



Capitolo 2

Linguaggi e Grammatiche

Corso di Laurea Magistrale in Ingegneria Informatica

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)



COS'È UN LINGUAGGIO?

Dice il dizionario:

“Un linguaggio è un *insieme di parole* e di *metodi di combinazione* delle parole usate e *comprese da una comunità di persone*.”

È una definizione *poco precisa*:

- *non evita le ambiguità* dei linguaggi naturali
- non si presta a descrivere processi computazionali *meccanizzabili*
- *non aiuta a stabilire proprietà*



LA NOZIONE DI LINGUAGGIO

- Occorre una nozione di linguaggio *più precisa*
- Linguaggio come *sistema formale* che consenta di rispondere a domande come:
 - quali sono le *frasi lecite*?
 - si può stabilire se una *frase appartiene al linguaggio*?
 - come si stabilisce il *significato* di una frase?
 - *quali elementi linguistici primitivi* ?



SINTASSI & SEMANTICA

- **Sintassi**: l'insieme di *regole formali* per la scrittura di frasi corrette («programmi») in un linguaggio, che dettano le *modalità per costruire frasi corrette* nel linguaggio stesso.
- **Semantica**: l'insieme dei significati da attribuire alle frasi (sintatticamente corrette) del linguaggio.

MA:

Una frase può essere *sintatticamente corretta* e tuttavia *non avere significato*!

SINTASSI & SEMANTICA ☺



Una frase può essere *sintatticamente corretta* e tuttavia *non avere significato*!



SINTASSI & SEMANTICA

- La *sintassi* è solitamente espressa tramite *notazioni formali* come
 - BNF, EBNF
 - diagrammi sintattici
- La *semantica* è esprimibile:
 - *a parole* (poco precisa e ambigua)
 - mediante *azioni*
 - *semantica operativa*
 - mediante *funzioni matematiche*
 - *semantica denotazionale*
 - mediante formule logiche
 - *semantica assiomatica*



INTERPRETAZIONE vs COMPILAZIONE

Un *interprete* per un linguaggio L:

- accetta in ingresso **le singole frasi** di L
- e **le esegue una per volta**.

Il risultato è la *valutazione* della frase.

Un *compilatore* per un linguaggio L, invece:

- accetta in ingresso **un intero programma scritto in L**
- e **lo riscrive in un altro linguaggio** (più semplice).

Il risultato è dunque una *riscrittura* della "macro-frase".

A volte la differenza è più sfumata di quel che si può pensare..



ANALISI LESSICALE & SINTATTICA

- L'*analisi lessicale* consiste nella individuazione delle singole parole (*token*) di una frase
 - L'analizzatore lessicale (detto *scanner* o *lexer*), data una sequenza di caratteri, li aggrega in token di opportune categorie (nomi, parole chiave, simboli di punteggiatura, etc.)
- L'*analisi sintattica* consiste nella verifica che la frase, intesa come *sequenza di token*, rispetti le regole grammaticali del linguaggio.
 - L'analizzatore sintattico (detto *parser*), data la sequenza di token prodotta dallo scanner, genera una rappresentazione interna della frase – solitamente sotto forma di *opportuno albero*.



ANALISI SEMANTICA

- L'*analisi semantica* consiste nel determinare il significato di una frase
 - L'analizzatore semantico, data la rappresentazione intermedia prodotta dal parser, controlla la *coerenza logica* della frase
 - se le variabili sono usate solo dopo essere state definite
 - se sono rispettate le regole di compatibilità in tipo
 - ...
 - Può anche trasformare ulteriormente la rappresentazione delle frasi in una forma più adatta alla generazione finale di codice.
- Già, ma... *cos'è il "significato"* di una frase?

SIGNIFICATO DI UNA FRASE

- Chiedersi quale sia il *significato* di una frase significa *associare a quella frase un concetto* nella nostra mente
 - Lo facciamo in base alla nostra cultura ed esperienza di vita

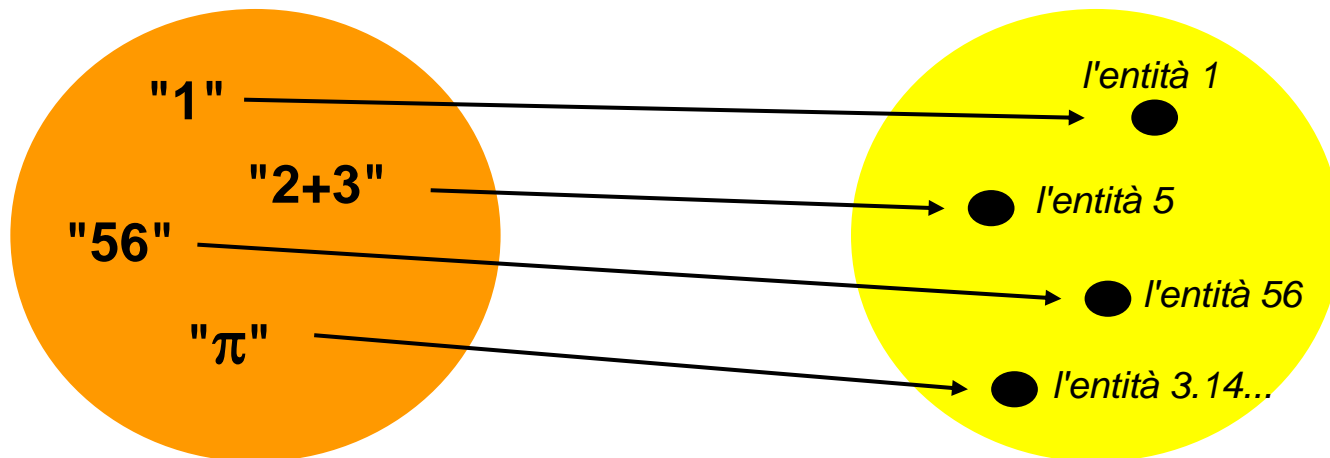
Ad esempio,
se siamo italiani la stringa "*spaghetti pomodoro e basilico*" (frase)
verrà probabilmente associata dalla nostra mente al *concetto* di



SIGNIFICATO DI UNA FRASE

- Per farlo, nella nostra mente deve evidentemente esserci una *funzione* che *associa a ogni frase*
 - cioè a ogni *stringa di caratteri* *lecita nel linguaggio*
- *un concetto*
 - cioè un *elemento* di un qualche *dominio*

Ad esempio, se il dominio è la *matematica*, la funzione potrebbe essere:





SIGNIFICATO DI UNA FRASE

- Tale funzione deve quindi dare significato:
 - prima a ogni *simbolo* (*carattere dell'alfabeto*)
 - poi a ogni *parola* (*sequenza lecita di caratteri*)
 - infine a ogni *frase* (*sequenza lecita di parole*).
- Nel caso dell'esempio:
 - l'alfabeto potrebbe consistere nei simboli "1", "2", .. "9"
se consideriamo la nostra cultura attuale
→ *ma Giulio Cesare avrebbe scelto "I", "V", "X", ...*
 - le parole potrebbero essere sequenze di tali simboli, come "51",
da intendersi ovviamente secondo la nostra cultura
 - "51" per noi rappresenta il concetto *cinquantuno*..
 - ...*ma per Giulio Cesare "VI" avrebbe rappresentato l'entità sei !*

DEFINIZIONI

Alfabeto

- un alfabeto A è un **insieme finito e non vuoto di simboli atomici**. Esempio: $A = \{ a, b \}$

Stringa

- un stringa è una **sequenza di simboli**, ossia un elemento del prodotto cartesiano A^n .
Esempi: a ab aba $bb \dots$
- **Lunghezza** di una stringa: il **numero di simboli** che la compongono.
- **Stringa vuota ε** : stringa di lunghezza zero.
⇒ Si noti che $A^0 = \{ \varepsilon \}$



DESCRIZIONE DI UN LINGUAGGIO

Linguaggio L su un alfabeto A

- Un linguaggio L è **un insieme di stringhe** su A
- **Frase (sentence)** di un linguaggio: **una stringa** appartenente a quel linguaggio.
- **Cardinalità** di un linguaggio: il **numero delle frasi** del linguaggio
 - linguaggio *finito*: ha cardinalità *finita*
 - linguaggio *infinito*: ha cardinalità *infinita*

Esempi:

$L1 = \{ aa, baa \}$

linguaggio a cardinalità finita

$L2 = \{ a^n, n \text{ primo} \}$

linguaggio a cardinalità infinita

$L3 = \{ a^n b^n, n > 0 \}$

linguaggio a cardinalità infinita



DESCRIZIONE DI UN LINGUAGGIO

Chiusura A^* di un alfabeto A (o ling. universale su A)

- È l'insieme infinito di tutte le stringhe composte con simboli di A :

$$A^* = A^0 \cup A^1 \cup A^2 \cup \dots$$

Chiusura positiva A^+ di un alfabeto A

- È l'insieme infinito di tutte le stringhe non nulle composte con simboli di A :

$$A^+ = A^* - \{ \varepsilon \}$$



SPECIFICA DI UN LINGUAGGIO

- Problema: **come specificare** il sotto-insieme di A^* che definisce ***uno specifico linguaggio?***
 - per specificare un linguaggio **finito**,
basta ovviamente *elencarne tutte le frasi*
 - per specificare un linguaggio **infinito**, invece,
serve una qualche **notazione** capace di **descrivere
in modo finito un insieme infinito** di elementi.
- Nasce così la nozione di ***grammatica formale***

GRAMMATICA FORMALE

Una *Grammatica* è una *notazione formale* con cui esprimere in modo rigoroso la *sintassi* di un linguaggio.

Una grammatica è una *quadrupla* $\langle VT, VN, P, S \rangle$ dove:

- VT è un *insieme finito di simboli terminali*
- VN è un *insieme finito di simboli non terminali*
- P è un *insieme finito di produzioni*, ossia di *regole di riscrittura* $\alpha \rightarrow \beta$ dove α e β sono *stringhe*: $\alpha \in V^+$, $\beta \in V^*$
 - ogni regola esprime una trasformazione lecita che permette di scrivere, *nel contesto di una frase data*, una stringa β al posto di un'altra stringa α .
- S è un *particolare simbolo non-terminale* detto *simbolo iniziale* o *scopo* della grammatica.

GRAMMATICA FORMALE

Una **Grammatica** è una *quadrupla* $\langle VT, VN, P, S \rangle$ in modo rigoroso la **sintassi** di una lingua L .
I simboli **terminali** sono **caratteri** o **stringhe** su un alfabeto A .

Una grammatica è una *quadrupla* $\langle VT, VN, P, S \rangle$, dove:

- VT è un insieme finito di simboli **terminali**
- VN è un insieme finito di simboli **non terminali**
- P è un insieme finito di **prodotti**, ossia di **regole di**

I simboli **non terminali** sono dei **meta-simboli** che rappresentano le diverse **categorie sintattiche**.
 $\alpha \in V^+, \beta \in V^*$
che permette di
sostituire, nel contesto di una frase data, una stringa β al posto di

- Gli insiemi VT e VN devono essere **disgiunti**: $VT \cap VN = \emptyset$
- L'unione $VT \cup VN$ si dice **vocabolario** della grammatica.



GRAMMATICHE: CONVENZIONI

CONVENZIONI SUI SIMBOLI

Nelle formule teoriche, per comodità:

- i simboli *terminali* si indicano con lettere **minuscole**
- i *meta-simboli* si indicano con lettere **MAIUSCOLE**
- le *lettere greche* indicano *stringhe mixed di terminali e meta-simboli*

CONVENZIONI SULLE PRODUZIONI

- una produzione $\alpha \rightarrow \beta$ riscrive una stringa non nulla $\alpha \in V^+$ sotto forma della nuova stringa (eventualmente anche nulla) $\beta \in V^*$



FRASI (sentences) vs. FORME DI FRASI (sentential forms)

- Si dice forma di frase (*sentential form*) una qualsiasi stringa *comprendente sia simboli terminali sia meta-simboli*, ottenibile dallo scopo applicando una o più regole di produzione.
 - una sentential form è un *prodotto intermedio*, in cui alcune parti della (futura) frase sono già finali, mentre altre sono ancora "in itinere", soggette a ulteriori trasformazioni.
- Si dice frase una forma di frase *comprendente solo simboli terminali*.
 - una sentence è invece un *prodotto finale*, in cui tutte le parti "in itinere" sono state ormai trasformate e non c'è più nulla di ulteriormente trasformabile.

DERIVAZIONE

Siano α, β due *stringhe* $\in (VN \cup VT)^*$, $\alpha \neq \varepsilon$

Si dice che β *deriva direttamente* da α ($\alpha \rightarrow \beta$) se

- le stringhe α, β si possono *decomporre* in

$$\alpha = \eta A \delta$$

$$\beta = \eta \gamma \delta$$

- ed esiste la produzione $A \rightarrow \gamma$.

Si dice che β *deriva da* α (anche non direttamente) se

- esiste una *sequenza di N derivazioni dirette* che da α possono infine produrre β

$$\alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \dots \rightarrow \alpha_N = \beta$$



SEQUENZA DI DERIVAZIONE

Si dice *sequenza di derivazione* la sequenza di passi che producono una *forma di frase* σ dallo scopo S .

$S \Rightarrow \sigma$ σ deriva da S con una sola applicazione di produzioni (in un solo passo)

$S \Rightarrow^+ \sigma$ σ deriva da S con una o più applicazioni di produzioni (in uno o più passi)

$S \Rightarrow^* \sigma$ σ deriva da S con zero o più applicazioni di produzioni (in zero o più passi)



GRAMMATICA & LINGUAGGIO

Data una grammatica G , si dice perciò

Linguaggio L_G generato da G

l'insieme delle frasi

- derivabili dal **simbolo iniziale S**
- applicando le **produzioni P**

ossia

$$L_G = \{ \mathbf{s} \in \mathbf{VT}^* \text{ tale che } S \xRightarrow{*} \mathbf{s} \}$$

ESEMPIO 1

Il linguaggio $L = \{ a^n b^n, n > 0 \}$ può essere descritto dalla grammatica $G = \langle VT, VN, P, S \rangle$ dove:

- $VT = \{ a, b \}$
- $VN = \{ F \}$
- $S \in VN = F$
- $P = \{$
 - $F \rightarrow a b$
 - $F \rightarrow a F b$ $\}$
- La prima regola stabilisce che F può essere riscritto come ab : è la frase più corta di L .
- La seconda regola stabilisce che lo scopo F può essere riscritto come aFb ; data la presenza di F nella forma di frase, è possibile proseguire con un nuovo passo generativo – di nuovo scegliendo *una qualsiasi* delle due regole:
 - se si sceglie la prima, si avrà $aabb$
 - se si sceglie la seconda, si avrà $aaFbb$, che apre la porta a un terzo passo.. e così via.
- Il linguaggio contiene dunque infinite frasi, tutte della forma $aa...bb$ con egual numero di a e b .



GRAMMATICHE EQUIVALENTI

- Una grammatica G1 è *equivalente* a una grammatica G2 se *generano lo stesso linguaggio*
 - una grammatica potrebbe però essere *preferibile a un'altra* ad essa equivalente al punto di vista dell'analisi sintattica
- Purtroppo, stabilire se due grammatiche sono *equivalenti* è in generale un problema *indecidibile*
 - le faccenda cambia se ci si restringe a *tipi particolari* di grammatiche, aventi regole di produzione "*sufficientemente semplici*".



GRAMMATICHE, LINGUAGGI & AUTOMI RICONOSCITORI

Grammatiche di diversa struttura

comportano

linguaggi con diverse proprietà

e implicano

automati di diversa “potenza computazionale”

per *riconoscere* tali linguaggi.



CLASSIFICAZIONE DI CHOMSKY TIPO 0

Le grammatiche sono classificate in *4 tipi*
in base alla *struttura delle produzioni*

- Tipo 0:
nessuna restrizione sulle produzioni

In particolare, le regole possono specificare riscritture che
accorciano la forma di frase corrente.

Esempio (grammatica di tipo 0)

$S \rightarrow aSBC$ $CB \rightarrow BC$ $SB \rightarrow bF$ $FB \rightarrow bF$
 $FC \rightarrow cG$ $GC \rightarrow cG$ $G \rightarrow \varepsilon$

Possibile derivazione: $S \rightarrow aSBC \rightarrow abFC \rightarrow \underline{abcG} \rightarrow \underline{abc}$
lung=4 lung=3



CLASSIFICAZIONE DI CHOMSKY

TIPO 1

Le grammatiche sono classificate in *4 tipi*
in base alla *struttura delle produzioni*

- Tipo 1 (*dipendenti dal contesto*):
produzioni vincolate alla forma:

$$\beta A \delta \rightarrow \beta \alpha \delta$$

con $\beta, \delta, \alpha \in (VT \cup VN)^*$, $A \in VN$, $\alpha \neq \varepsilon$

Quindi, A può essere sostituita da α solo nel contesto $\beta A \delta$

Le riscritture non accorciano mai la forma di frase corrente.

Una **definizione alternativa equivalente** (a parte la generazione della stringa vuota) prevede infatti produzioni della forma $\alpha \rightarrow \beta$ con $|\beta| \geq |\alpha|$

ESEMPIO

Esempio (grammatica di tipo 1)

S \rightarrow **aBC** | **aSBC**

CB \rightarrow **DB**

DB \rightarrow **DC**

DC \rightarrow **BC**

aB \rightarrow **ab**

bB \rightarrow **bb**

bC \rightarrow **bc**

cC \rightarrow **cc**

Infatti, secondo la definizione $\beta \mathbf{A} \delta \rightarrow \beta \alpha \delta$ si può trasformare un metasimbolo per volta (**A**), lasciando intatto ciò che gli sta intorno:

Osserva: la lunghezza del lato destro delle produzioni non è mai inferiore a quella del lato sinistro.

S \rightarrow **aBC** | **aSBC**

CB \rightarrow **DB**

DB \rightarrow **DC**

DC \rightarrow **BC**

aB \rightarrow **ab**

bB \rightarrow **bb**

bC \rightarrow **bc**

cC \rightarrow **cc**

$\beta = \epsilon$ $\delta = \epsilon$

$\beta = \epsilon$ $\delta = \mathbf{B}$

$\beta = \mathbf{D}$ $\delta = \epsilon$

$\beta = \epsilon$ $\delta = \mathbf{C}$

$\beta = \mathbf{a}$ $\delta = \epsilon$

$\beta = \mathbf{b}$ $\delta = \epsilon$

$\beta = \mathbf{b}$ $\delta = \epsilon$

$\beta = \mathbf{c}$ $\delta = \epsilon$

OSSERVAZIONE

La definizione di Chomsky: $\beta A \delta \rightarrow \beta \alpha \delta$

fa capire perché queste grammatiche siano definite *dipendenti dal contesto* (o *contestuali*).

LA DEFINIZIONE ALTERNATIVA: $\alpha \rightarrow \beta$ con $|\beta| \geq |\alpha|$

esprime lo stesso concetto in modo *più pratico*,
ma non esplicita più l'idea di contesto.

Formalmente, essa *ammette produzioni vietate dalla definizione di Chomsky*, come ad esempio $BC \rightarrow CB$

tuttavia, *esiste sempre una grammatica equivalente che rispetta la definizione di Chomsky*, ad esempio $BC \rightarrow BD$; $BD \rightarrow CD$; $CD \rightarrow CB$

quindi *i due formalismi sono equivalenti*

[*purché* la definizione originale non venga arricchita ammettendo la produzione $S \rightarrow \varepsilon$, che la definizione alternativa non può esprimere].



CLASSIFICAZIONE DI CHOMSKY

TIPO 2

Le grammatiche sono classificate in *4 tipi*
in base alla *struttura delle produzioni*

- Tipo 2 (*libere dal contesto*):
produzioni vincolate alla forma:

$$A \rightarrow \alpha$$

con $\alpha \in (VT \cup VN)^*$, $A \in VN$

**Attenzione: non c'è
più il vincolo $\alpha \neq \varepsilon$**

Qui A può sempre essere sostituita da α , *indipendentemente dal contesto*, giacché non esiste più l'idea stessa di contesto.

CASO PARTICOLARE: se α ha la forma u oppure $u B v$, con $u, v \in VT^*$ e $B \in VN$, la grammatica si dice *lineare*.

CLASSIFICAZIONE DI CHOMSKY

TIPO 3

Le grammatiche sono classificate in *4 tipi*
in base alla *struttura delle produzioni*

- Tipo 3 (*grammatiche regolari*):
produzioni vincolate alle forme lineari:

si sviluppano solo
a destra o sinistra

lineare a destra

$$A \rightarrow \sigma$$

$$A \rightarrow \sigma B$$

con $A, B \in VN$, e $\sigma \in VT^*$

lineare a sinistra

$$A \rightarrow \sigma$$

$$A \rightarrow B \sigma$$

IMPORTANTE: le produzioni di una data grammatica devono essere o **tutte** lineari a destra, o **tutte** lineari a sinistra – non mischiate.

Si noti che anche qui σ può essere ϵ .



QUALI MACCHINE PER QUALI LINGUAGGI?

Chi riconosce i diversi tipi di linguaggi?

GRAMMATICHE	AUTOMI RICONOSCITORI
<ul style="list-style-type: none">• Tipo 0• Tipo 1	<ul style="list-style-type: none">• <u>Se $L(G)$ è riconoscibile</u>, serve una Macchina di Turing• Macchina di Turing (<i>con nastro di lunghezza proporzionale alla frase da riconoscere</i>)
<ul style="list-style-type: none">• Tipo 2 (<i>context-free</i>)• Tipo 3 (<i>regolari</i>)	<ul style="list-style-type: none">• Push-Down Automaton (PDA) (<i>cioè ASF + stack</i>)• Automa a Stati Finiti (ASF)



GRAMMATICHE REGOLARI

CASO PARTICOLARE

Per grammatiche regolari, è sempre possibile e spesso conveniente trasformare la grammatica in *forma strettamente lineare*

- non più $\sigma \in VT^*$ (σ è una stringa di caratteri)

lineare a destra

$A \rightarrow \sigma$

$A \rightarrow \sigma B$

lineare a sinistra

$A \rightarrow \sigma$

$A \rightarrow B \sigma$

- ma bensì $a \in VT$ (a è un singolo carattere)

lineare a destra

$X \rightarrow a$

$X \rightarrow a Y$

lineare a sinistra

$X \rightarrow a$

$X \rightarrow Y a$



GRAMMATICHE LINEARI: ESEMPI

$VT = \{ a, +, - \}, \quad VN = \{ S \}$

- Grammatica G1 (*lineare* a sinistra: $A \rightarrow B y$, con $y \in VT^*$)

$S \rightarrow a \quad S \rightarrow S + a \quad S \rightarrow S - a$

- Grammatica G2 (*lineare* a destra: $A \rightarrow x B$, con $x \in VT^*$)

$S \rightarrow a \quad S \rightarrow a + S \quad S \rightarrow a - S$

- Grammatica G3 (G2 resa *strettamente lineare* a destra)

$S \rightarrow a \quad S \rightarrow a A \quad A \rightarrow + S \quad A \rightarrow - S$

diventa una regola
a singolo carattere

- Grammatica G4 (*lineare* a destra e anche a sinistra)

$S \rightarrow \text{ciao}$

- Grammatica G5 (G4 resa *strettamente lineare* a destra)

$S \rightarrow c T \quad T \rightarrow i U \quad U \rightarrow a V \quad V \rightarrow o$

RELAZIONE GERARCHICA

Le grammatiche sono in *relazione gerarchica*:

- una grammatica *regolare* (Tipo 3) è un *caso particolare* di grammatica *context-free* (Tipo 2),
- che a sua volta è un *caso particolare* di grammatica *context-dependent* (Tipo 1),
- che a sua volta è – ovviamente – un *caso particolare* di grammatica qualsiasi (Tipo 0).

NB: poiché le grammatiche di tipo 2 (e quindi di tipo 3) possono generare la stringa vuota ε , la relazione di inclusione vale solo se si conviene di ammettere nelle grammatiche tipo 1 anche la produzione $S \rightarrow \varepsilon$



CLASSIFICAZIONE DI CHOMSKY

IL PROBLEMA DELLA STRINGA VUOTA

Nella classificazione di Chomsky,

- Le grammatiche di Tipo 1 non ammettono la stringa vuota ε sul lato destro delle produzioni:

$$\beta A \delta \rightarrow \beta \alpha \delta \quad \alpha \neq \varepsilon$$

- Viceversa, le grammatiche di Tipo 2 la ammettono:

$$A \rightarrow \alpha \quad \alpha \in V^* \quad (\alpha \text{ può essere } \varepsilon)$$

- e lo stesso vale per le grammatiche di Tipo 3:

lin. a destra *lin. a sinistra*

$$A \rightarrow \sigma$$

$$A \rightarrow \sigma$$

$$A \rightarrow \sigma B$$

$$A \rightarrow B \sigma$$

$$\sigma \in VT^* \quad (\sigma \text{ può essere } \varepsilon)$$

MA COME? NON C'È CONTRADDIZIONE ??



CLASSIFICAZIONE DI CHOMSKY

IL PROBLEMA DELLA STRINGA VUOTA

COME È POSSIBILE che

- le grammatiche siano in relazione gerarchica tra loro
- ma al contempo **la stringa vuota non sia ammessa nel Tipo 1 e sia invece ammessa nei Tipi 2 e 3 ?**

Sembrerebbe esserci una evidente contraddizione.

L'assenza di contraddizione è dovuta al seguente

TEOREMA

le produzioni di grammatiche di Tipo 2 (e quindi anche 3)
possono sempre essere riscritte in modo da evitare la stringa vuota: al più, possono contenere la regola $S \rightarrow \varepsilon$



CLASSIFICAZIONE DI CHOMSKY

IL PROBLEMA DELLA STRINGA VUOTA

TEOREMA

quindi dal tipo 2 in \mathcal{P}

- Se G è una grammatica *context free* con produzioni della forma $A \rightarrow \alpha$, con $\alpha \in V^*$ (cioè, α può essere ε)
- allora *esiste una grammatica context free G' che genera lo stesso linguaggio $L(G)$ ma le cui produzioni hanno o la forma $A \rightarrow \alpha$, con $\alpha \in V^+$ (α non è ε) oppure la forma $S \rightarrow \varepsilon$, ed S non compare sulla destra in nessuna produzione.*

In pratica, il teorema assicura che la **sola differenza** fra una grammatica context free **con o senza ε -rules** è che **il linguaggio generato dalla prima include la stringa vuota.**

I linguaggi di programmazione (Pascal, C, ...) hanno spesso produzioni che ammettono la stringa vuota, di solito per descrivere parti *opzionali*.



ELIMINAZIONE DELLE ϵ -RULES

Come determinare la grammatica equivalente G' ?

Siano

- YES_ϵ l'insieme dei metasimboli $A_1..A_k$ da cui si può ricavare ϵ
- NO_ϵ l'insieme dei metasimboli $B_1..B_m$ da cui non si può ricavare ϵ

Allora:

- se G contiene la regola $S \rightarrow \epsilon$, anche G' contiene tale regola
- se G contiene *altre regole* della forma $X \rightarrow \epsilon$, G' non le contiene
- se G contiene una produzione della forma $X \rightarrow C_1 C_2 \dots C_r$ ($r \geq 1$), G' contiene la produzione $X \rightarrow \alpha_1 \alpha_2 \dots \alpha_r$ dove:

$$\alpha_i = C_i \quad \text{se } C_i \in VT \cup NO_\epsilon$$

$$\alpha_i = C_i \mid \epsilon \quad \text{se } C_i \in YES_\epsilon$$

con il *vincolo* che non tutti gli α_i possono essere ϵ .

ESEMPIO 1

Si consideri l'esempio a lato.

Qui $YES_\varepsilon = \{A\}$ e $NO_\varepsilon = \{S, B\}$.

- G *non contiene* la regola $S \rightarrow \varepsilon$ quindi *neppure G' la contiene*.
- G *contiene una regola* della forma $X \rightarrow \varepsilon$
È la regola $A \rightarrow \varepsilon$, *che va quindi tolta*.
- Al suo posto, poiché $A \in YES_\varepsilon$, *ogni occorrenza di A va sostituita da* $(A | \varepsilon)$
- Nel caso della regola $S \rightarrow A B | B$ ciò non ha effetti, poiché $(A | \varepsilon) B$ riproduce $A B | B$
- Invece, $A \rightarrow a A$ diventa $A \rightarrow a (A | \varepsilon)$
- Semplificando e riscrivendo si ottiene la nuova grammatica G'

Grammatica G (con ε -rules)

$S \rightarrow A B | B$

$A \rightarrow a A | \varepsilon$

$B \rightarrow b B | c$

Grammatica G'

$S \rightarrow A B | B$

$A \rightarrow a (A | \varepsilon)$

$B \rightarrow b B | c$

Grammatica G'

$S \rightarrow A B | B$

$A \rightarrow a A | a$

$B \rightarrow b B | c$

La nuova grammatica non genera mai la stringa vuota.

ESEMPIO 2

Grammatica G (con ε -rules)

$S \rightarrow A B$

$A \rightarrow a A \mid \varepsilon$

$B \rightarrow b B \mid \varepsilon$

Grammatica G'

$S \rightarrow (A \mid \varepsilon) (B \mid \varepsilon)$

$A \rightarrow a (A \mid \varepsilon)$

$B \rightarrow b (B \mid \varepsilon)$

Grammatica G'

$S \rightarrow A B \mid B \mid A \mid \varepsilon$

$A \rightarrow a A \mid a$

$B \rightarrow b B \mid b$

- La nuova grammatica può generare la stringa vuota **solo al primo passo della derivazione** (regola $S \rightarrow \varepsilon$), *ma non nei passi intermedi*
- Ergo, il linguaggio in sé comprende la stringa vuota, *ma le forme di frase non possono comunque accorciarsi.*



GRAMMATICHE e LINGUAGGI

- Poiché le grammatiche sono in relazione gerarchica, può accadere che un linguaggio possa essere **generato da più grammatiche, anche di tipo diverso**
 - un linguaggio di **Tipo 3** potrebbe in realtà essere generato anche da grammatiche di **Tipo 2**, **Tipo 1**, **Tipo 0**
 - un linguaggio di **Tipo 2** potrebbe in realtà essere generato anche da grammatiche di **Tipo 1**, **Tipo 0**
 - un linguaggio di **Tipo 1** potrebbe in realtà essere generato anche da grammatiche di **Tipo 0**

Non è detto infatti che la prima grammatica che si trova per generare un dato linguaggio sia necessariamente *la migliore (più semplice)*

CONSEGUENZA

- Il tipo del linguaggio *può non coincidere* col tipo della grammatica che lo genera
 - il linguaggio generato potrebbe essere di *un tipo più semplice* della sua grammatica
- D'ora in poi, dicendo che un linguaggio è di un certo tipo intenderemo che è *il tipo della grammatica più semplice* in grado di generarlo
 - per *linguaggi dipendenti da contesto* (o di Tipo 1) si intendono linguaggi che *richiedono come minimo una grammatica di Tipo 1* per essere generati
 - analogamente, per *linguaggi liberi da contesto* (o di Tipo 2) si intendono linguaggi che *richiedono come minimo una grammatica di Tipo 2..*
 - .. e lo stesso vale per i *linguaggi regolari* (o di Tipo 3)

ESEMPIO $a^n b^n c^n$ (1/3)

Il linguaggio $L = \{ a^n b^n c^n, n \geq 0 \}$ è (almeno) **di Tipo 1** in quanto esiste una grammatica di Tipo 1 che lo genera:

G1

$S \rightarrow aBC \mid aSBC$

$CB \rightarrow DB$

$DB \rightarrow DC$

$DC \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

La grammatica diventa *più compatta* se espressa con la *definizione alternativa* di grammatica di Tipo 1, che ammette lo scambio:

G2

$S \rightarrow aBC \mid aSBC$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

Il linguaggio sarebbe però generabile *anche da una grammatica di Tipo 0*, come ad esempio quella mostrata in precedenza:

G0

$S \rightarrow aSBC$

$CB \rightarrow BC$

$SB \rightarrow bF$

$FB \rightarrow bF$

$FC \rightarrow cG$

$GC \rightarrow cG$

$G \rightarrow \varepsilon$



ESEMPIO $a^n b^n c^n$ (2/3)

Una grammatica ancora più semplice potrebbe essere:

G3

$S \rightarrow abc \mid aBSc$

$Ba \rightarrow aB$

$Bb \rightarrow bb$

DUBBI & DOMANDE

- Ci si potrebbe quindi chiedere se non esista per questo linguaggio una grammatica ancora più semplice, *magari di Tipo2*
- Più in generale, ci si potrebbe chiedere se ci sia un modo generale per capire se una grammatica più semplice esista.. e magari trovarla.

Risponderemo presto a entrambe le domande 😊



ESEMPIO $a^n b^n c^n$ (3/3)

Derivazione della frase **aabbcc**

Grammatica G2:

S \rightarrow **aBC** | **aSBC**

CB \rightarrow **BC**

aB \rightarrow **ab**

bB \rightarrow **bb**

bC \rightarrow **bc**

cC \rightarrow **cc**

Derivazione:

S \rightarrow **a****S****BC** \rightarrow **aa****B****C****BC** \rightarrow **aa****B****B****C****C** \rightarrow
 \rightarrow **aab****B****C****C** \rightarrow **aabb****C****C** \rightarrow **aabb****c****C** \rightarrow **aabbcc**

Grammatica G3:

S \rightarrow **abc** | **aBSc**

Ba \rightarrow **aB**

Bb \rightarrow **bb**

Derivazione:

S \rightarrow **a****B****S****c** \rightarrow **a****B****a****b****cc** \rightarrow **aa****B****b****cc** \rightarrow **aabbcc**

RAMI DI DERIVAZIONE "MORTI"

- Nelle grammatiche di Tipo 1 *non è garantito che qualunque sequenza di derivazione porti a una frase*
 - Può succedere di trovarsi in una *strada chiusa*, impossibilitati a proseguire *perché non ci sono regole di produzione applicabili*
 - Questo non succede mai nel Tipo 2 e nel Tipo 3

Esempio

Grammatica G2:

$S \rightarrow aBC \mid aSBC$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

Derivazione su ramo morto:

$S \rightarrow a\underline{S}BC \rightarrow aa\underline{B}CBC \rightarrow aab\underline{C}CBC \rightarrow$
 $\rightarrow aabcBC \rightarrow ???????$



GRAMMATICHE DI TIPO 1 e DI TIPO 2

C'è dunque una *caratteristica cruciale* che discrimina una grammatica di **Tipo 1** da una di **Tipo 2** ?

Dice Chomsky:

Tipo 1: produzioni della forma $\sigma A \delta \rightarrow \alpha$

Tipo 2: produzioni della forma $A \rightarrow \alpha$

In particolare, il Tipo 1 ammette produzioni della forma

$BC \rightarrow CB$

che *scambiano due simboli*

- Questa caratteristica è impossibile da esprimere nel Tipo 2
- Per esprimerla occorre infatti poter scrivere *due elementi sul lato sinistro della produzione*, ma il Tipo 2 ammette in tale posizione *un unico metasimbolo!*



GRAMMATICHE DI TIPO 2 e DI TIPO 3

Analogamente, c'è una *caratteristica* che distingue una grammatica di **Tipo 2** da una di **Tipo 3** ?

Dice Chomsky:

Tipo 2: produzioni della forma $A \rightarrow \alpha$
dove α *può contenere più metasimboli, in qualsiasi posizione*
[$\alpha \in (VT \cup VN)^*$, $A \in VN$]

Tipo 3: produzioni lineari, della forma
 $A \rightarrow \sigma$ o $A \rightarrow \sigma B$ (*a destra*) oppure
 $A \rightarrow \sigma$ o $A \rightarrow B \sigma$ (*a sinistra*)
dove *ci può essere un solo metasimbolo, o in testa o in coda*
[$A, B \in VN$, $\sigma \in VT^*$]

GRAMMATICHE DI TIPO 2 e DI TIPO 3

Analogamente, c'è una *caratteristica* che distingue una grammatica di **Tipo 2** da una di **Tipo 3** ?

Dice Chomsky:

Tipo 2: produzioni della forma $A \rightarrow \alpha$
dove α *può contenere più metasimboli, in qualsiasi posizione*
[$\alpha \in (VT \cup VN)^*$]

Tipo 3: produzioni della forma $A \rightarrow \sigma$ o $A \rightarrow B\sigma$ (a sinistra)
dove *ci può essere un solo metasimbolo, o in testa o in coda*
[$A, B \in VN, \sigma \in VT^*$]

Nel Tipo 2, i meta-simboli possono trovarsi *in mezzo* alla forma di frase; nel Tipo 3, no.

SELF - EMBEDDING

Una grammatica contiene *self-embedding* quando una o più produzioni hanno la forma

$$A \xRightarrow{*} \alpha_1 A \alpha_2 \quad (\text{con } \alpha_1, \alpha_2 \in V^+)$$

TEOREMA: una grammatica di Tipo 2 *che non contenga self-embedding* genera un *linguaggio regolare*

Dunque, è *il self-embedding* la *caratteristica cruciale* di una grammatica di Tipo 2, che la differenzia da una di Tipo 3.

- Se non c'è self-embedding in nessuna produzione, esiste una grammatica equivalente di Tipo 3, quindi il linguaggio generato è regolare.
- Non vale necessariamente il viceversa: una grammatica **con** self-embedding *potrebbe comunque generare un linguaggio regolare*, se il self-embedding è "finto" (ovvero, "disattivato" da altre regole)

SELF-EMBEDDING: ESEMPIO

La grammatica G:

$$S \rightarrow aSc \quad S \rightarrow A \quad A \rightarrow bAc \quad A \rightarrow \varepsilon$$

presenta self-embedding e genera il linguaggio $L(G)$:

$$L(G) = \{ a^n b^m c^{n+m} \mid n, m \geq 0 \}$$

Il ruolo del self-embedding è introdurre una ricorsione in cui *si aggiungono contemporaneamente simboli a sini-stra e a destra*, garantendo di procedere "di pari passo".

È essenziale per definire linguaggi le cui frasi devono contenere simboli bilanciati, come ad esempio le parentesi:

$$S \rightarrow (S) \quad S \rightarrow a$$

Questa grammatica genera il linguaggio $L(G) = \{ ({}^n a)^n \mid n \geq 0 \}$



"FINTO" SELF – EMBEDDING (1/4)

Nonostante la presenza di self-embedding, il linguaggio generato *può essere regolare se la regola con self-embedding è disattivata da altre regole meno restrittive, che vanificano il vincolo che il self-embedding vorrebbe imporre*

- Identificare casi del genere non è banale

Riferimento: "*Self-embedded context-free grammars with regular counterparts*", by S.Andrei, W.Chin, S.Cavadini; Acta Informatica 40, 349-365, 2004, Springer

- Ci limiteremo a illustrarlo tramite alcuni esempi.



"FINTO" SELF – EMBEDDING (2/4)

ESEMPIO 1

$S \rightarrow a S a \mid X$

$X \rightarrow a X \mid b X \mid a \mid b$

- Sembrerebbe che le frasi dovessero avere la forma $a^n Y a^n$..
- .. ma la parte centrale X si espande in una *sequenza qualunque* di a e b , vanificando il vincolo che le a in testa e in coda siano in egual numero.
- Risultato: $L(G)$ è regolare, in quanto comprende qualunque sequenza di a e b

"FINTO" SELF – EMBEDDING (3/4)

ESEMPIO 2

S → **a b S b a** | **a b a**

- In questo esempio, il self-embedding viene disattivato in un modo *particolarmente subdolo e sottile*
- Apparentemente i due lati "sinistro" e "destro" crescono in parallelo, producendo un numero identico di gruppi $(a b)^k$ e $(b a)^k$...
- .. ma sul più bello, nel mezzo viene piazzato un **a b a** che *spariglia le carte e "distrugge i confini"* fra i due gruppi $(a b)^k$ e $(b a)^k$ rendendoli *indistinguibili*

S → **a b S b a** → **a b a b S b a b a** → **a b a b a b S b a b a b a** →
→ **a b a b a b S b a b a b a** → **a b a b a b a b a b a b a**

- Risultato: la frase è *una sequenza di una quantità dispari di gruppi a b*, seguiti da una **a** finale –un linguaggio regolare: $L(G) = \{ (a b)^{2n+1} a, n \geq 0 \}$
- Grammatica di Tipo 3 equivalente: $S \rightarrow X a$
 $X \rightarrow a b \mid X a b a b$



"FINTO" SELF – EMBEDDING (4/5)

ESEMPIO 3

$$S \rightarrow a S a \mid \varepsilon$$

$$L(G) = \{ (a a)^n, n \geq 0 \}$$

- Qui *il self-embedding, più che disattivato, è inutile*, perché con un alfabeto di un solo carattere si possono generare solo frasi *estremamente semplici*
- In effetti, *è impossibile distinguere un "gruppo di sinistra" da un "gruppo di destra" se sono fatti tutti solo da un unico possibile simbolo!*
- Grammatica di Tipo 3 equivalente : $S \rightarrow a a S \mid \varepsilon$

L'osservazione precedente è generalizzata dal seguente

TEOREMA: ogni linguaggio *context-free di alfabeto unitario* è in realtà un *linguaggio regolare*.



RICONOSCIBILITÀ DEI LINGUAGGI

- I linguaggi generati da grammatiche di Tipo 0 *possono in generale NON essere riconoscibili* (decidibili)
 - Non è garantita l'esistenza di una MdT capace di decidere se una frase appartiene o meno al linguaggio
- Al contrario, i linguaggi generati da grammatiche di Tipo 1 (e quindi di Tipo 2 e 3) *SONO riconoscibili*
 - Esiste sempre una MdT capace di decidere se una frase appartiene o meno al linguaggio
 - L'efficienza del processo di riconoscimento, però, è un'altra faccenda...



RICONOSCIBILITÀ DEI LINGUAGGI

- Per ottenere un *traduttore efficiente* occorre adottare linguaggi generati da (classi speciali di) grammatiche di Tipo 2
 - Tutti i linguaggi di programmazione sono infatti *context free*
 - Il riconoscitore prende il nome di **PARSER** parser e scanner sono ovviamente due componenti separati, non monolite
- Per ottenere particolare efficienza in sotto-parti di uso estremamente frequente, si adottano spesso per esse linguaggi generati da grammatiche di Tipo 3
 - identificatori & numeri
 - Il riconoscitore prende il nome di **SCANNER** (o *lexer*)



QUALI MACCHINE PER QUALI LINGUAGGI?

Chi riconosce i diversi tipi di linguaggi?

GRAMMATICHE	AUTOMI RICONOSCITORI
<ul style="list-style-type: none">• Tipo 0• Tipo 1	<ul style="list-style-type: none">• <u>Se $L(G)$ è riconoscibile</u>, serve una Macchina di Turing• Macchina di Turing (<i>con nastro di lunghezza proporzionale alla frase da riconoscere</i>)
<ul style="list-style-type: none">• Tipo 2 (<i>context-free</i>)• Tipo 3 (<i>regolari</i>)	<ul style="list-style-type: none">• Push-Down Automaton (PDA) (<i>cioè ASF + stack</i>)• Automa a Stati Finiti (ASF)

da qui ci si concentra dal tipo 2 in giù



NOTAZIONI PER GRAMMATICHE DI TIPO 2

- Alla luce del discorso precedente, **d'ora in poi ci concentreremo sulle *grammatiche di Tipo 2* (e 3)**
- Passando dalla teoria alla pratica, è opportuno modificare le notazioni fin qui utilizzate
 - non è pratico utilizzare lettere greche
 - non è il caso di continuare a riservare le lettere maiuscole ai meta-simboli, perché vogliamo poterle usare nelle frasi (e dunque nell'alfabeto *terminale*)
 - serve un nuovo modo per indicare i metasimboli
 - nelle tastiere e nei font "di base", non ci sono frecce e altri simboli particolari → sarebbe meglio farne senza



GRAMMATICHE B.N.F.

In una *Grammatica BNF*

$::=$ rappresenta una freccia

- le regole di produzione hanno la forma $\alpha ::= \beta$
con $\alpha \in V^+$, $\beta \in V^*$
- i meta-simboli $X \in VN$ hanno la forma $\langle \text{nome} \rangle$
- il meta-simbolo $|$ indica *l'alternativa*

Questa estensione permette di esprimere un *insieme di regole aventi la stessa parte sinistra*:

$$X ::= A_1 \quad \dots \quad X ::= A_N$$

in forma compatta:

$$X ::= A_1 \mid A_2 \mid \dots \mid A_N$$

ESEMPIO 2

$G = \langle VT, VN, P, S \rangle$, dove:

$VT = \{ \text{il, gatto, topo, sasso, mangia, beve} \}$

$VN = \{ \langle \text{frase} \rangle, \langle \text{soggetto} \rangle, \langle \text{verbo} \rangle, \text{meta simboli} \langle \text{compl-ogg} \rangle, \langle \text{articolo} \rangle, \langle \text{nome} \rangle \}$

$S = \langle \text{frase} \rangle$

$P = \{ \text{P definisce la struttura per i meta siml}$

$\langle \text{frase} \rangle ::= \langle \text{soggetto} \rangle \langle \text{verbo} \rangle \langle \text{compl-ogg} \rangle$

$\langle \text{soggetto} \rangle ::= \langle \text{articolo} \rangle \langle \text{nome} \rangle$

$\langle \text{articolo} \rangle ::= \text{il}$

$\langle \text{nome} \rangle ::= \text{gatto} \mid \text{topo} \mid \text{sasso}$

$\langle \text{verbo} \rangle ::= \text{mangia} \mid \text{beve}$

$\langle \text{compl-ogg} \rangle ::= \langle \text{articolo} \rangle \langle \text{nome} \rangle$

}



ESEMPIO 2: DERIVAZIONE

ESEMPIO: derivazione della frase

“il gatto mangia il topo”

(ammesso che tale frase *sia derivabile*)

<frase>

- **<soggetto> <verbo> <compl-ogg>**
- **<articolo> <nome> <verbo> <compl-ogg>**
- **il <nome> <verbo> <compl-ogg>**
- **il gatto <verbo> <compl-ogg>**
- **il gatto mangia <compl-ogg>**
- **il gatto mangia <articolo><nome>**
- **il gatto mangia il <nome>**
- **il gatto mangia il topo**

EXTENDED B.N.F. (EBNF)

La notazione **EBNF** è una *forma estesa* della notazione B.N.F., rispetto a cui introduce alcune *notazioni compatte* per *alleggerire la scrittura* delle regole di produzione

	Forma EBNF	BNF equivalente	significato
1	$X ::= [a] B$	$X ::= B \mid aB$	a può comparire 0 o 1 volta
2	$X ::= \{a\}^n B$	$X ::= B \mid aB \mid \dots \mid a^n B$	a può comparire da 0 a n volte
3	$X ::= \{a\} B$	$X ::= B \mid aX$	a può comparire 0 o più volte

a opzionale

la differenza tra 2 e 3 sta nel numero finito (n) o infinito di a (nel 3 si usa una regola ricorsiva per ottenere infinite a)

NOTA: la produzione $X ::= B \mid aX$ è ricorsiva (a destra).

Forma EBNF	BNF equivalente	significato
$X ::= (a \mid b) D \mid c$	$X ::= a D \mid b D \mid c$	raggruppa categorie sintattiche



ESEMPIO 3: NUMERI NATURALI

$G = \langle VT, VN, P, S \rangle$

dove:

$VT = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$

$VN = \{ \langle \text{num} \rangle, \langle \text{cifra} \rangle, \langle \text{cifra-non-nulla} \rangle \}$

$S = \langle \text{num} \rangle$

$P = \{$

$\langle \text{num} \rangle ::= \langle \text{cifra} \rangle \mid \langle \text{cifra-non-nulla} \rangle \{ \langle \text{cifra} \rangle \}$

$\langle \text{cifra} \rangle ::= 0 \mid \langle \text{cifra-non-nulla} \rangle$

$\langle \text{cifra-non-nulla} \rangle ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

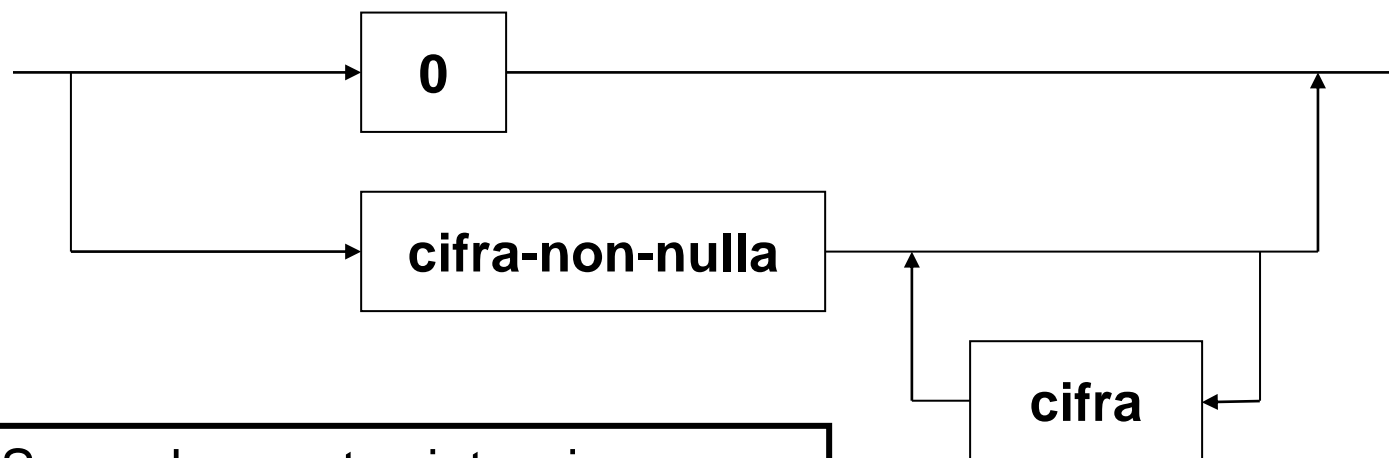
$\}$

non si può usare per scrivere "03" (nei linguaggi perché 0 se
per ottale e 0x per esadecimale)

EBNF

ESEMPIO 3: NUMERI NATURALI

La stessa sintassi può essere espressa tramite un *diagramma sintattico*



Secondo questa sintassi:

- "0" è una frase **lecita**
- "131" è una frase **lecita**
- "013" è una frase **illecita**



ESEMPIO 4: NUMERI INTERI

- Sintassi analoga alla precedente
- ma *con la possibilità di mettere un segno (+,-) davanti al numero naturale*

Quindi:

- stesse regole di produzione
più una (al top level) per generare il segno
- stesso alfabeto terminale
più i due simboli + e -



ESEMPIO 4: NUMERI INTERI

$G = \langle VT, VN, P, S \rangle$, dove:

$VT = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, - \}$

$VN = \{ \langle int \rangle, \langle num \rangle, \langle cifra \rangle, \langle cifra-non-nulla \rangle \}$

$P = \{$

$\langle int \rangle ::= [+|-] \langle num \rangle$

Lo scopo ora è $\langle int \rangle$,
non più $\langle num \rangle$

$\langle num \rangle ::= \langle cifra \rangle \mid \langle cifra-non-nulla \rangle \{ \langle cifra \rangle \}$

$\langle cifra \rangle ::= 0 \mid \langle cifra-non-nulla \rangle$

$\langle cifra-non-nulla \rangle ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\}$



ESEMPIO 5: IDENTIFICATORI

Nell'uso pratico **si danno di solito solo le regole di produzione**, definendo VT, VN e S *implicitamente*

- i non-terminali hanno la forma BNF $\langle \dots \rangle$
- il primo di essi è il simbolo iniziale

$P = \{$

$\langle \text{id} \rangle ::= \text{scopo} \{ \langle \text{lettera} \rangle \mid \langle \text{cifra} \rangle \}$

$\langle \text{lettera} \rangle ::= A \mid B \mid C \mid D \mid \dots \mid Z$

$\langle \text{cifra} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\}$

Diagram annotations:

- scopo** (green text) is annotated with a box labeled **VN** (orange text).
- A**, **B**, **C**, **D**, **...**, and **Z** are annotated with a box labeled **VT** (red text).
- 0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, and **9** are annotated with a box labeled **VT** (red text).



ALBERI DI DERIVAZIONE

Per le sole grammatiche di Tipo 2 si introduce il concetto di *albero di derivazione*

- ogni *nodo dell'albero* è associato a un *simbolo* del vocabolario $V = V_T \cup V_N$
- la *radice* dell'albero coincide con lo *scopo* S
- se a_1, a_2, \dots, a_k sono i k figli ordinati di un dato *nodo* X (associato al simbolo $X \in V_N$), significa che la grammatica contiene la produzione

$$X ::= A_1 A_2 \dots A_k$$

dove A_i è il simbolo associato al nodo a_i

Si noti che *l'albero di derivazione non può esistere per grammatiche di Tipo 1 e 0* perché in esse il lato sinistro delle produzioni ha *più di un simbolo* e dunque i nodi figli avrebbero *più di un padre* (ergo non si otterrebbe più un albero, ma un generico grafo).



RIPRENDENDO L' ESEMPIO 2

$G = \langle VT, VN, P, S \rangle$ con

$VT = \{ \text{il, gatto, topo, sasso, mangia, beve} \}$

$VN = \{ \langle \text{frase} \rangle, \langle \text{soggetto} \rangle, \langle \text{verbo} \rangle, \langle \text{compl-ogg} \rangle, \langle \text{articolo} \rangle, \langle \text{nome} \rangle \}$

$S = \langle \text{frase} \rangle$

$P = \{$
 $\langle \text{frase} \rangle ::= \langle \text{soggetto} \rangle \langle \text{verbo} \rangle \langle \text{compl-ogg} \rangle$
 $\langle \text{soggetto} \rangle ::= \langle \text{articolo} \rangle \langle \text{nome} \rangle$
 $\langle \text{articolo} \rangle ::= \text{il}$
 $\langle \text{nome} \rangle ::= \text{gatto} \mid \text{topo} \mid \text{sasso}$
 $\langle \text{verbo} \rangle ::= \text{mangia} \mid \text{beve}$
 $\langle \text{compl-ogg} \rangle ::= \langle \text{articolo} \rangle \langle \text{nome} \rangle$
}

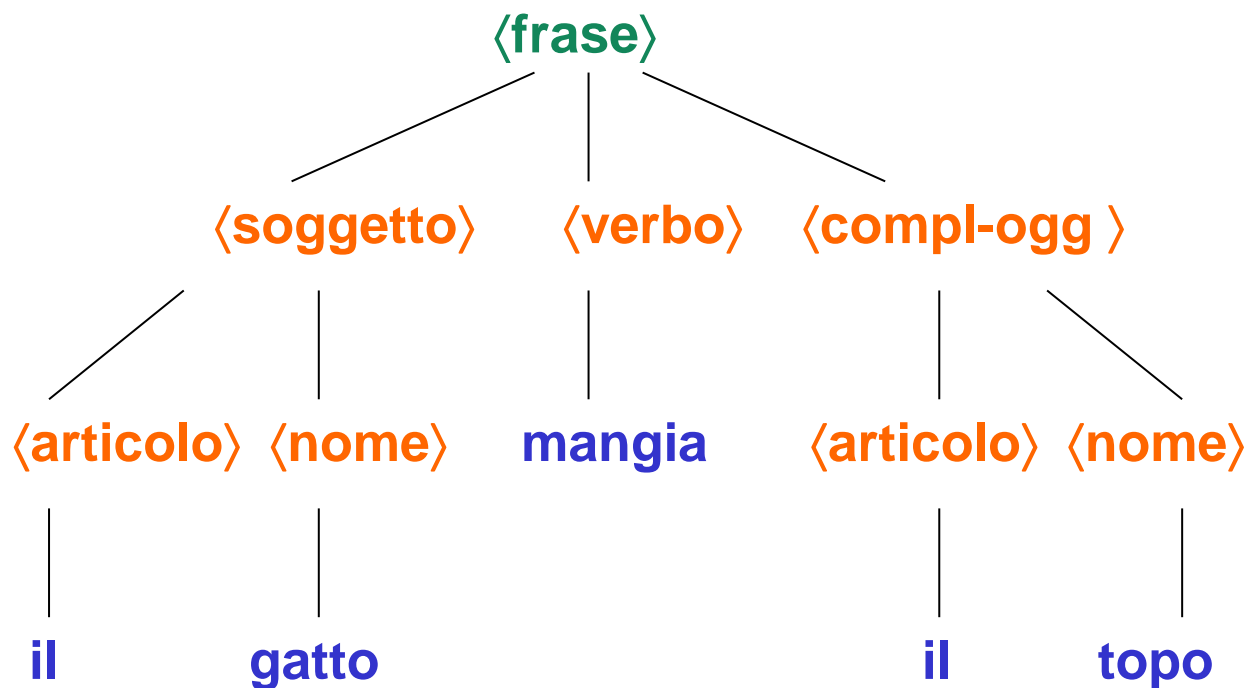


RIPRENDENDO L' ESEMPIO 2

Derivazione della frase

“il gatto mangia il topo”

(ammesso che tale frase sia derivabile)





RIPRENDENDO L' ESEMPIO 4

$G = \langle VT, VN, P, S \rangle$, dove:

$VT = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, - \}$

$VN = \{ \langle int \rangle, \langle num \rangle, \langle cifra \rangle, \langle cifra-non-nulla \rangle \}$

$P = \{$

$\langle int \rangle ::= [+|-] \langle num \rangle$

$\langle num \rangle ::= \langle cifra \rangle \mid \langle cifra-non-nulla \rangle \{ \langle cifra \rangle \}$

$\langle cifra \rangle ::= 0 \mid \langle cifra-non-nulla \rangle$

$\langle cifra-non-nulla \rangle ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\}$

EBNF



RIPRENDENDO L' ESEMPIO 4

- Qui **una regola è scritta in EBNF** (Extended BNF), che *non è direttamente mappabile* su un albero.
- Occorre perciò **riscriverla in BNF standard**, ricordando le equivalenze:

$$X ::= \{a\} B \quad \leftrightarrow \quad X ::= B \mid aX$$

$$X ::= B \{a\} \quad \leftrightarrow \quad X ::= B \mid Xa$$

- Dunque, la regola:

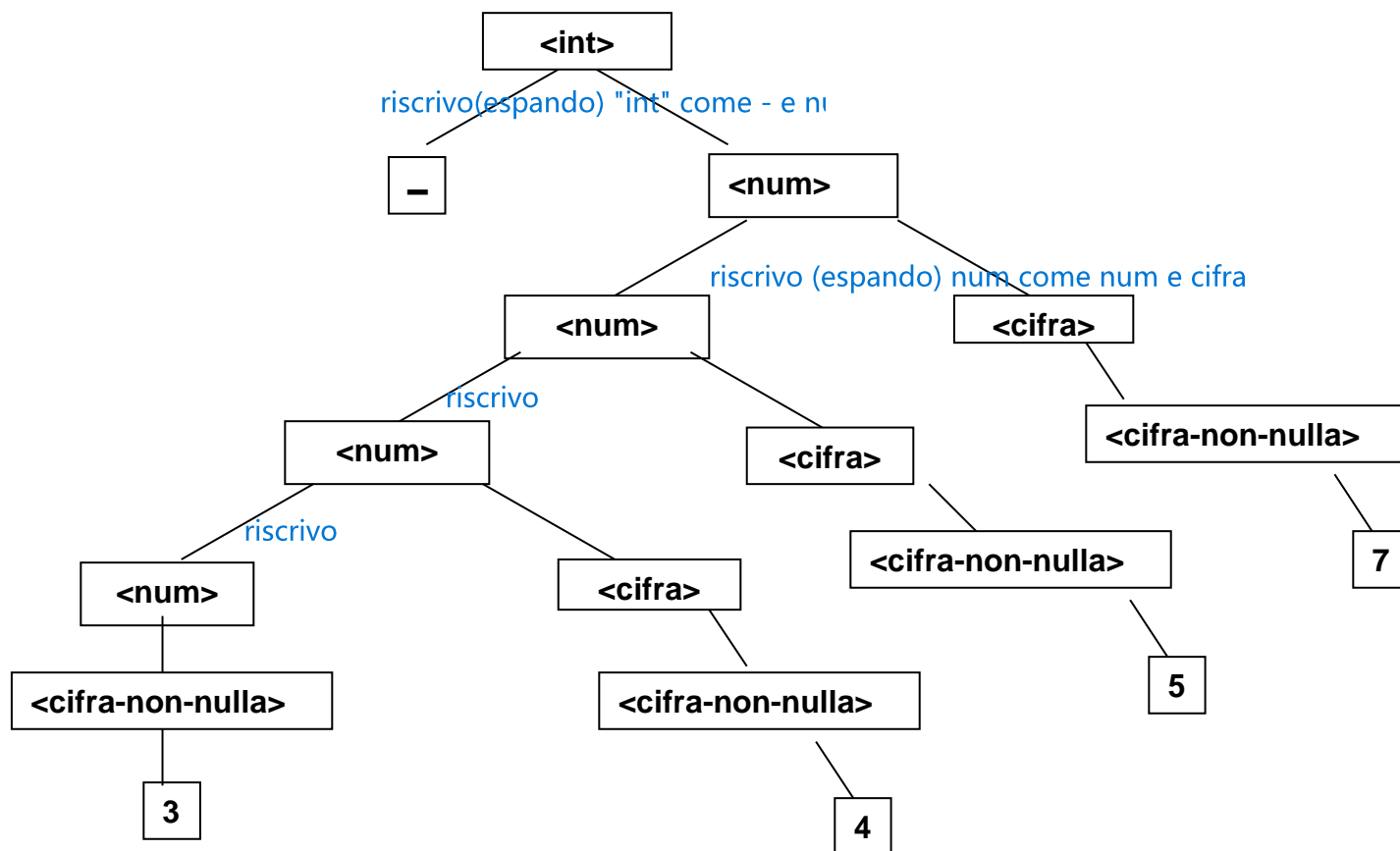
$$\langle \text{num} \rangle ::= \langle \text{cifra-non-nulla} \rangle \{ \langle \text{cifra} \rangle \}$$

va riscritta come:

$$\langle \text{num} \rangle ::= \langle \text{cifra-non-nulla} \rangle \mid \langle \text{num} \rangle \langle \text{cifra} \rangle$$

RIPRENDENDO L' ESEMPIO 4

Albero di derivazione del numero intero **-3457**:





DERIVAZIONI CANONICHE

DERIVAZIONE “LEFT-MOST” (*deriv. canonica sinistra*)

- A partire dallo scopo della grammatica, si riscrive sempre *il simbolo non-terminale più a sinistra*.

DERIVAZIONE “RIGHT-MOST” (*deriv. canonica destra*)

- A partire dallo scopo della grammatica, si riscrive sempre *il simbolo non-terminale più a destra*.

AMBIGUITÀ

- Una grammatica è *ambigua* se esiste almeno una frase che ammette *due o più derivazioni canoniche sinistre distinte* (i.e. per cui esistono almeno due alberi sintattici distinti).
 - Grado di ambiguità = numero di alberi sintattici distinti
- L'ambiguità è una caratteristica *indesiderabile*.

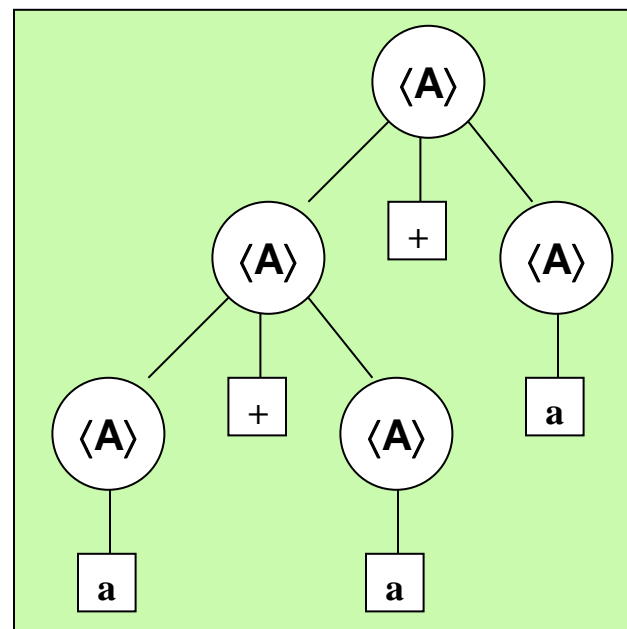
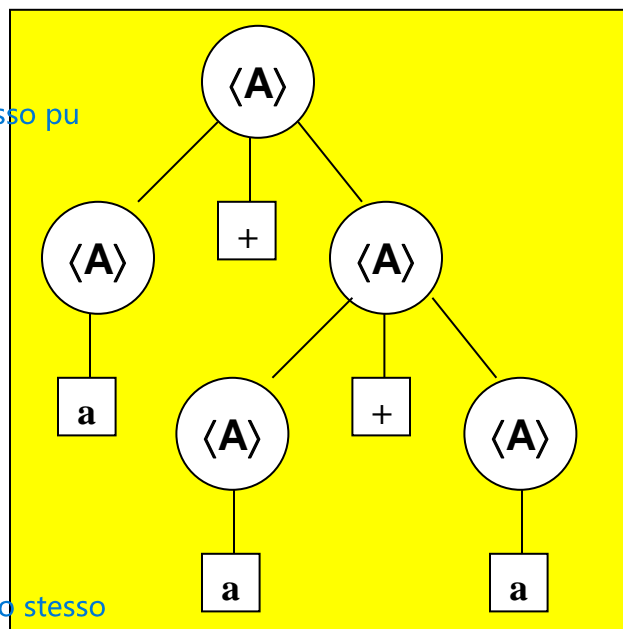
due percorsi diversi per arrivare allo stesso pu

ESEMPIO

$A ::= A + A$

$A ::= a$

La frase $a+a+a$ è *ambigua*:



ambigua per via di processi di derivazione diversi che conducono allo stesso risultato

AMBIGUITÀ

- Purtroppo, stabilire se una grammatica di Tipo 2 sia ambigua **è un problema indecidibile**
 - però, in pratica, un certo numero di derivazioni è spesso sufficiente per "convincersi" della (non per dimostrare la) ambiguità di G
- Se una grammatica è ambigua, spesso se ne può trovare un'altra che non lo sia – *ma non sempre*.
- Un **linguaggio si dice intrinsecamente ambiguo** se tutte le grammatiche che lo generano sono ambigue.
come detto a lezione, sono linguaggi progettati apposta per essere ambigui avendo cose che vanno in conflitto tra loro
 - ESEMPIO: $L = \{ a^n b^n c^* \} \cup \{ a^* b^n c^n \}$, con $n \geq 0$
Intuitivamente, tutte le frasi della forma $a^n b^n c^n$ appartengono a entrambi i sotto-linguaggi, quindi esistono due derivazioni distinte
 - ESEMPIO: $L = \{ a^n b^n c^m d^m \} \cup \{ a^n b^m c^m d^n \}$, con $m, n \geq 1$
Come sopra, almeno alcune frasi della forma $a^n b^n c^n d^n$ ammettono due derivazioni distinte

LA STRINGA VUOTA

- La stringa vuota può far parte delle frasi generate da una grammatica di Tipo 0, poiché la generica regola di produzione $\alpha \rightarrow \beta$ prevede $\alpha \in V^+$, $\beta \in V^*$
 - le forme di frase possono accorciarsi durante la riscrittura
- La stringa vuota invece non può far parte delle frasi generate da una grammatica di Tipo 1 (e quindi neanche di tipo 2 e 3) perché lì vige la condizione $\alpha \neq \varepsilon$ e perciò la generica forma di frase non può mai accorciarsi.

RICORDA: questo non è in contraddizione con il fatto che le produzioni di grammatiche di Tipo 2 e 3 possano "apparentemente" ammettere ε sul lato destro delle produzioni, perché esiste sempre una grammatica equivalente senza ε -rules (escluso al più S).

LA STRINGA VUOTA

- Talora però *farebbe comodo* avere la stringa vuota ε nel linguaggio, per esprimere *parti opzionali*
- È possibile farlo *senza alterare il tipo della grammatica* purché se ne ammetta la presenza *nella sola produzione di top-level* $S \rightarrow \varepsilon$ ed S non compaia altrove.
 - in questo modo, il solo caso in cui ε entra in gioco è se è scelta all'inizio, *al primo passo di derivazione*
 - tutte le altre stringhe sono generate da S *usando regole diverse*, che non contengono ε : ergo, le forme di frase non possono comunque accorciarsi.
- Questa proprietà è catturata dal seguente teorema:



LA STRINGA VUOTA

TEOREMA

- Dato un linguaggio L di tipo 0, 1, 2, o 3
- i linguaggi $L \cup \{\varepsilon\}$ e $L - \{\varepsilon\}$ *sono dello stesso tipo.*

Ad esempio, le produzioni:

$$S ::= \varepsilon \mid X$$
$$X ::= ab \mid a X b$$

definiscono il linguaggio (context-free) $L = \{ a^n b^n, n \geq 0 \}$

(Vale ovviamente la convenzione $a^0 = b^0 = \varepsilon$)

FORME NORMALI

Un linguaggio di tipo 2 *non vuoto* può essere sempre generato da una grammatica di tipo 2 in cui:

- ogni simbolo, terminale o non terminale, compare nella derivazione di qualche frase di L
 - ossia, non esistono simboli o meta-simboli inutili
- non ci sono produzioni della forma $A \rightarrow B$ con $A, B \in VN$
 - ossia non esistono produzioni che “cambiano solo nome” a un meta-simbolo
- se il linguaggio non comprende la stringa vuota ($\varepsilon \notin L$) allora *non ci sono* produzioni della forma $A \rightarrow \varepsilon$.

FORME NORMALI

In particolare si può fare in modo che *tutte le produzioni abbiano una forma ben precisa*:

- *forma normale di Chomsky*

produzioni della forma $A \rightarrow B C \mid a$

con $A, B, C \in VN$, $a \in VT \cup \varepsilon$

- *forma normale di Greibach* (per linguaggi privi di ε)

produzioni della forma $A \rightarrow a \alpha$

con $A \in VN$, $a \in VT$, $\alpha \in VN^*$

La forma normale di Greibach facilita, come si vedrà, la costruzione di riconoscitori.

ESEMPIO 1

Esiste un algoritmo che trasforma ogni grammatica di tipo 2 in forma normale di Chomsky.

Qui lo vediamo solo applicato a un esempio.

- Grammatica data:

$$S \rightarrow d A \mid c B$$

$$A \rightarrow d A A \mid c S \mid c$$

$$B \rightarrow c B B \mid d S \mid d$$

- Forma normale di Chomsky*

$$S \rightarrow M A \mid N B$$

$$M \rightarrow d \qquad N \rightarrow c$$

$$A \rightarrow M P \mid N S \mid c$$

$$P \rightarrow A A$$

$$B \rightarrow N Q \mid M S \mid d$$

$$Q \rightarrow B B$$

La trasformazione in forma di Greibach richiede alcune tecniche extra.



TRASFORMAZIONI IMPORTANTI

- Per facilitare la costruzione dei riconoscitori, è spesso rilevante poter **trasformare la struttura delle regole di produzione** per renderle *più adatte* allo scopo.
- Alcune trasformazioni particolarmente importanti sono
 - *la sostituzione*
 - *il raccoglimento a fattor comune*
 - *la eliminazione della ricorsione sinistra.*

Tra gli altri usi, queste trasformazioni sono la base per trasformare una qualsiasi grammatica di tipo 2 in forma normale di Greibach.

TRASFORMAZIONI IMPORTANTI

- Per facilitare la costruzione dei riconoscitori, è spesso rilevante poter trasformare la struttura delle regole di produzione per renderle *più adatte* allo scopo.
 - Alcune trasformazioni particolarmente importanti sono
 - la *sostituzione*
 - il *raccoglimento a fattor comune*
 - la *eliminazione della ricorsione sinistra*.
- Banali, analoghe all'algebra
- Importante, ma con conseguenze

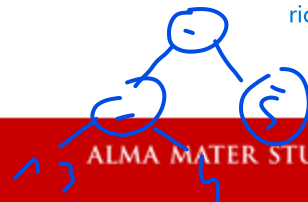
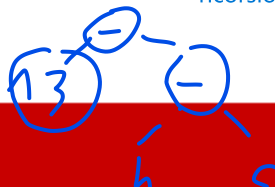
Tra gli altri usi, queste trasformazioni sono la base per trasformare una qualsiasi grammatica di tipo 2 in forma normale di Greibach.



IL PROBLEMA DELLA RICORSIONE SINISTRA

- La ricorsione sinistra **bad**
$$X \rightarrow X a c \mid p$$
- *nasconde l'iniziale delle frasi prodotte*, che si può determinare **solo guardando altre regole**
 - nell'esempio sopra, tutte le frasi iniziano per **p**, ma questo *non si vede* dalla regola ricorsiva $X \rightarrow X a c$
 - non è così nella ricorsione destra, che invece evidenzia proprio l'iniziale delle produzioni: $X \rightarrow r X \mid a$
- La **buona notizia** è che, tecnicamente, si può *sempre* sostituire la ricorsione sinistra con una destra.
- La **cattiva notizia** è che spesso *non ci potremo permettere il lusso di farlo*, a causa delle conseguenze.

esempio 13-4-5



equivalente, ma la ricorsione a destra pone il problema della semantica: le operazioni sono associative a sinistra, quindi prima faccio 13-4, poi 9-5 ecc. se parto da destra invece (ricorsione a DESTRA) significa espandere prima 4 e 5, quindi fare prima 4-5 (risultato sbagliato)

SOSTITUZIONE

La sostituzione consiste nell' *espandere un simbolo non terminale* che compare nella parte destra di una regola di produzione, sfruttando a tale scopo un'altra regola di produzione.

Nella grammatica a lato è possibile sostituire il metasimbolo **S** nella seconda produzione, usando a tale scopo la prima produzione.

ESEMPIO

$$\begin{aligned} S &\rightarrow x a \\ x &\rightarrow b Q \mid S c \mid d \end{aligned}$$

Espandiamo quindi **S** come indicato: la nuova regola per **x** non contiene più alcun riferimento a **S**

ESEMPIO

$$\begin{aligned} S &\rightarrow x a \\ x &\rightarrow b Q \mid x a c \mid d \end{aligned}$$



IL RACCOGLIMENTO A FATTOR COMUNE

Il raccoglimento a fattor comune consiste nell' **isolare il prefisso più lungo comune a due produzioni.**

ipotesi A: appena ottieni un carattere, il "compilatore" deve sapere quale regola applicare, senza aspettare. questo significa che nel primo esempio bisogna per forza tirare a caso

Nella grammatica a lato è possibile isolare il prefisso **a S** comune alle prime due produzioni.

ESEMPIO

$S \rightarrow a S b \mid a S c$

Raccogliamo quindi a fattore comune il prefisso comune **a S** ...

ESEMPIO

$S \rightarrow a S (b \mid c)$

...e introduciamo un **nuovo meta-simbolo x** per esprimere *la parte che segue* il prefisso comune.

ESEMPIO

$S \rightarrow a S x$
 $x \rightarrow b \mid c$

la x serve per efficientare il programma: con X succede che quando si arriva a X si e' certi: nel primo caso invece sotto ipotesi A si sarebbe dovuto "indovinare" la regola giusta, causando una cascata di "tiri di monetine" che causano irrimediabilmente errori



ELIMINAZIONE DELLA RICORSIONE SINISTRA

E' una trasformazione *sempre possibile*, articolata in due passi:

- Fase 1: eliminazione dei cicli ricorsivi a sinistra
- Fase 2: eliminazione della ricorsione sinistra diretta.

Fase preliminare

- si stabilisce una *relazione d'ordine* fra i meta-simboli coinvolti del ciclo ricorsivo
- Nel nostro caso, sia dunque $C > B > A$

ESEMPIO

$$\begin{aligned} A &\rightarrow B \ a \\ B &\rightarrow C \ b \\ C &\rightarrow A \ c \mid p \end{aligned}$$

Fase 1

- si modificano tutte le produzioni del tipo $Y \rightarrow X\alpha$ in cui $Y > X$, sostituendo a X le forme di frase stabilite dalle produzioni relative a X

Si ottiene quindi:

$$\begin{aligned} A &\rightarrow B \ a \\ B &\rightarrow C \ b \\ C &\rightarrow C \ b \ a \ c \mid p \end{aligned}$$

Fase 2

- le produzioni ricorsive dirette $X \rightarrow X \alpha \mid p$ si modificano introducendo un metasimbolo z e scrivendo $X \rightarrow p \mid p z$ e $z \rightarrow \alpha \mid \alpha z$

Ergo, $C \rightarrow C \ b \ a \ c \mid p$ diventa

$$\begin{aligned} C &\rightarrow p \mid p z \\ z &\rightarrow b \ a \ c \mid b \ a \ c \ z \end{aligned}$$



ELIMINAZIONE DELLA RICORSIONE SINISTRA

- Perché, dunque, potremmo *non poterlo (volarlo) fare*?
 - Sostituendo la ricorsione sinistra con una destra, si generano le stesse frasi, ma con regole (dichiaratamente!) diverse
 - Ergo, se interessa solo il risultato finale «ai morsetti», rimpiazzare la ricorsione sinistra con una destra è lecito e privo di conseguenze
 - se, invece, interessa anche *come ci si è arrivati* (ossia, la sequenza di derivazione), allora il rimpiazzo non è lecito perché cambiando le regole cambia anche la sequenza di derivazione.
- In un **puro riconoscitore**, che **deve solo dire «sì o no»**, eliminare la ricorsione sinistra è *fattibile* senza conseguenze
- In un **vero parser**, che **deve anche dare significato alle frasi** (lecite), *regole diverse* tipicamente implicano *significato diverso* per alcune frasi, con ciò *alterando il linguaggio*

ESEMPIO

- Nelle espressioni aritmetiche, la cultura matematica diffusa nei secoli richiede associatività sinistra:
 - in matematica, $13-5-4$ ha il significato di $(13-5)-4$ cioè *quattro*
 - non sarebbe la stessa cosa se fosse $13-(5-4)$ cioè *dodici*
- Sfortunatamente, ciò richiede regole grammaticali con ricorsione a sua volta sinistra:

$$E \rightarrow E + t \mid E - t \mid t$$

- Sostituirle con la ricorsione destra è possibile, ma porta a un albero di derivazione con associatività destra, che dà alle espressioni in significato *totalmente diverso!*
 - tecnicamente fattibile
 - **culturalmente improponibile**

ESEMPIO 2

Queste trasformazioni consentono di trasformare una grammatica in forma normale di Greibach.

Qui lo vediamo solo applicato a un esempio.

- Grammatica data:

$$S \rightarrow X a$$

$$X \rightarrow b S \mid S c \mid d$$

- **Forma normale di Greibach** ($A \rightarrow p \alpha$, $A \in VN$, $p \in VT$, $\alpha \in VN^*$)
 - eliminazione ciclo ricorsivo a sinistra
 - eliminazione ricorsione sinistra diretta
 - sostituzione
 - ridenominazione dei terminali tramite non-terminali ausiliari

ESEMPIO 2

Fase 1

- relazione d'ordine fra i simboli non terminali coinvolti del ciclo ricorsivo : $x > s$

Grammatica data:

$$S \rightarrow x a$$

$$x \rightarrow b s \mid s c \mid d$$

Fase 2

- modifica della produzione $x \rightarrow s c$ sostituendo a s la produzione $s \rightarrow x a$

Si ottiene quindi:

$$S \rightarrow x a$$

$$x \rightarrow (b s \mid d) \mid x a c$$

Fase 3

- eliminazione ricorsione sinistra $x \rightarrow x \alpha \mid p$, qui con $p = (b s \mid d)$, introducendo il nuovo simbolo z tale che $z \rightarrow \alpha \mid \alpha z$ e $x ::= p z \mid p$

da cui:

$$S \rightarrow x a$$

$$z \rightarrow a c \mid a c z$$

$$x \rightarrow (b s \mid d) z \mid (b s \mid d)$$

Fase 4

- sostituzione del simbolo x nella prima regola

$$S \rightarrow b s a \mid b s z a \mid d z a \mid d a$$

$$z \rightarrow a c \mid a c z$$

Fase 5

- introduzione dei non-terminali ausiliari A e C per rappresentare a e c dove appropriato

$$S \rightarrow b s A \mid b s z A \mid d z A \mid d A$$

$$z \rightarrow a C \mid a C z$$

$$A \rightarrow a$$

$$C \rightarrow c$$



COME CAPIRE SE UN LINGUAGGIO (NON) È DI TIPO 2 (3) ?

- Capire se un linguaggio è di Tipo 2 (o di Tipo 3)
"solo guardandolo" in generale non è banale
 - interessante non basta "immaginare" come possano essere le produzioni, perché nessuno assicura che le immaginiamo "bene"
- Il *PUMPING LEMMA* dà una condizione necessaria, ma non sufficiente, perché un linguaggio sia di Tipo 2 (o 3)
 - può quindi essere usato per dimostrare "in negativo" che un linguaggio non è di Tipo 2 (o di Tipo 3)...
 - .. ma purtroppo non per affermarlo "in positivo"



IL PUMPING LEMMA (o "lemma del pompaggio")

L'IDEA DI FONDO

- in un linguaggio infinito, **ogni stringa sufficientemente lunga deve avere una parte che si ripete**
- ergo, essa ***può essere "pompatà" un qualunque numero di volte ottenendo sempre altre stringhe del linguaggio***
 - È con questo lemma che si dimostra, ad esempio, che:
 $L1 = \{a^n b^n c^n\}$ non è di Tipo 2 (quindi è almeno di Tipo 1)
 $L2 = \{a^p, p \text{ primo}\}$ non è di Tipo 3 (quindi è almeno di Tipo 2)^(*)

La formulazione è leggermente diversa a seconda che si tratti di linguaggi di Tipo 2 o 3, ma la sostanza non cambia.

(*) in realtà non è neppure di Tipo 2, come si dimostra ri-applicando il lemma.



IL PUMPING LEMMA

per linguaggi context-free

Se L è un linguaggio di Tipo 2, **esiste un intero N tale che, per ogni stringa z di lunghezza almeno pari a N :**

- z è decomponibile in 5 parti: $z = uvwxy$ $|z| \geq N$
- la parte centrale vw ha lunghezza limitata: $|vw| \leq N$
- v e x non sono entrambi nulle: $|vx| \geq 1$
- la 2^a e la 4^a parte possono essere "*pompate*" quanto si vuole ottenendo sempre altre frasi del linguaggio; ovvero,
 $uv^iwx^iy \in L \quad \forall i \geq 0$

- Il numero N (lunghezza minima delle stringhe decomponibili in 5 parti di cui 2 pompabili) dipende dallo specifico linguaggio
- La dimostrazione si basa sulle lunghezze dei cammini nell'albero di derivazione associato (cfr. Hopcroft/Motwani/Ullman, p. 292)



IL PUMPING LEMMA per linguaggi regolari

Se L è un linguaggio di Tipo 3, **esiste un intero N tale che, per ogni stringa z di lunghezza almeno pari a N :**

- z può essere riscritta come: $z = xyw$ $|z| \geq N$
- la parte centrale xy ha lunghezza limitata: $|xy| \leq N$
- y non è nulla: $|y| \geq 1$
- **la parte centrale può essere *pompata* quanto si vuole**
ottenendo sempre altre frasi del linguaggio;
ovvero, $xy^iw \in L \quad \forall i \geq 0$

- Il numero N dipende caso per caso dallo specifico linguaggio
- La dimostrazione si basa sull'automa a stati associato
(cfr. Hopcroft/Motwani/Ullman, p. 135)

ESEMPIO 1

$L = \{a^p, p \text{ primo}\}$ non è un linguaggio regolare.

- se L fosse regolare, esisterebbe un intero N in grado di soddisfare il pumping lemma; sia allora **P un primo $\geq N+2$** (che esiste perché i numeri primi sono infiniti): consideriamo allora la stringa **$z = a^P$**
- scomponiamo ora **z** nei tre pezzi **xyw** , con **$|y| = r$** ; r quindi rappresenta la lunghezza del pezzo di y
ne segue che **$|xw| = p - r$**
- in base al lemma, se L fosse regolare, la nuova stringa **$xy^{p-r}w$** dovrebbe anch'essa appartenere al linguaggio, ma...
- la lunghezza di tale stringa sarebbe:
 $|xy^{p-r}w| = |xw| + (p-r)|y| = (p-r) + (p-r)|y| = (p-r)(1+|y|) = (p-r)(1+r)$
ovvero non un numero primo
- pertanto, *essa non appartiene a L* e dunque esso non è regolare.

ESEMPIO 2 (1)

$L = \{a^n b^n c^n\}$ non è context-free

- se L fosse context-free, esisterebbe un intero N in grado di soddisfare il pumping lemma; consideriamo allora la stringa $z = a^N b^N c^N$
- scomponiamo z nei cinque pezzi $uvwxy$, con $|vwx| \leq N$
- poiché fra l'ultima "a" e la prima "c" ci sono N posizioni, il pezzo centrale " vwx " non può contenere sia "a" sia "c", perché se contiene le une, non è abbastanza lungo da contenere le altre. Quindi, delle due:
 1. o " vwx " non contiene "c": allora " vx " è fatta solo di "a" e "b". Ma allora " uwy ", che in base al pumping lemma dovrebbe appartenere a L , ha tutte le "c" (che sono N) ma meno "a" o meno "b" del necessario, ergo non appartiene a $L \rightarrow$ assurdo
 2. o " vwx " non contiene "a": allora " vx " è fatta solo di "b" e "c", dunque " uwy " ha N "a" ma meno "b" o meno "c" del necessario, ergo non appartiene a $L \rightarrow$ assurdo.

ESEMPIO 2 (2)

ESEMPIO: $N=6 \rightarrow z = \text{"aaaaabbbbbbbcccccc"}$

Scomponiamo z nei cinque pezzi $uvwxy$, con $|\text{vwx}| \leq N$

- si può fare in vari modi, dipende da *come* e *dove* si prende **vwx**
- supponiamo di prenderla lunga 5 (comunque, al più 6): la suddivisione può quindi essere una delle seguenti illustrate in tabella:

u	vwx	y
...
aaaaa	abbbb	bbcccccc
aaaaaa	bbbbb	bcccccc
aaaaaab	bbbbb	cccccc
aaaaaabb	bbbbs	cccccc
...

- come si vede, **vwx** non può contenere sia "a" sia "c": se contiene le une, per evidenti motivi di lunghezza non può contenere le altre.

ESEMPIO 2 (3)

– supponiamo, per fissare le idee, che la scelta sia questa:

u	vwx	y
aaaaa	abbbb	bbcccccc

– alla luce di ciò, la sotto-stringa **vx**, lunga almeno 1 ma priva del pezzo centrale **w**, a sua volta è fatta solo di "a" e "b"

– qual è il pezzo centrale **w** in " **abbbb** "? di nuovo, ci sono più possibilità:

v	w	x
a	bbbb	(vuota)
(vuota)	abbb	b
ab	bbb	(vuota)
(vuota)	abb	bb
...

– ora, fra le altre stringhe del linguaggio, della forma **uvⁱwxⁱy**, c'è anche quella per cui $i=0$, ossia in cui **v** e **x** mancano: è la stringa **uw⁰y**, ossia..

ESEMPIO 2 (4)

- ..ossia, dato che **u** e **y** sono quelle scelte da noi poco fa:

u	vwx	y
aaaaa	abbbb	bbcccccc

- e che il pezzo centrale **w** è uno di questi:

v	w	x
a	bbbb	(vuota)
(vuota)	abbb	b
ab	bbb	(vuota)
(vuota)	abb	bb

- la stringa **uwy** risulta "**aaaaa**" + **w** + "**bbcccccc**", ovvero ha tutte le sei "c" previste in fondo, ma meno "a" e/o meno "b" del necessario, perché alcune sono state "mangiate" dalla sotto-stringa vx
- ergo, la stringa **uwy** non appartiene al linguaggio, violando l'ipotesi.

ESEMPIO 2 (6)

$L = \{a^n b^n c^n\}$ non è context-free

- in alternativa si può dare una dimostrazione analoga all'Esempio 1
- consideriamo allora la stringa $z = a^N b^N c^N$
- scomponiamo z nei cinque pezzi $uvwxy$, con $|vwx| \leq N$: sia $vwx = b^N$
- in particolare, sia $|v| = p$, $|x| = q \rightarrow |w| = N - p - q$
- in base al lemma, se L fosse context free, la nuova stringa uv^2wx^2y dovrebbe anch'essa appartenere al linguaggio..
- ... ma tale stringa sarebbe: $a^N b^{2p} b^{N-p-q} b^{2q} c^N = a^N b^{2p+N-p-q+2q} c^N$
ovvero $a^N b^{N+p+q} c^N$ che ***non avrebbe la forma richiesta***
- pertanto, *essa non appartiene a L* e dunque esso non è context-free.

ESPRESSIONI REGOLARI

il prof fa un mega discorso sul fatto di usare il pumping lemma per escludere, in caso, il tipo 3. a quel punto sempre in qualche (forse sempre col PL) afferma che non sono neanche di tipo 0 o 1, quindi può essere, solo 2. nonostante a quanto pare non è condizione sufficiente ma solo necessaria sto teorema. [non sicuro]



ESPRESSIONI REGOLARI

Un formalismo di particolare interesse [per descrivere linguaggi] è quello delle *espressioni regolari*.

Le espressioni regolari sono *tutte e sole le espressioni* ottenibili tramite le seguenti regole:

- *la stringa vuota ε* è una espressione regolare
- dato un alfabeto A ,
ogni elemento $a \in A$ è una espressione regolare
- se X e Y sono espressioni regolari, lo sono anche:

$X+Y$ (*unione*)

$X \bullet Y$ (*concatenazione*)

X^* (*chiusura*)

definite come segue:



ESPRESSIONI REGOLARI

[definizione delle tre operazioni]

Unione (+)

(operatore meno prioritario)

$$X + Y = \{ x \mid x \in X \vee x \in Y \}$$

Concatenazione (•)

(associativa ma non commutativa)

$$X \bullet Y = \{ x \mid x = a b, a \in X \wedge b \in Y \}$$

$$\{ \} \bullet X = \{ \} \text{ per qualsiasi } X$$

Chiusura(*)

(operatore più prioritario)

$$X^* = X^0 \cup X^1 \cup X^2 \cup \dots$$

$$\text{dove } X^0 = \varepsilon$$

$$\text{e } X^k = X^{k-1} \bullet X$$



UN PRIMO ESEMPIO

ESEMPIO

$$X1 = \{00, 11\}$$

$$X2 = \{01, 10\}$$

$$X1 + X2 = \{00, 11, 01, 10\}$$

$$X1 \bullet X2 = \{0001, 1101, 0010, 1110\}$$

$$X2 \bullet X1 = \{0100, 0111, 1000, 1011\}$$

$$X1^* = \{\varepsilon, 00, 11, 0000, 0011, 1100, 1111, 000000, 000011, 001100, 001111, 110000, 110011, 111100, 111111, \dots\}$$

ATTENZIONE: uno stesso linguaggio può essere descritto da *molte espressioni regolari diverse* !

ALTRI ESEMPI

Con riferimento a linguaggi:

- ε denota il *linguaggio vuoto*
- un elemento $a \in A$ denota *il linguaggio $\{a\}$*
- $R1+R2$ denota *l'unione dei linguaggi* denotati da $R1$ e $R2$
- $R1 \bullet R2$ denota *la concatenazione dei linguaggi* denotati da $R1$ e $R2$
- R^* denota il risultato dell'operatore di chiusura applicato al linguaggio denotato da R .

ESEMPIO sull'alfabeto $A = \{ 0, 1 \}$

$$0 + 1^* = \{ 0, \varepsilon, 1, 11, 111, 1111, 11111, \dots \}$$

$$\begin{aligned} (0 + 1)^* &= \{ 0+1, \varepsilon, (0+1)(0+1), (0+1)(0+1)(0+1), \dots \} = \\ &= \{ \varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \dots \} \\ &= A^* \end{aligned}$$

$$(10 \bullet 01)^* = (1001)^* = \{ \varepsilon, 1001, 10011001, 100110011001, \dots \}$$



ESPRESSIONI vs LINGUAGGI REGOLARI

- Ma perché ci interessa tutto questo?
- Cosa hanno a che fare queste *curiose espressioni* con le grammatiche e i linguaggi ?

La risposta è nel seguente

TEOREMA

i linguaggi generati da *grammatiche regolari*
coincidono
con i linguaggi descritti da *espressioni regolari*.

Grammatiche ed espressioni regolari sono quindi *due rappresenta-zioni diverse della stessa realtà:*

- una è **costruttiva** – dice **COME** si fa, ma **non COSA** si ottiene
- l'altra **descrittiva** – dice **COSA** si ottiene, ma **non COME** si ottiene

RAPPRESENTAZIONI DIVERSE DELLA STESSA REALTÀ

GRAMMATICA

- rappresentazione **costruttiva**
- dice **COME** si fa
- **non COSA** si ottiene

sintetizza anche le regole

Grammatica

$S \rightarrow a S \mid b$

qui definisce il percorso in modo singolare: spiega punto per punto

ESP. REGOLARE

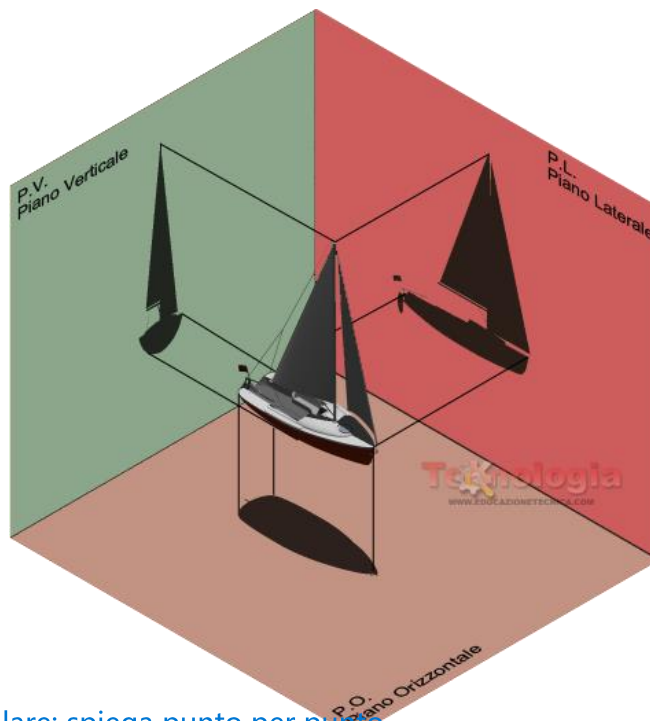
- rappresentazione **descrittiva**
- dice **COSA** si ottiene
- **non COME** si fa

sintetizza la logica

Espressione regolare

$L = \{ a^* b \}$

qui spiega la "soluzione" senza dirti come, le regole qui non sono chiare



Si può passare dall'una dall'altra?



PASSAGGI FRA RAPPRESENTAZIONI

Dalla grammatica all'espressione regolare

- si risolvono le cosiddette *equazioni sintattiche*

Dall'espressione regolare alla grammatica

- si interpretano gli operatori dell'espressione regolare in base alla loro semantica (sequenza, ripetizione, alternativa) mappandoli in opportune regole



DALLA GRAMMATICA ALL'ESPRESSIONE REGOLARE

Per passare **dalla grammatica all'espressione regolare** si interpretano le produzioni come **equazioni sintattiche**, in cui

- i simboli terminali sono i termini noti,
- i linguaggi generati da ogni simbolo non terminale sono le incognite e **si risolvono con le normali regole algebriche**.

ESEMPIO: la grammatica lineare a destra vista in precedenza:

$$S \rightarrow a \mid a + S \mid a - S$$

può essere letta come un'equazione con

- tre termini noti: $a, +, -$
- una incognita, L_S

che impone il vincolo (*usiamo per l'unione il simbolo \cup anziché $+$*)

$$L_S = a \cup (a + L_S) \cup (a - L_S) = (a + \cup a -) L_S \cup a$$

la cui soluzione, come vedremo ora, è l'espressione regolare

$$S = (a + \cup a -)^* a$$



SOLUZIONE DI EQUAZIONI SINTATTICHE

- Le equazioni sintattiche si risolvono tramite un *algoritmo*, che esiste in due versioni:
 - per grammatiche regolari a destra
 - per grammatiche regolari a sinistra
- Le due versioni differiscono però solo per un raccoglimento a fattor comune, in cui l'elemento raccolto:
 - nelle grammatiche regolari a destra, è raccolto a destra
 - nelle grammatiche regolari a sinistra, è raccolto a sinistrae nella conseguente posizione dei termini nell'espressione risultante.



ALGORITMO

(grammatiche regolari a destra)

1. Riscrivere ogni gruppo di produzioni del tipo $X \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ come $X = \alpha_1 + \alpha_2 + \dots + \alpha_n$
2. Poiché la grammatica è lineare a destra, ogni α_k ha la forma uX_k dove $X_k \in VN \cup \varepsilon$, $u \in VT^*$
Ergo, si raccolgano a destra i simboli non-terminali dei vari $\alpha_1 \dots \alpha_n$ scrivendo $X = (u_1 + u_2 + \dots) X_1 \cup \dots \cup (z_1 + z_2 + \dots) X_n$
dove $X_k \in VN$, $u_k, z_k \in VT^*$
Ciò porta a un sistema di M equazioni in M incognite dove M è la cardinalità dell'alfabeto VN (cioè il numero di simboli non terminali)
3. Eliminare dalle equazioni le ricorsioni dirette, data l'equivalenza
$$X = uX \cup \delta \quad \Leftrightarrow \quad X = (u)^* \delta$$

Ognuna delle forme di frase δ conterrà altre incognite, ma non X .
4. Risolvere il sistema rispetto a S per eliminazioni successive (metodo di Gauss), eventualmente ri-applicando (2) e (3) per trasformare le equazioni via via ottenute.
5. La soluzione del sistema è il linguaggio regolare cercato.

ALGORITMO

(grammatiche regolari a sinistra)

1. Riscrivere ogni gruppo di produzioni del tipo $X \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ come $X = \alpha_1 + \alpha_2 + \dots + \alpha_n$

2. Poiché la grammatica è lineare a sinistra, ogni α_k ha la forma $X_k u$ dove $X_k \in VN \cup \varepsilon$, $u \in VT^*$

Ergo, si raccolgano a sinistra i simboli non-terminali dei vari $\alpha_1 \dots \alpha_n$ scrivendo $X = X_1 (u_1 + u_2 + \dots) + \dots + X_n (z_1 + z_2 + \dots)$

dove $X_k \in VN$, $u_k, z_k \in VT^*$

Ciò porta a un sistema di M equazioni in M incognite dove M è la cardinalità dell'alfabeto VN (cioè il numero di simboli non terminali)

3. Eliminare dalle equazioni le ricorsioni dirette, data l'equivalenza

$$X = X u \cup \delta \quad \Leftrightarrow \quad X = \delta (u)^*$$

Ognuna delle forme di frase δ conterrà altre incognite, ma non X .

4. Risolvere il sistema rispetto a S per eliminazioni successive (metodo di Gauss), eventualmente ri-applicando (2) e (3) per trasformare le equazioni via via ottenute.

5. La soluzione del sistema è il linguaggio regolare cercato.



ESEMPIO

(grammatica lineare a destra)

Fase 1

- scrittura di un'equazione per ogni regola:

Grammatica data:

$$S \rightarrow a B \mid a S$$

$$B \rightarrow d S \mid b$$

Fase 2

- eventuali raccoglimenti a fattore comune per evidenziare suffissi: *qui non ce ne sono*

Equazioni:

$$S = a B + a S$$

$$B = d S + b$$

Fase 3

- eliminare la ricorsione diretta $X = u X + \delta$ riscrivendola come $X = u^* \delta$ (qui $\delta = a B$)

$$S = a^* a B$$

$$B = d S + b$$

Fase 4

- sostituzione della 2^a equazione nella 1^a e sviluppo dei relativi calcoli

$$\begin{aligned} S &= a^* a (d S + b) = \\ &= a^* a d S + a^* a b \end{aligned}$$

Fase 5

- nuova eliminazione della ricorsione introdotta al punto precedente: risultato finale.

$$S = a^* a d S + a^* a b$$

$$S = (a^* a d)^* a^* a b$$



ESEMPIO – VARIANTE

Fase 1

- scrittura di un'equazione per ogni regola:

Grammatica data:

$$S \rightarrow a B \mid a S$$

$$B \rightarrow d S \mid b$$

Fase 2

- se ora eliminiamo subito B, sostituendo la 2^a equazione nella 1^a e raccogliamo S:

Equazioni:

$$S = a B + a S$$

$$B = d S + b$$

Fase 3

- eliminando ora la ricorsione $X = u X + \delta$ riscrivendola come $X = u^* \delta$ (qui $\delta = a b$)

$$S = a (d S + b) + a S = (a d + a) S + a b$$

- che costituisce già una espressione regolare (risultato finale)

$$S = (a d + a)^* a b$$

Poco fa però avevamo ottenuto:

$$S = (a^* a d)^* a^* a b$$

non sembra affatto la stessa cosa.. ☹

RIFLESSIONE

LA PRIMA ESPRESSIONE ottenuta:

$$S = (a^* a d)^* a^* a b$$

LA SECONDA ESPRESSIONE ottenuta:

$$S = (a d + a)^* a b$$

Denoteranno lo stesso linguaggio? *si spera..!*

Una terza espressione (deterministica) equivalente:

$$S = a (d a + a)^* b$$

Fraasi del linguaggio:

ab, adab, aab, aadadab, ...

ossia tutte le frasi che iniziano per "a", terminano per "b", e hanno eventualmente in mezzo "a" o "da" ripetuti un numero arbitrario di volte.

In generale, uno stesso linguaggio può essere denotato da più espressioni regolari equivalenti.

RIFLESSIONE

Come si possono ottenere espressioni equivalenti?

- manipolando algebricamente quelle di partenza
 - la manipolazione algebrica diretta è ardua perché gli operatori hanno poche proprietà e quindi trasformare è faticoso e difficile
 - occorre capire "con fantasia" quale trasformazione applicare
- operando sulle "corrispondenti macchine"
 - lì *esistono algoritmi pratici* per trasformare macchine in altre macchine
 - il risultato finale può essere ri-trasformato in espressione regolare 😊

Espressioni regolari in Java [package java.util.regex]

- un'istanza della classe Pattern rappresenta un'espressione regolare, ossia *descrive il linguaggio* (metodo Pattern.compile)
- un'istanza della classe Matcher fa match con una stringa data, ossia *riconosce se la stringa data appartiene al linguaggio* denotato dall'espressione regolare medesima.



DALL'ESPRESSIONE REGOLARE ALLA GRAMMATICA

Per passare dall'espressione regolare alla grammatica si interpretano gli operatori in base alla loro semantica

- **sequenza** → simboli accostati nella grammatica
- **operatore +** → simbolo di alternativa nella grammatica (regole distinte)
- **operatore *** → regola ricorsiva nella grammatica (ciclo)

ESEMPIO: l'espressione regolare vista in precedenza

$$L = \{ a^* b \}$$

Tutte le frasi di L sono composte dal prefisso a^* (che può mancare) e dal suffisso b (che invece c'è sempre)

$$S \rightarrow A b \mid b$$

Il prefisso a^* può essere prodotto da una regola ricorsiva, del tipo:

$$A \rightarrow A a \quad \text{o anche} \quad A \rightarrow a A$$