

# Corso di Linguaggi di Programmazione A.A. 2017-18 Docente: Pasquale Lops

## Presentazione

# Informazioni generali

- Corso: Linguaggi di Programmazione
- Docente: Lops Pasquale
- Email: [pasquale.lops@uniba.it](mailto:pasquale.lops@uniba.it)
- Lezioni: 26 Febbraio 2018 - 1 Giugno 2018
  - Martedì 11.30-14.30
  - Giovedì 8.30-11.30
  - Venerdì 11.30-13.30
- Piattaforma ADA: <http://elearning.di.uniba.it>  
Contiene *informazioni sul corso*, materiale didattico, esercitazioni, *tracce d'esame*, *avvisi*, ...
  - Codice iscrizione corso: **TFOLAB1718LOPS**

# Distribuzione CFU

- L'insegnamento prevede 9 CFU
  - 7 crediti T1 = 56 ore di lezione frontale + 119 ore di studio individuale
  - 2 crediti T2 = 30 ore di esercitazione / laboratorio + 20 di rielaborazione personale
  - 86 ore in aula + 139 ore di studio individuale
  - Totale impegno = 225 ore

# Programma preliminare

- Prerequisiti
  - Matematica discreta
  - Programmazione imperativa
  - Conoscenza del linguaggio C
- Strutturazione del corso
  - Prima parte:
    - Si illustrano gli aspetti più significativi dei linguaggi di programmazione, la loro evoluzione ed i concetti che stanno alla base della traduzione dei linguaggi di alto livello
    - Si forniscono i concetti più significativi della teoria dei linguaggi formali, enfatizzando gli aspetti generativi e riconoscitivi dei linguaggi formali
  - Seconda parte:
    - Studio del processo di compilazione
    - Si presentano/sviluppano programmi per la manipolazione di grammatiche e implementazione automi

# Programma preliminare

- Parte I: Linguaggi di Programmazione
  - Linguaggi di programmazione. Gerarchia di linguaggi di programmazione e di macchine astratte: linguaggi macchina, linguaggi assembler, linguaggi di alto livello. Linguaggi di alto livello: interpretazione e compilazione, sintassi, semantica, analisi di programmi.
  - Linguaggi formali. Inquadramento della teoria dei linguaggi formali nell'informatica teorica. Classificazione di Chomsky. Operazioni sui linguaggi e proprietà di chiusura delle classi di linguaggi rispetto alle operazioni. Generazione di linguaggi: un'introduzione alle grammatiche. Carte sintattiche e BNF, grammatiche generative, derivazione, linguaggio generato da una grammatica, equivalenza tra grammatiche, relazione tra grammatiche e linguaggi, non determinismo, correttezza di una grammatica: indecidibilità del problema

# Programma preliminare

- Parte II: Linguaggi di Programmazione e Compilatori
  - I compilatori. Il modello di un compilatore: analizzatore lessicale, analizzatore sintattico, analizzatore semantico, generazione e ottimizzazione del codice
  - Analisi lessicale. Linguaggi regolari, espressioni regolari, automi e linguaggi a stati finiti. Teoremi di equivalenza
  - Analisi sintattica. Linguaggi liberi da contesto

# Programma preliminare

- Parte II: Linguaggi di Programmazione e Compilatori
  - Tabella dei simboli (TS)
  - Gestione della memoria
  - Cenni su evoluzione dei linguaggi e paradigmi di programmazione

# Programma preliminare

## ■ Parte II: Esercitazioni

- Esercitazioni sulla teoria dei linguaggi formali
- Realizzazione di programmi per manipolare le grammatiche e/o implementazione automi

# Obiettivi formativi

## Quali competenze acquisirò

- Capacità di ricondurre un problema al riconoscimento di un linguaggio formale
  - Riconoscere un IBAN, una targa automobilistica, un identificatore
- Capacità di riconoscere il tipo di un linguaggio
  - Se so classificare un linguaggio, so anche come riconoscerlo
- Comprensione dei meccanismi alla base dei linguaggi di programmazione
  - Capacità di comprendere com'è gestita la memoria, come sono implementate le regole di visibilità, etc.
- Comprensione dei meccanismi alla base del processo di compilazione
  - Capacità di comprendere come funziona un compilatore, quali tecniche di analisi dei programmi adotta per segnalare gli errori

# Obiettivi professionalizzanti

Quanto mi servirà ciò che studierò in questo insegnamento nella mia vita di informatico

- Capacità di apprendere velocemente un nuovo linguaggio di programmazione
- Capacità di acquisire nuovi paradigmi di programmazione oltre a quello imperativo
- Conoscenza delle tecniche di analisi e traduzione dei linguaggi di programmazione
  - Un informatico non può usare un compilatore come una scatola nera!
- Conoscenza delle espressioni regolari, degli automi e delle grammatiche
  - Capacità di descrivere, riconoscere, generare un linguaggio

# Prove d'esame

- La prova d'esame è scritta
  - esecuzione di esercizi sulla teoria dei linguaggi formali
  - enunciazione di definizioni
  - dimostrazione di teoremi della teoria dei linguaggi formali
  - quesiti su linguaggi di programmazione e compilatori
- Una prova scritta intermedia
  - durante l'interruzione prevista da manifesto
  - stessi argomenti delle prove d'appello

# Organizzazione e valutazione prove d'esame

## ■ Prove scritte d'appello

- Già calendarizzate e visibili su ESSE3
- Valutazione in trentesimi
- Si supera con una valutazione minima di 18

## ■ Prova intermedia

- Valutazione in trentesimi
- Si supera con un minimo di 16
- Esonera dallo svolgimento di una parte della prova finale,  
**LIMITATAMENTE AL PRIMO APPELLO DI GIUGNO**
- Valutazione COMPLESSIVA = media delle valutazioni delle due prove (itinere + appello giugno)
- IF (valutazione COMPLESSIVA  $\geq$  18) THEN esame\_superato

# Validità delle prove

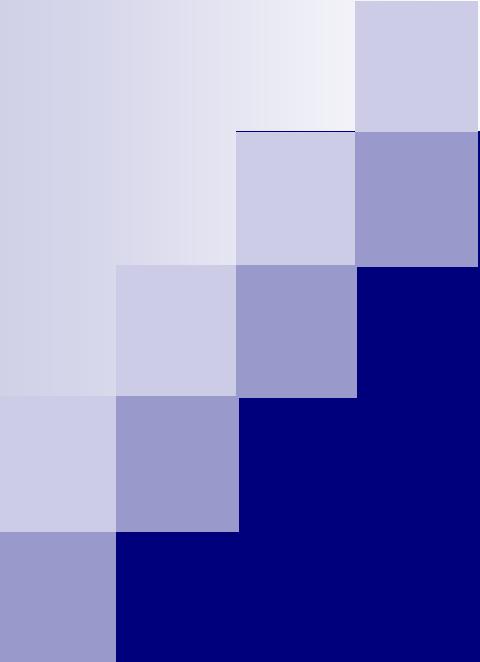
- Per sostenere una prova occorre prenotarsi OBBLIGATORIAMENTE mediante il sistema ESSE3
- Prova scritta d'appello
  - valida solo per l'appello sostenuto
  - non si può posticipare la verbalizzazione in un appello successivo
- Prova intermedia
  - Prenotazione tramite ESSE3
  - Valida fino a giugno

# Materiale Didattico

- *Informazioni sul corso, avvisi, esami, tracce delle prove d'esame, trasparenze usate a lezione*
  - <http://elearning.di.uniba.it>
- Testi di Riferimento
  - **G. Semeraro, Elementi di teoria dei linguaggi formali, ilmiolibro.it (2017) (\*)**
  - **Maurizio Gabbrielli, Simone Martini**  
**Linguaggi di Programmazione, Principi e paradigmi. Seconda Edizione McGraw-Hill.**
  - Libera scelta riguardo a manuali di C
  - Attenzione: le dispense non sostituiscono i libri di testo!



(\*) <http://ilmiolibro.kataweb.it/libro/informatica-e-internet/317883/elementi-di-teoria-dei-linguaggi-formali/>



# Linguaggi di Programmazione

## Capitolo 1 – Introduzione ai linguaggi di programmazione ed alla teoria dei linguaggi formali

Si ringrazia il Dott. Marco de Gemmis per la concessione di parte del  
materiale didattico di base

# Apprendimento di un linguaggio di programmazione

- Perché un insegnamento “Linguaggi di Programmazione” oltre a quello di “Programmazione”?

# Apprendimento di un linguaggio di programmazione

- Una delle competenze fondamentali di un *buon* informatico è quella di apprendere un nuovo linguaggio di programmazione con naturalezza e velocità
- Questa competenza non la si acquisisce soltanto imparando *ex novo* molti linguaggi diversi
  - conoscere tanti linguaggi di programmazione oppure conoscere i fondamenti che sono alla base dei linguaggi di programmazione?

# Apprendimento di un linguaggio di programmazione

- Lingue “naturali”
  - esistono analogie, somiglianze derivanti dalla genealogia
  - Es.: italiano – rumeno
    - cioccolata = ciocolata
    - insalata = salata
    - treno = tren
- Linguaggi di programmazione
  - è difficile conoscerne un gran numero in modo approfondito
  - è possibile conoscerne a fondo i meccanismi che ne ispirano il *progetto* e l'*implementazione*

Lo studio degli aspetti generali dei linguaggi di programmazione è un passaggio chiave della formazione universitaria e professionale di un informatico

# Macchine Astratte e Implementazione dei linguaggi di programmazione

- Qual è il significato dell'espressione:  
“**implementazione di un linguaggio di programmazione**”?
- Si tratta di un concetto strettamente correlato a quello di:

**MACCHINA ASTRATTA**

# Macchine Astratte e Implementazione dei linguaggi di programmazione

- Calcolatore = macchina fisica
  - consente di eseguire algoritmi opportunamente formalizzati perché siano “comprensibili” all’esecutore
  - la formalizzazione consiste nella codifica degli algoritmi in un certo linguaggio  $\mathcal{L}$  definito da una specifica sintassi
  - la sintassi di  $\mathcal{L}$  permette di utilizzare determinati costrutti per comporre programmi in  $\mathcal{L}$
  - Un programma in  $\mathcal{L}$  è una sequenza di istruzioni del linguaggio  $\mathcal{L}$
- Macchina astratta = astrazione del concetto di calcolatore fisico

# Definizioni di macchina astratta e linguaggio macchina

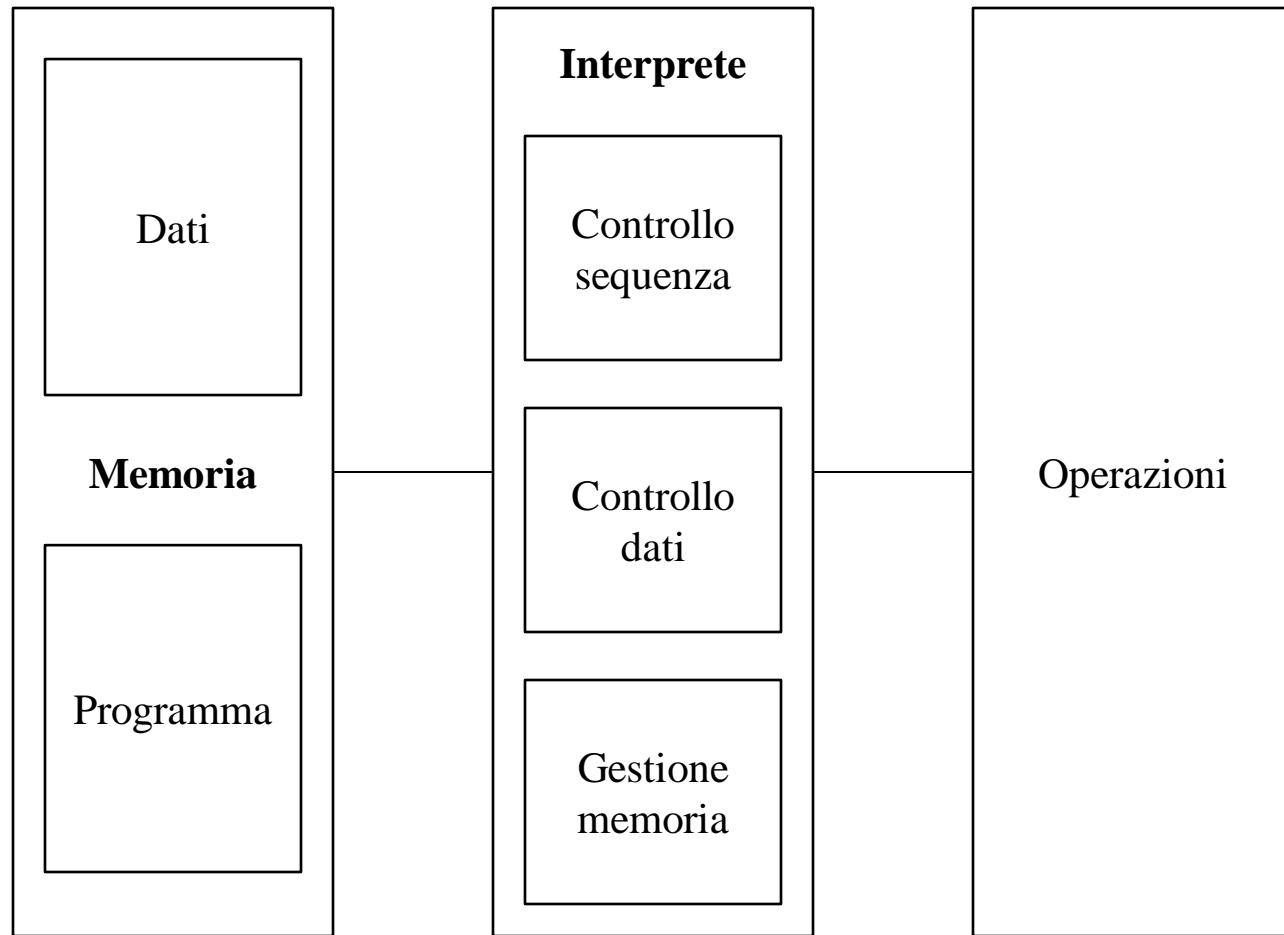
## ■ Macchina astratta per $\mathcal{L}$

- Un insieme di algoritmi e strutture dati che permettono di memorizzare ed eseguire programmi scritti in  $\mathcal{L}$
- Si denota con  $\mathcal{M}_{\mathcal{L}}$
- È composta da:
  - una memoria per immagazzinare dati e programmi
  - un interprete che esegue le istruzioni contenute nei programmi

## ■ Linguaggio Macchina

- Data una macchina astratta  $\mathcal{M}_{\mathcal{L}}$ , il linguaggio  $\mathcal{L}$  “compreso” dall’interprete di  $\mathcal{M}_{\mathcal{L}}$  è detto linguaggio macchina di  $\mathcal{M}_{\mathcal{L}}$

# Macchina Astratta



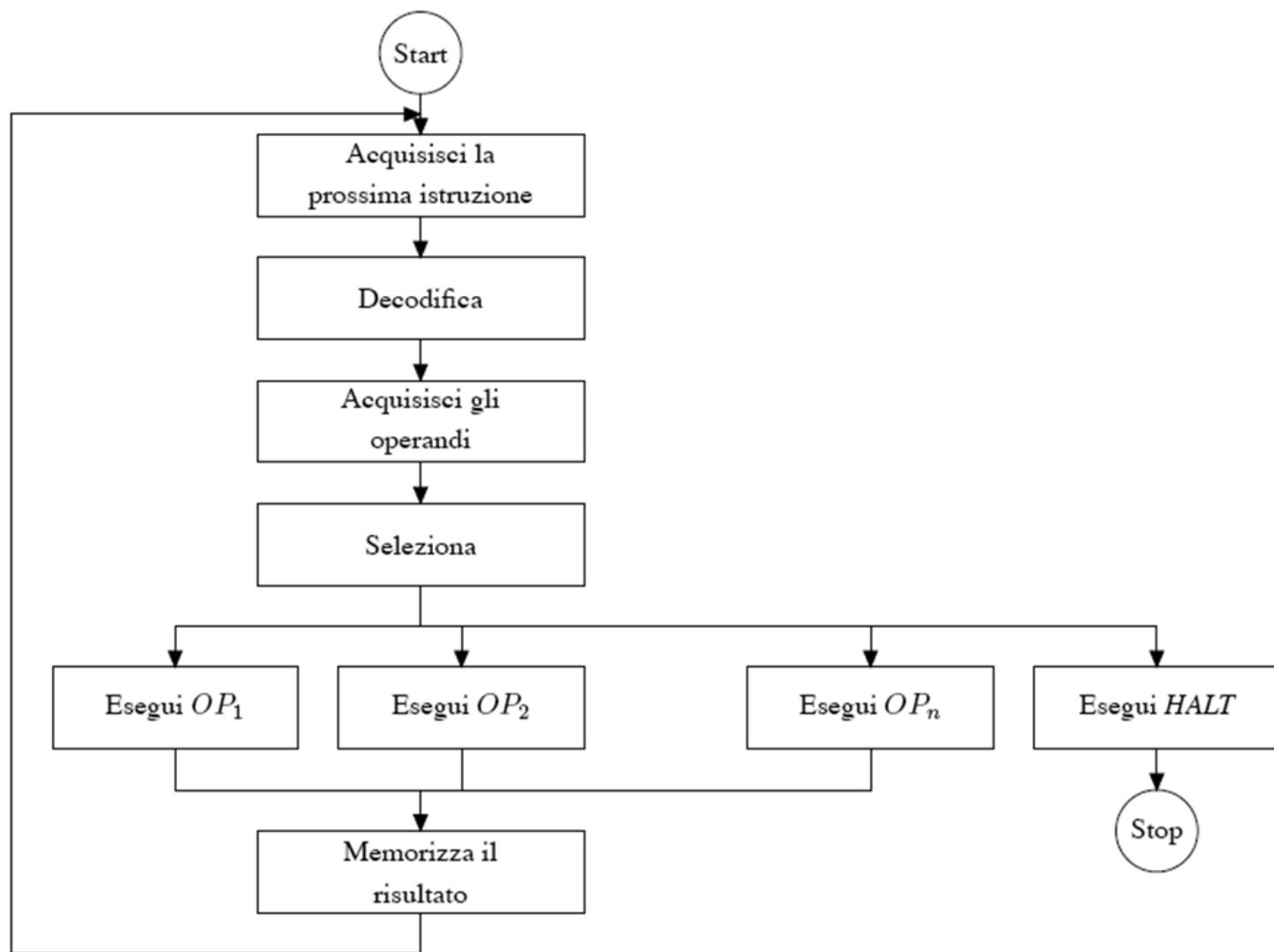
# Le operazioni dell'Interprete

- Operazioni per l'elaborazione dei dati primitivi
  - numeri interi, reali
  - operazioni aritmetiche
- Operazioni e strutture dati per il controllo della sequenza di esecuzione
  - servono per gestire il flusso di controllo delle istruzioni
  - strutture dati per memorizzare l'indirizzo della prossima istruzione
  - operazioni per manipolare le strutture dati
    - es.: calcolo dell'indirizzo della prossima istruzione

# Le operazioni dell'Interprete

- Operazioni e strutture dati per il controllo del trasferimento dei dati
  - gestiscono il trasferimento dei dati dalla memoria all'interprete e viceversa
    - es.: recupero degli operandi
  - possono far uso di strutture dati ausiliarie
    - es.: pila
- Operazioni e strutture dati per la gestione della memoria
  - relative all'allocazione di memoria per dati e programmi

# Ciclo di esecuzione del generico interprete



# Realizzazione di una macchina astratta

- Per essere effettivamente utilizzata,  $\mathcal{M}_{\mathcal{L}}$  dovrà prima o poi utilizzare qualche dispositivo fisico
  - realizzazione “fisica” in hardware
    - algoritmi di  $\mathcal{M}_{\mathcal{L}}$  realizzati direttamente mediante dispositivi fisici
- Possiamo pensare a realizzazioni che usino livelli intermedi tra  $\mathcal{M}_{\mathcal{L}}$  ed il dispositivo fisico
  - **simulazione mediante software**
  - emulazione mediante firmware
    - microprogrammi in linguaggi di basso livello

# Realizzazione mediante software

- Algoritmi e strutture dati di  $\mathcal{M}_{\mathcal{L}}$  realizzati in un altro linguaggio  $\mathcal{L}'$  già implementato
- $\mathcal{M}_{\mathcal{L}}$  è realizzata mediante programmi in  $\mathcal{L}'$  che simulano le funzionalità di  $\mathcal{M}_{\mathcal{L}}$
- $\mathcal{M}_{\mathcal{L}}$  è realizzata attraverso la macchina  $\mathcal{M}'_{\mathcal{L}'}$
- $\mathcal{M}'_{\mathcal{L}'}$  è detta *macchina ospite* e si denota con  $\mathcal{M}_{\mathcal{L}o}$
- Intuitivamente l'implementazione di  $\mathcal{L}$  sulla macchina ospite avviene mediante una qualche “*traduzione*” di  $\mathcal{L}$  in  $\mathcal{L}o$
- A seconda di come avvenga la traduzione di parla di:
  - implementazione interpretativa
  - implementazione compilativa

# Funzioni Parziali

- Una funzione *parziale*

$$f: A \rightarrow B$$

è una corrispondenza tra elementi dell'insieme A e quelli dell'insieme B

- Può essere NON DEFINITA per qualche elemento di A
- Dato  $a \in A$ , **se** esiste un corrispondente in B, esso è denotato con  $f(a)$

# I programmi definiscono funzioni parziali

```
read(x);  
if x==1 then print(x) else  
    while (x<>1) do skip;
```

$$f(x) = \begin{cases} 1 & \text{se } x=1 \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

# Definizione di interprete

In generale un programma scritto in  $\mathcal{L}$  si può vedere come una funzione parziale:

$$\mathcal{P}^{\mathcal{L}} : \mathcal{D} \rightarrow \mathcal{D} \quad \text{t.c. } \mathcal{P}^{\mathcal{L}}(\text{Input}) = \text{Output}$$

Possiamo dare la seguente definizione di interprete di  $\mathcal{L}$  in  $\mathcal{Lo}$ :

$$I_{\mathcal{L}}^{\mathcal{Lo}} : (\text{Prog}^{\mathcal{L}} \times \mathcal{D}) \rightarrow \mathcal{D} \quad \text{ta le che}$$

$$I_{\mathcal{L}}^{\mathcal{Lo}}(\mathcal{P}^{\mathcal{L}}, \text{Input}) = \mathcal{P}^{\mathcal{L}}(\text{Input})$$

Non vi è traduzione esplicita dei programmi scritti in  $\mathcal{L}$ , ma solo un procedimento di decodifica



L'interprete, per eseguire un'istruzione  $i$  di  $\mathcal{L}$ , le fa corrispondere un insieme di istruzioni di  $\mathcal{Lo}$ . Tale decodifica non è una traduzione esplicita poiché il codice corrispondente a  $i$  è eseguito direttamente e non prodotto in uscita

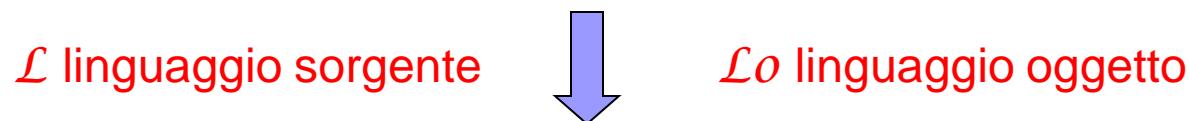
# Definizione di compilatore

Un compilatore da  $\mathcal{L}$  a  $\mathcal{L}o$  è un programma che realizza la funzione:

$$C_{\mathcal{L}, \mathcal{L}o} : \text{Prog } \mathcal{L} \rightarrow \text{Prog } \mathcal{L}o$$

$$C_{\mathcal{L}, \mathcal{L}o}(\mathcal{P}^{\mathcal{L}}) = \mathcal{P}C^{\mathcal{L}o} \quad \mathcal{P}^{\mathcal{L}}(\text{Input}) = \mathcal{P}C^{\mathcal{L}o}(\text{Input})$$

Traduzione esplicita dei programmi scritti in  $\mathcal{L}$  in programmi scritti in  $\mathcal{L}o$



Per eseguire  $\mathcal{P}^{\mathcal{L}}$  su  $\text{Input}$ , bisogna eseguire  $C_{\mathcal{L}, \mathcal{L}o}$  con  $\mathcal{P}^{\mathcal{L}}$  come input. Si avrà come risultato un programma compilato  $\mathcal{P}C^{\mathcal{L}o}$  scritto in  $\mathcal{L}o$ , che sarà eseguito su  $\mathcal{M}o_{\mathcal{L}o}$  con il dato in ingresso  $\text{Input}$

# Interpretazione vs. Compilazione

## ■ Interpretazione

👎 scarsa efficienza

- decodifica costrutti a tempo di esecuzione
- decodifica di uno stesso comando ripetuto

👍 maggiore flessibilità

- debugging più semplice

## ■ Compilazione

👍 maggiore efficienza

- decodifica di un'istruzione fatta una sola volta indipendentemente da quante volte è eseguita

👎 minore flessibilità

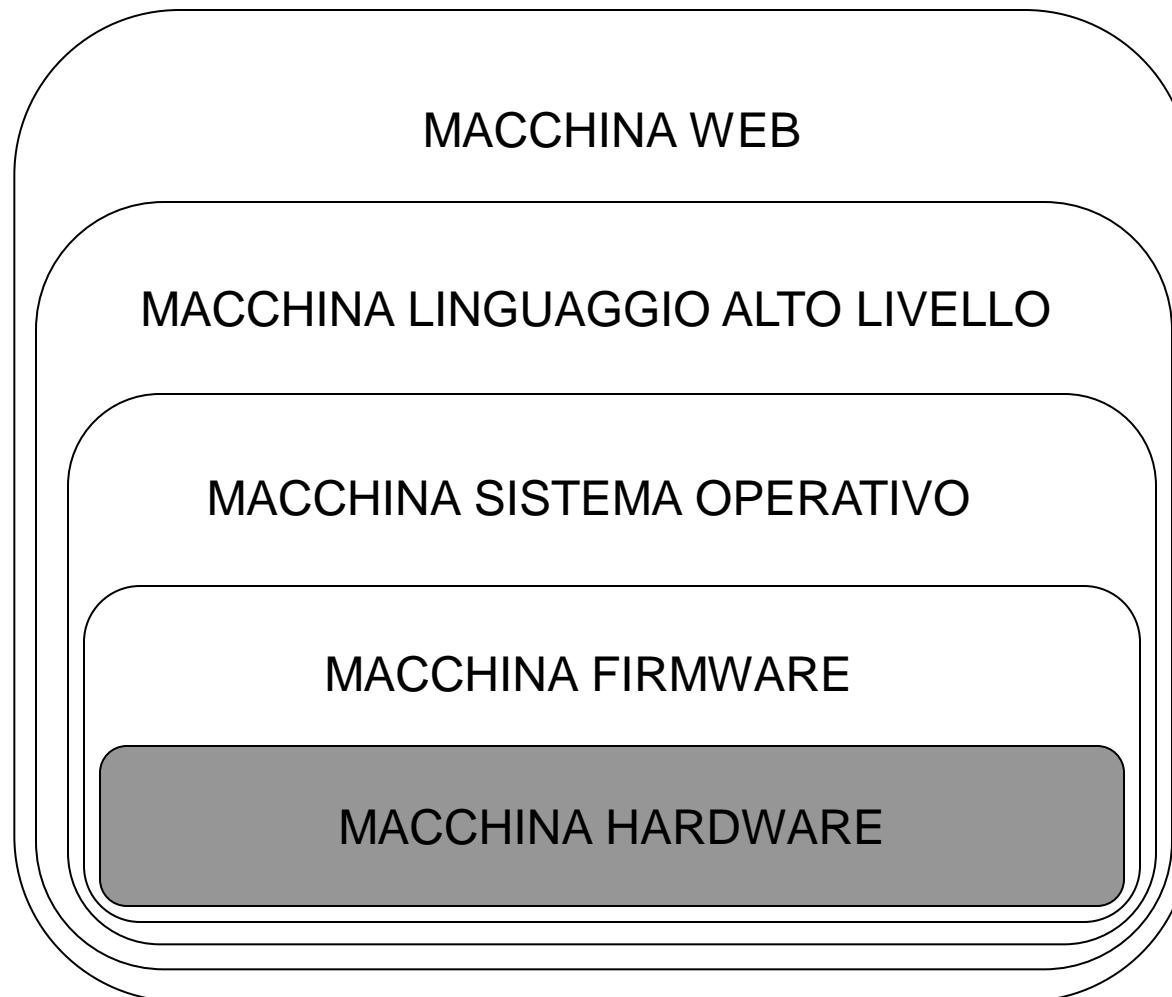
- perdita di informazioni rispetto al programma sorgente (difficile tracciare errori e dunque maggiori difficoltà nel debugging)

In realtà i due approcci sono spesso combinati

# Gerarchie di Macchine Astratte

- Possiamo pensare ad una gerarchia di macchine astratte  $\mathcal{M}_{\mathcal{L}0}, \mathcal{M}_{\mathcal{L}1}, \dots \mathcal{M}_{\mathcal{Ln}}$
- $\mathcal{M}_{\mathcal{Li}}$  è implementata sfruttando il linguaggio della macchina sottostante  $\mathcal{M}_{\mathcal{Li-1}}$
- $\mathcal{M}_{\mathcal{Li}}$  fornisce a sua volta il proprio linguaggio alla macchina sovrastante  $\mathcal{M}_{\mathcal{Li+1}}$
- Indipendenza tra livelli
  - modifiche *interne* alle funzionalità di un livello non hanno influenza sugli altri livelli

# Una gerarchia di macchine astratte



# Linguaggi di Programmazione, Problemi, Interpreti

- Linguaggio di programmazione = formalismo per portare al livello di macchina fisica l'algoritmo solutivo  $A_p$  per un certo problema  $P$

## IN GENERALE



Deve essere in grado di **interpretare** la  
descrizione del metodo solutivo

# Programmi e funzioni calcolabili

- Esistono dei problemi per i quali non esiste un programma che li risolva?
- La risposta dipende dal linguaggio di programmazione?

# Funzioni (Parziali) Calcolabili

- Una funzione parziale  $f: A \rightarrow B$  è calcolabile nel linguaggio  $\mathcal{L}$  se esiste un programma  $P$  scritto in  $\mathcal{L}$  tale che:
  - Se  $f(a) = b$  allora  $P(a)$  termina e produce come output  $b$
  - Se  $f(a)$  non è definita allora  $P$  con input  $a$  va in ciclo
- Esistono dei problemi per i quali non esiste un programma che li risolva? → Esistono funzioni **non calcolabili**?

# Il problema della fermata

- Vogliamo stabilire se un programma termina su un dato input
  - Programma di *debugging* H
- H riceve in ingresso un qualsiasi programma P scritto nel linguaggio  $\mathcal{L}$  ed un generico input x per tale programma:

```
Boolean H(P,x)
    boolean term;
    if (P(x) termina) then term=true;
        else term=false;
    return term;
```

H( $P, x$ ) ritorna

{  
    true se  $P(x)$  termina  
  
    false se  $P(x)$  va in loop

# Il problema della fermata

- Possiamo costruire un altro programma  $K$  scritto in  $\mathcal{L}$  che prenda in input un programma  $P$  (scritto sempre in  $\mathcal{L}$ )
  - Il programma  $K$  sfrutta  $H$  per decidere sulla terminazione di  $P$

$K(P)$

```
if (H(P,P)=false) then print("LOOP");  
else while (true) do print("TERMINA");
```

$K(P)$  {

Stampa “LOOP” se  $P(P)$  NON termina  
va in loop se  $P(P)$  termina

- In pratica la terminazione di  $K$  è opposta rispetto a quella del suo input

# Il problema della fermata

- Cosa accade se diamo in input a K il suo stesso testo?

$K(K)$  { Stampa “LOOP” se  $K(K)$  NON termina  
va in loop se  $K(K)$  termina

- Assurdo!
  - $K(K)$  termina con una stampa quando  $K(K)$  non termina
  - $K(K)$  non termina quando  $K(K)$  termina
  - Assurdo conseguente dall'aver supposto l'esistenza del programma H

# Indecidibilità

- Il problema della terminazione è *indecidibile*
  - Non possiamo costruire (e formalizzare) un algoritmo che lo risolva
- Esistono dei problemi per i quali non esiste un programma che li risolva? → Esistono funzioni **non calcolabili**?
- Sì → il problema della terminazione è una di queste
- Questo risultato dipende dal linguaggio  $\mathcal{L}$ ?
  - ovvero dipende dal formalismo usato per descrivere l'algoritmo?

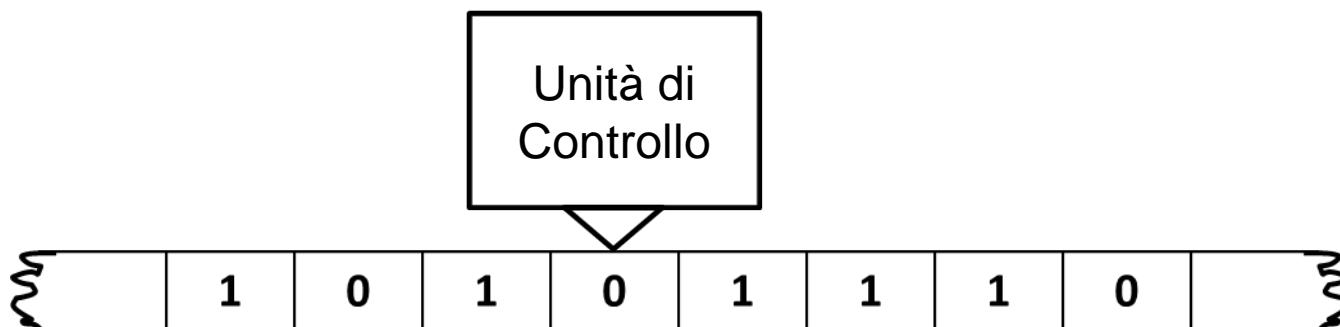
# Macchine di Turing (MdT)

- Modello matematico di computazione introdotto negli anni '30
- Primo formalismo (=linguaggio) con il quale è stata dimostrata l'indecidibilità del problema della fermata



# Macchine di Turing (MdT)

- Nastro potenzialmente infinito diviso in celle (memoria)
  - *ogni cella contiene un simbolo preso da un alfabeto finito*
- Testina di lettura/scrittura
  - può leggere/scrivere in una cella per volta
  - può spostarsi a destra o a sinistra di una cella per volta
  - gestita da un controllo espresso da un numero finito di stati
- Unità di controllo
  - decodifica ed esegue comandi rivolti alla testina



# Macchine di Turing (MdT)

- L'unità di controllo esegue un programma P sui dati memorizzati sul nastro
- Le istruzioni di P sono del tipo:  
< simbolo\_letto,  
    stato\_corrente,  
    simbolo\_da\_scrivere,  
    sinistra/destra,  
    nuovo\_stato >

# MdT: il modello matematico

Una MdT è definita da una quintupla:

$$M = (X, Q, fm, fd, \delta)$$

- $X$  = insieme finito di simboli
  - comprende il *blank* ovvero *cella vuota*
- $Q$  = insieme finito di stati
  - comprende *HALT* che definisce la terminazione

# MdT: il modello matematico

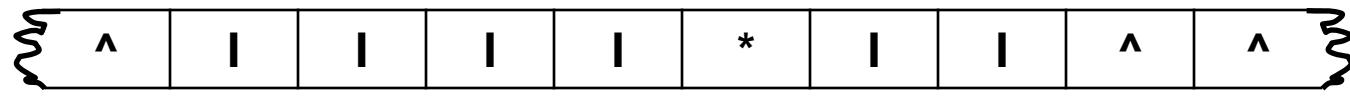
Una MdT è definita da una quintupla:

$$M = (X, Q, fm, fd, \delta)$$

- Funzione di macchina  $fm: Q \times X \rightarrow X$ 
  - Determina il simbolo da scrivere sul nastro
- Funzione di direzione  $fd: Q \times X \rightarrow \{S, D, F\}$ 
  - Determina lo spostamento della testina
  - S=sinistra, D=destra, F=ferma
- Funzione di transizione di stato  $\delta: Q \times X \rightarrow Q$ 
  - Definisce lo stato successivo della computazione

# Scrivere algoritmi per MdT: nastro

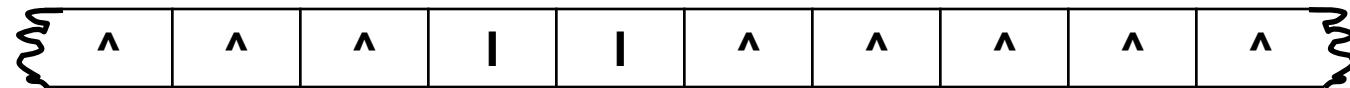
- Definire un'opportuna configurazione iniziale del nastro
  - Codificare i dati
  - Es.: nastro iniziale per problema della sottrazione tra interi



$$4 - 2$$

operandi codificati con '1' e separati da \*  
blank=^

- Definire un'opportuna configurazione finale del nastro che rappresenti la soluzione



# Scrivere algoritmi per MdT: il controllo

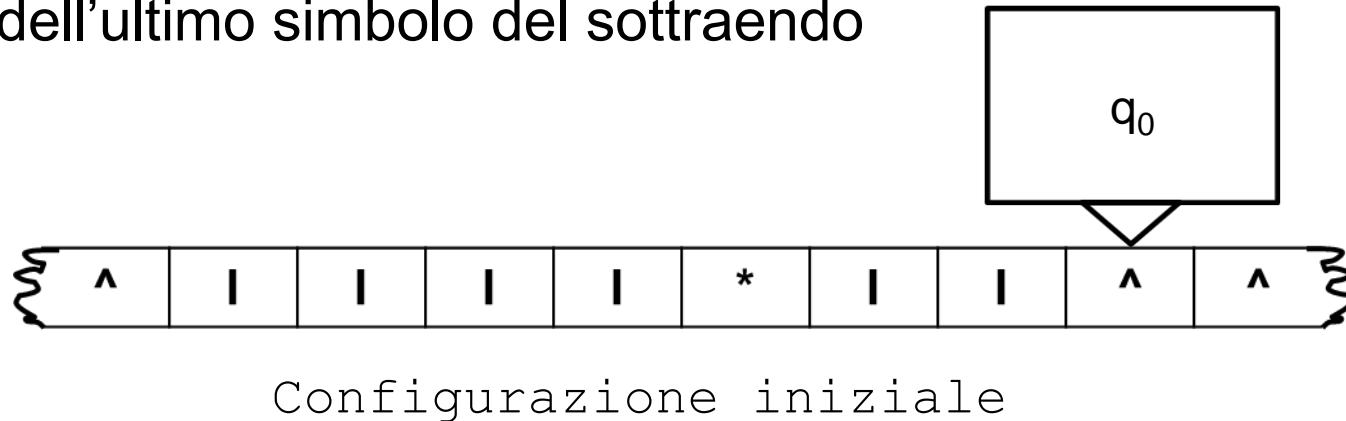
- Definire le funzioni  $fm$ ,  $fd$ ,  $\delta$ 
  - in modo da trasformare la configurazione iniziale in quella che rappresenta la soluzione
- In pratica il programma per una MdT è una sequenza di quintuple:

$$\langle x_i \in X, q_j \in Q, x_{ij} \in X, \{S, D, F\}, q_{ij} \in Q \rangle$$

- In corrispondenza di un simbolo letto  $x_i$  e dello stato  $q_j$  in cui si trova l'unità di controllo, si determinano:
  - Il simbolo  $x_{ij} = fm(x_i, q_j)$  da scrivere nella cella corrente
  - Lo spostamento della testina  $fd(x_i, q_j)$
  - Il nuovo stato  $q_{ij} = \delta(x_i, q_j)$  in cui MdT continuerà la computazione

# Scrivere algoritmi per MdT: sottrazione tra interi

- Progettiamo un algoritmo per eseguire la sottrazione tra due numeri interi  $n$  e  $m$ ,  $n \geq 0$ ,  $m \geq 0$ 
  - Per semplicità assumiamo che  $n \geq m$
  - La testina è posizionata sulla prima cella vuota a destra dell'ultimo simbolo del sottraendo



- Il modello di calcolo ci "obbliga" a pensare l'algoritmo in base alle operazioni possibili

# Un possibile algoritmo per calcolare $n-m$



Cancellare ugual numero di simboli dal minuendo  $n$  e dal sottraendo  $m$  in modo che sul nastro resti solo il risultato finale

1. Diminuisci di una unità  $m$ 
  - ricorda di aver cancellato un simbolo da  $m$
2. Spostati a S in cerca del primo simbolo di  $n$
3. Cancellalo
  - Ricorda che ora entrambi gli operandi sono stati diminuiti di una unità
4. Spostati a D in cerca dell'ultimo simbolo di  $m$ 
  - Se non ci sono più simboli da cancellare da  $m$  allora cancella il separatore → HALT
  - In caso contrario torna al punto 1

# Le 5-ple di una MdT per la sottrazione

$$X = \{I, *, ^\wedge\}$$

$$Q = \{q_0, q_1, q_2, q_3, \text{HALT}\}$$

$q_0 \equiv$  stato iniziale della computazione ovvero  
ricerca ultimo simbolo di  $m$

$q_1 \equiv$  diminuito  $m$

$q_2 \equiv$  raggiunto simbolo iniziale di  $n$

$q_3 \equiv$  diminuiti entrambi operandi

$\langle ^\wedge, q_0, ^\wedge, S, q_0 \rangle$

$\langle |, q_0, ^\wedge, S, q_1 \rangle$

$\langle *, q_0, ^\wedge, F, \text{HALT} \rangle$

$\langle ^\wedge, q_1, ^\wedge, D, q_2 \rangle$

$\langle |, q_1, |, S, q_1 \rangle$

$\langle *, q_1, *, S, q_1 \rangle$

$\langle |, q_2, ^\wedge, D, q_3 \rangle$

$\langle ^\wedge, q_3, ^\wedge, S, q_0 \rangle$

$\langle |, q_3, |, D, q_3 \rangle$

$\langle *, q_3, *, D, q_3 \rangle$

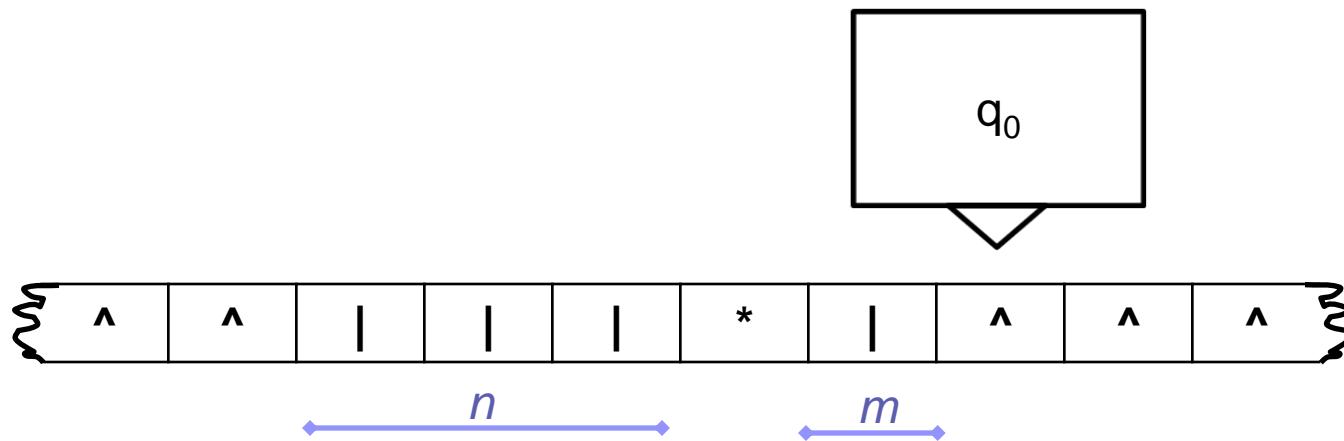
# La Matrice Funzionale

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$

Perché alcune transizioni non sono definite?

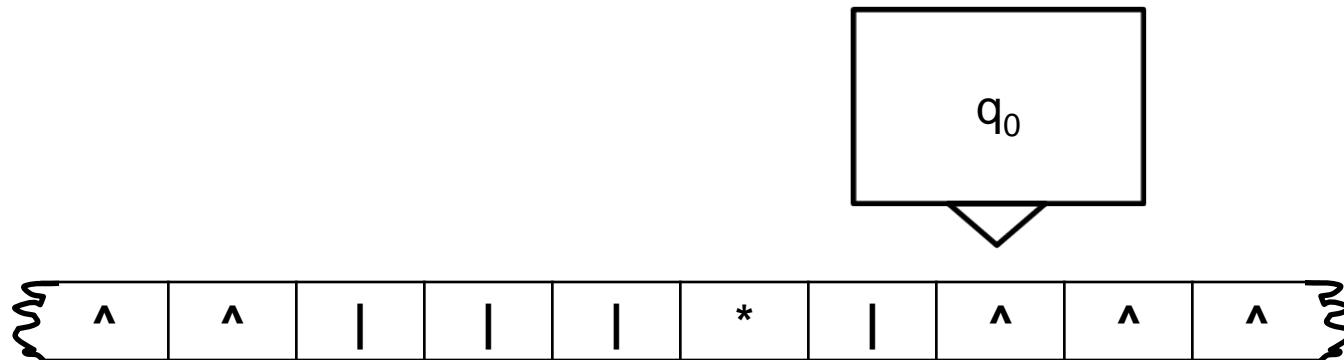
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
$ $	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
$*$	$\wedge F \text{ HALT}$	$* Sq_1$		$* Dq_3$



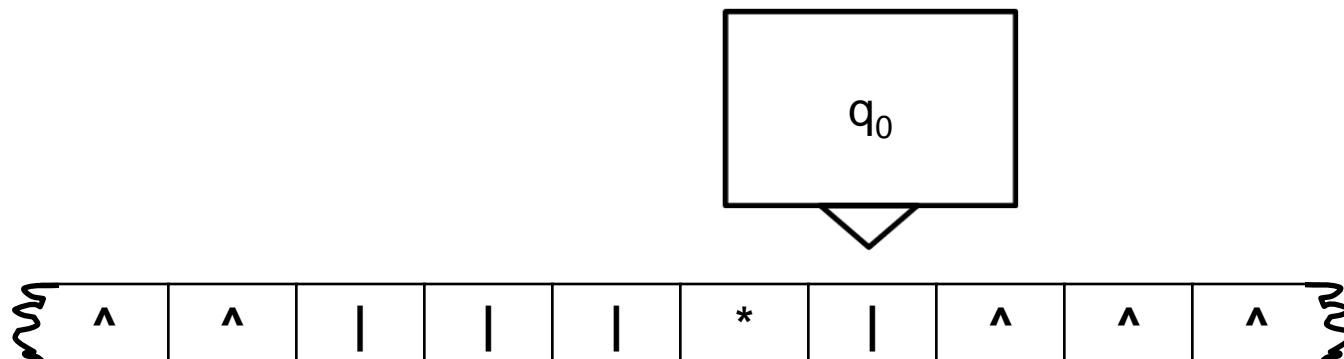
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



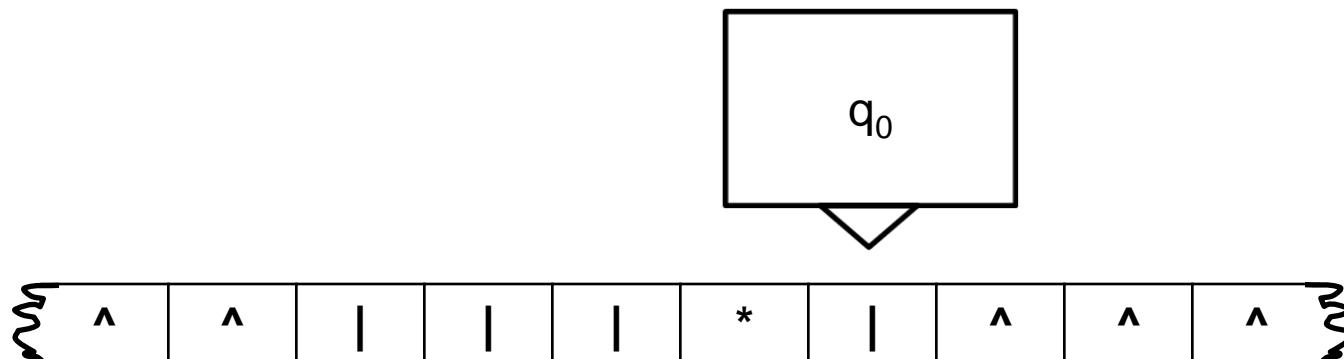
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



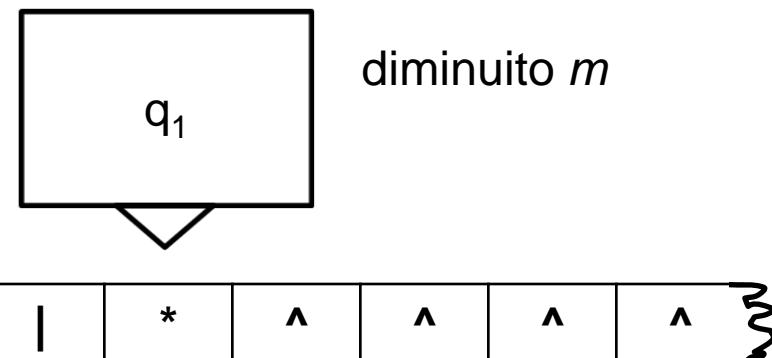
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



# Computazione 3-1

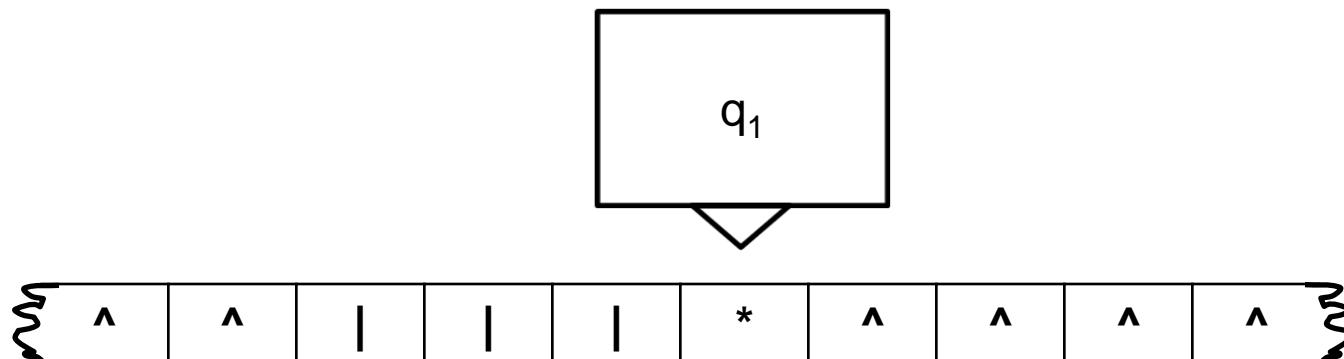
$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



diminuito  $m$

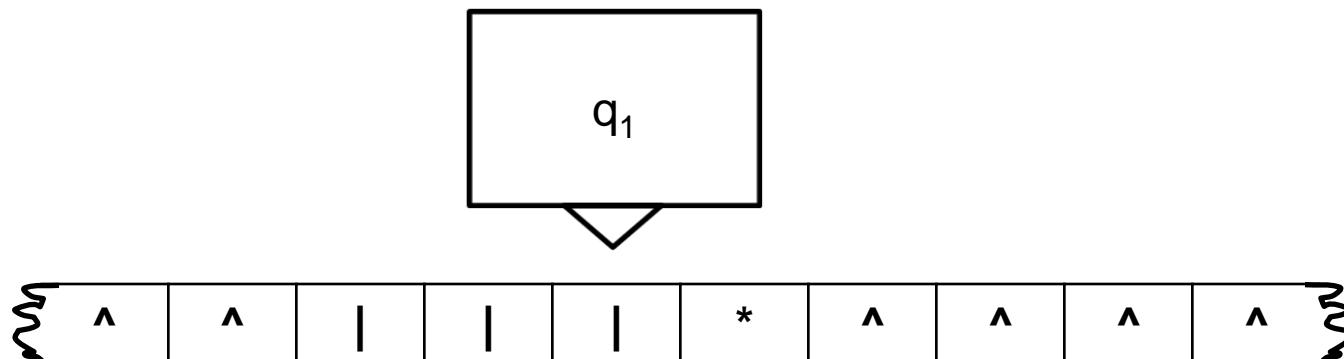
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



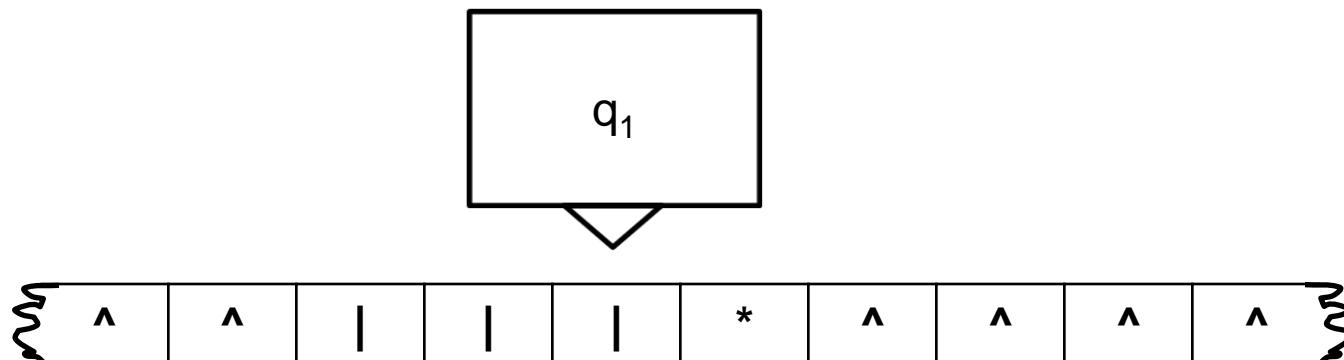
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



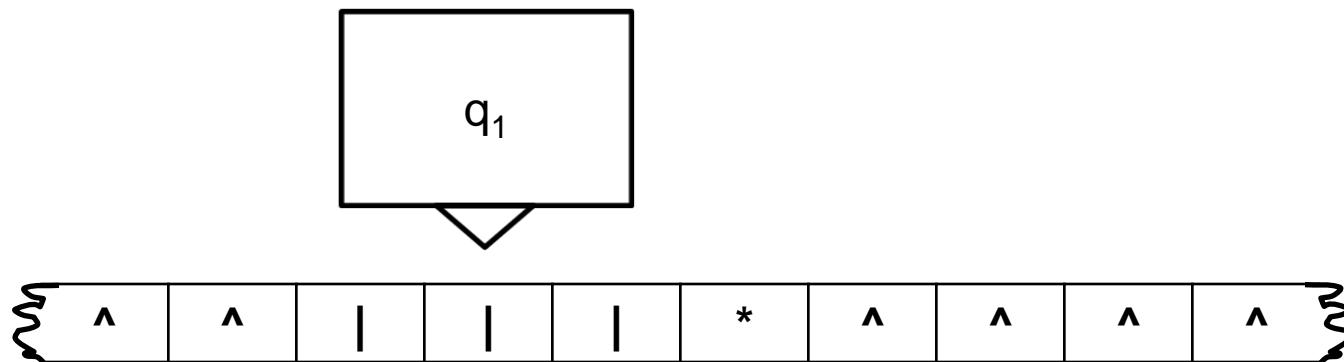
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



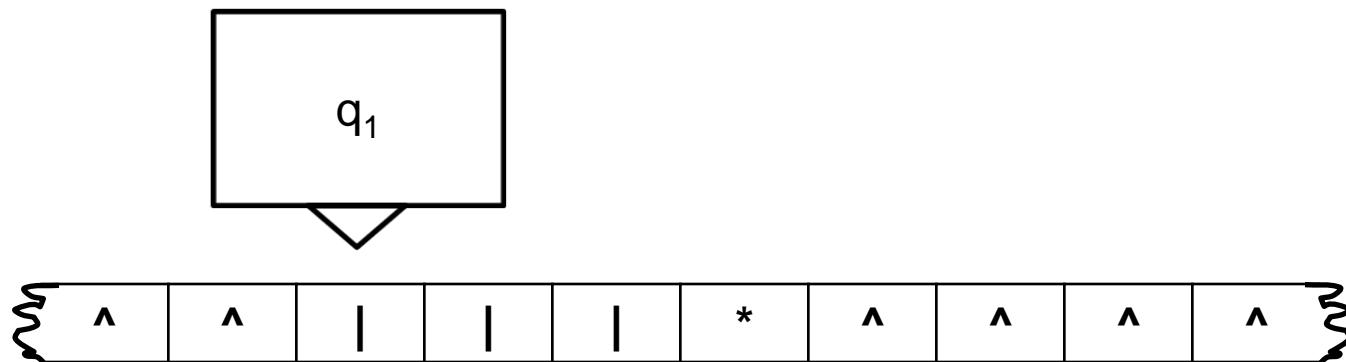
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



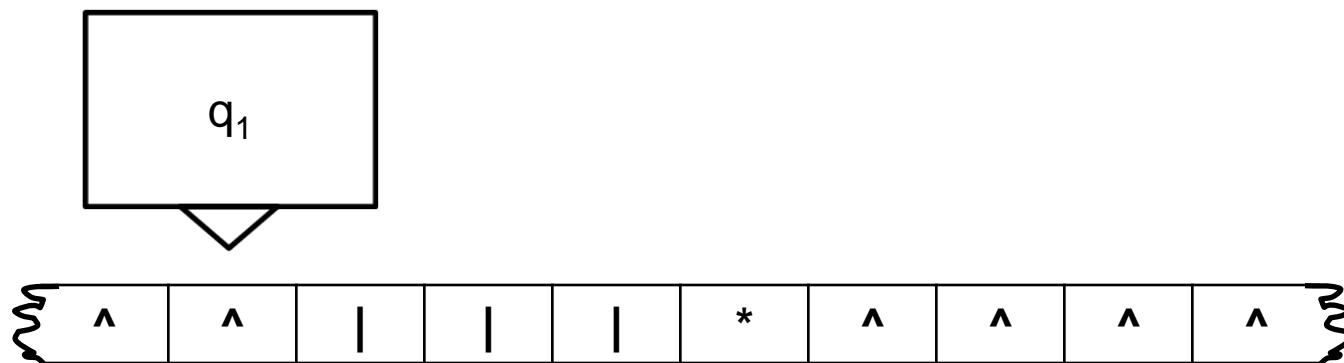
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



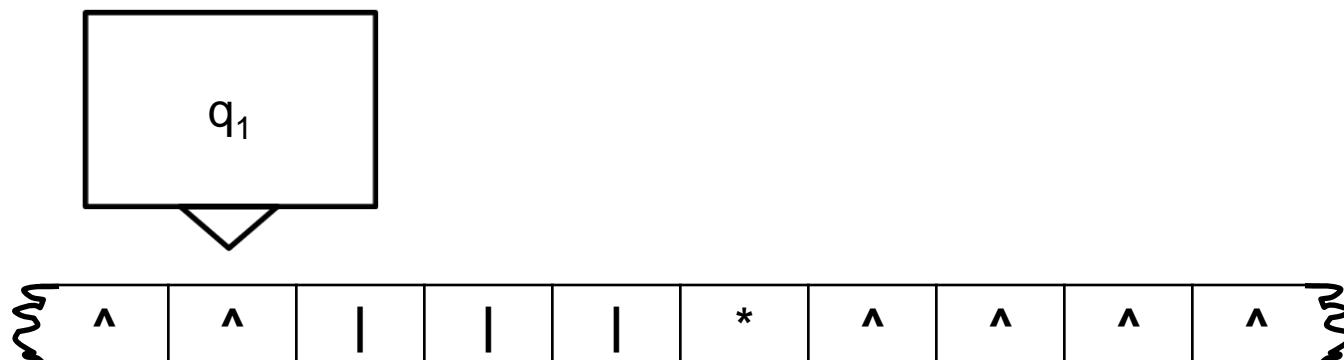
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



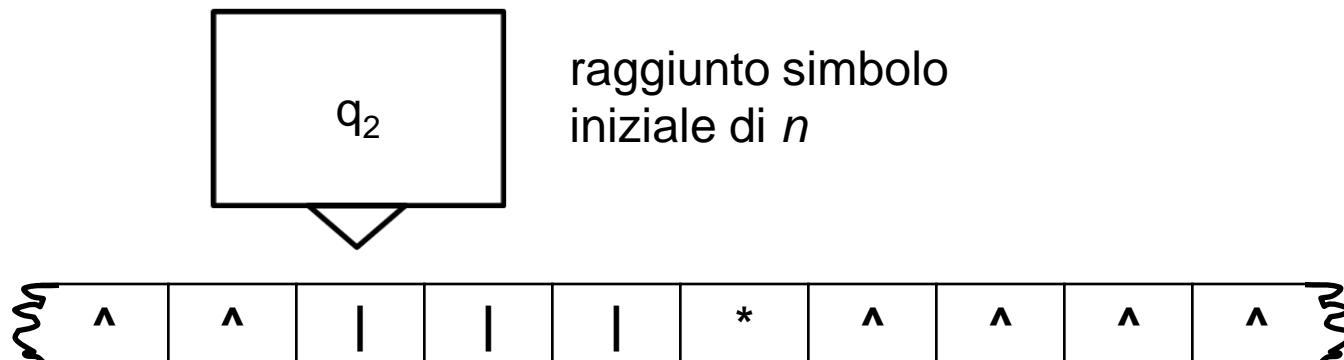
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



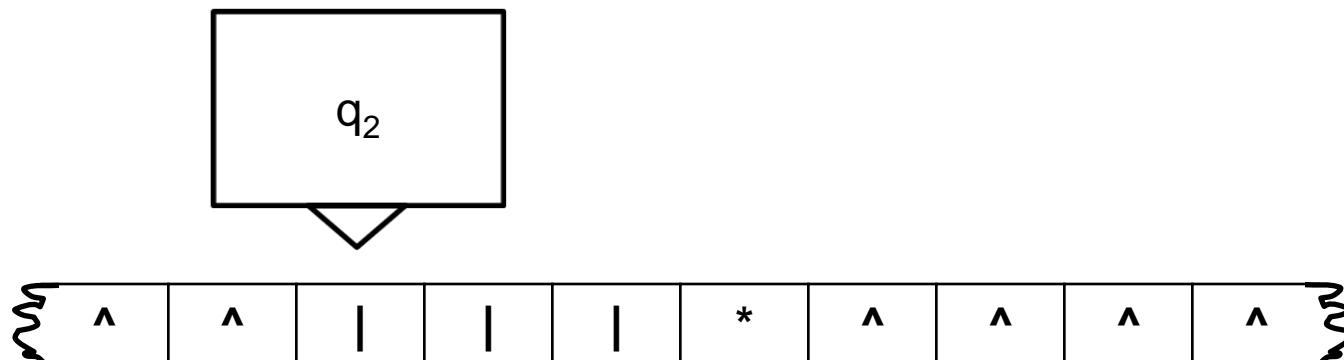
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



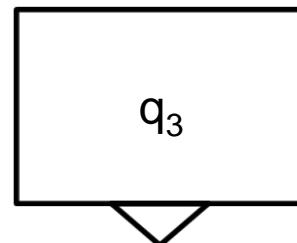
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$

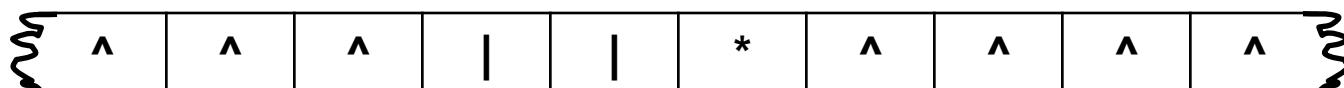


# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
$ $	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$

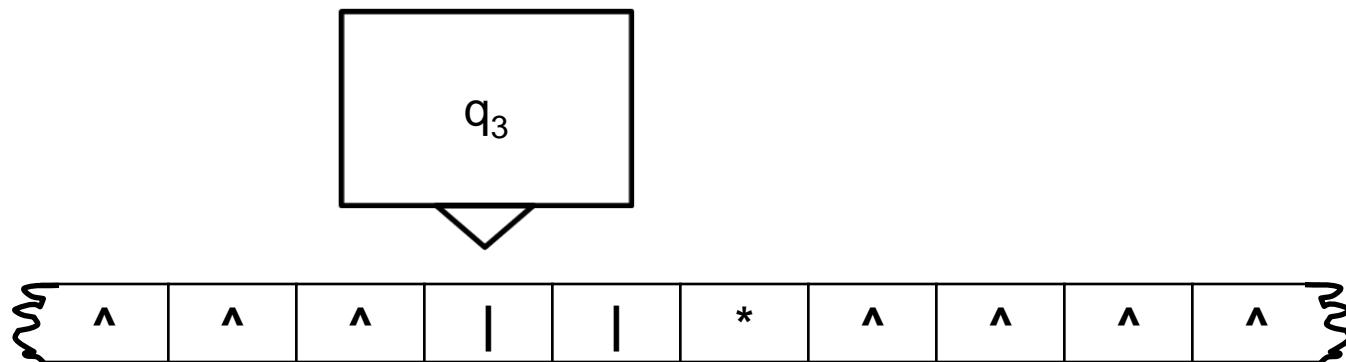


Diminuiti entrambi  
gli operandi



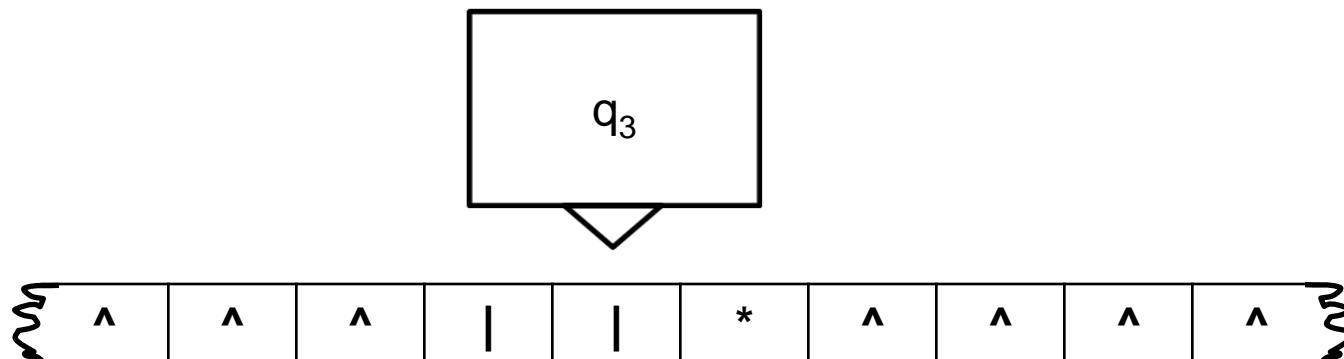
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



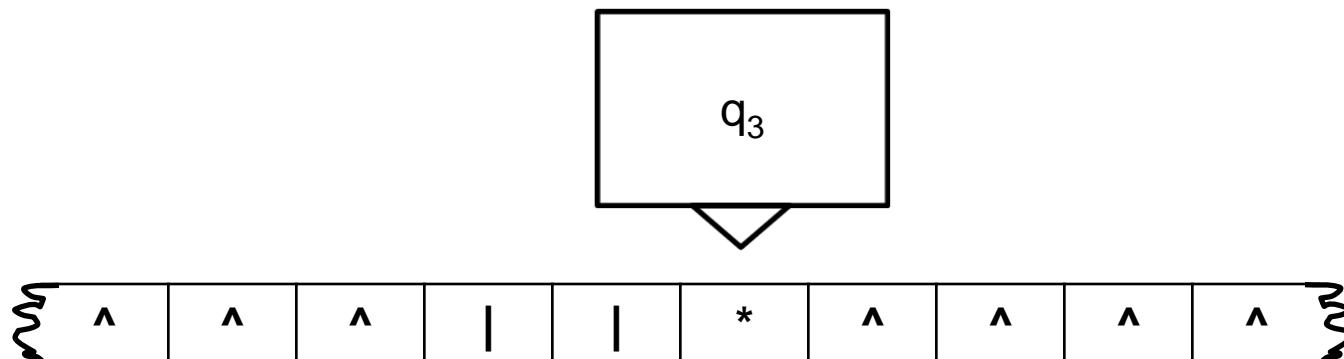
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



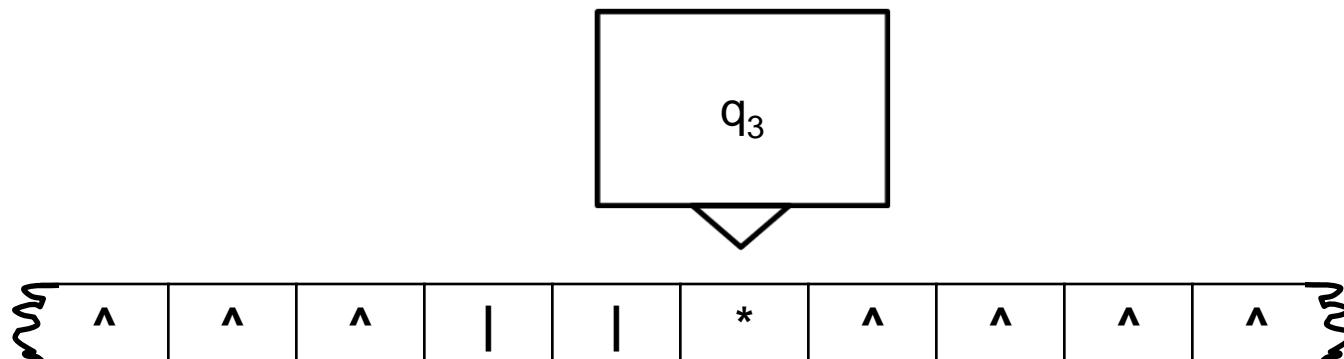
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



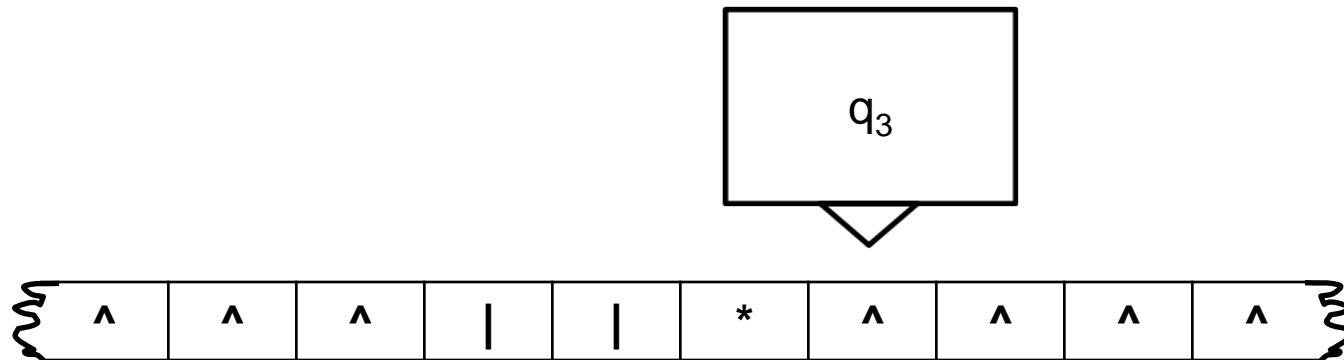
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



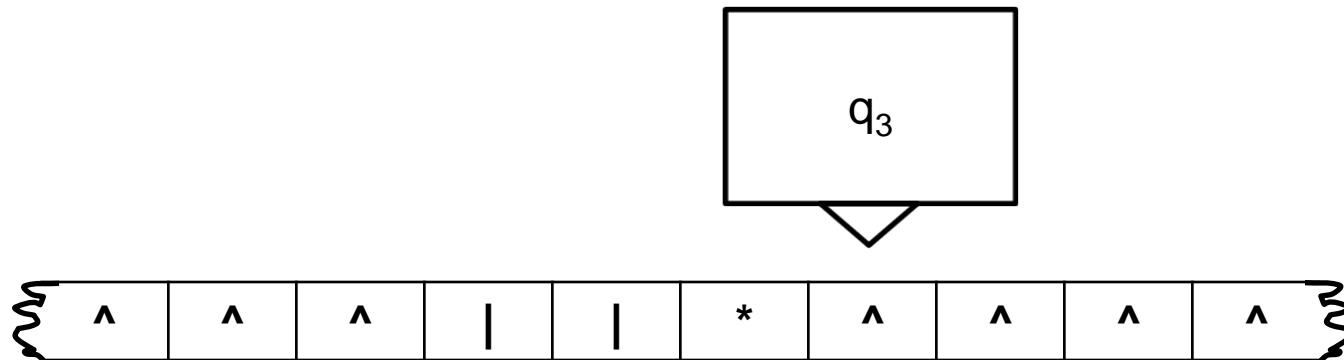
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$^S q_0$	$^D q_2$		$^S q_0$
	$^S q_1$	$  S q_1$	$^D q_3$	$  D q_3$
*	$^F \text{ HALT}$	$* S q_1$		$* D q_3$



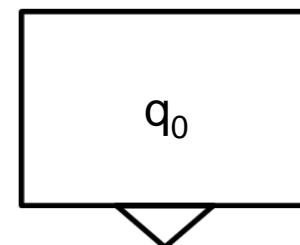
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$^S q_0$	$^D q_2$		$^S q_0$
	$^S q_1$	$  S q_1$	$^D q_3$	$  D q_3$
*	$^F \text{ HALT}$	$* S q_1$		$* D q_3$



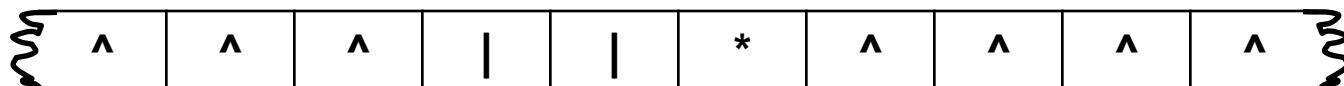
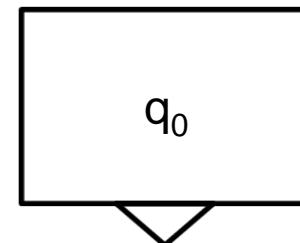
Stato iniziale della  
computazione



# Computazione 3-1

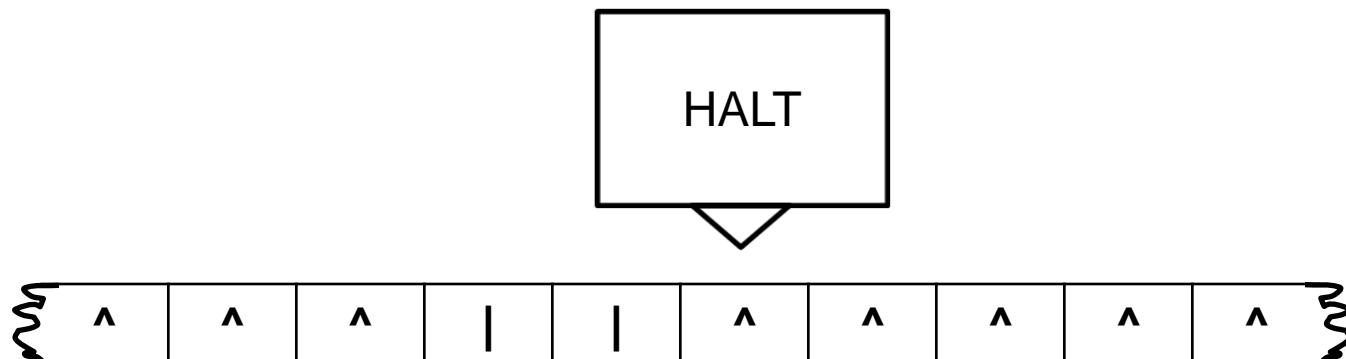
$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ <u>HALT</u>	$* Sq_1$		$* Dq_3$

trovare un \* nello stato iniziale  
della computazione è segno  
del fatto che non ci sono più  
simboli da processare in  $m$



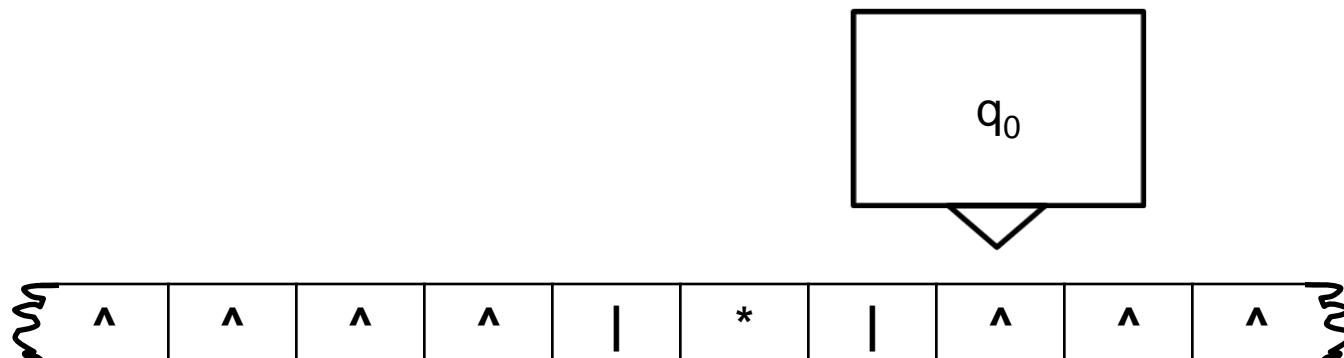
# Computazione 3-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



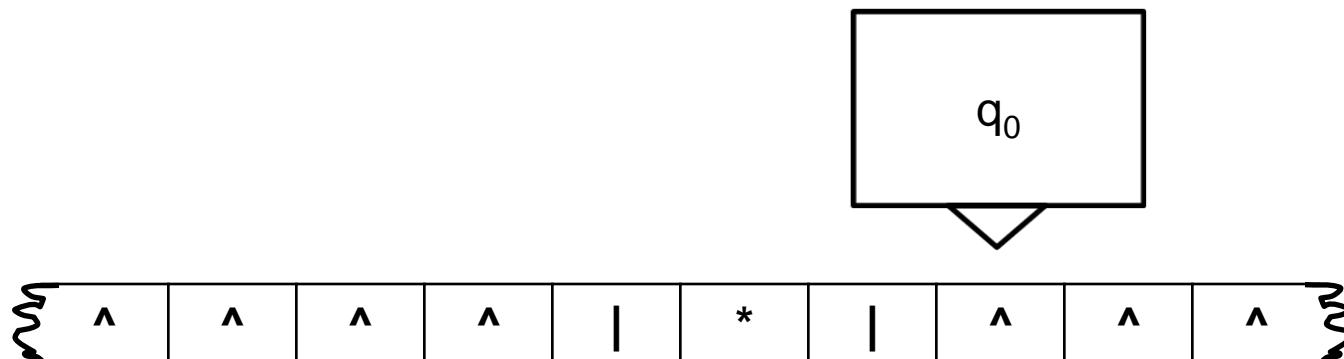
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$^S q_0$	$^D q_2$		$^S q_0$
	$^S q_1$	$  S q_1$	$^D q_3$	$  D q_3$
*	$^F \text{ HALT}$	$* S q_1$		$* D q_3$



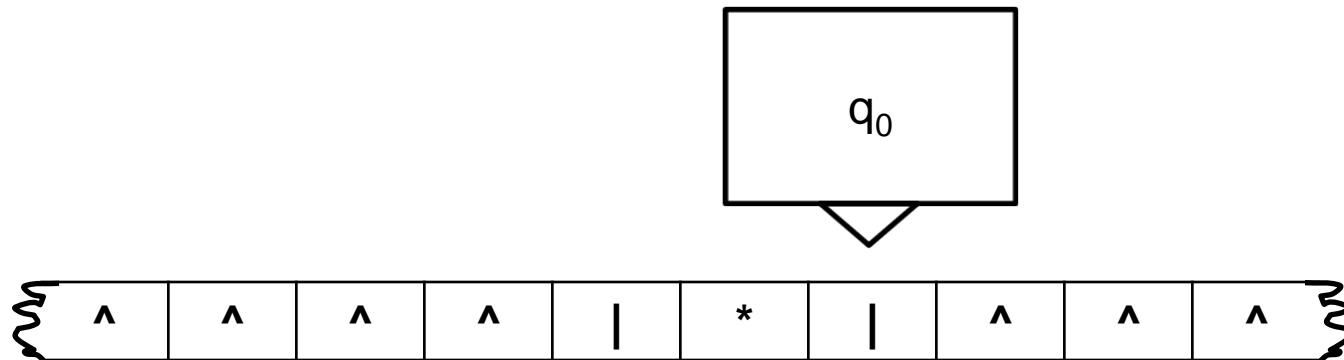
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



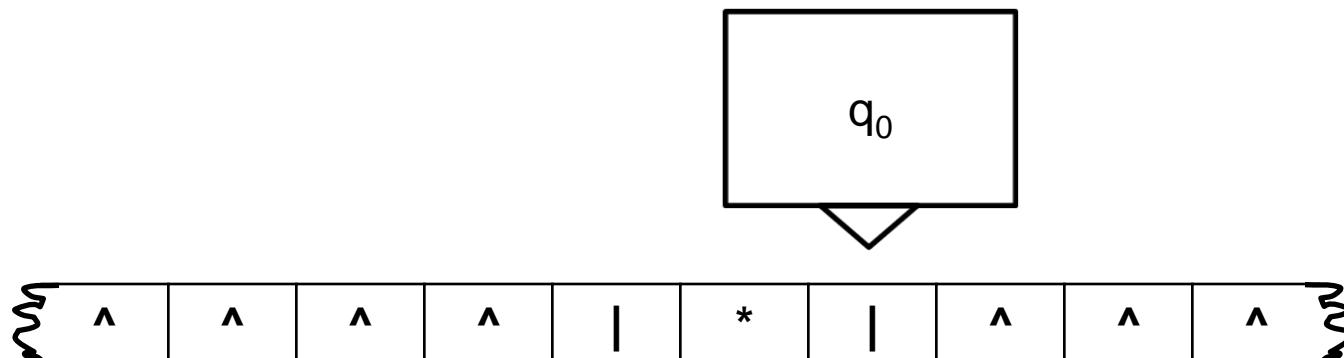
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$^S q_0$	$^D q_2$		$^S q_0$
	$^S q_1$	$  S q_1$	$^D q_3$	$  D q_3$
*	$^F \text{ HALT}$	$* S q_1$		$* D q_3$



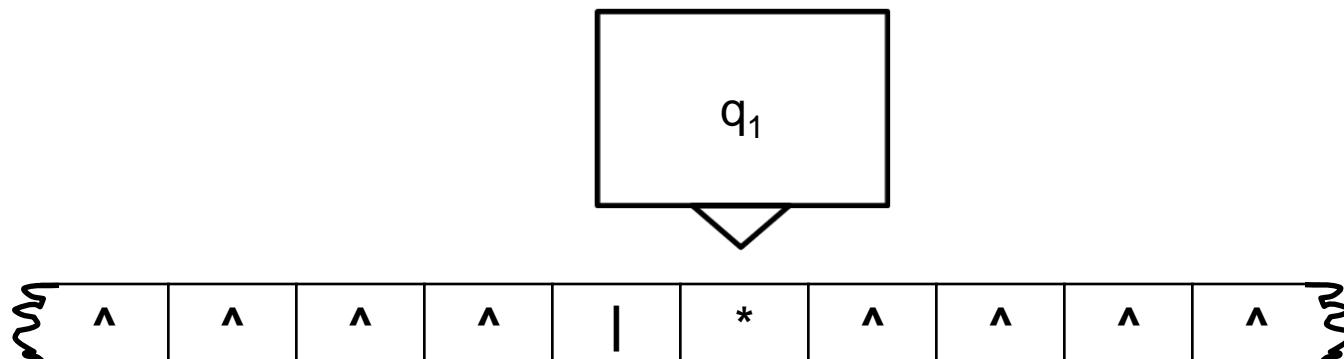
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



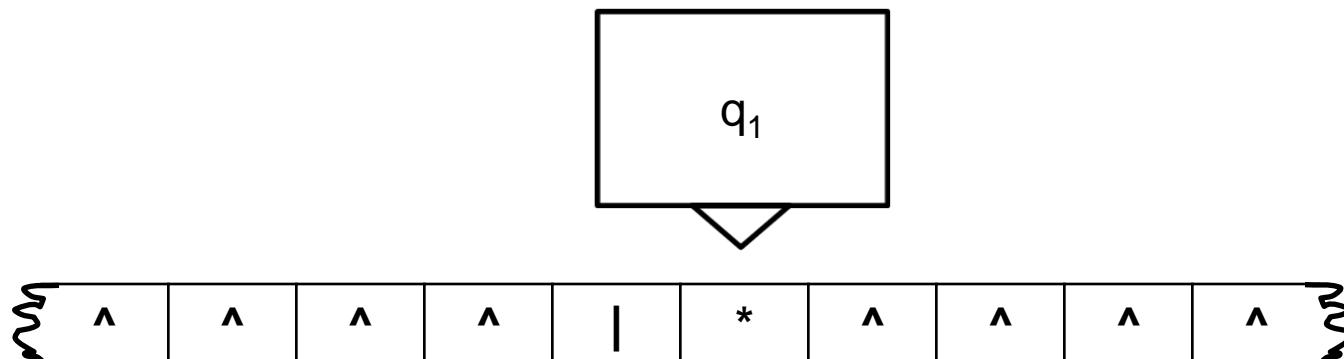
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$^S q_0$	$^D q_2$		$^S q_0$
	$^S q_1$	$  S q_1$	$^D q_3$	$  D q_3$
*	$^F \text{ HALT}$	$* S q_1$		$* D q_3$



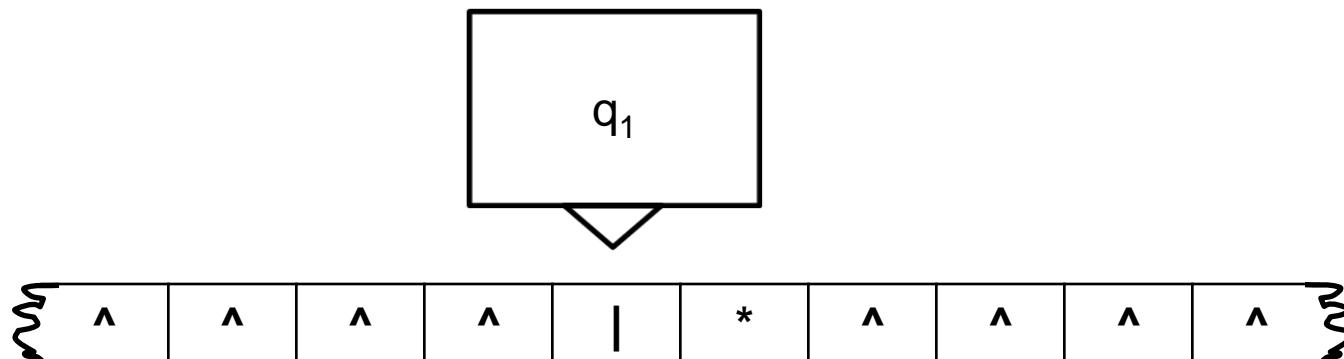
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



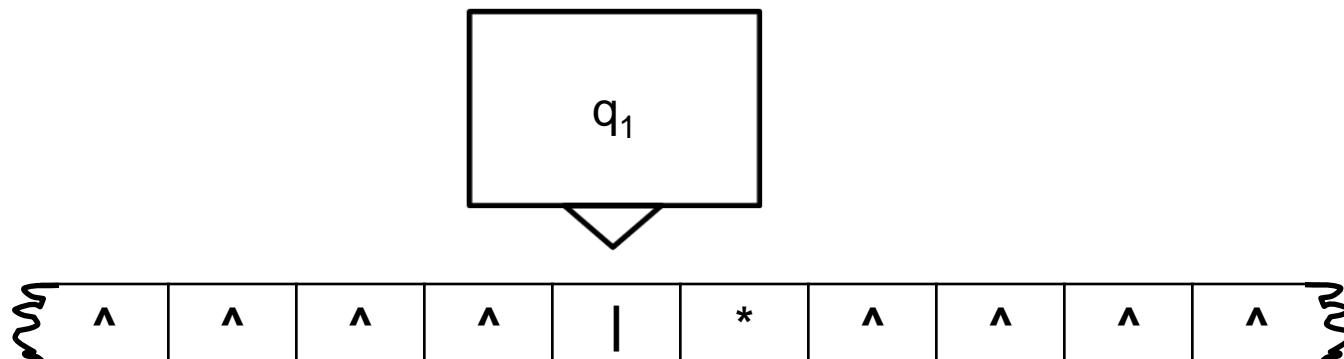
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$^S q_0$	$^D q_2$		$^S q_0$
	$^S q_1$	$  S q_1$	$^D q_3$	$  D q_3$
*	$^F \text{ HALT}$	$* S q_1$		$* D q_3$



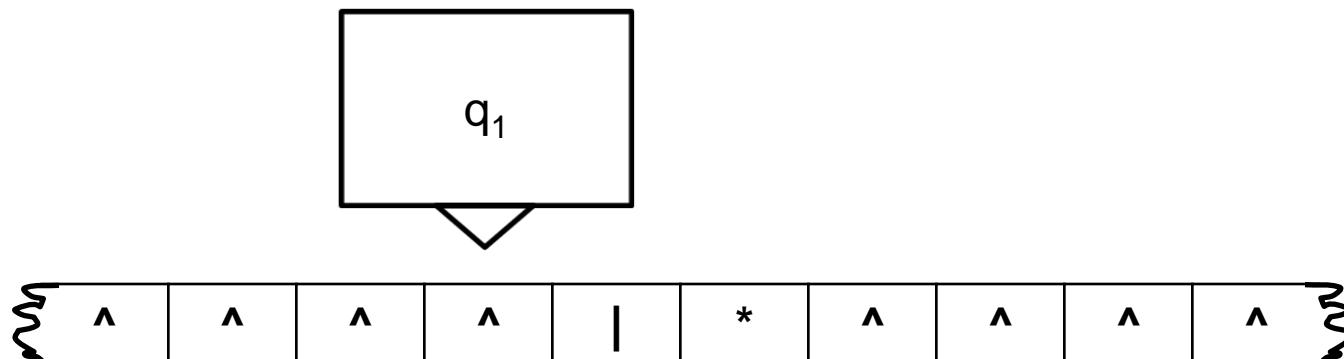
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



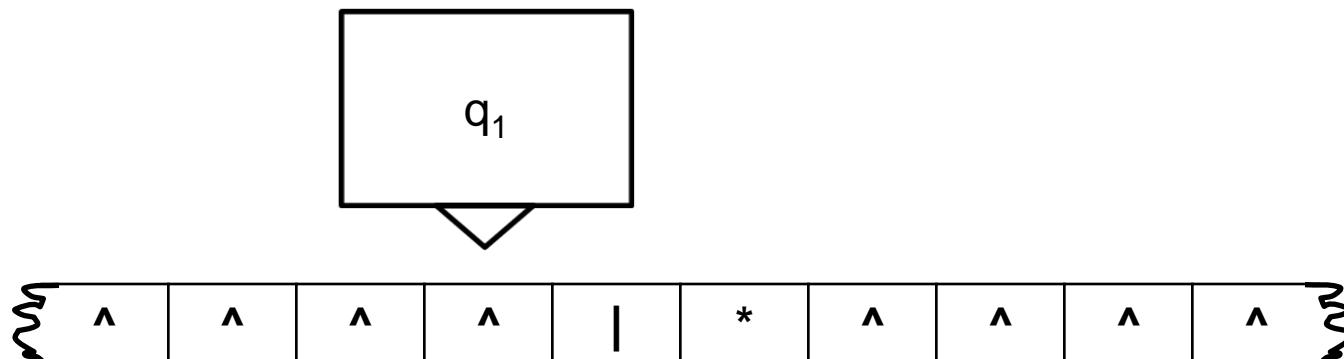
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



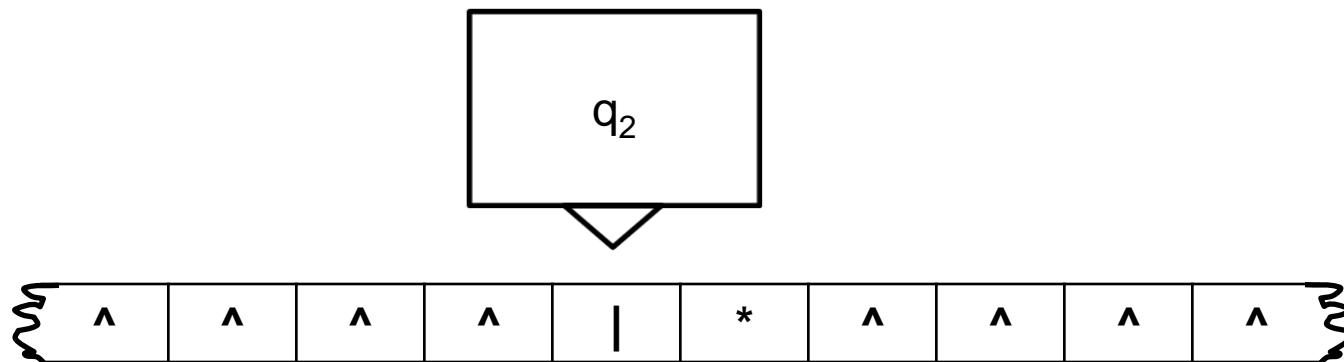
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



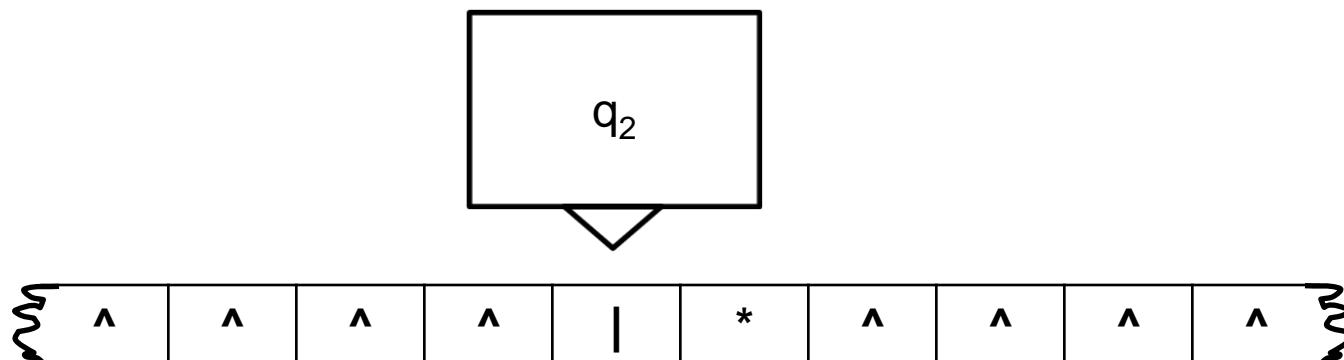
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



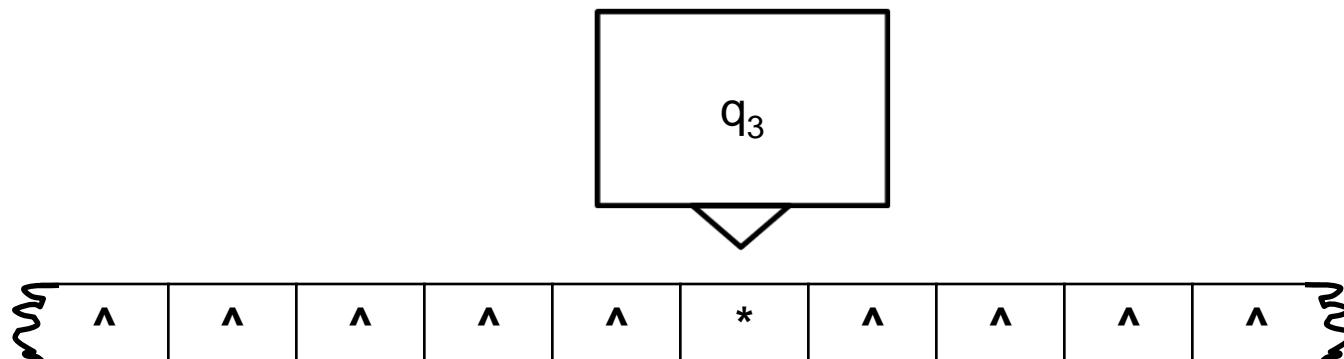
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



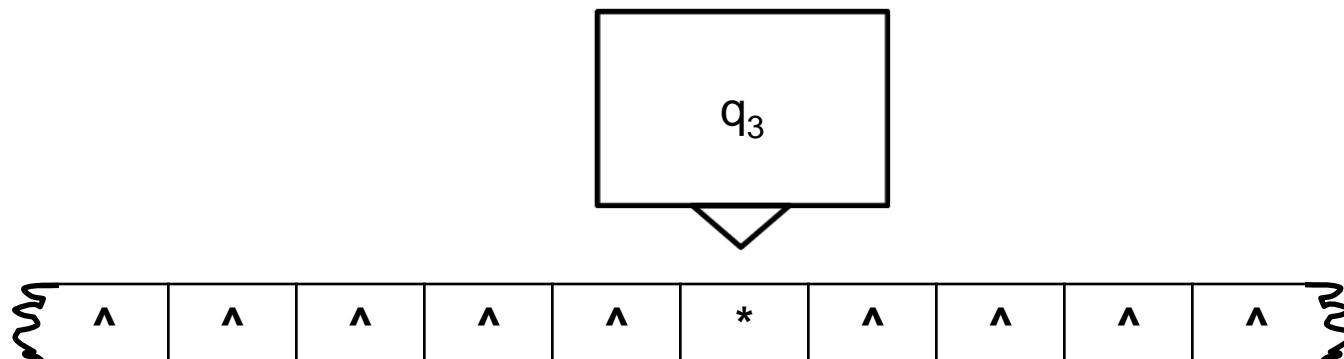
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



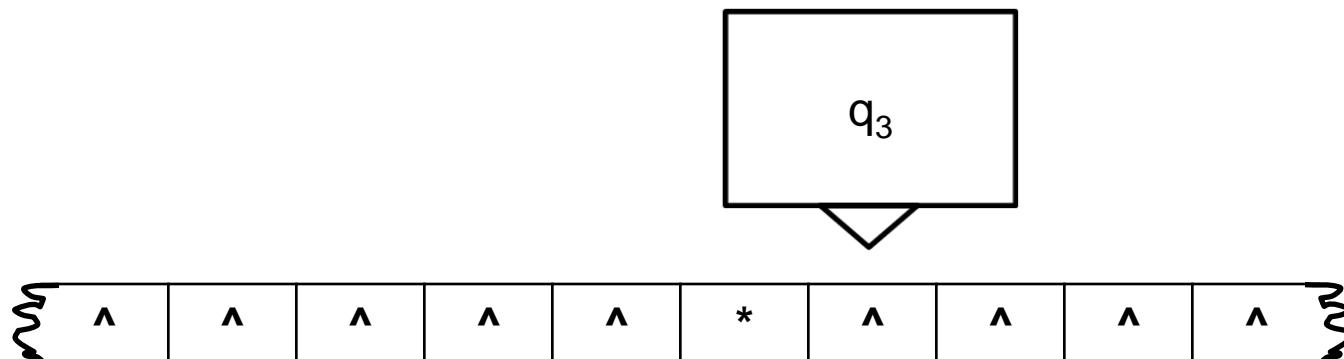
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



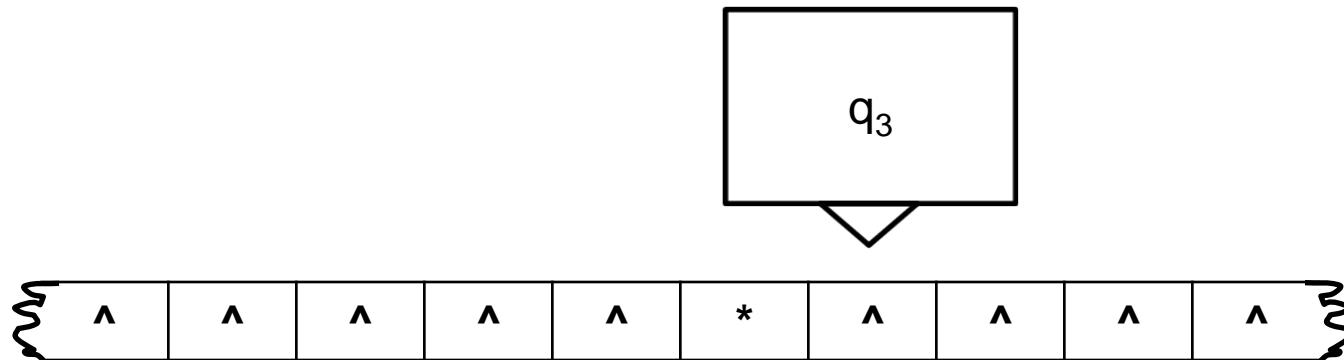
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



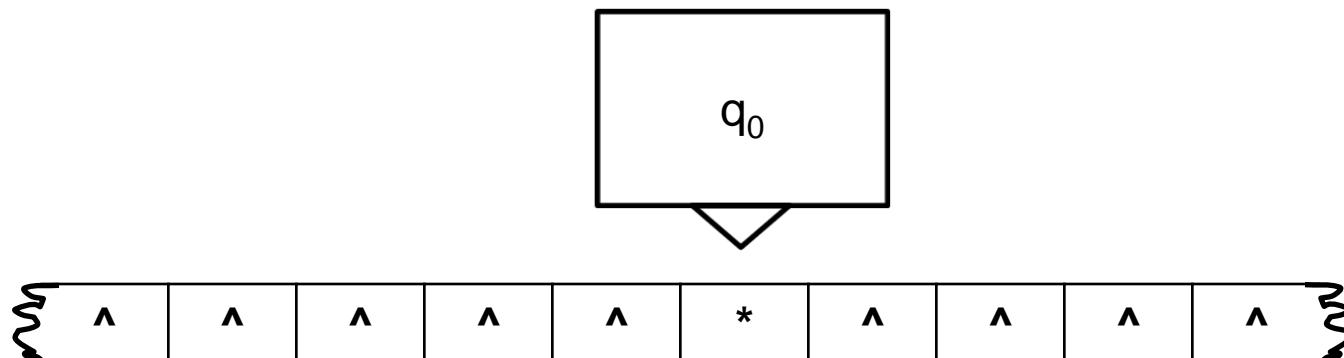
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



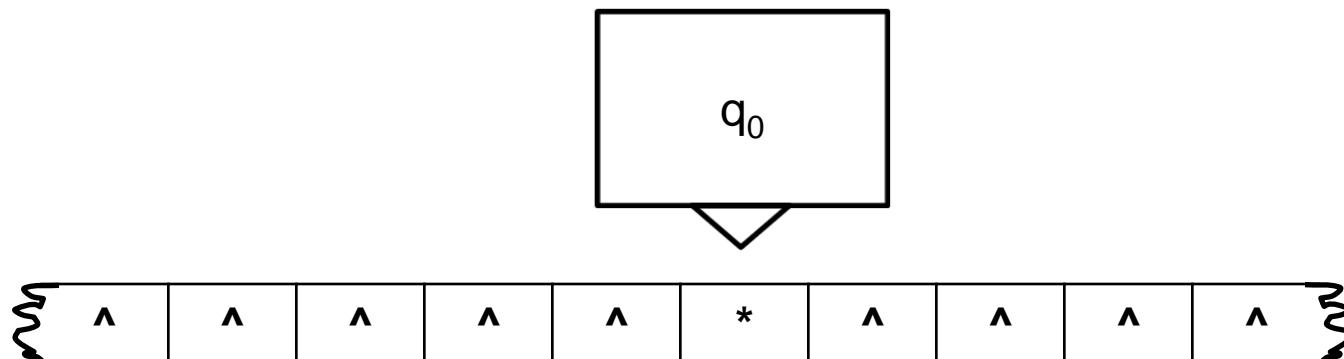
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$^S q_0$	$^D q_2$		$^S q_0$
	$^S q_1$	$  S q_1$	$^D q_3$	$  D q_3$
*	$^F \text{ HALT}$	$* S q_1$		$* D q_3$



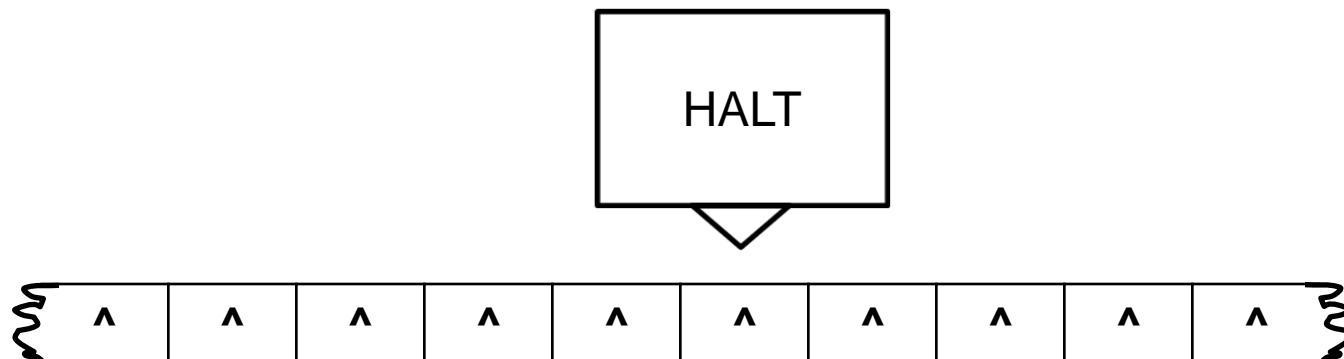
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ <u>HALT</u>	$* Sq_1$		$* Dq_3$



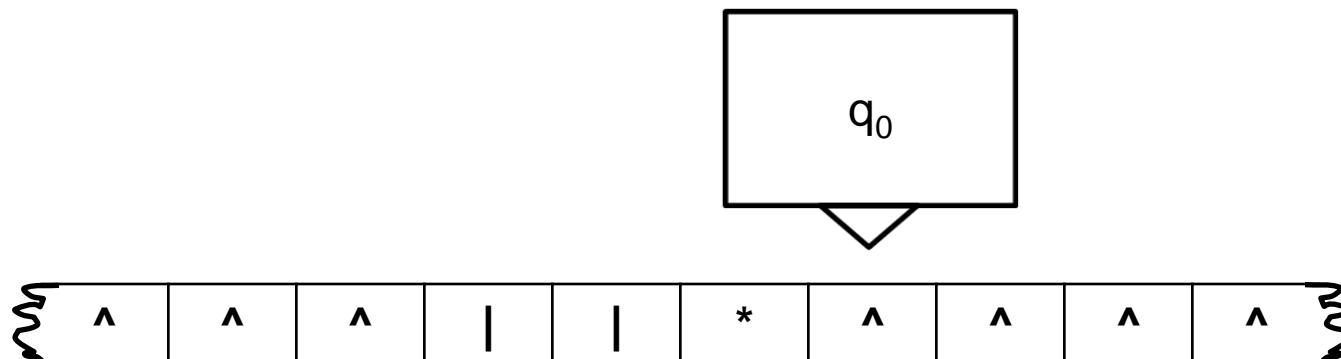
# Test casi particolari: computazione 1-1

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$^S q_0$	$^D q_2$		$^S q_0$
	$^S q_1$	$  S q_1$	$^D q_3$	$  D q_3$
*	$^F \text{ HALT}$	$* S q_1$		$* D q_3$



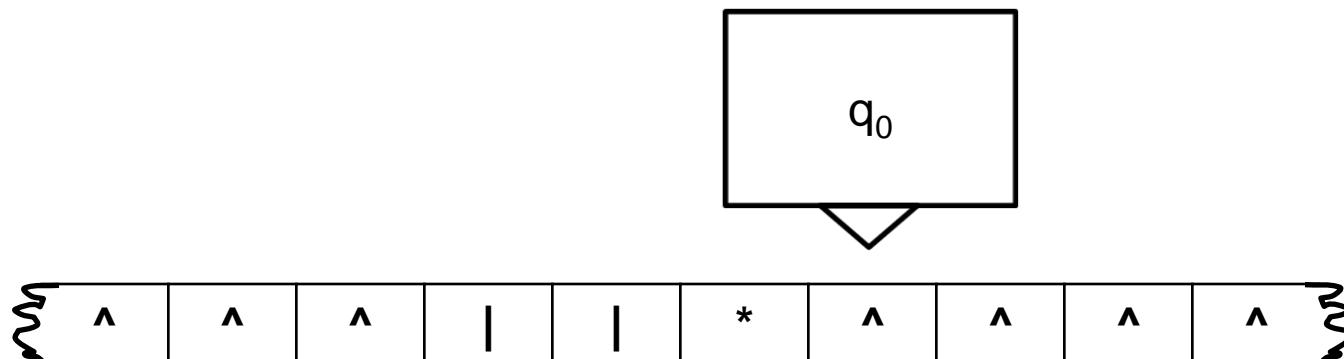
# Test casi particolari: computazione 2-0

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



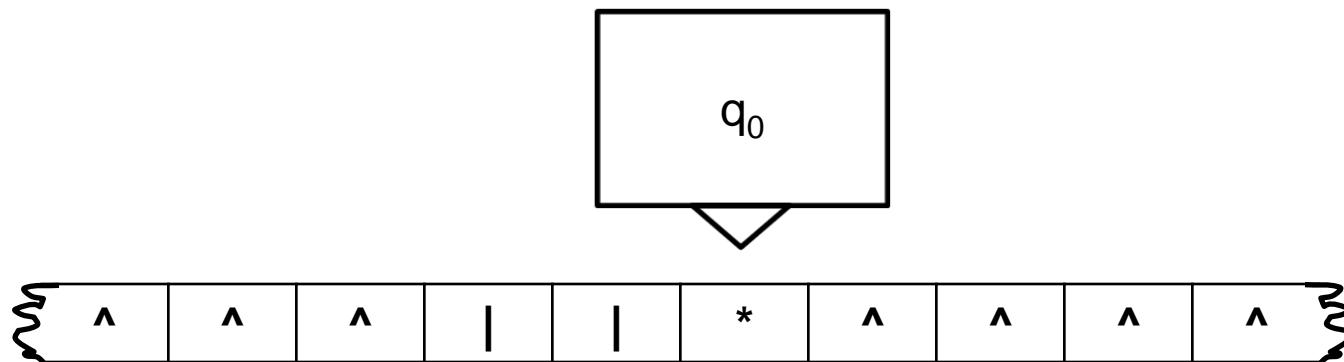
# Test casi particolari: computazione 2-0

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



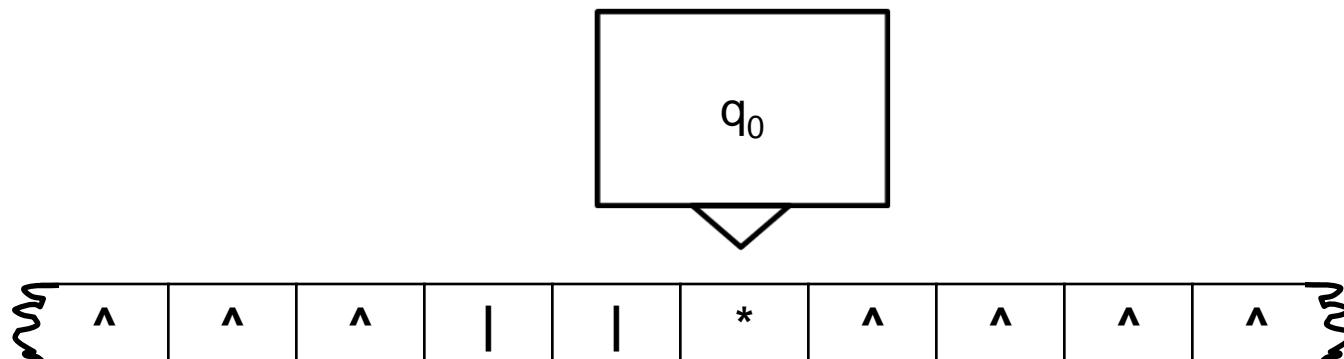
# Test casi particolari: computazione 2-0

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



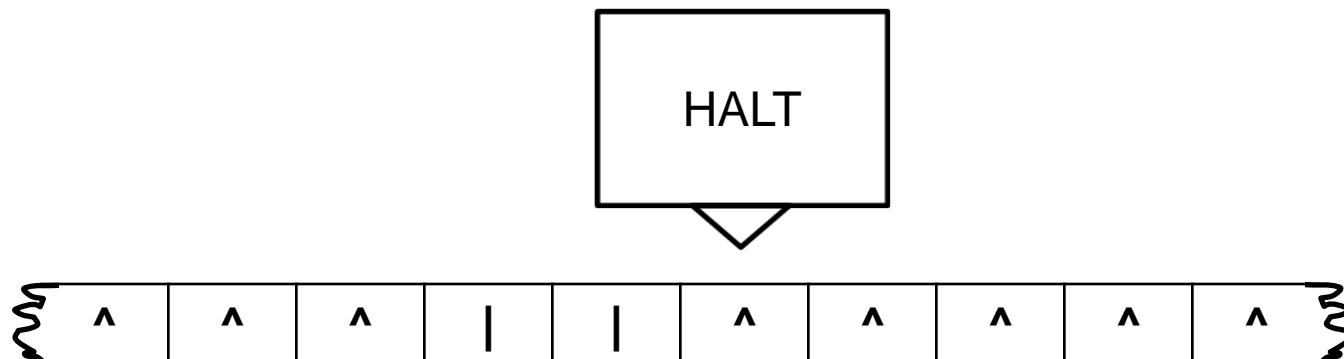
# Test casi particolari: computazione 2-0

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$^S q_0$	$^D q_2$		$^S q_0$
	$^S q_1$	$  S q_1$	$^D q_3$	$  D q_3$
*	$^F \text{ } \underline{\text{HALT}}$	$* S q_1$		$* D q_3$



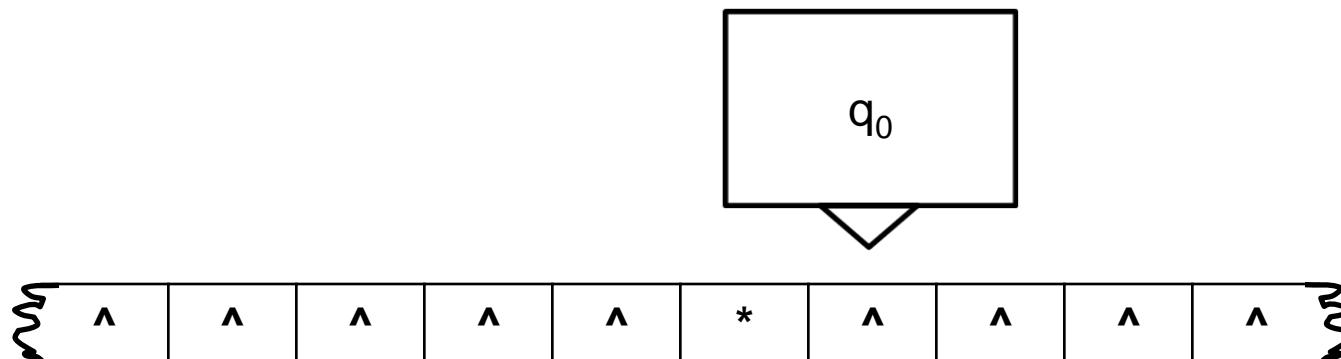
# Test casi particolari: computazione 2-0

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$^S q_0$	$^D q_2$		$^S q_0$
	$^S q_1$	$  S q_1$	$^D q_3$	$  D q_3$
*	$^F \text{ HALT}$	$* S q_1$		$* D q_3$



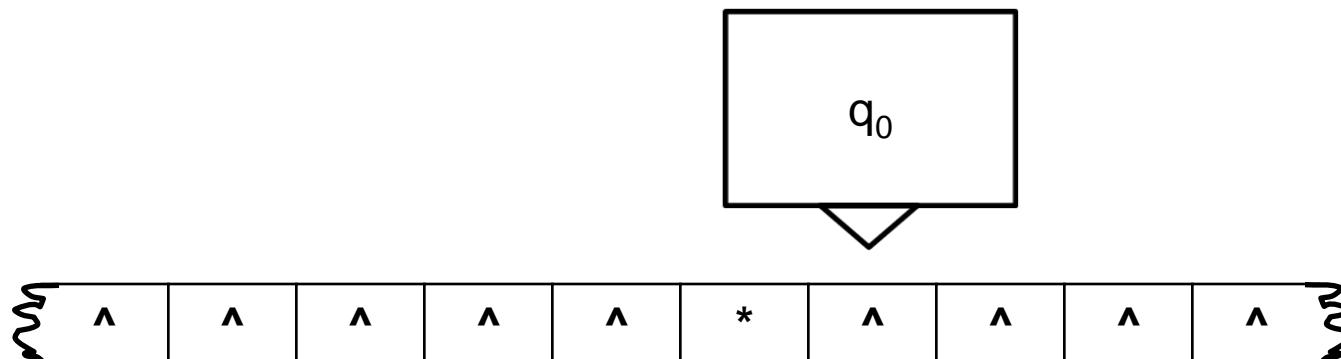
# Test casi particolari: computazione 0-0

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge S q_0$	$\wedge D q_2$		$\wedge S q_0$
	$\wedge S q_1$	$  S q_1$	$\wedge D q_3$	$  D q_3$
*	$\wedge F$ HALT	$* S q_1$		$* D q_3$



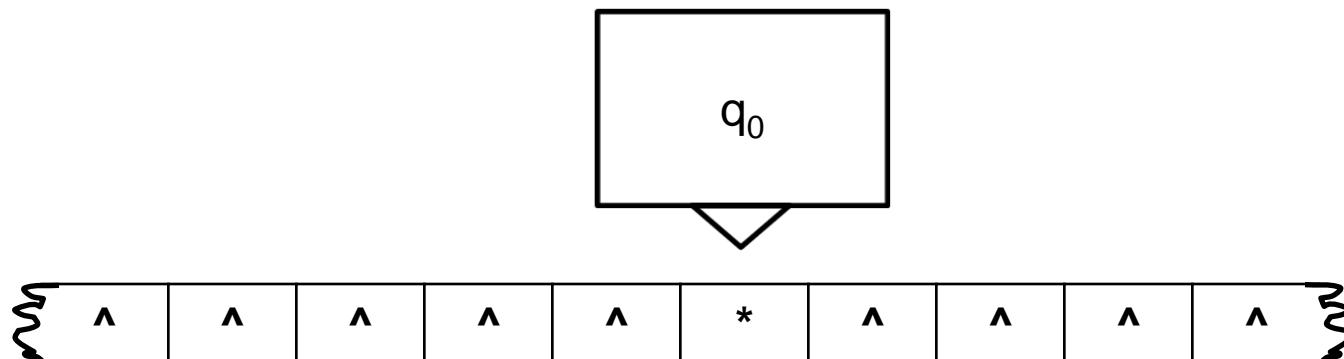
# Test casi particolari: computazione 0-0

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



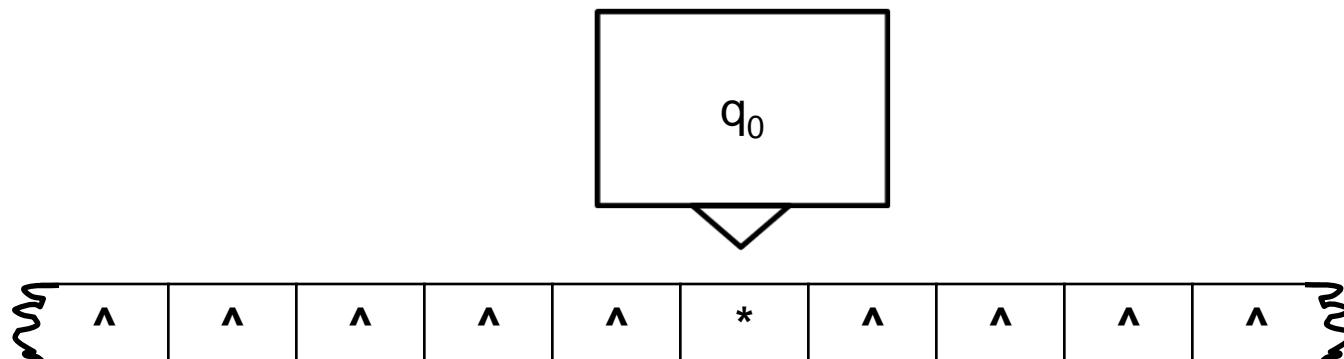
# Test casi particolari: computazione 0-0

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$\wedge Sq_0$	$\wedge Dq_2$		$\wedge Sq_0$
	$\wedge Sq_1$	$  Sq_1$	$\wedge Dq_3$	$  Dq_3$
*	$\wedge F$ HALT	$* Sq_1$		$* Dq_3$



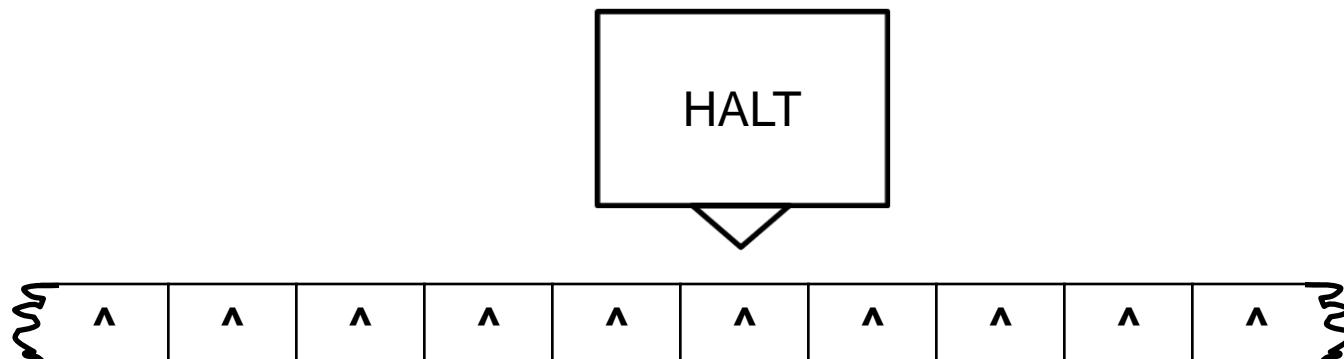
# Test casi particolari: computazione 0-0

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$^S q_0$	$^D q_2$		$^S q_0$
	$^S q_1$	$  S q_1$	$^D q_3$	$  D q_3$
*	$^F \text{ } \underline{\text{HALT}}$	$* S q_1$		$* D q_3$



# Test casi particolari: computazione 0-0

$x \setminus Q$	$q_0$	$q_1$	$q_2$	$q_3$
$\wedge$	$^S q_0$	$^D q_2$		$^S q_0$
	$^S q_1$	$  S q_1$	$^D q_3$	$  D q_3$
*	$^F \text{ HALT}$	$* S q_1$		$* D q_3$



# Have Fun with MdT



- Stabilire una stringa binaria contiene lo stesso numero di '0' e '1'
- Stabilire se una stringa binaria è palindroma (ovvero si legge indifferentemente da S a D, es.: 01000010)
- Stabilire se un numero rappresentato con 'l' è pari oppure dispari

# Tesi di Church-Turing

- La classe delle funzioni calcolabili coincide con la classe delle funzioni calcolabili da MdT
  - *ogni funzione calcolabile è calcolata da una MdT*
  - *Non esiste alcun formalismo capace di risolvere una classe di problemi più ampia di quella che si può risolvere con MdT*
- Le funzioni calcolabili con C o Java sono di più di quelle calcolabili con MdT?

NO

# Un parallelo tra MdT e moderni processori

## ■ MdT

- ① Legge / scrive su nastro
- ② Transita in un nuovo stato
- ③ Si sosta sul nastro di cella in cella
- ④ Esegue un programma **specifico CABLATO** nella macchina → è specifica per un certo problema

## ■ CPU

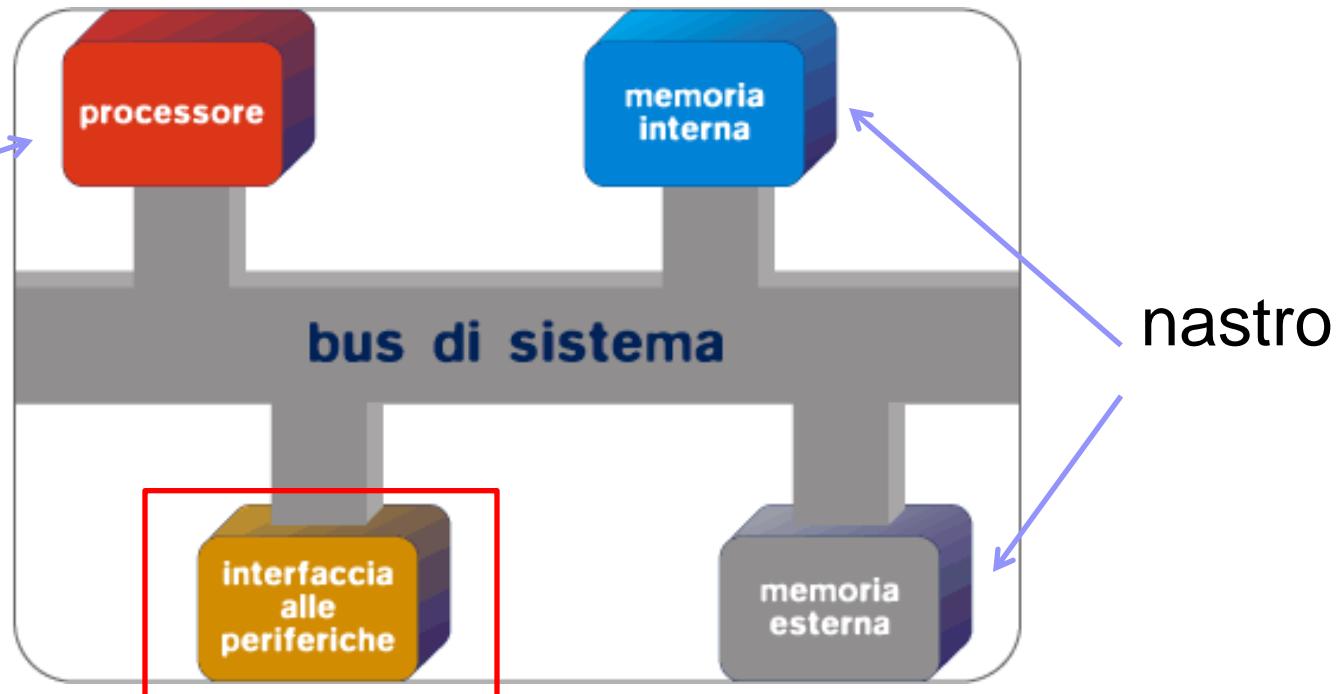
- ① lettura / scrittura da / su memoria RAM o ROM
- ② nuova configurazione dei registri della CPU
- ③ scelta della cella di memoria su cui operare (indirizzo contenuto nel RI)
- ④ È **generale**, nel senso che può eseguire programmi diversi

# La MdT Universale (MdTU)

- Legge dal nastro DATI e PROGRAMMA
  - *Il programma non è più cablato nell'unità di controllo*
  - *Codificato sul nastro come i dati*
  - *In pratica sono rappresentate sul nastro anche le 5-ple che definiscono l'algoritmo solutivo*
- E' una macchina programmabile
  - prende le 5-ple (istruzioni) dal nastro → FETCH
  - le decodifica → DECODE
  - le esegue scrivendo sul nastro → EXECUTE
- E' un interprete!!!!!

# MdTU e Macchina Von Neumann

MdTU  
(controllo)



MdTU è un modello della macchina di Von Neumann ovvero un  
modello degli attuali calcolatori!!!

(manca solo la parte di I/O)

# Recap

- Un linguaggio di programmazione  $\mathcal{L}$  è un formalismo per portare al livello di macchina fisica gli algoritmi
  - *Implementare  $\mathcal{L}$  significa realizzarne l'interprete ovvero il programma che esegue la traduzione di  $\mathcal{L}$  per la macchina ospite*
- La possibilità di risolvere un problema non dipende da  $\mathcal{L}$ 
  - Tutti i linguaggi di programmazione calcolano esattamente le stesse funzioni calcolate dalle MdT
  - Tutti i linguaggi di programmazione sono Turing-completi

# Introduzione alla teoria dei linguaggi formali

- La **teoria dei linguaggi formali** rappresenta spesso uno scoglio per gli studenti di informatica a causa della sua forte dipendenza dalla notazione.
- L'argomento, d'altra parte, non può essere evitato perché ogni laureato in informatica deve possedere una buona comprensione dell'operazione di **compilazione** e questa, a sua volta, si fonda pesantemente sulla teoria dei linguaggi formali.

# Introduzione alla teoria dei linguaggi formali

## ■ Livelli di descrizione di un linguaggio

- grammatica

- quali frasi sono corrette?

- semantica

- cosa significa una frase corretta?

- pragmatica

- come usare una frase corretta e sensata?

- implementazione (per i linguaggi di programmazione)

- come eseguire una frase corretta in modo da rispettarne il significato?

# Concetto intuitivo di grammatica

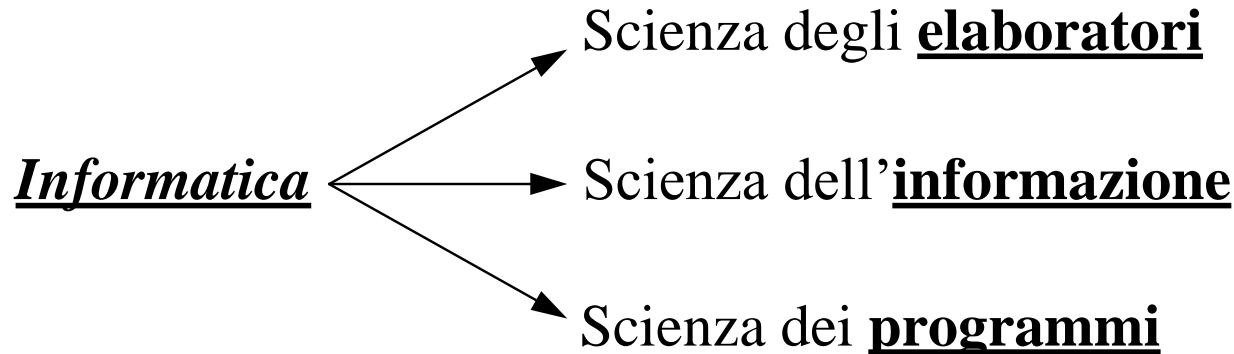
- Alfabeto del linguaggio
  - simboli con cui “costruire” le parole del linguaggio
  - es.: alfabeto latino su 22 o 26 lettere
- Lessico
  - parole del linguaggio
  - es.: informatica
- Sintassi
  - determina quali sequenze di parole costituiscono frasi legali (corrette)
- **Le grammatiche sono uno strumento utile per descrivere la sintassi di un linguaggio di programmazione**

# Introduzione alla teoria dei linguaggi formali

- Questa prima parte del programma ha l'obiettivo di fornire una **base** sufficiente alla comprensione delle **tecniche di compilazione**, oggetto della seconda parte del corso.
- E' per questa ragione che ci concentreremo quasi esclusivamente su due tipi di **grammatiche**:
  - **regolari**
  - **libere da contesto**poiché i linguaggi formali utilizzati in informatica sono per lo più di questi tipi.

# Introduzione

- L'informatica teorica è la scienza degli algoritmi, in tutti i loro aspetti, poiché è il concetto di algoritmo che è in qualche modo comune a tutti i settori dell'informatica.



elaboratori

≡ macchine che eseguono gli algoritmi

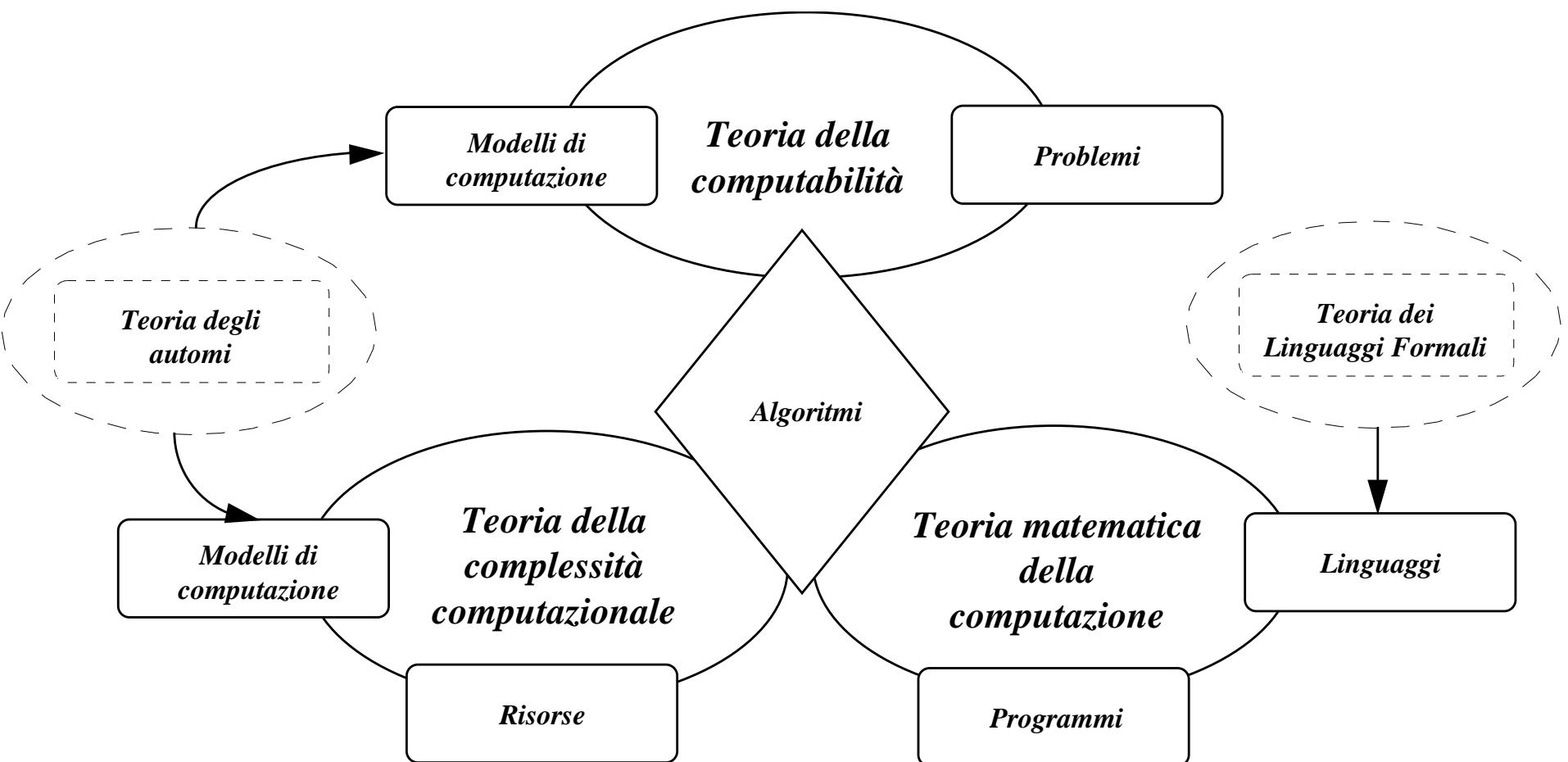
informazione

≡ materia su cui lavorano gli algoritmi

programmi

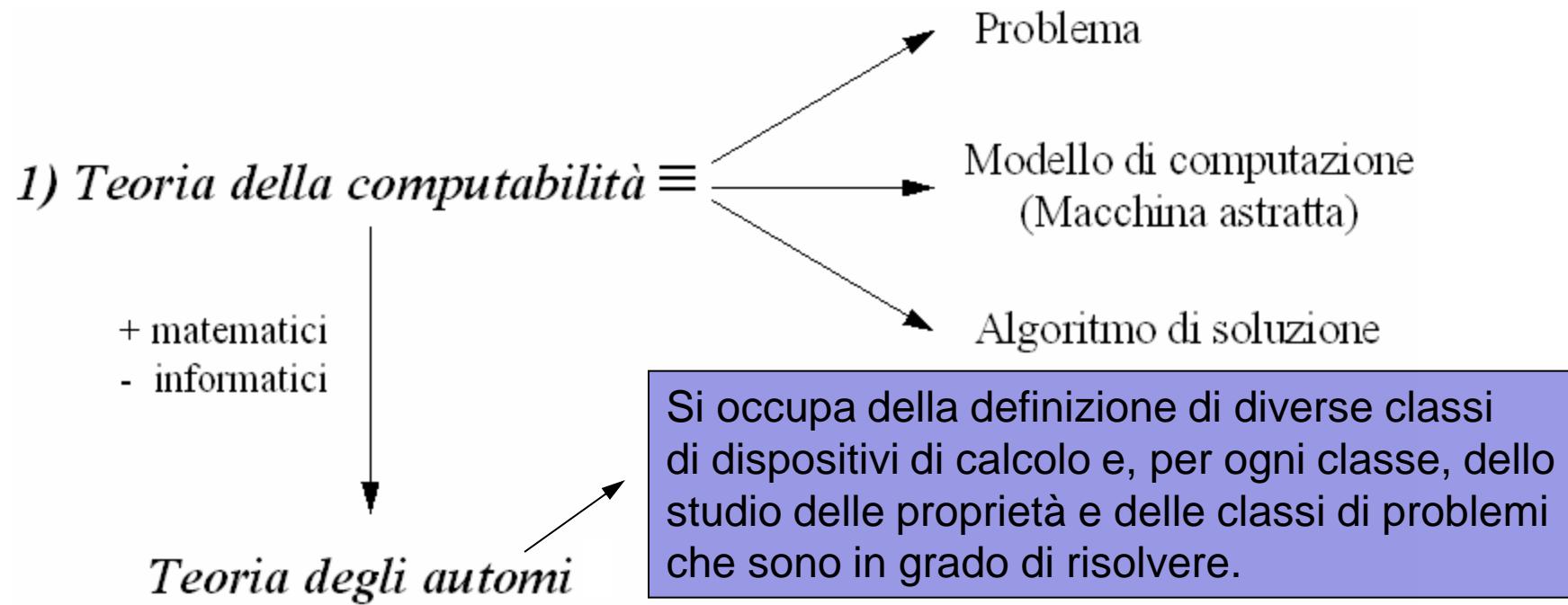
≡ algoritmi descritti in un particolare linguaggio

# Aree di ricerca dell'informatica teorica



# Aree di ricerca dell'informatica teorica

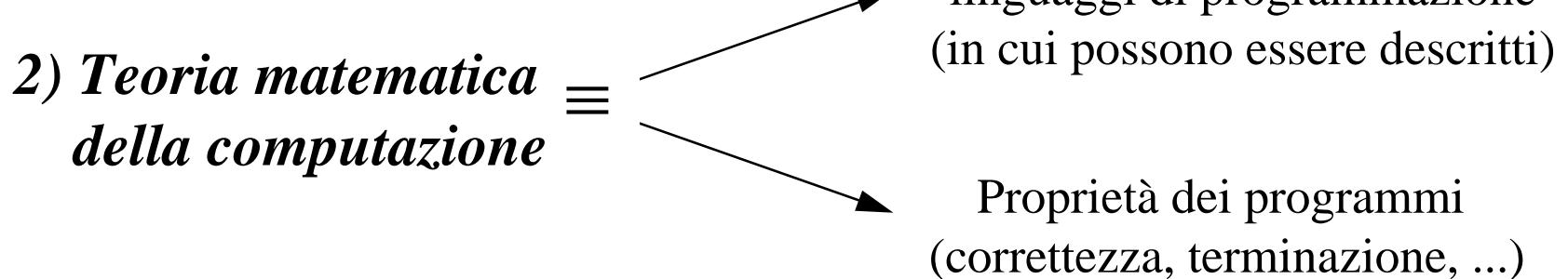
- Lo schema riportato in figura fornisce una panoramica delle aree di ricerca che ricadono nell'ambito più generale dell'informatica teorica. Tali aree sono:



# Aree di ricerca dell'informatica teorica

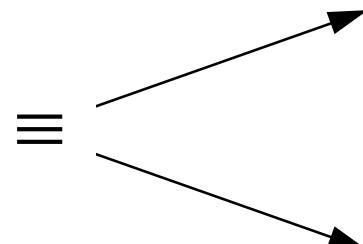
- Particolarmente feconda si è rivelata l'interazione tra la ***Teoria degli Automi*** e la ***Teoria dei Linguaggi Formali***.
- Quest'ultima, nata in ambito linguistico per caratterizzare i linguaggi naturali, è stata sviluppata ed utilizzata dagli informatici teorici, che avevano l'esigenza di descrivere gli algoritmi con linguaggi di programmazione comprensibili dalla macchina, ma non troppo difficili per l'uomo e soprattutto **non ambigui**.

# Aree di ricerca dell'informatica teorica



# Aree di ricerca dell'informatica teorica

## 3) *Teoria della complessità computazionale*



Rapporto tra gli algoritmi e le risorse necessarie per eseguirli

Aspetti quantitativi del procedimento di soluzione di un problema

# Studio dei linguaggi

- Quando iniziamo a studiare un linguaggio, formale o meno, tutti quanti godiamo di un grande vantaggio: siamo esperti in un linguaggio, quello con cui comunichiamo con gli altri.
- Oltre ad essere competenti in uno o più linguaggi naturali, lo studente in informatica ha familiarità con diversi linguaggi di programmazione, come il FORTRAN, il PASCAL, il C, il PROLOG, ...

# Studio dei linguaggi

- Questi linguaggi sono usati per scrivere programmi e quindi per comunicare con il computer.
- Chiunque utilizzi un computer potrà notare che esso è particolarmente limitato nel comprendere il significato desiderato dei programmi.
- In linguaggio naturale possiamo di solito comunicare anche attraverso frasi mal strutturate e incomplete
- Quando dobbiamo comunicare con un computer la situazione cambia.

# Sintassi e semantica

- I nostri programmi devono aderire rigorosamente a regole stringenti e anche differenze minori vengono rigettate come errate.
- Idealmente, ogni frase in un linguaggio dovrebbe essere corretta sia **semanticamente** (avere cioè il corretto significato) sia **sintatticamente** (avere la corretta struttura grammaticale).
- Nell'italiano parlato, le frasi sono spesso sintatticamente sbagliate, ciononostante convogliano la semantica desiderata.

# Sintassi e semantica

- In programmazione, comunque, è essenziale che la **sintassi** sia corretta al fine di comunicare una qualsivoglia semantica.
- Quando studiamo la semantica di una frase, ne stiamo studiando il significato.

# Sintassi e semantica

- Tutte le frasi che seguono hanno la stessa interpretazione semantica, cioè lo stesso significato, sebbene siano sintatticamente differenti:

**The man hits the dog**

**The dog is hit by the man**

**L'homme frappe le chien**

# Sintassi e semantica

- Lo studio della sintassi è lo studio della grammatica, cioè della struttura delle frasi. La frase:

***The man hits the dog***

può essere analizzata sintatticamente, cioè risolta nelle parti grammaticali componenti, come segue:

The man

< parte nominale >

hits

< parte verbale >

the dog

< parte nominale >

ed ogni frase in questa forma è sintatticamente valida in inglese.

# Sintassi e semantica

- Possiamo descrivere un particolare insieme di tali frasi semplici in inglese usando le seguenti regole:

< frase semplice > :: = < parte nominale > < parte verbale >

                  < parte nominale >

< parte nominale > :: = < articolo > < nome >

< nome > :: = **car** | **man** | **dog**

< articolo > :: = **The** | **a**

< parte verbale > :: = **hits** | **eats**

# Sintassi e semantica

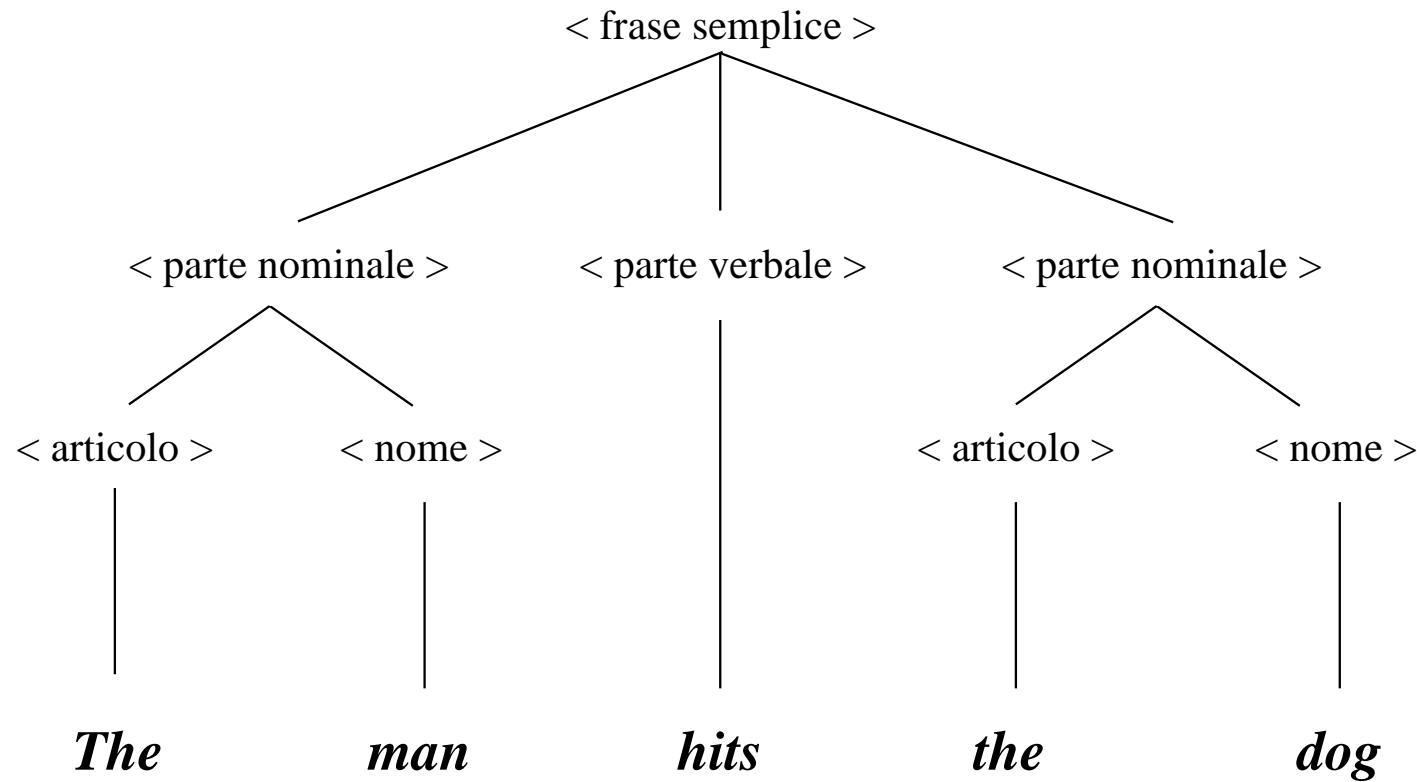
- Queste regole sono scritte in **BNF**, una notazione usata comunemente per descrivere la sintassi dei linguaggi di programmazione.
- BNF (Backus Naur Form) è un metalinguaggio.
- Nell'esempio, *<frase semplice>* è definita come una *<parte nominale>* seguita da una *<parte verbale>* seguita, a sua volta, da un'altra *<parte nominale>*.
- Ciascuna delle due occorrenze di *<parte nominale>* deve essere espansa in *<articolo>* seguito da un *<nome>*.

# Sintassi e semantica

- Scegliendo di espandere la prima occorrenza di *<articolo>* in **The**, la prima occorrenza di *<nome>* in **man**, la *<parte verbale>* in **hits**, il secondo *<articolo>* in **The** ed infine l'ultimo *<nome>* in **dog**, si dimostra che la frase ***The man hits the dog*** è una delle nostre frasi semplici.

# Sintassi e semantica

- Tutto ciò si può riassumere nel seguente  
**Albero di derivazione**



# Sintassi e semantica

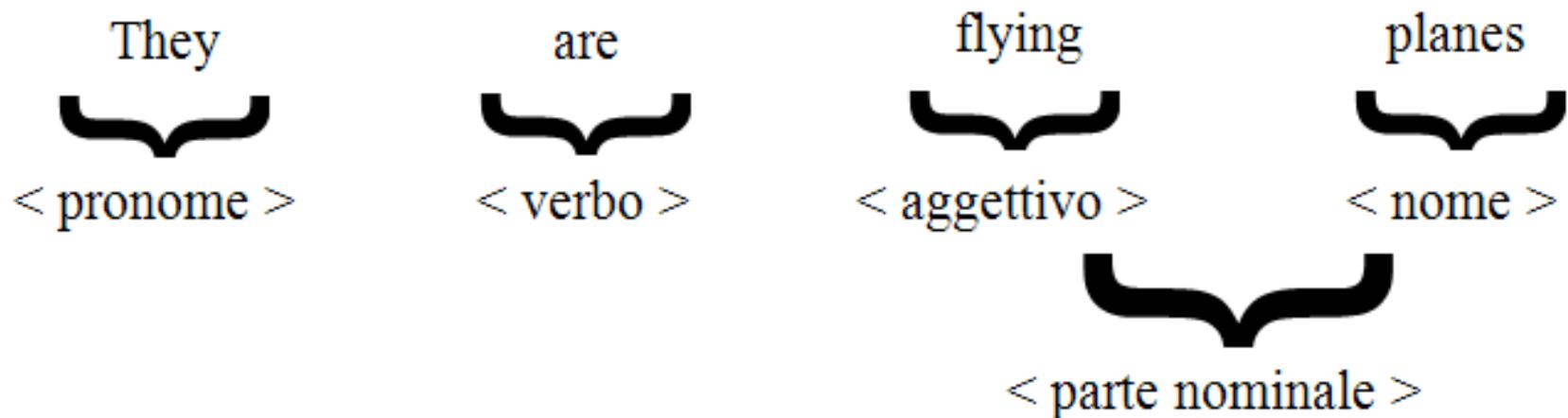
- La definizione data di <frase semplice> dà luogo a 72 diverse frasi derivabili.
  - Sebbene tutte siano sintatticamente corrette, in base alla nostra definizione, alcune non hanno un'interpretazione semantica sensata (non hanno senso compiuto).
  - Una di queste è la frase: ***The car eats the man***
  - Non necessariamente una frase corretta ha senso.

# Sintassi e semantica

- Consideriamo la seguente frase (non derivabile da <frase semplice> utilizzando le regole date in precedenza):

***They are flying planes***

- Un'analisi sintattica di questa frase è:



# Sintassi e semantica

- Quest'analisi suggerisce che **they** si riferisce a **planes** (*aerei nel cielo che volano*). Un'analisi alternativa sarebbe:

They  
  
< pronome >

are flying  
  
< verbo >

planes  
  
< nome >

e questa implicherebbe un'interpretazione semantica completamente differente, in cui **they** si riferisce a chiunque sia attualmente al controllo dei **planes** (*essi stanno pilotando gli aerei*).

# Sintassi e semantica

- Nell'esempio, l'analisi sintattica è di aiuto all'interpretazione semantica.
- Per quanto sia scelto "ad hoc" e mostri un fenomeno poco comune nel linguaggio naturale, l'esempio è illustrativo di un concetto importante in computazione.
- Una fase cruciale nel processo di compilazione di un programma è l'**analisi sintattica**, come passo essenziale per la sua **interpretazione semantica**.

# Teoria dei linguaggi formali

- Negli anni '50, N. Chomsky, linguista americano cerca di descrivere la sintassi del linguaggio naturale secondo semplici **regole di riscrittura e trasformazione**.
- Chomsky considera alcune **restrizioni** sulle regole sintattiche e classifica i linguaggi in base alle restrizioni imposte alle regole che generano tali linguaggi.
- Una classe importante di regole che generano linguaggi formali va sotto il nome di **grammatiche libere da contesto** (prive di contesto) o **Context-free grammars (C.F.)**.

# Teoria dei linguaggi formali

- L'importanza di tale classe (e dei linguaggi da essa generati) risiede nel fatto che lo sviluppo dei primi linguaggi di programmazione di alto livello (ALGOL 60), che segue di pochi anni il lavoro di Chomsky, dimostra che le grammatiche C.F. sono strumenti adeguati a descrivere la sintassi di base di molti linguaggi di programmazione.

# Esempio di linguaggio C.F.

- Un linguaggio C.F. è il linguaggio delle parentesi ben formate che comprende tutte le stringhe di parentesi aperte e chiuse bilanciate correttamente.

Esempio:

( ) è ben formata;

( ( ) ( ) ) è ben formata;

( ( ) ( ) non è ben formata.

- Questo linguaggio è fondamentale in informatica come notazione per contrassegnare il “raggio d’azione” nelle espressioni matematiche e nei linguaggi di programmazione

# Linguaggio delle parentesi ben formate

## ■ Definizione

- i) La stringa  $( )$  è ben formata;
- ii) se la stringa di simboli  $A$  è ben formata, allora lo è anche  $(A)$ ;
- iii) se le stringhe  $A$  e  $B$  sono ben formate, allora lo è anche la stringa  $AB$ .

# Linguaggio delle parentesi ben formate

- In corrispondenza di questa definizione induttiva, possiamo considerare un sistema di riscrittura che genera esattamente l'insieme delle stringhe lecite di parentesi ben formate:
  - (1)  $S \rightarrow ()$
  - (2)  $S \rightarrow (S)$
  - (3)  $S \rightarrow SS$
- Le regole di riscrittura (1), (2) e (3) sono dette *produzioni* o *regole di produzione*.
  - Stabiliscono che “data una stringa, si può formare una nuova stringa sostituendo una  $S$  o con  $()$  o con  $(S)$  o con  $SS$ ”.

# Linguaggio delle parentesi ben formate

- Confrontando la definizione di parentesi ben formate e le tre produzioni, si osserva una corrispondenza diretta tra le parti i) e ii) della definizione e le produzioni (1) e (2). La produzione (3) è apparentemente diversa dalla parte (iii) della definizione.  
Abbiamo utilizzato simboli distinti A e B nella definizione induttiva per evidenziare il fatto che le due stringhe di parentesi ben formate che consideriamo non sono necessariamente uguali.  
Nella produzione corrispondente non c'è necessità di usare simboli distinti perché in  $S \rightarrow SS$  le due S che compaiono a destra possono essere sostituite indipendentemente.
- Possiamo cioè applicare una produzione ad una delle due S (alla prima o alla seconda) indifferentemente. Il risultato non cambia.

# Linguaggio delle parentesi ben formate

- Esempio: Generazione di  $((\ ))( (\ )( \ ))$

- Primo modo:

$$S \xrightarrow{(3)} SS \xrightarrow{(2)} (S)S \xrightarrow{(1)} ((\ ))S \xrightarrow{(2)} ((\ ))(S) \xrightarrow{(3)} ((\ ))(SS) \xrightarrow{(1)} ((\ ))((\ ))S \xrightarrow{(1)} ((\ ))((\ )( \ ))$$

- La corrispondente sequenza di applicazione delle produzioni è:

$$(3) \rightarrow (2) \rightarrow (1) \rightarrow (2) \rightarrow (3) \rightarrow (1) \rightarrow (1)$$

# Linguaggio delle parentesi ben formate

- Esempio: Generazione di  $((\ ))( (\ )( \ ))$ 
  - Secondo modo:

$$S \xrightarrow{(3)} SS \xrightarrow{(2)} S(S) \xrightarrow{(3)} S(SS) \xrightarrow{(1)} S((\ )S) \xrightarrow{(1)} S((\ )( )) \xrightarrow{(2)} (S)((\ )( )) \xrightarrow{(1)} ((\ ))( (\ )( ))$$

- La corrispondente sequenza di applicazione delle produzioni è:
$$(3) \rightarrow (2) \rightarrow (3) \rightarrow (1) \rightarrow (1) \rightarrow (2) \rightarrow (1)$$
- Una sequenza di applicazioni di regole di produzione prende il nome di *derivazione*.

# MdT per linguaggio delle parentesi ben formate

- Possiamo costruire una MdT per riconoscere il linguaggio delle parentesi ben formate?
- Stampa ‘Y’ se la stringa è ben formata, ‘N’ altrimenti

# Notazione

- Nei due esempi precedenti abbiamo fatto uso della notazione  $\alpha \Rightarrow \beta$ 
  - L'interpretazione è la seguente: "da  $\alpha$  si produce direttamente  $\beta$  per effetto dell'applicazione della regola di riscrittura  $n$ ". Ad esempio
$$SS \xrightarrow{(n)} (S)S$$
  - si legge: "SS produce direttamente  $(S)S$  per effetto dell'applicazione della regola di riscrittura (2)".
- Le derivazioni precedenti (modi 1 e 2) vengono riassunte attraverso la notazione:

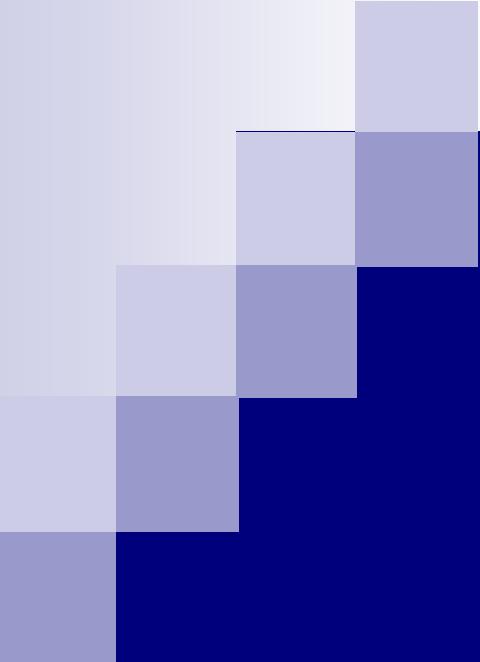
$$S \xrightarrow{*} (( ))(( )( ))$$

che leggiamo come "S produce  $(( ))(( )( ))$ "

# Notazione

- Nelle regole di produzione si è fatto uso di due tipi di simboli: caratteri che possono apparire nelle derivazioni, ma non nelle stringhe finali, detti *simboli nonterminali* o *variabili* (nell'esempio,  $S$  è il solo nonterminale) e caratteri che possono apparire nelle stringhe finali, detti *simboli terminali* (nell'esempio,  $e$  sono i simboli terminali).
- In BNF si usa la seguente notazione:

$S$	$<S>$
$\rightarrow$	$::=$



# Linguaggi di Programmazione

## Capitolo 2 – Grammatiche e Linguaggi

# Linguaggi formali e monoidi liberi

- Il concetto di **linguaggio formale** è strettamente correlato a quello di *monoide libero* (generato da un insieme).

# Definizione di Alfabeto

- Un insieme  $X$  finito e non vuoto di simboli è un *alfabeto*.
- Esempi
  - L'alfabeto latino, con l'aggiunta dei simboli di interpunzione e dello spazio bianco:  $a\ b\ c\ \dots z\ ;\ ,\ :\ \dots$
  - L'insieme delle dieci cifre arabe:  $0\ 1\ \dots\ 9$
- Con i simboli primitivi dell'alfabeto si formano le parole (es.:  $abc$ ,  $127$ ,  $casa$ ,...).

# Definizione di Parola o Stringa

- Una sequenza finita di simboli  $x_1x_2\dots x_n$ , dove ogni  $x_i$  è preso da uno stesso alfabeto  $X$  è una *parola* (su  $X$ ).
- Esempio
  - $X = \{0,1\}$ .
  - 001110 è una parola su  $X$
- Una parola è ottenuta giustapponendo o concatenando simboli (caratteri) dell'alfabeto.
- Se una stringa ha  $m$  simboli (non necessariamente distinti) allora diciamo che ha lunghezza  $m$ .

# Lunghezza di una Parola o Stringa

- La lunghezza di una stringa  $w$  è denotata con  $|w|$ .
- Le parole di lunghezza 1 sono i simboli di  $X$ .
  - Quindi 001110 è una parola di lunghezza 6

$$|001110| = 6$$

- La parola vuota (o stringa vuota), denotata con  $\lambda$ , è una stringa priva di simboli ed ha lunghezza 0

$$|\lambda| = 0$$

# Definizioni

- **Uguaglianza tra stringhe**
  - Due stringhe sono *uguali* se i loro caratteri, letti ordinatamente da sinistra a destra, coincidono.
- **$X^*$** 
  - L'insieme di tutte le stringhe di lunghezza finita sull'alfabeto  $X$  si denota con  $X^*$ .
- **Esempio**
  - Se  $X = \{0,1\}$ , allora  $X^* = \{\lambda, 0, 1, 00, 01, 10, 11, \dots\}$
- **$X^*$  ha un numero di elementi che è un infinito numerabile.**
  - Dalla definizione, segue che  $\lambda \in X^*$ , per ogni insieme  $X$ .

# Definizioni

## ■ Concatenazione o prodotto

- Sia  $\alpha \in X^*$  una stringa di lunghezza  $m$  e  $\beta \in X^*$  una stringa di lunghezza  $n$ , la **concatenazione** di  $\alpha$  e  $\beta$ , denotata con  $\alpha\beta$  o  $\alpha \cdot \beta$ , è definita come la stringa di lunghezza  $m+n$ , i cui primi  $m$  simboli costituiscono una stringa uguale a  $\alpha$  ed i cui ultimi  $n$  simboli costituiscono una stringa uguale a  $\beta$ .
- Quindi se  $\alpha = x_1x_2\dots x_m$  e  $\beta = x'_1x'_2\dots x'_n$ , si ha:

$$\alpha\beta = x_1x_2\dots x_m x'_1x'_2\dots x'_n$$

- Se  $X = \text{alfabeto latino}$

$$\alpha = \text{capo} \quad \beta = \text{stazione} \quad \alpha\beta = \text{capostazione}$$

# Operazione di concatenazione

- La concatenazione di stringhe su  $X$  è una operazione binaria su  $X^*$ :

$$\cdot : X^* \times X^* \rightarrow X^*$$

- è **associativa**:  $(\alpha\beta)\gamma = \alpha(\beta\gamma) = \alpha\beta\gamma$ ,  $\forall \alpha, \beta, \gamma \in X^*$
  - non è **commutativa**:  $\exists \alpha, \beta \in X^* : \alpha\beta \neq \beta\alpha$ 
    - *capostazione*  $\neq$  *stazione capo*
  - ha **elemento neutro**  $\lambda$ :  $\lambda\alpha = \alpha\lambda = \alpha$ ,  $\forall \alpha \in X^*$
- 
- Dunque  $(X^*, \cdot)$  è un **monoide** (non commutativo).

# Osservazione

- In base alla definizione di prodotto, ogni parola non vuota  $\alpha = x_1 x_2 \dots x_n$  si può scrivere in uno ed un solo modo come prodotto di parole di lunghezza 1, cioè di elementi di  $X$ .
- Ciò si esprime dicendo che:
  - $(X^*, \cdot)$  è il monoide libero generato dall'insieme  $X$ .

# Definizioni

- **Prefisso, Suffixo**
  - Se  $\gamma \in X^*$  è della forma  $\gamma = \alpha\beta$ , ove  $\alpha, \beta \in X^*$ , allora  $\alpha$  è un *prefisso* di  $\gamma$  e  $\beta$  è un *suffisso* di  $\gamma$ .
- **Sottostringa**
  - Se  $\delta, \beta \in X^*$  e  $\delta$  è della forma  $\delta = \alpha\beta\gamma$ , ove  $\alpha, \beta \in X^*$  allora  $\beta$  è una *sottostringa* di  $\delta$ .
- **Esempio**
  - Sia  $\gamma = 00110$ . Allora:
    - $\{\lambda, 0, 00, 001, 0011, \gamma\}$  è l'insieme dei prefissi di  $\gamma$
    - $\{\lambda, 0, 10, 110, 0110, \gamma\}$  è l'insieme dei suffissi di  $\gamma$
    - $\{\lambda, 0, 1, 00, 01, 10, 11, 001, 011, 110, 0011, 0110, \gamma\}$  è l'insieme delle sottostringhe di  $\gamma$ .

# Definizioni

## ■ Potenza di una stringa

- Data una stringa  $\alpha$  su  $X$ , la *potenza  $h$ -esima* di  $\alpha$  è definita (induttivamente) come segue:

$$\alpha^h = \begin{cases} \lambda & \text{se } h = 0 \\ \alpha\alpha^{h-1} & \text{altrimenti} \end{cases}$$

con  $h = 0, 1, 2, \dots$

- La potenza  $h$ -esima di una stringa è un caso speciale di concatenamento (in quanto la si ottiene concatenando una stringa  $h$  volte con se stessa).

# Definizioni

## ■ Potenza di un alfabeto

□ Sia  $X$  un alfabeto, poniamo:

$$1) \quad X^1 = X$$

$$2) \quad X^2 = \{x_1 x_2 \mid x_1, x_2 \in X, \ x_1 x_2 \equiv x_1 \cdot x_2\}$$

$$3) \quad X^3 = \{x_1 x_2 x_3 \mid x_1 x_2 \in X^2, \ x_3 \in X, \ x_1 x_2 x_3 \equiv x_1 x_2 \cdot x_3\}$$

..) .....

$$i) \quad X^i = \{x_1 x_2 \dots x_{i-1} x_i \mid x_1 x_2 \dots x_{i-1} \in X^{i-1}, \ x_i \in X, \ x_1 x_2 \dots x_i \equiv x_1 x_2 \dots x_{i-1} \cdot x_i\}$$

# Definizioni

## ■ Potenza di un alfabeto

- Se  $i \geq 2$  si ha:

$$X^+ = X \cup X^2 \cup \dots \cup X^i \cup \dots = \bigcup_{i=1}^{+\infty} X^i$$

- Se  $\lambda$  è la parola vuota e prendiamo un  $w \in X^+$  tale che  $w \cdot \lambda = \lambda \cdot w = w$  si ha:

$$X^* = \{\lambda\} \cup X^+$$

- Inoltre si ha:

$$X^h = \begin{cases} \{\lambda\} & \text{se } h = 0 \\ X \cdot X^{h-1} & \text{altrimenti} \end{cases}$$

# Definizioni

## ■ Linguaggio formale

- Un *linguaggio formale*  $L$  su un alfabeto  $X$  è un sottoinsieme di  $X^*$ .

$$L \subseteq X^*$$

## ■ Esempio:

- Il linguaggio delle parentesi ben formate è un linguaggio formale in quanto, denotato con  $M$  tale linguaggio, si ha:

$$M \subset \{ (, ) \}^*$$

- I linguaggi formali possono essere di natura molto diversa l'uno dall'altro.

# Esempi di linguaggi formali

- Un linguaggio di programmazione può essere costruito a partire dall'alfabeto  $X$  dei simboli sulla tastiera.
- L'insieme, finito o infinito, dei programmi ben costruiti sintatticamente (ossia, che rispettano la sintassi) costituisce un linguaggio.
- Consideriamo l'insieme dei teoremi di una teoria matematica. I teoremi sono particolari stringhe di simboli del nostro alfabeto. L'insieme dei teoremi "ben formati" rappresenta un linguaggio. Ad esempio, la stringa " $ab=ba$ " non è un teorema della teoria dei gruppi, ma della teoria dei gruppi abeliani.

# Generazione e riconoscimento di linguaggi formali

- A noi interessano i linguaggi formali da almeno due *punti di vista*
  - Descrittivo/Generativo
  - Riconoscitivo

# Generazione e riconoscimento di linguaggi formali

## ■ *Punto di vista Descrittivo/Generativo*

- Come possiamo *generare* gli elementi di un dato linguaggio  $L$ ? Un linguaggio finito può essere descritto/generato per elencazione degli elementi (se il numero non è troppo grande). Un linguaggio infinito non è elencabile. Questi sono i più interessanti perché devono essere specificati necessariamente attraverso una *proprietà* che ne caratterizza gli elementi, che ne definisce l'intensione. Tale proprietà può essere vista come una regola da seguire per generare gli elementi del linguaggio. Il vero problema è trovare la(e) regola(e) generativa(e) (di produzione) di un linguaggio. È quello che accade quando si impara un linguaggio: non è possibile memorizzare tutte le frasi del linguaggio.

# Generazione di linguaggi formali

## ■ Esempio

- Non è possibile “elencare” tutti i teoremi della teoria dei gruppi, perché sono infiniti i teoremi realizzabili combinando quelli noti.
- Un libro di teoria dei gruppi non è l’elencazione dei teoremi, ma fornisce una serie di assiomi e le regole con le quali, a partire dagli assiomi, è possibile costruire tutti i teoremi della teoria dei gruppi.
- Per descrivere la regola di produzione di un linguaggio, utilizzeremo una notazione insiemistica.

# Generazione di linguaggi formali

## ■ Esempio

- Sia  $L$  il linguaggio su  $X = \{0\}$  costituito da tutte e sole le stringhe che hanno un numero pari di 0, cioè:

$$L = \{\lambda, 00, 0000, 000000, \dots\}$$

La regola di produzione di  $L$  viene espressa come segue:

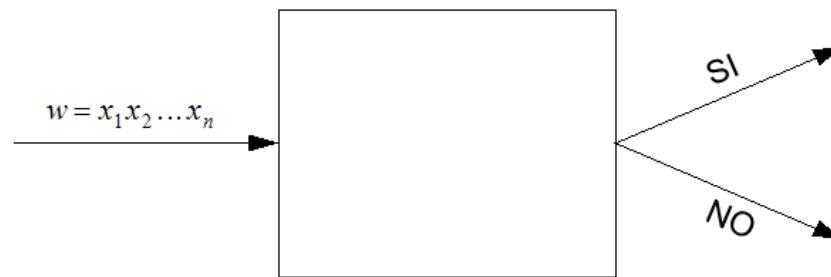
$$L = \{\lambda\} \cup \left\{ w^n \mid w = 00, n = 1, 2, \dots \right\}$$

# Generazione e riconoscimento di linguaggi formali

## ■ *Punto di vista Riconoscitivo*

- Come possiamo *riconoscere* gli elementi di un dato linguaggio  $L$ ? Questo secondo punto di vista ha come obiettivo la costruzione di “macchine” in grado di decidere/stabilire se una stringa è un elemento di  $L$  oppure no. Si intende costruire una “macchinetta” cui dare in ingresso una particolare parola e che produce una tra due possibili risposte:

$$sì \equiv ' \in L' \quad \text{e} \quad no \equiv ' \notin L'$$



# Riconoscimento di linguaggi formali

- Esempio
  - L'esecuzione di un programma errato sintatticamente viene inibita. Questo è indice dell'esistenza di una "macchinetta" che stabilisce se il programma appartiene o no all'insieme dei programmi sintatticamente ben costruiti.

# Generazione di linguaggi formali: esempio

- Sia dato l'alfabeto:  $X = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}$

Voglio generare il linguaggio  $L$  dei numeri interi relativi. Ovviamente:  $L \subseteq X^*$

Più precisamente,  $L \subset X^*$  poiché, ad esempio,

$$1++-5 \notin L$$

Non possiamo elencare gli elementi di  $L$ . Cerchiamo dunque una serie di regole mediante le quali è possibile produrre tutti e soli gli elementi di  $L$ .

Assumiamo, per semplicità, che un numero relativo sia costituito da una serie di cifre precedute da + o -.

# Generazione di linguaggi formali: esempio

- Adottiamo la BNF per descrivere le produzioni:

$$\langle S \rangle ::= + \langle I \rangle | - \langle I \rangle$$
$$\langle I \rangle ::= \langle D \rangle | \langle I \rangle \langle D \rangle$$
$$\langle D \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$$

- Queste regole generano tutti gli interi relativi purché partiamo dal simbolo nonterminale  $S$ .
- $I$  è il simbolo nonterminale (da ora in poi, talvolta abbreviato in  $NT$ ), anche detto **categoria sintattica**, che sta ad indicare (e da cui si genera) la classe dei numeri interi.
- $I$  è definito ricorsivamente o come una cifra oppure come un intero seguito da una cifra.
- Ogni intero relativo è generato da queste regole e niente che non sia un intero relativo può essere generato da queste regole.

# Generazione di linguaggi formali: esempio

- Generazione ad albero:
  - proviamo a generare l'intero relativo -375
- Tale albero prende il nome di *albero di derivazione*.

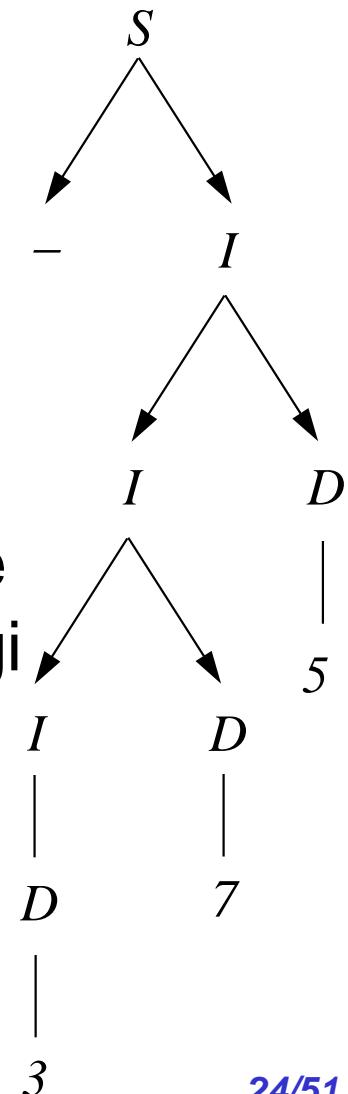
\*

- $S \Rightarrow -375 \Leftrightarrow -375 \in L$
- Nella notazione vista per il linguaggio delle parentesi ben formate, tipica per i linguaggi formali, la grammatica diventa:

$$S \rightarrow +I | -I$$

$$I \rightarrow D | ID$$

$$D \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$$



# Grammatiche generative

- Dagli esempi di linguaggi visti, possiamo trarre le seguenti conclusioni. Per generare un linguaggio sono necessari:
  - un insieme  $X$  di simboli primitivi con cui si formano le parole del linguaggio, detto *alfabeto dei simboli terminali* o *alfabeto terminale*;
  - un insieme  $V$  di simboli ausiliari o variabili con cui si identificano le categorie sintattiche del linguaggio, detto *alfabeto dei simboli nonterminali* (*ausiliari*) o *alfabeto nonterminale* o *alfabeto delle variabili*;
  - un simbolo speciale  $S$ , scelto tra i nonterminali, da cui far partire la generazione delle parole del linguaggio. Tale simbolo è detto *assioma* o *scopo* o *simbolo distintivo* o *simbolo di partenza* o *simbolo iniziale*;
  - un insieme  $P$  di *produzioni*, espresse in un formalismo quali regole di riscrittura, BNF ( $a ::= b$ ), carte sintattiche, ...

# Definizione di Grammatica generativa o a struttura di frase

- Una *grammatica generativa* o a struttura di frase  $G$  è una quadrupla

$$G = (X, V, S, P)$$

ove:

- $X$  è l'*alfabeto terminale* per la grammatica;
- $V$  è l'*alfabeto nonterminale* o delle variabili per la grammatica;
- $S$  è il *simbolo di partenza* per la grammatica;
- $P$  è l'insieme delle *produzioni* della grammatica ed inoltre valgono le seguenti condizioni:

$$X \cap V = \emptyset \quad \text{e} \quad S \in V$$

# Definizione di Produzione

- Una *produzione* è una coppia  $(v, w)$ ,  
ove  $v \in (X \cup V)^+$  e  $v$  contiene un  $NT \Leftrightarrow v \in (X \cup V)^* V (X \cup V)^*$   
 $w \in (X \cup V)^*$  ( $w$  può essere anche  $\lambda$ ).

*Un elemento  $(v, w)$  di  $P$  viene comunemente scritto nella forma:*

$$v \rightarrow w$$

Una produzione deve, in qualche modo, riscrivere un  $NT$ .

# Definizione di Produzione

- Per convenzione, gli elementi di  $X$  sono rappresentati di solito con lettere minuscole (con o senza pedici) e di solito sono le prime lettere dell'alfabeto) o cifre ed operatori (connettivi), mentre gli elementi di  $V$  sono rappresentati con lettere maiuscole (con o senza pedici) o con stringhe delimitate dalle parentesi angolari “ $<$ ” e “ $>$ ”.
- La notazione  $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_k$  è impiegata come abbreviazione della seguente:

$$\alpha \rightarrow \beta_1$$

$$\alpha \rightarrow \beta_2$$

...

$$\alpha \rightarrow \beta_k$$

# Esempi di grammatiche

- La grammatica per il linguaggio delle parentesi ben formate

$$G_1 = (\{ (, ) \}, \quad \{S\}, \quad S, \quad \{S \rightarrow ( ), \quad S \rightarrow (S), \quad S \rightarrow SS\})$$

- La grammatica per il linguaggio dei numeri interi relativi

$$G_2 = (\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}, \quad \{S, I, D\}, \quad S, \\ \{S \rightarrow +I, \quad S \rightarrow -I, \quad I \rightarrow D, \quad I \rightarrow ID, \quad D \rightarrow 0, \quad D \rightarrow 1, \dots, \quad D \rightarrow 9\})$$

# Definizione di derivazione o produzione diretta

- Sia  $G = (X, V, S, P)$  una grammatica e siano  $y$  e  $z$  due stringhe finite di simboli in  $X \cup V$  (stringhe di terminali e nonterminali) tali che:

$y = \gamma\alpha\delta$  e  $z = \gamma\beta\delta$ , ove  $\gamma \in (X \cup V)^+$ ,  $z \in (X \cup V)^*$ ,  
 $\beta, \gamma, \delta \in (X \cup V)^*$        $\alpha \in (X \cup V)^+$  e  $\alpha$  contiene un NT

1) Scriviamo

$$y \Rightarrow z$$

e diciamo che  $y$  *produce direttamente*  $z$  o che  $z$  è *derivata direttamente* da  $y$  se:

$$\alpha \rightarrow \beta \in P$$

ossia se esiste in  $G$  una produzione  $\alpha \rightarrow \beta$

# Definizione di derivazione o produzione diretta

2) Scriviamo

$$y \xrightarrow{*} z$$

e diciamo che  $y$  *produce*  $z$  o che  $z$  è *derivabile* da  $y$  se  $y = z$  o esiste una sequenza di stringhe

$w_1, w_2, \dots, w_n$ , con  $w_1, w_2, \dots, w_{n-1} \in (X \cup V)^+$ ,  $w_n \in (X \cup V)^*$

$w_1 = y$  e  $w_n = z$  tali che  $\forall i, i = 1, 2, \dots, n-1 : w_i \xrightarrow{G} w_{i+1}$   
( $w_i$  produce direttamente  $w_{i+1}$ ), cioè:

$$y \xrightarrow{*} z \iff \begin{cases} y = z \\ \text{oppure} \\ w_1 = y \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \dots \Rightarrow w_{n-1} \Rightarrow w_n = z \end{cases}$$

# Osservazione

- La nozione di derivazione diretta stabilisce una relazione binaria in  $(X \cup V)^*$ .  
Date due stringhe  $y$  e  $z$ , il simbolo  $\Rightarrow$  può esserci o meno; dipende dall'esistenza di una produzione.  
Allora possiamo anche definire una composizione di relazioni:

$$y \xrightarrow{2} z \stackrel{\text{def}}{\iff} \exists w: y \Rightarrow w \text{ e } w \Rightarrow z$$

dove 2 è il numero di trascrizioni necessarie per passare da  $y$  a  $z$  (ossia, la *lunghezza della derivazione*).

# Osservazione

- Da ciò si ha:  $\xrightarrow{*} = I \cup \xrightarrow{2} \cup \xrightarrow{3} \cup \dots$

ove  $I$  è la relazione identica e  $\xrightarrow{n}$  indica la composizione della relazione  $\Rightarrow$   $n$  volte con se stessa.

- \*
  - $\xrightarrow{*}$  è la **chiusura riflessiva e transitiva** della relazione di derivazione diretta;
  - +
  - $\xrightarrow{+}$  è la **chiusura transitiva** della stessa relazione.

# Definizione di linguaggio generato da una grammatica

- Sia  $G = (X, V, S, P)$  una grammatica. Il *linguaggio generato da G*, denotato con  $L(G)$ , è l'insieme delle stringhe di terminali derivabili dal simbolo di partenza  $S$ .

$$L(G) = \left\{ w \in X^* \mid S \xrightarrow[G]{*} w \right\}$$

- Sono, dunque, stringhe di  $L(G)$  le stringhe che:
  - consistono di soli terminali;
  - possono essere derivate da  $S$  in  $G$ .

# Definizione di forma di frase

- Sia  $G = (X, V, S, P)$  una grammatica. Una stringa  $w$ ,  $w \in (X \cup V)^*$ , è una *forma di frase* di  $G$  se:

$$S \xrightarrow[G]{*} w$$

- Alle forme di frase si applicano le stesse definizioni (es.: potenza) e gli stessi operatori (es.: concatenazione) dati per le stringhe.
- **Proposizione:**
  - Data una grammatica  $G = (X, V, S, P)$ ,  $L(G)$  è l'insieme delle forme di frase terminali (o *frasi*) di  $G$ .

# Definizione di grammatiche equivalenti

- Due grammatiche  $G$  e  $G'$  si dicono *equivalenti* se generano lo stesso linguaggio, ossia se

$$L(G) = L(G')$$

## Esempio

- Sia  $G = (X, V, S, P)$ , ove

$$X = \{a, b\}, \quad V = \{S\}, \quad P = \left\{ S \xrightarrow{(1)} aSb, S \xrightarrow{(2)} ab \right\}$$

Determiniamo  $L(G)$ .

$ab \in L(G)$  poiché  $S \xrightarrow{(2)} ab$

Se numeriamo le produzioni, possiamo indicare la produzione usata immediatamente al di sotto del simbolo  $\Rightarrow$ .

$\Rightarrow \equiv$  ho applicato la produzione n

$(n)$   
 $k$

$y \xrightarrow{k} z \equiv y$  produce  $z$  in  $k$  passi, dove  $k$ =lunghezza della derivazione

## Esempio

- $a^2b^2 \in L(G)$  poiché  $S \xrightarrow{(1)} aSb \xrightarrow{(2)} a^2b^2$
- $a^3b^3 \in L(G)$  poiché  $S \xrightarrow{3} a^3b^3$
- .....
- Inoltre, qualsiasi derivazione da  $S$  in  $G$  produce frasi del tipo  $a^n b^n$ .
  - Dunque  $L(G) \subseteq \{a^n b^n \mid n > 0\}$  e quindi

$$L(G) = \{a^n b^n \mid n > 0\}$$

# Notazione

- Per rendere più concisa la descrizione di una grammatica, spesso ci limiteremo ad elencarne le produzioni, quando sia chiaro quale sia il simbolo di partenza e quali siano i terminali ed i nonterminali.
- Inoltre, le produzioni con la stessa parte sinistra vengono accorpate attraverso l'uso del simbolo “|” (preso a prestito dalla BNF).
- Infine, ometteremo l'indicazione della grammatica dalla simbologia di derivazione e derivazione diretta quando sia chiaro dal contesto a quale grammatica si fa riferimento.

## Esempio

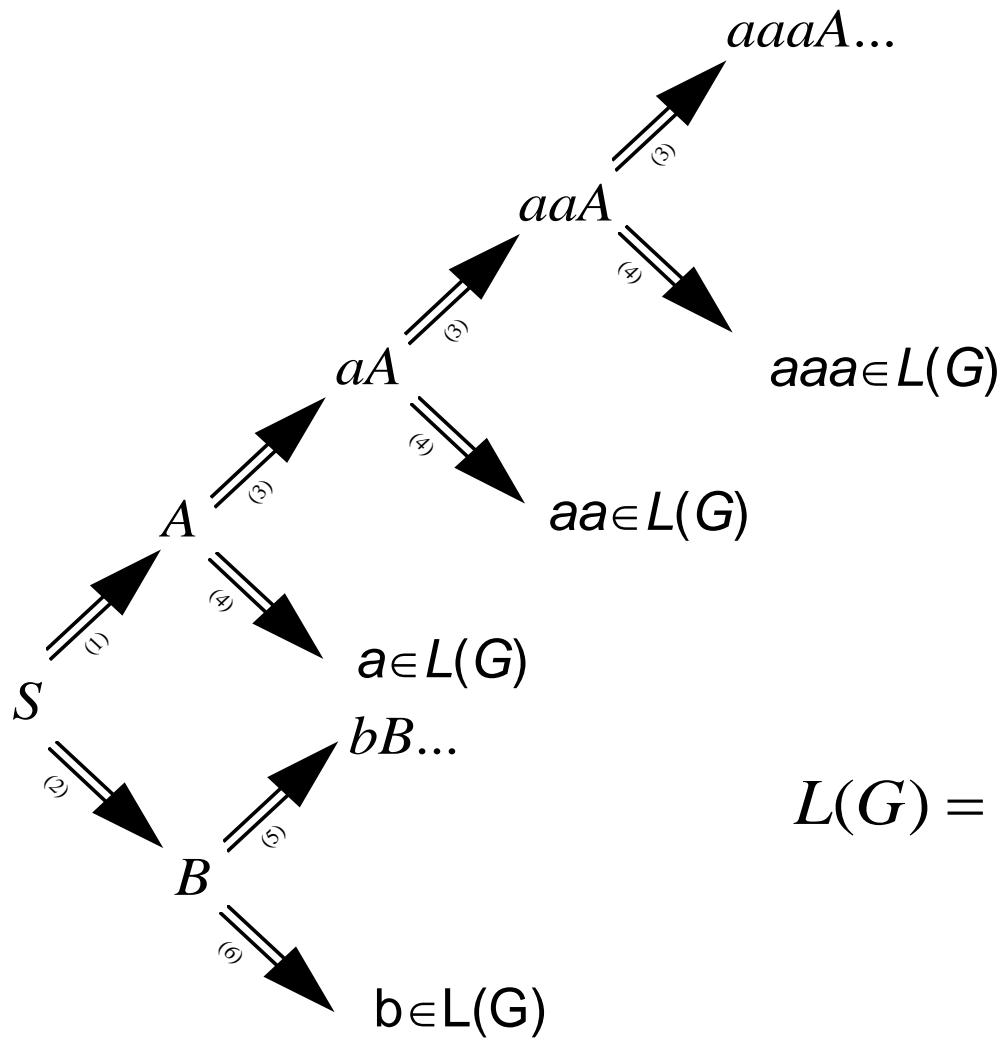
- Sia data la seguente grammatica:

$$S \rightarrow A|B, \quad A \rightarrow aA \mid a, \quad B \rightarrow bB \mid b$$

Determinare  $L(G)$ .

Non sappiamo se applicare  $S \rightarrow A$  oppure  $S \rightarrow B$  inizialmente. I meccanismi di costruzione di un linguaggio sono generalmente **non deterministici**, poiché può non essere univoca la sostituzione da operare ad una forma di frase se uno stesso NT si trova a sinistra di 2 o più produzioni, come illustrato nella figura seguente.

# Esempio



$$L(G) = \{a^n \mid n > 0\} \cup \{b^n \mid n > 0\}$$

# Osservazione

- Dunque, una grammatica è uno *strumento generativo* di un linguaggio perché, data una qualsiasi parola di quel linguaggio, possiamo risalire mediante le produzioni al simbolo di partenza della grammatica.
- Viceversa, dato il simbolo di partenza di una grammatica, seguendo uno qualsiasi dei cammini dell'albero di derivazione, si produce una parola “valida” del linguaggio.

# Osservazione

- In generale, dato un linguaggio  $L$  ed una grammatica  $G$ , non esiste un algoritmo in grado di dimostrare che la grammatica genera il linguaggio, ossia che  $L = L(G)$ .  
Più specificamente, non esiste un algoritmo che stabilisce se una data stringa è generata o no dalla grammatica presa in considerazione.
- Tutto ciò si riassume nella seguente **proposizione**:
  - Il problema di dimostrare la correttezza di una grammatica non è risolubile algoritmicamente, in generale.

## Osservazione

- In molti casi importanti, però, è possibile dimostrare per induzione che una particolare grammatica genera proprio un particolare linguaggio.
- Queste dimostrazioni ci consentono di stabilire se, data una grammatica  $G$  ed un linguaggio  $L$ , risulta:
  - $w \in L(G) \Rightarrow w \in L$  cioè  $L(G) \subseteq L$
  - $w \in L \Rightarrow w \in L(G)$  cioè  $L \subseteq L(G)$

## Osservazione

- In molti casi importanti, però, è possibile dimostrare per induzione che una particolare grammatica genera proprio un particolare linguaggio.
- Queste dimostrazioni ci consentono di stabilire se, data una grammatica  $G$  ed un linguaggio  $L$ , risulta:

- $w \in L(G) \Rightarrow w \in L$  cioè  $L(G) \subseteq L$
- $w \in L \Rightarrow w \in L(G)$  cioè  $L \subseteq L(G)$

La grammatica  $G$  genera solo stringhe appartenenti al linguaggio  $L$ .

# Osservazione

- In molti casi importanti, però, è possibile dimostrare per induzione che una particolare grammatica genera proprio un particolare linguaggio.
- Queste dimostrazioni ci consentono di stabilire se, data una grammatica  $G$  ed un linguaggio  $L$ , risulta:
  - $\square w \in L(G) \Rightarrow w \in L$  cioè  $L(G) \subseteq L$
  - $\square w \in L \Rightarrow w \in L(G)$  cioè  $L \subseteq L(G)$



Il linguaggio  $L$  comprende solo parole generabili dalla grammatica  $G$ .

# Principio di induzione

- Sia  $n_0$  un intero e sia  $P=P(n)$  un enunciato che ha senso per ogni intero maggiore o uguale ad  $n_0$ . Se:
  - $P(n_0)$  è vero
  - Per ogni  $n > n_0$ ,  $P(n-1)$  vero implica  $P(n)$  veroallora  $P(n)$  è vero per tutti gli  $n$  maggiori o uguali ad  $n_0$

## Esercizi

- Determinare una grammatica che genera il seguente linguaggio:

$$L = \{a^n b^n \mid n > 0\}$$

e dimostrare questo risultato.

- Che tipo di grammatica genera  $L$  ?

Soluzione esercizio

## Esercizi

- Determinare una grammatica che genera il seguente linguaggio:

$$L = \{a^n b^{2n} \mid n > 0\}$$

e dimostrare questo risultato.

- Di che tipo è la grammatica che genera  $L$  ?

Soluzione esercizio

# Esercizi

- Sia data la seguente grammatica:

$$G = (X, V, S, P)$$

$$X = \{0, 1\} \quad V = \{S, A, B\}$$

$$P = \left\{ S \xrightarrow{(1)} 0B \mid 1A, \quad A \xrightarrow{(2)} 0 \mid 0S \mid 1AA, \quad B \xrightarrow{(3)} 1 \mid 1S \mid 0BB \right. \\ \left. \xrightarrow{(4)} 0 \mid 0S \mid 1AA, \quad \xrightarrow{(5)} 1 \mid 1S \mid 0BB \right. \\ \left. \xrightarrow{(6)} 1 \mid 1S \mid 0BB \right. \\ \left. \xrightarrow{(7)} 1 \mid 1S \mid 0BB \right. \\ \left. \xrightarrow{(8)} 1 \mid 1S \mid 0BB \right\}$$

- Determinare il linguaggio generato da  $G$ .

Soluzione esercizio

# Esercizi

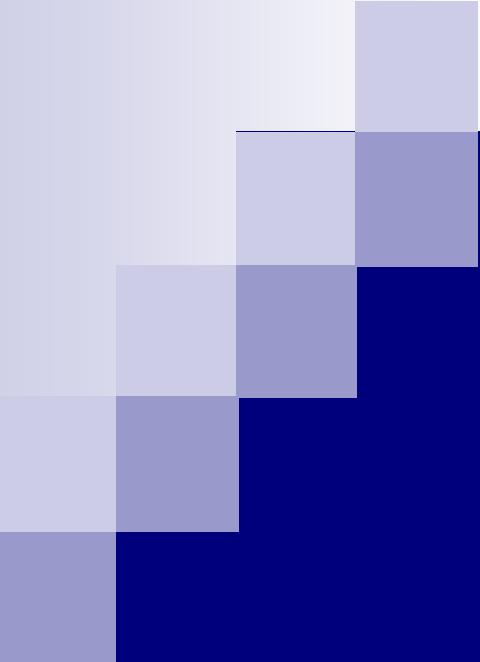
- Dimostrare per induzione che il linguaggio  $L$  generato dalla seguente grammatica è vuoto:

$$G = (X, V, S, P)$$

$$X = \{a, b, c\} \quad V = \{S, A, B\}$$

$$P = \left\{ S \xrightarrow{(1)} aBS \mid bA, \quad S \xrightarrow{(2)} aB \xrightarrow{(3)} Ac \mid a, \quad S \xrightarrow{(4)} bA \xrightarrow{(5)} S \mid Ba \xrightarrow{(6)} \right\}$$

Soluzione esercizio



# Linguaggi di Programmazione

Capitolo 3 – Linguaggi liberi da  
contesto e linguaggi dipendenti da  
contesto

# Definizione di grammatica libera da contesto

- Una grammatica  $G = (X, V, S, P)$  è *libera da contesto* (o *context-free* - C.F.) se, per ogni produzione ,  $v \rightarrow w$   $v$  è un nonterminale.

$G$  è libera da contesto  $\overset{def}{\iff} \forall v \rightarrow w \in P : v \in V$

# Definizione di linguaggio libero da contesto

- Un linguaggio  $L$  su un alfabeto  $X$  è *libero da contesto* se può essere generato da una grammatica libera da contesto.

*def*

$L$  libero da contesto  $\Leftrightarrow \exists G$  libera da contesto tale che  $L(G) = L$ .

- Se si ha una grammatica C.F. che genera  $L$ , non è detto che non esista un'altra grammatica che generi lo stesso linguaggio.

# Linguaggi liberi da contesto

- La maggior parte dei linguaggi di programmazione ricade nella classe dei linguaggi C.F.
- Il termine C.F. nasce dal fatto che la sostituzione di un  $NT$  non è condizionata dal contesto - ossia dai caratteri adiacenti - in cui compare.
- Un  $NT A$  in una forma di frase può sempre essere sostituito usando una produzione del tipo  $A \rightarrow \beta$ . La sostituzione è sempre valida.
- Viceversa, se  $L = L(G)$  e  $G$  non è C.F., non possiamo concludere che  $L$  non è C.F. perché non possiamo escludere che esista una grammatica C.F.  $G'$  per cui  $L=L(G')$ .

# Esempi di linguaggi C.F.

- Il linguaggio delle parentesi ben formate
- Il linguaggio dei numeri interi relativi
- Il linguaggio  $L = \{a^n b^n \mid n > 0\}$
- Il linguaggio delle stringhe con ugual numero di 0 e di 1.
- Il linguaggio  $L = \{a^n b^{2n} \mid n > 0\}$

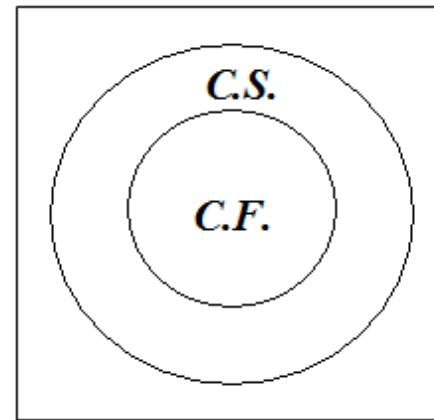
# Definizione di grammatica dipendente da contesto

- Una grammatica  $G = (X, V, S, P)$  è *dipendente da contesto* (o *context-sensitive* - C.S.) se ogni produzione è in una delle seguenti forme:
  - (1)  $yAz \rightarrow ywz$  con  $A \in V$ ,  $y, z \in (X \cup V)^*$ ,  $w \in (X \cup V)^+$   
che si legge: “ $A$  può essere sostituita con  $w$  nel contesto  $y-z$ ” (contesto sinistro  $y$  e contesto destro  $z$ ).
  - (2)  $S \rightarrow \lambda$  purché  $S$  non compaia nella parte destra di alcuna produzione.

# Definizione di linguaggio dipendente da contesto

- Un linguaggio  $L$  è *dipendente da contesto* se può essere generato da una grammatica dipendente da contesto.

# Relazione tra linguaggi C.F. e C.S.



- Tale relazione sussiste perché le regole di produzione C.S. sono una generalizzazione di quelle C.F.
- Le produzioni C.F. sono un caso particolare delle produzioni di tipo (1) delle grammatiche C.S., che si verifica quando:

$y = z = \lambda$     contesto destro e sinistro  
equivalenti alla parola vuota (c'è una eccezione).

# Eccezione

- Le produzioni C.F. sono un caso particolare delle produzioni di tipo (1) delle grammatiche C.S., che si verifica quando contesto destro e sinistro sono equivalenti alla parola vuota.
  - Osservando con attenzione la definizione di grammatica C.F. si nota che,  $w \in (X \cup V)^*$  mentre nella definizione di grammatica C.S.  $w \in (X \cup V)^+$ . Dunque le grammatiche C.F. ammettono produzioni del tipo,  $A \rightarrow \lambda$  con  $A$  che può anche non essere il simbolo iniziale, mentre le grammatiche C.S. non ammettono tali produzioni.
  - Chiameremo tutte le produzioni del tipo  *$\lambda$ -produzioni* o  *$\lambda$ -regole*.

# Esempi

- Esempi di produzioni contestuali
    - $bC \rightarrow bc$
    - $baACbA \rightarrow baAabA$
  - Esempio di grammatica contestuale
    - $S \rightarrow \lambda \mid bC$
    - $bC \rightarrow bc$
  - Esempio di produzione non C.S. (né C.F.)
    - $CB \rightarrow BC$
- $S \rightarrow \lambda$  è una produzione C.S. ed  
 $S$  non compare a destra  
di un'altra produzione.
- non è né C.S. né C.F. È una  
produzione **monotona** perché del tipo  
 $v \rightarrow w$  con  $|v| \leq |w|$

# Definizione di grammatica monotona

- Una grammatica  $G = (X, V, S, P)$  è *monotona* se ogni sua produzione è monotona, cioè se

$$\forall v \rightarrow w \in P : |v| \leq |w|$$

# Definizione di linguaggio monotono

- Un linguaggio  $L$  è *monotono* se può essere generato da una grammatica monotona.

# Esempio

- Produzioni monotone
  - $AB \rightarrow CDEF$
  - $CB \rightarrow BC$
- Una produzione monotona può essere sostituita da una sequenza di produzioni contestuali senza alterare il linguaggio generato.
  - $AB \rightarrow CDEF$  può essere sostituita dalle seguenti produzioni contestuali:
    - $AB \rightarrow AG$
    - $AG \rightarrow CG$
    - $CG \rightarrow CDEF$

# Esempio

## ■ Produzioni monotone

□  $CB \rightarrow BC$  può essere sostituita dalle seguenti produzioni contestuali:

- $CB \rightarrow XB$
  - $XB \rightarrow XC$
  - $XC \rightarrow BC$
- oppure
- $CB \rightarrow X_1B$
  - $X_1B \rightarrow X_1X_2$
  - $X_1X_2 \rightarrow X_1C$
  - $X_1C \rightarrow BC$

# Proposizione

- La classe dei linguaggi contestuali coincide con la classe dei linguaggi monotoni.
- Tale proposizione deriva immediatamente dal teorema che segue

# Teorema

- Sia  $G$  una grammatica monotona, cioè tale che ogni produzione di  $G$  è della forma  $v \rightarrow w$ , con  $|v| \leq |w|$ , eccetto che ci può essere un'unica  $\lambda$ -produzione  $S \rightarrow \lambda$  se  $S$  non appare alla destra di una produzione. Esiste allora una grammatica C.S.  $G'$  equivalente a  $G$ , cioè tale che  $L(G) = L(G')$ .
- Il teorema precedente può essere enunciato anche nella seguente forma:

# Teorema (seconda formulazione)

- Un linguaggio  $L$  è dipendente da contesto se e solo se esiste una grammatica  $G$  tale che  $L = L(G)$  ed ogni produzione di  $G$  nella forma  $u \rightarrow v$  ha la proprietà che:  $0 < |u| \leq |v|$ , con una sola eccezione: se  $\lambda \in L(G)$  allora  $S \rightarrow \lambda$  è una produzione di  $G$  ed in tal caso  $S$  non può comparire nella parte destra di altre produzioni.

## Dimostrazione

# Dimostrazione

## ■ $\Rightarrow)$ Banale.

Se  $L$  è dipendente da contesto allora, per definizione, esiste  $G$  dipendente da contesto tale che  $L = L(G)$ .

$$L \text{ è C.S.} \Leftrightarrow \exists G \text{ C.S. : } L = L(G).$$

Allora ogni produzione di  $G$  è in una delle due forme:

- (1)  $yAz \rightarrow ywz$  con  $A \in V$ ,  $y, z \in (X \cup V)^*$ ,  $w \in (X \cup V)^+$
- (2)  $S \rightarrow \lambda$  con  $S$  che non compare nella parte destra di alcuna produzione.

Dunque, ogni produzione di  $G$  verifica la condizione  $u \rightarrow v$ , con  $0 < |u| \leq |v|$ , se è del tipo (1), mentre se è del tipo (2) con  $S$  che non compare a destra di alcuna produzione, ricade nell'eccezione. Pertanto  $G$  è la grammatica cercata.

# Dimostrazione

- $\Leftrightarrow$ )  
Sia  $G$  una grammatica in cui ogni produzione è nella forma  $u \rightarrow v$ , con  $0 < |u| \leq |v|$ . Senza perdere la generalità della dimostrazione, possiamo supporre che una generica produzione di  $G$  abbia il formato:  
 $A_1 A_2 \dots A_m \rightarrow B_1 B_2 \dots B_n \quad m \leq n \quad A_i \in V, \quad i = 1, 2, \dots, m$   
È legittimo fare questa assunzione in quanto, se  $A_j$  fosse un terminale potremmo sostituirlo nella produzione con un nuovo nonterminale ed aggiungere la nuova produzione  $A'_j \rightarrow A_j$ .  
Denotiamo con  $C_1, C_2, \dots, C_m$   $m$  simboli nonterminali non presenti in  $G$ .

# Dimostrazione

- Utilizziamo le  $C_k$ ,  $k = 1, 2, \dots, m$  per costruire nuove regole contestuali che riscrivono la stringa  $A_1A_2\dots A_m$  con  $B_1B_2\dots B_n$ .

$$A_1A_2\dots A_m \rightarrow C_1A_2\dots A_m$$

$$C_1A_2\dots A_m \rightarrow C_1C_2A_3\dots A_m$$

...

$$C_1C_2\dots C_{m-1}A_m \rightarrow C_1C_2\dots C_{m-1}C_mB_{m+1}\dots B_n$$

$$C_1C_2\dots C_{m-1}C_mB_{m+1}\dots B_n \rightarrow C_1\dots C_{m-1}B_mB_{m+1}\dots B_n$$

...

$$C_1B_2\dots B_n \rightarrow B_1B_2\dots B_n$$

$\left. \begin{array}{c} \\ \\ \dots \\ \\ \end{array} \right\} 2m$   
produzioni

La nuova grammatica che incorpora queste produzioni è contestuale e si può dimostrare che  $L(G)=L(G')$ .

Lasciamo per esercizio tale dimostrazione.

c.v.d.

# Esempio

$$\underbrace{ABC}_{m=3} \rightarrow \underbrace{DEFGH}_{n=5}$$

6 produzioni contestuali

$$ABC \rightarrow C_1BC$$

$$C_1BC \rightarrow C_1C_2C$$

$$C_1C_2C \rightarrow C_1C_2C_3GH$$

$$C_1C_2C_3GH \rightarrow C_1C_2FGH$$

$$C_1C_2FGH \rightarrow C_1EFGH$$

$$C_1EFGH \rightarrow DEFGH$$

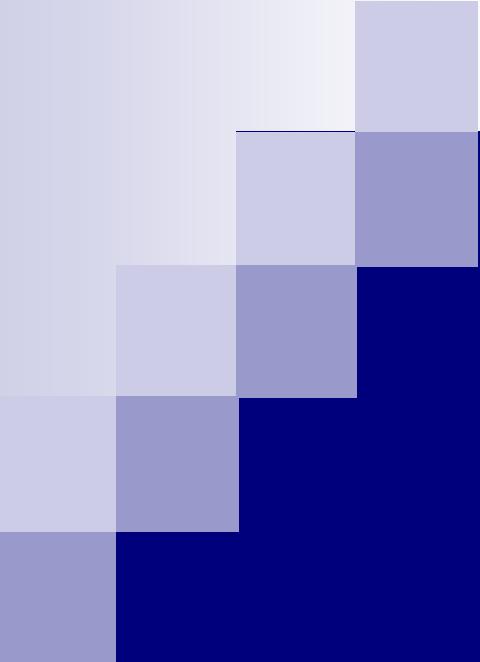
# Esercizio

- Consideriamo il linguaggio:

$$L = \{a^n b^n c^n \mid n > 0\}$$

Determiniamo una grammatica che genera tale linguaggio.

Soluzione esercizio



# Linguaggi di Programmazione

## Capitolo 4 – Linguaggi liberi da contesto

# Alberi di derivazione

- Le derivazioni in una grammatica libera da contesto possono essere rappresentate da *alberi* (detti *alberi di derivazione*).

La sequenza delle regole sintattiche utilizzate per generare una stringa  $w$  da una grammatica  $G$  di simbolo iniziale  $S$  definisce la *struttura* di  $w$ , che dunque potrebbe essere rappresentata da una delle derivazioni  $s \xrightarrow{*} w$ . Al fine di disporre di una rappresentazione univoca, si preferisce ricorrere agli alberi di derivazione.

# Definizioni preliminari

- Un **Albero** è un grafo orientato, aciclico, connesso e avente al massimo un arco entrante in ciascun nodo.
- La definizione rigorosa di “albero” è la seguente:
  - Un *albero*  $T$  è un insieme finito, non vuoto di vertici (*nodi*) tali che:
    - esiste un vertice speciale, detto *radice* dell’albero;
    - esiste una partizione  $T_1, T_2, \dots, T_m$ , con  $m \geq 0$ , degli altri vertici, tale che esista un ordinamento  $T_1, T_2, \dots, T_m$ , e ogni sottoinsieme di vertici  $T_i$ ,  $1 \leq i \leq m$ , sia a sua volta un albero.
  - Questa definizione è di tipo ricorsivo, ma non è circolare in quanto ogni sottoalbero  $T_i$  contiene meno vertici dell’albero  $T$ .  
 $T_1, T_2, \dots, T_m$  sono detti *sottoalberi* di  $T$ . I vertici privi di discendenti sono le *foglie* dell’albero, gli altri sono i *nodi interni*.  
La stringa dei simboli che etichettano le foglie di un albero  $T$ , letti nell’ordine da sinistra a destra, prende il nome di **frontiera** di  $T$ .

# Definizione di partizione

- Gli insiemi (non vuoti)  $T_1, T_2, \dots, T_m$ ,  $m \geq 0$ , costituiscono una *partizione* di un insieme (non vuoto)  $T$  se:
  - i)  $T_i \cap T_j = \emptyset$  per  $i \neq j$ ,  $i, j = 1, 2, \dots, m$
  - ii)  $\bigcup_{i=1}^m T_i = T$

# Definizione di albero di derivazione

- Sia  $G = (X, V, S, P)$  una grammatica C.F. e  $w \in X^*$  una stringa derivabile da  $S$  in  $G$ ,  $S \xrightarrow{*} w$ .

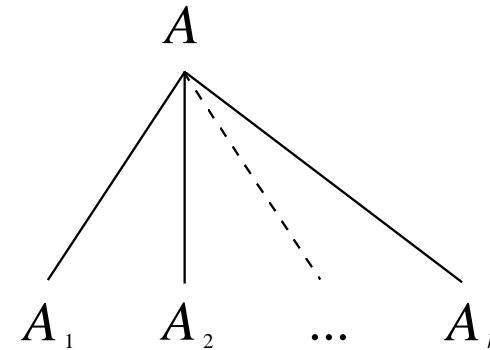
Dicesi ***albero di derivazione*** l'albero  $T$  avente le seguenti proprietà:

- (1) la radice è etichettata con il simbolo iniziale  $S$ ;
- (2) ogni nodo interno (nodo non foglia) è etichettato con un simbolo di  $V$  (un nonterminale);
- (3) ogni nodo foglia è etichettato con un simbolo di  $X$  (un terminale) o con  $\lambda$ ;

# Definizione di albero di derivazione

- (4) se un nodo  $N$  è etichettato con  $A$ , ed  $N$  ha  $k$  discendenti diretti  $N_1, N_2, \dots, N_k$  etichettati con i simboli  $A_1, A_2, \dots, A_k$ , rispettivamente, allora la produzione:

$$A \rightarrow A_1 A_2 \dots A_k$$



deve appartenere a  $P$ ;

- (5) la stringa  $w$  può essere ottenuta leggendo (e concatenando) le foglie dell'albero da sinistra a destra.

# Definizioni

- Lunghezza di un cammino
  - Dato un albero di derivazione, la *lunghezza di un cammino* dalla radice ad una foglia è data dal numero di nonterminali su quel cammino.
- Altezza o profondità
  - L'*altezza* di un albero è data dalla lunghezza del suo cammino più lungo.

# Osservazione

- Un albero di derivazione non impone alcun ordine sull'applicazione delle produzioni in una derivazione. In altri termini, *data una derivazione, esiste uno ed un solo albero di derivazione* che la rappresenta, mentre *un albero di derivazione rappresenta in generale più derivazioni* (in funzione dell'ordine col quale si espandono i nonterminali).

Esempio

# Principio di sostituzione di sottoalberi

## ■ Definizione di derivazione destra (sinistra)

- Data una grammatica  $G = (X, V, S, P)$ , diremo che una *derivazione*  $S \Rightarrow^* w$ , ove:

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_n = w$$

$$w_i = y_i A z_i, \quad w_{i+1} = y_i w_i z_i, \quad i = 1, 2, \dots, n-1$$

è *destra (sinistra)* se, per ogni  $i$ ,  $i = 1, 2, \dots, n-1$ , risulta:

$$z_i \in X^* \quad (y_i \in X^*)$$

In altre parole in una *derivazione destra (sinistra)* ad ogni passo si espande il nonterminale posto più a destra (sinistra).

# Esempio

- Riconsideriamo la grammatica che genera tutte e sole le stringhe che hanno un ugual numero di 1 e di 0.

$$S \rightarrow 0B|1A$$

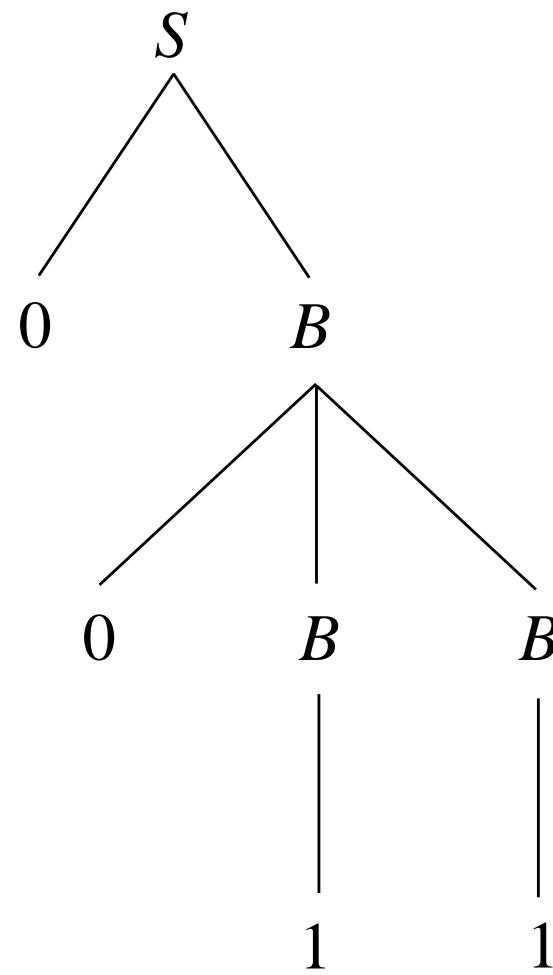
$$A \rightarrow 0|0S|1AA$$

$$B \rightarrow 1|1S|0BB$$

Consideriamo la stringa 0011. Una possibile derivazione è:  $S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 001B \Rightarrow 0011$   
Un'altra possibile è:  $S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 00B1 \Rightarrow 0011$

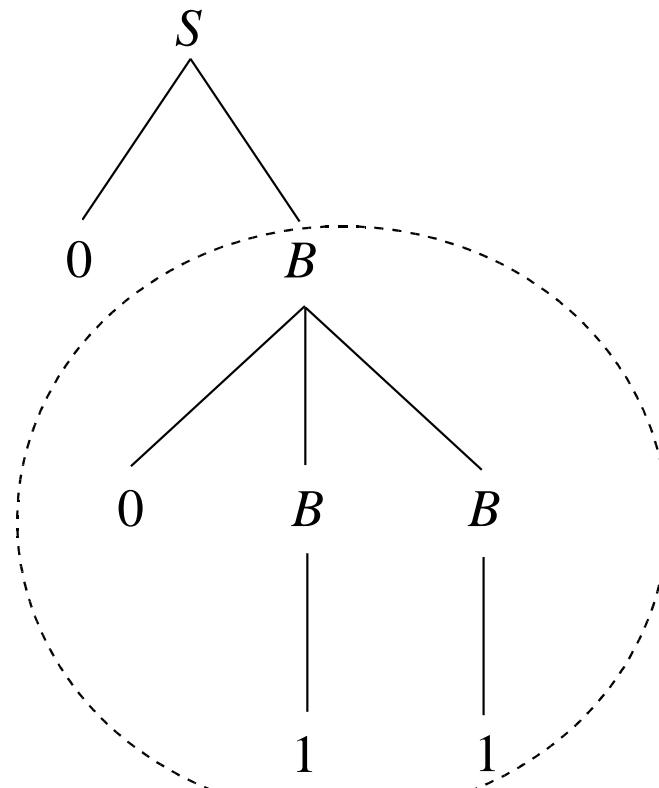
- Il non determinismo insito nella derivazione scompare quando si considera il relativo albero di derivazione (vedi figura seguente).

# Esempio



# Principio di sostituzione di sottoalberi

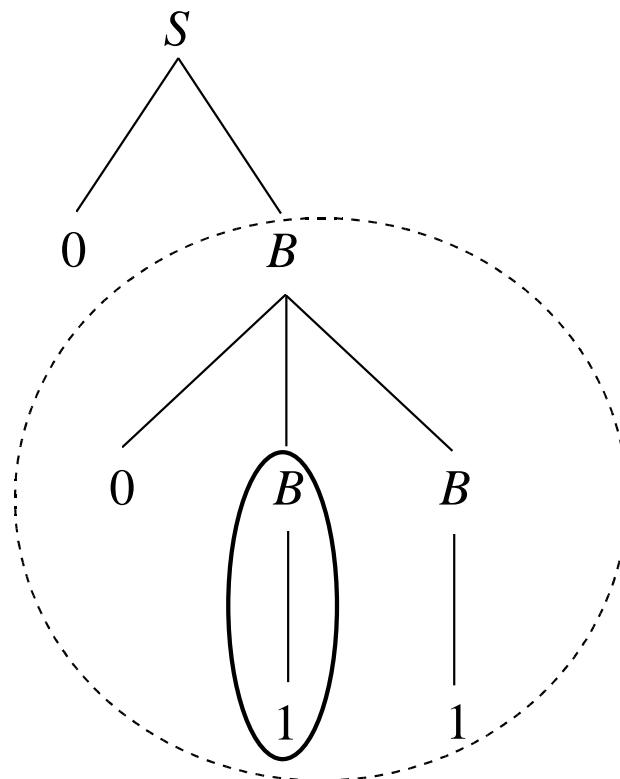
- Consideriamo ora il sottoalbero con radice nel nodo di profondità minore etichettato con una  $B$ .



# Principio di sostituzione di sottoalberi

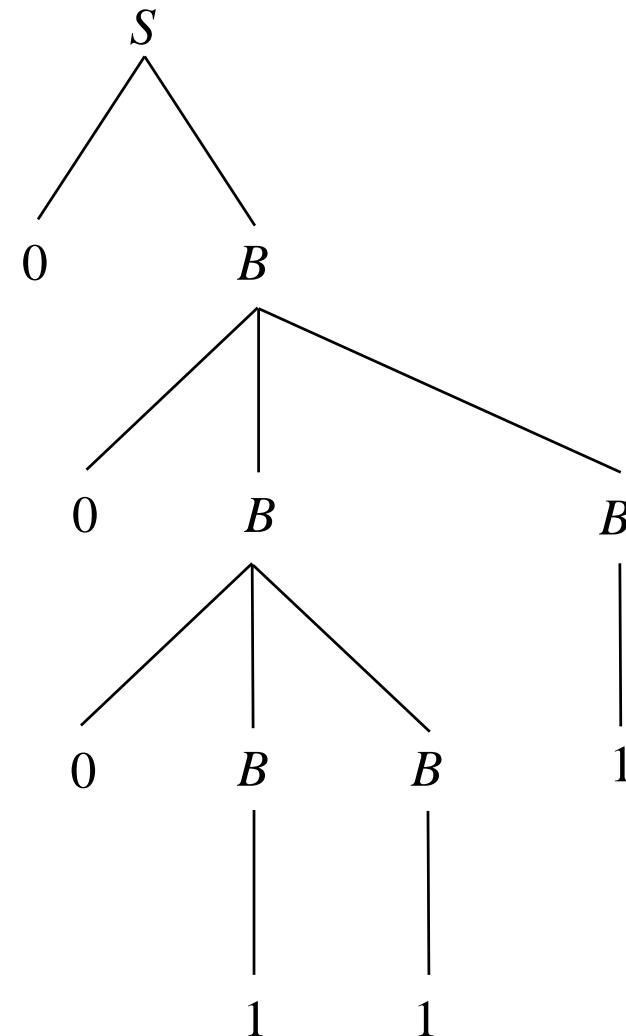
- Cosa accade se un qualsiasi sottoalbero con radice in un nodo etichettato con una  $B$  viene sostituito con il sottoalbero  ?
- Il nuovo albero è ancora un albero di derivazione (ovviamente, per una stringa differente da 0011).
- Consideriamo, ad esempio, il sottoalbero individuato dal cerchio pieno  nella figura seguente

# Principio di sostituzione di sottoalberi



- Il nuovo albero di derivazione è rappresentato nella seguente figura:

# Principio di sostituzione di sottoalberi



■ per cui:  $S \xrightarrow{*} 000111$

# Principio di sostituzione di sottoalberi

- Possiamo ripetere indefinitamente questo processo di sostituzione di sottoalberi, ottenendo parole del linguaggio di lunghezza crescente:

$$\begin{array}{ll} 0011 & |w|=4 \\ 000111 & |w|=6 \\ w = 00001111 & |w|=8 \\ M & M \\ 00^n11^n & |w|=2n+2 \quad n=1, 2, K \end{array}$$

La lunghezza cresce in maniera costante.

# Principio di sostituzione di sottoalberi

- Nelle grammatiche C.F., dunque, possiamo sostituire alberi più piccoli con alberi di dimensioni maggiori, purché abbiano la stessa radice - più precisamente, purché i nodi radice siano etichettati con lo stesso  $NT$  - ottenendo ancora alberi di derivazione validi.
- Una caratteristica dei linguaggi C.F., che discende direttamente dal processo descritto in precedenza, è che la lunghezza delle parole cresce in maniera costante.
- Dunque, se una grammatica genera parole la cui lunghezza cresce in maniera esponenziale, allora il linguaggio generato non è libero.
- Controesempio  $L = \{a^n b^n c^n \mid n > 0\}$

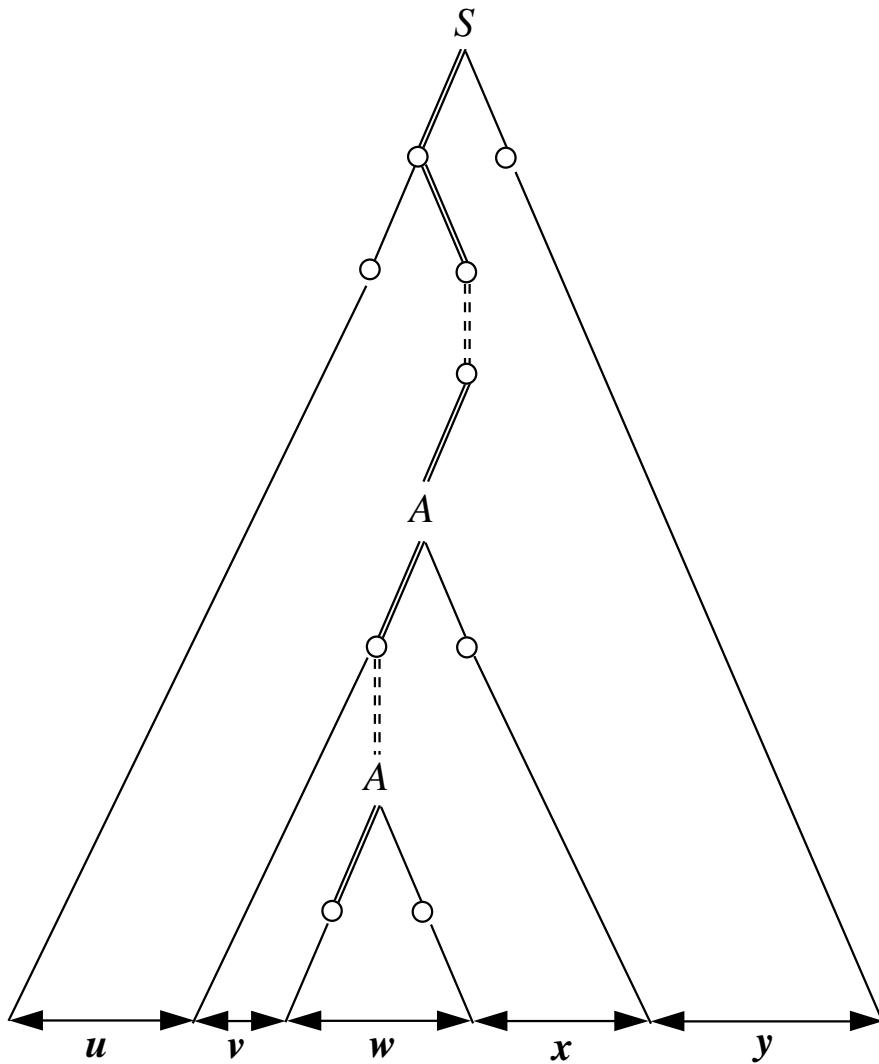
# Principio di sostituzione di sottoalberi

- Generalizziamo ora il discorso:

Supponiamo di avere un albero di derivazione  $T_z$  per una stringa  $z$  di terminali generata da una grammatica C.F.  $G$ , e supponiamo inoltre che il simbolo  $NT A$  compaia due volte su uno stesso cammino.

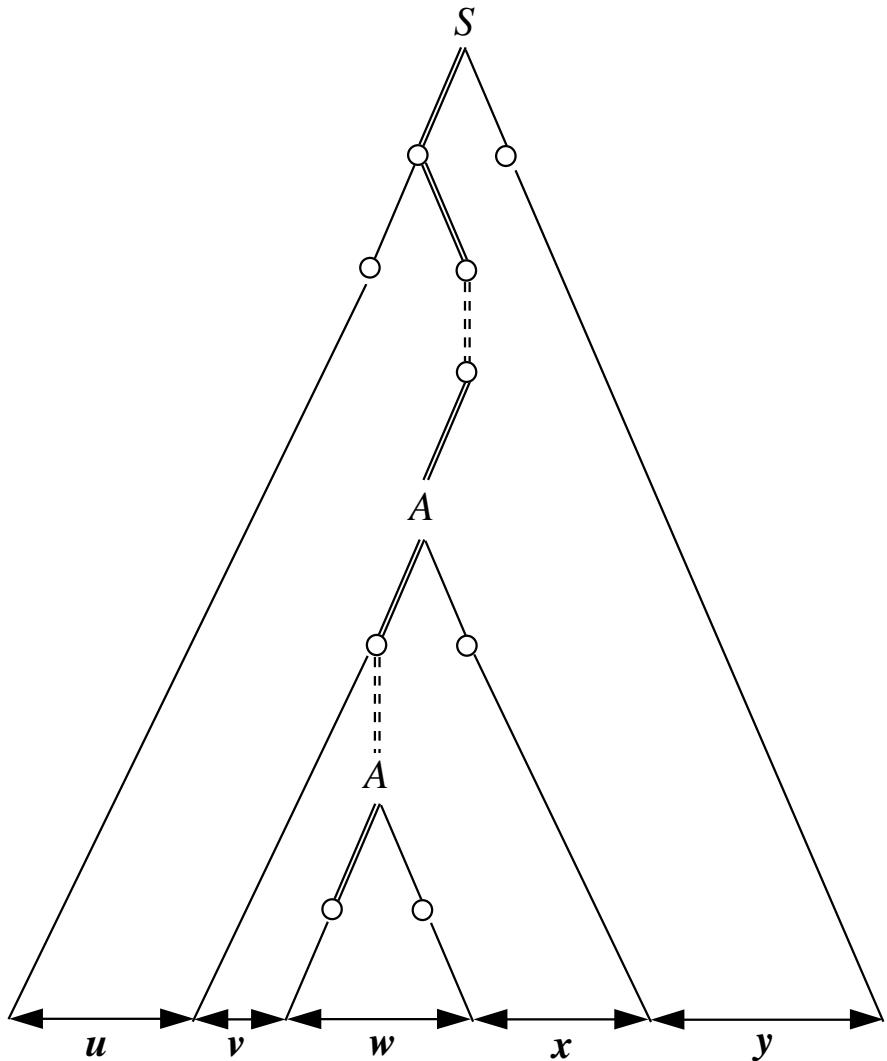
La situazione è rappresentata graficamente in Figura.

# Principio di sostituzione di sottoalberi



Il sottoalbero più in basso con radice nel nodo etichettato con una A genera la sottostringa  $w$ , mentre quello più in alto genera la sottostringa  $vwx$ . Poiché la  $G$  è C.F., sostituendo il sottoalbero più in alto con quello più in basso, si ottiene ancora una derivazione valida. Il *nuovo albero* genera la *stringa uw*.

# Principio di sostituzione di sottoalberi



Se effettuiamo la sostituzione inversa (il sottoalbero più in basso viene rimpiazzato da quello più in alto), otteniamo un albero di derivazione lecito per la stringa  $uvwxy$ , ossia  $uv^2wx^2y$ . Ripetendo questa sostituzione un numero finito di volte, si ottiene l'insieme di stringhe:

$$\{uv^nwx^n y \mid n \geq 0\}$$

# Proposizione

- Ogni linguaggio C.F. infinito deve contenere almeno un sottoinsieme infinito di stringhe della forma:

$$uv^nwx^ny \quad n \geq 0$$

Formalizziamo ora alcuni risultati connessi con il processo di sostituzione di sottoalberi.

## Lemma

- Sia  $G = (X, V, S, P)$  una grammatica C.F. e supponiamo che:

$$m = \max \left\{ |v| \mid A \rightarrow v \in P \right\}$$

Sia  $T_w$  un albero di derivazione per una stringa  $w$  di  $L(G)$ . Se l'altezza di  $T_w$  è al più uguale ad un intero  $j$ , allora:  $|w| \leq m^j$

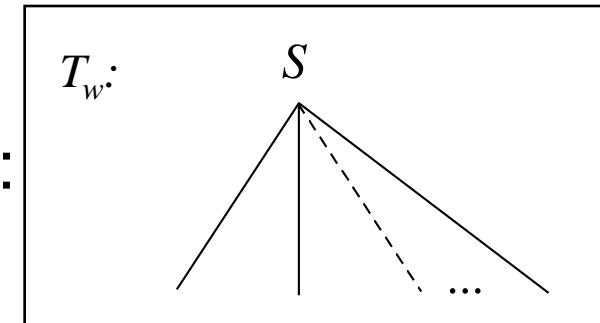
In formule:  $\text{height}(T_w) \leq j \Rightarrow |w| \leq m^j$

# Dimostrazione

- La dimostrazione vale per un albero di derivazione che abbia come radice un qualunque simbolo  $NT$ , non necessariamente  $S$ .  
Procediamo per induzione su  $j$ .

Passo base       $j = 1$

$T_w$  rappresenta un'unica produzione:



Dunque la parola generata è composta al più da  $m$  caratteri terminali e si ha:  $|w| \leq m$

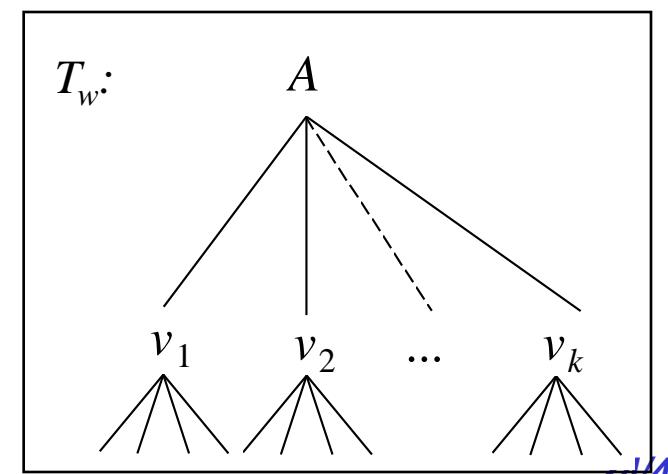
# Dimostrazione

## ■ Passo induttivo

Supponiamo che il lemma valga per ogni albero di altezza al più uguale a  $j$  e la cui radice sia un simbolo  $NT$  e dimostriamolo per un albero di altezza  $j+1$ .

Supponiamo che il livello più alto dell'albero rappresenti la produzione  $A \rightarrow v$ , dove

$v = v_1 v_2 \dots v_k$ ,  $|v| = k$ ,  $k \leq m$  (il sottoalbero di profondità 1 ha al più  $m$  figli).



# Dimostrazione

- Ogni simbolo  $v_i$ ,  $i = 1, 2, \dots, k$  di  $v$  può essere radice di un sottoalbero di altezza al più uguale a  $j$ , poiché  $T_w$  ha altezza uguale a  $j+1$  (un  $v_i$  potrebbe anche essere un terminale).

Dunque, per ipotesi di induzione, ciascuno di questi alberi ha al più  $m^j$  foglie. Poiché  $|v| = k \leq m$ , la stringa  $w$ , frontiera dell'albero  $T_w$ , ha lunghezza:

$$|w| \leq \underbrace{m^j + m^j + \dots + m^j}_{|v|=k \text{ volte}} = |v| \cdot m^j = k \cdot m^j \leq m \cdot m^j = m^{j+1}$$

c.v.d.

# Pumping Lemma per i linguaggi liberi da contesto o teorema $uvwxy$

- Sia  $L$  un linguaggio libero da contesto. Allora esiste una costante  $p$ , che dipende solo da  $L$ , tale che se  $z$  è una parola di  $L$  di lunghezza maggiore di  $p$  ( $|z| > p$ ), allora  $z$  può essere scritta come  $uvwxy$  in modo tale che:
  - (1)  $|vwx| \leq p$
  - (2) al più uno tra  $v$  e  $x$  è la parola vuota ( $vx \neq \lambda$ );
  - (3)  $\forall i, i \geq 0 : uv^i wx^i y \in L$

# Osservazione

- Dato un linguaggio generato da una grammatica che non è C.F., non possiamo escludere immediatamente che non esiste una grammatica C.F. che generi lo stesso linguaggio.
- Se un linguaggio infinito non obbedisce al Pumping Lemma, non può essere generato da una grammatica C.F.
- Il Pumping Lemma per i linguaggi liberi fornisce una *condizione necessaria* affinché un linguaggio sia libero.

$$L \text{ libero} \Rightarrow \exists p, \dots$$

- Dunque può essere utilizzato per dimostrare che un linguaggio non è libero (con una dimostrazione per assurdo - modus tollens).

$$\nexists p, \dots \Rightarrow L \text{ non è libero}$$

# Dimostrazione Pumping Lemma

- Sia  $G = (X, V, S, P)$  una grammatica C.F. che genera  $L$ . Denotiamo con  $m$  il numero di simboli della più lunga parte destra di una produzione di  $G$ :

$$m = \max \left\{ |v| \mid A \rightarrow v \in P \right\}$$

Denotiamo con  $k$  la cardinalità dell'alfabeto nonterminale di  $G$ :

$$k = |V|$$

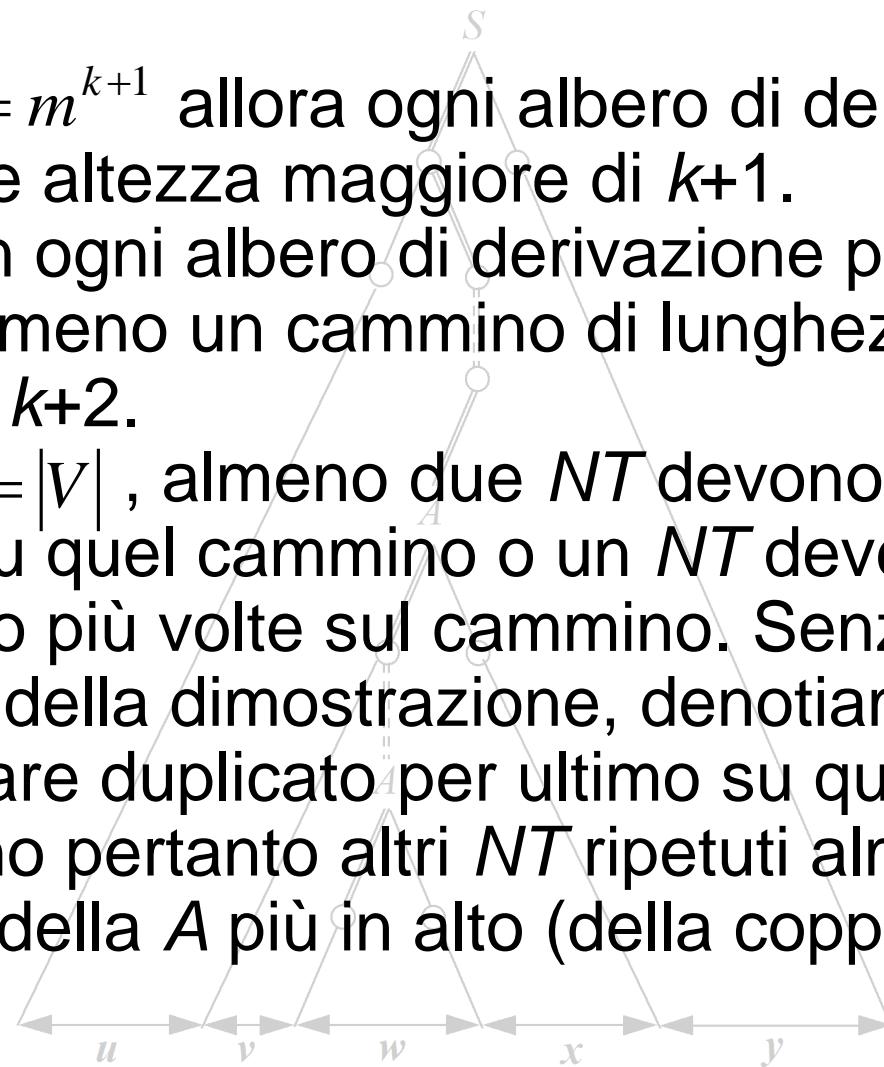
Poniamo  $p = m^{k+1}$  e sia  $z \in L$ , con  $|z| > p$

Per il lemma precedente:

$$\left[ \left( \text{height}(T_z) \leq j \Rightarrow |z| \leq m^j \right) \Leftrightarrow \left( |z| > m^j \Rightarrow \text{height}(T_z) > j \right) \right]$$

# Dimostrazione Pumping Lemma

- se  $|z| > p = m^{k+1}$  allora ogni albero di derivazione per z deve avere altezza maggiore di  $k+1$ . Dunque, in ogni albero di derivazione per z deve esistere almeno un cammino di lunghezza non inferiore a  $k+2$ . Poiché  $k = |V|$ , almeno due NT devono comparire duplicati su quel cammino o un NT deve essere ripetuto 3 o più volte sul cammino. Senza perdere la generalità della dimostrazione, denotiamo con A il NT che compare duplicato per ultimo su quel cammino. Non vi sono pertanto altri NT ripetuti almeno due volte al di sotto della A più in alto (della coppia di A).



# Dimostrazione Pumping Lemma

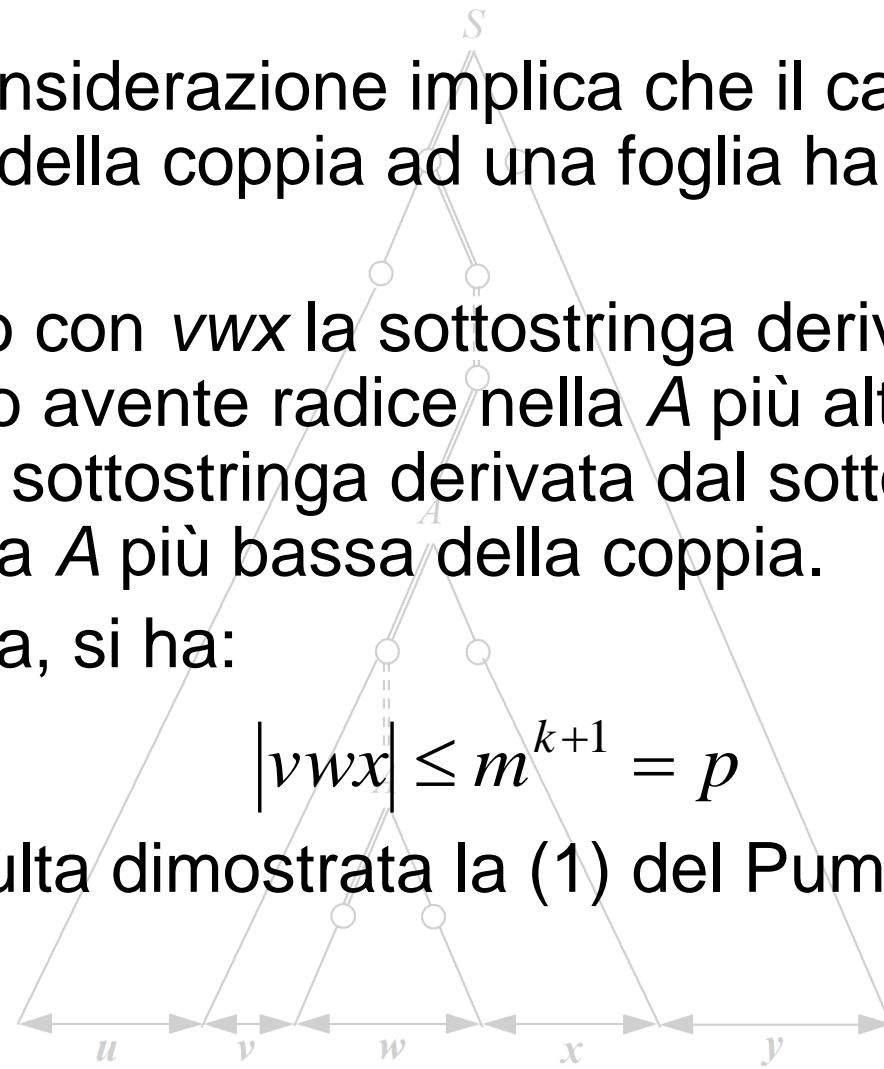
- Questa considerazione implica che il cammino dalla  $A$  più in alto della coppia ad una foglia ha lunghezza al più  $k+1$ .

Indichiamo con  $vwx$  la sottostringa derivata dal sottoalbero avente radice nella  $A$  più alta della coppia, ove  $w$  è la sottostringa derivata dal sottoalbero avente radice nella  $A$  più bassa della coppia.

Dal Lemma, si ha:

$$|vwx| \leq m^{k+1} = p$$

per cui risulta dimostrata la (1) del Pumping Lemma.



# Dimostrazione Pumping Lemma

- Scriviamo  $z$  nella forma  $uvwxy$ , ed applichiamo il principio di sostituzione di sottoalberi. Se sostituiamo al sottoalbero avente radice nella  $A$  più alta della coppia quello avente radice nella  $A$  più bassa, otteniamo un albero di derivazione per la stringa:

$$uwy = uv^0wx^0y \quad \text{che è la (3) per } i = 0.$$

Se operiamo la sostituzione inversa, otteniamo un albero di derivazione per la stringa:  $uvvwxxy = uv^2wx^2y$  che è la (3) per  $i = 2$ .

Se ripetiamo la suddetta sostituzione  $i-1$  volte, la stringa derivata è:  $u \underbrace{vv \dots v}_{i \text{ volte}} w \underbrace{xx \dots x}_{i \text{ volte}} y = uv^iwx^i y$

per cui la (3) risulta dimostrata.

# Dimostrazione Pumping Lemma

- La (2) può essere dimostrata per assurdo.

Sia:  $v = \lambda = x$

La sostituzione del sottoalbero avente radice nella  $A$  più alta della coppia con quello avente radice nella  $A$  più bassa non provoca alcun cambiamento nella stringa  $z$  derivata dall'intero albero. Ma tale sostituzione provoca la diminuzione della lunghezza del cammino che dalla  $A$  (in origine quella più in alto) porta ad una foglia. Dunque, anche il cammino di lunghezza non inferiore a  $k+2$  (su cui compariva la coppia di  $A$ ) nell'albero di derivazione per  $z$  risulta accorciato. In questo modo abbiamo ottenuto un albero di derivazione per  $z$  con altezza almeno uguale a  $k+1$ . Ma questo è assurdo per il Lemma.

c.v.d.

# Definizione di grammatica ambigua

- Una grammatica  $G$  libera da contesto è **ambigua** se esiste una stringa  $x$  in  $L(G)$  che ha due alberi di derivazione differenti.
- In modo alternativo,  $G$  è ambigua se esiste una stringa  $x$  in  $L(G)$  che ha due derivazioni sinistre (o destre) distinte.

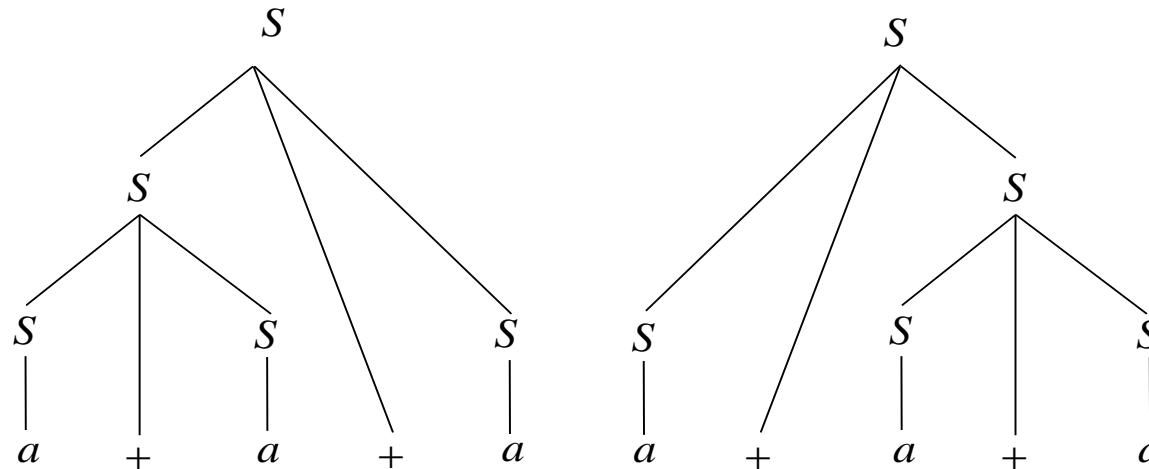
# Esempio di grammatica ambigua

- La seguente grammatica libera da contesto:

$$G_2 = (X, V, S, P)$$

$$X = \{a, +\}, \quad V = \{S\}, \quad P = \{S \rightarrow S + S, \quad S \rightarrow a\}$$

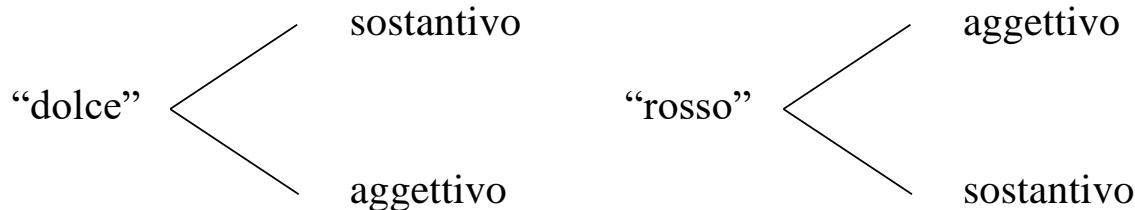
è ambigua. Infatti la stringa  $w = a + a + a$  in  $L(G_2)$  ha due differenti alberi di derivazione.



Nel linguaggio naturale, l'ambiguità (sintattica) è un fenomeno intrinseco, che si verifica frequentemente.

# Esempio

“un dolce rosso”



“una torta di colore rosso”

“un dolce rosso”

“un uomo rosso dal carattere dolce”

- Agli effetti delle applicazioni ai linguaggi di programmazione, l’ambiguità è una proprietà negativa. Infatti, il significato di una frase può essere definito come una funzione del suo albero di derivazione.

## Esempio

- Sicché, se esiste più di un albero di derivazione per una stessa frase, essa può avere un significato non univoco.

Si preferisce perciò cercare una grammatica differente che, pur generando lo stesso linguaggio, non sia ambigua.

L'unico vantaggio delle grammatiche ambigue risulta essere, in generale, il minore numero di regole che possono avere rispetto ad una grammatica non ambigua.

## Esempio

- Il linguaggio *precedente* può essere generato anche dalla seguente grammatica:

$$G_3 = (X, V, S, P)$$

$$X = \{a, +\}, \quad V = \{S\}, \quad P = \{S \rightarrow S + a, \quad S \rightarrow a\}$$

$G_3$  non è ambigua.

Inoltre:  $L(G_2) = L(G_3) = \left\{ aw \mid w \in \{+a\}^* \right\} = a \cdot \{+a\}^*$

- Esistono però dei linguaggi, detti *inherentemente ambigi*, per i quali tutte le grammatiche che li generano risultano ambigue.

## Definizione di linguaggio inerentemente ambiguo

- Un linguaggio  $L$  è *inerentemente ambiguo* se ogni grammatica che lo genera è ambigua.

$$(\forall G) \ L = L(G) : G \text{ ambigua}$$

## Esempio di linguaggio inerentemente ambiguo

- Un linguaggio inerentemente ambiguo è:

$$L = \{a^i b^j c^k \mid a^i b^j c^k, (i = j \vee j = k)\}$$

Intuitivamente, ci si rende conto di ciò pensando che in ogni grammatica che genera  $L$  devono esistere due diversi insiemi di regole per produrre le stringhe  $a^i b^j c^k$  e le stringhe  $a^i b^j c^j$ . Le stringhe  $a^i b^j c^j$  saranno necessariamente prodotte in due modi distinti, con distinti alberi di derivazione e conseguente ambiguità.

# Esempio di linguaggio ambiguo

## ■ Linguaggi di programmazione

□ Si consideri la seguente grammatica  $G = (X, V, S, P)$ :

$$X = \{\text{if}, \text{then}, \text{else}, a, b, p, q\}$$

$$V = \{S, C\}$$

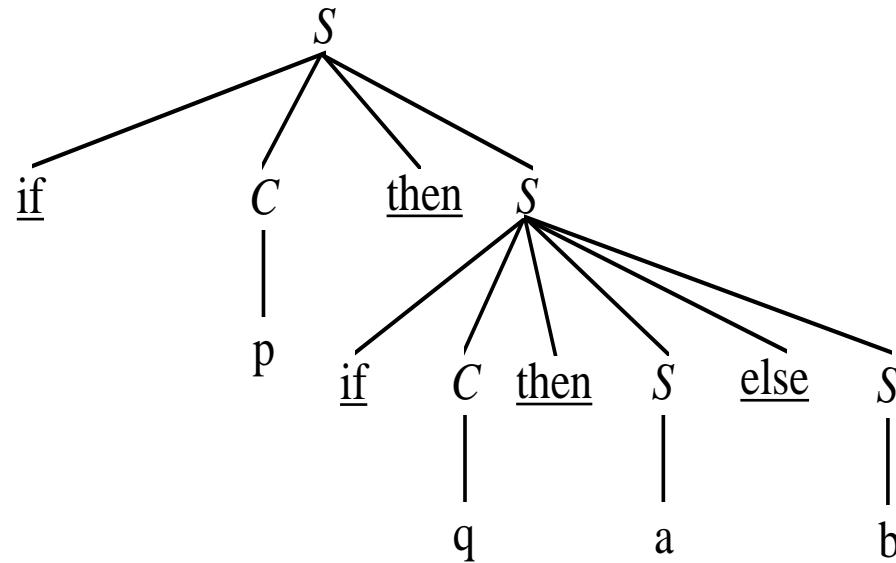
$$P = \{S \rightarrow \underline{\text{if}} \ C \ \underline{\text{then}} \ S \ \underline{\text{else}} \ S \mid \underline{\text{if}} \ C \ \underline{\text{then}} \ S \mid a \mid b, C \rightarrow p \mid q\}$$

$G$  è ambigua. Infatti la stringa:

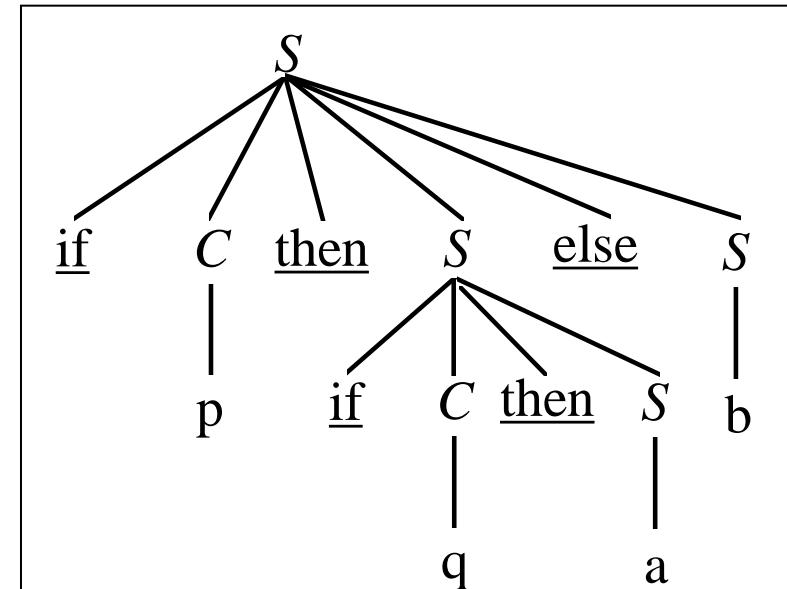
$$w = \underline{\text{if}} \ p \ \underline{\text{then}} \ \underline{\text{if}} \ q \ \underline{\text{then}} \ a \ \underline{\text{else}} \ b$$

può essere generata in due modi.

# Esempio di linguaggio ambiguo



if *p* then (if *q* then *a* else *b*)



*if* *p* then (if *q* then *a*) else *b*

# Esempio di linguaggio ambiguo

- Per rendere non ambigua  $G$  si usa solitamente la convenzione di associare ogni istruzione else con l'istruzione if più vicina.  
La grammatica che riflette questa convenzione è la seguente:

$$G' = (X, V, S, P)$$

$$X = \{\text{if}, \text{then}, \text{else}, \text{a}, \text{b}, \text{p}, \text{q}\}$$

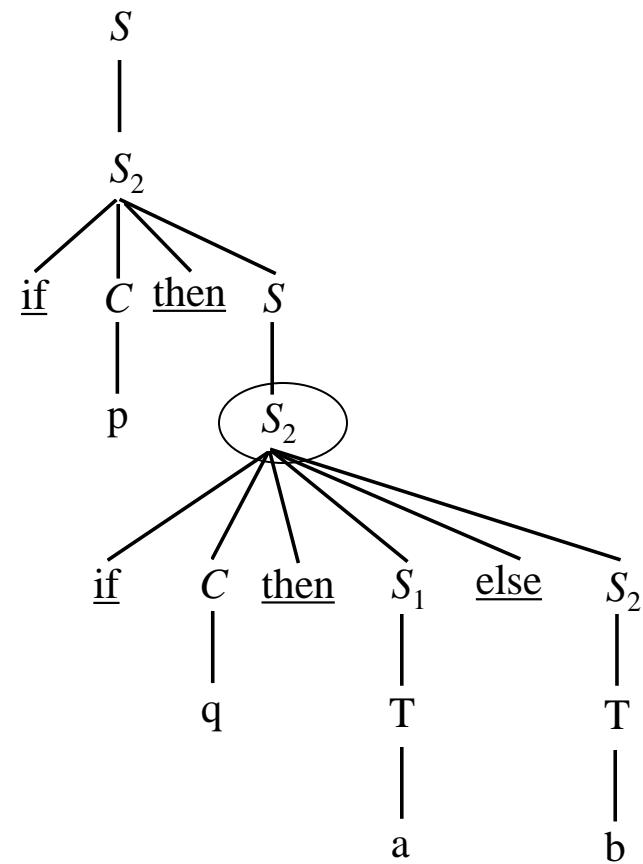
$$V = \{S, S1, S2, C, T\}$$

$$P = \{S \rightarrow S1 \mid S2, S1 \rightarrow \underline{\text{if}} \ C \ \underline{\text{then}} \ S1 \ \underline{\text{else}} \ S2 \mid T, S2 \rightarrow \underline{\text{if}} \ C \ \underline{\text{then}} \ S \mid \underline{\text{if}} \ C \ \underline{\text{then}} \ S1 \ \underline{\text{else}} \ S2 \mid T, C \rightarrow p \mid q, T \rightarrow a \mid b\}$$

# Esempio di linguaggio ambiguo

- In  $G'$  la stringa  $w = \underline{\text{if}} \underline{p} \underline{\text{then}} \underline{\text{if}} \underline{q} \underline{\text{then}} \underline{a} \underline{\text{else}} \underline{b}$  ha un unico albero di derivazione:

- Ma è proprio vero che l'albero di derivazione per  $w$  è unico? Si tenga conto che l'ambiguità di un linguaggio prescinde dal nome delle variabili ma dipende dalla struttura.



# Esercizi

- Determinare la grammatica che genera il seguente linguaggio:

$$L = \{a^n b^n c^n \mid n > 0\}$$

e dimostrare la correttezza di tale grammatica.

- Di che tipo è la grammatica che genera  $L$ ?
- Applicare il Pumping Lemma per dimostrare che  $L$  non è libero.

Soluzione esercizio

# Esercizi

- Applicare il Pumping Lemma per dimostrare che il seguente linguaggio  $L$  non è libero da contesto:

$$L = \left\{ a^{n^2} \mid n \geq 0 \right\}$$

Soluzione esercizio

# Esercizi

- Applicare il Pumping Lemma per dimostrare che il seguente linguaggio  $L$  non è libero da contesto:

$$L = \left\{ a^i b^j \mid i = 2^j, \ i, j \geq 0 \right\}$$

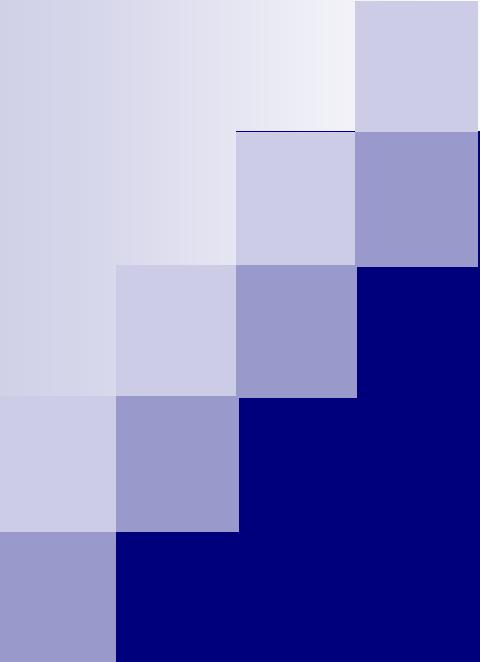
Soluzione esercizio

# Esercizi

- Dimostrare che il seguente linguaggio non è libero da contesto:

$$L = \left\{ a^k b^r \mid k > 0, \ r > k^2 \right\}$$

Soluzione esercizio



# Linguaggi di Programmazione

## Capitolo 5 – Grammatiche e macchine

# Classificazione delle grammatiche secondo Chomsky

## ■ Definizione: Gerarchia di Chomsky (1956-1959)

- Sia  $G = (X, V, S, P)$  una grammatica. Dalla definizione di grammatica, si ha:

$$P = \left\{ v \rightarrow w \mid v \in (X \cup V)^+ \text{ e } v \text{ contiene almeno un } NT, \ w \in (X \cup V)^* \right\}$$

A seconda delle restrizioni imposte sulle regole di produzione, si distinguono le varie classi di grammatiche.

# Classificazione delle grammatiche secondo Chomsky

## ■ Definizione: Gerarchia di Chomsky (1956-1959)

- **Tipo '0'** - Quando le stringhe che appaiono nella produzione  $v \rightarrow w$  non sono soggette ad alcuna limitazione.
- **Tipo '1' - Dipendente da contesto** - quando le produzioni sono limitate alla forma:
  - (1)  $yAz \rightarrow ywz$ , con  $A \in V$ ,  $y, z \in (X \cup V)^*$ ,  $w \in (X \cup V)^+$
  - (2)  $S \rightarrow \lambda$ , purché  $S$  non compaia nella parte destra di alcuna produzione.
- **Tipo '2' - Libera da contesto** - quando le produzioni sono limitate alla forma:  $v \rightarrow w$  con  $v \in V$
- **Tipo '3' - Lineare destra** - quando le produzioni sono limitate alla forma:
  - (1)  $A \rightarrow bC$  con  $A, C \in V$  e  $b \in X$ ;
  - (2)  $A \rightarrow b$  con  $A \in V$  e  $b \in X \cup \{\lambda\}$ .

# Classificazione delle grammatiche secondo Chomsky

- Una grammatica di tipo ‘3’ è detta lineare destra perché il  $NT$ , se c’è, compare a destra (nella parte destra della produzione).  
Un linguaggio generato da una tale grammatica è detto *di tipo ‘3’* o *lineare a destro*.

## Esempi di linguaggi di tipo ‘3’

- Consideriamo la grammatica  $G_1$ :

$$S \rightarrow \lambda | 0 | 0S | 1 | 1S$$

$G_1$  è lineare destra.  $L(G_1) = \{0, 1\}^*$  (l'insieme di tutte le stringhe binarie).

- Consideriamo la grammatica  $G_2$ :

$$S \rightarrow \lambda | 0S | 1T$$

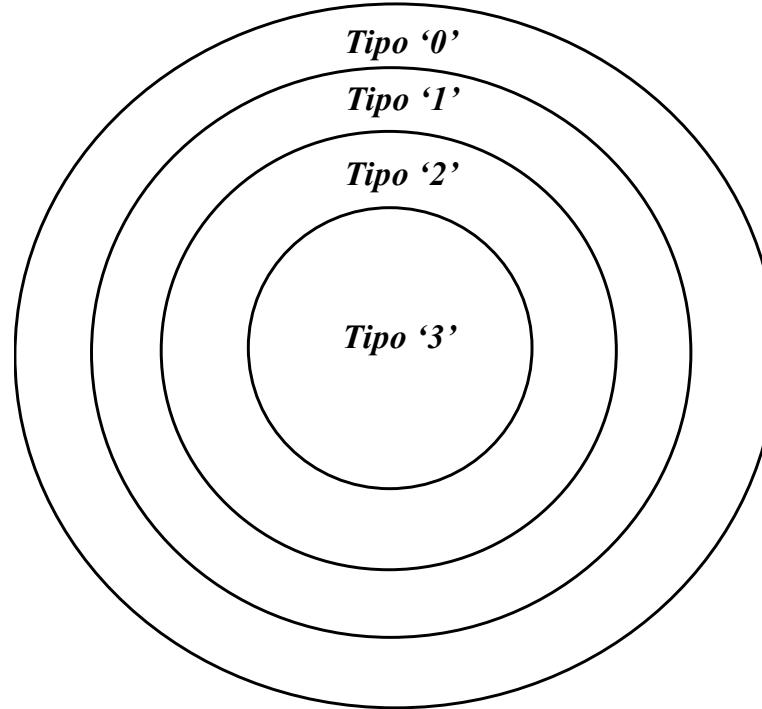
$$T \rightarrow 0T | 1S$$

$G_2$  è lineare destra.

$$L(G_2) = \left\{ w \in \{0, 1\}^* \mid w \text{ ha un numero pari di } 1 \right\}$$

# Teorema della gerarchia

- Dimostriamo formalmente che le quattro classi di linguaggi viste costituiscono una gerarchia



# Teorema della gerarchia

- Denotiamo con  $\mathcal{L}_i$  il seguente insieme:

$$\mathcal{L}_i = \{L \subset X^* \mid L = L(G), \ G \text{ di tipo } i\}$$

(classe dei linguaggi di tipo  $i$ ).

La gerarchia di Chomsky è una gerarchia in senso stretto di classi di linguaggi:

$$\mathcal{L}_3 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_0$$

## Dimostrazione

# Teorema della gerarchia

- $\mathcal{L}_3 \subsetneq \mathcal{L}_2$   
Per dimostrare che la classe di linguaggi  $\mathcal{L}_3$  è *inclusa propriamente* nella classe di linguaggi  $\mathcal{L}_2$  si deve dimostrare che ogni linguaggio di tipo ‘3’ può essere generato da una grammatica di tipo ‘2’ e che esiste almeno un linguaggio C.F. (di tipo ‘2’) che non può essere generato da una grammatica di tipo ‘3’.

$\mathcal{L}_3 \subseteq \mathcal{L}_2$  discende dalle definizioni di linguaggio di tipo ‘3’ e di grammatica di tipo ‘2’. Infatti, si osserva facilmente che ogni grammatica di tipo ‘3’ è anche una grammatica di tipo ‘2’.

$\mathcal{L}_3 \neq \mathcal{L}_2$ : posponiamo questa dimostrazione.

Mostreremo che  $L = \{a^k b^k \mid k > 0\}$  non è di tipo ‘3’ (abbiamo già determinato una grammatica di tipo ‘2’ che genera  $L$ ).

# Teorema della gerarchia

- $\mathcal{L}_2 \subsetneq \mathcal{L}_1$   
Abbiamo già osservato che le produzioni C.F. sono un caso particolare delle produzioni C.S. con l'unica eccezione rappresentata dalle produzioni:

$$A \rightarrow \lambda, \quad A \neq S$$

che sono C.F. ma **non** C.S. Dunque:

$$\forall L : L \in \mathcal{L}_2 \stackrel{\text{def}}{\iff} \exists G, G \text{ è C.F.} : L = L(G).$$

Se  $A \rightarrow \lambda, A \in V \setminus \{S\}$  non è una produzione di  $G$ , allora  $G$  è anche C.S. (di tipo '1') e l'asserto è dimostrato. Il problema sorge se  $G$  ha almeno una  $\lambda$ -produzione. In tal caso, ci avvaliamo del seguente risultato:

## Lemma della stringa vuota

- Sia  $G = (X, V, S, P)$  una grammatica C.F. con almeno una  $\lambda$ -produzione. Allora esiste una grammatica C.F.  $G'$  tale che:
  - i)  $L(G)=L(G')$  ( $G'$  è equivalente a  $G$ );
  - ii) se  $\lambda \notin L(G)$  allora in  $G'$  non esistono produzioni del tipo  $A \rightarrow \lambda$ ;
  - iii) se  $\lambda \in L(G)$  allora in  $G'$  esiste un'unica produzione  $S' \rightarrow \lambda$ , ove  $S'$  è il simbolo iniziale di  $G'$  ed  $S'$  non compare nella parte destra di alcuna produzione di  $G'$ .

# Teorema della gerarchia

- Riprendiamo la dimostrazione di  $\mathcal{L}_2 \subset \mathcal{L}_1$

$$\forall L : L \in \mathcal{L}_2 \stackrel{\text{def}}{\iff} \exists G, G \text{ è C.F.} : L = L(G).$$

Se  $G$  ha almeno una  $\lambda$ -produzione, utilizziamo il Lemma della stringa vuota per determinare una grammatica C.F.  $G'$  equivalente a  $G$ , ma priva di  $\lambda$ -produzioni (al più, in  $G'$  compare la produzione  $S' \rightarrow \lambda$ , ed  $S'$  non compare nella parte destra di alcuna produzione di  $G'$ ).  $G'$  è di tipo '1'.

Questo dimostra che  $\mathcal{L}_2 \subseteq \mathcal{L}_1$ .

# Teorema della gerarchia

- $\mathcal{L}_2 \neq \mathcal{L}_1$

$$L = \{a^n b^n c^n \mid n > 0\}$$

è di tipo ‘1’ ma non di tipo ‘2’. Si osservi che, per asserire che  $L$  è di tipo ‘1’, ci siamo avvalsi del teorema che stabilisce l’equivalenza delle classi di linguaggi contestuali e monotoni.

- $\mathcal{L}_1 \subset \mathcal{L}_0$

Non lo dimostriamo formalmente. La dimostrazione comporta la conoscenza degli automi limitati lineari e delle macchine di Turing (che riconoscono linguaggi di tipo ‘0’ o ricorsivamente enumerabili). Ci limitiamo ad osservare che  $\mathcal{L}_1 \subseteq \mathcal{L}_0$  discende direttamente dalle definizioni di linguaggio di tipo ‘1’ e di grammatica di tipo ‘0’.

c.v.d.

# Operazioni sui linguaggi

- Siano  $L_1$  ed  $L_2$  due linguaggi definiti su uno stesso alfabeto  $X$  ( $L_1, L_2 \subseteq X^*$ ).

- *L'unione* di  $L_1$  ed  $L_2$  è:

$$L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$$

- *La concatenazione* di  $L_1$  ed  $L_2$  (anche detta *il prodotto* di  $L_1$  ed  $L_2$  o “ $L_1$  punto  $L_2$ ”) è:

$$L_1 \cdot L_2 = \{w \mid w = w_1 w_2, \quad w_1 \in L_1, \quad w_2 \in L_2\}$$

- *L'iterazione* di  $L_1$  (o *chiusura riflessiva e transitiva* di  $L_1$  rispetto all'operazione di concatenazione, anche detta *stellatura* di  $L_1$  o “ $L_1$  star” o *chiusura di Kleene*) è:

$$L_1^* = \{w \mid w = w_1 w_2 \dots w_n, \quad n \geq 0 \text{ e } \forall i : w_i \in L_1\}$$

# Operazioni sui linguaggi

- Siano  $L_1$  ed  $L_2$  due linguaggi definiti su uno stesso alfabeto  $X$  ( $L_1, L_2 \subseteq X^*$ ).
  - Il *complemento* di  $L_1$  è:

$$\overline{L_1} = X^* - L_1$$

- L'*intersezione* di  $L_1$  ed  $L_2$  è:

$$L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$$

# Proprietà

- Dati  $L_1, L_2, L_3 \subseteq X^*$  ( $\equiv L_1, L_2, L_3 \in 2^{X^*}$ ), risulta:
  - $(L_1 \cdot L_2) \cdot L_3 = L_1 \cdot (L_2 \cdot L_3)$  (proprietà associativa)
  - $L_1 \cdot L_2 \neq L_2 \cdot L_1$
  - $L_1 \cdot \{\lambda\} = \{\lambda\} \cdot L_1 = L_1$  ( $\{\lambda\}$  è l'elemento neutro rispetto all'operazione di concatenazione di linguaggi)
- Dunque anche  $(2^{X^*}, \cdot)$  è un monoide;
  - $L_1 \cdot \emptyset = \emptyset \cdot L_1 = \emptyset$  ( $\emptyset$  è l'elemento assorbente);
  - Se  $\lambda \in L_1$        $L_2 \subseteq L_1 \cdot L_2$   
                                 $L_2 \subseteq L_2 \cdot L_1$
  - Se  $\lambda \in L_2$        $L_1 \subseteq L_1 \cdot L_2$   
                                 $L_1 \subseteq L_2 \cdot L_1$

## Esempio

Linguaggi di Programmazione

# Definizione di potenza di un linguaggio

- Sia  $L$  un linguaggio definito su un alfabeto  $X$ . Dicesi **potenza  $n$ -esima** di  $L$ , e si denota con  $L^n$ ,  $n \geq 0$ , il seguente linguaggio:

$$L^n = \begin{cases} \{\lambda\} & \text{se } n = 0 \\ L^{n-1} \cdot L & \text{altrimenti} \end{cases}$$

Posto:

$$L^+ = \bigcup_{i \geq 1} L^i$$

si ha:

$$L^* = \{\lambda\} \cup L^+ = L^0 \cup L^+ = \bigcup_{i \geq 0} L^i$$

# Definizione di potenza di un linguaggio

- $L^+$  è detta chiusura transitiva rispetto alla operazione di concatenazione.

Dunque si ha:

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

$$L^1 = L^0 \cdot L = L$$

$$L^2 = L^1 \cdot L = (L^0 \cdot L) \cdot L$$

$$L^3 = L^2 \cdot L = (L^1 \cdot L) \cdot L = ((L^0 \cdot L) \cdot L) \cdot L$$

# Proposizione

- Sia  $L$  un linguaggio definito su un alfabeto  $X$ . Si ha:

$$L^* = \bigcup_{i \geq 0} L^i = \{\lambda\} \cup L \cup L^2 \cup L^3 \cup \dots$$

Esempio

# Proprietà di chiusura delle classi di linguaggi

## ■ Definizioni

- $L$  *linguaggio* definito su  $X \stackrel{def}{\Leftrightarrow} L \subseteq X^* \Leftrightarrow L \in 2^{X^*}$
- $\mathcal{L}$  *classe di linguaggi* su  $X \Leftrightarrow \mathcal{L} \subseteq 2^{X^*} \Leftrightarrow \mathcal{L} \in 2^{2^{X^*}}$

# Proprietà di chiusura delle classi di linguaggi

## ■ Definizione di chiusura

- Sia  $\mathcal{L}$  una classe di linguaggi su  $X$ .

Sia  $\alpha$  un'operazione binaria sui linguaggi di  $\mathcal{L}$ :

$$\alpha : 2^{X^*} \times 2^{X^*} \rightarrow 2^{X^*}, \quad (L_1, L_2) \alpha \quad \alpha(L_1, L_2)$$

Sia  $\beta$  un'operazione unaria sui linguaggi di  $\mathcal{L}$ :

$$\beta : 2^{X^*} \rightarrow 2^{X^*}, \quad \underset{\text{def}}{L \alpha} \quad \beta(L)$$

$\mathcal{L}$  è **chiusa** rispetto ad  $\alpha \iff \forall L_1, L_2 \in \mathcal{L} : \alpha(L_1, L_2) \in \mathcal{L}$

$\mathcal{L}$  è **chiusa** rispetto a  $\beta \iff \forall L_1 \in \mathcal{L} : \beta(L_1) \in \mathcal{L}$

## ■ Esempio

- $\mathcal{L}$  è chiusa rispetto all'iterazione se  $\forall L_1 \in \mathcal{L} : L_1^* \in \mathcal{L}$

# Teorema di chiusura

- La classe dei linguaggi di tipo  $i$ ,  $i = 0, 1, 2, 3$ , è chiusa rispetto alle operazioni di *unione*, *concatenazione* ed *iterazione*.

## Dimostrazione

La dimostrazione di questo teorema è **costruttiva**.

Siano  $L_1$  ed  $L_2$  due linguaggi:

$$L_1 = L(G_1)$$

$$G_1 = (X, V_1, S_1, P_1)$$

$$L_2 = L(G_2)$$

$$G_2 = (X, V_2, S_2, P_2)$$

Assumiamo che:  $V_1 \cap V_2 = \emptyset$   $\underline{S \notin V_1 \cup V_2}$

Poniamo:  $V = V_1 \cup V_2 \cup \{S\}$

Nel caso in cui tale assunzione non sia vera, ridenominiamo i nonterminali in comune.

# Dimostrazione Teorema di chiusura

- Lo schema generale della dimostrazione è il seguente:
  - consideriamo un'operazione alla volta (denotata con  $\alpha$ );
  - date  $G_1$  e  $G_2$ , costruiamo  $G$ ;
  - si dimostra che, se  $G_1$  e  $G_2$  sono di tipo  $i$ , allora  $G$  è di tipo  $i$ ;
  - si dimostra che  $L(G) = \alpha(L_1, L_2)$  e dunque la classe di linguaggi  $\mathcal{L}_i$  è chiusa rispetto alla operazione  $\alpha$ .

# Dimostrazione Teorema di chiusura

## ■ UNIONE

Costruiamo la grammatica:  $G_3 = (X, V, S, P_3)$  ove:

$$P_3 = \{S \rightarrow S_1, S \rightarrow S_2\} \cup P_1 \cup P_2$$

Osserviamo che, se  $G_1$  e  $G_2$  sono entrambe di tipo  $i$ ,  $i = 0, 1, 2$ , lo è anche  $G_3$ . In ciascuno di questi casi, si ha:

$$L(G_3) = L_1 \cup L_2$$

Infatti una derivazione da  $S$  in  $G_3$  deve necessariamente iniziare o con:

$S \Rightarrow S_1$  ed in tal caso può generare unicamente parole di  $L(G_1)$  oppure con

$S \Rightarrow S_2$  ed in tal caso genera una parola di  $L(G_2)$ .

Dunque, risulta dimostrato che  $\mathcal{L}_0$ ,  $\mathcal{L}_1$  e  $\mathcal{L}_2$ , sono chiuse rispetto all'unione.

# Dimostrazione Teorema di chiusura

## ■ UNIONE

Però, se  $G_1$  e  $G_2$  sono di tipo ‘3’,  $G_3$  non è lineare destra, perché le produzioni  $S \rightarrow S_1$  ed  $S \rightarrow S_2$  non sono ammesse.

Per avere ancora produzioni lineari destre che simulino il passo iniziale di una derivazione in  $G_1$  ed anche il passo iniziale di una derivazione in  $G_2$ , costruiamo pertanto la grammatica:  $G_4 = (X, V, S, P_4)$  ove:

$$P_4 = \{S \rightarrow w \mid S_1 \rightarrow w \in P_1\} \cup \{S \rightarrow w \mid S_2 \rightarrow w \in P_2\} \cup P_1 \cup P_2$$

$G_4$  è lineare destra se  $G_1$  e  $G_2$  lo sono e inoltre:

$$L(G_4) = L_1 \cup L_2$$

ed  $\mathcal{L}_3$  è chiusa rispetto all'unione.

Esempio

# Dimostrazione Teorema di chiusura

## ■ CONCATENAZIONE

Costruiamo la grammatica:  $G_5 = (X, V, S, P_5)$  ove:

$$P_5 = \{S \rightarrow S_1 S_2\} \cup P_1 \cup P_2$$

Osserviamo che, se  $G_1$  e  $G_2$  sono entrambe di tipo  $i$ ,  $i = 0, 1, 2$ , lo è anche  $G_5$ .

Se  $G_1$  e  $G_2$  sono C.F. (tipo '2'), allora si ha:

$$L(G_5) = L_1 \cdot L_2$$

in quanto ogni derivazione da  $S$  in  $G_5$  ha la seguente "struttura":

$$S \xrightarrow[G_1]{*} S_1 S_2 \xrightarrow[G_2]{*} w_1 S_2 \xrightarrow{*} w_1 w_2$$

ove, evidentemente, si ha:

$$S_1 \xrightarrow{*} w_1 \quad S_2 \xrightarrow{*} w_2$$

# Dimostrazione Teorema di chiusura

## ■ CONCATENAZIONE

Il seguente controesempio mostra che la condizione che  $G_1$  e  $G_2$  siano di tipo '2' è fondamentale per la validità di

$$L(G_5) = L_1 \cdot L_2$$

**Controesempio:** Consideriamo  $G_1$  con  $P_1 = \{S_1 \rightarrow b\}$  e  $G_2$  con  $P_2 = \{bS_2 \rightarrow bb\}$ . Da cui  $L_1 = L(G_1) = \{b\}$  ed  $L_2 = L(G_2) = \emptyset$

Dunque  $L_1 \cdot L_2 = \emptyset$

Se costruiamo  $G_5$ , si ha:  $P_{G_5} = \{S \rightarrow S_1S_2, S_1 \rightarrow b, bS_2 \rightarrow bb\}$  ed  $L(G_5) \neq \emptyset$  in quanto  $S \Rightarrow bb$  attraverso la seguente derivazione:  $S \Rightarrow S_1S_2 \xrightarrow{G_5} bS_2 \Rightarrow bb$

Dunque la  $G_5$  va bene solo per grammatiche di tipo '2' e risulta dimostrato che  $\mathcal{L}_2$  è chiusa rispetto alla concatenazione.

# Dimostrazione Teorema di chiusura

## ■ CONCATENAZIONE

La dimostrazione fatta non va bene per grammatiche di tipo ‘0’ e di tipo ‘1’, in quanto entrambe queste classi di grammatiche presentano produzioni dipendenti da contesto.

In presenza di grammatiche di tali tipi è necessario impedire che le derivazioni da  $S_2$  si servano di precedenti derivazioni da  $S_1$  e/o viceversa. In altri termini, è necessario evitare **interferenze tra derivazioni** da  $S_1$  e derivazioni da  $S_2$  nella definizione dei contesti. È possibile ottenere ciò considerando copie distinte di  $X$  in derivazioni distinte (da  $S_1$  e da  $S_2$ ), in modo che nessun  $NT$  derivato da  $S_1$  possa far uso di parte di una forma di frase derivata da  $S_2$  come contesto per l’applicazione di una produzione (e viceversa).

# Dimostrazione Teorema di chiusura

## ■ CONCATENAZIONE

Siano pertanto:  $X' = \{x' \mid x \in X\}$  e  $X'' = \{x'' \mid x \in X\}$

due copie distinte di  $X$  tali che:

$$X' \cap X'' = \emptyset$$

$$X' \cap X = \emptyset$$

$$X' \cap V = \emptyset$$

$$X'' \cap X = \emptyset$$

$$X'' \cap V = \emptyset$$

e sia  $P'_1$  l'insieme delle produzioni ottenute da  $P_1$ , sostituendo ogni occorrenza di un terminale  $x$  in  $X$  con il corrispondente nonterminale  $x'$  in  $X'$ :

$$P'_1 = P_1 \left[ \begin{array}{c} x' \\ \diagup \\ x \end{array} \right]$$

Similmente, sia  $P''_2$  l'insieme delle produzioni ottenute da  $P_2$ , sostituendo ogni occorrenza di un terminale  $x$  in  $X$  con il corrispondente  $x''$  in  $X''$ :

$$P''_2 = P_2 \left[ \begin{array}{c} x'' \\ \diagup \\ x \end{array} \right]$$

# Dimostrazione Teorema di chiusura

## ■ CONCATENAZIONE

In questo modo evitiamo l'interferenza tra contesti.

Costruiamo ora la grammatica:  $G_6 = (X, V \cup X' \cup X'', S, P_6)$

ove:  $P_6 = \{S \rightarrow S_1 S_2\} \cup P'_1 \cup P''_2 \cup \{x' \rightarrow x \mid x \in X\} \cup \{x'' \rightarrow x \mid x \in X\}$

Se  $G_1$  e  $G_2$  sono entrambe di tipo  $i$ ,  $i = 0, 1$ , lo è anche  $G_6$ .

Inoltre, si ha:  $L(G_6) = L_1 \cdot L_2$

Il controesempio visto in precedenza non dà più problemi:

$$P'_1 = \{S_1 \rightarrow b'\} \quad P''_2 = \{b'' S_2 \rightarrow b'' b''\}$$

e:  $G_6 = (X, V \cup \{b'\} \cup \{b''\}, S, P_6)$  dove:

$$P_6 = \{S \rightarrow S_1 S_2, S_1 \rightarrow b', b'' S_2 \rightarrow b'' b'', b' \rightarrow b, b'' \rightarrow b\}$$

ed  $L(G_6) = \emptyset$  in quanto  $S \not\Rightarrow bb$ ,  $S \Rightarrow S_1 S_2 \Rightarrow b'S_2 \Rightarrow bS_2$

Risulta così dimostrato che  $\mathcal{L}_0$  ed  $\mathcal{L}_1$  sono chiuse rispetto alla concatenazione.

# Dimostrazione Teorema di chiusura

## ■ CONCATENAZIONE

È immediato osservare che, se  $G_1$  e  $G_2$  sono di tipo ‘3’ né  $G_5$  né  $G_6$  sono di tipo ‘3’, per la presenza della produzione  $S \rightarrow S_1S_2$ .

Dobbiamo simulare l’effetto della produzione  $S \rightarrow S_1S_2$ , che determina la concatenazione delle parole generate da  $S_1$  e da  $S_2$ . A tale scopo osserviamo che, data una grammatica di tipo ‘3’, ogni forma di frase derivata dal simbolo iniziale di tale grammatica ha due peculiarità:

- (1) in essa compare al più un NT
- (2) se in essa compare un NT, questo è il simbolo più a destra

$$\left( S \Rightarrow x_1 A \Rightarrow x_1 x_2 B \stackrel{*}{\Rightarrow} x_1 x_2 x_3 \dots x_n N \Rightarrow x_1 x_2 x_3 \dots x_n x_{n+1} \right)$$

# Dimostrazione Teorema di chiusura

## ■ CONCATENAZIONE

Modifichiamo pertanto ogni produzione del tipo  $A \rightarrow b$  in  $G_1$  in modo che essa non costituisca l'ultima produzione applicata in una derivazione da  $S_1$  in  $G_1$ , ma possa innescare una derivazione da  $S_2$  in  $G_2$ . Poniamo dunque a destra della  $b$  il simbolo iniziale  $S_2$  di  $G_2$ .

Le produzioni del tipo  $A \rightarrow b$  in  $G_1$  vengono trasformate in:  $A \rightarrow bS_2$ . Costruiamo dunque la grammatica:

$$G_7 = (X, V - \{S\}, S_1, P_7)$$

ove:

$$\begin{aligned} P_7 = & \{A \rightarrow bB \mid A \rightarrow bB \in P_1\} \cup \{A \rightarrow bS_2 \mid A \rightarrow b \in P_1, b \neq \lambda\} \cup \\ & \cup \{A \rightarrow bS_2 \mid A \rightarrow bB \in P_1, B \rightarrow \lambda \in P\} \cup \end{aligned} \tag{*}$$

$$\begin{aligned} & \cup \{S_1 \rightarrow w \mid S_2 \rightarrow w \in P_2, S_1 \rightarrow \lambda \in P_1\} \cup \\ & \cup P_2 \end{aligned} \tag{**}$$

# Dimostrazione Teorema di chiusura

## ■ CONCATENAZIONE

Le (\*) e (\*\*) sono state introdotte per garantire la correttezza della grammatica generata, anche in presenza di  $\lambda$ -produzioni in  $G_1$ .

$G_7$  è di tipo ‘3’ se  $G_1$  e  $G_2$  lo sono ed inoltre:

$$L(G_7) = L_1 \cdot L_2$$

ed  $\angle_3$  è chiusa rispetto alla concatenazione.

$$\begin{aligned} P_7 = & \{A \rightarrow bB \mid A \rightarrow bB \in P_1\} \cup \{A \rightarrow bS_2 \mid A \rightarrow b \in P_1, b \neq \lambda\} \cup \\ & \cup \{A \rightarrow bS_2 \mid A \rightarrow bB \in P_1, B \rightarrow \lambda \in P\} \cup \tag{*} \\ & \cup \{S_1 \rightarrow w \mid S_2 \rightarrow w \in P_2, S_1 \rightarrow \lambda \in P_1\} \cup \tag{**} \\ & \cup P_2 \end{aligned}$$

# Dimostrazione Teorema di chiusura

## ■ ITERAZIONE

Costruiamo la grammatica:

$$G_8 = (X, V_1 \cup \{S\}, S, P_8)$$

ove:

$$P_8 = \{S \rightarrow \lambda, S \rightarrow S_1 S\} \cup P_1$$

Data la grammatica  $G_1 = (X, V_1, S_1, P_1)$  che genera  $L_1$ , la grammatica  $G_8$  genera la parola vuota  $\lambda$  e tutte le parole che si possono ottenere per concatenazione di parole generate da  $S_1$  in  $G_1$ . Si ha infatti:

$$S \xrightarrow[G_8]{n} \underbrace{S_1 S_1 \dots S_1}_n S \Rightarrow S_1 S_1 \dots S_1 \xrightarrow{*} w_1 w_2 \dots w_n$$

con:  $w_i \in L_1, i = 1, 2, \dots, n$

# Dimostrazione Teorema di chiusura

## ■ ITERAZIONE

Vediamo per quali classi di grammatiche la  $G_8$  è la grammatica che stiamo cercando:

- se  $G_1$  è di tipo '3',  $G_8$  non è di tipo '3', perché non è lin. dx;
- se  $G_1$  è di tipo '2',  $G_8$  è di tipo '2' e si ha:  $L(G_8) = L_1^*$  e risulta dimostrato che  $\mathcal{L}_2$  è chiusa rispetto all'iterazione.
- se  $G_1$  è di tipo '1',  $G_8$  non è di tipo '1', perché  $S$  compare nella parte destra della produzione  $S \rightarrow S_1S$  ed  $S \rightarrow \lambda$  è una produzione in  $P_8$ ; possiamo trasformare facilmente  $G_8$  nella grammatica equivalente  $G'_8$ , definita come segue:

$$G'_8 = (X, V_1 \cup \{S, S'\}, S, P'_8)$$

$$P'_8 = \{S \rightarrow \lambda \mid S', S' \rightarrow S_1 \mid S_1S'\} \cup P_1$$

ma  $G'_8$  incorre nello stesso problema di interferenza nella definizione dei contesti visto per la concatenazione.

# Dimostrazione Teorema di chiusura

## ■ ITERAZIONE

Vediamo per quali classi di grammatiche la  $G_8$  è la grammatica che stiamo cercando:

- se  $G_1$  è di tipo '0', lo è anche  $G_8$ , ma si incorre nello stesso problema di interferenza nella definizione dei contesti visto nella dimostrazione per la concatenazione.

Il problema dell'interferenza dei contesti, comune agli ultimi due casi -  $G_1$  di tipo  $i$ ,  $i = 0, 1$  - esiste ancora come mostrato dal seguente controesempio:

# Dimostrazione Teorema di chiusura

## ■ ITERAZIONE

Controesempio:

Consideriamo  $G_1$  con  $P_1 = \{S_1 \rightarrow b, bS_1 \rightarrow bc\}$  allora  
 $L_1 = L(G) = \{b\}$  e  $L_1^* = \{b\}^*$

Del resto:  $P_8 = \left\{ S \xrightarrow{(1)} \lambda \mid S', S' \xrightarrow{(2)} S_1 \mid S_1 S', S_1 \xrightarrow{(3)} b, bS_1 \xrightarrow{(4)} bc \right. \right.$

Se consideriamo la derivazione:  $S \xrightarrow{(2)} S' \xrightarrow{(4)} S_1 S' \xrightarrow{(3)} S_1 S_1 \xrightarrow{(5)} bS_1 \xrightarrow{(6)} bc$

si ottiene che:  $bc \in L(G'_8)$  e quindi  $L(G'_8) \neq L_1^*$

Il problema dell'interferenza dei contesti può essere quindi risolto come segue.

# Dimostrazione Teorema di chiusura

## ■ ITERAZIONE

Eliminiamo dapprima le produzioni del tipo  $S_1 \rightarrow \lambda$

Utilizziamo gli insiemi ausiliari  $X'$  e  $X''$  di  $NT$  per evitare interferenze:

$$X' = \{x' \mid x \in X\} \quad X'' = \{x'' \mid x \in X\}$$

$$X' \cap X'' = \emptyset \quad X' \cap X = \emptyset \quad X' \cap V_1 = \emptyset$$

$$X'' \cap X = \emptyset \quad X'' \cap V_1 = \emptyset$$

## ■ Costruiamo due copie di $G_1$ :

$$G'_1 = (X, V_1 \cup X', S_1, P'_1)$$

$$G''_1 = (X, V_1 \cup X'', S_2, P''_1)$$

ove:  $P'_1 = P_1[x'/x]$        $P''_1 = P_1[x''/x]$

# Dimostrazione Teorema di chiusura

## ■ ITERAZIONE

Infine, combiniamo  $G'_1$ ,  $G''_1$  con produzioni che costruiscono sequenze finite di copie di  $S_1$  ed  $S_2$  che si alternano, in modo da ottenere  $L_1^*$ .

Dunque, la grammatica che otteniamo è:

$$G_9 = (X, V_1 \cup X' \cup X'' \cup \{S, S'_1, S_2, S'_2\}, S, P_9)$$

ove:

$$\begin{aligned} P_9 = & \{S \rightarrow \lambda \mid S'_1 \mid S'_2, S'_1 \rightarrow S_1 \mid S_1 S'_2, S'_2 \rightarrow S_2 \mid S_2 S'_1\} \cup \\ & \cup P'_1 \cup P'', \cup \{x' \rightarrow x \mid x \in X\} \cup \{x'' \rightarrow x \mid x \in X\} \end{aligned}$$

Se  $G_1$  è di tipo  $i$ ,  $i = 0, 1$ , lo è anche  $G_9$  e si ha:

$$L(G_9) = L_1^*$$

e risulta dimostrato che  $\mathcal{L}_0$  e  $\mathcal{L}_1$  sono chiuse rispetto all'iterazione.

# Dimostrazione Teorema di chiusura

## ■ ITERAZIONE

Resta da dimostrare che  $\mathcal{L}'_3$  è chiusa rispetto all'iterazione.

Per costruire la nuova grammatica, introduciamo dapprima un nuovo simbolo iniziale  $S$  e la produzione  $S \rightarrow \lambda$  che genera la stringa vuota.

Inoltre, eliminiamo da  $P_1$ , se c'era, la produzione  $S_1 \rightarrow \lambda$  ed aggiungiamo una produzione  $S \rightarrow w$  per ogni produzione "iniziale"  $S_1 \rightarrow w$  in  $P_1$ .

# Dimostrazione Teorema di chiusura

## ■ ITERAZIONE

Infine, per ogni produzione la cui parte destra contiene solo un terminale, del tipo  $A \rightarrow b$ , nell'insieme delle produzioni che stiamo costruendo, aggiungiamo la produzione  $A \rightarrow bS$  in modo che, avendo derivato una forma di frase del tipo  $w_1w_2\dots w_j A$  tale che ogni sottostringa  $w_1, w_2, \dots, w_j = w'_j b$  è una parola di  $L_1$ , abbiamo la possibilità di terminare la derivazione con  $w_1w_2\dots w_j$  o di continuarla generando la forma di frase  $w_1w_2\dots w_j S$ , che consente di generare una parola più lunga di  $L_1^*$ .

# Dimostrazione Teorema di chiusura

## ■ ITERAZIONE

Si noti che, per garantire la correttezza della grammatica generata anche in presenza di  $\lambda$ -produzioni in  $G_1$ , occorre aggiungere una produzione  $A \rightarrow bS$  per ogni produzione  $A \rightarrow bB$  nell'insieme delle produzioni che stiamo costruendo, quando  $B \rightarrow \lambda$  è pure una produzione di tale insieme, alla stregua di quanto fatto per la concatenazione in  $\mathcal{L}_3$ .

Costruiamo dunque la grammatica:  $G_{10} = (X, V_1 \cup \{S\}, S, P_{10})$  ove:  $P_{10} = \{S \rightarrow \lambda\} \cup (P_1 - \{S_1 \rightarrow \lambda\}) \cup \{S \rightarrow w \mid S_1 \rightarrow w \in P_1\} \cup \{A \rightarrow bS \mid A \rightarrow b \in P_{10}, b \neq \lambda\} \cup \{A \rightarrow bS \mid A \rightarrow bB \in P_{10}, B \rightarrow \lambda \in P_{10}\}$

Si noti che la definizione di  $P_{10}$  è ricorsiva. Se  $G_1$  è di tipo '3', lo è anche  $G_{10}$  e si ha:  $L(G_{10}) = L_1^*$   
e risulta dimostrato che  $\mathcal{L}_3$  è chiusa rispetto all'iterazione.

# Teorema di chiusura

- Per la loro importanza pratica, riassumiamo le modalità di costruzione delle grammatiche che generano i linguaggi *unione*, *concatenazione*, *iterazione* di  $L_1$  ed  $L_2$  nella Tavola che segue.

# Teorema di chiusura

	<i>UNIONE</i>	<i>CONCATENAZIONE</i>	<i>ITERAZIONE</i>
$\mathcal{L}_0$	$G_3 = (X, V, S, P_3)$ $P_3 = \{S \rightarrow S_1, S \rightarrow S_2\} \cup P_1 \cup P_2$	$X' = \{x' \mid x \in X\}$ $X'' = \{x'' \mid x \in X\}$ $X' \cap X'' = \emptyset$ $X' \cap X = \emptyset$ $X' \cap V = \emptyset$ $X'' \cap X = \emptyset$ $X'' \cap V = \emptyset$ $P'_1 = P_1[x'/x]$ $P''_2 = P_2[x''/x]$	<p>Eliminiamo le produzioni del tipo:  <math>S_1 \rightarrow \lambda</math></p> $X' = \{x' \mid x \in X\}$ $X'' = \{x'' \mid x \in X\}$ $X' \cap X'' = \emptyset$ $X' \cap X = \emptyset$ $X' \cap V_1 = \emptyset$ $X'' \cap X = \emptyset$ $X'' \cap V_1 = \emptyset$ <p>Costruiamo due copie di <math>G_1</math>:</p> $G'_1 = (X, V_1 \cup X', S_1, P'_1)$ $G''_1 = (X, V_1 \cup X'', S_2, P''_1)$ $P'_1 = P_1[x'/x]$ $P''_1 = P_1[x''/x]$
$\mathcal{L}_1$		$G_6 = (X, V \cup X' \cup X'', S, P_6)$ $P_6 = \{S \rightarrow S_1 S_2\} \cup P'_1 \cup P''_2 \cup$ $\cup \{x' \rightarrow x \mid x \in X\} \cup$ $\cup \{x'' \rightarrow x \mid x \in X\}$	$G_9 = (X, V_1 \cup X' \cup X'' \cup \{S, S'_1, S'_2, S'_2\}, S, P_9)$ $P_9 = \{S \rightarrow \lambda \mid S'_1   S'_2, S'_1 \rightarrow S_1   S_1 S'_2, S'_2 \rightarrow S_2   S_2 S'_1\} \cup$ $\cup P'_1 \cup P''_2 \cup \{x' \rightarrow x \mid x \in X\} \cup \{x'' \rightarrow x \mid x \in X\}$
$\mathcal{L}_2$		$G_5 = (X, V, S, P_5)$ $P_5 = \{S \rightarrow S_1 S_2\} \cup P_1 \cup P_2$	$G_8 = \{X, V_1 \cup \{S\}, S, P_8\}$ $P_8 = \{S \rightarrow \lambda \mid S_1 S\} \cup P_1$
$\mathcal{L}_3$	$G_4 = (X, V, S, P_4)$ $P_4 = \{S \rightarrow w \mid S_1 \rightarrow w \in P_1\} \cup$ $\cup \{S \rightarrow w \mid S_2 \rightarrow w \in P_2\} \cup$ $\cup P_1 \cup P_2$	$G_7 = (X, V - \{S\}, S_1, P_7)$ $P_7 = \{A \rightarrow bB \mid A \rightarrow bB \in P_1\} \cup$ $\cup \{A \rightarrow bS_2 \mid A \rightarrow b \in P_1, b \neq \lambda\} \cup$ $\cup \{A \rightarrow bS \mid A \rightarrow bB \in P_1, B \rightarrow \lambda \in P_1\} \cup$ $\cup \{S_1 \rightarrow w \mid S_2 \rightarrow w \in P_2, S_1 \rightarrow \lambda \in P_1\} \cup P_2$	$G_{10} = (X, V_1 \cup \{S\}, S, P_{10})$ $P_{10} = \{S \rightarrow \lambda\} \cup (P_1 - \{S_1 \rightarrow \lambda\}) \cup$ $\cup \{S \rightarrow w \mid S_1 \rightarrow w \in P_1\} \cup$ $\cup \{A \rightarrow bS \mid A \rightarrow b \in P_{10}, b \neq \lambda\} \cup$ $\cup \{A \rightarrow bS \mid A \rightarrow bB \in P_{10}, B \rightarrow \lambda \in P_1\} \cup P_2$

# Altri teoremi di chiusura

- La classe dei linguaggi **lineari destri** (tipo ‘3’) è **chiusa** rispetto al **complemento** ed all’**intersezione**.
- La classe dei linguaggi **non contestuali** (tipo ‘2’) non è **chiusa** rispetto al **complemento** ed all’**intersezione**.
- La classe dei linguaggi **contestuali** (tipo ‘1’) è **chiusa** rispetto al **complemento** (e dunque anche rispetto all’**intersezione**).
- La classe dei linguaggi di tipo ‘0’ non è **chiusa** rispetto al **complemento**.

# Dimostrazione

- La classe dei linguaggi lineari destri (tipo '3') è chiusa rispetto al complemento ed all'intersezione.

Dimostreremo prossimamente la chiusura di  $\mathcal{L}_3$  rispetto al complemento. Dimostriamola rispetto all'intersezione:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}} \quad (\text{Leggi di De Morgan})$$

La chiusura di  $\mathcal{L}_3$  rispetto all'intersezione discende direttamente da questo risultato.

# Dimostrazione

- La classe dei linguaggi non contestuali (tipo ‘2’) non è chiusa rispetto al complemento ed all’intersezione.

Consideriamo i linguaggi

$$L_1 = \{a^n b^n c^m \mid n, m > 0\} \qquad L_2 = \{a^n b^m c^m \mid n, m > 0\}$$

$L_1$  e  $L_2$  sono linguaggi liberi, mentre  $L_1 \cap L_2$  non lo è:

$$L_1 \cap L_2 = \{a^k b^k c^k \mid k > 0\}$$

Il complemento è:  $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

Dunque, se  $L_1, L_2 \in \mathcal{L}_2$  e se  $\mathcal{L}_2$  fosse chiusa rispetto al complemento, ...

# Dimostrazione

- La classe dei linguaggi contestuali (tipo ‘1’) è chiusa rispetto al complemento (e dunque anche rispetto all’intersezione).

Un risultato recente ha stabilito che  $L_1$  è chiusa rispetto al complemento (e quindi all’intersezione). Non se ne conosce la dimostrazione.

- La classe dei linguaggi di tipo ‘0’ non è chiusa rispetto al complemento.

Non lo dimostriamo

# L'operazione di riflessione

## ■ Definizione di stringa riflessa

Sia  $w$  una parola su un alfabeto  $X = \{x_1, x_2, \dots, x_k\}$ ,  $w = x_{i_1} x_{i_2} \dots x_{i_{n-1}} x_{i_n}$ . Dicesi **stringa riflessa** (o **riflessione**) di  $w$  la stringa

$$w^R = x_{i_n} x_{i_{n-1}} \dots x_{i_2} x_{i_1}$$

## ■ Operazione di riflessione

Sia  $w$  una parola su un alfabeto  $X = \{x_1, x_2, \dots, x_k\}$  e sia  $w^R$  la stringa riflessa di  $w$ . L'operazione che trasforma  $w$  in  $w^R$  è detta **operazione di riflessione**.

# L'operazione di riflessione

## ■ Definizione di parola palindromica

Un *palindromo* (o *parola palindromica*) è una parola la cui lettura a ritroso riproduce la parola di partenza:

$$w \text{ palindromo} \stackrel{\text{def}}{\Leftrightarrow} w = w^R$$

Un palindromo è dunque una parola che coincide con la sua riflessione.

# Esempio

- Alcuni palindromi sull'alfabeto  $\{a,b,\dots,z\}$  sono:
  - *a*;
  - *ii* (plurale di *io*?);
  - *non, ala, ara, ici*;
  - *osso, alla, arra*;
  - *radar, alalà, arerà* (ignorando l'accento);
  - *osesso, ingegni*;
  - *avallava, ovattavo*;
  - *onorarono*;
  - *accavallavacca, accumolomucca*;
  - *fecì nulla all'Unicef, ogni tela male tingo* (ignorando spazi bianchi, punteggiatura e differenza tra maiuscole e minuscole).

# Palindromi

- I palindromi (su un qualunque alfabeto) sono di due tipi:
  - palindromi di lunghezza pari: hanno un “asse di simmetria” costituito dalla parola vuota
  - palindromi di lunghezza dispari: hanno un “asse di simmetria” costituito da uno dei simboli dell’alfabeto
- Più precisamente, si ha la seguente caratterizzazione (senza dimostrazione):

# Teorema

- Sia  $w$  una parola su un alfabeto  $X$ .  $w$  è palindromo se e solo se

$$w = \alpha x \alpha^R, \quad x \in X \cup \{\lambda\}$$

# Teorema

- La classe dei linguaggi non contestuali (tipo ‘2’) è chiusa rispetto all’operazione di riflessione.

## Dimostrazione

Sia  $G_1 = (X, V_1, S_1, P_1)$  una grammatica non contestuale.

Dobbiamo dimostrare che:

$L(G_1)$  non contestuale  $\Rightarrow (L(G_1))^R = \{w^R \mid L(G_1)\}$   
è non contestuale.

Costruiamo la grammatica:  $G_{11} = (X, V_1, S_1, P_{11})$

ove:  $P_{11} = \{A \rightarrow w^R \mid A \rightarrow w \in P_1\}$

Risulta allora:  $L(G_{11}) = (L(G_1))^R$

Quindi se in  $P_1$  abbiamo la produzione:  $A \rightarrow BaC$

in  $P_{11}$  avremo la produzione:  $A \rightarrow CaB$

## Esercizi

# Utilizzo proprietà di chiusura

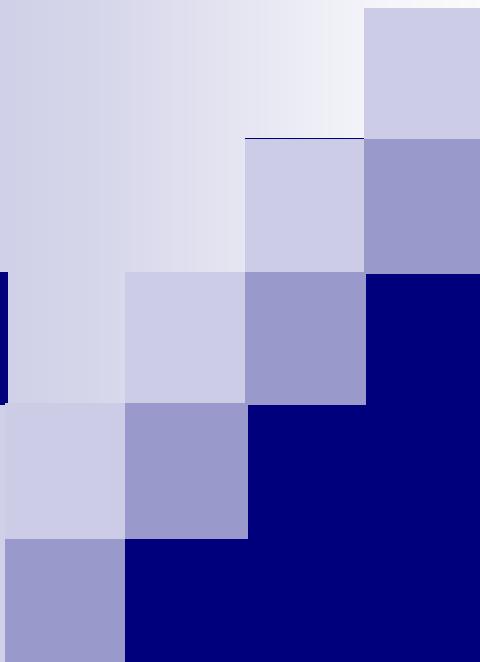
## *Schema di Ragionamento per Utilizzo Proprietà di Chiusura*

Siano  $L$ ,  $L_1$  ed  $L_2$  tre linguaggi tali che:  $L = \alpha(L_1, L_2)$  ove  $\alpha = \cup, \cdot$

<i>Esatto</i>	<i>Errato</i>
<p>Supponiamo che <math>L_2 \in \mathcal{L}_i</math></p> <p>Se <math>L \notin \mathcal{L}_i</math> allora <math>L_1 \notin \mathcal{L}_i</math></p>	<p>Se <math>L_j \notin \mathcal{L}_j</math> allora <math>L \notin \mathcal{L}_j</math></p>

$$\text{Esempio: } a^n b^m = \underset{n,m>0}{a^n b^n} \cup \underset{n \neq m}{a^n b^m}$$

Lineare dx    Non sono lineari dx



# Linguaggi di Programmazione

## Capitolo 6 – Automi a stati finiti (deterministici e non deterministici)

# Automi a stati finiti deterministici

- Definizione di automa a stati finiti o accettore a stati finiti o FSA

Sia  $X$  un alfabeto. Un *automa a stati finiti* (FSA) è una quadrupla:

$$M = (Q, \delta, q_0, F)$$

ove:

- $X$  è detto *alfabeto di ingresso*;
- $Q$  è un insieme finito e non vuoto di *stati*;
- $\delta$  è una funzione da  $Q \times X$  in  $Q$ , detta *funzione di transizione*:

$$\delta : Q \times X \rightarrow Q$$

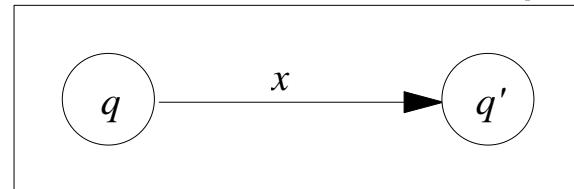
- $q_0$  è lo *stato iniziale*;
- $F \subseteq Q$  è l'insieme degli *stati di accettazione* o *finali*.

# Automi a stati finiti deterministici

- Talora i valori della funzione di transizione  $\delta$  non sono definiti per tutte le coppie (stato-simbolo di ingresso)  $(q, x)$ . In tal caso, si dice che  $\delta$  è una **funzione parziale** o definita parzialmente. Questo significa che la lettura di  $x$  dà luogo in  $q$  ad un comportamento dell'automa che non si ritiene utile descrivere ai fini del riconoscimento (nel senso che produrrebbe stringhe non accettate).
- Evidentemente questo fatto può essere descritto in modo equivalente, seguendo la definizione data di automa a stati finiti, passando da  $q$ , per effetto di  $x$ , in uno stato dal quale non si possa mai raggiungere uno stato finale (**stato pozza**).

# Rappresentazione di FSA

- Un FSA può essere rappresentato mediante:
    - *Grafo degli Stati* o *Diagramma di Transizione* o *Diagramma di Stato*
- È una rappresentazione grafica in cui:
- ogni stato  $q \in Q$  è rappresentato da un cerchio (*nodo*) con etichetta  $q$ ;
  - lo stato iniziale (nodo  $q_0$ ) ha un arco orientato entrante libero (ossia, che non proviene da nessun altro nodo);
  - per ogni stato  $q \in Q$  e per ogni simbolo  $x$  dell'alfabeto di ingresso,  $x \in X$ , se  $\delta(q, x) = q'$ , esiste un arco orientato etichettato con  $x$  uscente dal nodo  $q$  ed entrante nel nodo  $q'$ .



# Rappresentazione di FSA

- Un FSA può essere rappresentato mediante:

- *Tavola di Transizione*

È una tabella in cui sono riportati gli stati sulle righe e i simboli dell'alfabeto di ingresso sulle colonne.

Per ogni coppia (stato-ingresso) si legge nella tavola lo stato successivo:

$\delta$	$x_1$	$x_2$	...	....	$x_n$
$q_0$	$q_0^1$	$q_0^2$	...	...	$q_0^n$
$q_1$	$q_1^1$	$q_1^2$	...	...	$q_1^n$
...	...	...	...	...	...
...	...	...	...	...	...
$q_m$	$q_m^1$	$q_m^2$	...	...	$q_m^n$

dove l'alfabeto di ingresso e l'insieme degli stati sono rispettivamente:

$$X = \{x_1, x_2, \dots, x_n\} \text{ e } Q = \{q_0, q_1, \dots, q_m\}$$

Quindi si ha che:

$$\delta(q_i, x_j) = q_i^j \text{ con } q_i, q_i^j \in Q, x_j \in X$$

# Estensione della funzione di transizione

- Si può definire un'estensione della funzione di transizione  $\delta$  come segue:

**Definizione:  $\delta^*$  per FSA**

Dato un FSA  $M = (Q, \delta, q_0, F)$  con alfabeto di ingresso  $X$ , definiamo per induzione la funzione:

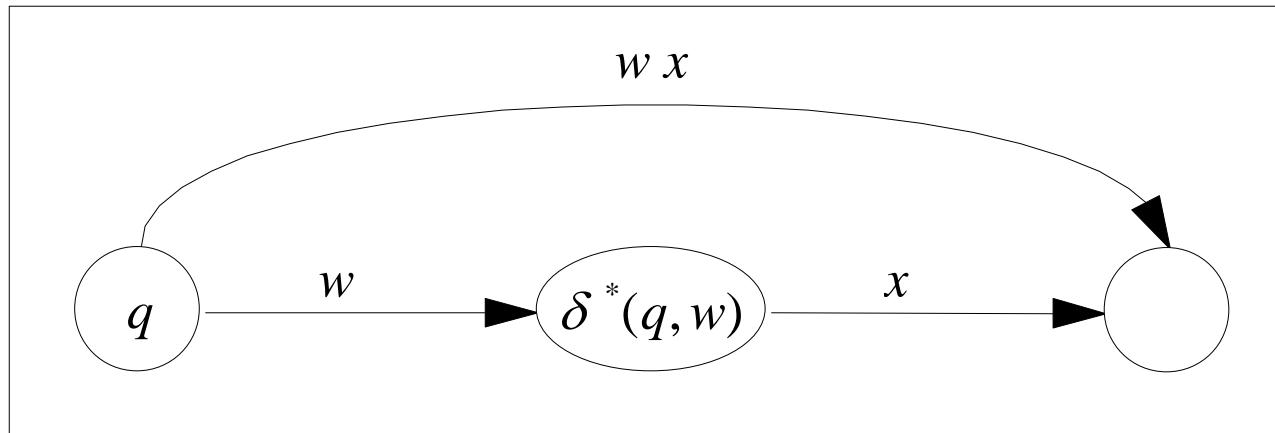
$$\delta^* : Q \times X^* \rightarrow Q$$

tale che  $\delta^*(q, w)$ , per  $q \in Q$  e  $w \in X^*$ , sia lo stato in cui  $M$  si porta avendo in ingresso la parola  $w$  su  $X$  e partendo dallo stato  $q$ .

$$\begin{cases} \delta^*(q, \lambda) = q \\ \delta^*(q, wx) = \delta(\delta^*(q, w), x) \end{cases} \quad \text{per ogni } q \in Q, x \in X, w \in X^*$$

# Estensione della funzione di transizione

- La figura sottostante riporta una descrizione grafica della definizione precedente.



# Parola accettata o riconosciuta da un FSA

## ■ Definizione

Sia  $M = (Q, \delta, q_0, F)$  un FSA con alfabeto di ingresso  $X$ . Una *parola*  $w \in X^*$  è *accettata* (o *riconosciuta*) da  $M$  se, partendo dallo stato iniziale  $q_0$ , lo stato  $q$  in cui l'automa si porta alla fine della sequenza di ingresso  $w$  è uno stato finale.

$$w \text{ accettata} \stackrel{\text{def}}{\iff} \delta^*(q_0, w) \in F$$

# Linguaggio accettato o riconosciuto da un FSA

## ■ Definizione

Sia  $M = (Q, \delta, q_0, F)$  un FSA con alfabeto di ingresso  $X$ . Il *linguaggio accettato* o *riconosciuto* da  $M$  è il seguente sottoinsieme di  $X^*$ :

$$T(M) = \{ w \in X^* \mid \delta^*(q_0, w) \in F \}$$

(l'insieme delle parole accettate da  $M$ ).

# FSA equivalenti

## ■ Definizione

Sia  $M_1 = (Q_1, \delta_1, q_1, F_1)$  ed  $M_2 = (Q_2, \delta_2, q_2, F_2)$  due FSA di alfabeto di ingresso  $X$ .  $M_1$  ed  $M_2$  si dicono *equivalenti* se:

$$T(M_1) = T(M_2)$$

# Linguaggi a stati finiti

## ■ Definizione

Dato un alfabeto  $X$ , un linguaggio  $L$  su  $X$  è un *linguaggio a stati finiti* (o FSL - Finite State Language) se esiste un FSA  $M$  con alfabeto di ingresso  $X$  tale che  $L = T(M)$ .

# Classe dei linguaggi a stati finiti

## ■ Definizione

Di seguito la definizione della *Classe dei Linguaggi a Stati Finiti*

$$\mathcal{L}_{FSL} = \left\{ L \in 2^{X^*} \mid \exists M, \text{ } M \text{ è un } \mathcal{FSA}: L = T(M) \right\}$$

# Proposizione

- I linguaggi a stati finiti sono *chiusi rispetto al complemento.*

## Dimostrazione

Sia  $L \in \mathcal{L}_{FSL}$  un linguaggio a stati finiti sull'alfabeto  $X$ . Dalla definizione di linguaggio a stati finiti,  $L = T(M)$ , ove  $M = (Q, \delta, q_0, F)$ .

Consideriamo il complemento di  $L$ :  $\overline{L} = X^* - L$  e l'automa a stati finiti  $\overline{M} = (Q, \delta, q_0, Q - F)$ . Si ha:  $\overline{L} = T(\overline{M})$  (si dimostra per induzione sulla lunghezza di una parola  $w$ ).

$M$  è un automa stati finiti con funzione di transizione  $\delta$  **definita totalmente** sul proprio dominio.

# Automa a stati finiti non deterministico o accettore a stati finiti non deterministico

## ■ Definizione

Un *automa a stati finiti non deterministico* (NDA) con alfabeto di ingresso  $X$  è una quadrupla:

$$M = (Q, \delta, q_0, F)$$

ove:

- per  $Q$ ,  $q_0$  ed  $F$  valgono le definizioni date per gli FSA;
- $\delta : Q \times X \rightarrow 2^Q$  è la funzione di transizione che assegna ad ogni coppia (stato-simbolo di ingresso)  $(q, x)$  un insieme  $\delta(q, x) \subseteq Q$  di possibili stati successivi.

# Osservazione

- Un NDA è un FSA con l'unica eccezione che, in corrispondenza di una coppia (stato-simbolo di ingresso)  $(q, x)$ , vi è un insieme di stati in cui l'automa può transitare (*stati successivi possibili*).

# Estensione della funzione di transizione per un NDA

- Come per gli FSA si può definire un'estensione della funzione di transizione  $\delta$  come segue:

**Definizione:  $\delta^*$  per NDA**

Dato un NDA  $M = (Q, \delta, q_0, F)$  con alfabeto di ingresso  $X$ , definiamo per induzione la funzione:

$$\delta^* : 2^Q \times X^* \rightarrow 2^Q$$

$$\begin{cases} \delta^*(p, \lambda) = p \\ \delta^*(p, wx) = \bigcup_{q \in \delta^*(p, w)} \delta(q, x) \end{cases} \quad \text{per ogni } p \in 2^Q \ (p \subset Q), \ x \in X, \ w \in X^*$$

# Osservazione

- Analogamente a quanto fatto per gli FSA, si dovrebbero riformulare le definizioni di parola accettata e di linguaggio accettato da un NDA.
- La complicazione, rinveniente dalla computazione non deterministica dello stato successivo in cui un NDA transita, comporta che una stessa parola può indurre cammini multipli attraverso un NDA, alcuni che terminano in stati di accettazione, altri che terminano in stati di non accettazione.

## Esempio

# Parola accettata o riconosciuta da un NDA

## ■ Definizione

Sia  $M = (Q, \delta, q_0, F)$  un NDA con alfabeto di ingresso  $X$ . Una parola  $w \in X^*$  è **accettata** (o **riconosciuta**) da  $M$  se, partendo dallo stato iniziale  $q_0$ , esiste almeno un modo per  $M$  di portarsi in uno stato di accettazione alla fine della sequenza di ingresso  $w$ . In formule:

$$w \text{ accettata} \stackrel{\text{def}}{\iff} \exists p : p \in \delta^*(\{q_0\}, w) \cap F \iff \delta^*(\{q_0\}, w) \cap F \neq \emptyset$$

# Esempio

- Sia dato l'NDA dell'es. precedente. Calcoliamo  $\delta^*(\{S_1\}, aba)$

$$\delta^*(\{S_1\}, aba) = \bigcup_{q \in \delta^*(\{S_1\}, ab)} \delta(q, a)$$

$$\delta^*(\{S_1\}, ab) = \bigcup_{q' \in \delta^*(\{S_1\}, a)} \delta(q', b)$$

$$\delta^*(\{S_1\}, a) = \bigcup_{q'' \in \delta^*(\{S_1\}, \lambda)} \delta(q'', a)$$

$$\delta^*(\{S_1\}, \lambda) = \{S_1\}$$

$$\delta^*(\{S_1\}, a) = \delta(S_1, a) = \{A, S_2\}$$

$$\delta^*(\{S_1\}, ab) = \delta(A, b) \cup \delta(S_2, b) = \{A\} \cup \emptyset = \{A\}$$

$$\delta^*(\{S_1\}, aba) = \delta(A, a) = \{A, S_2\}$$

Poiché  $F = \{A\}$ , si ha:

$$\delta^*(\{S_1\}, aba) \cap F = \{A\} \neq \emptyset$$

e  $w = aba$  è accettata da  $M_1$ .

# Linguaggi accettati o riconosciuto da un NDA

## ■ Definizione

Sia  $M = (Q, \delta, q_0, F)$  un NDA con alfabeto di ingresso  $X$ . Il *linguaggio accettato* o *riconosciuto* da  $M$  è l'insieme delle parole su  $X$  accettate da  $M$ :

$$T(M) = \left\{ w \in X^* \mid \delta^*(\{q_0\}, w) \cap F \neq \emptyset \right\}$$

(è l'insieme delle parole  $w$  per le quali esiste almeno un cammino, etichettato con lettere di  $w$  nell'ordine da sinistra a destra, attraverso il diagramma degli stati che porta  $M$  dallo stato iniziale ad uno degli stati di accettazione).

# NDA equivalenti

## ■ Definizione

Siano  $M_1 = (Q_1, \delta_1, q_1, F_1)$  ed  $M_2 = (Q_2, \delta_2, q_2, F_2)$  due NDA di alfabeto di ingresso  $X$ .  $M_1$  ed  $M_2$  si dicono *equivalenti* se:

$$T(M_1) = T(M_2)$$

# Classe dei linguaggi riconosciuti da automi a stati finiti non deterministici

## ■ Definizione

Di seguito la definizione della *Classe dei Linguaggi riconosciuti da automi a Stati Finiti non deterministici*

$$\mathcal{L}_{NDL} = \left\{ L \in 2^{X^*} \mid \exists M, \text{ } M \text{ è un NDA: } L = T(M) \right\}$$

# Equivalenza delle classi di linguaggi accettati da automi a stati finiti deterministici e non deterministici.

## ■ Teorema

Le classi dei linguaggi  $\mathcal{L}_{FSL}$  e  $\mathcal{L}_{NDL}$  sono equivalenti (1<sup>a</sup> formulazione).

■ Sia  $L$  un linguaggio su  $X$ .  $L$  è un linguaggio a stati finiti se e solo se  $L = T(M)$  per qualche NDA  $M$  (2<sup>a</sup> formulazione).

## Dimostrazione (2<sup>a</sup> formulazione)

$\Rightarrow)$  Sia  $L \in 2^{X^*}$  ed  $L \in \mathcal{L}_{FSL}$ . Dalla definizione di linguaggio a stati finiti, si ha:

$\exists M_1 : M_1$  è un FSA,  $M_1 = (Q_1, \delta_1, q_1, F_1)$  con alfabeto di ingresso  $X$ :  $L = T(M_1)$ .

# Equivalenza delle classi di linguaggi accettati da automi a stati finiti deterministici e non deterministici.

## ■ Dimostrazione (2<sup>a</sup> formulazione, continuazione)

Sulla base di  $M_1$ , definiamo il seguente NDA con alfabeto di ingresso  $X$ :

$$M_2 = (Q_2, \delta_2, q_2, F_2)$$

ove:

- $Q_2 = Q_1$ ;
- $\delta_2$  è così definita:

$$\delta_2 : Q_2 \times X \rightarrow 2^{Q_2},$$

$$\delta_2(q, x) = \{\delta_1(q, x)\} \quad \forall q \in Q_2 = Q_1, \quad x \in X;$$

- $q_2 = q_1$ ;
- $F_2 = F_1$

$M_2$  è un NDA ed inoltre accetta lo stesso linguaggio accettato da  $M_1$ , ossia si ha:  $T(M_2) = T(M_1)$

# Equivalenza delle classi di linguaggi accettati da automi a stati finiti deterministici e non deterministici.

## ■ Dimostrazione (2<sup>a</sup> formulazione)

$\Leftrightarrow$ ) Sia  $L \in 2^{X^*}$  ed  $L \in \mathcal{L}_{NDL}$ . Dalla definizione della classe di linguaggi  $\mathcal{L}_{NDL}$ , si ha:

$\exists M, M$  è un NDA,  $M = (Q, \delta, q_0, F)$

con alfabeto di ingresso  $X : L = T(M)$ .

Si può definire allora il seguente algoritmo per la costruzione di un FSA equivalente all'NDA  $M$ :

# Trasformazione di un automa a stati finiti non deterministico in un automa deterministico equivalente

- Sia  $M = (Q, \delta, q_0, F)$  un automa accettore a stati finiti non deterministico di alfabeto di ingresso  $X$ .  $M$  può essere trasformato in un automa deterministico  $M'$  di alfabeto di ingresso  $X$  come segue:  $M' = (Q', \delta', q'_0, F')$  ove:

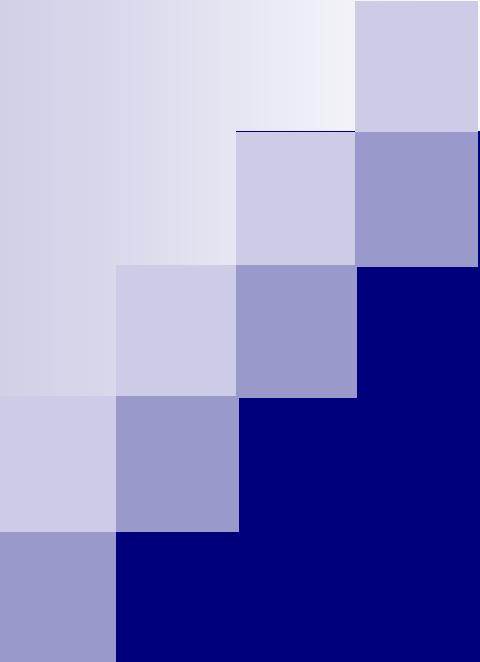
- $Q' = 2^Q$
- $q'_0 = \{q_0\}$
- $F' = \{p \subseteq Q \mid p \cap F \neq \emptyset\}$
- $\delta': Q' \times X \rightarrow Q'$     $\exists' \quad \forall q' = \{q_1, q_2, \dots, q_i\} \in Q', \quad \forall x \in X :$   
$$\delta'(q', x) = \delta'(\{q_1, q_2, \dots, q_i\}, x) = \bigcup_{j=1}^i \delta(q_j, x) = \bigcup_{q \in q'} \delta(q, x).$$

Si può dimostrare che  $M'$  è equivalente a  $M$ , ossia che

$$T(M') = T(M)$$

# Esercizi

Esercizi



# Linguaggi di Programmazione

Capitolo 7 – Linguaggi regolari,  
espressioni regolari e teorema di  
Kleene.

# Descrizione algebrica dei linguaggi lineari destri

Grammatiche generative	Automi a stati finiti accettori	Espressioni regolari
<i>Generatori</i>	<i>Riconoscitori</i>	
procedure effettive per		
<i>Costruire stringhe che appartengono al linguaggio</i>	<i>Stabilire se una stringa appartiene o no ad un linguaggio</i>	
<i>Definizioni operazionali</i>	<i>Definizione denotazionale</i>	

# Linguaggi regolari ed espressioni regolari

## ■ Definizione di linguaggio regolare

Sia  $X$  un alfabeto finito. Un *linguaggio*  $L \subseteq X^*$  è **regolare** se:

- $L$  è finito;

oppure

- $L$  può essere ottenuto per induzione utilizzando una delle seguenti operazioni:

- $L = L_1 \cup L_2$ , con  $L_1, L_2$  regolari;
- $L = L_1 \cdot L_2$ , con  $L_1, L_2$  regolari;
- $L = L_1^*$ , con  $L_1$  regolare.

Si noti che  $\emptyset$  e  $\{\lambda\}$  sono linguaggi regolari.

Denotiamo con  $\mathcal{L}_{REG}$  la classe dei linguaggi regolari.

# Espressioni regolari

## ■ Definizione di espressione regolare

Sia  $X$  un alfabeto finito. Una stringa  $R$  di alfabeto  $X \cup \{\lambda, +, *, ., \emptyset, (, )\}$  (con  $X \cap \{\lambda, +, *, ., \emptyset, (, )\} = \emptyset$ ) è una **espressione regolare** di alfabeto  $X$  se e solo se vale una delle seguenti condizioni:

- (i)  $R = \emptyset$
- (ii)  $R = \lambda$
- (iii)  $R = a$ , per ogni  $a \in X$
- (iv)  $R = (R_1 + R_2)$ , con  $R_1, R_2$  espressioni regolari di alfabeto  $X$
- (v)  $R = (R_1 \cdot R_2)$ , con  $R_1, R_2$  espressioni regolari di alfabeto  $X$
- (vi)  $R = (R_1)^*$ , con  $R_1$  espressione regolare di alfabeto  $X$

# Espressioni regolari e linguaggi regolari

- Ad ogni espressione regolare  $R$  corrisponde un linguaggio regolare  $S(R)$  definito nel modo seguente:

<i>Espressione regolare</i>	<i>Linguaggio regolare corrispondente</i>
$\emptyset$	$\emptyset$
$\lambda$	$\{\lambda\}$
$a$	$\{a\}$
$(R_1 + R_2)$	$S(R_1) \cup S(R_2)$
$(R_1 \cdot R_2)$	$S(R_1) \cdot S(R_2)$
$(R_1)^*$	$(S(R_1))^*$

# Espressioni regolari e linguaggi regolari

- Da una espressione regolare si possono eliminare le coppie di parentesi superflue, tenuto conto che le operazioni ai punti (iv), (v) e (vi) sono elencate in ordine crescente di priorità.

# Proposizione

- Un linguaggio su  $X$  è regolare se e solo se corrisponde ad una espressione regolare su  $X$ .  
Quindi, denotato con  $\mathcal{R}$  l'insieme delle espressioni regolari di alfabeto  $X$ , definiamo la funzione:

$$S : \mathcal{R} \rightarrow 2^{X^*}$$

che ad ogni espressione regolare  $R$  associa il corrispondente linguaggio regolare  $S(R)$ . Si ha dunque:

$$\mathcal{L}_{REG} = \left\{ L \in 2^{X^*} \mid \exists R \in \mathcal{R}, L = S(R) \right\}$$

## Esempio

# Osservazione

- Un linguaggio regolare può essere descritto da più di una espressione regolare, ossia  $S : \mathcal{R} \rightarrow 2^{X^*}$  non è una funzione iniettiva.

# Esempio

- Il linguaggio costituito da tutte le parole su  $X = \{a, b\}$  con  $a$  e  $b$  che si alternano (e che cominciano e terminano con  $b$ ) può essere descritto sia dall'espressione regolare

$$b \cdot (ab)^*$$

sia da

$$(ba)^* b$$

# Definizione di espressioni regolari equivalenti

- Due espressioni regolari  $R_1$  e  $R_2$  su  $X$  sono *equivalenti* (per abuso di notazione, scriviamo  $R_1 = R_2$ ) se e solo se  $S(R_1) = S(R_2)$ .

Relativamente all'esempio precedente, si ha, pertanto:

$$b \cdot (ab)^* = (ba)^* b$$

# Proprietà delle espressioni regolari

- Siano  $R_1$ ,  $R_2$  ed  $R_3$  espressioni regolari di alfabeto  $X$ , risulta:
  1.  $(R_1 + R_2) + R_3 = R_1 + (R_2 + R_3) = R_1 + R_2 + R_3$  Prop. associativa
  2.  $R_1 + R_2 = R_2 + R_1$  Prop. commutativa
  3.  $R_1 + \emptyset = \emptyset + R_1 = R_1$   $\emptyset$  è l'elemento neutro rispetto all'operazione “+” definita sulle espressioni regolari
  4.  $R_1 + R_1 = R_1$  Idempotenza
  5.  $(R_1 \cdot R_2) \cdot R_3 = R_1 \cdot (R_2 \cdot R_3) = R_1 \cdot R_2 \cdot R_3$  Prop. associativa
  6.  $R_1 \cdot R_2 \neq R_2 \cdot R_1$  In generale
  7.  $R_1 \cdot \lambda = \lambda \cdot R_1 = R_1$   $\lambda$  è l'elemento neutro rispetto all'operazione “.” definita sulle espressioni regolari

# Proprietà delle espressioni regolari

- Siano  $R_1$ ,  $R_2$  ed  $R_3$  espressioni regolari di alfabeto  $X$ , risulta:
  8.  $R_1 \cdot \emptyset = \emptyset \cdot R_1 = \emptyset$   $\emptyset$  è l'elemento assorbente rispetto all'operazione “.”.
  9.  $R_1 \cdot (R_2 + R_3) = (R_1 \cdot R_2) + (R_1 \cdot R_3)$  Proprietà distributiva di “.” rispetto all'operazione “+” (distributività sinistra).
  10.  $(R_1 + R_2) \cdot R_3 = (R_1 \cdot R_3) + (R_2 \cdot R_3)$  Proprietà distributiva di “.” rispetto all'operazione “+” (distributività destra).
  11.  $(R_1)^* = (R_1)^* \cdot (R_1)^* = ((R_1)^*)^* = (\lambda + R_1)^*$
  12.  $(\emptyset)^* = (\lambda)^* = \lambda$

# Proprietà delle espressioni regolari

- Siano  $R_1$ ,  $R_2$  ed  $R_3$  espressioni regolari di alfabeto  $X$ , risulta:

$$13. \quad (R_1)^* = \lambda + R_1 + R_1^2 + \dots + R_1^n + (R_1^{n+1} \cdot R_1^*)$$

Caso particolare:

$$(R_1)^* = \lambda + R_1 \cdot R_1^* = \lambda + R_1^* \cdot R_1$$

$$14. \quad (R_1 + R_2)^* = (R_1^* + R_2^*)^* = (R_1^* \cdot R_2^*)^* = \\ = (R_1^* \cdot R_2)^* \cdot R_1^* = R_1^* \cdot (R_2 \cdot R_1^*)^*$$

$$15. \quad (R_1 + R_2)^* \neq R_1^* + R_2^* \quad \text{In generale}$$

$$16. \quad R_1^* \cdot R_1 = R_1 \cdot R_1^*$$

$$17. \quad R_1 \cdot (R_2 \cdot R_1)^* = (R_1 \cdot R_2)^* \cdot R_1$$

# Proprietà delle espressioni regolari

- Siano  $R_1$ ,  $R_2$  ed  $R_3$  espressioni regolari di alfabeto  $X$ , risulta:

$$18. \left( R_1^* \cdot R_2 \right)^* = \lambda + (R_1 + R_2)^* \cdot R_2$$

$$19. \left( R_1 \cdot R_2^* \right)^* = \lambda + R_1 \cdot (R_1 + R_2)^*$$

20. Supponiamo che:  $\lambda \notin S(R_2)$

$$R_1 = R_2 \cdot R_1 + R_3 \text{ se e solo se } R_1 = R_2^* \cdot R_3$$

$$R_1 = R_1 \cdot R_2 + R_3 \text{ se e solo se } R_1 = R_3 \cdot R_2^*$$

# Dimostrazione proprietà espressioni regolari

## ■ Per esercizio

- Aiuto: le 1) - 5) e le 7) - 14) si dimostrano ricorrendo alla funzione  $S$ ;
- comunque la maggior parte delle 1) - 20) può essere provata con una tecnica generale, detta *dimostrazione mediante riparsificazione*.

# Dimostrazione mediante riparsificazione

- Illustriamo questa tecnica dimostrando la proprietà 17)

$$R_1 \cdot (R_2 \cdot R_1)^* = (R_1 \cdot R_2)^* \cdot R_1$$

Si consideri una qualunque parola:

$$w \in S(R_1 \cdot (R_2 \cdot R_1)^*), \quad w = r_1^0 (r_2^1 \cdot r_1^1) \cdot (r_2^2 \cdot r_1^2) \cdot \dots \cdot (r_2^n \cdot r_1^n), \quad n \geq 0$$

ove:

$$r_1^i \in S(R_1), \quad i = 0, 1, 2, \dots, n$$

$$r_2^j \in S(R_2), \quad j = 0, 1, 2, \dots, n$$

Riparsificando  $w$  ed utilizzando la proprietà associativa della concatenazione, si ha:  $w = (r_1^0 \cdot r_2^1) \cdot (r_1^1 \cdot r_2^2) \cdot \dots \cdot (r_1^{n-1} \cdot r_2^n) \cdot r_1^n$

dunque:  $w \in S((R_1 \cdot R_2)^* \cdot R_1)$

da cui:  $S(R_1 \cdot (R_2 \cdot R_1)^*) \subseteq S((R_1 \cdot R_2)^* \cdot R_1)$

In modo analogo si dimostra:  $S(R_1 \cdot (R_2 \cdot R_1)^*) \supseteq S((R_1 \cdot R_2)^* \cdot R_1)$

# Dimostrazione proprietà espressioni regolari

- Un'altra tecnica comune per dimostrare tali proprietà è semplicemente quella di utilizzare proprietà già note.
- Mostreremo ora come si usano le proprietà delle espressioni regolari per provare l'equivalenza di espressioni regolari.

# Esercizio

■ Dimostrare la seguente equivalenza:

$$(b + aa^*b) + (b + aa^*b) \cdot (a + ba^*b)^* \cdot (a + ba^*b) = a^*b \cdot (a + ba^*b)^*$$

$$\begin{aligned}(b + aa^*b) + (b + aa^*b) \cdot (a + ba^*b)^* \cdot (a + ba^*b) &\stackrel{9)}{=} (b + aa^*b) \cdot [\lambda + (a + ba^*b)^* (a + ba^*b)] = \\ &\stackrel{13)}{=} (b + aa^*b) \cdot (a + ba^*b)^* = \\ &\stackrel{10)}{=} (\lambda + aa^*) \cdot b \cdot (a + ba^*b)^* = \\ &\stackrel{13)}{=} a^*b \cdot (a + ba^*b)^*\end{aligned}$$

# Teorema di Kleene

- $\mathcal{L}_3 \equiv \mathcal{L}_{FSL} \equiv \mathcal{L}_{REG}$

## Dimostrazione

Lo schema della dimostrazione è il seguente:

- $\square \mathcal{L}_3 \subset \mathcal{L}_{FSL}$   $(\mathcal{L}_{FSL} \subset \mathcal{L}_3)$
- $\square \mathcal{L}_{FSL} \subset \mathcal{L}_{REG}$
- $\square \mathcal{L}_{REG} \subset \mathcal{L}_3$

# Dimostrazione teorema di Kleene

- $\mathcal{L}_3 \subset \mathcal{L}_{FSL} \quad def$   
Sia  $L \in \mathcal{L}_3 \Leftrightarrow \exists G = (X, V, S, P)$ ,  $G$  di tipo '3':  $L(G) = L$ .  
Vogliamo costruire un automa a stati finiti  
 $M = (Q, \delta, q_0, F)$  tale che  $T(M) = L(G)$ .

Allo scopo si fornisce il seguente algoritmo per la costruzione di un automa a stati finiti non deterministico che riconosce il linguaggio generato da una fissata grammatica lineare destra.

## Algoritmo: Costruzione di un automa a stati finiti non deterministico equivalente ad una grammatica lineare destra

- Data una grammatica lineare destra:

$$G = (X, V, S, P)$$

l'automa accettore a stati finiti equivalente ( $T(M) = L(G)$ ) viene costruito come segue:

$$M = (Q, \delta, q_0, F)$$

- (I)  $X$  come alfabeto di ingresso;
- (II)  $Q = V \cup \{q\}$ ,  $q \notin V$
- (III)  $q_0 = S$
- (IV)  $F = \{q\} \cup \{B \mid B \rightarrow \lambda \in P\}$
- (V)  $\delta : Q \times X \rightarrow 2^Q$      $\exists' \quad V.a \quad \forall B \rightarrow aC \in P, \quad C \in \delta(B, a)$   
 $V.b \quad \forall B \rightarrow a \in P, \quad q \in \delta(B, a)$

## Algoritmo: Costruzione di un automa a stati finiti non deterministico equivalente ad una grammatica lineare destra

- L'algoritmo può generare un automa *non deterministico* per effetto dei passi V.a. e V.b. Si può facilmente constatare che, se:  $w = x_1 x_2 \dots x_n \in L(G)$   
 $w$  può essere generata da una derivazione del tipo:

$$S \Rightarrow x_1 X_2 \Rightarrow x_1 x_2 X_3 \Rightarrow \dots \Rightarrow x_1 x_2 \dots x_{i-1} X_i \Rightarrow x_1 x_2 \dots x_n$$

Dalla definizione data, l'automa  $M$ , esaminando la stringa  $w = x_1 x_2 \dots x_n$  compie una serie di mosse (o transizioni) che lo portano dallo stato  $S$  ad  $X_2, X_3, \dots, X_i$  e  $q$ ; pertanto  $L(G) \subseteq T(M)$ .

In modo del tutto analogo, ogni  $w$  in  $T(M)$  comporta una sequenza di mosse dell'automa a cui corrisponde una derivazione in  $G$ , e pertanto  $T(M) \subseteq L(G)$ .

Se ne deduce che:  $L(G) = T(M)$

c.v.d.

## Algoritmo: Costruzione di un automa a stati finiti non deterministico equivalente ad una grammatica lineare destra

- Sebbene non sia strettamente necessario per la dimostrazione del Teorema di Kleene, per il suo interesse pratico si riporta di seguito l'algoritmo per la costruzione di una grammatica lineare destra che genera il linguaggio accettato da un automa a stati finiti.
- Tale algoritmo costituisce una dimostrazione costruttiva del seguente risultato:

$$\mathcal{L}_{FSL} \subset \mathcal{L}_3$$

## Algoritmo: Costruzione di una grammatica lineare destra equivalente ad un automa accettore a stati finiti

- Sia dato un automa accettore a stati finiti:

$$M = (Q, \delta, q_0, F)$$

con alfabeto di ingresso  $X$ .

La grammatica lineare destra  $G$  equivalente a  $M$ , ossia tale che  $L(G) = T(M)$ , si costruisce come segue:

- (I)  $X = \text{alfabeto di ingresso di } M$
- (II)  $V = Q$ ;
- (III)  $S = q_0$ ;
- (IV)  $P = \{q \rightarrow xq' \mid q' \in \delta(q, x)\} \cup \{q \rightarrow x \mid \delta(q, x) \in F\} \cup \{q_0 \rightarrow \lambda \mid q_0 \in F\}$

# Pumping Lemma per i linguaggi regolari

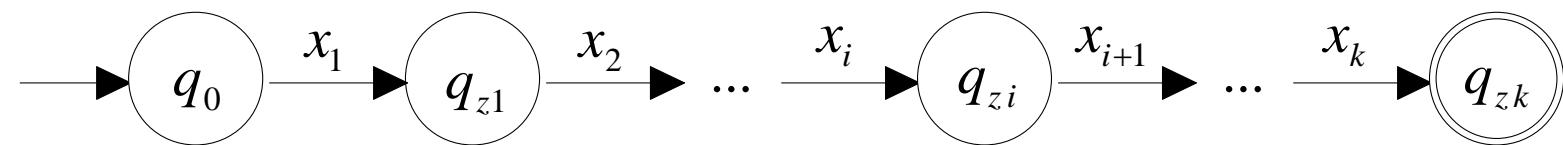
- Sia  $M = (Q, \delta, q_0, F)$  un automa accettore a stati finiti con  $n$  stati ( $|Q|=n$ ) e sia  $z \in T(M)$ ,  $|z| \geq n$ .  
Allora  $z$  può essere scritta come  $uvw$ , e  $uv^*w \subset T(M)$  (ossia  $\forall i, i \geq 0 : uv^i w \in T(M)$ ).
- Una formulazione alternativa è la seguente:  
Sia  $L = T(M)$  un linguaggio regolare con  
 $M = (Q, \delta, q_0, F)$  un automa accettore a stati finiti.  
Allora  $\exists n = |Q|$  t.c.  $\forall z \in L, |z| \geq n : z = uvw$  e:
  - $|uv| \leq n$
  - $v \neq \lambda$
  - $uv^i w \in L, \forall i, i \geq 0$

# Pumping Lemma per i linguaggi regolari

## ■ Dimostrazione

Sia  $z = x_1 x_2 \dots x_k, z \in T(M)$ .

Possiamo rappresentare il comportamento dell'automa  $M$ , con ingresso  $z$ , come segue:



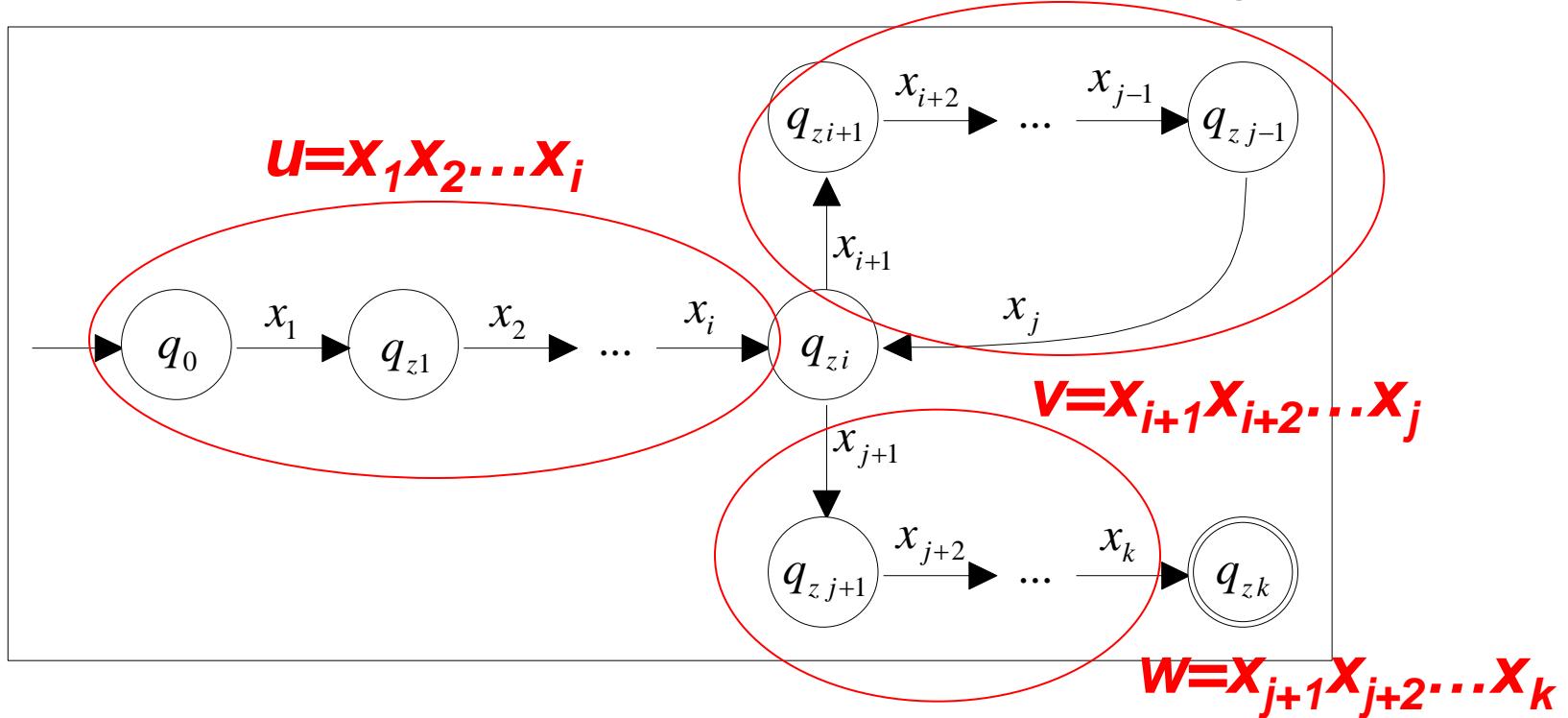
Se si ha:  $|z| \geq n$ , nella figura devono comparire almeno  $n+1$  stati ma, poiché  $M$  ha solo  $n$  stati distinti ( $|Q| = n$ ) almeno uno stato tra  $q_0, q_{z1}, q_{z2}, \dots, q_{zk}$  deve comparire due volte.

Supponiamo che si abbia  $q_{zi} = q_{zj}, i < j$ .

# Pumping Lemma per i linguaggi regolari

## ■ Dimostrazione

Si ha dunque la situazione rappresentata in figura:



Possiamo scrivere  $z$  nella forma:  $z = uvw$

# Pumping Lemma per i linguaggi regolari

## ■ Dimostrazione

Poiché  $z \in T(M)$ , l'automa  $M$ , per effetto dell'ingresso di  $z = uvw$ , si porta per definizione in uno stato finale ( $\delta^*(q_0, z) \in F$ ).

Ma è immediato osservare che tale stato è lo stesso in cui  $M$  si porta per effetto dell'ingresso delle parole  $uw$ ,  $uv^2w, \dots uv^i w$ ,  $i \geq 0$ .

Dunque si ha:  $uv^i w \in T(M)$ ,  $i \geq 0$ .

c.v.d.

# Esercizi

Esercizi