UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 2
Linguaggi e Grammatiche

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

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 risponde 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 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) costruite nel linguaggio.

  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 *operazionale***
  - 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.

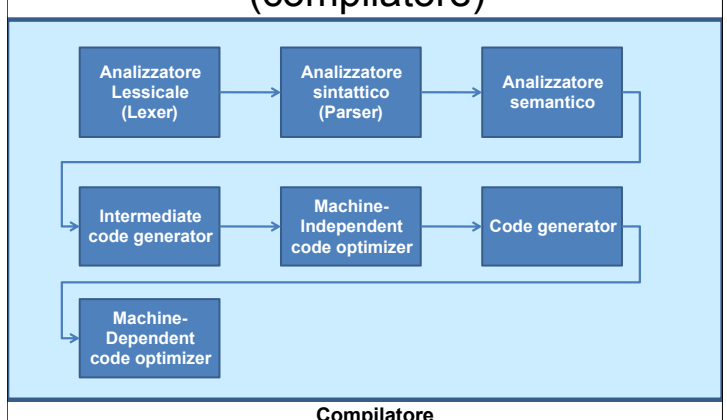
Un *compilatore* per un linguaggio L, invece:
- **accetta in ingresso un intero programma scritto in L**
- **e lo riscrive in un altro linguaggio** (più semplice).

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

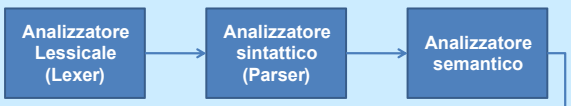
A volte la differenza è più sfumata di quel che si può pensare..

Creazione di un'applicazione (compilatore)

Creazione di un'applicazione (compilatore)



Analisi lessicale



- 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.)

Analisi Lessicale

- il primo scopo dell'analisi lessicale è dividere il testo nelle unità lessicali: gruppi di caratteri chiamate *lessemi* (Ingl. *lexemes*).

```
if (i2==j)
     z=230;
else z=1;
```

```
\nif (i2==j)\n\tz=230;\nelse z=1;\n
```

Token classes

- Il secondo scopo è classificare i lessemi a seconda del loro ruolo, chiamato token class
- Nei linguaggi naturali le classi potrebbero essere
  - *nome, verbo, aggettivo*, …
- Nei linguaggi di programmazione possono essere
  - *Whitespace*, *Keyword*, **(** , **)** , **=**, *Operator*, *Identifier*, **;**, *Number*

```
\nif (i2==j)\n\tz=230;\nelse z=1;\n
```

Token classes

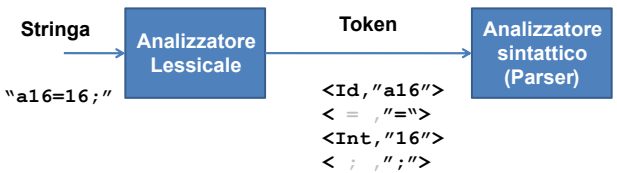
- Le token classes corrispondono a insiemi di stringhe
- Es:
  - Identifier: stringhe costituite da lettere e cifre, che iniziano con una lettera
  - Integer: una sequenza non vuota di cifre
  - Keyword: "if" oppure "while" oppure "else", …
  - Whitespace: sequenze non vuote di spazi, a capo, tabulazioni, …

Analisi lessicale

- Il risultato della lexical analysis è una sequenza di coppie

  <token class, string>

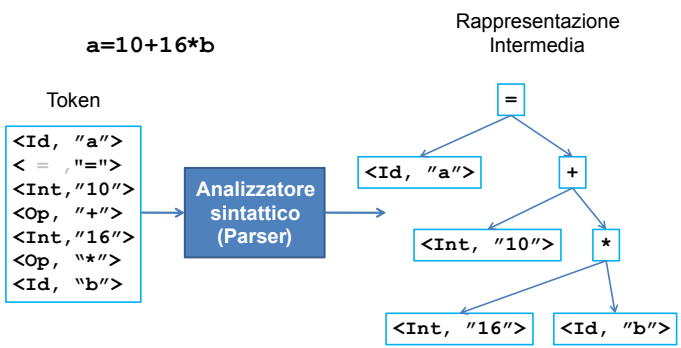
- dette *token*



ANALISI SINTATTICA

- **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 sottoforma di *opportuno albero*.

ANALISI SINTATTICA



L'albero mostra l'ordine in cui vanno eseguite le operazioni

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 di 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

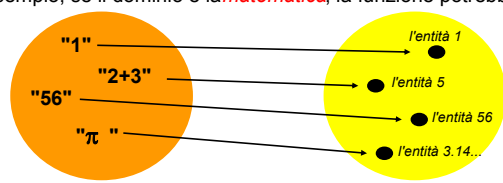
- **Chiedersi quale sia il *significato* di una frase significa *associare a quella frase un concetto* nella nostra mente**
  - Lo facciamo in base alla nostra cultura ed esperienza di vita

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 "0", "1", "2", .. "9"
    se consideriamo la nostra cultura attuale
    … ma Giulio Cesare avrebbe scelto "I", "V", "X", "L", …
  - 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

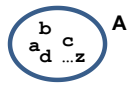
Alfabeto

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

Stringa

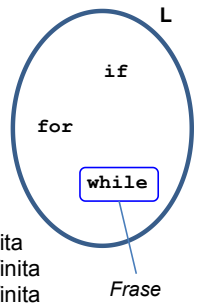
- **un stringa è una sequenza di simboli, ossia un elemento del prodotto cartesiano A^n.**
  Esempi: a ab aba bb …
- ***Lunghezza*** **di una stringa: il numero di simboli che la compongono.**
- ***Stringa vuota*** **ε : stringa di lunghezza zero.**
  ⇨ Si noti che $A^0 = \{ \varepsilon \}$

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* primo } linguaggio a cardinalità infinita
L3 = { $a^m b^n$, $n>0$ } linguaggio a cardinalità infinita

DESCRIZIONE DI UN LINGUAGGIO

Chiusura A* di un alfabeto A **(Kleene closure o linguaggio universale su A)**

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

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

A*
ε
a b c d e f g h i j k l
aa ab ac ad ae af ag ah
aaa aab aac aad aae aaf
aaaa aaab aaac aaad aaae
…

Chiusura positiva A⁺ di un alfabeto A

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

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

A+
a b c d e f g h i j k l
aa ab ac ad ae af ag ah
aaa aab aac aad aae aaf
…

SPECIFICA DI UN LINGUAGGIO

- **Problema: come specificare il sottoinsieme 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.*

Esempio con l'Italiano

simboli NON terminali

- Una **FRASE** è costituita da **SOGGETTO, VERBO** e **COMPLEMENTO**

Scopo

FRASE -> SOGGETTO VERBO COMPLEMENTO

- Il **SOGGETTO** e il **COMPLEMENTO** sono un **ARTICOLO** seguito da un **SOSTANTIVO**

SOGGETTO -> ARTICOLO SOSTANTIVO
COMPLEMENTO -> ARTICOLO SOSTANTIVO

Regole di produzione

- L'**ARTICOLO** può essere **il, lo, la, i, gli, le**

ARTICOLO -> il | lo | la | i | gli | le

- Il **VERBO** è **mangia**

VERBO -> mangia

- Il **SOSTANTIVO** può essere **gatto** o **topo**

simboli terminali

SOSTANTIVO -> gatto | topo

GRAMMATICA FORMALE

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

Una grammatica è una ***quadrupla*** ⟨ VT,VN,P,S⟩ 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 ***notazione formale*** con cui esprimere ***in modo rigoroso*** la ***sintassi*** di un linguaggio.

I simboli ***terminali*** sono ***caratteri*** o ***stringhe*** su un alfabeto A.

Una grammatica è una ***quadrupla*** ⟨ VT,VN,P,S⟩ 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

I simboli ***non terminali*** sono dei ***meta-simboli*** che rappresentano le diverse ***categorie sintattiche***

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

GRAMMATICHE: CONVENZIONI

CONVENZIONI SUI SIMBOLI

- **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.

Esercizio

Data la grammatica G
- VT = {0, 1}
- VN = {S, Z, U}
- P={ S -> ZU
  Z -> 0
  Z -> 0Z
  U -> 1
  U -> U1
  }

Si scrivano 3 forme di frase e 3 frasi che possono essere generate a partire dallo scopo S

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 $A \rightarrow \gamma$.

$\eta A \delta$

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**

$$\alpha = \alpha 0 \rightarrow \alpha 1 \rightarrow \alpha 2 \rightarrow \ldots \rightarrow \alpha N = \beta$$

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)**

$S \overset{+}{\Rightarrow} \sigma$ — **σ deriva da S con una o più applicazioni di produzioni (in uno o più passi)**

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

Esercizio

Data la grammatica G
- VT = {0, 1}
- VN = {S, Z, U}
- P={ S -> ZU
  Z -> 0
  Z -> 0Z
  U -> 1
  U -> U1
  }

mostrare una derivazione per la frase 0011

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

$L_G = \{ s \in VT^* \text{ tale che } S \overset{*}{\Rightarrow} s \}$

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 ∈ VN = F**
- **P = {**
  F → a b
  F → 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.

Esempio

Data la grammatica G:

- VT = {0,1}
- VN = {C, N}
- S = C
- P = {

  C -> 1 | 1 N

  N -> 1 | 0 | 1 N | 0 N

  }

dire qual è il linguaggio generato da G

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 dal 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

automi di diversa "potenza computazionale"

per *riconoscere* tali linguaggi.

CLASSIFICAZIONE DI CHOMSKY TIPO 3

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

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

lineare a destra	*lineare a sinistra*
$A \to \sigma$	$A \to \sigma$
$A \to \sigma B$	$A \to B \sigma$

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

Si intende che le produzioni di una data grammatica devono essere **TUTTE o lineari a destra, o lineari a sinistra– non mischiate.**

Si noti che σ può essere ε.

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	*lineare a sinistra*
$A \to \sigma$	$A \to \sigma$
$A \to \sigma B$	$A \to B \sigma$

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

lineare a destra	*lineare a sinistra*
$X \to a$	$X \to a$
$X \to a Y$	$X \to Y a$

GRAMMATICHE LINEARI: ESEMPI

VT = { a, +, – }, VN = { S }

- **Grammatica G1** (lineare a sinistra: $A \to B\ y$, con $y \in VT^*$)
  $S \to a$ $S \to S + a$ $S \to S - a$
- **Grammatica G2** (lineare a destra: $A \to x\ B$, con $x \in VT^*$)
  $S \to a$ $S \to a + S$ $S \to a - S$
- **Grammatica G3** (G2 resa *strettamente lineare* a destra)
  $S \to a$ $S \to a\ A$ $A \to + S$ $A \to - S$
- **Grammatica G4** (lineare a destra e anche a sinistra)
  $S \to ciao$
- **Grammatica G5** (G4 resa *strettamente lineare* a destra)
  $S \to c\ T$ $T \to i\ U$ $U \to a\ V$ $V \to o$

CLASSIFICAZIONE DI CHOMSKY
TIPO 2

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

- **Tipo 2: context free (*indipendenti dal contesto*):**
  produzioni vincolate alla forma:

$$A \rightarrow \alpha$$

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

Qui A può *sempre* essere sostituita da α, *indipendentemente dal contesto*.

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 1

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

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

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

con $\beta, \delta \in V^*$, $\alpha \in V^+$, $A \in VN$

$\alpha \neq \varepsilon$

Quindi, A può essere sostituita da α solo *nel contesto* β A δ
Le riscritture *non accorciano mai la forma di frase corrente.*

ESEMPIO

Esempio (grammatica di tipo 1)
S → aBC | aSBC
CB → DB DB → DC DC → BC
aB → ab bB → bb bC → bc cC → cc

Infatti, secondo la definizione $\beta\ A\ \delta \rightarrow \beta\ \alpha\ \delta$ 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 → aBC | aSBC	$\beta = \varepsilon \quad \delta = \varepsilon$
CB → DB	$\beta = \varepsilon \quad \delta = B$
DB → DC	$\beta = D \quad \delta = \varepsilon$
DC → BC	$\beta = \varepsilon \quad \delta = C$
aB → ab	$\beta = a \quad \delta = \varepsilon$
bB → bb	$\beta = b \quad \delta = \varepsilon$
bC → bc	$\beta = b \quad \delta = \varepsilon$
cC → cc	$\beta = c \quad \delta = \varepsilon$

ESEMPIO

Esempio (grammatica di tipo 1)
S → aBC | aSBC
CB → DB DB → DC DC → BC
aB → ab bB → bb bC → bc cC → cc

S
→ aSBC
→ aaSBCBC
→ aaaSBCBCBC
→ aaaaSBCBCBCBC
→ aaaaaBCBCBCBCBC
…
→ aaaaaBBBBBCCCCC
…
→ aaaaabbbbbccccc

Riconoscibilità delle grammatiche di tipo 1

- **Teorema [Chomsky, 1959]**: Il linguaggio generato da una grammatica *G* di tipo 1 è riconoscibile da una macchina di Turing
- **Dim**: Nelle grammatiche di tipo 1, in una derivazione le forme di frase si allungano sempre

$$S \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \dots$$

- Supponiamo di dover riconoscere una stringa *s*, di lunghezza *l* = |*s*|
- Bozza di algoritmo:
  partendo dallo scopo *S*, genera tutte le derivazioni in cui
  - non ci sono ripetizioni (loop nella derivazione)
  - la lunghezza massima è al massimo |*s*| (non appena si arriva ad una forma di frase di lunghezza > |*s*| ci si può fermare)
- Le stringhe che hanno lunghezza fino a |*s*| sono in numero finito, quindi l'elenco è di dimensione finita

Altra definizione

… ma se il teorema si basa sul fatto che le forme di frase si allungano solo durante una derivazione, non avrebbe più senso definire il Tipo 1 basandosi sulle lunghezze delle stringhe?

- Se facessi così, potrei forse includere anche altre grammatiche
- Sì, ma non potrei generare altri linguaggi, perché si riesce a generare una grammatica equivalente che rispetta la regola del contesto $\beta\ A\ \delta \rightarrow \beta\ \alpha\ \delta$

G=<{b,f,g}, {A,C,D,E,H,S},S,P>
P={
S → AbK
K → CD
AbCD → EfgH
Ef → ef
gH → gh
}

G=<{b,f,g}, {A,C,D,E,H,S,B},S,P>
P={
S → ABK
K → CD
ABCD → EfgH
Ef → ef
gH → gh
B → b
}

AbCD → EbCD
EBCD → EfCD
EfCD → EfgD
EfgD → EfgH

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)

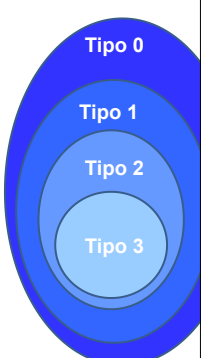
S → aSBC CB → BC SB → bF FB → bF
FC → cG GC → cG G → ε

Possibile derivazione: **S → aSBC → abFC → abcG → abc**
lung=4 lung=3

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 convienedi ammettere nelle grammatiche tipo 1 anche la produzione **S → ε**, dove **S** non compare a destra in alcuna produzione

Esercizio (29 giugno 2016)

- Si consideri la seguente grammatica

$$S \rightarrow KX$$
$$aX \rightarrow aYb$$
$$bX \rightarrow bYa$$
$$K \rightarrow a \mid b$$
$$Y \rightarrow aYb \mid a$$

- Si classifichi la grammatica secondo Chomsky.
- Si scriva il linguaggio L_S generato dallo scopo S.
- Si scriva una grammatica di tipo 2o di tipo 3 non ambigua ed equivalente alla grammatica data.

$S \rightarrow KX$
$aX \rightarrow aYb$
$bX \rightarrow bYa$
$K \rightarrow a \mid b$
$Y \rightarrow aYb \mid a$

- Si scriva il linguaggio L_S generato dallo scopo S.

$S \rightarrow KX$
$aX \rightarrow aYb$
$bX \rightarrow bYa$
$K \rightarrow a \mid b$
$Y \rightarrow aYb \mid a$

- Si scriva una grammatica di tipo 2o di tipo 3 non ambigua ed equivalente alla grammatica data.

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:**

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

- **Viceversa, le grammatiche di Tipo 2 *la ammettono*:**

  $A \rightarrow \alpha \qquad \alpha \in V^*$ (α può essere ε)

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

lin. a destra	*lin. a sinistra*	
$A \rightarrow \sigma$	$A \rightarrow \sigma$	
$A \rightarrow \sigma\ B$	$A \rightarrow B\ \sigma$	$\sigma \in VT^*$ (σ può essere ε)

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 ?**

L'*assenza di contraddizione* è dovuta al seguente teorema, secondo il quale **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→ ε

CLASSIFICAZIONE DI CHOMSKY
IL PROBLEMA DELLA STRINGA VUOTA

TEOREMA

Se G è una grammatica *context free* con produzioni della forma $A \to \alpha$, 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*** $A \to \alpha$, **con** $\alpha \in V^+$ (α ***non*** **è** ε)
- ***oppure la forma*** $S \to \varepsilon$,

ed *S non compare sulla destra in nessuna produzione*

In pratica, il teorema assicura che **la sola differenza fra una grammatica context free con ε *-rules* o senza esse 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

- Idea base: se si ha una regola

  $A \to \alpha \mid \beta \mid \varepsilon$

  allora si può eliminare la regola $A \to \varepsilon$ se in tutti i punti in cui compare A sostituisco con (A | ε)

A → BC B → aD | ε C → DB D → bD | ε	A → BC B → a (D | ε) | ε C → (D | ε) B D → b (D | ε)	A → BC B → aD | aε | ε C → DB | εB D → bD | bε	A → BC B → aD | a | ε C → DB | B D → bD | b
A → (B| ε) C B → aD | a C → D(B| ε) | (B| ε) D → bD | b	A → BC | C B → aD | a C → DB| D | B| ε D → bD | b	A → B (C| ε) | (C| ε) B → aD | a C → DB| D | B D → bD | b	A → BC| B | C | ε B → aD | a C → DB| D | B D → bD | b

Perché l'idea base non basta

- Partendo da una grammatica G che genera un linguaggio L(G), vogliamo creare una grammatica G', senza ε-rules che che genera lo stesso linguaggio (eventualmente, tolta la stringa vuota).

  $L(G') = L(G) - \{\varepsilon\}$
- Eventualmente, se ci interessa generare esattamente lo stesso linguaggio, accetteremo di aggiungere un'unica ε-rule: $S \to \varepsilon$ (dove S è lo scopo della grammatica)
- Applicando meccanicamente l'idea base, due cose possono andare storte

1 - Scopo compare a destra

- Se lo scopo S compare sul lato destro di qualche regola, non è più vero che aggiungere o togliere la regola S → ε equivale ad aggiungere o togliere la stringa vuota dal linguaggio

S → Ub
U → ab | S

L(S) = { abb* b }

S → Ub | ε
U → ab | S

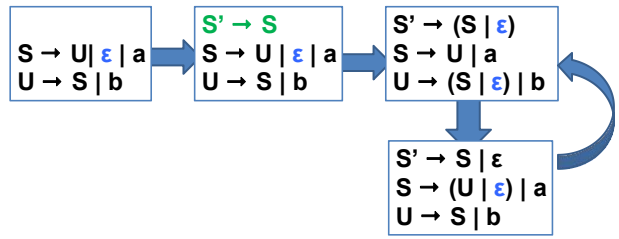
L(S) = {abb*b} ∪ {ε} ∪ {b*b}

- **Soluzione**: in questi casi aggiungiamo un nuovo scopo fittizio S' con la regola S' → S

S' → S | ε
S → Ub
U → ab | S

L(S') = {abb*b} ∪ {ε}

2 - loop



Algoritmo completo

1. calcolo dell'insieme *NULL* dei nonterminali da cui si può derivare la stringa vuota

   NULL ← { $A \in VN \mid A \to \varepsilon \in P$}
   ripeti finché *NULL* non cambia più
   NULL ← *NULL* ∪ { $A \in VN \mid A \to B_1 \ldots B_k \in P$, $\forall i, B_i \in NULL$}

   Tutti i B_i devono essere in *NULL*, quindi nella parte destra della regola ci sono solo nonterminali

2. sostituire nelle regole ogni simbolononterminale $X \in NULL$ con $(X|\varepsilon)$ (e distribuire)
3. se è presente la regola che riscrive lo scopo $S \to \varepsilon$, aggiungere un nuovo scopo $S' \to S|\varepsilon$;
4. rimuovere tutte le regole $X \to \varepsilon$ (esclusa eventualmente quella che riscrive lo scopo in ε)

L'esempio precedente

S → U| ε | a
U → S | b

1. Calcolo *NULL* — ***NULL = {S,U}***

S → (U| ε) | ε | a
U → (S| ε) | b

2. Per ogni $X \in NULL$ sostituire con $(X|\varepsilon)$

S → U | ε | a
U → S | ε | b

Distribuire

S' → S | ε
S → U | ε | a
U → S | ε | b

3. Se c'è la regola $S \to \varepsilon$, aggiungere un nuovo scopo *S'*

S' → S | ε
S → U | a
U → S | b

4. Eliminare le altre regole $X \to \varepsilon$

Riassumendo

- Le grammatiche di tipo 1 non ammettono la stringa vuota nelle produzioni, mentre quelle di tipo 0, di tipo 2 e di tipo 3 le ammettono.
- I linguaggi generati dalle grammatiche di tipo *N* sono comunque un sottoinsieme dei linguaggi generati da grammatiche di tipo *N*-1 (a parte, eventualmente, la presenza della stringa vuota nel linguaggio generato).
- Infatti, data una grammatica di tipo 2 con ε-rules, è sempre possibile trovare una grammatica equivalente che ha al più una ε-rule, in corrispondenza dello scopo.
- Abbiamo visto un algoritmo che genera la grammatica equivalente

Esercizio (25 lug 2017)

- Si consideri la grammatica
  G = <{a, b, c, d}, {S, A, B, C}, P, S>, dove:

$$P= \begin{cases} S \rightarrow ABC \\ A \rightarrow aAd \mid \varepsilon \\ B \rightarrow bBd \mid \varepsilon \\ C \rightarrow c \mid dA \end{cases}$$

- Se possibile, si scriva una grammatica G' equivalente a G che non abbia produzioni con la stringa vuota.
  Se non è possibile, si motivi il perché.

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 2
Linguaggi e Grammatiche

Gerarchia di Chomsky e Linguaggi

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

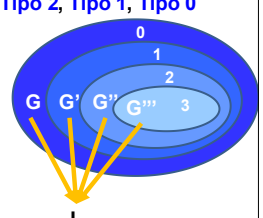
Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

GRAMMATICHE e LINGUAGGI

Poiché le grammatiche sono in relazionegerarchica, **può accadere che un linguaggio possa essere generato da più grammatiche, *anche di tipo diverso***

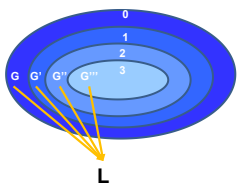
- **un linguaggio generato da una grammatica di Tipo 3 potrebbe essere generato anche da grammatiche di Tipo 2, Tipo 1, Tipo 0**
- **un linguaggio generato da una grammatica di Tipo 2 potrebbe essere generato anche da grammatiche di Tipo 1, Tipo 0**
- **un linguaggio generato da una grammatica di Tipo 1 potrebbe essere generato anche da grammatiche di Tipo 0**



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

CLASSIFICAZIONE DEI LINGUAGGI

- Diremo che **un linguaggio è di un certo tipo se quello 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 indipendenti 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 → aBC | aSBC
CB → DB **DB → DC** **DC → BC**
aB → ab **bB → bb** **bC → bc** **cC → cc**

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

G2

S → aBC | aSBC
CB → BC
aB → ab **bB → bb** **bC → bc** **cC → cc**

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

G0

S → aSBC **CB → BC** **SB → bF** **FB → bF**
FC → cG **GC → cG** **G → ε**

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

Una grammatica ancora più semplice potrebbe essere:

G3

S → abc | aBSc
Ba → aB
Bb → 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 → aBC | aSBC
CB → BC
aB → ab
bB → bb
bC → bc
cC → cc

Derivazione:

S → aSBC → aaBCBC → aaBBCC →
→ aabBCC → aabbCC → aabbcC → aabbcc

S
aSBC
aaBCBC
aaBBCC
aabBCC
aabbCC
aabbcC
aabbcc

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

Derivazione della frase aabbcc

Grammatica G2:

S → aBC | aSBC
CB → BC
aB → ab
bB → bb
bC → bc
cC → cc

Derivazione:

S → aSBC → aaBCBC → aaBBCC →
→ aabBCC → aabbCC → aabbcC → aabbcc

Grammatica G3:

S → abc | aBSc
Ba → aB
Bb → bb

Derivazione:

S → aBSc → aBabcc → aaBbcc → 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 → aBC | aSBC
CB → BC
aB → ab
bB → bb
bC → bc
cC → cc

S
aSBC
aaBCBC
aaBBC C
aabBCC
aabb CC
aab bcC
aabbcc

S
aSBC
aaBCBC
a abCBC
aabc BC
???

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*, mentre il Tipo 2 ammette in tale posizione *un unico metasimbolo!***

GRAMMATICHE DI TIPO2 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 TIPO2 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 ∪ VN)* , A ∈ VN]

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

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

SELF - EMBEDDING

Una grammatica è *self-embedding* quando esiste un nonterminale A tale che

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

TEOREMA: una grammatica di Tipo 2 *che non sia 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, 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 unlinguaggio 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):

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

Il ruolo del self-embedding è introdurre una ricorsione in cui *si aggiungono contemporaneamente simboli a sinistra 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 $L(G) = \{ (^n a)^n \quad n \geq 0 \}$

SELF - EMBEDDING

Una grammatica è *self-embedding* quando esiste un nonterminale A tale che

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

Nota: Perché il self-embedding sia evidente, possono essere necessari anche più passi di derivazione

Es:

$S \rightarrow aA$
$A \rightarrow aB$
$B \rightarrow Ac$
$A \rightarrow \varepsilon$

$A \rightarrow aB \rightarrow aAc$

"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 èbanale

  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 → a S a | X
X → a X | b X | a | 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 una **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 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 ≥0 }
- Grammatica di Tipo 3 equivalente: **S → X a**
  X → a b | X a b a b

"FINTO" SELF – EMBEDDING (44)

ESEMPIO 3

S → a S a | ε

L(G) = { **(a a)**^{n} , n ≥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 → a a S | ε**

L'osservazione precedente è generalizzata dal seguente

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

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 2
Linguaggi e Grammatiche

Riconoscibilità dei Linguaggi
Backus-Naur Form
Alberi di derivazione

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

RICONOSCIBILITÀ DEI LINGUAGGI

- **I linguaggi generati da grammatiche di Tipo0 *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

- Perché il traduttore possa essere realizzato in modo efficiente, conviene 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**
- 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?

LINGUAGGI	AUTOMI RICONOSCITORI
• **Tipo 0**	• ***Se L(G) è riconoscibile*, serve una Macchina di Turing**
• **Tipo 1**	• **Macchina di Turing** ***(con nastro di lunghezza proporzionale alla frase da riconoscere)***
• **Tipo 2** ***(context-free)***	• **Push-Down Automaton (PDA)** ***(cioè ASF + stack)***
• **Tipo 3** ***(regolari)***	• **Automa a Stati Finiti (ASF)**

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 imetasimboli
  - nelle tastiere e nei font "di base", non ci sono frecce e altri simboli particolari ➔ sarebbe meglio farne senza

GRAMMATICHE Backus-Naur Form

In una *Grammatica BNF*

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

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

$X ::= A_1$ .... $X ::= A_N$

in forma compatta:

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

ESEMPIO 2

G = ⟨ **VT,VN,P,S**⟩ , dove:

VT = { il, gatto, topo, sasso, mangia, beve }
VN = { <frase>, <soggetto>, <verbo>, <compl-ogg>, <articolo>, <nome> }
S = <frase>
P = {
<frase> ::= <soggetto> <verbo> <compl-ogg>
<soggetto> ::= <articolo><nome>
<articolo> ::= il
<nome> ::= gatto | topo | sasso
<verbo> ::= mangia | beve
<compl-ogg> ::= <articolo> <nome>
}

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
X ::= [a]B	X ::= B | aB	a può comparire 0 o 1 volta
X ::= {a}^{n}B	X ::= B | aB |…| a^nB	a può comparire da 0 a n volte
X ::= {a}B	X ::= B | aX	a può comparire 0 o più volte

NOTA: la produzione X ::= B | aX è ricorsiva (a destra).

Forma EBNF	BNF equivalente	significato
X ::= (a|b)D | c	X ::= aD | bD | c	raggruppa categorie sintattiche

ESEMPIO 3: NUMERI NATURALI

G = ⟨ VT,VN,P,S⟩

dove:

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

VN = { <num>, <cifra>, <cifra-non-nulla> }

S = <num>

P = {

EBNF

<num> ::= <cifra> | <cifra-non-nulla> {<cifra>}

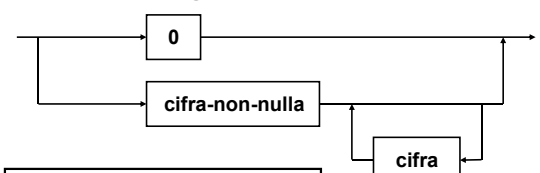
<cifra> ::= 0 | <cifra-non-nulla>

<cifra-non-nulla> ::=1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

}

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 = ⟨ VT,VN,P,S⟩ , dove:

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

VN = {<int>, <num>,
<cifra>, <cifra-non-nulla> }

P = {

<int> ::= [+|-] <num>

Lo scopo ora è <int>, non più <num>

<num> ::= <cifra> | <cifra-non-nulla> {<cifra>}

<cifra> ::= 0 | <cifra-non-nulla>

<cifra-non-nulla> ::=1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 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 <…>
- il primo di essi è il simbolo iniziale

P = {

scopo

<id> ::= <lettera> { <lettera> | <cifra>}

VN VN

<lettera> ::= A | B | C | D | ... | Z

VT

<cifra> ::= 0|1|2|3|4|5|6|7|8|9

VT

}

ALBERI DI DERIVAZIONE

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

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

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

dove A_i è il simbolo associato al nodo a_i

RIPRENDENDO L' ESEMPIO 2

Derivazione della frase *"il gatto mangia il topo"* (ammesso che tale frase *sia* derivabile)

P = {

<frase> ::= <soggetto> <verbo> <compl-ogg>

<soggetto> ::= <articolo><nome>

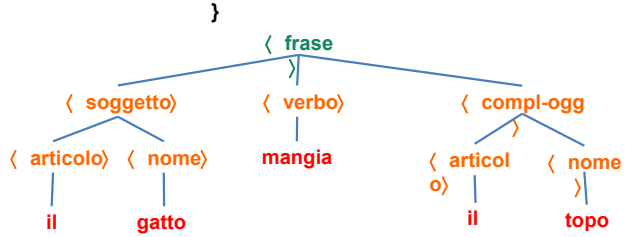
<articolo> ::= il

<nome> ::= gatto | topo | sasso

<verbo> ::= mangia | beve

<compl-ogg> ::= <articolo> <nome>

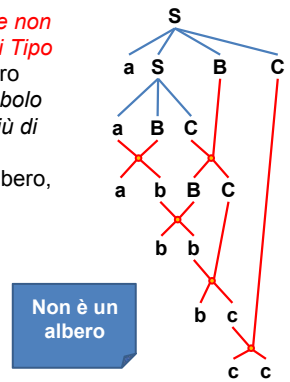
}



ALBERI DI DERIVAZIONE

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).

S → aBC | aSBC
CB → BC
aB → ab
bB → bb
bC → bc
cC → cc



RIPRENDENDO L' ESEMPIO 4

G = ⟨ VT,VN,P,S⟩ , dove:

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

VN = {<int>, <num>,
<cifra>, <cifra-non-nulla> }

P = {

EBNF

<int> ::= [+|-] <num>
<num> ::= <cifra> | <cifra-non-nulla> {<cifra>}
<cifra> ::= 0 | <cifra-non-nulla>
<cifra-non-nulla> ::=1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

}

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 ↔ X ::= B | aX

X ::= B {a} ↔ X ::= B | Xa

- **Dunque, la regola:**

<num> ::= <cifra-non-nulla> {<cifra>}

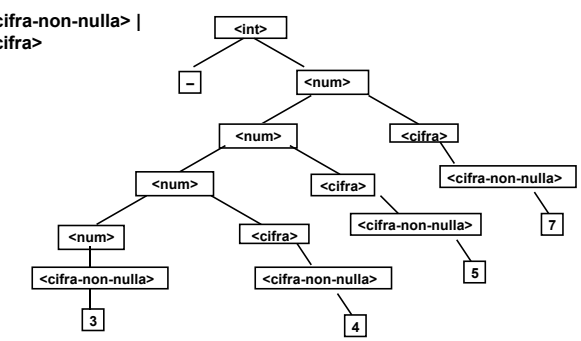
va riscritta come:

<num> ::= <cifra-non-nulla> | <num> <cifra>

RIPRENDENDO L' ESEMPIO 4

Albero di derivazione del numero intero-3457:

<num> ::= <cifra-non-nulla> |
<num> <cifra>



UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 2
Linguaggi e Grammatiche

Ambiguità

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

DERIVAZIONI CANONICHE

- In molti casi, esistono più derivazioni per la stessa frase

  S → ABC A → a |Aa
  B→ b | Bb C → c |Cc

S→ABC→AaBC→AaBbC→AabbC→AabbCc→Aabbcc→aabbcc
S→ABC→ABCc→ABcc→ABbcc→Abbcc→Aabbcc→aabbcc

- in queste ho applicato le stesse regole, ma in ordine diverso: ho fatto scelte diverse nella selezione del non-terminale da riscrivere
- In una grammatica context-free, il fatto di selezionare per primo un non-terminale o un altro non cambia il fatto che la frase sia derivabile
- Ha senso stabilire un ordine "canonico"

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)***

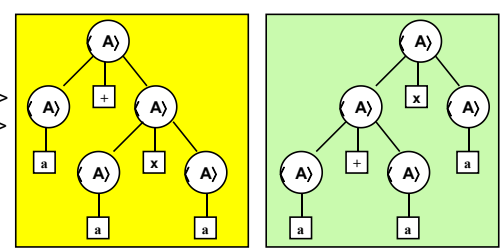
- **A partire dallo scopo della grammatica, si riscrive sempre *il simbolo non-terminale più a destra*.**

S → ABC
A → a | Aa
B→ b | Bb
C → c | Cc

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 può essere *indesiderabile*, a seconda delle applicazioni

ESEMPIO
<A> ::= <A> + <A>
<A> ::= <A> x <A>
<A> ::= a
La frase a+axa è *ambigua*:



AMBIGUITÀ

- **Purtroppo, stabilire se una grammatica di Tipo 2 sia ambigua è un problema *indecidibile***
  - in pratica, spesso un certo numero di derivazioni è sufficiente per "convincersi" (non per dimostrare!) dell'ambiguità di G
- **Se una grammatica è ambigua, spesso se ne può trovare un'altra che non lo sia –*ma non sempre.***

Esercizio (21 set 2016)

- Si consideri il linguaggio

$$L = \{ab^n \mid n \geq 0\} \cup \{a^n b^n \mid n \geq 0\}.$$

- Si scriva una grammatica non ambigua per il linguaggio *L*.
  - Lo studente cerchi di trovare una grammatica di livello più basso possibile nella classificazione secondo Chomsky, intendendo il livello 3 come minimo e il livello 0 come massimo.
- Si mostri poi l'albero di derivazione della stringa *ab*.

$$L = \{ab^n \mid n \geq 0\} \cup \{a^n b^n \mid n \geq 0\}.$$

$$L = \{ab^n \mid n \geq 0\} \cup \{a^n b^n \mid n \geq 0\}.$$

Linguaggio Intrinsecamente Ambiguo

- **Un linguaggio si dice *intrinsecamente ambiguo* se *tutte le grammatiche che lo generano sono ambigue***
- ***Es:*** **L = { $a^n b^n c^m d^m$ } ∪ { $a^n b^m c^m d^n$ }** con n,m≥ 0
- Si scriva una grammatica per il linguaggio L.
- Si mostrino poi due alberi di derivazione per la stringa *aabbccdd*.

Riassumendo

- L'ambiguità può essere indesiderabile, a seconda delle applicazioni
- Una grammatica è ambigua se esistono due derivazioni canoniche sinistre diverse per qualche frase
- Questo è equivalente a dire che esistono due alberi di derivazione diversi per quella frase
- Esistono linguaggi intrinsecamente ambigui

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 $\alpha \in V^+$, $\beta \in V^*$**
  - Infatti, in un tale linguaggio accade che le formedi frase *si accorcino* 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 ::= ε | X
X ::= ab | a X b

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

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

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 2
Linguaggi e Grammatiche

Forme Normali
Trasformazioni importanti

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

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 c'è un nonterminale A che ha come unica produzione $A \rightarrow B$** 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:**

  $S \to d\,A \mid c\,B$
  $A \to d\,A\,A \mid c\,S \mid c$
  $B \to c\,B\,B \mid d\,S \mid d$

- ***Forma normale di Chomsky***

  $S \to M\,A \mid N\,B$ $\quad M \to d$ $\quad N \to c$
  $A \to M\,P \mid N\,S \mid c$ $\quad P \to A\,A$
  $B \to N\,Q \mid M\,S \mid d$ $\quad Q \to B\,B$

La trasformazione in forma diGreibach 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**
  - **l'eliminazione della ricorsione sinistra.**

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

SOSTITUZIONE

La sostituzione consiste nell'espandere un simbolo non terminale che compare nella parte destra di una regola di produzione, sfruttando a tale scopo le regole in cui compare a sinistra.

Nella grammatica a lato è possibile sostituire il metasimbolo S nella seconda produzione, usando a tale scopo la prima produzione.	ESEMPIO $S \to X\,a$ $X \to b\,Q \mid S\,c \mid d$
Espandiamo quindi S come indicato: la nuova regola per X non contiene più alcun riferimento a S	ESEMPIO $S \to X\,a$ $X \to b\,Q \mid X\,a\,c \mid d$

SOSTITUZIONE

La sostituzione consiste nell'espandere un simbolo non terminale che compare nella parte destra di una regola di produzione, sfruttando a tale scopo le regole in cui compare a sinistra.

Nella grammatica a lato, il metasimbolo S compare in due regole a sinistra del simbolo di produzione →.	ESEMPIO 2 $S \to X\,a$ $S \to c\,X$ $X \to b\,Q \mid S\,c \mid d$
Prima di sostituire è necessario raggrupparle in un'unica produzione	$S \to X\,a \mid c\,X$ $X \to b\,Q \mid S\,c \mid d$
A questo punto è possibile sostituire	$S \to X\,a \mid c\,X$ $X \to b\,Q \mid (X\,a \mid c\,X)\,c \mid d$
Infine si ottiene	$S \to X\,a \mid c\,X$ $X \to b\,Q \mid X\,a\,c \mid c\,X\,c \mid d$

IL RACCOGLIMENTO A FATTOR COMUNE

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

Nella grammatica a lato è possibile isolare il prefisso a S comune alle prime due produzioni.	ESEMPIO S → a S b | a S c
Raccogliamo quindi a fattore comune il prefisso comune a S …	ESEMPIO S → a S (b | c)
…e introduciamo un nuovo meta-simbolo X per esprimere *la parte che segue* il prefisso comune.	ESEMPIO S → a S X X → b | c

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 → X a
  X → b S | S c | d
- ***Forma normale di Greibach*** (A → p α , A∈ VN, p∈ VT, α ∈ VN*)
  - eliminazione ciclo ricorsivo a sinistra
  - eliminazione ricorsione sinistra diretta
  - sostituzione
  - ridenominazione dei terminali tramite non-terminali ausiliari

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 A → B a B → C b C → A c | p
Fase 1 • **si modificano tutte le produzioni del tipo Y → Xα *in cui* Y > X, sostituendo a X le forme di frase stabilite dalle produzioni relative a X**	Si ottiene quindi: A → B a B → C b C → C b a c | p
Fase 2 • **le produzioni ricorsive dirette X → X α | p si modificano introducendo un metasimbolo Z e scrivendo X → p | p Z e Z → α | α Z**	Ergo, C → C b a c | p diventa C → p | p Z Z → b a c | b a c Z

ESEMPIO 2

Fase 1 **relazione d'ordine fra i simboli non terminali coinvolti del ciclo ricorsivo: X > S**	Grammatica data: S → X a X → b S | S c | d
Fase 2 **modifica della produzione X → S c sostituendo a S la produzione S → X a**	Si ottiene quindi: S → X a X → b S | X a c | d
Fase 3 **Raggruppiamo per semplicità i casi base della ricorsione sinistra in un nuovo simbolo D**	S → X a X → D | X a c D → bS | d
Fase 4 **eliminazione della ricorsione sinistra diretta X → X α | D introducendo il meta-simbolo Z tale che Z → α | α Z e X ::= D Z | D**	S → X a X → D | DZ D → bS | d Z → a c | a c Z
Fase 5 • **sostituzione del simbolo D in X e X in S.** **D non serve più**	X → bS | d | bSZ |dZ S → bSa| da| bSZa|dZa Z → a c | a c Z
Fase 6 • **introduzione dei non-terminali ausiliari A e C per rappresentare a e c dove appropriato. X non serve più**	S → b S A | dA|bSZA|dZA Z → a C | a C Z A → a C → c

IL "PUMPING LEMMA"

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**
  - 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 SIA 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 "pompata" 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 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.

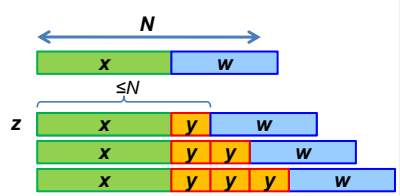
(*) in realtà non è neppure di Tipo 2, come si dimostra-applicando il lemma.

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*:

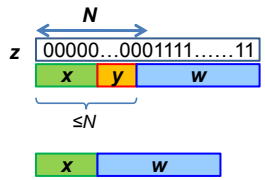
- ***z* può essere riscritta come: $z = xyw$** $|z| \geq N$
- **la parte centrale *xy* ha lunghezza limitata:** $|xy| \leq N$
- **y non è nulla:** $|y| \geq 1$
- **la parte centrale può essere *pompata quanto si vuole* ottenendo sempre altre frasi del linguaggio; ovvero,** $xy^iw \in L \quad \forall\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

•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

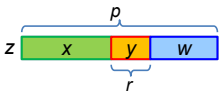
- Linguaggio costituito dalle stringhe che hanno lo stesso numero di 0 e di 1
- Se fosse regolare, esisterebbe un N che soddisfa il pumping lemma
- Consideriamo la stringa $z=0^N1^N$
- Per il pumping lemma, z si può scomporre in $z = xyw$, con $|xy| \leq N$ e $|y|>0$
- Chiaramente, x e y contengono solo 0, mentre w contiene 1^N
- Per il pumping lemma, anche la stringa xw appartiene al linguaggio
- ma xw ha N simboli 1, ma meno di N simboli 0 → **assurdo**



ESEMPIO

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

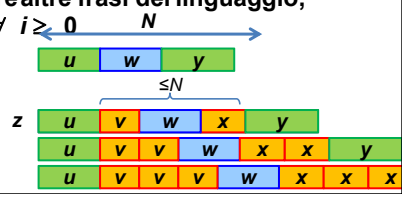
- se L fosse regolare, esisterebbe un intero N in grado di soddisfare il pumping lemma; sia allora p un primo $\geq N+2$ (che esiste perché i numeri primi sono infiniti):
  consideriamo allora la stringa $z = a^p$
- scomponiamo ora z nei tre pezzi xyw;
  sia $r = |y|$; ne segue che $|xw| = p - r$
- in base al lemma, se L fosse regolare, qualunque stringa xy^iw dovrebbe appartenere al linguaggio.
- In particolare, prendiamo la stringa $xy^{p-r}w$: 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 L non è regolare.



IL PUMPING LEMMA per linguaggicontext-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 vwx ha lunghezza limitata:** $|vwx| \leq N$
- **v e x non sono entrambe nulle:** $|vx| \geq 1$
- **la 2ª e la 4ª parte possono essere *"pompate" quanto si vuole* ottenendo sempre altre frasi del linguaggio; ovvero, $uv^iwx^iy \in L \quad \forall\ i \geq 0$**

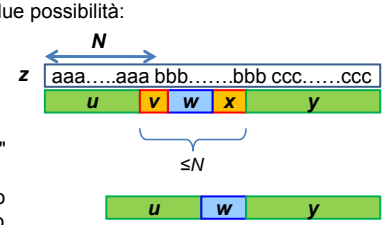


ESEMPIO 2 (1)

$L = \{a^n b^n c^n\}$ 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 l'una, non contiene l'altra. Questo apre due possibilità:

1. "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 → assurdo



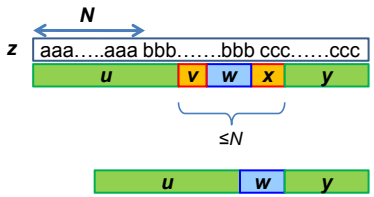
ESEMPIO 2 (2)

$L = \{a^n b^n c^n\}$ 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 l'una, non contiene l'altra. Questo apre due possibilità:

2. "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 → assurdo.



UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 2
Linguaggi e Grammatiche

Espressioni regolari

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

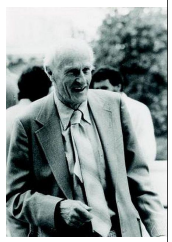
Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

Espressioni regolari

- Un formalismo di particolare interesse [per descrivere linguaggi] è quello delle *espressioni regolari*
- Le espressioni regolari sono ampiamente usate in
  - editor di testo avanzati
  - comandi Unix (e.g., **`grep`**)
  - strumenti per l'analisi lessicale (e.g., **`lex`**)
  - librerie in linguaggi di programmazione (java.util.regex, Perl 5, .NET, Python, …)



Stephen Cole Kleene
Foto: Konrad Jacobs, Erlangen, Copyright is MFO Mathematisches Forschungsinstitut

Composizione di parole

- Un modo per esprimere linguaggi è definire un insieme di operazioni fra linguaggi, partendo da operazioni sulle parole.
- Un'operazione significativa sulle parole è la composizione o concatenazione:

  $w \circ w'$

  giustapposizione dei simboli della prima parola alla seconda
- Es

  abc ∘ 1a2 = abc1a2
- Il simbolo ∘ è spesso sottointeso

Operazioni regolari fra linguaggi

- Unione

$$A \cup B = \{a | a \in A \vee a \in B\}$$

- Composizione

$$A \circ B = \{a \circ b \mid a \in A, b \in B\}$$

- Iterazione (o chiusura di Kleene)

$$A^* = \{a_1 \circ a_2 \circ a_3 \ldots \circ a_n \mid a_i \in A, n \in \mathbb{N}\}$$

O anche

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

dove $A^0 = \varepsilon$

e $A^k = A^{k-1} \circ A$

Espressioni regolari

Definizione induttiva.

Simbolo ➡ Linguaggio

sono espressioni regolari:

- *la stringa vuota* ε ➡ $\{\varepsilon\}$
- dato un alfabeto *A*, *ogni elemento* $a \in A$ a ➡ $\{a\}$
- l'insieme vuoto $\emptyset$ ➡ $\emptyset$

Se *A* e *B* sono espressioni regolari, lo sono anche

- l'unione $A+B$ ➡ $A \cup B$
- la composizione $A \circ B$ ➡ $A \circ B$
- la chiusura A^* ➡ A^*

priorità

min

max

UN PRIMO ESEMPIO

ESEMPIO

X1 = {00, 11}

X2 = {01, 10}

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

X1 ∘ X2 = {0001, 1101, 0010, 1110}

X2 ∘ X1 = {0100, 0111, 1000, 1011}

X1* = {ε , 00,11, 0000,0011,1100,1111, 000000, 000011, 001100, 001111, 110000, 110011, 111100, 111111, … }

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

Esempi

- L'insieme delle cifre in base 10

digit = 0+1+2+…+ 9

- L'insieme delle stringhe che rappresentano numeri naturali

digit digit*

- L'insieme di tutte le parole su un alfabeto dato A

A*

- L'insieme delle stringhe sull'alfabeto A={0,1} che iniziano e finiscono per 1

1(0+1)*1 + 1

Esercizi

- Scrivere delle espressioni regolari che rappresentano
- il linguaggio sull'alfabeto A={0,1} delle stringhe che contengono due "1"

- Il linguaggio che contiene le stringhe in cui "0" e "1" si alternano

Equazioni con espressioni regolari

- Si possono anche scrivere equazioni con espressioni regolari.
- Es: se α e β sono espressioni regolari, l'equazione

$$X = \alpha X + \beta$$

è un'equazione con incognita X
- Una soluzione dell'equazione è

$$X = \alpha^*\beta$$

infatti $\alpha X + \beta = \alpha\alpha^*\beta + \beta = (\alpha\alpha^*+\varepsilon)\beta = \alpha^*\beta = X$
- Si possono scrivere anche sistemi di equazioni

Linguaggi Regolari $\subseteq$ Espressioni Regolari

- Sia data una grammatica regolare a destra

$$A_1 \rightarrow a\,A_1 \mid b\,A_2 \mid \ldots \mid w\,A_n \mid z$$
$$A_2 \rightarrow b\,A_1 \mid a\,A_2 \mid \ldots \mid k\,A_n \mid f$$
$$\ldots$$

qual è il linguaggio generato dalla grammatica?
- È il linguaggio denotato dall'espressione regolare ottenuta come soluzione del sistema di equazioni

$$\begin{cases} A_1 = a\,A_1 + b\,A_2 + \ldots + w\,A_n + z \\ A_2 = b\,A_1 + a\,A_2 + \ldots + k\,A_n + f \\ \ldots \end{cases}$$

Proprietà operatori

- commutatività + $\quad A + B = B + A$
- associatività + $\quad (A+B)+C = A+(B+C)$
- associatività $\circ$ $\quad (AB)C = A(BC)$
- distributività $\quad (A+B)C = AC + BC$
- distributività $\quad A(B+C) = AB + AC$
- $A + \emptyset = A$
- $A\,\varepsilon = \varepsilon\,A = A$
- $A^* = A^*A^* = (A^*)^* = AA^* + \varepsilon$.

ESPRESSIONI vs LINGUAGGI REGOLARI

Per passare dalla grammatica all'espressione regolare si interpretano le produzioni comeequazioni 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: numeri naturali in base 2:

$S \rightarrow 0 \mid 1N$

$N \rightarrow 0 \mid 1 \mid 1N \mid 0N$

La grammatica può essere letta come un sistema di equazioni con
- due termini noti: 0, 1
- due incognite: L_S, L_N

in cui si sostituisce il simbolo| delle grammatiche con il simbolo+ delle espressioni regolari

$S \rightarrow 0 \mid 1 \mid 1N$

$N \rightarrow 0 \mid 1 \mid 1N \mid 0N$

$$\begin{cases} S = 0 + 1 + 1N \\ N = 0 + 1 + 1N + 0N \end{cases}$$

$$\begin{cases} S = 0 + 1 + 1N \\ N = (0 + 1) + (1 + 0)N \end{cases}$$

$$\begin{cases} S = 0 + 1 + 1N \\ N = (1 + 0)^* (0 + 1) \end{cases}$$

$$\begin{cases} S = 0+1+1(1+0)^*(0+1) \\ N = (1 + 0)^* (0 + 1) \end{cases}$$

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 sinistra

  e 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** $X = (u_1 + u_2 + \dots) X_1 + \dots + (z_1 + z_2 + \dots) X_n$
   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**
   $X = uX + \delta \quad \leftrightarrow \quad X = (u)^* \delta$
   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** $X \to \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ **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**
   $X = X u + \delta \quad \leftrightarrow \quad X = \delta (u)^*$
   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 → a B | a S B → d S | 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 2ª equazione nella 1ª e sviluppo dei relativi calcoli**	S = a* a (d S + b) = = a* a d S + a* a b
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 → a B | a S B → d S | b
Fase 2 • **se ora eliminiamo subito B, sostituendo la 2ª equazione nella 1ª 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

Una terza espressione (deterministica) equivalente:
S = a (d a + a)* b

Frasi 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

Linguaggi Regolari ⊆ Espressioni Regolari

- Data una grammatica regolare a destra, si riesce a generare una espressione regolare che denota lo stesso linguaggio
- Quindi i linguaggi generati da grammatiche regolari a destra sono un sottoinsieme dei linguaggi generati da espressioni regolari
- E data un'espressione regolare, si riesce a generare una grammatica?

Espressioni Regolari ⊆ Linguaggi Regolari

- Un'espressione regolare è definita induttivamente, quindi basta spiegare come scrivere una grammatica basandosi sulle operazioni regolari
- Stringa vuota: $S \rightarrow \varepsilon$
- carattere a: $S \rightarrow a$
- insieme vuoto: (nessuna produzione)

Espressioni Regolari ⊆ Linguaggi Regolari

- Se $A=<VN_A, VT_A, P_A, S_A>$ e $B=<VN_B, VT_B, P_B, S_B>$ sono due grammatiche che non hanno simboli nonterminali in comune

$$P_A = \{S_A \rightarrow aC \mid c \quad C \rightarrow bD \mid d \quad D \rightarrow wC \mid c\} \qquad P_B = \{S_B \rightarrow aK \mid c \quad K \rightarrow aF \mid c \quad F \rightarrow dK \mid a\}$$

- $L_A \cup L_B$: $U=<VN_U, VT_U, P_U, S'>$

$$P_U = P_A \cup P_B \cup \{S' \rightarrow S_A \mid S_B\}$$

Espressioni Regolari ⊆ Linguaggi Regolari

- Se $A=<VN_A,VT_A,P_A,S_A>$ e $B=<VN_B,VT_B,P_B,S_B>$ sono due grammatiche che non hanno simboli nonterminali in comune

P_A = {$S_A \rightarrow aC \mid c$
$C \rightarrow bD \mid d$
$D \rightarrow wC \mid c$}

P_B = {$S_B \rightarrow aK \mid c$
$K \rightarrow aF \mid c$
$F \rightarrow dK \mid a$}

- $L_A \circ L_B$: $C=<VN_C,VT_C,P_C,S_C>$

P_C = {$S_A \rightarrow aC \mid c\ S_B$
$C \rightarrow bD \mid d\ S_B$
$D \rightarrow wC \mid c\ S_B$} ∪ P_B

Espressioni Regolari ⊆ Linguaggi Regolari

- Se $A=<VN_A,VT_A,P_A,S_A>$

P_A = {$S_A \rightarrow aC \mid c$
$C \rightarrow bD \mid d$
$D \rightarrow wC \mid c$}

- $(L_A)^*$: $K=<VN_K,VT_K,P_K,S'>$

P_K = {$S_A \rightarrow aC \mid c\ S'$
$C \rightarrow bD \mid d\ S'$
$D \rightarrow wC \mid c\ S'$} ∪ {$S' \rightarrow S_A \mid \varepsilon$}

ESPRESSIONI E LINGUAGGI REGOLARI

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

Grammatiche ed espressioni regolari sono quindi *due rappresentazioni 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**

Riassumendo

- Le espressioni regolari sono un metodo molto usato per rappresentare linguaggi regolari
- Vengono definite induttivamente tramite composizione di operazioni sui linguaggi
- È possibile passare da espressione regolare ad una grammatica regolare (a destra o a sinistra) e viceversa