Risposte

Domande e Risposte orale modulo 1 - capitolo 1

Ndr: le domande `evidenziate` sono quelle da sapere per passare l'esame (citando il prof: "se non sapete rispondervi non presentatevi nemmeno"), le altre domande non sono così determinanti ma saperle non fa male.

Capitolo 1 - Macchine astratte, interpreti, compilatori

1. `Che cosa è una macchina astratta? In cosa si differenzia da una macchina fisica?`

   È un'astrazione del concetto di calcolatore fisico. Una macchina fisica funziona esclusivamente per eseguire il proprio linguaggio. Una macchina fisica corrisponde ad un unico linguaggio.

   > Definizione di macchina astratta:
   >
   > Supponiamo che sia dato un linguaggio di programmazione L, definiamo una macchina astratta per L - indicandola con M_L - un qualsiasi insieme di strutture dati e algoritmi che permettano di memorizzare ed eseguire programmi scritti in L. Una generica macchina astratta M_L è composta da una **memoria** (divisa in memoria dati e memoria programma) e da un **interprete**.

2. `Che cos'è un interprete? In cosa consiste il ciclo fetch-decode-execute?`

   L'interprete è il componente essenziale di una macchina astratta. È il componente che **esegue il ciclo fetch-decode-execute** e interpreta le istruzioni. Esso è costituito da:

   - Operazioni per l'**elaborazione dei dati primitivi** (sulla macchina fisica la ALU);
   - Operazioni e strutture dati per il **controllo della sequenza** di esecuzione delle operazioni (su m.f. il PC);
   - Operazioni e strutture dati per il **controllo del trasferimento dei dati** (su m.f. gestione dei metodi di indirizzamento);
   - Operazioni e strutture dati per la **gestione della memoria** (su m.f. indirizzamento e trasferimento dei blocchi); La struttura di un interprete è la stessa per qualsiasi macchina astratta, ciò che cambia sono i componenti.

   Il ciclo fetch-decode-execute è alla base del funzionamento dei calcolatori basati sulla macchina di Von Neumann, che lo eseguono continuamente. Esso consiste in 3 fasi: nella fase di **fetch** il calcolatore carica dalla memoria le istruzioni del programma; nella fase **decode** il calcolatore decodifica e identifica il tipo di istruzioni da eseguire; infine nella fase **execute** vengono eseguite le istruzioni ricevute.

3. *Cos'è il linguaggio macchina?*

   Def. di **Linguaggio Macchina**: Data una macchina astratta M_L, il linguaggio L "compreso" dall'interprete di M_L è detto linguaggio macchina di M_L.

4. *Possono esistere macchine diverse con lo stesso linguaggio macchina?*

   Sì, poiché ad una macchina corrisponde un unico linguaggio, ma un linguaggio può essere eseguito da più macchine.

5. *In quali modi è possibile implementare una macchina astratta? Elenca vantaggi e svantaggi delle varie tecniche.*

   Una macchina astratta può essere implementata in tre modi:

   - **Hardware**: è sempre possibile realizzare M_L mediante hardware, implementando tutti i costrutti fisicamente (con memorie, porte logiche, bus, ecc).

     PRO: è l'approccio che garantisce le prestazioni migliori;

     CONTRO: una macchina simile è immutabile (una volta realizzata è impossibile da modificare), inoltre è via via più complessa man mano che il livello del linguaggio L è più alto.

   - **Emulazione firmware**: È la via di mezzo fra le tre implementazioni. Gli algoritmi e le strutture dati vengono simulati mediante **microprogrammi** che hanno le stesse possibilità dei programmi scritti in alto livello (vedi simulazione software) ma sono scritti in linguaggi di basso livello, mantenendo così un buon livello di prestazioni. Inoltre i microprogrammi risiedono in memorie di sola lettura (le ROM) per garantire un'alta velocità.

     PRO: prestazioni migliori, best of both worlds.

     CONTRO: flessibilità comunque ridotta in quanto le memorie ROM sono difficili da modificare (va fatto in laboratorio con raggi UV e cazzi vari, uno sbatti assurdo).

   - **Simulazione software**: le strutture dati e gli algoritmi vengono implementati mediante un linguaggio L' (che diamo per già implementato mediante una macchina $M'_{L'}$) di più basso livello rispetto al linguaggio L da implementare.

     PRO: massima flessibilità, possiamo cambiare in ogni momento le implementazioni dei costrutti.

     CONTRO: prestazioni peggiori, dal momento che dobbiamo passare da più livelli di interpretazione.

6. `Che cos'è un compilatore?`

Un compilatore da L a L_0 (che indichiamo con C_{L,L_0}) è un programma che realizza una funzione $C_{L,L_0} : Prog^L \leftrightarrow Prog^{L_0}$ tale che dato in input un programma scritto nel linguaggio L (linguaggio sorgente), produce un programma compilato scritto nel linguaggio L_0 (linguaggio oggetto) che potremo eseguire sulla macchina $M_{0_{L_0}}$.

7. `Descrivere la tecnica d'implementazione pura e quella compilativa pura.`

   L'approccio **interpretativo puro** consiste nel realizzare un programma, l'interprete, scritto nel linguaggio L_0 della macchina in cui dobbiamo eseguire il codice, che legge le istruzioni nel nostro codice scritto in linguaggio L e le esegue.

   I **pro** di un approccio interpretativo puro sono:

   - Non bisogna aspettare che il programma venga compilato;
   - È molto flessibile, è facile creare strumenti che interagiscano col programma a runtime ed è facile fare debugging;
   - Più semplice da realizzare rispetto ad un compiler;
   - Occupa meno memoria.

   I **contro** sono:

   - L'esecuzione è più lenta per via della decodifica in tempo reale;
   - La decodifica deve essere eseguita ogni volta. Tipico esempio: **Java**.

   L'approccio **compilativo puro** consiste nel tradurre un programma scritto nel linguaggio L (linguaggio sorgente) in un programmascritto nel linguaggio L_0 (linguaggio oggetto). La traduzione è affidata al compiler, indicato con C_{L,L_0}.

   I **pro** sono:

   - Approccio efficiente, il programma viene decodificato una volta sola e ogni esecuzione è più rapida;

   I **contro** sono:

   - È più difficile da implementare;
   - Poco flessibile (ogni modifica richiede la ricompilazione);
   - Perdita di informazioni sulla struttura del programma, quindi difficile debugging a runtime; Tipico linguaggio compilato: **C**.

8. `Quando un interprete si può dire corretto? Quando un compilatore si può dire corretto?`

   Sia un interprete che un compilatore si dicono corretti quando rispettano la semantica del linguaggio da interpretare/compilare.

9. *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 è la 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.

10. `Che cos'è la macchina intermedia? (Come vengono implementate nella realtà le macchine astratte?)`

    Una macchina intermedia viene usata per implementare una macchina astratta: fra la macchina M_L del linguaggio che vogliamo implementare e la macchina ospite $M_{0_{L_0}}$ esiste un livello caratterizzato da un proprio linguaggio L_i e la sua relativa macchina astratta $M_{i_{L_i}}$, che sono rispettivamente il linguaggio intermedio e la macchina intermedia.

11. `Quando si dice che una implementazione è di tipo interpretativo e quando di tipo compilativo?`

    Un'implementazione si dice di tipo **interpretativo** nel caso in cui la macchina intermedia sia effettivamente presente e l'interprete di questa sia diverso dall'interprete di $M_{0_{L_0}}$ (ovvero l'interprete della macchina fisica). Esempi: **LISP, ML, Perl, Postscript, Pascal, Prolog, Smalltalk, Java.**

    Un'implementazione è di tipo **compilativo** se la macchina intermedia è più vicina alla macchina ospite e ne condivide l'interprete. Esempi: **C, C++, FORTRAN, Pascal, ADA.** (Si Pascal c'è in entrambi, non è un errore).

12. *L'interprete e il compilatore si possono sempre realizzare?*

    L'esistenza dell'interprete e del compilatore è garantita a patto che il linguaggio L_0 che usiamo per l'implementazione sia sufficientemente espressivo

rispetto al linguaggio L che vogliamo implementare.

Praticamente questo accade sempre perché i linguaggi che usiamo (quelli di uso comune) sono tutti turing-completi.

13. *Cos'è l'implementazione via kernel?*

    L'implementazione via kernel è uno dei due modi per implementare un compilatore (l'altro è l'implementazione via bootstrapping). Per implementare L via kernel devo individuare al suo interno l'insieme minimale di primitive, tale insieme lo chiamo H, ed implemento il compilatore in H; implemento poi a mano un interprete o un compilatore per L.

    È il tipico approccio usato per realizzare i sistemi operativi: viene prima implementato il kernel e poi partendo da questo si implementa il resto del SO.

    In questo modo si semplifica l'implementazione di L e lo si rende facilmente portatile, dal momento che basta reimplementare di volta in volta solo il kernel H nel nuovo linguaggio macchina.

14. *Quando si parla di bootstrapping?*

    L'implementazione di tipo bootstrapping è quella usata ad esempio per il linguaggio pascal.

    In origine pascal era fornito di:

    - Un compilatore in Pascal, da Pascal a P-code: $C^{Pascal}_{Pascal,P-code}$;
    - Lo stesso compilatore, tradotto in P-code: $C^{P-code}_{Pascal,P-code}$;
    - Un interprete per P-code, scritto in Pascal: I^{Pascal}_{P-code};

    Per poter implementare il linguaggio su una specifica macchina M_0 si produce a mano una traduzione dell'interprete I^{Pascal}_{P-code} nel linguaggio M_0 ottenendo $I^{L_0}_{P-code}$.

    A questo punto è già possibile eseguire su M_0 un programma P in Pascal, ma per migliorare l'efficienza realizziamo a mano un compilatore scritto in M_0:

    - Produco C^{Pascal}_{Pascal,L_0}

    E adesso, con tutti gli strumenti realizzati, applico il bootstrapping:

    - $I^{L0}_{P-code}(C^{P-code}_{Pascal,P-code}, C^{Pascal}_{Pascal,L0}) = C^{P-code}_{Pascal,L0}$
    - $I^{L0}_{P-code}(C^{P-code}_{Pascal,L0}, C^{Pascal}_{Pascal,L0}) = C^{L0}_{Pascal,L0}$

    Ovvero do in input all'interprete realizzato a mano i due compilatori (quello tradotto in P-code di prima e quello scritto a mano poco fa) e ottendo un compilatore scritto in P-code da Pascal a L_0. Infine riutilizzo lo stesso compilatore dandogli questa volta in input il compilatore appena prodotto

e quello che avevamo scritto a mano prima, ottendo in output il compilatore finale da Pascal a L_0 e scritto in L_0 per l'appunto.

5. In quali modi è possibile implementare una macchina astratta? Elencare vantaggi e svantaggi delle varie tecniche.

Possiamo realizzare una MA:

1) realizzazione in HARDWARE. TEORICAMENTE SEMPRE POSSIBILE MA → USATA SOLO PER MACCHINE DI BASSO LIVELLO O MACCHINE DEDICATE; → MASSIMA VELOCITÀ; → FLESSIBILITÀ NULLA

2) emulazione o simulazione via Firmware. STRUTTURE DATI E ALGORITMI MA REALIZZATI MEDIANTE MICROPROGRAMMI, CHE RISIEDONO IN UNA MEMORIA DI SOLA LETTURA → MACCHINA OSPITE (FISICA) MICROPROGRAMMABILE; → ALTA VELOCITÀ; → FLESSIBILITÀ MAGGIORE CHE HW PURO

3) interpretazione o simulazione via SOFTWARE. STRUTTURE DATI E ALGORITMI DELLA MACCHINA ASTRATTA MA REALIZZATI MEDIANTE *programmi scritti nel linguaggio della macchina ospite MO.* → macchina ospite qualsiasi; → minore velocità; → massima flessibilità

6. Che cos'è un compilatore?

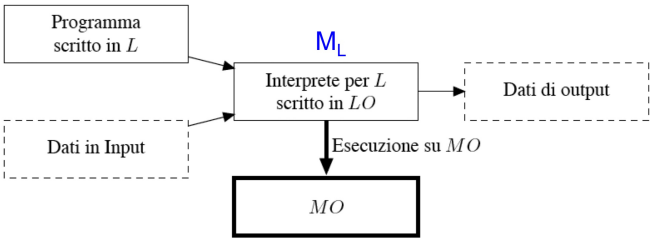
Il compilatore è un programma informatico che traduce una serie di istruzioni scritte in un determinato linguaggio di programmazione in istruzioni di un altro linguaggio. Il compilatore preserva la semantica del programma, ovvero programmi che producono lo stesso risultato se sottoposti agli stessi DATI IN INGRESSO

7. Relativamente alla tecnica d'implementazione software, **descrivere la tecnica d'implementazione interpretativa pura e quella compilativa pura.**

Implementazione Interpretativa pura (macchina ospite, con il suo ling. Lo)

M_L è realizzata scrivendo un interprete per L su M_{Lo}.

cioè realizzare una macchina astratta ← ling. da implementare



In altre parole:

Definizione 1.3 (Interprete) *Un interprete per il linguaggio $\mathcal{L}$, scritto nel linguaggio $\mathcal{L}o$, è un programma che realizza una funzione parziale*

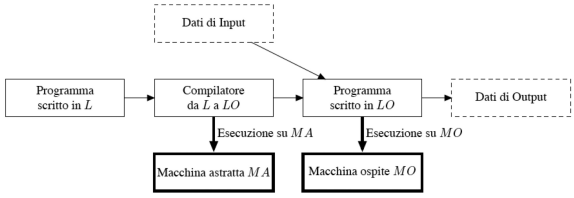
$$\mathcal{I}_{\mathcal{L}}^{\mathcal{L}o} : (Prog^{\mathcal{L}} \times \mathcal{D}) \to \mathcal{D} \quad tale\ che \quad \mathcal{I}_{\mathcal{L}}^{\mathcal{L}o}(\mathcal{P}_r^{\mathcal{L}}, Input) = \mathcal{P}^{\mathcal{L}}(Input). \quad (1.1)$$

Ovvero l'interprete calcola correttamente la semantica.

Implementazione compilativa pura

I programmi in $\mathcal{L}$ sono Tradotti in programmi equivalenti in $\mathcal{L}o$. Avviene la Traduzione effettuata da un (altro) progr. $C^{\mathcal{L}a}_{\mathcal{L},\mathcal{L}o}$

Il compilatore da $\mathcal{L}$ a $\mathcal{L}o$ scritto in $\mathcal{L}a$.



In altre parole:

Definizione 1.4 (Compilatore) *Un compilatore da $\mathcal{L}$ a $\mathcal{L}o$ è un programma che realizza una funzione*

$$\mathcal{C}_{\mathcal{L},\mathcal{L}o} : Prog^{\mathcal{L}} \to Prog^{\mathcal{L}o}$$

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

$$\mathcal{C}_{\mathcal{L},\mathcal{L}o}(\mathcal{P}_r^{\mathcal{L}}) = \mathcal{P}c^{\mathcal{L}o} \quad (1.2)$$

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

$$\mathcal{P}^{\mathcal{L}}(Input) = \mathcal{P}c^{\mathcal{L}o}(Input). \quad (1.3)$$

Ovvero il compilatore preserva la semantica.

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

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

Implementazione interpretativa pura:

- scarsa efficienza della macchina M_L

Ai tempi di esecuzione, vanno aggiunti i tempi necessari alla decodifica.

- buona flessibilità: permette di interagire con l'esecuzione del programma
- Più facile da realizzare
- Occupa meno memoria, perchè non viene effettivamente generato codice da memorizzare

Implementazione Compilativa, la Traduzione viene fatta prima di eseguire

- difficile, data la lontananza fra $\mathcal{L}$ e $\mathcal{L}o$.
- buona efficienza:
  1) costo decodifica a carico del compilatore
  2) Ogni istruzione è Tradotta una sola volta
- scarsa flessibilità
- perdita di info sulla struttura (astrazione) del programma sorgente.
- occupazione di memoria del codice prodotto.

10. Come vengono implementate nella realtà le macchine astratte? **Che cos'è la macchina intermedia?**

In una macchina astratta ci sono due componenti che coesistono:

1) Alcune istruzioni (Ingresso/uscita) sono sempre simulate

2) I programmi devono essere Tradotti nella rappresentazione interna o in un codice intermedio

- ***L'interprete:*** **un componente essenziale della macchina astratta che ne caratterizza il comportamento, mettendo in relazione "operazionale" il linguaggio della macchina astratta col mondo fisico circostante.**

CAPITOLO 2

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

La descrizione di un linguaggi avviene su 3 dimensioni:

- SINTASSI → è una relazione Tra segni: Tra tutte le sequenze di possibili parole, ne seleziona un sottoinsieme che costituisce le frasi del linguaggio stesso.
- SEMANTICA → attribuisce un significato a ogni frase corretta. Tratta la relazione Tra segni e significato.
- PRAGMANTICA → relazione Tra segni, significato e utente. In quale modo frasi corrette e sensate sono usate.

Per un linguaggio eseguibile:

- IMPLEMENTAZIONE → come eseguire una frase corretta rispetto la semantica

16. Sintassi: qual è l'aspetto lessicale e quale quello grammaticale di un linguaggio?

- Aspetto lessicale → elenchiamo le parole che si possono usare
  descrizione del lessico: . Dizionari (LING. NAT.)
  . Strutture più complesse
- Aspetto Grammatica → frasi corrette che si possono costruire con il lessico
  - DESCRIZIONE ATTRAVERSO REGOLE GRAMMATICALI (in numero finito)
  - LE FRASI GENERABILI SONO INFINITE

17. Cos'è un alfabeto? Cos'è una parola o stringa? **Cos'è A*?** Tale insieme è enumerabile?

Alfabeto A è un insieme finito di elementi detti simboli.

Una parola su alfabeto A è una sequenza finita di simboli in A.

A^* è un insieme infinito contabile. (Sarebbe numerabile anche se A fosse infinito) → perché ha una corrispondenza bionivoca $\mathbb{N}$

$A^* = \bigcup_{m \geq 0} A^m \qquad A^0 = \{\varepsilon\}$

18. Definizione di potenza di una stringa. Definizione di potenza di un linguaggio. Definizione di chiusura/iterazione (o stella di Kleene) di un linguaggio.

POTENZA DI UNA STRINGA

$w^0 = \varepsilon$

$w^{n+1} = w^n w \quad \forall m \geq 0$

POTENZA DI UN LINGUAGGIO

$L^0 = \{\varepsilon\}$

$L^{m+1} = L^m \cdot L \quad \forall m \geq 0$

CHIUSURA / STELLA DI KLEENE / RIPETIZIONE DI UN LINGUAGGIO

$L^* = \bigcup_{m \geq 0} L^m$

$L^+ = \bigcup_{m \geq 1} L^m$ chiusura positiva

19. **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 un insieme finito di simboli non-Terminali
- T è un insieme finito di simboli Terminali
- $S \in NT$ è detto simbolo iniziale.
- R è un insieme finito di produzioni (o regole) della forma $V \to w$ dove $V \in NT$ e $w \in (T \cup NT)^*$

Una stringa si può derivare in un passo ($\Rightarrow$) o in più passi ($\Rightarrow^*$).

$$\frac{v = xAz \quad (A \to y) \in R \quad w = xyz}{v \Rightarrow w} \qquad \frac{v \Rightarrow^* w \quad w \Rightarrow^* z}{v \Rightarrow^* z}$$

CHIUSURA TRANSITIVA

$v \Rightarrow^* w$ → dirsi deriva w se $\exists$ una sequenza finita di derivazione immediata

$v \Rightarrow^* v$ → o eventualmente vuota — CHIUSURA RIFLESSIVA

Il linguaggio generato da una grammatica $G = (NT, T, S, R)$ è l'insieme $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$

20. **Cos'è un albero di derivazione?** Cos'è una derivazione canonica sinistra/destra? Esiste una corrispondenza biunivoca tra alberi di derivazioni e derivazioni canoniche?

(1) 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
- se il nodo n:
  - ha etichetta $A \in NT$
  - e
  - i suoi figli sono nell'ordine $n_1,\dots,n_k$ con etichetta $x_1,\dots,x_k$ (in $NT \cup T$), allora $A \to x_1,\dots,x_k$ è una produzione in R.
- se il nodo n ha etichetta ε, allora n è una foglia, è figlio unico e, detto A suo padre, $A \to \varepsilon$ è una produzione di R.
- se inoltre ogni nodo figlio è etichettato su $T \cup \{\varepsilon\}$, allora l'albero corrisponde ad una derivazione completa.

(2) Esistono due tipi di derivazioni:

LEFTMOST: ad ogni passo riscriviamo il NT più a Sx

RIGHTMOST: ad ogni passo riscriviamo il NT più a dx

Esse generano lo stesso albero di derivazione

(3) Un albero di derivazione "riassume" tante derivazioni diverse, ma tutte equivalenti.

Esiste una corrispondenza biunivoca tra derivazioni canoniche e alberi di derivazione.

Nel senso che:
- dato un albero di derivazione, $\exists!$ derivazione LEFTMOST che lo genera
- dato un albero di derivazione, $\exists!$ derivazione RIGHTMOST che lo genera
- data una derivazione LEFTMOST (o RIGHTMOST), ad essa viene associata univocamente un albero di derivazione.

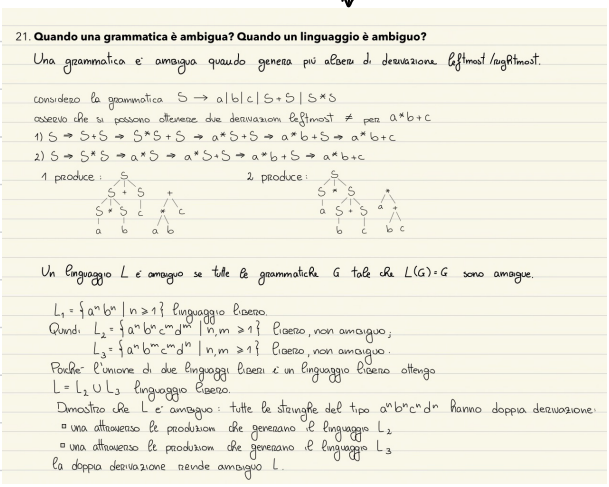
21. **Quando una grammatica è ambigua? (fare un esempio) Quando un linguaggio è ambiguo? (fare un esempio)**

Una grammatica libera G è ambigua se $\exists\, w \in L(G)$ che ammette più alberi di derivazione.

ESEMPIO: $G=(\{S\},\{a,b,+,*\},S,R)$ con $R=\{S\to a,\ S\to b,\ S\to S+S,\ S\to S*S\}$ FORMA COMPATTA $S\to a|b|S+S|S*S$

Un linguaggio L è ambiguo se tutte le grammatiche G, t.c. $L(G)=L$ sono ambigue.

ESEMPIO $L=\{a^n b^n \mid n\geq 1\}$ linguaggio libero


21. Quando una grammatica è ambigua? Quando un linguaggio è ambiguo?
Una grammatica è ambigua quando genera più alberi di derivazione leftmost / rightmost.
considero la grammatica $S \to a|b|c|S+S|S*S$
osservo che si possono ottenere due derivazioni leftmost $\neq$ per $a*b+c$
1) $S \Rightarrow S+S \Rightarrow S*S+S \Rightarrow a*S+S \Rightarrow a*b+S \Rightarrow a*b+c$
2) $S \Rightarrow S*S \Rightarrow a*S \Rightarrow a*S+S \Rightarrow a*b+S \Rightarrow a*b+c$
1 produce: 2 produce:
Un linguaggio L è ambiguo se tutte le grammatiche G tale che $L(G)=G$ sono ambigue.
$L_1=\{a^n b^n \mid n\geq 1\}$ linguaggio libero.
Quindi $L_2=\{a^n b^n c^m d^m \mid n,m \geq 1\}$ libero, non ambiguo;
$L_3=\{a^n b^m c^m d^n \mid n,m \geq 1\}$ libero, non ambiguo.
Poiché l'unione di due linguaggi liberi è un linguaggio libero ottengo
$L=L_2 \cup L_3$ linguaggio libero.
Dimostro che L è ambiguo: tutte le stringhe del tipo $a^n b^n c^n d^n$ hanno doppia derivazione:
- una attraverso le produzioni che generano il linguaggio L_2
- una attraverso le produzioni che generano il linguaggio L_3

la doppia derivazione rende ambiguo L.

22. È possibile rimuovere l'ambiguità dalla grammatica delle espressioni aritmetiche? Come?

Sì è possibile rimuovere l'ambiguità dalla grammatica delle espressioni aritmetiche:

1. Dando precedenza all'operatore
2. Scegliendo il tipo di associatività $\angle^{dx}_{Sx}$

Per disambiguare si può usare lo ZUCCHERO SINTATTICO.

23. **Cos'è l'albero di sintassi astratta?** Che differenza c'è tra sintassi concreta e sintassi astratta? Cos'è lo zucchero sintattico?

Un albero di sintassi astratta è un albero composto solo dai T. È molto semplice e intuitiva, ma ambigua.

Un albero di sintassi concreta è ottenuto da una grammatica non ambigua che fa uso di zucchero sintattico → serve per disambiguare → Sono le "(" e ")".

Dall'albero di derivazione da SINTASSI CONCRETA, estraendo un albero sintattico da SINTASSI ASTRATTA, detto albero SINTATTICO ASTRATTO.

24. **Fare esempi di vincoli sintattici contestuali.** Possono essere catturati attraverso grammatiche libere?

ESEMPI:
- una variabile in uso deve prima essere dichiarata
- compatibilità di tipo in un assegnamento $\rightsquigarrow$ $x := e$ "x" e "e" (espressione) devono essere dello stesso tipo
- il numero (e il tipo) dei parametri attuali di una chiamata di procedura deve essere uguale al numero (e al tipo) dei parametri formali della dichiarazione.

Questi sono vincoli Sintattici, ma non esprimibili per mezzo di grammatiche libere (o BNF). per questo motivo non sono in grado di descrivere vincoli che dipendono dal contesto.

Abbiamo due possibili soluzioni:

1) Usare grammatiche dipendenti dal contesto → pocco pratico

2) Usare controlli "ad hoc" → controlli sintattici eseguiti durante la fase "Analisi Semantica"

25. Cosa s'intende per semantica statica? E per semantica dinamica?

SEMANTICA STATICA → insieme dei controlli che possono essere fatti sul testo del programma, SENZA ESEGUIRLO. (ad esempio il TYPE CHECKING)

↓

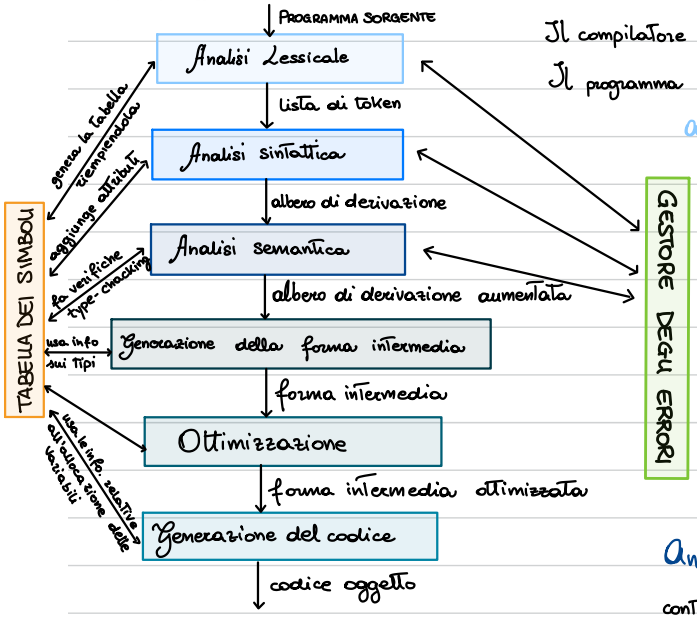
Tutto quello che non si può esprimere tramite BNF

SEMANTICA DINAMICA → si intende una rappresentazione formale dell'esecuzione del programma, la quale può mostrare errori "dinamici". [la divisione per 0]

↓

si intende: fornisce un modello matematico che descriva il "comportamento" del programma → indipendente dall'architettura in cui viene eseguito il prog.

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



Il compilatore è un programma che prende in input un programma sorgente.

Il programma ha le seguenti fasi:

Analisi Lessicale (SCANNER) spezza il programma sorgente nei componenti sintattici primitivi (TOKEN) → possono essere: identificatori, numeri, operatori, parentesi, ...

⇝ controlla solo che il "lessico" sia ammissibile

⇝ riempe parzialmente la tabella dei simboli

Analisi Sintattica (Parser) a partire dalla lista di token, generata dallo Scanner, il parser produce l'albero di derivazione del programma, riconoscendo se le fasi sono sintatticamente corrette.

Consulta la tabella dei simboli e aggiunge dei attributi. Inoltre interagisce insieme all'analisi sintattica, con gestore degli errori.

Analisi Semantica esegue dei controlli di semantica statica (ovvero sintattici contestuali) per risolvere eventuali errori "semantici".

produce in output l'albero di derivazione aumentato ← - arricchisce l'albero di derivazione con informazioni sul tipo.

- verifica i tipi negli assegnamenti, parametri attuali vs formali, dichiarazioni e uso di variabili.
- genera eventualmente opportuni messaggi d'errore.

Gestione degli errori: Ogni errore relativo (nelle prime 3 fasi) non blocca il compilatore, ma genera un opportuno messaggio d'errore.

Generazione della forma intermedia: genera codice scritto in un linguaggio intermedio, indipendentemente dall'architettura, facilmente traducibile nel linguaggio macchina di varie macchine diverse:

- utilizza operazioni molto semplici, tipicamente "THREE-ADDRESS CODE" → è ridondante e poco efficiente, per questo
- nel generatore codice intermedio, si segue la struttura dell'albero sintattico, ricavato dall'albero di derivazione.

Ottimizzazione: si effettuano ottimizzazione sul codice intermedio per renderlo più efficiente.

- RIMOZIONE DEL CODICE INUTILE (dead code)
- ESPRESSIONE IN LINEA DI CHIAMATE DI FUNZIONI
- FATTORIZZAZIONE DI SOTTOESPRESSIONI
- METTERE FUORI DAI CICLI SOTTOESPRESSIONI CHE NON VARIANO

Alla fine si ottiene un codice intermedio ottimizzato.

Generazione del codice: viene generato codice per una specifica architettura (include anche l'assegnazione dei registri e ottimizzazione specifiche per quell'architettura)

In tutte queste fasi viene usata la Tabella dei Simboli: memorizza le informazioni sui nomi presenti nel programma (identificatori di valori, funzioni, procedure, ...)

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

La SEMANTICA DINAMICA serve:

- AL PROGRAMMATORE: analisi del programma → deve sapere esattamente cosa debba fare il suo programma
  → deve poter dimostrare proprietà del suo programma.
- AL PROGETTISTA DEL LINGUAGGIO → strumenti di specifica del linguaggio
  → deve poter dimostrare proprietà del linguaggio stesso
- ALL'IMPLEMENTATORE DEL LINGUAGGIO → riferimento per dimostrare la correttezza dell'implementazione.

28. Quali tecniche si usano per dare semantica ad un linguaggio di programmazione?

Esistono due tecniche per dare semantica:

- OPERAZIONALE → si costruisce una specie di automa che, passo a passo, mostra l'effetto dell'esecuzione delle varie istruzioni. Mette enfasi su come si calcola.
- DENOTAZIONALE → si associa ad ogni programma sequenziale una funzione da input a output. Mette enfasi su cosa si calcola. (vengono nascosti i passi intermedi del calcolo)

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? Cosa si intende per implementazione di un linguaggio?

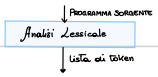
La pragmatica è un insieme di "regole"/consigli sul modo in cui è meglio usare le istruzioni a disposizione. (evitare le istruzioni di salto quando possibile, ecc...)

L'implementazione, significa scrivere un compilatore (o un interprete) per una macchina ospite già realizzata, costruendo così una macchina astratta per il linguaggio. Bisogna considerare due aspetti:

- CORRETTEZZA → bisogna dimostrare che preserva la semantica
- COSTO → che il compilatore è in grado di produrre codice efficiente.

CAPITOLO 3

31. Cosa fa l'analizzatore lessicale?



Riconosce nella stringa in ingresso gruppi/sequenze di simboli che corrispondono a specifiche categorie sintattiche. La stringa in input è trasformata in una sequenza di simboli astratti, detti Token.

32. Cos'è un token?

Il token è una coppia: (nome, valore)

simbolo astratto che rappresenta → una categoria sintattica. È l'informazione che identifica una classe di token.

→ una sequenza di simboli del testo in ingresso. È l'informazione che identifica uno specifico token

33. Cos'è un pattern? Come lo si rappresenta? Cos'è un lessema?

Pattern è la descrizione generale della forma dei valori di una classe di token. Lo si rappresenta con un'espressione regolare.

Un Lessema è una stringa istanza di un pattern.

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

DEFINIZIONE DI ESPRESSIONI REGOLARI: Fissato un alfabeto $A = \{a_1, \dots, a_n\}$ definiamo le espressioni regolari su A con la seguente BNF

$$r := \emptyset \mid \varepsilon \mid a \mid r \cdot r \mid r \mid r \mid r^*$$

$\mathcal{L}[\emptyset] = \emptyset$ $\quad$ $\mathcal{L}[r \cdot r] = \mathcal{L}[r] \cdot \mathcal{L}[r]$

$\mathcal{L}[\varepsilon] = \{\varepsilon\}$ $\quad$ $\mathcal{L}[r \mid r] = \mathcal{L}[r] \cup \mathcal{L}[r]$

$\mathcal{L}[a] = \{a\}$ $\quad$ $\mathcal{L}[r^*] = (\mathcal{L}[r])^*$

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

Un linguaggio $L \subseteq A^*$ è detto regolare $\iff \exists$ una espressione regolare r t.c. $L = \mathcal{L}[r]$

Ogni linguaggio finito è regolare. $L = \{a, bc\}$ $r = a \mid bc$.

Esistono anche linguaggi regolari infiniti $\mathcal{L}[a^*b]$ → Sono esempi che vanno dimostrati

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

Due espressioni regolari r e s sono equivalenti $\iff \mathcal{L}[r] = \mathcal{L}[s]$ (cioè denotano lo stesso linguaggio) e lo denotiamo come $r \simeq s$

Alcune legge di equivalenza:

$r \mid s \simeq s \mid r$ | è commutativa

$r \mid (s \mid t) \simeq (r \mid s) \mid t$ | è associativa

$r \mid r \simeq r$ | è idempotente

$r \cdot (s \cdot t) \simeq (r \mid s) \cdot t$ · è associativo

$(r^*)^* \simeq r^*$ * è idempotente

$\varepsilon \cdot r \simeq r \simeq r \cdot \varepsilon$ ε è l'elemento neutro per ·

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

Una definizione regolare su l'alfabeto A è costituita da una lista di definizioni $d_1 = r_1 \dots d_k = r_k$, dove i valori di sono simboli "nuovi" e ogni r_i è un'espressione regolare sull'alfabeto esteso $A \cup \{d_1, \dots, d_k\}$

Le espressioni regolari servono per specificare il pattern di una categoria sintattica, ovvero la forma dei possibili lessemi.

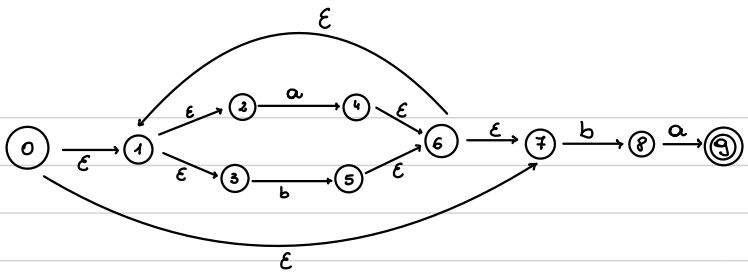
38. **Definizione di NFA** (automa finito nondeterministico). Discutere cosa si intende per nondeterminismo. Mettere in relazione la definizione formale con la rappresentazione grafica come diagramma di transizioni.

Un automa finito non deterministico (NFA) è una quintupla $(\Sigma, Q, \delta, q_0, F)$ dove

- Σ è un alfabeto finito di simboli in input
- Q è un insieme finito di stati
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali
- δ è la funzione di transizione con tipo $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ (insieme delle parti di Q.) $(\delta(q, \sigma) = Q' \subseteq Q)$

È nondeterminismo per 2 motivi: _ l'insieme di stati che raggiungono $\mathcal{P}(Q)$ non è semplicemente uno solo, ma è un insieme.

_ ho anche le transizioni etichettate $\{\varepsilon\}$ → rappresentano le transizioni dove l'automa cambia di stato senza leggere nessun carattere in input



39. **Come si definisce il linguaggio accettato da un NFA?** Definizione di equivalenza tra NFA.

Un NFA $N=(\Sigma, Q, \delta, q_0, F)$ accetta $w=a_1 \dots a_n$ $\Leftrightarrow$ nel diagramma di transizione $\exists$ un cammino da q_0 ad uno stato in F nel quale la stringa che si ottiene concatenando le etichette degli archi percorsi è esattamente w.

Due NFA N_1 e N_2 si dicono equivalenti $\Leftrightarrow$ accettano lo stesso linguaggio, cioè se $L[N_1]=L[N_2]$

40. **Definizione di DFA (automa finito deterministico).** Discutere cosa si intende 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 automa finito deterministico (DFA) è una quintupla $(\Sigma, Q, \delta, q_0, F)$ dove:

- Σ è un alfabeto finito di simboli in input
- Q è un insieme finito di stati
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali
- δ è la funzione di transizione con tipo $\delta : Q \times \Sigma \to Q$ $(\delta(q, \sigma) = q')$

Determinismo → si ha una sola mossa possibile $\delta(q, \sigma)$, quindi non ci sono le mosse ε.

Un DFA è un particolare tipo di NFA tale che:

- $\forall q \in Q \quad \delta(q, \varepsilon) = \emptyset$
- $\forall \sigma \in \Sigma, \forall q \in Q \ \exists q' \in Q \ . \ \delta(q, \sigma) = \{q'\}$

Vogliamo ora dimostrare che i DFA sono tanto espressivi quanto gli NFA, sebbene siano un sottoinsieme proprio degli NFA.

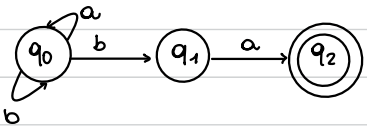
41. Dato un NFA, come si ricava un equivalente DFA? Ovvero **descrivere come è definita la costruzione per sottoinsiemi**. Definizione di epsilon-closure e algoritmo associato. Qual è la complessità della costruzione per sottoinsiemi nel caso pessimo? Ovvero se NFA ha n stati, quanti stati può avere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

1) Usiamo la proposizione che dice: $\forall$ NFA, è possibile costruire un DFA ad esso equivalente. Successivamente usiamo la tecnica della COSTRUZIONE PER SOTTOINSIEMI.

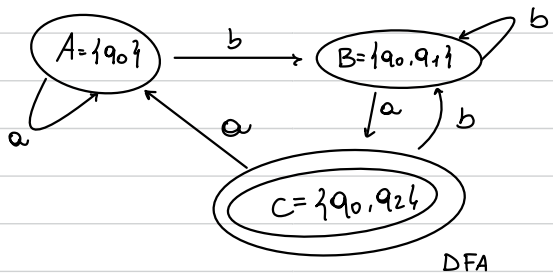
2) COSTRUZIONE DEI SOTTOINSIEMI:

Dato un NFA $N=(\Sigma, Q, \delta, q_0, F)$:

NFA

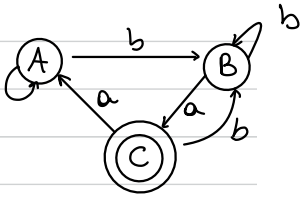


- Inizializza $S = \varepsilon\text{-closure}(q_0)$; // S stato iniziale del DFA → stato iniziale del DFA : $\{q_0\} = A$
- Inizializza $T = \{S\}$ // T è l'insieme degli stati del DFA
  // S non è marcato all'inizio



- Finché c'è un PET non marcato {
  - marca P;
  - for each $a \in \Sigma$ {
    - $R = \varepsilon\text{-closure}(\text{mossa}(P, a))$
    - if $R \notin T$ {
      add R to T; // R non ha marca
    }

DFA



_ definisci $\Delta(P,a)=R$;

}

}

$(\Sigma, T, \Delta, \varepsilon\text{-closure}(q_0), \mathcal{F})$ dove $R \in \mathcal{F} \Leftrightarrow \exists q \in R$ con $q \in F$ (finale DFA; finali NFA)

5) Nel caso pessimo $T=\mathcal{P}(Q)$, cioè il DFA M_N costruito a partire dall'NFA N, ha un numero di stati pari a 2^m, dove $m=|Q|$

3) ε- closure e mossa: Sia q uno stato di un NFA. La ε-closure di q è l'insieme degli stati raggiungibili da q solo con mosse ε.

$$\frac{}{q \in \varepsilon\text{-closure}(q)} \qquad \frac{p \in \varepsilon\text{-closure}(q)}{\delta(p,\varepsilon) \subseteq \varepsilon\text{-closure}(q)}$$

Sia P un insieme di stati di un NFA.

ε- closure $(P) = \bigcup_{p \in P} \varepsilon$-closure (p)

mossa: $\mathcal{P}(Q) \times \Sigma \longrightarrow \mathcal{P}(Q)$

mossa $(P,a) = \bigcup_{p \in P} \delta(p,a)$

$\Delta(A,b) =$ ε-closure (mossa (A,b))

4) ALGORITMO PER CALCOLARE LA ε-closure (P)

Inizializzare $T=P$;

Inizializzare ε-closure $(p)=P$.

while $T=\emptyset$ do {

"scegli un $r \in T$ e rimuovilo da T"

for each $s \in \delta(r,\varepsilon)$ do

if $s \notin \varepsilon$-closure (p) {

add S to ε-closure (p);

add S to T;

}

}

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.**

Teorema: Sia $N=(\Sigma, Q, \delta, q_0, F)$ un NFA e sia M_N l'automa ottenuto con la costruzione per sottoinsiemi. Allora M_N è un DFA e si ha che $L[N] = L[M_N]$

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?

Data una espressione regolare S, possiamo costruire un NFA $N[S]$ tale che $\mathcal{L}[S] = L[N[S]]$

(ovvero gli NFA riconoscono tutti e soli i linguaggi regolari)

Dimostriamo per induzione sulla sintassi (astratta) della espressione regolare S.

Costruiamo $N[S]$ (automa), cioè un possibile NFA associato all'espressione regolare S, in modo da mantenere i seguenti due invarianti:

1) lo stato iniziale non ha archi entranti

2) $N[S]$ ha un solo stato finale senza archi uscenti

$S=(a|b)^*ba$ grammatica regolare

possiamo costruire l'albero sintattico

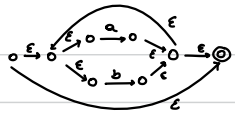
Partiamo dalle foglie e risaliamo alla radice $N[a]$ $\circ \xrightarrow{a} \circ\!\!\circ$

$N[b]$ $\circ \xrightarrow{b} \circ\!\!\circ$

ESAMINIAMO I VARI CASI:

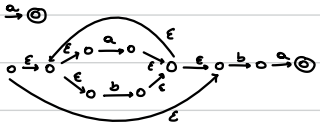
- $S=\emptyset$ $N[S]=$ $\longrightarrow\circ$ $\quad\circ\!\!\circ$ (due stati non connessi) Osserva che $\mathcal{L}[\emptyset]=\emptyset=L[N[S]]$
- $S=\varepsilon$ $N[S]=$ $\longrightarrow\circ \xrightarrow{\varepsilon} \circ\!\!\circ$ Osserva $\mathcal{L}[\varepsilon]=\{\varepsilon\}=L[N[S]]$
- $S=a$ $N[S]=$ $\longrightarrow\circ \xrightarrow{a} \circ\!\!\circ$ Osserva $\mathcal{L}[a]=\{a\}=L[N[S]]$

N[a|b]

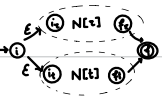


N[ba]

N[S]

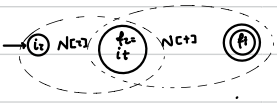


- $S = r|t$ $\quad N[S] =$



Osserva $\mathcal{L}[r|t] = \mathcal{L}[r] \cup \mathcal{L}[t] \overset{\text{per ipotesi induttiva}}{=} L[N[r]] \cup L[N[t]] = L[N[r|t]]$

- $S = r \cdot t$ $\quad N[S] =$

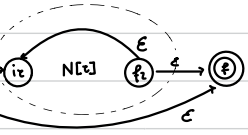


Abbiamo fuso insieme il finale di $N[r]$ con l'iniziale di $N[t]$

per ipotesi induttiva

Osserva che $\mathcal{L}[r \cdot t] = \mathcal{L}[r] \cdot \mathcal{L}[t] \overset{\text{per ipotesi induttiva}}{=} L[N[r]] \cdot L[N[t]] = L[N[r \cdot t]]$

- $S = r^*$ $\quad N[S] =$



Osserva che $\mathcal{L}[r^*] = (\mathcal{L}[r^*]) \overset{\text{per ipotesi induttiva}}{=} (L[N[r]])^* = L[N[r^*]]$

44. Definizione di grammatica regolare.

Una grammatica libera è regolare $\Leftrightarrow$ ogni produzione è della forma $V \to aW$ oppure $V \to a$ dove $V, W \in NT$ e $a \in T$. Per il simbolo iniziale S è ammessa anche la produzione $S \to \varepsilon$.

(N.B: A volte useremo una definizione più lasca che permette produzioni $V \to \varepsilon$ anche per NT diversi da S)

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

C'è un Teorema che afferma ciò: Data una grammatica regolare G si può costruire un NFA N_G equivalente.

Dimostriamo Sia $G = (NT, T, R, S)$, allora $N_G = (T, Q, \delta, S, \{\varepsilon\})$ è definita come segue

- $Q = NT \cup \{\varepsilon\}$

($F = \{\varepsilon\}$ • $q_0 = S$)

- δ è definita come:
  - $Z \in \delta(V, a)$ se $V \to aZ \in R$
  - $\varepsilon \in \delta(V, a)$ se $V \to a \in R$
  - $\varepsilon \in \delta(S, \varepsilon)$ se $S \to \varepsilon \in R$

Si può dimostrare che

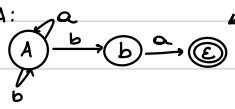
$S \overset{*}{\Rightarrow}_G W$ (con la grammatica G)

sse $(S, W) \overset{*}{\vdash}_{N_G} (\varepsilon, \varepsilon)$ (con l'automa N_G)

$A \to aA \mid bB \mid bA$] G è regolare

$B \to a$] $L(G) = (a|b)^* ba$

NFA:



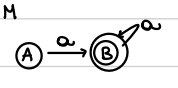
46. Dato un DFA, come si costruisce una grammatica regolare equivalente?

Dato un DFA M, possiamo definire una grammatica regolare G_M tale che $L[M] = \mathcal{L}(G_M)$

Dimostrazione Sia $M = (\Sigma, Q, \delta, q_0, F)$ il DFA. La grammatica $G_M = (Q, \Sigma, R, q_0)$ ha:

- per NT gli stati di M
- per T, l'alfabeto di M
- per simbolo iniziale è lo stesso iniziale di M
- per produzione R:
  - $\forall\, \delta(q_i, a) = q_j$ la produzione $q_i \to a\, q_j \in R$, inoltre se $q_j \in F$, anche $q_i \to a \in R$
  - se $q_0 \in F$, allora $q_0 \to \varepsilon \in R$

M



aa^* è l'espressione regolare che descrive L[M]

G_M [$A \to aB \mid a$ perché B è stato finale

$B \to aB \mid a$

Secondo la variante q

G_M [$A \to aB$ non è regolare, perché

$B \to aB \mid \varepsilon$ ammette $B \to \varepsilon$

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

Il linguaggio definito da una grammatica regolare G è un linguaggio, cioè è possibile costruire una espressione regolare S_G tale che $L(G) = \mathcal{L}[S_G]$

Idea della costruzione:

Caso Semplice: un solo NT

$A \to aA \mid bB \mid \varepsilon$ è intuitivo vedere che $a^*(b|c)$ è la espres. regolare associata

Caso generale:

$A \to aA \mid bB \mid c$ } lo si può vedere come un sistema
$B \to cA \mid aB \mid d$ } da risolvere

Ricaviamo B dalla seconda "equazione" $B \approx a^*(cA|d)$ dove A compare nella espr. regolare

Ora sostituiamo B nella prima "equazione" $A \approx aA|ba^*(cA|d)|c$

Con opportune manipolazioni su espr. regolari, usando leggi che abbiamo visto, possiamo scrivere $A \approx aA|ba^*cA|ba^*d|c$ e quindi $A \approx (a|ba^*c)A|ba^*d|c$

Ora siamo nella forma "semplice" e sappiamo come fare: A ha associata la esp. regolare $(a|ba^*c)^*(ba^*d|c)$

In generale

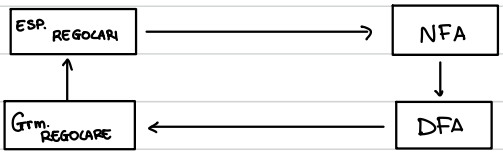
$A_1 \approx a_{11}A_1 | \dots | a_{1m}A_m | b_{11} | \dots | b_{1p_1}$
$\vdots$
$A_m \approx a_{m1}A_1 | \dots | a_{mm}A_m | b_{m1} | \dots | b_{mp_m}$

Si parte con $A_m = S_m[A_1, \dots, A_{m-1}]$ cioè si costruisce una esp. regolare per A_m che usa $A_1, \dots, A_{m-1}$

Poi si procede sostituendo A_m (o meglio $S_m[A_1, \dots, A_{m-1}]$ al posto di A_m) nell'equazione per A_{m-1}, cioè

$A_{m-1} \simeq S_{m-1}[A_1, \dots, A_{m-2}]$ e così via fino ad arrivare ad A_1 (che è il simbolo iniziale)

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.



Tutti questi formalismi sono equivalenti

Tutti generano / riconoscono / descrivono la stessa classe di linguaggi, ovvero i linguaggi REGOLARI.
gr. reg. — NFA/DFA — ESP. REG

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

Due stati q_1 e q_2 di un DFA N sono equivalenti (o indistinguibili) se $\forall x \in \Sigma^*$ $\hat{\delta}(q_1,x) \in F$ sse $\hat{\delta}(q_2,x) \in F$, cioè se $L[N,q_1] = L[N,q_2]$

simmetricamente, due stati q_1 e q_2 non sono equivalenti se $\exists x \in \Sigma^*$ tale che $\hat{\delta}(q_1,x) \in F$ ma $\hat{\delta}(q_2,x) \notin F$
$\hat{\delta}(q_1,x) \notin F$ ma $\hat{\delta}(q_2,x) \in F$ q_1 e q_2 sono "distinguibili".

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

ALGORITMO ITERATIVO

0) Costruire la tabella a scala

1) Marcare X_0 ogni coppia (q_1,q_2) tale che $q_1 \in F$ e $q_2 \in Q/F$ (o viceversa)

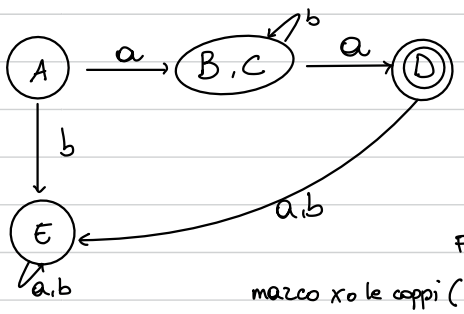
2) b := True, i := 1;

3) while b do {

3.1) b := false

3.2) $\forall$ coppia (q_1,q_2) non marcata
do { if $\exists a \in \Sigma$ con $(\delta(q_1,a), \delta(q_2,a))$ già marcata
then { marca (q_1,q_2) con X_i;
b := True;
}
}

3.3) i := i+1
}



Finali non finali
marco X_0 le coppie (F, NF)

marco con X_1 le coppie che seguono il cammino mi portano ad altre coppie che sono già state marcate al round precedente.

B	X_1			
C	X_1			
D	X_0	X_0	X_0	
E	X_2	X_1	X_1	X_0
	A	B	C	D

4) Al termine sia J l'insieme delle coppie non marcate

5) La relazione di equivalenza $\sim$ è la chiusura riflessiva e simmetrica J, cioè

$$\sim = J \cup \{(q_1,q_2) \mid (q_1,q_2) \in J\} \cup \{(q,q) \mid q \in Q\}$$

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

Dato un DFA $M=(\Sigma, Q, \delta, q_0, F)$, l'automa $M_{min}=(\Sigma, Q_{min}, \delta_{min}, [q_0], F_{min})$ riconosce lo stesso linguaggio di M, ed ha il minimo numero di stati tra tutti gli automi deterministici per questo linguaggio.

52. Cos'è Lex? Qual è il suo input e il suo output?

Lex è un generatore di analizzatori lessicali.

Input: un file di tipo.l che contiene un insieme di definizioni (pattern) regolari ed una serie di assiomi corrispondenti.

Output: un programma C che realizza l'automa riconoscitore e che associa ad ogni istanza di una definizione (lessema) la relativa asioma.

53. Qual è la struttura di un file .l? Come funziona l'analizzatore lessicale prodotto da Lex?

File.l è diviso in 3 parti: 1. DICHIARAZIONI che sono delle definizioni regolari

2. REGOLE : {esp. regolare (PATTERN)} c {azione} → frammento di codice C

3. FUNZIONI AUSILIARI : Se si usano funzioni complesse nella parte "azione", queste potrebbero essere definite qui.

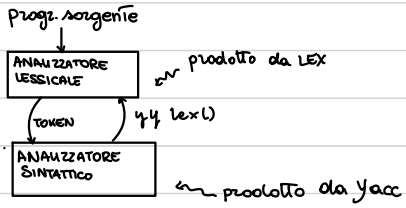
L'analizzatore lessicale prodotto da LEX è in realtà un programma in C, che deve essere compilato per renderlo eseguibile in a.out, implementa essenzialmente il DFA riconoscitore dell'insieme delle espressioni regolari contenute nelle "Regole".

- Scandisce il testo sorgente alla ricerca di una stringa che corrisponde a (cioè sia un lessema per) una delle espr. regolari (pattern di categoria sintattica).
- quando riconosce un lessema, esegue l'azione specificata, e passa in output il risultato dell'azione al posto del lessema.
- quando l'input non corrisponde a nessun pattern, lo lascia inalterato e "segnala" la cosa al "gestore degli errori".

54. Come si interfaccia Lex con Yacc?

Il programma generato da LEX non è usato da solo, ma insieme a YACC è un generatore di analizzatori sintattici.

Alcune variabili comuni ai due programmi (yy.Lval) permettono di scambiare informazioni. Lex.yy.c è usato da YACC "on demond", richiedendo il token successivo yy.lex() restituisce il nome del Token, mentre il valore del token è condiviso nella variabile yy.lval.



55. **Intestazione e dimostrazione del pumping lemma.**

Se L è un ling. regolare, allora $\exists N>0$ t.c. $\forall z \in L$ con $|z| \geq N$, $\exists u,v,w$ t.c.:
- $z = uvw$
- $|uv| \leq N$
- $|v| \geq 1$
- $\forall k \geq 0 \quad uv^k w \in L$

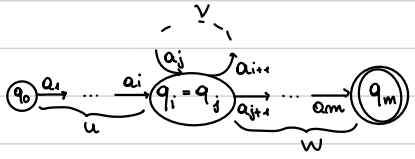
Inoltre N è minore o uguale del numero di stati del DFA minimo che accetta L.

DIM: Sia $N=|Q_M|$, dove M è il DFA minimo che accetta L.

Sia $z = a_1 a_2 \dots a_m \in L$ con $m \geq N$

Quindi $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_m} q_m \in F$

Ora a_m è dato da $m+1$ stati con $m+1 > N \Rightarrow \exists i,j \ (i \neq j)$ tale che $q_i = q_j$



- Poichè $i \neq j \Rightarrow v = a_{i+1} \dots a_j$ e t.c. $|v| \geq 1$
- La condizione $|uv| \leq N$ mi dice che (se m è molto grande potrebbe esserci più cicli) prende il primo ciclo!

$uv^0w = uw \in L$
$uv^1w = uvw = z \in L$
$\vdots$
$uv^kw \in L$

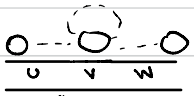
$\forall k \geq 0 \quad uv^kw \in L$ perché il ciclo v può essere percorso un numero arbitrario di volte

CI SONO DUE OSS DA FARE

56. Come si può utilizzare il pumping lemma (a rovescio) per dimostrare che un linguaggio non è regolare?

$L = \{ a^m b^m \mid m \geq 1 \}$

- Fissiamo un N ($\forall N \geq 0$)
- Scegliamo $z = a^N b^N$ ($\exists z \in L$ con $|z| \geq N$)
- Per ogni possibile scomposizione di z in sottostringhe t.c. $\forall uvw$
  - (i) $z = uvw$
  - (ii) $|uv| \leq N$
  - (iii) $|v| \geq 1$



Se consideriamo $|uv| \leq N$ impone che $u \in v$ siano fatte solo di "a" z quindi $v = a^j$ con $j \geq 1$.

- $\exists k = 2$ t.c $uv^2w = uvvw = uvvw$
  $a^{N+j} b^N \notin L$

$\Rightarrow$ L non è regolare

57. Quali sono le proprietà di chiusura dei linguaggi regolari? Quali proprietà si possono decidere (ovvero verificare algoritmicamente)?

La classe dei ling. regolari è chiusa per

1. UNIONE
2. CONCATENAZIONE
3. STELLA DI KLEENE
4. COMPLEMENTAZIONE
5. INTERSEZIONE da De morgan

Si possono decidere ?

CAPITOLO 4

58. Cosa si intende per analisi sintattica? **Cos'è un parser?**

Analisi Sintattica (Parser) a partire dalla lista di token, generata dallo Scanner, il parser produce l'albero di derivazione del programma, riconoscendo se le fasi sono sintatticamente corrette. Consulta la tabella dei simboli e aggiunge dei attributi. Inoltre interagisce insieme all'analisi sintattica, con gestore degli errori.

59. **Definizione di automa a pila nondeterministico (PDA).** Definizione di configurazione (o descrizione istantanea), mossa in un passo e mossa in più passi.

Un automa a pila nondeterministico (PDA) è una 7-pla $(\Sigma, Q, \Gamma, \delta, q_0, \bot, F)$ dove:

- Σ è un alfabeto finito (simboli in input)
- Q è un insieme finito di stati
- Γ è un insieme finito di simboli della pila
- δ è la funzione di transizione con tipo $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow \mathcal{P}_{fin}(Q \times \Gamma^*)$
- q_0 è lo stato iniziale
- $\bot \in \Gamma$ è il simbolo iniziale sulla pila
- $F \subseteq Q$ è l'insieme degli stati finali

TRANSIZIONI DI UN PDA

- DESCRIZIONE ISTANTANEA (O CONFIGURAZIONE) (q, w, β)
  - $q \in Q$ (stato corrente)
  - $w \in \Sigma^*$ (input non ancora letto)
  - $\beta \in \Gamma^*$ (stringa sulla pila)
- MOSSA
  - (1) quella in cui consumo un pezzo dell'input $\dfrac{(q', \alpha) \in \delta(q, a, X) \quad a \in \Sigma}{(q, aw, X\beta) \vdash_N (q', w, \alpha\beta)}$
  - (2) quella in cui faccio una mossa ε $\dfrac{(q', \alpha) \in \delta(q, \varepsilon, X)}{(q, w, X\beta) \vdash_N (q', w, \alpha\beta)}$
- computazione / cammino

$$\frac{}{(q,w,\beta) \vdash_N^* (q,w,\beta)} \quad \text{chiusura riflessiva } \vdash_N$$

$$\frac{(q,w,\beta) \vdash_N^* (q',w',\beta') \vdash_N (q'',w'',\beta'')}{(q,w,\beta) \vdash_N^* (q'',w'',\beta'')} \quad \text{chiusura Transitiva } \vdash_N$$

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

Esistono due modalità di riconoscimento:

① per stato finale $L[N] = \{ w \in \Sigma^* \mid (q_0, w, \bot) \vdash_N^* (q, \varepsilon, \alpha) \text{ con } q \in F \}$

② per pila vuota $P[N] = \{ w \in \Sigma^* \mid (q_0, w, \bot) \vdash_N^* (q, \varepsilon, \varepsilon) \}$

con $N = (\Sigma, Q, \Gamma, \delta, q_0, \bot, F)$

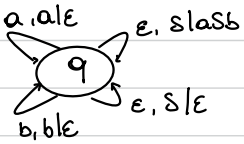
Per un certo PDA N, spesso $L[N] \neq P[N]$

Dimostriamo però che se $L = L[N]$, allora esiste N' tale che $L = P[N']$, e viceversa, se $L = P[N]$, allora $\exists N''$ t.c $L = L[N'']$ cioè non cambia la classe dei linguaggi riconosciuti da PDA per stato finale o per pila vuota.

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.

Con il metodo Top-Down

$S \to aSb \mid \varepsilon$



costruiamo un automa a un solo stato con due transizioni

- MATCH → quelle a Sx
- corrisponde alla produzione della grammatica → se c'è S in cima alla pila → o si svuota → si cancella S; o viene espansa

Teorema: Un linguaggio L è libero da contesto ⟺ è accettato da un PDA. (se è generato da una gram. libera)

Lemma 1: Ogni PDA N può essere simulato da un PDA N' con un solo stato

Lemma 2: Ogni PDA con un solo stato ha una equivalente gram. libera.

Concludo dicendo che un linguaggio L è libero sse è accettato da un PDA.

62. Quali sono le proprietà di chiusura dei linguaggi liberi? L'intersezione di un linguaggio libero con un regolare è un linguaggio libero?

Th: I linguaggi liberi sono chiusi per : 1) Unione
2) Concatenazione
3) Ripetizione (Stella di Kleene)

Th: L'intersezione $L_1 \cap L_2$ di un ling. libero L_1 con un ling. regolare L_2 è un ling. libero

63. Intestazione e dimostrazione del "Pumping Theorem".

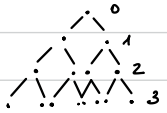
Se L è libero, allora $\exists N>0$ t.c. $\forall z \in L$ con $|z| \geq N$, $\exists u,v,w,x,y$ tale che: ① $z = uvwxy$
② $|vwx| \leq N$
③ $|vx| \geq 1$
④ $\forall k \geq 0 \quad uv^k wx^k y \in L$

Dimostrazione: Dato che il linguaggio è libero, Sia $G=(NT,T,R,S)$ una grammatica libera tale che $L=L(G)$

- Sia b il massimo fattore di ramificazione in un albero di derivazione (ovvero il massimo numero di simboli che compaiono nella parte dx di una produzione in R) $b = \max\{|\alpha| \mid A \to \alpha \in R\}$

(oss: $b \geq 2$, altrimenti la grammatica sarebbe banale)

- Un albero di altezza h (conto la radice) e fattore di ramificazione b, ha al più b^h foglie



$h = 3$
$b = 2$
2^3 foglie

Fissiamo $N = b^{|NT|+1}$ (quindi $N > b^{|NT|}$ dato che $b \geq 2$)

Allora ogni albero di derivazione per z, con $|z| \geq N$, deve avere altezza almeno $|NT|+1$.

Prendiamo una qualunque $z \in L$, con $|z| \geq N$

Consideriamo il suo albero di derivazione (se ne possiede più di uno, perché G è ambigua, prendiamo quello con il minor numero di nodi).

Dunque $|z| \geq N \Rightarrow$ albero con altezza $\geq |NT|+1$

$\Rightarrow \exists$ un cammino da radice S ad una foglia con almeno $|NT|+2$ nodi

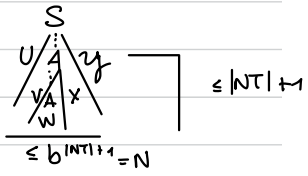
$\Rightarrow$ Quel cammino attraversa $|NT|+1$ nodi intermedi etichettati con un NONTERMINALE (la foglia è etichettata da un Terminale)

$\Rightarrow$ almeno un nonterminale si ripete in quel cammino.

Allora $S \Rightarrow^* z$ può essere diviso come $S \Rightarrow^* uAy \Rightarrow uvAxy \Rightarrow uvwxy$

$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uv^2Axy \Rightarrow uv^2wx^2y$

come albero



Allora

$S \Rightarrow^* uAy \Rightarrow^* uwy \quad k=0$

$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvwxy \quad k=1$

$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uv^2Ax^2y \quad k=2$

$\vdots$

Bisogna solo verificare i vincoli:

- $|vx| \geq 1$ ovvio: se entrambe ϵ, allora l'albero per $k=0$ genererebbe ancora z ed avrebbe meno nodi contraddicendo l'ipotesi di aver scelto il più piccolo albero
- $|vwx| \leq N$ ovvio: il cammino da A alla foglia è di lunghezza $\leq |NT|+1$ (cioè usa il Terminale foglia) $\Rightarrow$ da A in alto non può generare parole più lunghe di $b^{|NT|+1} = N$

partendo dal basso, prendo il primo "ciclo" che si ferma!

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

Pumping Theorem

Se L è libero $\Rightarrow P$

Pumping Theorem al rovescio

Se $\forall N>0\ \exists z\in L$ con $|z|\geq N$ t.c $\forall uvwxy$ ($z=uvwxy$ (1) $z=uvwxy$

(2) $|vwx|\leq N$

(3) $|vx|\geq 1$

allora $\exists k\geq 0.\ uv^k wx^k y\in L$)

allora L non è libero.

65. **Classificazione di Chomsky delle grammatiche e dei linguaggi.** Definizione di grammatica dipendente dal contesto e di grammatica monotona. Quale tipo di automi corrisponde ad ogni classe?

Grammatiche regolari $A\to aB$ $A\to a$ $S\to\varepsilon$

Grammatiche libere da contesto $A\to\gamma$ con $\gamma\in(N\cup T)^+$ $S\to\varepsilon$

Grammatiche dipendenti da contesto $\gamma A\delta\longrightarrow\gamma w\delta$ $\gamma,\delta\in(N\cup T)^*$ $w\in(N\cup T)^+$

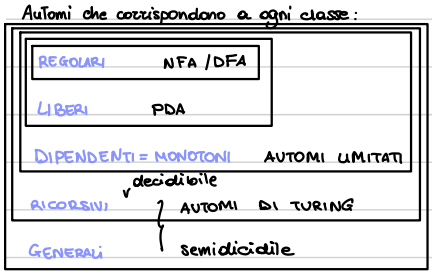
$S\to\varepsilon$

Grammatiche monotone $\gamma\to\delta$ con $|\gamma|\leq|\delta|$

Grammatiche generali (a struttura di frase) $\gamma\to\delta$ (senza alcun vincolo)

Teorema: $\forall G_1$ monotona, $\exists G_2$ dipendente da contesto t.c. $L(G_1)=L(G_2)$

Automi che corrispondono a ogni classe:



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

Un PDA $N=(\Sigma,Q,\Gamma,\delta,q_0,\perp,F)$ è deterministico (DPDA) sse

(1) $\forall q\in Q\ \forall z\in\Gamma$, se $\delta(q,\varepsilon,z)\neq\emptyset$ allora $\delta(q,a,z)=\emptyset\ \forall a\in\Sigma$

(2) $\forall q\in Q\quad\forall z\in\Gamma\quad\forall a\in\Sigma\cup\{\varepsilon\}\ |\ \delta(q,a,z)\leq 1.$

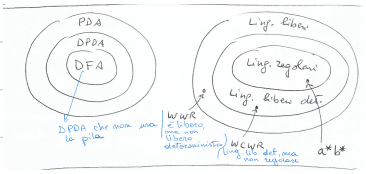
Def. Un linguaggio è libero deterministico se è accettato per stato finale da un DPDA

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

La classe dei ling. liberi deterministici è inclusa propriamente nella classe dei ling. liberi

Se L è regolare, allora $\exists$ DFA M tale che $L=L[M]$

Ogni ling. regolare è anche libero deterministico



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

Un linguaggio libero deterministico L è riconosciuto da un DPDA per pila vuota sse L gode della "prefix property":

$\nexists x,y\in L$ tali che x è prefisso di y

69. Usando un endmarker $, si può riconoscere un linguaggio libero deterministico che non gode della prefix property anche per pila vuota? Come?

Se L libero deterministico non gode della PREFIX PROPERTY, non può essere riconosciuto da un DPDA per pila vuota.
↳ allora $L\$ = \{w\$ \mid w \in L\}$ gode della prefix property
Quindi $L\$$ può essere riconosciuto da un DPDA per pila vuota.

70. Un linguaggio libero deterministico è ambiguo?

Se L è libero deterministico (cioè riconosciuto da un DPDA per stato finale), allora L è generabile da una grammatica libera (non sono ambigue) NON AMBIGUA.

71. Proprietà di chiusura dei linguaggi liberi deterministici: chiusi per complementazione, ma non per intersezione né per unione.

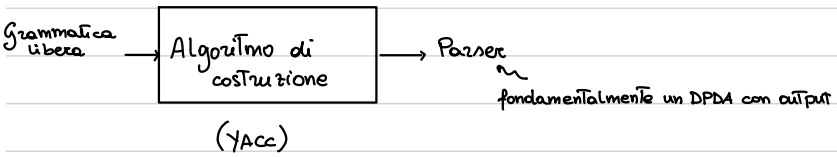
Proprietà:

- Ling. liberi deterministici sono chiusi per complementazione, cioè se $\exists$ DPDA N t.c. $L = L[N]$ allora esiste un DPDA N' t.c. $\overline{L} = L[N']$, dove $\overline{L} = \Sigma^* \setminus L$
- Ling. liberi deterministici non sono chiusi per intersezione

  $L_1 = \{a^n b^n c^m \mid n,m \geq 0\}$ è lib. det.

  $L_2 = \{a^m b^n c^n \mid n,m \geq 0\}$ è lib det. ma $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ NON È LIB.
- Ling liberi deterministici non sono chiusi per unione, se, per assurdo, lo fossero, allora $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ e quindi risulterebbe chiusi per intersezione, il che è impossibile!

72. Da cosa si parte per costruire un analizzatore sintattico (ovvero parser)? Da una espressione regolare? Da una grammatica libera? Da un PDA?



73. Cosa prende in input e cosa produce in output un parser?

74. Che differenza c'è tra un parser nondeterministico ed uno deterministico?

I parsa possono essere di due tipi diversi:

- NONDETERMINISTICI: se, durante la ricerca di una derivazione, si scopre che una scelta è improduttiva e non porta a riconoscere l'input, il parser torna indietro (BACKTRAKING), disfa parte della derivazione appena costruita, e sceglie un'altra produzione, tornando a leggere (parte de)l'input
- DETERMINISTICI: leggono l'input una sola volta; ogni loro decisione è definitiva.

Entrambi cercano di sfruttare informazioni dall'input per guidare la ricerca della derivazione.

76. Le tecniche top-down e bottom-up in che cosa differiscono?

Parser Top-down: ricostruiscono una derivazione LEFTMOST, per una stringa a partire dal simbolo iniziale S (all'inizio sulla pila)

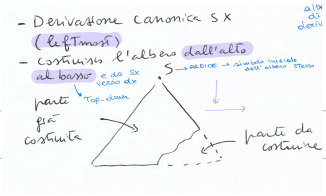
Parser BOTTOM-UP: ricostruiamo una derivazione RIGHTMOST (a rovescio) a partire dalla stringa w, cercando di ridurla al simbolo iniziale S (alla fine sulla pila)

Entrambi cercano di sfruttare quello che vedono dell'input per guidare la ricerca della derivazione.

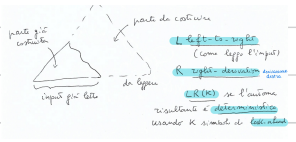
77. Quali tipi di grammatiche non sono adatte al top-down parsing? Quali tipi di produzioni sono poco adatte al bottom-up parsing?

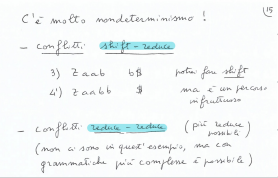
Abbiamo visto i Parser top-down deterministici (ottenuti a partire da grammatiche LL(k), in particolare LL(1)).
Non tutte le grammatiche sono adatte per TOP-DOWN parsing, ad esempio quando contengono la RICORSIONE, per questo motivo dobbiamo manipolarla in modo da ottenerne una equivalente, ma senza ricorsione. Inoltre non è affatto anche quando è presente il NONDETERMINISMO.

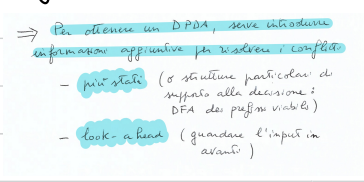
Può essere risolto scegliendo la produzione in base al simbolo in lettura (look-ahead!)



Parser Bottom-up deterministici (ottenuti a partire da grammatiche LR(k), in particolare LR(0), SLR(1), LR(1) e LALR(1)).







non è non è adatta la produzione ε

78. Cosa sono le produzioni epsilon e cosa sono i simboli non-terminali annullabili?

$S \to \varepsilon$ è una produzione epsilon solitamente si cerca di togliere dalla grammatica libera, in quanto causa problemi, ad esempio la Togliamo quando vogliamo utilizzare un bottom-up parsing. → $L(G') = L(G) \setminus \{\varepsilon\}$

I simboli annullabili: $A \in NT$ t.c. $A \Rightarrow^* \varepsilon$ $N(G) = \{A \in NT \mid A \Rightarrow^* \varepsilon\}$ viene calcolato induttivamente come segue

$$N_0(G) = \{A \in NT \mid A \to \varepsilon \in R\}$$

$$N_{i+1}(G) = N_i(G) \cup \{B \in NT \mid B \to C_1 \dots C_k \in R \text{ e } C_1, \dots, C_k \in N_i(G)\}$$

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?

ALGORITMO PER ELIMINARE LA PRODUZIONE ε (22

Una volta calcolato $N(G)$ per $G = (NT, T, S, R)$, costruiamo la grammatica $G' = (NT, T, S, R')$ dove (sono diversi)

per ogni produzione $A \to \alpha \in R$, con $\alpha \neq \varepsilon$, in cui occorrono (prendiamo tutte le produzioni di R una non quella!)

i simboli annullabili $z_1, \dots, z_k$, mettiamo in R' tutte le produzioni del tipo $A \to \alpha'$ dove α' si ottiene da α cancellando tutti i possibili sottoinsiemi di $z_1, \dots, z_k$ (incluso $\emptyset$), ad eccezione del caso in cui α' risulta ε.

- in G' non mettiamo produzioni $A \to \varepsilon \in R$
- in G' non introduciamo mai produzioni del tipo $A \to \varepsilon$

80. Cosa sono le produzioni unitarie e cosa sono le coppie unitarie?

- produzioni unitarie: $A \to B$ con $A, B \in NT$
- coppie unitarie: (A, B) tale che $A \Rightarrow^* B$ (A si può riscrivere in B in 0 o più passi nel non terminale B) (dobbiamo calcolarle)

usando solo produzioni unitarie

81. Come si può trasformare una grammatica G che contiene produzioni unitarie in una equivalente G' che non ne contiene?

11

Algoritmo: Data $G = (NT, T, R, S)$ libera, si definisce $G' = (NT, T, R', S)$ dove, per ogni $(A, B) \in U(G)$ (appartenente all'insieme delle coppie unitarie), R' contiene tutte le produzioni $A \to \alpha$, dove $B \to \alpha \in R$ e non è unitaria.

Oss: Poiché, per ogni $A \in NT$, la coppia $(A, A) \in U(G)$, R' contiene tutte le produzioni non-unitarie di R e in aggiunta un po' di altre.

82. Data una grammatica G, quali sono i suoi simboli utili? Quali sono i suoi simboli generatori? Quali i suoi simboli raggiungibili?

Def: Un simbolo $X \in T \cup NT$ è detto

- un **generatore** sse $\exists w \in T^*$ con $X \Rightarrow^* w$ (una stringa possibile del linguaggio; è un generatore se lui stesso è un terminale o un non terminale che può generare qualcosa; t.c. X si può ridurre in w)
- **raggiungibile** sse $S \Rightarrow^* \alpha X \beta$ per qualche $\alpha, \beta \in (T \cup NT)^*$ (a partire dal simbolo iniziale della gram. (S) posso raggiungere una forma sequenziale data che ha X; sono sequenze arbitrarie di T e NT)
- **utile** sse è sia generatore, sia raggiungibile

ovvero se $S \Rightarrow^* \alpha X \beta \Rightarrow^* z \in L(G)$, cioè X compare in almeno una derivazione di una stringa $z \in L(G)$.

83. Come si calcolano i generatori? Come si calcolano i raggiungibili?

Ricorda: X è un generatore se $\exists w \in T^*$ t.c. $X \Rightarrow^* w$

1) CASO BASE $G_0(G) = T$ (insieme dei generatori) — a si può derivare in se stesso in zero passi: se $a \in T$, $a \Rightarrow^* a$! allora tutti i terminali
2) caso successivo $G_{i+1}(G) = G_i(G) \cup \{ B \in NT \mid B \to C_1 \dots C_k \in R \text{ e } C_1, \dots, C_k \in G_i(G) \}$ (elementi presi nei passi precedenti sono generatori)

Ricordo: si calcolano per induzione sulla lunghezza della derivazione

X è raggiungibile sse $S \Rightarrow^* \alpha X \beta$ per qualche $\alpha, \beta \in (NT \cup T)^*$ forma sentenziale in cui compare il simbolo X

1) caso base $R_0(G) = \{S\}$ (simbolo iniziale → S → S, raggiungibile in zero passi)
2) caso succ. $R_{i+1}(G) = R_i(G) \cup \{X_1, \dots, X_k\}$ (aggiungo tutti i simboli che compaiono nella parte dx delle derivazioni di $R_i(G)$) $\forall B \in R_i(G)$, $B \to X_1 \dots X_k \in R$ (simboli che ho al passo i)

84. In che modo si eliminano i simboli inutili di una grammatica? E importante l'ordine delle operazioni da svolgere?

Come rimuovere i simboli inutili?

Algoritmo:

1) Prima elimino tutti i non-generatori (e tutte le produzioni che usano almeno uno di questi)
2) Poi, dalla nuova grammatica, elimino tutti i non-raggiungibili (e tutte le prod. che li usano)

⇒ La grammatica risultante è equivalente all'originale e non contiene simboli inutili

} se invertiamo l'ordine, allora può capitare che non elimino tutti i simboli inutili

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?**

Eliminare la ricorsione sinistra (35)

(La ricorsione sinistra può presentarsi nelle produzioni se abbiamo nella nostra grammatica delle funzioni ricorsive sinistre)

(problema per parsa top-down)

- produzione ricorsiva sx: $A \to A\alpha \in R$ (la parte destra della produzione inizia con lo stesso NT della parte sx)
- G è ricorsiva sx: $A \Rightarrow^+ A\alpha$ per qualche $A \in NT$ e $\alpha \in (T \cup NT)^*$ (la gram. per qualche NT A è possibile avere una derivazione in uno o più passi che parte da A ed arriva ad A stesso)

Come rimuove la ricorsione sx immediata?

$A \to A\alpha_1 | \dots | A\alpha_n | \beta_1 | \dots | \beta_m$ (stringhe che non cominciano per A)

Queste produzioni possono essere rimpiazzate da

$A \to \beta_1 A' | \dots | \beta_m A'$ dove β_1 e β_m non sono mai ricorsive

$A' \to \alpha_1 A' | \dots | \alpha_n A' | \varepsilon$ (posso aggiungerla)

Se nella gr. originale abbiamo la derivazione

$A \Rightarrow A\alpha_{i_1} \Rightarrow A\alpha_{i_2}\alpha_{i_1} \Rightarrow \dots \Rightarrow A\alpha_{i_k} \dots \alpha_{i_2}\alpha_{i_1} \Rightarrow \beta_i \alpha_{i_k} \dots \alpha_{i_2}\alpha_{i_1}$

allora nella nuova grammatica avremo

$A \Rightarrow \beta_i A' \Rightarrow \beta_i \alpha_{i_\mu} A' \Rightarrow \dots \Rightarrow \beta_i \alpha_{i_k} \dots \alpha_{i_2} A' \Rightarrow \beta_i \alpha_{i_k} \dots \alpha_{i_2} \alpha_{i_1} A' \Rightarrow \beta_i \alpha_{i_k} \dots \alpha_{i_2} \alpha_{i_1}$

e viceversa

Ricorsione SX non-immediata

(37) SOLITAMENTE NON DA ESERCIZI DI QUESTO TIPO

la ricorsione si ottiene in più passi ($S \Rightarrow Ba \Rightarrow Sca$)

Consideriamo

$$\left.\begin{array}{l} S \to Ba \mid b \\ B \to Bc \mid Sc \mid d \end{array}\right] G$$

In G c'è ricorsione SX immediata ($B \to Bc$), la ricorsione SX si vede in un passo

ma anche non-immediata ($S \Rightarrow Ba \Rightarrow Sca$). S: simbolo iniziale SX

Algoritmo: **Input:** G libera senza ε-produzioni, senza prod. unitarie, ma con ricorsione SX non-immediata

Output: G' libera, senza ricorsione SX, ma può avere ε-produzioni

Sia $NT = \{A_1, A_2, .., A_n\}$ un ordine fissato

For i from 1 to n { (per i che va da 1 a n)

1) For j from 1 to i-1 { (per j che va da 1 a i-1)

sostituisco ogni produzione della forma $A_i \to A_j \alpha$ con le produzioni

$A_i \to \beta_1 \alpha \mid .. \mid \beta_k \alpha$ dove $A_j \to \beta_1 \mid .. \mid \beta_k$ sono le produzioni correnti per A_j

A_i è un NT che viene dopo A_j in $j < i$

} operazione di sostituzione, perché tutte via delle produzioni e al loro posto ne metto altre

Può capitare che da β_i a β_k uno di loro sia uguale a A_i ↳ Si crea un'altra prod. ricorsiva a SX → risolvo il problema al secondo passo

2) Elimina la ricorsione immediata su A_i

}

86. Cosa vuol dire fattorizzare a sinistra una grammatica? Perché serve fattorizzare?

Fattorizzazione a Sinistra

(importante per top-down parsing)

Se nella nostra gram. ci fosse qualcosa di questo tipo:

$$A \to aBbC \mid aBd$$

Se, in top-down parsing, sulla pila ho A e leggo in input a, non sono in grado di determinare quale produzione scegliere (cioè del nondeterminismo)

⇒ raccolgo la parte comune ("aB") alle 2 produzioni e introduco un nuovo nonterminale per rappresentare il "resto"

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

87. Cos'è un parser a discesa ricorsiva?

Parser a discesa ricorsiva è una collezione di funzioni, una per ogni NT della grammatica

Data una grammatica libera $G = (NT, T, S, R)$, per ogni nonTerminale A con produzioni

$$A \to X_1^1 \ldots X_{m_1}^1 \mid \ldots \mid X_1^k \ldots X_{m_k}^k$$

definisco la funzione

function A() {

- scelgo nondeterministicamente k tra 1 e k_e, ovvero una produzione $A \to X_1^k \ldots X_{m_k}^k$;
- for i = 1 to m_k {
  if $X_i^k \in NT$ then $X_i^k()$;
  else if X_i^k = simbolo corrente dell'input then avanza di un simbolo sull'input
  else Fail();

} backtracking! si torna al punto e si sceglie un'altra produzione

} altrimenti se il simbolo che sta esaminando la produzione non corrisponde al simbolo corrente dell'input => fallisci

fai backtracking: Torni al passo iniziale e scegli nondeterministicamente un'altra produzione fra quelli possibili

return;

}

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

FIRST

Data una grammatica libera G e $\alpha \in (T \cup NT)^*$ (qualunque sequenza di T e NT), diciamo che $First(\alpha)$ è l'insieme dei terminali che possono stare in prima posizione in una stringa che si deriva da α.

- per $a \in T$, $a \in First(\alpha)$ se $\alpha \Rightarrow^* a\beta$ per $\beta \in (T \cup NT)^*$ (α può anche ricevere in regalo o più passi in $a\beta$ dove β è una qualunque sequenza di T e NT)
- inoltre se $\alpha \Rightarrow^* \varepsilon$ (α si annulla), allora $\varepsilon \in First(\alpha)$ (First di α sono o dei T o la stringa vuota (in questo caso))

Follow

(si applica solo ai NT)

Data una grammatica libera G e $A \in NT$, diciamo che $Follow(A)$ è l'insieme dei terminali che possono comparire immediatamente a destra di A in una forma sentenziale.

- per $a \in T$, $a \in Follow(A)$ se $S \Rightarrow^* \alpha A a \beta$ per qualche $\alpha, \beta \in (T \cup NT)^*$ ("a" compare immediatamente a dx del simbolo A)
- $\$ \in Follow(A)$ se $S \Rightarrow^* \alpha A$ dove A non ha nulla alla sua dx (vale sempre)

(Poiché $S \Rightarrow^* S$, allora $\$ \in Follow(S)$!)

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

Come calcolare i First?

Sia $N(G) \subseteq NT$ l'insieme dei simboli annullabili (quei simboli NT che si possono definire in ε)

($A \in N(G)$ se $A \Rightarrow^* \varepsilon$)

- Per ogni $x \in T$, $First(x) = \{x\}$
- Per ogni $X \in NT$, inizialmente $First(X) = \emptyset$;
- Ripeti il seguente ciclo finché nessun $First(X)$ viene più modificato in una iterazione:
  - Per ogni produzione $X \to Y_1 \dots Y_k$ (dove la parte dx è una stringa $Y_1 \dots Y_k$)
    - per ogni i da 1 a k
      se $(Y_1, \dots, Y_{i-1} \in N(G))$ (insieme dei annullabili) / (vera se $i = 1$)
      allora $First(X) := First(X) \cup (First(Y_i) \setminus \{\varepsilon\})$
      es: prendo y_1 se questo è annullabile prendo quelli che seguono
- Per ogni $X \in N(G)$, $First(X) := First(X) \cup \{\varepsilon\}$

First può essere esteso ad $\alpha \in (T \cup NT)^*$ come segue:

- $First(\varepsilon) = \{\varepsilon\}$
- $First(X\beta) = First(X)$ se $X \notin N(G)$ (X non è un simbolo annullabile)
- $First(X\beta) = (First(X) \setminus \{\varepsilon\}) \cup First(\beta)$ se $X \in N(G)$

Procedura per calcolare Follow(X) (con $X \in NT$)

- Per ogni $X \in NT$, inizialmente $Follow(X) := \emptyset$ (il follow di X con l'insieme vuoto)
- $Follow(S) := \{\$\}$ (prendo inizialmente almeno il simbolo di $)
- Ripeti il seguente ciclo finché nessun $Follow(X)$ viene più modificato in una iterazione:

1) Per ogni produzione $X \to \alpha Y \beta$
$Follow(Y) := Follow(Y) \cup (First(\beta) \setminus \{\varepsilon\})$

2) Per ogni produzione $X \to \alpha Y$ e per ogni produzione $X \to \alpha Y \beta$ con $\varepsilon \in First(\beta)$
$Follow(Y) := Follow(Y) \cup Follow(X)$ (Follow della Testa della mia produzione)

In pratica, bisogna cercare tutte le produzioni in cui $Y \in NT$ appare (nella parte dx), e per ognuna di esse, applicare la 1 o la 2 sopra.

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

Tabella di Parsing LL(1)

(i passeggeri chiusono questa Tabella. Leggono l'input da sx verso dx)

- strumento per risolvere il nondeterminismo!
- input left-to-right
- un simbolo di look-ahead (per prendere decisioni)
- sono portati top-down quindi generano una derivazione leftmost

è una **Matrice bidimensionale M**

- righe: nonterminali
- colonne: Terminali (più $)
- casella (A, a) (coppia): $M[A, a]$ contiene le produzioni che possono essere scelte dal parser mentre tenta di espandere A e l'input corrente è a.

(abbiamo tante righe quanti sono i NT della grammatica e tante colonne quanti sono i terminali della medesima gram + 1 → $)

Se ogni casella contiene **al più una produzione**, allora il parser è **deterministico**!

Come si riempie la tabella?

Algoritmo

Per ogni produzione $A \to \alpha$

1) per ogni $a \in T$ e $a \in First(\alpha)$, inserisco $A \to \alpha$ (la produzione) nella casella $M[A, a]$
2) se $\varepsilon \in First(\alpha)$, inserisci $A \to \alpha$ in tutte le caselle $M[A, x]$ per $x \in Follow(A)$ (x può essere $)

Ogni casella vuota, dopo aver elaborato tutte le produzioni, è un errore (cioè la funzione ricorsiva chiama "fail")

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)?

(12)

Def Una grammatica è LL(1) sse ogni casella della tabella di parsing LL(1) contiene al più una produzione.

Teorema G è LL(1) sse per ogni coppia di produzioni distinte con la stessa testa
$A \to \alpha \mid \beta$ (se A avesse le produzioni $A \to \alpha\beta \mid \sigma$ io dovrei fare il procedimento per $\alpha \in \beta$, $\beta \in \sigma$, $\alpha \in \sigma$)

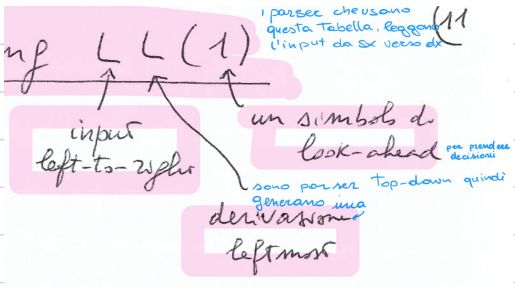
si ha che

1) $First(\alpha) \cap First(\beta) = \emptyset$
2) a) se $\varepsilon \in First(\alpha)$, allora $First(\beta) \cap Follow(A) = \emptyset$
   b) se $\varepsilon \in First(\beta)$, allora $First(\alpha) \cap Follow(A) = \emptyset$

Dim Se sono soddisfatte le condizioni 1) e 2) per ogni coppia di produzioni distinte con medesima testa, allora la tabella di parsing LL(1) contiene al più una produzione in ogni casella. Ma vale anche il viceversa! (NON POSSO INSERIRE DUE PRODUZIONI ALL'INTERNO DELLA MIA TABELLA)

92. Perché un parser di questo tipo è chiamato LL?

(11)



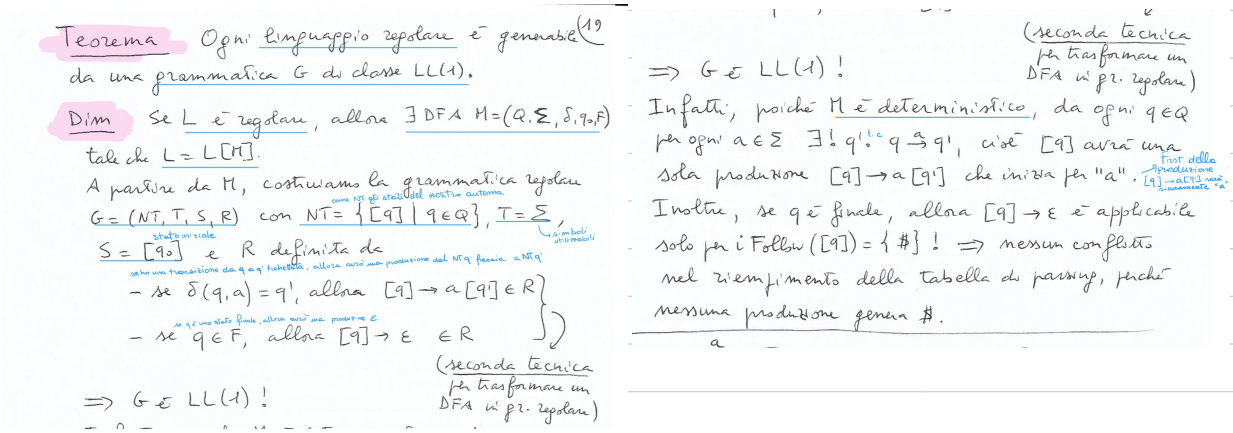
93. Come funziona il parser LL(1) con una pila?

(15)

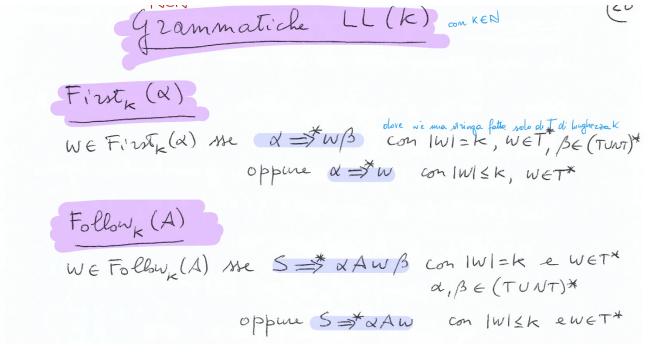
Parser LL(1) non ricorsivo (usando esplicitamente una pila)

- Pila := S$ (cima della pila a sinistra);
- X := S (top della pile);
- input := w$; i_c := primo carattere dell'input;
- While ($X \neq \$$) { // finché la pila non è vuota
  - if (X è un terminale) {
    1) if ($X = i_c$) { // operazione di match
       pop X dalla pila; avanza i_c sull'input;
       }
       else errore(); // caso no match (se G è LL(1) questo caso non si verifica)
    2) X := top della pila }
  - se X non è un terminale { // si verifica
    1) if ($M[X, i_c] = X \to Y_1 \dots Y_n$) { // vado a consultare la tabella in corrispondenza di X e il simbolo corrente in input
       Tolgo pop X dalla pila;
       Metto push $Y_1 \dots Y_n$ sulla pila (Y_1 in cima);
       Ottengo output la produzione $X \to Y_1 \dots Y_n$;
       }
       caso della casella bianca ← else errore(); // caso "bianco"
    2) X := top della pila;
    } continuo a fare il while
  } esce quando la pila è stata svuotata
- if ($i_c \neq \$$) errore(); // caso "ho svuotato la pila ma non ho finito l'input!"

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

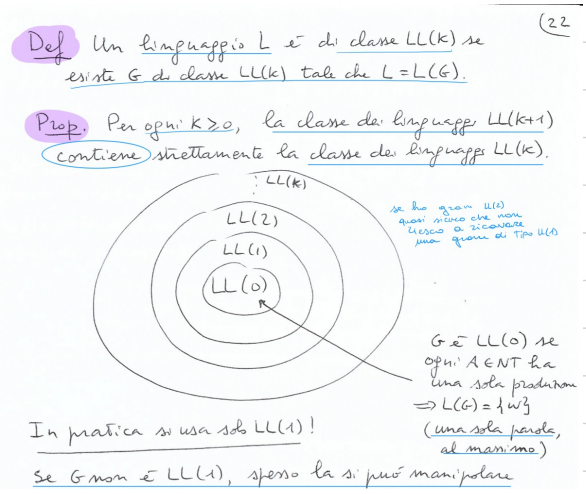


95. Come sono definiti $First_k(\alpha)$ e di $Follow_k(A)$ per k≥2.

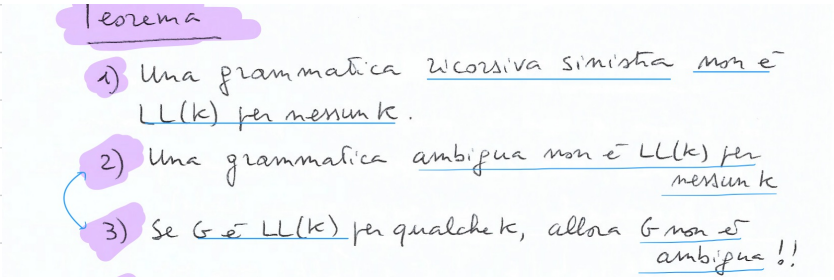


96. Come si definisce una grammatica LL(k)? Ed un linguaggio LL(k)? Come si relazionano tra di loro?

Se la G ammette una Tab con al più una produzione ∀ casella



97. Che relazione esiste tra grammatiche LL(k) e grammatiche ambigue? E con le grammatiche ricorsive sinistre?



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?

4) Se G è LL(K), allora L(G) è libero deterministico!

5) Esiste L libero deterministico tale che non esiste G di classe LL(k) – per nessun k – tale che L = L(G).

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

Le due operazioni fondamentali sono:

- **Shift**: un simbolo terminale viene spostato dall'input sulla pila
- **reduce**: una serie di simboli (terminali e non terminali) sulla cima della pila corrisponde al "reverse" di una parte destra di una produzione $A \to \alpha \in R$ — α^R sulla pila. La stringa α^R viene rimossa dalla pila e sostituita con A ("α viene ridotta ad A")

Riprendiamo la presentazione di un parser shift-reduce nondeterministico, che è essenzialmente un PDA con un solo stato che riconosce per pila vuota il linguaggio L·#

Parser LR: – L (leggo da sx a dx)
– R (derivazione rightmost)

Parser shift-reduce Nondeterministico

Input: – una grammatica libera G con simbolo iniziale S
– una stringa $w \in T^*$

Output: se $w \in L(G)$, allora restituisca la sua derivazione rightmost a rovescio

– Inizializziamo la pila a #;
– Inizializziamo l'input con w#;
– Usiamo il PDA seguente per trovare la derivazione per w#
$M = (T, \{q\}, T \cup \Gamma \cup \{\#\}, \delta, \#, \emptyset)$ (Γ simboli della pila)
dove
1. $(q, aX) \in \delta(q, a, X)$ $\forall a \in T$ $\forall X \in \Gamma$ (SHIFT)
2. $(q, A) \in \delta(q, \varepsilon, \alpha^R)$ se $A \to \alpha \in R$ (REDUCE)
3. $(q, \varepsilon) \in \delta(q, \#, S\#)$ (Accept)

– ogni volta che facciamo "Reduce", forniamo in output la produzione usata
– alla fine, S# sulla pila, con # in input.
⇒ ok, accettiamo

oss: generalizzazione della def. di PDA dove non si consuma solo il top della pila, ma una serie di caratteri contigui a cominciare dal top

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?

- costruzione dell'albero di derivazione bottom-up
- derivazione canonica destra a rovescio
- enorme nondeterminismo:
  - conflitti shift-reduce
    2') $a +b*b$ — shift → se avessi fatto questo sarebbe stata una decisione sbagliata
    3') $a+ b*b$
  - conflitti reduce-reduce
    11') $T+A*T$ — reduce $E \to T$ → perché non mi avrebbe portato a riconoscere la stringa
    12') $T+A*E$

Come risolvere i conflitti?

Strategia: bisogna scegliere l'azione giusta in modo che sulla pila ci sia un **prefisso viabile**

101. **Cos'è un prefisso viabile?** Come lo si definisce in termini di una grammatica?

Def(1) Un prefisso viabile è una stringa $\in (T \cup NT)^*$ che può apparire sulla pila di un parser bottom-up per una computazione che accetta un input.

Def(2) (su grammatica G libera)
Una stringa $\gamma \in (T \cup NT)^*$ è un prefisso viabile per G sse esiste una derivazione rightmost
$S \Rightarrow^* \delta A y \Rightarrow \delta\alpha\beta y = \gamma\beta y$ (A è il variabile non terminale più a dx; βy è una stringa di T*)
per qualche $y \in T^*$, $\delta \in (T \cup NT)^*$ e per una produzione $A \to \alpha\beta$. Inoltre S è un prefisso viabile per definizione.
Un prefisso viabile è completo se $\beta = \varepsilon$; in tal caso α è detta maniglia (handle) per γy,
(ovvero in cima alla pila trovo α^R e posso fare una reduce) (la stringa può essere ridotta al passo precedente)

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

Uno stato del DFA dei prefissi viabili (chiamato automa canonico LR(0)) è costituito da un insieme di item LR(0)

(come costruire: 1) Parto da una gram. 2) costruisco DFA dei suoi prefissi viabili → chiamato anche AUTOMA CANONICO LR(0)) (13)

ITEM LR(0): è una produzione con indicata, con un punto, una posizione della sua parte destra

Es: $A \to XYZ$ genera 4 item

$A \to .XYZ$
$A \to X.YZ$
$A \to XY.Z$
$A \to XYZ.$

Il punto "." indica quanta parte della produzione è già stata analizzata

- se $A \to \alpha.\beta$ (α in cima allo stack dei simboli; β in aspetto che arrivi dopo) è nello stato del DFA in cima alla pila, allora vuol dire che α è sulla pila dei simboli e che ci si aspetta che l'input da leggere contenga (o possa venir ridotto a) β
- se $A \to \alpha.$ è nello stato del DFA in cima alla pila, allora sulla pila dei simboli c'è la maniglia α e possiamo fare la reduce (quella parte destra che è pronta a fare la reduzione)

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

Data $G = (NT, T, S, R)$ libera, prendiamo la grammatica aumentata con un nuovo simbolo iniziale S' ed una produzione $S' \to S$

L'NFA dei prefissi viabili di G (primo passo verso la costruzione del DFA!) si ottiene così:

- $[S' \to .S]$ è lo stato iniziale
- dallo stato $[A \to \alpha.X\beta]$ c'è una transizione allo stato $[A \to \alpha X.\beta]$ etichettata X, per $X \in T \cup NT$
- dallo stato $[A \to \alpha.X\beta]$, per $X \in NT$ e per ogni produzione $X \to \gamma$, c'è una ε-produzione verso lo stato $[X \to .\gamma]$

Oss: non serve definire degli stati finali, perché l'NFA serve solo come ausilio al parser

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)?

Caso LR(0)

Per ogni stato s dell'automa canonico LR(0)

- se $x \in T$ e $s \xrightarrow{x} t$ nell'automa LR(0), inserisci shift t in $M[s,x]$
- se $A \to \alpha. \in s$ e $A \neq S'$, inserisci reduce $A \to \alpha$ in $M[s,x]$ per tutti gli $x \in T \cup \{\$\}$
- se $S' \to S. \in s$, inserisci Accept in $M[s,\$]$
- se $A \in NT$ e $s \xrightarrow{A} t$ nell'automa LR(0), inserisci GOTO t in $M[s,A]$

Def Una grammatica è di classe LR(0) se ogni casella nella tabella di parsing LR(0) contiene al più un elemento!

Tabella di Parsing LR (19)

- Matrice bidimensionale M
  - righe = stati dell'automa canonico LR(0) / LR(1)
  - colonne = $T \cup \{\$\}$ (AZIONI) $\cup$ NT (GOTO)
- $M[s,X]$ contiene le azioni che può compiere un parser LR con s in cima alla pila degli stati e X simbolo in input (o non terminale)
- se $M[s,X]$ è "bianca"/vuota, allora ERRORE
- se $M[s,X]$ contiene più azioni, allora CONFLITTO (il parser non è deterministico)

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

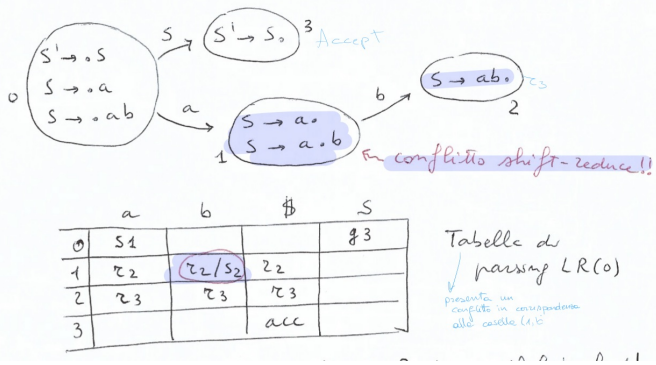
Il generico Parser LR (con solo lo stack degli stati) (22

- Inizializza la pila con $ s_0; % cima della pila a dx
  s_0 stato iniziale dell'automa
- Inizializza ic con il primo carattere di input;
- while (true) { (eseguo per sempre)
  s = top(pila); % top non rimuove la testa
  case M[s, ic] of (carattere corrente dell'input)
  shift t : { push t sulla pila;
  avanza ic sull'input; }
  accept : {output ('accept'); break; } (interrompere l'esecuzione)
  reduce A→α : { pop |α| stati dalla pila; (lunghezza di α)
  s_1 = Top(pila); % s_1 contiene B→γ.Aδ
  sia s_2 = M[s_1, A] % M[s_1, A] è goto s_2
  push s_2 sulla pila;
  output la produzione A→α;
  }
  else errore(); % casella bianca
  }

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

Una grammatica ridotta G può non essere LR(0)!
(facendo delle piccole modifiche potremo renderla deterministica e molto parsabile anche se non sarà mai LR(0))

(1) S'→S (2) S→a (3) S→ab L(G)={a, ab}



	a	b	$	S
0	s1			g3
1	r2	r2/s2	r2	
2	r3	r3	r3	
3			acc	

Tabella di parsing LR(0)
presenta un conflitto in corrispondenza alle caselle (1,b)

Come risolvere il conflitto? Usiamo il look-ahead!
Cioè guardiamo il Follow(S): se b ∈ Follow(S), allora il conflitto è reale! Altrimenti, no!
(la nostra produzione. Sotto siamo a S→a quindi se faccio la ridu sulla pila avrò S e quindi la stessa cosa che ho in input cioè il Follow(S))
(può essere) → non ha senso fare la reduce

Ma Follow(S) = {$} ⇒ risolviamo il conflitto a favore dello shift
(mette la reduce in corrispondenza dei caratteri che trova come Follow della testa di quella produzione)

	a	b	$	S
0	s1			g3
1		s2	r2	
2			r3	
3			acc	

Tabella di parsing SLR(1) (Perché in qualche modo sta considerando anche un simbolo di look-ahead)
↑ simile

- solo per i caratteri nel Follow(S) mettiamo r2

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?

(27

Tabella di Parsing SLR(1) ← simile

- n° colonne: numero ✓ T ∪ {$} ∪ NT (tante)
- n° righe: numero ✓ stati dell'automa canonico LR(0) (tante)

Come si riempie la tabella?

Per ogni stato s dell'automa LR(0)

1. se X∈T e s $\xrightarrow{X}$ t, inserisco shift t in M[s,X]
2. se A→α. ∈ s e A≠S', inserisco reduce A→α in M[s,X] per tutti gli x ∈ Follow(A) (stato)
3. se S'→S. ∈ s, inserisco Accept in M[s, $]
4. se A∈NT e s $\xrightarrow{A}$ t, inserisco goto t in M[s,A] (stato)

• prima, per LR(0), era "per tutti gli x ∈ T∪{$}"

SLR(1): S - simple
L - left-to-right
R - rightmost derivation
1 - un simbolo di look-ahead
(in modo non esplicito, ma attraverso i follow dei nonterminali)

Vedremo che LR(1) usa esplicitamente il look-ahead già nella definizione di item LR(1)

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

Una grammatica libera G potrebbe non essere nemmeno SLR(1)!

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

Item LR(1)

usiamo esplicitamente 1 carattere in avanti dell'input, associato direttamente all'item LR(0) - core

NFA LR(1)

- stati: item LR(1) della grammatica aumentata
- $[S' \to .S, \#]$ è lo stato iniziale
- dallo stato $[A \to \alpha.X\beta, a]$ c'è una transizione allo stato $[A \to \alpha X.\beta, a]$ etichettata X, per $X \in T \cup NT$ — CORRISPONDE ALLO SHIFT
- dallo stato $[A \to \alpha.X\beta, a]$, per $X \in NT$ e per ogni produzione $X \to \gamma$, c'è una ε-transizione verso lo stato $[X \to .\gamma, b]$ per ogni $b \in First(\beta a)$ — CORRISP ALLE CLOS

(N.B. $First(\beta a) \subseteq T \cup \{\#\}$)

Automa Canonico LR(1)

Si può ottenere in 2 modi:

- DFA ottenuto da NFA LR(1) con la costruzione dei sottoinsiemi
- In modo diretto, usando le funzioni Clos(I) e Goto(I, X)

partendo dallo stato iniziale Clos($[S' \to .S, \#]$)

```
Clos(I) {
  ripeti finché I è modificato {
    per ogni item [A → α.Xβ, a] ∈ I
      per ogni produzione X → γ
        per ogni b ∈ First(βa)
          aggiungi [X → .γ, b] a I;
        }
  return I;
}
```

```
Goto(I, X) {
  inizialmente J = ∅;
  per ogni item [A → α.Xβ, a] ∈ I
    aggiungi [A → αX.β, a] a J;
  return Clos(J);
```

Stato iniziale dell'automa canonico LR(1) è Clos($[S' \to .S, \$]$)

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

Per ogni stato s dell'automa canonico LR(1)

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 $M[s, x]$ (solo per x del look-ahead!)
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 $M[s, A]$

- Ogni casella rimasta vuota è un errore
- Def. Una grammatica libera G è di classe LR(1) se ogni casella della sua tabella di parsing LR(1) ha al più un elemento (no conflitti)

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

⇒ La tabella di parsing LALR(1) si ottiene da quella LR(1) fondendo insieme gli stati con lo stesso nucleo
- tante triple quanti gli stati dell'automa LR(0)
- meno "reduce" della tabella SLR(1)

La stessa di LR(1).

112. Esistono grammatiche LALR(1) che non sono SLR(1)? Esistono grammatiche LR(1) che non sono LALR(1)? Mostrare una grammatica che è LR(1) ma non LALR(1).

(42)

Passando da LR(1) a LALR(1)

- La fusione di due stati LR(1) con lo stesso core può causare conflitti.
- Sono possibili solo nuovi conflitti reduce/reduce. Infatti, supponiamo che in S, stato ottenuto per fusione di 2 stati LR(1) s_1 e s_2, presenti un conflitto shift-reduce. Allora, esiste in S un item $[A \to \alpha \cdot, a]$ e un item $[B \to \beta \cdot a\gamma, b]$. Supponiamo, w.l.o.g., che $[A \to \alpha \cdot, a] \in s_1$. Allora $[A \to \alpha \cdot, a]$ e $[B \to \beta \cdot a\gamma, c]$ (per qualche) appartengono ad s_1!

⇒ pure s_1 in LR(1) avrebbe un conflitto shift-reduce, contro l'ipotesi che la tabella LR(1) non presenti conflitti!

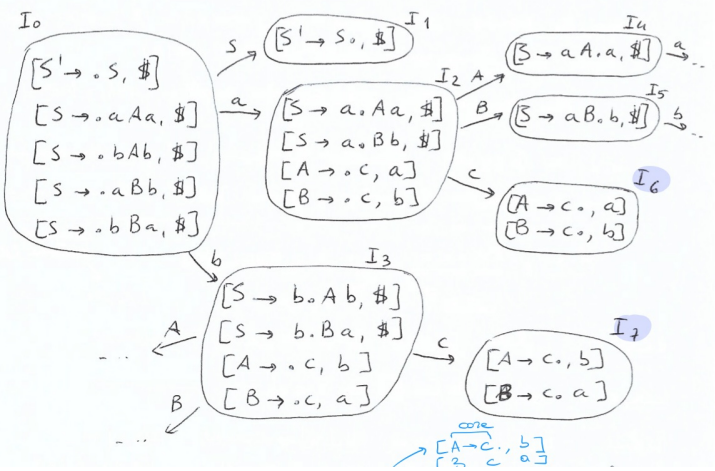
⇒ Se LR(1) è senza conflitti, LALR(1) potrebbe solo presentare conflitti reduce-reduce

Se si generano conflitti, allora G non è LALR(1), pur essendo LR(1).

(43)

Esempio: G è LR(1) ma non LALR(1)

$S' \to S \quad S \to aAa \mid bAb \mid aBb \mid bBa \quad A \to c \quad B \to c$] G



I_6 e I_7 hanno lo stesso core [A → c., b/a] [B → c., a/b] – ma se li fondo

I_{67} [A → c., a/b] [B → c., a/b]

ora è presente un conflitto reduce – reduce

$M'[67, a] = \{ A \to c \text{ (reduce)}, \; B \to c \text{ (reduce)} \}$

$M'[67, b] = \{ A \to c \text{ (reduce)}, \; B \to c \text{ (reduce)} \}$

N.B. Nello stato I_6 non c'è conflitto, e neanche nello stato I_7! ⇒ G è davvero LR(1), ma G non è 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)?

Grammatiche LR(k)

- devo prima definire che cos'è un item LR(k) = [item LR(0), β] (coppia; β = stringa fatta solo di Terminali, lunghezza di β ≤ k) — nella costruzione dell'item LR(k)
- item iniziale = $[S' \to .S, \$]$
- Quando $[A \to \alpha . X\gamma, \beta] \in s$ (stato dell'automa canonico LR(k)) (devo fare Clos(s); calcolo il $First_k(\gamma\beta)$), allora pure $[X \to .\delta, w] \in s$ se $X \to \delta$ è una produzione e $w \in First_k(\gamma\beta)$
- La tab di Parsing avrà delle colonne su T^k (delle stringhe lunghe k) e le righe sugli stati che sono riuscita a costruire dell'automa canonico LR(k)
- Se la tabella di parsing LR(k), ottenuta a partire dall'automa canonico LR(k), contiene al più una azione per ogni entrata, allora G è LR(k)

Grammatiche SLR(k)

Si parte dall'automa canonico LR(0) e si riempie la tabella di parsing SLR(k) – che ha colonne su T^k (stringhe di Terminali lunghe k) – secondo la legge:

2. se $[A \to \alpha .] \in s$ e $A \neq S'$, inserisci "reduce $A \to \alpha$" in $M[s, w]$ per tutti i $w \in Follow_k(A)$

Grammatiche LALR(k)

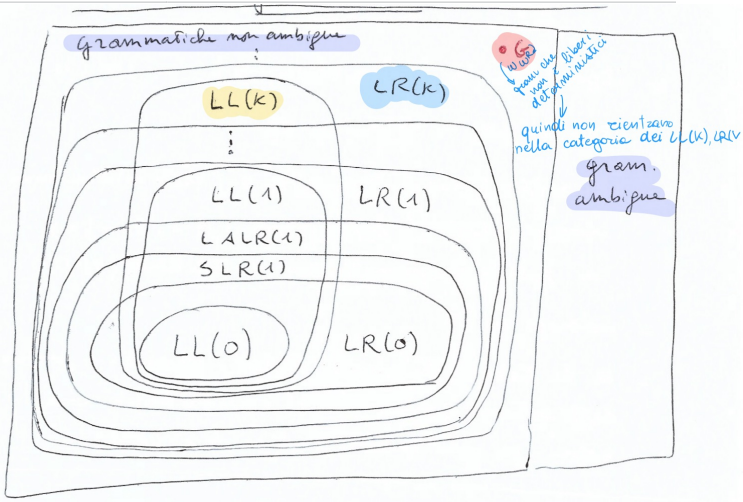
- Si parte dall'automa canonico LR(k) e si fondono insieme gli stati con lo stesso nucleo. Se la tabella risultante non presenta conflitti

⇒ G è 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?**

LR (incluso strettamente):
- $SLR(k) \subset LALR(k) \subset LR(k)$ per ogni $k \geq 1$
- $SLR(1) \subset SLR(2) \subset \dots \subset SLR(k)$
- $LALR(1) \subset LALR(2) \subset \dots \subset LALR(k)$
- $LR(0) \subset LR(1) \subset \dots \subset LR(k)$

LL vs LR:
- $LL(k) \subset LR(k)$ per ogni $k \geq 0$
- $LL(k) \not\subset LR(k-1)$ per ogni $k \geq 1$



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?**

Proposizione

1) Se G è $LL(k)$, allora G è non ambigua e $L(G)$ è deterministico

2) Se G è $LR(k)$, allora G è non ambigua e $L(G)$ è deterministico

Oss: Esistono linguaggi generati da grammatiche non ambigue, ma nondeterministici

$S \to aSa \mid bSb \mid \varepsilon$] G

$L(G) = \{ ww^R \mid w \in \{a,b\}^* \}$

Ma cosa possiamo dire dei linguaggi generati da tali grammatiche?

Def Un linguaggio L è di classe X se $\exists G$ di classe X tale che $L = L(G)$.

(dove X sta per $LR(0)/SLR(1)/LL(1)\ldots$)

Es: na gram. è di classe $LL(1)$ se la tabella di parsing $LL(1)$ non presenta conflitti
Un ling. è di classe $LL(1)$ se $\exists$ una gram. di classe $LL(1)$ che lo genera quel linguaggio

Se classifichiamo i linguaggi, anziché le grammatiche, il diagramma si semplifica molto.

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 e sufficiente affinché un linguaggio sia di classe LR(0)?

- Un ling. è libero deterministico se è accettato, per stato finale, da un DPDA
- Ogni ling. regolare è generato da una gram. di classe LL(1)
- Esistono ling. regolari che non sono LR(0) (ad es: $L = \{a, ab\}$ vedi pg. 23)

(1) Un ling. è libero det. sse è generato da una gram. LR(k) per qualche $k \geq 0$

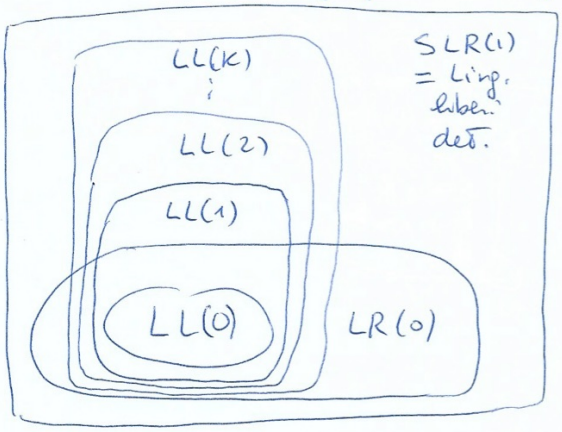
117. La classe dei linguaggi SLR(1) coincide con la classe dei linguaggi liberi deterministici?

Teoremi

(1) Un ling. è libero det. sse è generato da una gram. LR(k) per qualche $k \geq 0$

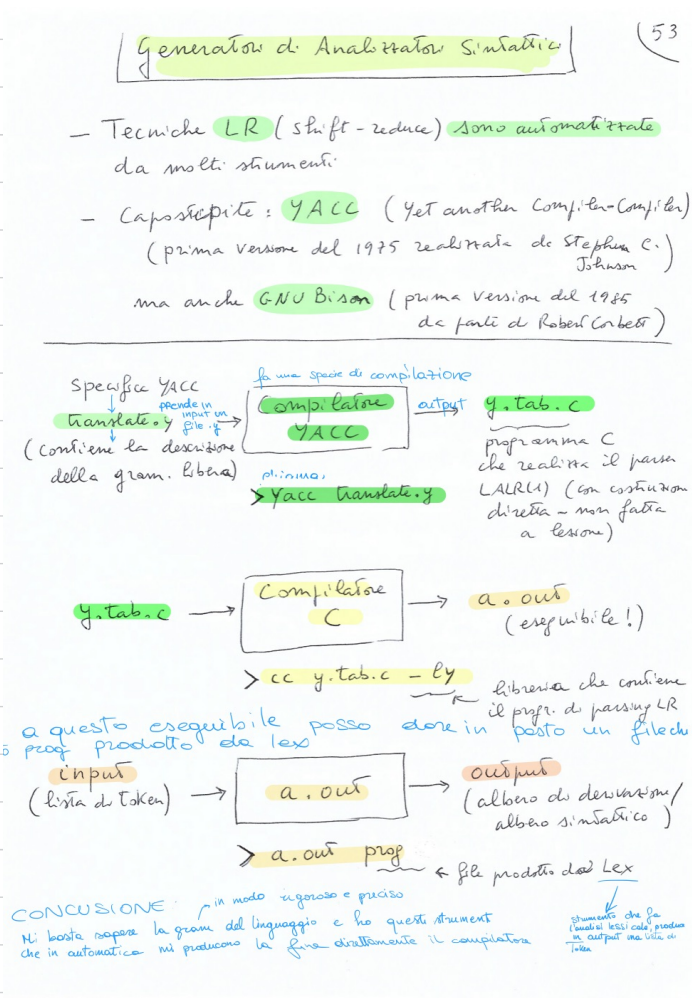
(2) Un ling. è libero det. sse è generato da una gram. SLR(1)

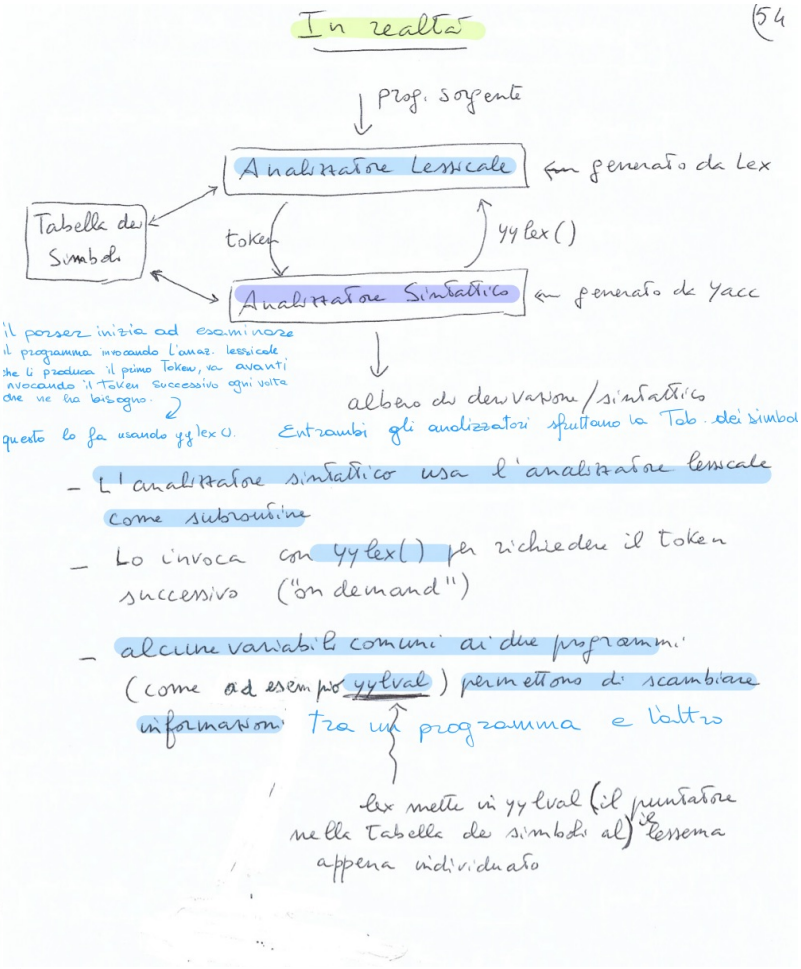
(3) I ling. generati da gram. LL(k) sono strettamente contenuti nei ling. generati da gram. SLR(1), $\forall k \geq 0$



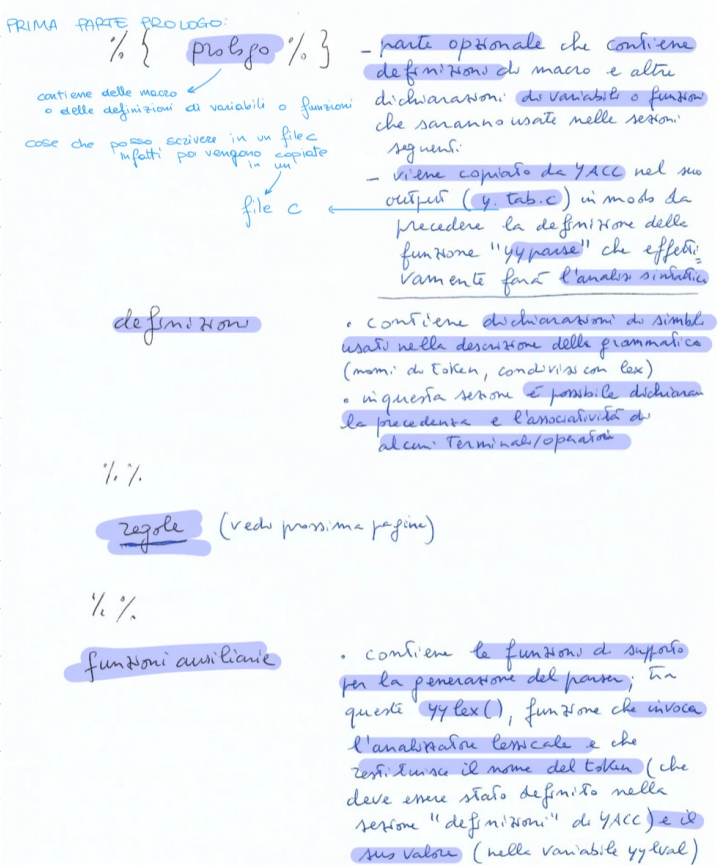
Oss: Se $G \in LR(k)$, esiste $G' \in SLR(1)$ equivalente, ma G' può essere molto più complessa di G

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?





119. **Come è la struttura di un file .y di YACC?** Cosa sono le regole? Cos'è l'azione semantica?



Le regole ... dove noi descriviamo la grammatica

(56)

Una produzione della forma

nonterm → corpo$_1$ | ... | corpo$_k$

è espressa in YACC con le regole

nonterm : corpo$_1$ {azione semantica$_1$}
...
| corpo$_k$ {azione semantica$_k$}
;

Se c'è corpo$_i$ e devo fare una riduzione e trasformarlo nel NT iniziale, allora l'"azione semantica" dice quale parte di codice C deve eseguire

Codice C eseguito al momento in cui il parser riduce mediante quella produzione

è una stringa alfanumerica non dichiarata in precedenza come token

- un carattere tra singoli apici, 'a', è un terminale!
- il simbolo iniziale della grammatica è il nonterm usato nella prima regola

L'azione semantica

(57)

L'azione semantica (codice C) calcola il "valore semantico" della Testa della "produzione" in funzione dei valori semantici dei simboli che compongono il corpo.

Es: Il "valore semantico" potrebbe essere:

(a) l'albero di derivazione, nel caso in cui stiamo producendo un compilatore che genera alberi esplicitamente.

(b) il codice intermedio, connesso alla produzione, se andiamo direttamente a produrre codice intermedio

(c) la vera e propria valutazione dell'espressione, se stiamo in realtà producendo un interprete

In una azione semantica:

NOTAZIONE
- $$ si riferisce al valore semantico della Testa
- $i si riferisce al valore semantico dell'i-esimo simbolo nel corpo della produzione

Vediamo un esempio di file .y, che lavora di concerto con Lex, perché nelle funzioni ausiliarie c'è #include "lex.yy.c" vuol dire che i due stanno lavorando insieme

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?

(60)

La Tabella LALR(1) che YACC genera, poiché la grammatica è - di base - ambigua, potrebbe generare conflitti. Ma usando le informazioni aggiuntive su
- associatività
- precedenza degli operatori

è possibile risolvere tutti i conflitti!!

In generale YACC, in assenza di indicazioni, risolve
- conflitti di tipo shift/reduce a favore dello shift → questo se non ha indicazioni, nei es precedenti avevamo indicazioni ad es. %left ... %right
- conflitti di tipo reduce/reduce a favore della produzione elencata prima

E' possibile invocare YACC con l'opzione -v
Questa opzione genera un file aggiuntivo y.output che contiene i Kernel degli insiemi di item trovati per la grammatica, una descrizione dei conflitti generati dall'algoritmo LALR, ed anche una rappresentazione leggibile della tabella di parsing che mostra come i conflitti sono stati risolti.

CAPITOLO 5

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?

$$\text{check}(P) = \begin{cases} 1 & \text{se P è corretta} \\ 0 & \text{se P presenta errori} \end{cases}$$

Vediamo un caso specifico, in cui l'**errore** è la **non-terminazione**: se il programma è scritto in un **ling. sequenziale**, ci si aspetta che il suo **calcolo Termini sempre**.

Dato un ling. di programmazione $\mathcal{L}$, proviamo a scrivere in $\mathcal{L}$ un programma H che calcola la seguente funzione

$$H(P, x) = \begin{cases} 1 & \text{se } P(x)\downarrow \text{ "termina"} \\ 0 & \text{se } P(x)\uparrow \text{ "diverge"} \end{cases}$$

(P: programma; x: input che diamo al programma)

Oss: Per rispondere "0", il programma H deve riconoscere **in tempo finito** che il prog. P con input x non terminerà mai il calcolo!

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?**

Ma può esistere un programma H siffatto?

Questo problema è noto in informatica come "**problema della fermata**"

HALTING PROBLEM ← proposto da Alan Turing

→ problema che non può essere risolto meccanicamente, quindi da nessun algoritmo

(3)

- Supponiamo, per assurdo, che H esista davvero.
- Allora, usando H, possiamo realizzare l'applicazione

$$K(P) = \begin{cases} 1 & \text{se } P(P)\downarrow \\ 0 & \text{se } P(P)\uparrow \end{cases}$$

(P(P): Programma che prende in input un altro programma)

se c'è un algoritmo per H allora c'è anche per K $= H(P, P)$

↑ qui P è usato come dato (che il P) usa.

(Vedi compilatori: programmi che usano altri programmi come dati)

Oss: Se H esiste, allora esiste anche K!

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

$$G(P) = \begin{cases} 1 & \text{se } K(P) = 0 \\ \uparrow \text{ diverge} & \text{se } K(P) = 1 \end{cases}$$

Oss: Se K esiste, allora anche G è facilmente programmabile!

- Ma cosa succede se a G diamo in input G?

$$\begin{cases} - \; G(G) = 1 & \text{se } K(G) = 0 \text{ se } G(G)\uparrow \text{ diverge} \\ - \; G(G) = \uparrow \text{ diverge} & \text{se } K(G) = 1 \text{ se } G(G)\downarrow \text{ converge} \end{cases}$$

contraddizione

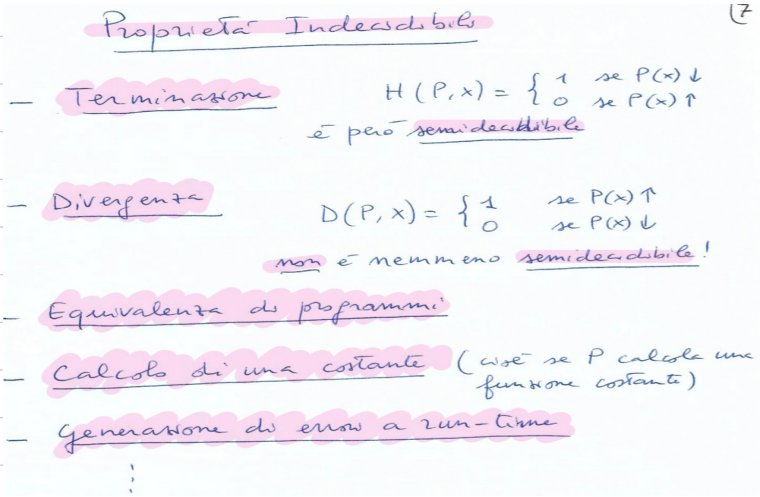
ASSURDO! ⇒ G non può esistere ⇒ K non può esistere ⇒ H non può esistere!

H è il primo esempio di funzione non calcolabile (o di problema non risolvibile – Turing 1936)

123. Quando un problema è decidibile?

Quando ha algoritmi che funzionano per argomenti arbitrari, risponde sì o no in tempo finito

124. Quali sono tipici esempi di proprietà indecidibili per i linguaggi di programmazione?

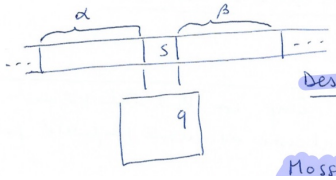


125. Cos'è una Macchina di Turing (MdT)? Che cosa calcola?

MdT $M = (Q, A, B, \delta, q_0, q_f)$ dove

- Q è un insieme finito di stati
- A è l'alfabeto finito dell'input (tipicamente le cifre 0-9)
- B è l'alfabeto finito del nastro ($A \subset B$, $\square \in B$) ← casella vuota
- q_0 è lo stato iniziale
- q_f è lo stato finale
- $\delta : Q \times B \longrightarrow Q \times B \times \{sx, dx\}$ ← funzione parziale, MdT deterministica

tale che $\delta(q_f, b)$ è indefinita $\forall b \in B$



nastro infinito!

Descrizione istantanea (forma sintattica)

$\alpha q s \beta$

Mosse:

Mosse:

- $\alpha q s \beta \vdash \alpha s' q' \beta$ se $\delta(q,s) = (q', s', dx)$
- $\alpha \bar{s} q s \beta \vdash \alpha q' \bar{s} s' \beta$ se $\delta(q,s) = (q', s', sx)$

(stato in cui mi trovo, cosa sto leggendo; carattere su cui si è posizionata la testina)

$$L[M] = \{ w \in A^* \mid q_0 w \overset{*}{\vdash} \alpha q_f \beta \}$$

facendo più mosse

ma può non raggiungere mai né q_f, né uno stato di blocco "erroneo", cioè può divergere

(9

Una Mdt (macchina di Turing) deterministica su alfabeto $A = [0-9]$ può essere vista come una macchina che calcola funzioni parziali binarie:

$$f_M(w) = \begin{cases} 1 & \text{se } q_0 w \vdash^* \alpha q_f \beta \\ 0 & \text{se } q_0 w \vdash^* \alpha q' \beta \not\vdash \text{ e } q' \neq q_f \\ \uparrow & \text{altrimenti} \end{cases}$$

stato bloccato e non è uno stato finale

Se consideriamo l'insieme di tutte le funzioni parziali a valori binari

$$\mathcal{F} = \{ f \mid f : \mathbb{N} \longrightarrow \{0,1\} \}$$

possiamo concludere che $f \in \mathcal{F}$ è Turing-calcolabile se $\exists$ MdT M tale che $f_M = f$.

un insieme proprio è molto piccolo, quindi sono calcolabili pochissime funzioni

126. Quando un linguaggio è detto Turing-completo? Cosa afferma la tesi di Church-Turing e perché non si può dimostrare?

(13)

Formalismo Turing-completo: ha la stessa potenza espressiva delle MdT — calcola le stesse funzioni calcolabili con MdT.

Tutti i Turing-completi, perché sia possibile usare tutta la memoria di cui possono necessitare.

Tesi di Church-Turing (1936-1937)

Se una funzione può essere calcolata algoritmicamente in un qualche formalismo, allora è calcolabile con MdT.

- Tesi perché:
  - non c'è una definizione formale di cosa è "algoritmicamente" calcolabile
  - è implicata una quantificazione universale su tutti i possibili formalismi! Infinite prove? E se domani uno arriva con un nuovo formalismo? ...
- Considerata vera perché in 80 anni nessuno è riuscito a confutarla !!
- Criterio di equivalenza tra linguaggi sequenziali: Se L_1 e L_2 sono entrambi Turing-completi, allora sono equamente espressivi per la Tesi di Church-Turing
- Nel caso dei ling. concorrenti, la questione è diversa perché i programmi concorrenti non calcolano solo funzioni, ma risolvono problemi, offrono servizi, ecc...

127. I normali linguaggi di programmazione sono Turing-completi? Cosa afferma il teorema di Jacopini-Bohm?

Teorema di Jacopini-Böhm (1966)

Un ling. di program. imperativo che contenga
- if-then-else (condizionale)
- while (iterazione indeterminata)
- ; (composizione sequenziale)

(+ istruzione di assegnamento) è Turing-completo!

Oss: il ling. usato per spiegare la semantica SOS è Turing-completo!

Oss: Questo teorema ha avuto un benefico effetto nel promuovere la "programmazione strutturata", cioè quel principio di programmazione secondo il quale un ling. doveva contenere solo operatori ben strutturati, la cui semantica era definibile "localmente", solo guardando i propri argomenti (Non è così per il GOTO!)

128. Quale relazione esiste tra espressività di un formalismo e decidibilità di proprietà dello stesso? Fare una panoramica prendendo in esame i tre formalismi MdT, PDA, DFA, e le due proprietà $w \varepsilon L(M)$ (è w riconosciuta dalla macchina M?) e $L(M_1) = L(M_2)$ (le due macchine sono equivalenti, ovvero riconoscono lo stesso linguaggio?).

Gerarchia di Macchine

Espressività vs. **Analizzabilità**

più espressivo, meno è analizzabile

MdT $\longleftrightarrow$ gram. generali $\longleftrightarrow$ $w \in L(M)/L(G)$?
è solo semidecidibile

Ma abbiamo visto formalismi più deboli ma significativi

PDA $\longleftrightarrow$ gr. liberi $\longleftrightarrow$ $w \in L[N]/L(G)$?
è <u>decidibile</u> !
Ma il problema dell'equival.

$$E(G_1, G_2) = \begin{cases} 1 & \text{se } L(G_1) = L(G_2) \\ 0 & \text{altrimenti} \end{cases}$$

è <u>indecidibile</u> !

DFA $\longleftrightarrow$ gr. regolari $\longleftrightarrow$ $E(G_1, G_2)$ è decidibile !

<u>Oss</u>:
- circuiti logici sono DFA
- vending machine sono DFA
- funzioni find/replace in text-editors sono DFA

$\Rightarrow$ esistono molte "funzioni" interessanti che si possono calcolare in formalismi <u>non</u> Turing-completi!
E su questi possiamo decidere molte proprietà interessanti!