza del sequenza di simboli a sinistra della produzione sia minore o uguale alla lunghezza della sequenza di simboli a destra della produzione.
25. Cosa s'intende per semantica statica? E per semantica dinamica?

Con **sintassi** non contestuale si intende descrivibile con una grammatica libera

Per semantica **statica** si intende (detta anche sintassi contestuale) descrivibile con vincoli contestuali verificabili staticamente all'interno del testo del programma.

Per semantica **dinamica** si intende quei vincoli che sono verificabili solo al momento dell'esecuzione del programma.

26. Elencare le varie fasi in cui si articola un compilatore. Descrivere in dettaglio ogni singola fase.

- Analisi **lessicale** (*scanning*)
  Scopo dell'analisi lessicale è quello di leggere sequenzialmente simboli di ingresso di cui è composto il programma e di raggruppare tali simboli in unità logicamente significative chiamate **token**. Nessun controllo è ancora fatto sulla sequenza di token (ad esempio il bilanciamento delle parentesi).

- Analisi **sintattica** (*parsing*)
  Costruita la lista di token il parser cerca di costruire un albero di derivazione per tale lista. Ogni foglia di tale albero è costituita da un token contenuto nella lista e leggendo da sinistra a destra le foglie dell'albero si deve ottenere una frase legale del linguaggio.
  Può accadere che il parser non sia in grado di costruire un albero. In quel caso la stringa in ingresso non è una stringa corretta secondo la grammatica del linguaggio.

- Analisi **semantica**
  L'albero di derivazione viene sottoposto ai controlli relativi ai vincoli contestuali. Via via che questi controlli vengono effettuati, l'albero di derivazione viene aumentato con la relativa informazione (ad es tipo, luogo della dichiarazione etc)

- Generazione forma **intermedia**
  Visitando l'albero viene generato un codice intermedio che è sia indipendente dal architettura che dal sorgente.
  - easy to produce (operazioni semplici)
  - easy to translate (segue la struttura dell'albero sintattico)

- Ottimizzazione forma intermedia
  - Rimozione di codice inutile
  - Espansioni *in-line* di chiamate di funzione
  - Fattorizzazione di sottoespressioni per non calcolare più volte lo stesso valore
  - Mette fuori dai cicli le sottoespressioni che non variano.

- Generazione del codice
  Viene generato codice oggetto per una specifica architettura (include assegnazione dei registri e ottimizzazioni specifiche per l'architettura e il codice oggetto)

27. A chi serve definire la semantica di un linguaggio e perché?

Definire la semantica di un linguaggio di programmazione è molto più difficile della sintassi. La complessità nasce dall'esigenza di mediare tra due istanze contrapposte: la ricerca dell'esattezza, da una parte, e quella della flessibilità dall'altra in modo da rimuovere l'ambiguità per un utente, ma anche da lasciare spazio all'implementazione.

28. Quali tecniche si usano per dare semantica ad un linguaggio di programmazione?
I metodi formali per la semantica si dividono in due grandi famiglie:

- Semantiche **denotazionali**
  È l'applicazione ai linguaggi di programmazione di tecniche sviluppate per la semantica del linguaggio logico-matematico. Il significato di un programma è dato da **una funzione**, che esprime il comportamento input/output del programma stesso.

- Semantiche **operazionali**
  Nell'approccio operazionale, invece, non vi sono entità esterne (es. funzioni) da associare ai costrutti del linguaggio. Una semantica operazionale specifica il comportamento della macchina astratta, ossia ne **definisce l'interprete**, facendo riferimento ad un formalismo di più basso livello.

29. Imparare le regole di semantica operazionale SOS per il semplice linguaggio presentato a lezione. Regole di valutazione interna--sinistra ed esterna--sinistra.
30. Cosa s'intende per pragmatica?
Per pragmatica si intende il modo corretto di utilizzare un costrutto corretto. È evidente, dunque, che la pragmatica non viene stabilita una volta per tutte al momento della definizione del linguaggio. Al contrario essa evolve assieme all'uso che del linguaggio viene fatto.

Capitolo 3 – Analisi lessicale--Linguaggi regolari

31. Cosa fa l'analizzatore lessicale?
Scopo dell'analizzatore lessicale è quello di riconoscere nella stringa in ingresso alcuni gruppi di caratteri che corrispondono a certe categorie sintattiche. In tal modo la stringa in ingresso è trasformata in una sequenza di simboli astratti, detti **token**, che sono poi passati all'analizzatore sintattico.

32. Cos'è un token?
È un'informazione astratta che rappresenta una stringa del testo in ingresso. È una **coppia** (nome, valore): il **nome** del token è un simbolo astratto che rappresenta una categoria sintattica, il **valore** invece è costituito da una sequenza di simboli del testo in ingresso.

33. Cos'è un pattern? Come lo si rappresenta? Cos'è un lessema?
Le sequenze di caratteri associati ad un token sono specificate mediante **pattern**, cioè una descrizione generale della forma che le sequenze di caratteri possono assumere. Questa descrizione generale è data mediante **espressioni regolari**.

Una stringa dell'alfabeto che corrisponde ad un pattern si dice **lessema**.

34. Definizione di espressioni regolari (sintassi) e di linguaggio associato (semantica).

1. *ϵ è un'espressione regolare;* $\mathcal{L}[\epsilon] = \{\epsilon\}$.
2. *Per ogni $a \in A$, $\mathbf{a}$ è un'espressione regolare;* $\mathcal{L}[\mathbf{a}] = \{a\}$.
3. *Se r è un'espressione regolare, (r) è un'espressione regolare;* $\mathcal{L}[(r)] = \mathcal{L}[r]$.
4. *Se r e s sono espressioni regolari, $(r) \mid (s)$ è un'espressione regolare;* $\mathcal{L}[(r)|(s)] = \mathcal{L}[r] \cup \mathcal{L}[s]$.
5. *Se r e s sono espressioni regolari, $(r) \cdot (s)$ è un'espressione regolare;* $\mathcal{L}[(r) \cdot (s)] = \mathcal{L}[r]\mathcal{L}[s]$.
6. *Se r è un'espressione regolare, $(r)^*$ è un'espressione regolare;* $\mathcal{L}[(r)^*] = \mathcal{L}[r]^*$.
7. *Nient'altro è un'espressione regolare.*

35. Quali sono i linguaggi regolari? I linguaggi finiti sono tutti regolari? Esistono linguaggi infiniti regolari?

Un linguaggio L è regolare sse L è vuoto, oppure esiste un espressione regolare $s \mid L = L[s]$.

I linguaggi finiti sono tutti regolari. Esistono linguaggi infiniti che sono regolari.

36. Definizione di equivalenza tra espressioni regolari. Elencare alcune leggi di equivalenza.

Due espressioni regolari sono equivalenti se generano lo stesso linguaggio.

- r | s = s | r
- r** = r*
- r(st) = (rs)t

37. Cos'è una definizione regolare e a cosa serve?

E' una lista di definizioni di simboli a cui ad ogni simbolo è associato un espressione regolare.

E' utile per esprimere espressioni regolari complesse, li si divide in sottoespressioni in cui a ognuna è associato un simbolo.

38. Definizione di NFA (automa finito non deterministico). Discutere cosa si intenda per non determinismo. Mettere in relazione la definizione formale con la rappresentazione grafica come diagramma di transizioni.

Un NFA è una quintupla $(\Sigma, Q, \delta, q0, F)$ dove:

- Σ (sigma) è un insieme finito di simboli
- Q è un insieme finito di stati
- δ (delta) è un insieme di funzioni definite come: $\delta : Qx(\Sigma \cup \{\varepsilon\}) \rightarrow P(Q)$
- $q_0 \in Q$ è uno stato iniziale
- $F \subseteq Q$ è un insieme finito di stati finali

La descrizione più compatta ed efficace di un NFA è mediante un **diagramma di transizione**, cioè un grafo orientato i cui nodi rappresentano gli stati, mentre gli archi, etichettati con i simboli dell' alfabeto dell'input rappresentano le possibili transizioni tra gli stati.

39. Come si definisce il linguaggio accettato da un NFA? Definizione di equivalenza tra NFA.
Il linguaggio **accettato** da un NFA N è l'insieme $L[N] = \{x \in \Sigma^* \mid N \text{ accetta } x\}$.

40. Definizione di DFA (automa finito deterministico). Discutere cosa si intenda per determinismo. Mostrare che un DFA è un caso speciale di NFA, ovvero la classe dei DFA è un sottoinsieme della classe degli NFA.
Un DFA è una quintupla $(\Sigma, Q, \delta, q_0, F)$ dove:

- Σ (sigma) è un insieme finito di simboli
- Q è un insieme finito di stati
- δ (delta) è un insieme di funzioni definite come: $\delta : Q \times \Sigma \rightarrow q$
- q_0 è lo stato iniziale
- $F \subseteq Q$ è un insieme finito di stati finali

Un DFA è un caso particolare di NFA che si ha quando non vi sono archi etichettati con ε e inoltre nella funzione di transizione si ha che δ(q,a) è sempre **esattamente** costituita da un solo stato.

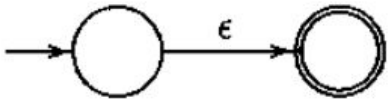
41. Dato un NFA, come si ricava un equivalente DFA? Ovvero descrivere come è definita la costruzione per sottoinsiemi. Definizione di epsilon--closure e algoritmo associato. Qual è la complessità della costruzione per sottoinsiemi nel caso pessimo? Ovvero se NFA ha n stati, quanti stati può avere il DFA equivalente?

L'idea di fondo è che uno stato di M_N può essere pensato come un insieme degli stati di N. In particolare, se N, partendo dallo stato iniziale e consumando l'input $a_1 \dots a_k$ può trovarsi in uno qualsiasi degli stati $q_1 \dots q_k$, allora M, iniziando nel suo stato iniziale e consumando lo stesso input $a_1 \dots a_k$ si troverà nello stato che corrisponde all'insieme $\{ q_1 \dots q_k \}$.

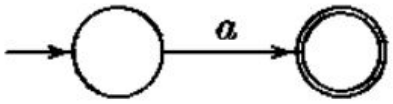
Il DFA equivalente può avere fino a 2^n stati.

42. Dato i due punti precedenti, enunciare il teorema che dice che la classe dei linguaggi riconosciuto da NFA coincide con la classe dei linguaggi riconosciuti da DFA.
Sia $N = (\Sigma, Q, \delta, q, F)$ un NFA e M_N l'automa ottenuto con la costruzione per sottoinsiemi. Allora M_N è un DFA e si ha $L[N] = L[M]$.

43. Come si costruisce un NFA a partire da una espressione regolare, in modo tale che il linguaggio riconosciuto dall'NFA sia lo stesso del linguaggio associato all'espressione regolare?
Per farlo basta costruire per induzione usando il seguente schema:

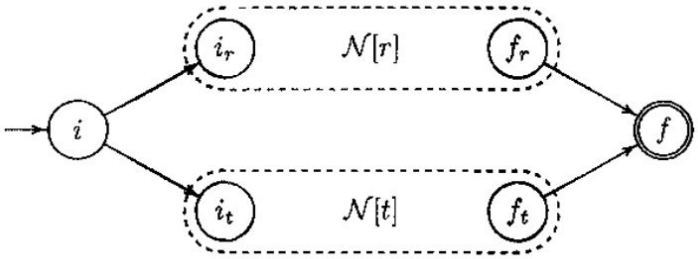


per eventuali epsilon oppure

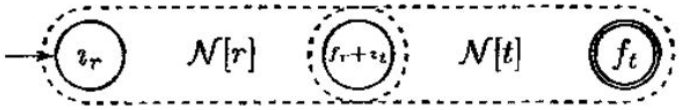


per ogni a appartenente all'alfabeto

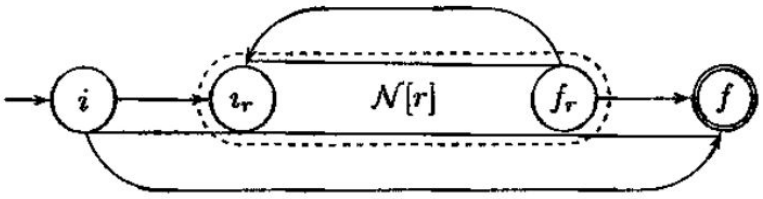
Assumendo di avere due NFA per 2 espressioni regolari, l'OR si costruisce



Assumendo di avere due NFA per 2 espressioni regolari, la concatenazione si costruisce



Assumendo di avere due NFA per 2 espressioni regolari, la stella di kleene si costruisce



44. Definizione di grammatica regolare.

Una grammatica libera è regolare sse ogni produzione è della forma

$$V \rightarrow aW \text{ oppure } V \rightarrow a$$

dove $V, W \in NT$, $a \in T$.

Per il simbolo iniziale è ammessa anche S → ε.

45. Come si associa ad una grammatica regolare un equivalente NFA?

È necessario passare attraverso un'espressione regolare.

46. Dato un DFA, come si costruisce una grammatica regolare equivalente?

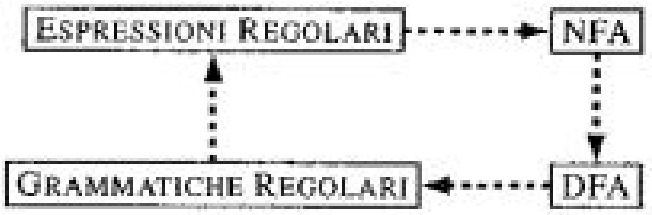
Dim. La costruzione non è difficile. Sia $M = (\Sigma, Q, \delta, q_0, F)$ il DFA assegnato. La grammatica $G_M = (Q, \Sigma, R, q_0)$ ha:

1. come non terminali, gli stati di M;
2. come terminali, l'alfabeto di M;
3. come simbolo iniziale, lo stato iniziale q_0 di M;
4. come produzioni R:
   (a) per ogni transizione $\delta(q_i, a) = q_j$ di M, la produzione $q_i \to aq_j \in R$; inoltre, se $q_j \in F$, anche la produzione $q_i \to a \in R$;
   (b) se lo stato iniziale di M è anche finale (cioè $q_0 \in F$), allora la produzione $q_0 \to \epsilon \in R$.

47. Data una grammatica regolare, come si costruisce un'espressione regolare equivalente?

Si deve creare un'espressione regolare che riconosca il linguaggio generato dalla grammatica.

48. Descrivere, con un diagramma riassuntivo, tutte le relazioni fra i formalismi introdotti: NFA, DFA, grammatiche regolari, espressioni regolari. Questo diagramma dimostra che tutti questi formalismi sono equivalenti e descrivono la classe dei linguaggi regolari.



49. Definizione di stati equivalenti in un DFA. Quando due stati di un DFA sono indistinguibili?

Due stati sono equivalenti se non sono distinguibili.

Una stringa *x* distingue due stati q_0, q_1 se il cammino che parte in q_0 e consuma x arriva in uno stato finale e il cammino che parte in q_1 e consuma x arriva in uno stato non finale; oppure viceversa.

50. Come funziona l'algoritmo iterativo con tabella a scala per produrre le classi di equivalenza di stati di un DFA?

- Si crea una tabella con tutte le coppie di stati.
- Si marca ogni coppia finale-non finale.
- finché nel ciclo c'è stata almeno una marcatura, scorri tutte le coppie non marcate
  prova tutte le mosse possibili (l'alfabeto) e se almeno una porta la coppia di stati su una coppia già marcata, marca anche questa.

51. Una volta determinate le classi di equivalenza degli stati di un DFA, come si costruisce l'automa minimo associato?

52. Cos'è lex? Qual è il suo input e il suo output?
53. Qual è la struttura di un file .lex? Come funziona l'analizzatore lessicale prodotto da lex?
54. Come si interfaccia lex con Yacc?
55. Intestazione e dimostrazione del pumping lemma.

Intestazione
Dato un linguaggio L regolare esiste una costante N>0 tale che qualsiasi stringa del linguaggio più lunga di N può essere suddivisa in tre stringhe uvw tale che:
la cardinalità di uv è minore o uguale a N
la cardinalità di v è maggiore o uguale a 1
puoi pompare v quante volte vuoi ma continua ad appartenere al linguaggio
Inoltre N è minore o uguale al numero di stati del DFA minimo che accetta L.

Dimostrazione (all'incirca per capirci, ma efficace)
Ipotesi
M = DFA che accetta L e lo prendiamo pure minimo così l'ultima parte è ok.
N = numero stati di M = $|Q_M|$
z = stringa appartenente al linguaggio L
m = lunghezza stringa z, $m \geq N$

1. prendiamo l'elenco dei **m+1 stati del percorso che riconoscono la stringa z** (stato iniziale + ogni stato per ogni carattere della stringa) .

2. Prendiamo gli **N+1** stati di questo elenco.

3. Visto che m>=M allora $m+1 \geq N+1$ e **si passa più volte nello stesso stato** nel percorso.

4. Quindi ad un certo punto c'è un loop dove $a_i = a_j$. Quindi diciamo che gli u sono gli stati fino a i poi nel loop ci sono gli stati di v e infine i w.

5. Di sicuro uv è minore di N perché i e j sono stati scelti tra i primi N+1 stati (con cui attenzione si consumano N archi/simboli). Visto che c'è un loop e i != j allora il loop ha almeno un nodo e quindi la cardinalità di v≥ 1 .

6. La stringa può essere pompata perché possiamo eseguire il ciclo quante volte ci pare tanto l'automa è deterministico e finirà sempre in uno stato finale quindi riconoscerà la stringa.

Ricordiamo che PL è una condizione **necessaria ma non sufficiente**. ci sono linguaggi che soddisfano il PL ma non sono regolari.

56. Come si può utilizzare il pumping lemma (a rovescio) per dimostrare che un linguaggio non è regolare?
57. Quali sono le proprietà di chiusura dei linguaggi regolari? Quali proprietà si possono decidere (ovvero verificare algoritmicamente)?

I linguaggi regolari sono chiusi per:

- Unione (ovvio per il fatto che i linguaggi regolari sono generati da un'espressione regolare)
- Intersezione $L[M_1] \cap L[M_2] = L[\overline{M_1}] \cup L[\overline{M_2}]$
- Complementazione: $L[N]$ $DFA\ N = (\Sigma, Q, \delta, s_0, F)$
  $L[\overline{N}]$ $DFA\ \overline{N} = (\Sigma, Q, \delta, s_0, Q/F)$ *inverto gli stati finali e non finali*
- Ripetizione (ovvio per il fatto che i linguaggi regolari sono generati da un'espressione regolare)
- Differenza (dimostrabile per il fatto di essere chiusi per intersezione)
- Concatenazione (ovvio per il fatto che i linguaggi regolari sono generati da un'espressione regolare)

Capitolo 4: Analisi sintattica – linguaggi liberi

58. Cosa si intende per analisi sintattica? Cos'è un parser?
L'analisi sintattica descrive quali sequenze di token costituiscono frasi legali.

Un **parser** è un riconoscitore di linguaggio libero detto anche *analizzatore sintattico*. Si tratta dunque di un programma, in genere basato su PDA per il linguaggio riconosciuto, che usa l'input per decidere le produzioni da utilizzare e costruisce un albero di derivazione se esiste.

59. Definizione di automa a pila non deterministico (PDA). Definizione di configurazione (o descrizione istantanea), mossa in un passo e mossa in più passi.
Un automa a pila non deterministico (PDA) è una settupla $(\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$

- Σ (Sigma) è l'insieme finito dei simboli in input
- Q è l'insieme finito degli stati
- Γ (Gamma) è l'insieme finito dei simboli della pila
- δ (delta) è la funzione di transizione che prende:
  - uno **stato corrente**,
  - un **simbolo in input**,
  - un **simbolo della pila** (in cima allo stack)

  e ritorna un **insieme di coppie stato** in cui vado a finire e sequenza di caratteri che vado a scrivere in cima alla pila.

  $$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow P(Q \times \Gamma^*)$$
- $q_0 \in Q$ è lo stato iniziale
- $\perp \in \Gamma$ è il simbolo di fondo della pila
- $F \subseteq Q$ è l'insieme degli stati finali

Una **configurazione** (o descrizione istantanea) è una tripla costituita da uno stato, da un simbolo in input e dalla stringa in cima alla pila.

$$(q, w, \beta) | q \in Q, w \in \Sigma^*. \beta \in \Gamma^*$$

Una **mossa** di un passo è una transizione tra due configurazioni:

$$(q_1, aw, X\beta) \rightarrow (q_2, w, \alpha\beta) \mid q_1, q_2 \in Q,\ a, w \in \Sigma^*,\ X, \beta \in \Gamma^*$$

60. Definizione di linguaggio accettato da un PDA per stato finale o per pila vuota. Sono equivalenti queste due modalità di riconoscimento?

Sia $P = (\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$ un PDA:

- Il linguaggio accettato da P per stato finale è:
  $L[P] = \{w \in \Sigma^* | (q_0, w, \perp) \rightarrow_* (q, \varepsilon, \alpha),\ q \in F\}$
- Il linguaggio accettato da P per pila vuota è:
  $N[P] = \{w \in \Sigma^* | (q_0, w, \perp) \rightarrow_* (q, \varepsilon, \varepsilon)$

Non sono equivalenti dato lo stesso PDA P, cioè $L[P] \neq N[P]$ però la classe dei linguaggi riconosciuti non cambia se accettiamo per pila vuota o per stato finale, cioè sono sempre i linguaggi liberi da contesto.

61. Mostrare come data una grammatica libera G, sia possibile costruire un PDA P equivalente. È possibile costruire, dato un PDA P, una grammatica equivalente G? Concludere che la classe dei linguaggi liberi coincide con la classe dei linguaggi riconosciuti da PDA.

Data una grammatica $G = (NT, T, R, S)$ è possibile costruire un PDA
$P = (T, \{q\}, T \cup NT, \delta, q, S, \varnothing)$ equivalente con un solo stato q.
$\forall r = (V \rightarrow W) \in R \mid V \in NT, W \in (T \cup NT)\ \exists\ \delta : (q, \varepsilon, V) \rightarrow (q, \varepsilon, W)$ mossa di espansione
$\forall a \in T\ \exists\ \delta : (q, a, a) \rightarrow (q, \varepsilon)$ mossa di match
Il riconoscimento del PDA è di tipo top-down ed è fortemente non deterministico in questo caso.
E' possibile costruire dato un PDA una grammatica equivalente perché ogni PDA P è riducibile a un PDA equivalente con un solo stato. Dato un PDA con un solo stato è possibile costruire una grammatica equivalente.
Quindi visto che è possibile ottenere una grammatica libera da un qualsiasi PDA non deterministico e la classe dei linguaggi liberi è uguale alla classe dei linguaggi generati da grammatiche libere, la classe dei linguaggi accettati da PDA è quella dei linguaggi liberi.

62. Quali sono le proprietà di chiusura dei linguaggi liberi?

L'intersezione di un linguaggio libero con un regolare è un linguaggio libero?

I linguaggi liberi sono chiusi per:

- Unione: si dimostra aggiungendo una produzione $S \rightarrow S_1 | S_2$

- Concatenazione: si dimostra aggiungendo una produzione $S \rightarrow S_1 S_2$
- Ripetizione: si dimostra aggiungendo una produzione $S_1 -> \varepsilon | S_1 S$

L'intersezione di linguaggio libero con un regolare non è necessariamente un linguaggio libero.

63. Intestazione e dimostrazione del "Pumping Theorem".

Se un linguaggio L è libero allora $\exists N > 0 \mid \forall w \in L$ si può scomporre in cinque sottostringhe uvwxy tale che:

- $|vwx| \leq N$
- $|vx| \geq 1$
- $\forall k \geq 0 \; uv^k wx^k x \in L$

64. Come si può utilizzare tale teorema (a rovescio) per dimostrare che un linguaggio non è libero?

Se L è libero allora vale il pumping theorem. Se il pumping theorem non vale (o vale il suo opposto) allora il linguaggio non è libero.
Il linguaggio L non è libero se $\forall N > 0 \; \exists w \in L$ tale che w può essere scomposto in cinque sottostringhe uvwxy tali che se:

- $|vwx| \leq N$
- $|vx| \geq 1$

allora $\forall k \geq 0 \quad uv^k wx^k y \notin L$

65. Classificazione di Chomsky delle grammatiche e dei linguaggi. Quale tipo di automi corrisponde ad ogni classe?

Grammatiche **illimitate** (o a struttura di fase): linguaggi semidecidibili : macchina di Turing
Grammatiche **dipendenti dal contesto** : linguaggi contestuali : automa limitato linearmente
Grammatiche **libere da contesto** : linguaggi liberi : automi a pila
Grammatiche **regolari** : automi a stati finiti

66. Definizione di DPDA (automa a pila deterministico) e di linguaggio libero deterministico.

Un PDA $P = (\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$ è deterministico se:

- $\forall q \in Q, \; Z \in \Gamma, \; se \; \delta(q, \varepsilon, Z) \neq \varnothing \; allora \; \delta(q, a, Z) = \varnothing \quad \forall a \in \Sigma$
  (se c'è una produzione epsilon allora non ci deve essere nessun'altra produzione)

- $\forall q \in Q, \; Z \in \Gamma \quad e \quad a \in \Sigma \cup \{\varepsilon\}, \; |\delta(q, a, Z)| \leq 1$
  (al massimo una mossa per ogni tripla stato, simbolo, stringa pila)

Un linguaggio libero deterministico è un linguaggio libero accettato da un automa a pila deterministico **per stato finale**.

67. La classe dei linguaggi liberi deterministici è strettamente inclusa in quella dei linguaggi liberi? Contiene strettamente la classe dei linguaggi regolari?
Sì, la classe dei linguaggi liberi deterministici è strettamente inclusa in quella dei linguaggi liberi.
Sì, la classe dei linguaggi liberi deterministici contiene strettamente la classe dei linguaggi regolari.

68. Che cosa dice la prefix property e perché è interessante per i DPDA?

Un linguaggio L gode della prefix property se $\forall w \in L \nexists y \in L \mid yx = w$
Se un linguaggio gode della prefix property allora può essere accettato da un DPDA.

69. Usando un endmarker $, si può riconoscere un linguaggio libero deterministico che non gode della prefix property anche per pila vuota? Come?
Sì, basta distinguere ogni parola con l'endmarker $ in modo che il DPDA sappia rilevare i prefissi. Quando siamo in uno stato finale è leggiamo l'endmarker e non ci sono più simboli in input possiamo svuotare la pila e accettare.

70. Un linguaggio libero deterministico è ambiguo?
No, non sono mai ambigui.

71. I linguaggi liberi deterministici sono chiusi per complementazione, ma non per intersezione o per unione. Perché?
72. Da cosa si parte per costruire un analizzatore sintattico (ovvero parser)? Da una espressione regolare? Da una grammatica libera? Da un PDA?
Da una grammatica libera.

73. Cosa prende in input e cosa produce in output un parser?
Un parser prende in input una sequenza formata da token e produce in output un albero di derivazione se la sequenza in input appartiene al linguaggio.

74. Che differenza c'è tra un parser non deterministico ed uno deterministico?
Un parser non deterministico davanti a una scelta tenta tutte le possibili opzioni (**enumerazione**) mentre un parser deterministico utilizza il **lookahead** per poter decidere qual è l'unica opzione che potrebbe portare al riconoscimento della sequenza in input.

75. Quali sono le due tecniche essenziali per costruire parsers?
Le tue tecniche essenziali sono il top-down parsing e il bottom-up parsing.

76. Le tecniche top-down e bottom-up in che cosa differiscono?
I parser **top-down** sfruttano i simboli in input per costruire una derivazione sinistra a partire dal simbolo iniziale. Costruiscono incrementalmente l'albero di derivazione, espandendo le foglie finchè tutte le foglie sono etichettate con terminali.

I parser **bottom-up** invece analizzano l'input e cercano di capire quali produzioni possono aver generato quella sequenza di simboli terminali. I parser bottom-up ricercano una

derivazione destra e costruiscono incrementalmente l'albero partendo da delle foglie già etichettate come terminali e cercando di far crescere l'albero verso la radice che dovrà essere etichettata come il simbolo iniziale.

77. Quali tipi di grammatiche non sono adatte al top--down parsing? Quali tipi di produzioni sono poco adatte al bottom--up parsing?
Non sono adatte al top-down parsing le grammatiche che presentano **ricorsione sinistra** o sono ambigue.
Invece non sono adatte al bottom-up parsing solamente le grammatiche ambigue.

78. Cosa sono le produzioni epsilon e cosa sono i simboli non terminali annullabili?
Una produzione epsilon è una produzione nella forma $A \rightarrow^* \varepsilon$.
Un simbolo non terminale A è annullabile se $A \rightarrow^+ \varepsilon$.

79. Come si può trasformare una grammatica G che contiene produzioni epsilon in una grammatica G' che non ne contiene, preservando il linguaggio a meno di epsilon?
Costruiamo una nuova grammatica G' che ha come produzioni R' tutte le possibili produzioni che si ottengono cancellando dalle produzioni di R combinazioni di simboli non-terminali annullabili.

80. Cosa sono le produzioni unitarie e cosa sono le coppie unitarie?
Una produzione unitaria è una produzione nella forma $A \rightarrow B \; con \; A, B \in NT$
Due simboli non terminali formano una coppia unitaria (A;B) se $A \rightarrow^* B$

81. Come si può trasformare una grammatica G che contiene produzioni unitarie in una equivalente G' che non ne contiene?
Costruiamo una grammatica $G' = (NT, T, R_1, S)$ a partire da G tale che:
$\forall (A;B) \in U(G)$ R_1 contiene tutte le produzioni $B \rightarrow \alpha$ non unitarie.

82. Data una grammatica G, quali sono i suoi simboli utili? Quali sono i suoi simboli generatori? Quali i suoi simboli raggiungibili?
I simboli utili sono i simboli che sono sia generatori che raggiungibili.
Il simbolo $X \in T \cup NT$ è:

- generatore $sse \; \exists w \in T^* \mid X \rightarrow^* w$
- raggiungibile $sse \; S \rightarrow^* \alpha X \beta$ per qualche $\alpha, \beta \in (T \cup NT)^*$

83. Come si calcolano i generatori? Come si calcolano i raggiungibili?
Data una grammatica G l'insieme dei suoi simboli generatori è calcolato iterativamente come segue:

- $G_0(G) = T$
- $G_{i+1}(G) = G_i(G) \cup \{B \in NT | B \rightarrow C_1 ... C_k \in R, \; C_1 ... C_k \in G_i(G)\}$

L'insieme dei simboli raggiungibili è calcolato sempre iterativamente:

- $R_0(G) = \{S\}$
- $R_{i+1}(G) = R_i(G) \cup \{X_i \in T \cup NT \mid B \rightarrow X_1 ... X_n \in R, \; per \; qualche \; B \in R_i(G) \}$

84. In che modo si eliminano i simboli inutili di una grammatica? È importante l'ordine delle operazioni da svolgere?
Prima vanno eliminati i non generatori, poi i non raggiungibili. Sì, l'ordine è importante.

85. Quando si dice che una grammatica è ricorsiva sinistra? Come si elimina la ricorsione sinistra immediata? E quella non immediata? Perché serve eliminare la ricorsione sinistra?
Una grammatica si dice ricorsiva sinistra quando presenta delle produzione nella forma
$A \rightarrow^{+} A\alpha$
Per eliminare la ricorsione sinistra immediata:
se $A \rightarrow Aa_1|...|Aa_n|B_1|...|B_m$ che presenta ricorsione sinistra
allora divido la produzione nelle seguenti produzioni che non presentano la ricorsione sinistra:
$A \rightarrow B_1A'|...|B_mA'$
$A' = a_1A'|...|a_nA'|\varepsilon$
Per eliminare la ricorsione sinistra non immediata esiste un algoritmo più complesso che però non è stato trattato a lezione.

---> Eliminare la ricorsione sinistra è importante per poter utilizzare parser **top-down**.

86. Cosa vuol dire fattorizzare a sinistra una grammatica? Perché serve fattorizzare?
Fattorizzare una grammatica significa cambiarne le produzioni in modo da raccogliere la parte comunque iniziale tra esse.

---> Fattorizzare una grammatica è importante perché se in più produzioni i primi k simboli sono uguali, allora un parser *left-to-right* con solo k simboli di look-ahead non potrebbe scegliere che produzione utilizzare e necessiterebbe di k+1 simboli di look-ahead. Fattorizzare serve a poter **ridurre** il numero di simboli di look-ahead.

$$A \rightarrow aBbC \mid aBd$$

$$A \rightarrow aBA'$$
$$A' \rightarrow bC \mid d$$

87. Cos'è un parser a discesa ricorsiva?
Un parser a discesa ricorsiva è un tipo di parser top-down in cui si associa una funzione ad ogni produzione di ogni simbolo **non terminale**. Questa funzione è ricorsiva e ha il compito di riconoscere se una certa parte di input può essere derivata dalla produzione a cui è associata.

88. Definizione di First(α) e di Follow(A).

- **First**
  Definiamo First(α) come l'insieme dei **terminali** che possono stare in prima posizione in una stringa derivata da α.
  $$First(\alpha) : \{x \in T | \ \alpha \rightarrow^{*} x\beta \ per \ qualche \ \beta \in (T \cup NT)^{*}\} \cup \{\varepsilon \ sse \ \alpha \rightarrow^{*} \varepsilon\}$$

- **Follow(A)**
  Definiamo Follow(A) come l'insieme dei terminali che possono comparire a destra di A in una forma sentenziale.
  $Follow(A) : \{x \in T | S \rightarrow^* \alpha A x \beta \ per\ qualche\ \alpha, \beta \in (T \cup NT)^*\} \cup \{\$ \mid se\ S \rightarrow^* \alpha A\}$

89. Algoritmi per calcolare First(α) e di Follow(A).

90. Come è fatta e come si riempie una tabella di parsing LL(1)?
E' una tabella in cui le righe sono indicizzate per i non terminali e le colonne sono indicizzate per $(T \cup \{\$\})$.
Nella casella $M[A, a]$ sono contenute le produzioni che possono venire scelte quando si sta espandendo una produzione A e il simbolo in input corrente è *a.*
Per riempirla **per ogni produzione** $A \rightarrow \alpha$**:**

- $\forall x \in T$ e $x \in First(\alpha)$, inserisci $A \rightarrow \alpha$ in $M[A, \alpha]$
- $se\ \varepsilon \in First(\alpha)$, inserisci $A \rightarrow \varepsilon$ in M[A,x] $\forall x \in Follow(A)$
-

91. Quando una grammatica G si dice di classe LL(1)? Quali sono le condizioni necessarie e sufficienti per G affinché sia di classe LL(1)?
Una grammatica si dice di classe LL(1) se la tabella di parsing contiene in ogni cella al più ogni produzione.
Questo succede se e solo se nella grammatiche G per ogni coppia di produzioni con la stessa testa: $A \rightarrow \alpha \mid \beta$ si ha:

- $First(\alpha) \cap First(\beta) = \varnothing$
- se $\varepsilon \in First(\alpha)$ allora $First(\beta) \cap Follow(\alpha) = \varnothing$
- se $\varepsilon \in First(\beta)$ allora $First(\alpha) \cap Follow(\beta) = \varnothing$

92. Perché un parser di questo tipo è chiamato LL?
La prima L sta per **left-to-right** ed è l'ordine di lettura dell'input.
La seconda L sta per **Left derivation**.

93. Come funziona il parser LL(1) con una pila?
Il meccanismo della pila è in realtà implicito nella ricorsione.
Inizialmente si fa push del simbolo iniziale.
Poi finchè la pila non è vuota si toglie un elemento da questa, se è un terminale allora si cerca di fare match con il primo simbolo in cima alla pila altrimenti se è un non terminale lo si espande.

94. È vero che ogni linguaggio regolare è pure LL(1)?
Sì è vero che ogni linguaggio regolare è pure LL(1).

95. Come sono definiti Firstk(α) e di Followk(A) per k≥2.
Gli insiemi generalizzano First e Follow e contengono (invece di singoli simboli) stringhe lunghe al più *k*.

96. Come si definisce una grammatica LL(k)? Ed un linguaggio LL(k)? Come si relazionano tra di loro?
Una grammatica è LL(k) se il parser risultate è deterministico. Un linguaggio è LL(k) se è generato da una grammatica LL(k + 1).
97. Che relazione esiste tra grammatiche LL(k) e grammatiche ambigue? E con le grammatiche ricorsive sinistre?
Una grammatica ambigua o ricorsiva sinistra **non è LL(k) per nessun k**.

98. Esistono linguaggi liberi che non sono LL(k) per nessun k? Ed esistono linguaggi liberi deterministici che non sono LL(k) per nessun k?
Esistono linguaggi liberi che non sono LL(k) per nessun k.
Non esistono linguaggi **liberi deterministici** che non sono LL(k) per nessun k.

99. Cos'è un parser bottom--up (o shift--reduce)? Qual è il suo input e il suo output? Perché sono chiamati parser LR?

I parser bottom-up sono PDA che alternano tra le seguenti 2 azioni:

- **shift**: un simbolo terminale è spostato dall'input alla cima della pila
- **reduce**: un insieme di simboli (terminali o non) sulla cima della pila, corrispondenti alla parte destra di una produzione, viene eliminato e al suo posto viene messo sulla pila il simbolo non terminale della parte sinistra della produzione.

I parser LR prendono in input una stringa e in output producono un albero di derivazione destro se la stringa appartiene al linguaggio a cui il parser è associato.
Sono chiamati parser LR perchè L sta per *left-to-right* e indica l'ordine di lettura. R sta per *right-derivation* e indica il tipo di albero di derivazione creato.

100. Che tipo di conflitti si possono presentare in un parser del genere? Quando si presenta un conflitto (shift/reduce o reduce/reduce), quale azione bisogna scegliere?
Possono esserci 2 distinti tipi di conflitti:

- shift/reduce: quando potremmo sia ridurre perchè sulla pila c'è la parte destra di una produzione, però l'input non è ancora finito quindi potremmo anche fare un operazione di shift.
- reduce/reduce: quando la cima della pila corrisponde alla parte destra di più produzioni.

L'azione da scegliere dipende dai prossimi simboli in input, anche qui si utilizza la tecnica del **look-ahead.**

101. Cos'è un prefisso viabile? Come lo si definisce in termini di una grammatica?
Un prefisso viabile è una stringa $w \in (T \cup NT)^*$ che può presentarsi in cima a alla pila di un parser LR durante il parsing di una stringa che verrà accettata.
In termini di grammatiche una stringa $w \in (T \cup NT)^*$ è un prefisso viabile se per la grammatica G esiste una derivazione destra:
$S \rightarrow^* \delta Ay \rightarrow \delta\alpha\beta y = \gamma\beta y$
Da notare la derivazione parte dal simbolo iniziale, vorrà dire che la stringa verrà accettata.

Una **maniglia** è il terminale che quando si legge ci si accorge di poter fare un'operazione di reduce.

102. Cos'è un item LR(0)? Come si generano tutti gli item di una grammatica (aumentata con un simbolo iniziale nuovo S')?

Un item LR(0) per una grammatica G è una sua produzione in cui è indicata esplicitamente **una posizione nella sua parte destra**, per esempio con un punto.
Se un item ha il punto **in ultima posizione** allora indica la presenza di una maniglia nella pila. Gli item di una grammatica si generano creando un item per ogni posizione di ogni produzione della grammatica.

103. Come è fatto il NFA dei prefissi viabili? Come si ricava il DFA dei prefissi viabili, detto anche automa canonico LR(0)?

L'NFA dei prefissi viabili è costituito da:

- uno stato per ogni item LR(0) della grammatica aumentata
- $[S' \rightarrow S]$ è lo stato iniziale
- dallo stato $[A \rightarrow \alpha.X\beta]$ c'è una transizione allo stato $[A \rightarrow \alpha X.\beta]$ etichettata con X, per $X \in T \cup NT$
- dallo stato $[A \rightarrow \alpha.X\beta]$ per ogni produzione $X \rightarrow y$ c'è una epsilon transizione allo stato $[X \rightarrow .y]$
- non è necessario definire gli stati finali

Il DFA dei prefissi viabili si può ottenere per la costruzione per sottoinsiemi del NFA.

104. Come è fatta una tabella di parsing LR(0)? Come la si riempie a partire dall'automa canonico LR(0)? Quando una grammatica è di classe LR(0)?

Una tabella di parsing LR(0) è una matrice bidimensionale dove le righe sono indicizzate per gli **stati dell'automa** LR(0) e le colonne sono indicizzate per i **simboli (terminali e non)** della grammatica (più $).
L'elemento [s,X] della matrice contiene l'azione da eseguire quanto il parser LR si trova nello stato s e in cima alla pila è presente X.
La tabella si riempie nel seguente modo, per ogni stato s

- se $x \in T$ e c'è una transizione da s a t leggendo x nell'automa LR(0) allora inserisci shift t in M[s,x]
- se $A \rightarrow a. \in s$ e $A \neq S'$, inserisci REDUCE $A \rightarrow \alpha$ in $M[s,x]$ per tutti gli $x \in T \cup \{\$\}$
- se $S' \rightarrow S. \in s$ inserisci ACCEPT in M[s,$]
- se $A \in NT$ e c'è una transizione da s a t leggendo A nell'automa LR(0) allora inserisci GOTO t nell'elemento M[s,A]

Una grammatica è di classe LR(0) quando la tabella di parsing associata non presenta conflitti.

105. Come è fatto il parser LR(0) che utilizza la tabella di parsing LR(0)? Quanti stack servono?

Il parser LR(0) inizia dallo stato s_0.

1. legge l'input e toglie uno stato dalla pila.
2. Dalla tabella di parsing ottiene la azione da eseguire in M[stato che era sulla pila, carattere letto]
3. Se l'azione da eseguire è REDUCE $A \to a$:
    - fa il pop di |a| stati dalla pila
    - prende lo stato (chiamiamolo s_1) che rimane ora nella pila
    - guarda che GOTO c'è in $M[s_1, A]$
    - metto lo stato che che indica il GOTO sulla pila

Serve uno solo stack.

106. Esistono grammatiche non LR(0)? Fare un esempio semplice.

La seguente grammatica genera un conflitto shift/reduce:

$S' \to S \qquad S \to (S)S \qquad S \to \varepsilon$

107. Come è fatta e come si riempie una tabella di parsing SLR(1)? Cosa vuol dire l'acronimo SLR? Perché si mette 1 come parametro?

Si riempie in modo simile a LR(0) a parte le azioni REDUCE che vengono inserite solo in M[a,x] con $x \in T$ solo se $x \in Follow(A)$ con $A \to a. \in s$ e $A \neq S$

La S di SLR sta per *Simple*. Il parametro 1 indica un simbolo di look-ahead.

108. Quando una grammatica è di classe SLR(1)? Esistono grammatiche non di classe SLR(1)?

Una grammatica è di classe SLR(1) se nella tabella di parsing SLR(1) ha al più un elemento per ogni casella. Esistono grammatiche non di classe SRL(1) perchè $SLR(1) \subset LR(1)$

109. Cos'è un item LR(1)? Come si costruisce il NFA LR(1)? E come l'automa canonico LR(1)?

Un item LR(1) per una grammatica G è una coppia formata da un item LR(0) e un simbolo di lookahead $\in T \cup \$$

A differenza del automa canonico LR(0), nella costruzione del automa canonico LR(1) si inserisce un epsilon transizione dallo stato $[A \to \alpha.X\beta, c]$ allora stato $[X \to .y, b]$ per ogni produzione $X \to y$ e per ogni $b \in FIRST(\beta c) \cap T$

110. Come è fatta e come si riempie una tabella di parsing LR(1)?

Riempire una tabella di parsing LR(1)

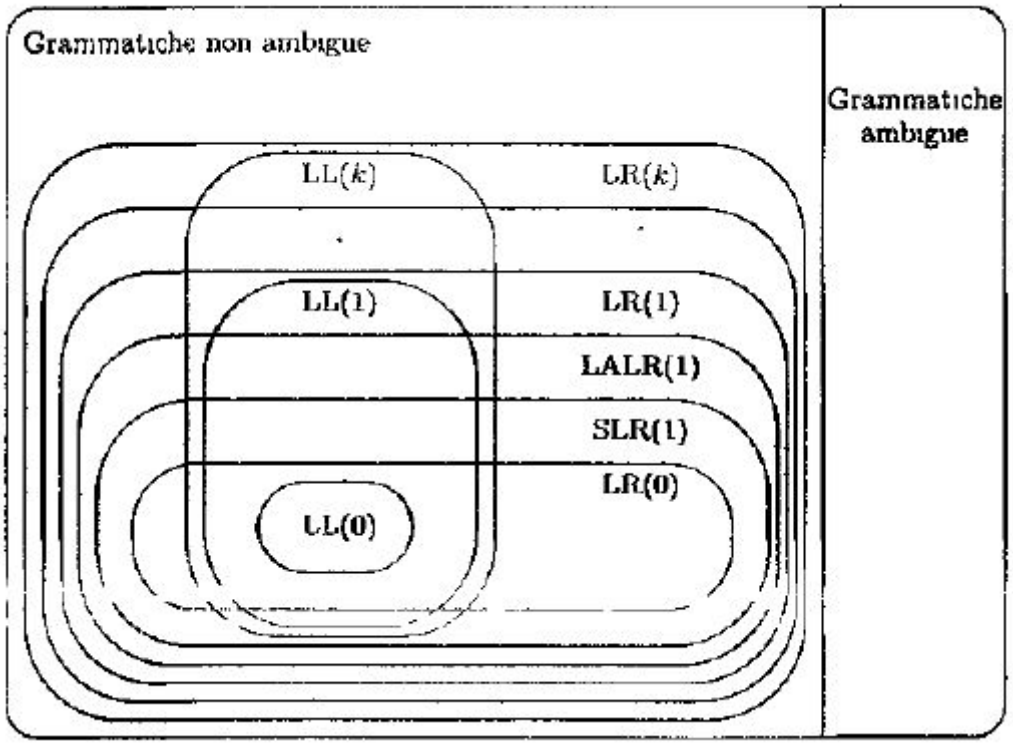
Per ogni stato s:

1. se $x \in T$ e $s \xrightarrow{x} t$ nell'automa LR(1), inserisci SHIFT t in $M[s, x]$;
2. se $[A \to \alpha., x] \in s$ e $A \neq S'$, inserisci REDUCE $A \to \alpha$ in $M[s, x]$;
3. se $[S' \to S., \$] \in s$, inserisci ACCEPT in $M[s, \$]$;
4. se $A \in NT$ e $s \xrightarrow{A} t$ nell'automa LR(1), inserisci GOTO t in $M[s, A]$.

111. Come è fatto una tabella di parsing LALR(1)? Quando una grammatica è di classe LALR(1)?

È riempita come la tabella LR(1) però a partire dall'automa LALR(1), che a sua volta si ottiene facendo l'unione insiemistica degli stati con lo stesso nucleo.

112. Esistono grammatiche LALR(1) che non sono SLR(1)? Esistono grammatiche LALR(1) che non sono LR(1)? Mostrare una grammatica che è LR(1) ma non LALR(1).
113. Come si può generalizzare l'idea per ogni k≥2? Ovvero quando una grammatica è di classe LR(k), SLR(k) o LALR(k)?
114. Come si relazionano le grammatiche della famiglia LR (LR, SLR, LALR) al variare di k? Come si relazionano le grammatiche LR(k) e LL(k) al variare di k? Le grammatiche LR(k) e LL(k) sono sempre non ambigue? Esistono grammatiche ambigue che sono LL(k) o LR(k) per qualche k? Esistono grammatiche che non sono LR(k) per nessun k?



115. Come si relazionano i linguaggi della famiglia LR(k) rispetto a quelli LL(k)? Esistono linguaggi liberi deterministici che non sono LR(k) per qualche k? Esistono linguaggi liberi deterministici che non sono LL(k) per qualche k?

116. Esiste un linguaggio regolare che non è LR(0)? Come si relazionano i linguaggi LR(0) rispetto a quelli LL(1)? La prefix property è una condizione necessaria affinchè un linguaggio sia di classe LR(0)?
117. La classe dei linguaggi SLR(1) coincide con la classe dei linguaggi liberi deterministici?

118. Cos'è YACC? Qual è il suo input e il suo output? Come si ottiene un parser eseguibile a partire da un file .y? Come agisce il parser generato da YACC in sintonia con lo scanner generato da Lex?
YACC è un generatori di analizzatori sintattici. Prende in input la descrizione di una grammatica e restituisce una programma C che è un parser LALR(1)
Per fare il LALR utilizza una costruzione diretta che non abbiamo visto a lezione.
119. Come è la struttura di un file .y di YACC? Cosa sono le regole? Cos'è l'azione semantica?
struttura di yacc
%{ prologo %} (opzionale, si possono mettere macro,variabili e funzioni per le regole)
definizioni dei simboli delle regole, lista dei token (opzionale)
%%
regole (coppie produzione-azione semantica). L'azione semantica è il codice che yacc esegue ogni volta che fa una reduce.
%%
funzioni ausiliarie (contiene le funzioni di supporto per la generazione del parser) deve contenere la funzione yylex() che invoca l'analizzatore lessicale.

120. È possibile gestire grammatiche ambigue con YACC, specificando le associatività e le priorità fra gli operatori per risolvere l'ambiguità? Come si comporta YACC in presenza di conflitti?

Capitolo 5 – Fondamenti

121. È possibile costruire un programma Check che, preso in input un qualunque programma P, restituisce 1 se P è corretto e 0 se P è scorretto? Ovvero esiste un qualche compilatore che può scovare tutti i possibili errori di un programma?
No, a causa di limiti intrinseci alla definizione di Macchina di Turing e algoritmo.

122. Cosa dice il problema della fermata (Halting Problem)? (L'errore in esame è la possibilità di non terminare il calcolo.) Come si dimostra che il problema non può essere risolto?
Esiste un programma H tale che, ricevuti in ingresso un programma P nel linguaggio L e un input x, termina stampando SI *se P(x) termina, e termina invece stampando* NO *se P(x) va in ciclo?*

1. Supponiamo di avere un programma H con le proprietà su esposte
2. Sfruttando H possiamo costruire un programma K con un solo input e le seguenti caratteristiche:
    a. K(P) stampa SI, se H(P,P) stampa NO
    b. K(P) va in ciclo se H(P,P) stampa SI
3. Eseguiamo ora K sul suo stesso testo, cioè interessiamoci a K(K)
    a. K(K) dovrebbe ritornare SI se K(K) non termina
    b. K(K) non termina se K(K) termina
4. Siamo evidentemente davanti all'assurdo che ovviamente non deriva da K che è un semplice programma ma da H che è un programma impossibile.

123. Quando un problema è decidibile?
Un problema è decidibile se e solo se esiste un programma che:
- funzioni per argomenti arbitrari
- termini sempre
- discrimini gli argomenti che sono soluzione del problema da quelli che non lo sono

124. Quali sono tipici esempi di proprietà indecidibili per i linguaggi di programmazione?
- Verificare se un programma calcola una funzione costante
- Verificare se due programmi calcolano la stessa funzione
- Verificare se un programma termina per ogni input
- Verificare se un programma diverge per ogni input
- Verificare se un programma causerà un errore di tipo

125. Cos'è una Macchina di Turing (MdT)? Che cosa calcola?
Una Macchina di Turing è un formalismo matematico inventato da Alan Turing alla fine degli anni 30.
Una macchina di Turing è costituita da un nastro infinito, diviso in celle ciascuna delle quali può memorizzare un unico simbolo appartenente ad un alfabeto finito. Sul nastro legge e scrive una testina magnetica che in ogni istante è posizionata su una cella.
Essa prende in input una descrizione di una macchina e un input codificati in un opportuno alfabeto ed esegue la computazione della macchina sull'input.

126. Quando un linguaggio è detto Turing--completo? Cosa afferma la tesi di Church--Turing e perché non si può dimostrare?
Un linguaggio è Turing-completo quando calcola l'insieme di funzioni calcolabili con una macchina di Turing.

Secondo la famosa tesi di Church (tuttora intatta) che afferma che l'insieme delle funzioni calcolabili dalla macchina di Turing corrisponde a quello delle funzioni intuitivamente calcolabili.

127. I normali linguaggi di programmazione sono Turing--completi? Cosa afferma il teorema di Jacopini--Bohm?
Si tutti i linguaggi di programmazione *general-purpose* sono Turing-completi.

Il teorema di Jacopini-Bohm afferma che qualsiasi programma con GO-TO è equivalente ad un programma senza.