FONDAMENTI DI PROGRAMMAZIONE
Corso di Laurea in MATEMATICA
a.a. 2022/2023

Chiara Bodei, Roberta Gori, Damiano Di Francesco Maesa

Dipartimento di Informatica
chiara.bodei@unipi.it, roberta.gori@unipi.it, damiano.difrancesco@unipi.it

Obiettivi

Pensare come un informatico o capire come pensa :)

Pensare come un informatico è molto di più che programmare un computer!

> *Computational thinking will be a fundamental skill used by everyone in the world by the middle of the 21st Century*[1]

- Il "pensiero computazionale": processo mentale che ha a che fare con la risoluzione di problemi (problem solving). Vedi video tratto dall'Apollo 13: `https://www.programmailfuturo.it/progetto/cose-il-pensiero-computazionale`
- Ogni problema si affronta, usando opportunamente l'astrazione e la decomposizione, ricorrendo alle tecniche sviluppate dall'informatica.
- Questo modo di pensare influenza altre discipline: biologia, neuro-scienze, chimica, etc.

[1] J.M. Wing, "Computational Thinking," CACM Viewpoint, March 2006, pp. 33-35.

Pensiero Computazionale in cinque parole-chiave

http://link-and-think.blogspot.it/2016/04/cinque-parole-chiave-pensiero-computazionale.html

Astrazione

- Astrazione: in informatica indica il processo concettuale con il quale si passa da una descrizione di basso livello a una descrizione dello stesso concetto a un livello più alto, che subordina alcuni dettagli ed evidenzia caratteristiche importanti sulle quali si vuole richiamare l'attenzione.
- Astrazione come strumento concettuale del computational thinking

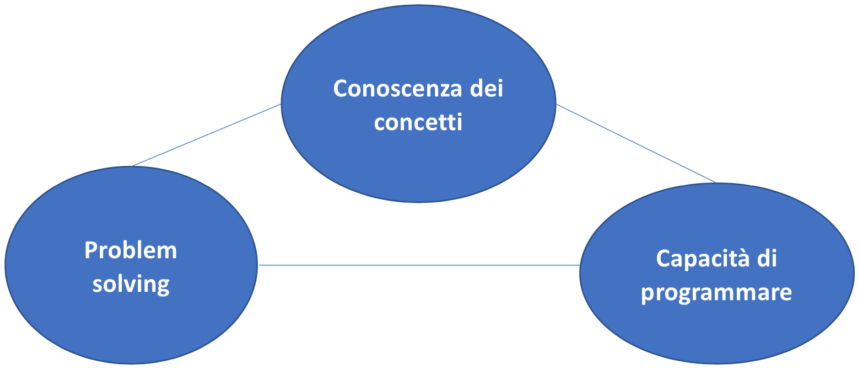
Pensiero Computazionale

I metodi caratteristici che si acquisiscono con lo studio dell'informatica includono:

- analizzare e organizzare i dati del problema in base a criteri logici;
- rappresentare i dati del problema tramite opportune astrazioni;
- formulare il problema in un formato che ci permette di usare un esecutore (in senso ampio) per risolverlo;
- automatizzare la risoluzione del problema definendo una soluzione algoritmica, consistente in una sequenza accuratamente descritta di passi, ognuno dei quali appartenente a un catalogo ben definito di operazioni di base;
- identificare, analizzare, implementare e verificare le possibili soluzioni con una efficace ed efficiente combinazione di passi e risorse (avendo come obiettivo la ricerca della soluzione migliore secondo tali criteri);
- generalizzare il processo di risoluzione del problema per poterlo trasferire ad un ampio spettro di altri problemi.

Obiettivi (cont)

- Capire come formulare i problemi in modo da poterli risolvere in modo computazionale
- Essere capaci di scrivere programmi di piccola/media dimensione
- Studiare i modelli astratti di calcolo per capire come funzionano i computer e come sono in grado di risolvere i problemi



Informazioni utili

- Docenti: Prof. Chiara Bodei, Roberta Gori, Damiano Di Francesco Maesa
- Orario Lezioni: LUN 11-13 , MER 9-11, GIO 9-11
- Orario Laboratorio: (MER 14-16): primo laboratorio: MER 5 ottobre
- Ricevimento studenti Bodei: per ora su appuntamento
- E-mail: chiara.bodei@unipi.it, roberta.gori@unipi.it, damiano.difrancesco@unipi.it
- Canale Teams del corso: 017AA 22/23 - FONDAMENTI DI PROGRAMMAZIONE CON LABORATORIO

  Pagina web del corso:
  www.di.unipi.it/∼chiara/CORSO_FP_22/FP/index.html

  Testi consigliati per la consultazione:
  - J. Hopcroft-R. Motwani-J. Ullman. Automi, linguaggi e calcolabilità. Addison-Wesley.
  - B.W. Kerninghan, D.M. Ritchie. Linguaggio C. Pearson.
  - Ceri-Mandrioli-Sbattella. Informatica: programmazione. McGraw-Hill

Programma di massima del corso

- Concetti di base della programmazione
- La programmazione nel linguaggio C
- Cenni di teoria degli automi e dei linguaggi

Informatica: cosa è e cosa non è

- **Non** è la scienza e la tecnica dei calcolatori.
- **Non** è lo studio degli utilizzi e delle applicazioni dei calcolatori e del software.
- **Non** è lo studio di come scrivere i programmi per i calcolatori.
- È la scienza che studia i procedimenti di calcolo effettivi

È lo studio sistematico degli algoritmi che descrivono e trasformano l'informazione: la loro teoria, analisi, progetto, efficienza, realizzazione e applicazione
Association for Computing Machinery.

Algoritmi

- Utilizziamo algoritmi nella vita quotidiana tutte le volte che, ad es., seguiamo le istruzioni per il montaggio di una apparecchiatura, per impostare il ciclo di lavaggio di una lavastoviglie, per prelevare contante da uno sportello Bancomat, ecc.

> Un algoritmo è una sequenza di passi che, se intrapresa da un esecutore, permette di ottenere i risultati attesi a partire dai dati forniti.

- Una volta in grado di specificare un algoritmo per risolvere un problema, siamo anche in grado di automatizzare il procedimento descritto dall'algoritmo.

Il termine algoritmo deriva dal nome del matematico persiano Muhammad ibn Musa al-Khwarizmi (Corasmia 780 circa - 850 circa). Esercitò la professione nella città di Baghdad, dove insegnava, e introdusse nel mondo arabo i numeri indiani. La sua opera "Il calcolo degli indiani" venne successivamente tradotta in latino da un monaco europeo, con il titolo Liber algarismi - (Il libro di al-Khwarizmi).

Algoritmi

- Il concetto di algoritmo ha origini molto lontane: l'uomo ha utilizzato spesso algoritmi per risolvere problemi di varia natura.
- È una tecnica trasmissibile che consente di passare dal risolvere i problemi al far risolvere i problemi. Le capacità intellettive necessarie per risolvere un problema sono "codificate" nell'algoritmo.
- Solo in era moderna ci si è posti il problema di caratterizzare problemi e classi di problemi per i quali è possibile individuare una soluzione algoritmica e solo nel secolo scorso è stato dimostrato che esistono problemi per i quali non è possibile individuare una soluzione algoritmica

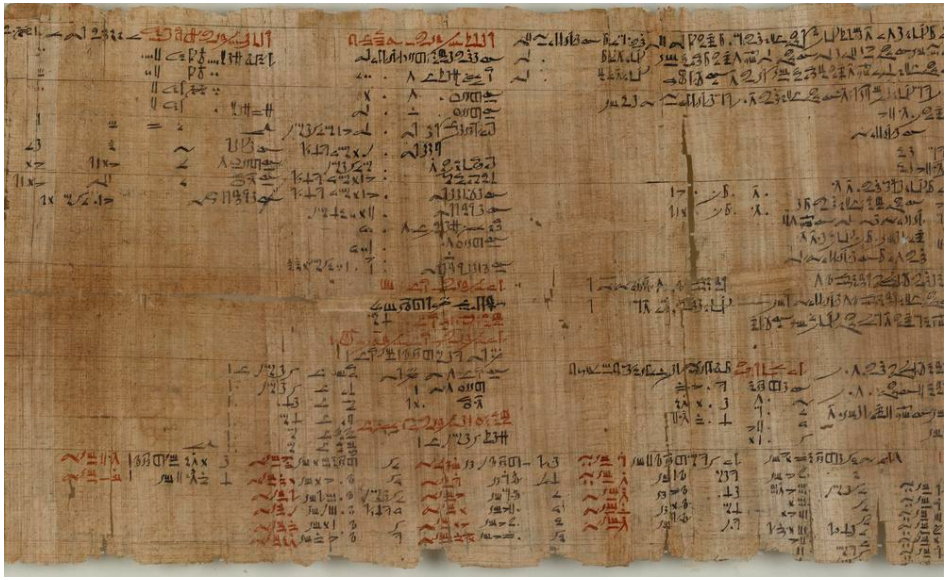
L'arte del problem solving

Un'idea geniale risolve spesso un grande problema, ma nella risoluzione di tutti i problemi interviene un pizzico di genialità.

Polya G., Come risolvere i problemi di matematica. Logica ed euristica nel metodo matematico. Feltrinelli, Milano, 1967

Il primo algoritmo documentato

Papiro di Ahmes o di Rhind
British Museum
ca 1650 a.C Scritto in ieratico



La moltiplicazione egizia

- Gli antichi non sapevano fare le operazioni in colonna (non c'era ancora la tecnologia giusta)
- Fare calcoli come le moltiplicazioni in generale era per loro complesso
- Fare calcoli più semplici come raddoppiamenti e dimezzamenti era invece più facile
- Pizzico di genialità: per calcolare A x B, raddoppio A e dimezzo B. Vado avanti semplificando così fino ad ottenere quanto mi serve.
- Uso la tecnica del *divide et impera*

Calcolare il MCD con l'algoritmo di Euclide

Per calcolare il Massimo Comun Divisore di x e y posso:

- Seguire la definizione: calcolare i divisori di x e di y, cercare quelli che sono divisori di entrambi e quindi identificare il massimo tra di loro; oppure
- sfruttare la proprietà che mi dice che:
  - MCD(x,x) = x
  - MCD(x,y) = MCD(x-y,y) se x >y
  - MCD(x,y) = MCD(x,y-x) se x >y

  e procedere con sottrazioni successive, riconducendo così un problema complesso a un problema più semplice e posso farlo più volte fino a trovare la soluzione

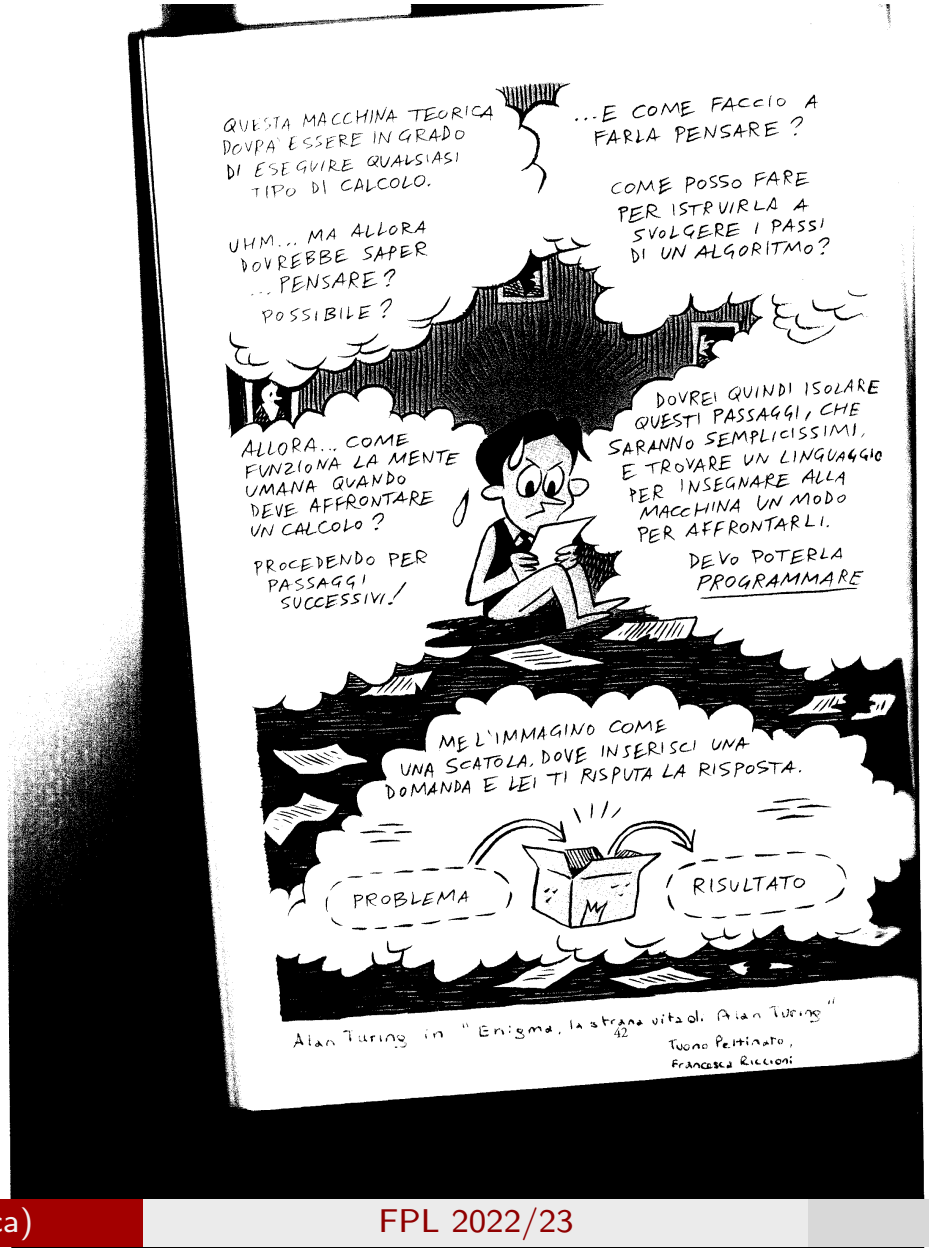
Astrazione di nuovo

- In informatica si separa il **cosa** (astrazione) dal **come** (implementazione), le proprietà funzionali da quelle strutturali
- Rispetto ai due esempi precedenti, possiamo separare il **cosa**, ovvero ottenere il risultato di una moltiplicazione in un caso e del Massimo Comun Divisore nell'altro dal **come**, ovvero farlo con un procedimento di calcolo oppure con un altro.

Domande fondamentali

- È sempre possibile trovare una soluzione algoritmica a un problema?
- Esiste un esecutore automatico in grado di eseguire un algoritmo e se sì come è fatto?

Il computer: una macchina per calcolare



Come nasce l'informatica? Facciamo un passo indietro

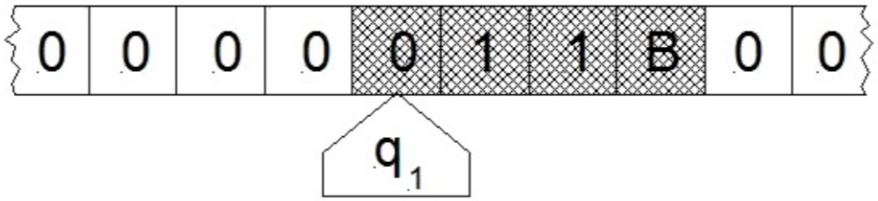
- È la scienza che studia i procedimenti di calcolo effettivi e nasce prima dei computer, studiando i processi di calcolo umano
- Turing vede il calcolo come manipolazione combinatoria di insiemi finiti e discreti di simboli secondo un insieme finito di regole

Domande fondamentali più dettagliate

- **Universalità: esiste una nozione astratta di macchina per calcolare?**
- Calcolabilità: esistono problemi che non possono essere risolti da nessuna macchina per il calcolo?
- Esistono dei limiti alla potenza espressiva delle macchine per calcolare?
- Esistono procedimenti più efficienti di altri?

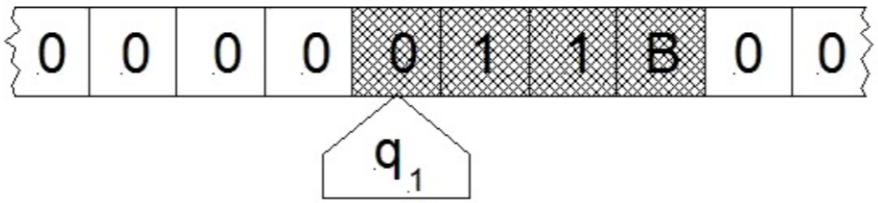
La Macchina di Turing

- È una macchina ideale (non fisica) di calcolo che manipola i dati contenuti su un nastro di lunghezza potenzialmente infinita, secondo un insieme di regole ben definite
- È il modello astratto di una macchina in grado di eseguire algoritmi



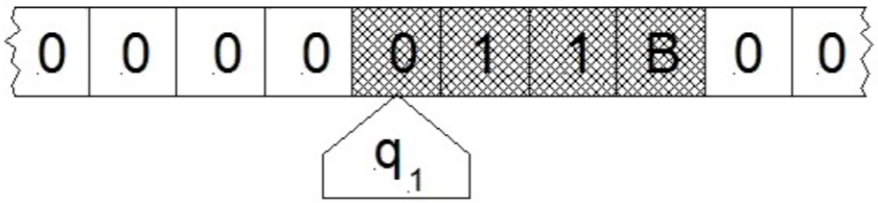
La Macchina di Turing (cont.)

- Se abbiamo una descrizione del funzionamento della macchina possiamo simularne il comportamento con carta e matita
- Questo processo è esso stesso un procedimento di calcolo
- Tra tutte le macchine di Turing ce ne è una che è in grado di simulare tutte le altre: è la Macchina Universale
- Con una descrizione e i dati la Macchina Universale eseguirà il calcolo per noi: è questo un modello teorico di computer



La Macchina di Turing (cont.)

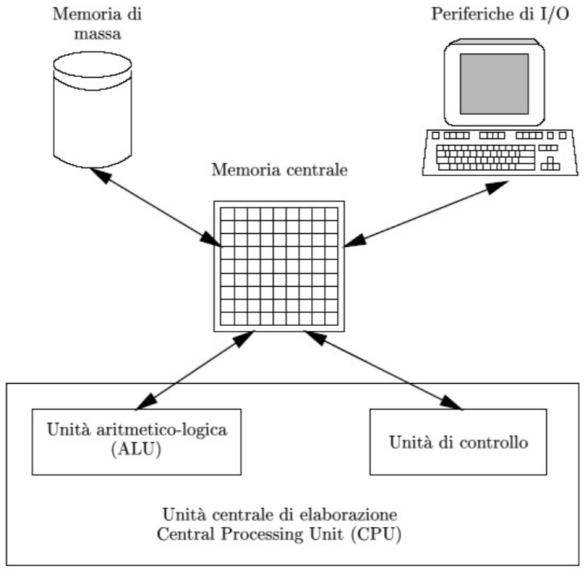
- Un problema è calcolabile se è calcolabile secondo Turing, ovvero se esiste una macchina di Turing in grado di risolverlo
- Le macchine di Turing possono eseguire tutte le computazioni che eseguono i moderni computer
- Modello universale di calcolo!



Il modello di Von Neumann

Modello concreto

- Equivalente alla Macchina di Turing
- Architettura della macchina
- Design alla base dei calcolatori attuali
- Esegue i calcoli predefiniti e quelli definiti dal programmatore
- La generalità è data dal programma memorizzato



Domande fondamentali più dettagliate

- Universalità: esiste una nozione astratta di macchina per calcolare?
- **Calcolabilità: esistono problemi che non possono essere risolti da nessuna macchina per il calcolo?**
- **Esistono dei limiti alla potenza espressiva delle macchine per calcolare?**
- Esistono procedimenti più efficienti di altri?

Calcolabilità

- Esistono problemi che non possono essere risolti in modo algoritmico
- Non è vero che possono risolvere tutti i nostri problemi

Domande fondamentali più dettagliate

- Universalità: esiste una nozione astratta di macchina per calcolare?
- Calcolabilità: esistono problemi che non possono essere risolti da nessuna macchina per il calcolo?
- Esistono dei limiti alla potenza espressiva delle macchine per calcolare?
- **Esistono procedimenti più efficienti di altri?**

Complessità

- Per uno stesso problema possono esistere soluzioni/algoritmi diversi
- È importante saperli confrontare e scegliere quelli più efficienti
- Ci sono problemi "facili": sono quelli che richiedono un tempo ragionevole
- Ci sono problemi "difficili" che richiedono invece tempi così lunghi da essere praticamente irrisolvibili

Come si parla alla macchina?

- Il linguaggio (di programmazione) è lo strumento che permette di descrivere gli algoritmi in modo che il calcolatore possa comprenderli ed eseguirli
- Esiste una Babele di linguaggi di programmazione
- Il concetto di "cosa è calcolabile" non cambia: hanno la stessa potenza espressiva

Quando l'esecutore è il calcolatore

- I calcolatori sono macchine in grado di eseguire velocemente e con precisione sequenze di operazioni elementari.
- Un programma è la descrizione di un algoritmo, espressa in un linguaggio di programmazione che il calcolatore è in grado di comprendere ed eseguire.
- Il calcolatore riceve in ingresso un programma e un insieme di dati iniziali e produce in uscita i risultati dell'esecuzione del programma.
- A differenza di altre macchine automatiche (lavatrici, calcolatrici tascabili, ecc.) i calcolatori sono programmabili: la funzione svolta dipende dal particolare programma che indica alla macchina quali azioni compiere. La macchina non cambia al variare della funzione.

Linguaggi di programmazione

- Esistono moltissimi linguaggi di programmazione che possono essere classificati a seconda del paradigma di programmazione che seguono
- Un paradigma di programmazione indica lo stile e l'insieme di strumenti concettuali da seguire per scrivere programmi, riflettendo un un modo di pensare il processo di computazione.
- Adottando il C seguiremo il paradigma imperativo, per il quale un programma viene come una sequenza di comandi che agiscono sui dati o sull'ordine di esecuzione delle istruzioni [Altri esempi sono: Assembly, FORTRAN, COBOL, Pascal]
- Esistono altri paradigmi di programmazioni, tra i quali: funzionale [ML, Haskell, Scheme], orientato a oggetti [Java, Smalltalk, Eiffel], logico [PROLOG] e concorrente [CCS, CSP].

Cosa intendiamo per programmazione

- Il procedimento che porta alla definizione dei programmi adatti a risolvere problemi è detto programmazione.
- I concetti che stanno alla base della programmazione si possono spiegare e comprendere senza far riferimento al calcolatore.
- La programmazione è tuttavia divenuta una vera e propria disciplina solo con l'avvento dei moderni calcolatori elettronici.

Le fasi della programmazione

Ad un primo livello di astrazione l'attività della programmazione può essere suddivisa in quattro (macro) fasi principali.

1. Definizione del problema (specifica)
2. Individuazione di un procedimento risolutivo (algoritmo)
3. Codifica dell'algoritmo in un linguaggio di programmazione (codifica)
4. Esecuzione e messa a punto (esecuzione)

Specifica

- La prima fase della programmazione consiste nel comprendere e definire (specificare) il problema che si vuole risolvere.
- La specifica del problema può essere fatta in maniera più o meno rigorosa, a seconda del formalismo descrittivo utilizzato.
- La specifica di un problema prevede la descrizione dello stato iniziale del problema (dati iniziali, input) e dello stato finale atteso (i risultati, output).
- La caratterizzazione degli stati iniziale e finale dipende dal particolare problema in esame e dagli oggetti di interesse.

Esempi di specifica informale

1. Dati due numeri, trovare il maggiore.
2. Dato un elenco telefonico e un nome, trovare il numero di telefono corrispondente.
3. Data la struttura di una rete stradale e le informazioni sui flussi dei veicoli, determinare il percorso più veloce da A a B.
4. Scrivere tutti i numeri pari (a parte 2) che non sono la somma di due numeri primi (Congettura di Goldbach [1742]).
5. Decidere per **ogni** programma e per ogni dato in ingresso, se il programma C termina quando viene eseguito su quel dato.

Esempi (contd.)

Caratteristiche comuni ai problemi

informazioni in ingresso $\Longrightarrow$ informazioni in uscita

stato iniziale $\Longrightarrow$ stato finale

Osservazioni sulla formulazione dei problemi:

- la descrizione non fornisce un metodo risolutivo (es. 3)
- la descrizione del problema è talvolta ambigua o imprecisa (es. 2, con Mario Rossi che compare più volte)
- per alcuni problemi non è noto un metodo risolutivo (es. 4)
- esistono problemi per i quali è stato dimostrato che non può esistere un metodo risolutivo (es. 5 - *indecidibili*)

Noi considereremo solo problemi per i quali
è noto che esiste un metodo risolutivo *decidibili*.

Algoritmi

- Una volta specificato il problema, si determina un procedimento risolutivo dello stesso (algoritmo), ovvero un insieme di azioni da intraprendere per ottenere i risultati attesi.
- Il concetto di algoritmo ha origini molto lontane: l'uomo ha utilizzato spesso algoritmi per risolvere problemi di varia natura. Le capacità intellettive necessarie per risolvere un problema sono "codificate" nell'algoritmo.
- Solo in era moderna, ci si è posti il problema di caratterizzare problemi e classi di problemi per i quali è possibile individuare una soluzione algoritmica e solo nel secolo scorso è stato dimostrato che esistono problemi per i quali non è possibile individuare una soluzione algoritmica.

Proprietà di un algoritmo

La descrizione di un procedimento risolutivo può considerarsi un algoritmo se rispetta alcuni requisiti essenziali, tra i quali:

Finitezza: un algoritmo deve essere composto da una sequenza finita di passi elementari.

Non-ambiguità: l'esecutore deve poter interpretare in modo univoco ogni singola azione.

Eseguibilità: il potenziale esecutore deve essere in grado di eseguire ogni singola azione in tempo finito con le risorse a disposizione.

Tipici procedimenti che non rispettano alcuni dei requisiti precedenti:

- le ricette di cucina: *aggiungere sale q.b.* - non rispetta 3)
- le istruzioni per la compilazione della dichiarazione dei redditi (!)

Proprietà di un algoritmo (cont.)

Sarà poi opportuno anche valutare l'efficienza di un algoritmo.

Se dato un elenco telefonico e un nome, per trovare il numero di telefono corrispondente guardo e confronto un nome dopo l'altro, ci metto molto più tempo che procedendo nel modo seguente:

- apro a metà e leggo il nome in cima;
- se trovo il nome che cerco ho finito, altrimenti
- se il nome cercato viene prima in ordine lessicografico, allora cerco nella prima metà;
- altrimenti cerco nella seconda metà;
- andando avanti nello stesso modo fino a trovare il nome o a decidere che non compare nell'elenco.

Quanto tempo ci metterò nei due casi?

Codifica

- Questa fase consiste nell'individuare una rappresentazione degli oggetti di interesse del problema ed una descrizione dell'algoritmo in un opportuno linguaggio noto all'esecutore.
- Nel caso in cui si intenda far uso di un elaboratore per l'esecuzione dell'algoritmo, quest'ultimo deve essere tradotto (codificato) in un opportuno linguaggio di programmazione. Il risultato in questo caso è un programma eseguibile per il calcolatore.
- Quanto più il linguaggio di descrizione dell'algoritmo è vicino al linguaggio di programmazione scelto, tanto più semplice è la fase di traduzione e codifica. Se addirittura il linguaggio di descrizione coincide con il linguaggio di programmazione, la fase di traduzione è superflua.

Codifica (cont.)

I linguaggi di programmazione forniscono strumenti linguistici per rappresentare gli algoritmi sotto forma di programmi che possano essere compresi da un calcolatore.

In particolare dobbiamo rappresentare nel linguaggio di programmazione

- l'algoritmo $\Longrightarrow$ programma
- le informazioni iniziali $\Longrightarrow$ dati in ingresso
- le informazioni utilizzate dall'algoritmo $\Longrightarrow$ dati ausiliari
- le informazioni finali $\Longrightarrow$ dati in uscita

In questo corso impareremo a codificare algoritmi utilizzando il linguaggio di programmazione denominato C.

Esecuzione

- La fase conclusiva consiste nell'esecuzione vera e propria del programma.
- Spesso questa fase porta alla luce errori che possono coinvolgere ciascuna delle fasi precedenti, innescando un procedimento di messa a punto tale da richiedere la revisione di una o più fasi (dalla specifica, alla definizione dell'algoritmo, alla codifica di quest'ultimo).
- Nel caso dei linguaggi di programmazione moderni, vengono forniti strumenti (denominati di solito debugger) che aiutano nella individuazione degli errori e nella messa a punto dei programmi.

Esempi

Negli esempi che seguono, utilizziamo un linguaggio pseudo-naturale per la descrizione degli algoritmi.

Tale linguaggio utilizza, tra l'altro, le comuni rappresentazioni simboliche dei numeri e delle operazioni aritmetiche.

Quanti hanno lo zaino in questa aula?

Specifica

Input: sequenza di postazioni e zaini in aula

Output: il numero di zaini

Algoritmo

Un semplice algoritmo è il seguente:

Passo 1.	Associa zero al *contatore* N
Passo 2.	Per ogni zaino che vedi **ripeti**
Passo 2.1.	Incrementa N di 1
Passo 3.	Ottieni il risultato dell'operazione in N

È corretto?

Quanti hanno lo zaino in questa aula? (cont.)

Specifica

Input: sequenza di postazioni e zaini in aula

Output: il numero di zaini

Algoritmo

Un algoritmo più veloce è il seguente:

Passo 1.	Associa zero al *contatore* N
Passo 2.	Per ogni coppia di zaini che vedi **ripeti**
Passo 2.1.	Incrementa N di 2
Passo 3.	Ottieni il risultato dell'operazione in N

È corretto?

Quanti hanno lo zaino in questa aula? (cont.)

Specifica

Input: sequenza di postazioni e zaini in aula

Output: il numero di zaini

Algoritmo

Un algoritmo più veloce ma corretto è il seguente:

Passo 1.	Associa zero al *contatore* N
Passo 2.	Per ogni coppia di zaini che vedi **ripeti**
Passo 2.1.	Incrementa N di 2
Passo 3.	Se rimane uno zaino
Passo 3.1.	Incrementa N di 1
Passo 4.	Ottieni il risultato dell'operazione in N

È corretto?

Problema 1: Calcolo del prodotto di due interi positivi

Specifica

Input: due valori interi positivi A e B

Output: il valore di A×B

Algoritmo

Se l'esecutore che scegliamo è in grado di effettuare tutte le operazioni di base sui numeri, un semplice algoritmo è il seguente:

Passo 1.	Acquisisci il primo valore, sia esso A
Passo 2.	Acquisisci il secondo valore, sia esso B
Passo 3.	Ottieni il risultato dell'operazione A×B

Problema 1 (cont.)

Codifica

Un esecutore che rispetta le ipotesi precedenti è una persona dotata di una calcolatrice tascabile. La codifica dell'algoritmo in questo caso può essere allora la seguente:

Passo 1.	Digita in sequenza le cifre decimali del primo valore
Passo 2.	Digita il tasto $*$
Passo 3.	Digita in sequenza le cifre decimali del secondo valore
Passo 4.	Digita il tasto $=$

Esecuzione

Problema 1 (cont.)

- Supponiamo ora che l'esecutore scelto sia in grado di effettuare solo le operazioni elementari di somma, sottrazione, confronto tra numeri.
- È necessario individuare un nuovo procedimento risolutivo che tenga conto delle limitate capacità dell'esecutore

Algoritmo 2

Passo 1.	Acquisisci il primo valore, sia esso A
Passo 2.	Acquisisci il secondo valore, sia esso B
Passo 3.	Associa 0 ad un terzo valore, sia esso C
Passo 4.	**Finché** B>0 **ripeti**
Passo 4.1.	Somma a C il valore A
Passo 4.2.	Sottrai a B il valore 1
Passo 5.	Il risultato è il valore C

Problema 1: (cont.)

Un esecutore che rispetta le ipotesi precedenti è un bimbo in grado di effettuare le operazioni richieste (somma, sottrazione e confronto) e di riportare i risultati di semplici calcoli su un quaderno.

Codifica

Passo 1.	Scrivi il primo numero nel riquadro A
Passo 2.	Scrivi il secondo numero nel riquadro B
Passo 3.	Scrivi il valore 0 nel riquadro C
Passo 4.	Ripeti i seguenti passi finché il valore nel riquadro B è maggiore di 0: - calcola la somma tra il valore in A e il valore in C - scrivi il risultato ottenuto in C - calcola la differenza tra il valore in B ed il numero 1 - scrivi il risultato ottenuto in B
Passo 5.	Il risultato è quanto contenuto nel riquadro C.

Cosa si intende con stato

- una particolare configurazione delle informazioni di una macchina, che in qualche modo "memorizza" le condizioni in cui si trova, e che cambia nel tempo passando ad un'altra configurazione, in funzione dei segnali d'ingresso.
- Ad esempio, lo stato può rappresentare la posizione dell'ascensore ad un certo piano di un edificio. In base allo stato si determina il modo in cui l'ascensore si muove: se si intende andare al terzo piano e ci troviamo al primo piano, occorre che l'ascensore salga.
- un sistema è *stateful* se "ricorda' gli eventi precedenti; le informazioni ricordate sono chiamate lo *stato* del sistema.

Il concetto di stato

- La specifica (astratta) di un problema consiste nella descrizione di uno stato iniziale (che descrive i dati del problema) e di uno stato finale (che descrive i risultati attesi).
- Si deve individuare una rappresentazione degli oggetti coinvolti (e dunque dello stato) direttamente manipolabile dall'esecutore.
- Un algoritmo è una sequenza di passi elementari che, se eseguiti, comportano ripetute modifiche dello stato fino al raggiungimento dello stato finale desiderato.
- Le azioni di base che l'esecutore è in grado di effettuare devono dunque prevedere, tra le altre, azioni che hanno come effetto cambiamenti dello stato.
- Possiamo pensare che lo stato sia il contenuto della memoria e che il cambiamento dello stato abbia come effetto il cambiamento di alcune posizioni della memoria.

Un'astrazione dello stato

Dato che il calcolo procede attraverso l'elaborazione dello stato, si devono poter esprimere valori dipendenti dallo stato.

Astrazione del concetto di Stato

Uno stato è un insieme di associazioni tra nomi simbolici e valori.

In uno stato, ad ogni nome simbolico è associato al più un valore.

Rappresentiamo una associazione tra il nome simbolico x ed il valore val con la notazione

$$x \rightsquigarrow val$$

Il valore di x dipende dallo stato corrente, che gli associa un particolare valore val.

Lo stato: esempi

Stati corretti

- $\{\text{nome} \rightsquigarrow \text{Antonio}, \text{cognome} \rightsquigarrow \text{Rossi}, \text{età} \rightsquigarrow 25\}$
- $\{\text{importo} \rightsquigarrow \$1650, \text{tasso} \rightsquigarrow 10\%, \text{interesse} \rightsquigarrow \$165\ \}$
- $\{a \rightsquigarrow 25, b \rightsquigarrow 3, c \rightsquigarrow 50\}$

Stati non corretti

- $\{\text{nome} \rightsquigarrow \text{Antonio}, \text{nome} \rightsquigarrow \text{Paolo}, \text{età} \rightsquigarrow 25\}$
- $\{b \rightsquigarrow 45, a \rightsquigarrow 150, b \rightsquigarrow 10\}$

Prima introduzione al linguaggio C

- Abbiamo visto come un programma non sia altro che un algoritmo codificato in un linguaggio di programmazione.
- Problema: quale linguaggio scegliere per la codifica di un algoritmo?
  - Il linguaggio naturale sarebbe facilmente comprensibile ma non è eseguibile da una macchina.
  - Il linguaggio macchina è eseguibile ma di difficile comprensione.
- Due requisiti fondamentali di un qualsiasi linguaggio per la descrizione di algoritmi:
  - deve essere preciso per non lasciare adito a dubbi interpretativi
  - deve essere sintetico per non rendere difficile la comprensione dei programmi.

- Il linguaggio naturale e il linguaggio macchina si collocano in posizioni opposte, soddisfacendo uno solo dei requisiti.
- I linguaggi di programmazione ad alto livello sono progettati proprio per colmare tale divario.
  $\Longrightarrow$ sono linguaggi adatti a codificare algoritmi pur rimanendo comprensibili.
- La fatica di tradurre un programma nel linguaggio macchina è affidata a particolari programmi, i compilatori, che traducono programmi scritti nel linguaggio di più alto livello in programmi equivalenti nel linguaggio macchina.

Il linguaggio C

- Introduciamo inizialmente l'insieme di costrutti linguistici che costituiscono il nucleo di un qualunque linguaggio di programmazione reale, usando già la sintassi del C.
- Senza entrare in eccessivi dettagli formali, nel presentare le notazioni utilizzate (sintassi) diamo anche una descrizione informale (semantica) di ciò che accade al momento dell'esecuzione in corrispondenza dei vari costrutti.

Il linguaggio contiene costrutti per:

- rappresentare semplici calcoli attraverso le comuni operazioni logico/aritmetiche (espressioni)
- modificare le associazioni nello stato (assegnamento e ingresso)
- controllare l'ordine di esecuzione delle azioni (controllo)
- fornire i risultati (produzione in uscita dello stato finale)

Espressioni

Il ruolo delle espressioni è quello di denotare valori (dip. dallo stato).

Espressioni Numeriche

Il linguaggio consente di rappresentare semplici calcoli algebrici, attraverso le usuali espressioni costruite a partire dai valori numerici e dalle operazioni di

- somma $+$
- sottrazione -
- prodotto $*$
- divisione intera $/$
- modulo o resto della divisione intera $\%$.

Il significato di un'espressione è il suo valore ottenuto secondo le usuali regole di calcolo.

Esempio:

- il valore di $3 * 5 + 6$ è 21
- il valore di $3 * (5 + 6)$ è 33

Espressioni (cont.)

Oltre ai valori numerici, le espressioni possono contenere nomi simbolici: il calcolo di un'espressione che contiene un nome simbolico x dipende dallo stato, e precisamente dal valore associato al nome x nello stato.

Esempio:
il valore di $3 * (5 + x)$ è

- 33 in uno stato che contiene l'associazione $x \rightsquigarrow 6$
- 18 in uno stato che contiene l'associazione $x \rightsquigarrow 1$

Gli operatori possiedono regole di precedenze che determinano come avviene la valutazione delle espressioni. Come nell'aritmetica tradizionale, + e - hanno lo stesso grado di priorità, inferiore a quello di *,/ e %.

Espressioni (cont.)

Espressioni Booleane

Si possono rappresentare condizioni ovvero espressioni il cui valore è un valore di verità (true o false).

Le condizioni sono costruite attraverso le usuali operazioni di confronto (==, !=, <, >, <=, >=) e, come nel caso delle espressioni aritmetiche, il loro valore può dipendere dallo stato.

Esempio: il valore di y==x+1 è

- true in uno stato che contiene le associazioni x $\rightsquigarrow$ 5 e y $\rightsquigarrow$ 6
- false in uno stato che contiene le associazioni x $\rightsquigarrow$ 5 e y $\rightsquigarrow$ 9

Condizioni più complesse possono essere costruite attraverso operatori logici quali negazione (simbolo !), congiunzione (simbolo &&) e disgiunzione (simbolo ||).

Espressioni (cont.)

- Significato di ! - il valore di verità di ! P è
  - true se il valore di verità di P è false
  - false se il valore di verità di P è true
- Significato di && - il valore di verità di P && Q è
  - true se i valori di verità di P e Q sono entrambi true
  - false altrimenti
  - Se P è false si restituisce false senza valutare Q
- Significato di || - il valore di verità di P || Q è
  - false se i valori di verità di P e Q sono entrambi false
  - true altrimenti
  - Se P è true si restituisce true senza valutare Q
- || and && hanno lo stesso grado di priorità, inferiore a quello di !

Esempio:

- Il valore di `(y >= x) &&(x > 5)` è
  - true in uno stato che contiene le associazioni x $\rightsquigarrow$ 15 e y $\rightsquigarrow$ 30
  - false in uno stato che contiene le associazioni x $\rightsquigarrow$ 3 e y $\rightsquigarrow$ 30
- Il valore di `(y >= x) || (x > 5)` è
  - true in uno stato che contiene le associazioni x $\rightsquigarrow$ 2 e y $\rightsquigarrow$ 30
  - false in uno stato che contiene le associazioni x $\rightsquigarrow$ 3 e y $\rightsquigarrow$ 1

Tavole di verità dei principali operatori logici

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \iff q$	$p \Leftarrow q$
F	F	T	F	F	T	T	T
F	T	T	F	T	T	F	F
T	F	F	F	T	F	F	T
T	T	F	T	T	T	T	T

Lo stato e le variabili

Lo stato rappresenta il contenuto (modificabile) della memoria.

Una posizione della memoria, destinata a contenere dati, si identifica con una variabile Le variabili dunque rappresentano, nei programmi, le associazioni dello stato
$\Longrightarrow$ cf. $x \rightsquigarrow val$ introdotto prima

Il valore di x dipende dallo stato corrente, che gli associa un particolare valore val.

Le associazioni possono essere modificate

Programmazione imperativa e modifica dello stato

- Nella programmazione imperativa, la computazione viene descritta in termini di stato di programma.
- Il ruolo delle istruzioni (I_i) è quello di modificare, con la loro esecuzione, lo stato.

$Stato_{in}$	$I_1 \ldots I_n$	$Stato_1$	$I_1 I_2 \ldots I_n$	$Stato_2$	...	$Stato_{n-1}$	$I_1 I_2 \ldots I_n$	$Stato_{fin}$

Modifica dello stato: ASSEGNAMENTO

- Una delle istruzioni che consentono di rappresentare modifiche di stato è l'assegnamento.
- L'assegnamento consente di cambiare un'associazione nello stato, ovvero il valore associato nello stato ad un nome simbolico.
- Useremo per l'assegnamento la seguente notazione

  $$\boxed{\texttt{x = exp;}}$$

  dove x è un nome simbolico (la variabile in C) e exp una espressione.
- L'esecuzione dell'assegnamento x = exp; consiste nel
  (i) calcolare il valore, sia esso val, dell'espressione exp
  (ii) introdurre nello stato l'associazione $x \rightsquigarrow val$
- Si noti che (ii) comporta la rimozione dallo stato della eventuale associazione già presente per x (si parla a questo proposito di assegnamento distruttivo).

ASSEGNAMENTO: esempi

Vediamo alcuni esempi, indicando lo stato prima e dopo l'esecuzione degli assegnamenti proposti. Le associazioni modificate nello stato finale sono evidenziate in verde.

Stato iniziale	Assegnamento	Stato Finale
$\{ x \rightsquigarrow 10, y \rightsquigarrow 20 \}$	x = 5;	$\{ x \rightsquigarrow 5, y \rightsquigarrow 20 \}$
$\{ x \rightsquigarrow 10, y \rightsquigarrow 20 \}$	x = y*2;	$\{ x \rightsquigarrow 40, y \rightsquigarrow 20 \}$
$\{ x \rightsquigarrow 10, y \rightsquigarrow 20 \}$	x = x+1;	$\{ x \rightsquigarrow 11, y \rightsquigarrow 20 \}$

Si noti come, nel terzo esempio, lo stesso nome simbolico x giochi un duplice ruolo:

- a destra del simbolo = indica un valore (il valore associato ad x nello stato iniziale)
- a sinistra del simbolo = indica l'associazione da modificare nello stato a seguito dell'assegnamento.

Modifica dello stato: INPUT

- La seconda istruzione di modifica dello stato consente di acquisire valori dal mondo esterno al momento dell'esecuzione. La notazione è

  `scanf(...,&x);`

  dove x indica un nome simbolico, di cui &x rappresenta l'indirizzo in memoria, e in ... troveremo la stringa usata per il controllo del formato (aspetti su cui torneremo in seguito).
- L'esecuzione consiste nel:
  (i) Acquisire un nuovo valore, sia esso val (quando viene eseguita il programma si mette in attesa che l'utente immetta un valore.)
  (ii) Introdurre nello stato l'associazione $x \rightsquigarrow val$
- Come nel caso dell'assegnamento, il punto (ii) comporta la rimozione dallo stato dell'eventuale associazione già presente per il nome simbolico (variabile in C) x
- La presenza di tale istruzione permette di descrivere algoritmi generali in cui non tutti i dati sono noti a priori, ma lo saranno solo al momento dell'esecuzione.

Istruzioni di controllo: SEQUENZA

- Negli esempi visti in precedenza gli algoritmi sono stati descritti come sequenze di passi elementari del tipo

  Passo 1. azione 1
  Passo 2. azione 2
  ...

- Abbiamo utilizzato una sorta di numerazione per indicare l'ordine di esecuzione delle varie azioni: prima azione 1 poi azione 2 poi ....
- Nel linguaggio che stiamo introducendo, e in C, una sequenza di azioni viene rappresentata mediante un blocco

```
{
istruzione 1
istruzione 2
...
istruzione n
}
```

SEQUENZA (cont.)

- Scriveremo dunque

  {
  istruzione 1
  istruzione 2
  ...
  }

  ad indicare che l'ordine di esecuzione dei singoli passi è quello testuale del programma.
- Si noti che ogni istruzione viene eseguita a partire dallo stato risultante dall'esecuzione dell'istruzione che la precede nella sequenza.
- Assegnamento e ingresso sono istruzioni semplici il blocco è un'istruzione composta.

SEQUENZA: esempi

Stato iniziale	Sequenza	Stato Finale
$\{ x \rightsquigarrow 10, y \rightsquigarrow 20\}$	$\{ x = 5; x = x+y; \}$	$\{ x \rightsquigarrow 25, y \rightsquigarrow 20\}$
$\{ x \rightsquigarrow 10, y \rightsquigarrow 20\}$	$\{ x = x+1; y = x+1; \}$	$\{ x \rightsquigarrow 11, y \rightsquigarrow 12\}$
$\{ x \rightsquigarrow 10, y \rightsquigarrow 20\}$	$\{ x = x+y; y = x+y; \}$	$\{ x \rightsquigarrow 30, y \rightsquigarrow 50\}$

- In tutti gli esempi, lo stato finale si ottiene dall'esecuzione del secondo assegnamento nello stato intermedio risultante dall'esecuzione del primo assegnamento.
- Ad esempio, nel terzo caso:

Stato iniziale	Prima istruzione	Stato Intermedio
$\{ x \rightsquigarrow 10, y \rightsquigarrow 20\}$	$x = x+y;$	$\{ x \rightsquigarrow 30, y \rightsquigarrow 20\}$
Stato intermedio	**Seconda istruzione**	**Stato Finale**
$\{ x \rightsquigarrow 30, y \rightsquigarrow 20\}$	$y = x+y;$	$\{ x \rightsquigarrow 30, y \rightsquigarrow 50\}$

Istruzioni di controllo: CONDIZIONALE

- Permette di determinare l'azione da intraprendere a seconda del verificarsi o meno di una condizione. La notazione utilizzata è la seguente

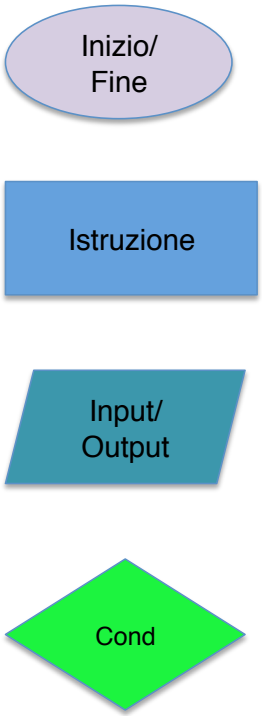
  if (condizione) istruzione1 **else** istruzione2

  dove condizione indica una espressione booleana, e istruzione1 istruzione2 sono istruzioni (semplici o composte).
- L'esecuzione del condizionale **if** (C) S1 **else** S2 consiste nel
  (i) Calcolare il valore, sia esso val, dell'espressione booleana C
  (ii) Eseguire S1 se val è true, eseguire S2 se val è false.
- Si noti che la presenza dell'istruzione condizionale non è in conflitto con il requisito di non ambiguità degli algoritmi: l'azione da intraprendere è univocamente determinata dal valore di verità della condizione (nello stato dato) e dunque l'esecutore non deve scegliere.
- Possiamo anche avere **if** annidati (in cascata) quando l'istruzione del ramo **then** o **else** è un'istruzione **if** o **if-else**.

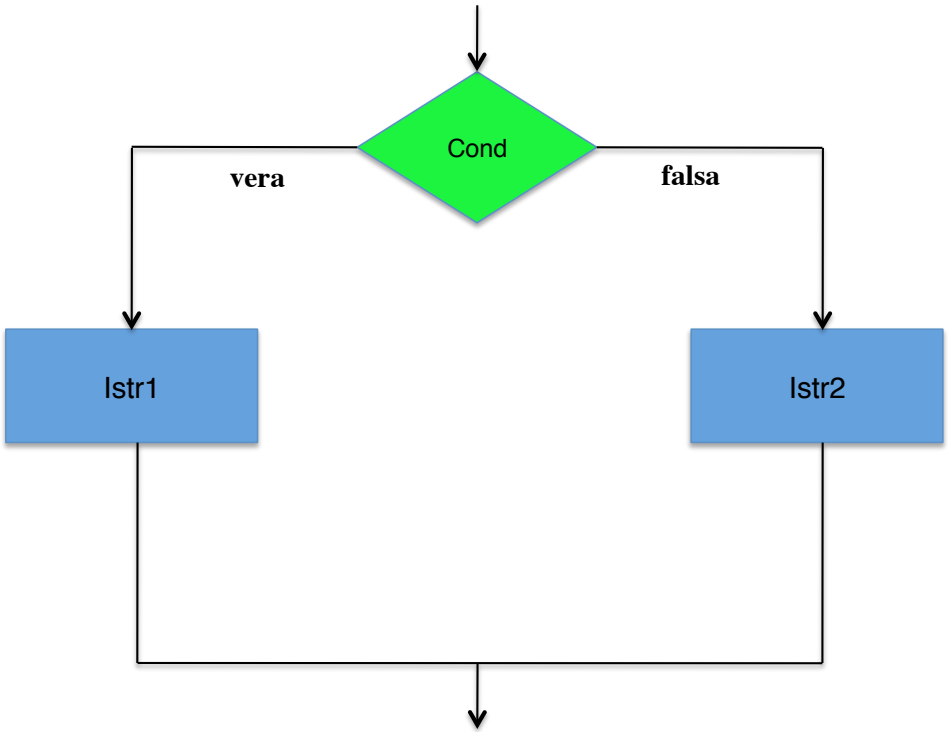
CONDIZIONALE: esempi

Stato iniziale	Condizionale	Stato Finale
	if (x > y) z = x; **else** z = y;	
$\{ x \rightsquigarrow 10, y \rightsquigarrow 20 \}$		$\{ x \rightsquigarrow 10, y \rightsquigarrow 20, z \rightsquigarrow 20 \}$
$\{ x \rightsquigarrow 10, y \rightsquigarrow 5 \}$		$\{ x \rightsquigarrow 10, y \rightsquigarrow 5, z \rightsquigarrow 10 \}$

Diagrammi di flusso: forme



if cond istr1 **else** istr2



Istruzioni di controllo: CONDIZIONALE (2)

- Esiste anche un'istruzione **if-else** in cui manca la parte **else**, nella quale se la condizione è vera, viene eseguita l'istruzione, altrimenti non si fa niente

```
if (condizione) istruzione1
```

- Esiste inoltre una selezione a più vie (vedi **switch**).

Iterazione determinata e indeterminata

- Le istruzioni iterative permettono di ripetere determinate azioni più volte:
  - un numero di volte fissato $\Longrightarrow$ iterazione determinata
    Esempio:
    ```
    fai un giro del parco di corsa per 10 volte
    ```
  - finchè una condizione rimane vera $\Longrightarrow$ iterazione indeterminata
    Esempio:
    ```
    finché non sei sazio
    prendi una ciliegia dal piatto e mangiala
    ```

Istruzioni di controllo: RIPETIZIONE

- Consente di ripetere l'esecuzione di una istruzione (o sequenza di istruzioni) fino al verificarsi di una certa condizione (cfr. esempio del calcolo del prodotto tra due numeri).

  while (condizione) Istruzione

  dove condizione indica una espressione booleana.
- Terminologia: in **while** (C) S , la condizione C è detta guardia e l'istruzione S è detta corpo (del ciclo).
- L'esecuzione di **while** (C) S consiste nel
  (i) Calcolare il valore, sia esso val, dell'espressione booleana C
  (ii) Se val è false, terminare l'esecuzione.
  (iii) Se val è true, eseguire S e ripetere dal punto (i).

- Nota: se condizione è falsa all'inizio, il ciclo non fa nulla.

RIPETIZIONE

Esempio: Riformulazione del passo 4 dell'algoritmo per il prodotto (versione 2):

```
while (b>0)
{
  c =  c+a;
  b =  b-1;
}
```

- Intuitivamente, l'esecuzione di

  while (guardia) corpo

  corrisponde all'esecuzione di una sequenza del tipo

  { corpo corpo corpo ... corpo ...}

- In tale sequenza, ogni ripetizione dell'istruzione corpo viene detta iterazione del ciclo.

  In generale, non è possibile determinare a priori il numero di iterazioni (che può anche essere 0 nel caso in cui la guardia sia falsa nello stato iniziale), dipende dal verificarsi della guardia.

RIPETITIVO: esempi

```
while (b>0)
{
  c =  c+a;
  b =  b-1;
}
```

Stato iniziale	Stato intermedio	Stato Finale
$\{ a \rightsquigarrow 3, b \rightsquigarrow 2, c \rightsquigarrow 0\}$	$\{ a \rightsquigarrow 3, b \rightsquigarrow 1, c \rightsquigarrow 3\}$	$\{ a \rightsquigarrow 3, b \rightsquigarrow 0, c \rightsquigarrow 6\}$

```
while (somma<16)
{
  somma =  somma+a;
}
```

Stato iniziale	Stato intermedio	Stato Finale
$\{ a \rightsquigarrow 9, somma \rightsquigarrow 0\}$	$\{ a \rightsquigarrow 9, somma \rightsquigarrow 9\}$	$\{ a \rightsquigarrow 9, somma \rightsquigarrow 18\}$

RIPETIZIONE

- L'aspetto più critico è il fatto che l'esecuzione di un ciclo può essere fonte di non terminazione dell'esecuzione dell'intero algoritmo.

  while (3>0) x = 0;

- Un ciclo siffatto provoca la non terminazione dell'esecuzione, dal momento che il valore di verità della guardia è sempre true e dunque l'esecuzione corrisponde ad una sequenza infinita

  x = 0; x = 0; .....; x = 0; .....

- È compito di chi definisce l'algoritmo assicurare che i cicli presenti non diano luogo a non terminazione.
- La pratica programmativa, ed il buon senso, suggeriscono ad esempio di utilizzare cicli in cui:
  - il valore di verità della guardia dipende dallo stato;
  - l'esecuzione del corpo comporta modifiche di associazioni nello stato dalle quali dipende il valore di verità della guardia.

RIPETIZIONE

- Le indicazioni appena viste non sono tuttavia sufficienti a garantire la terminazione dell'esecuzione. Si consideri ad esempio il ciclo:

  while (x>0) x = x+1;

  che soddisfa entrambi i requisiti richiesti, ma che può non terminare nel caso in cui venga eseguito a partire da uno stato dove il valore associato al nome x è un valore positivo.
- A partire dagli anni '70 sono stati sviluppati dei metodi formali per la verifica di correttezza di programmi, che comprendono tecniche per la dimostrazione formale di proprietà di terminazione dei cicli.

Istruzione **for**

Quando il numero di ripetizioni necessarie non è noto al momento della scrittura del programma si può ricorrere al **for**

for (istr-1; espr-2; istr-3)
  istruzione

- istr-1 serve a inizializzare il contatore o variabile di controllo
- espr-2 è la verifica di fine ciclo: si verifica se la variabile di controllo ha raggiunto un limite prefissato
- istr-3 serve ad aggiornare la variabile di controllo alla fine del corpo del ciclo
- istruzione è il corpo del ciclo

Ad esempio: `for (i = 1; i <= 10; i=i+1) x = x+1;`

FOR: esempi

```
while (b>0)
{
  c =  c+a;
  b =  b-1;
}
```

può essere reso dall'equivalente programma con il **for**

```
for (i = 0; i < b; i=i+1)
{
  c =  c+a;
}
```

dove `i` è la variabile di controllo che conta quante volte dobbiamo sommare `a`.

Istruzione **do-while**

- Nell'istruzione **while** la condizione viene controllata all'inizio di ogni iterazione.
- L'istruzione **do-while** è simile all'istruzione **while**, ma la condizione viene controllata alla fine di ogni iterazione

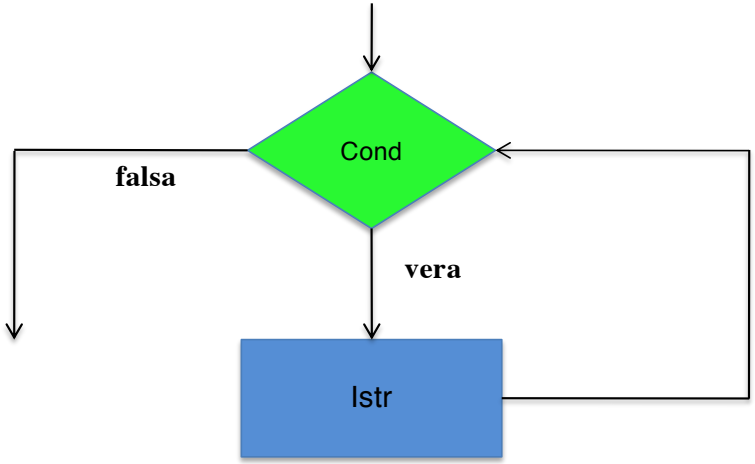
```
do
   istruzione
while (espressione);
```

L'istruzione è *semanticamente* equivalente a

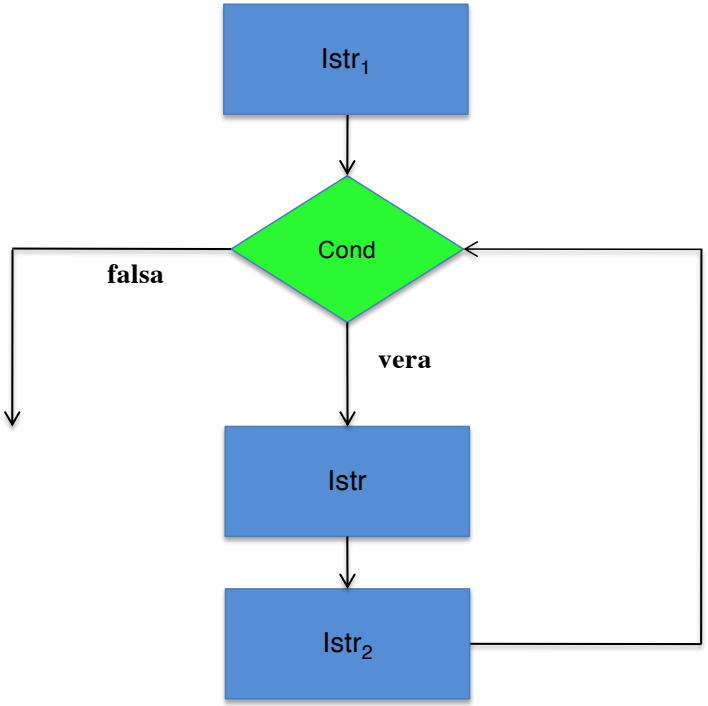
```
istruzione
while (espressione)
  istruzione
```

$\Longrightarrow$ una iterazione viene eseguita comunque.

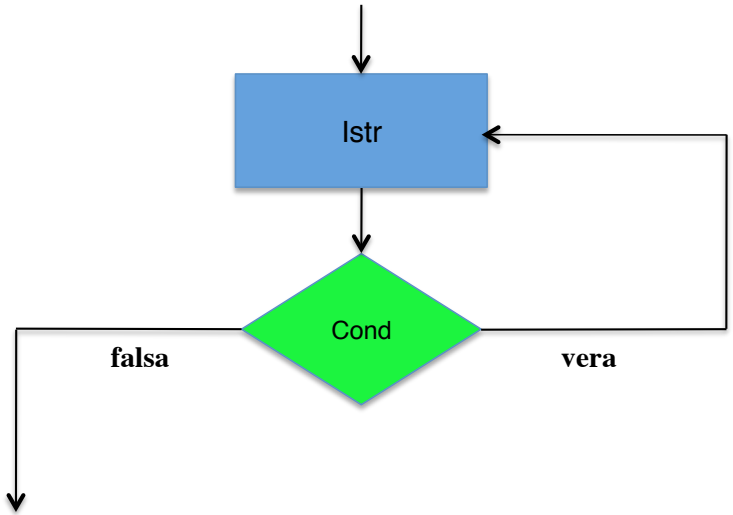
while cond istr



for (istr$_1$: cond; istr$_2$) istr;



do istr **while** cond



Istruzioni di uscita: OUTPUT

- Permette di rendere visibile all'esterno la parte desiderata dello stato. La notazione utilizzata è

  `printf(...,x);`

  dove x indica un nome simbolico e in ... troveremo la stringa usata per il controllo del formato (aspetti su cui torneremo).
- L'esecuzione di questa istruzione consiste nel:
  (i) Recuperare il valore, sia esso val, associato nello stato al nome simbolico x.
  (ii) Produrre in uscita tale valore.
- Questa operazione non comporta la modifica dello stato. Nei linguaggi di programmazione reali, l'esecuzione delle operazioni di uscita produce di solito la visualizzazione di valori su supporti fisici quali video, stampanti etc.

Programmazione strutturata

Si parla di programmazione strutturata se si utilizzano solo le seguenti strutture (concatenate in sequenza o annidate una dentro l'altra, ma non intrecciate tra loro), per alterare il flusso del controllo:

- sequenziale
- condizionale
- iterativa

È stato dimostrato che queste tre strutture sono sufficienti per esprimere un qualsiasi algoritmo. Inoltre ci permettono di:

- scrivere programmi facilmente leggibili e modificabili, e di
- evitare l'uso della programmazione a salti (`go to`) che può portare a quello che si definisce dispregiativamente spaghetti code, una programmazione che porta ad una struttura di controllo del flusso complessa e poco comprensibile.

Attributi degli algoritmi

Un algoritmo deve essere:

- corretto, ovvero deve fornire un risultato corretto;
- composto con i costrutti più opportuni;
- di facile comprensione, per favorire la manutenzione dei programmi;
- possibilmente elegante;
- efficiente in termini di tempo (quanto lavoro o istruzioni occorrono) e di spazio (quanta memoria si occupa). Solitamente l'efficienza si misura in termini di ordine di grandezza.

Problema: Calcolo del valore assoluto di un numero intero

Vediamo alcuni esempi di specifica di algoritmi.

Specifica:

Stato iniziale: $\{\ \text{numero} \rightsquigarrow A\ \}$

Stato finale: $\{\ \text{risultato} \rightsquigarrow |A|\ \}$

dove A indica un (generico) valore intero, diverso da zero.

Algoritmo:

```
if (numero > 0)
        risultato =  numero;
   else risultato =  - numero;
```

Problema: scambiare il valore di due variabili

Specifica:

Stato iniziale: $\{ \mathsf{x} \rightsquigarrow \mathsf{A}, \mathsf{y} \rightsquigarrow \mathsf{B} \}$
Stato finale: $\{ \mathsf{x} \rightsquigarrow \mathsf{B}, \mathsf{y} \rightsquigarrow \mathsf{A} \}$

Algoritmo:

```
temp = x;
x = y;
y = temp;
```

Problema: scambiare il valore di due variabili (cont.)

- Si noti l'utilizzo di un nome simbolico aggiuntivo (temp): è essenziale per poter effettuare correttamente lo scambio tra i valori associati ai due interi (a causa del carattere distruttivo dell'assegnamento e della sequenzialità dell'esecuzione).
- È facile verificare che una sequenza del tipo

```
x = y;
y = x;
```

non consente, in generale, di scambiare i valori associati a x e y.

Esempio:

Stato iniziale		Stato Intermedio
$\{ x \rightsquigarrow 10, y \rightsquigarrow 20\}$	$x = y;$	$\{ x \rightsquigarrow 20, y \rightsquigarrow 20\}$
Stato intermedio		**Stato Finale**
$\{ x \rightsquigarrow 20, y \rightsquigarrow 20\}$	$y = x$	$\{ x \rightsquigarrow 20, y \rightsquigarrow 20\}$

Problema: ordinare due interi positivi

Specifica:

Stato iniziale: $\{\ \text{num1} \rightsquigarrow \text{A},\ \text{num2} \rightsquigarrow \text{B}\ \}$

Stato finale: $\{\ \text{num1} \rightsquigarrow \mathit{max}(\text{A},\text{B}),\ \text{num2} \rightsquigarrow \mathit{min}(\text{A},\text{B})\ \}$

Algoritmo:

```
if (num1 < num2)
    {
        temp = num1;
        num1 = num2;
        num2 = temp;
    }
```

ATTENZIONE: non basta stabilire il massimo e il minimo; occorre anche che, alla fine, il massimo sia associato a num1 e il minimo a num2.

Problema: Elevamento a potenza

Specifica:

Stato iniziale: $\{$ base $\rightsquigarrow$ A, esponente $\rightsquigarrow$ B $\}$ con A$>$0 e B $\geq$0
Stato finale: $\{$ risultato $\rightsquigarrow$ $A^B\}$

- L'algoritmo si basa sulla seguente, ben nota, definizione di elevamento a potenza.

$$
\begin{aligned}
A^0 &= 1 \\
A^B &= \underbrace{A \times A \times \ldots \times A}_{B \text{ volte}} \qquad (\text{se } B > 0)
\end{aligned}
$$

Algoritmo:

```
risultato =  1;
while (esponente > 0)
{
    risultato =  risultato * base;
    esponente =  esponente - 1;
}
```

- Nell'algoritmo, si ipotizza che i valori iniziali di base ed esponente soddisfino i requisiti della specifica.

Input/Output

- Nei problemi visti fino ad ora non ci siamo preoccupati di acquisire i dati iniziali né di produrre in uscita i risultati finali.

  Esempio: Ordinare due valori interi acquisiti in input e stampare i valori ordinati.

  Algoritmo:

```
scanf("%d",&num1);
scanf("%d",&num2);
if num1 < num2
{
      temp =  num1;
      num1 =  num2;
      num2 =  temp;
}
printf("%d",num1);
printf("%d",num2);
```

Moltiplicazione egizia

Primo algoritmo documentato (Papiro di Ahmes, ca 1650 a.C)

Stato iniziale: $\{a \rightsquigarrow A, b \rightsquigarrow B\}$ con $A,B > 0$

Stato finale: $\{p \rightsquigarrow A*B\}$

Algoritmo:

```
scanf("%d",&a); scanf("%d",&b); c = 0
while (a > 0)
{
    if (a è dispari)
        {c = c + b;}
    a = a/2;
    b = b*2; }
printf("%d",c);
```

nota che la divisione è intera

a è dispari può essere verificato con la condizione a%2 == 1

Moltiplicazione egizia (cont.)

Algoritmo:

```
scanf("%d",&a); scanf("%d",&b); c = 0
while (a > 0)
{
    if (a è dispari)
        {c = c + b;}
    a = a/2;
    b = b*2; }
printf("%d",c);
```

a pari $\Rightarrow$ a*b = (a/2)*2b

$$a * b = (a/2) * 2b = (a-1+1)/2 * 2b$$

a dispari $\Rightarrow$

$$(a-1+1)/2 * 2b = ((a-1)/2 + 1/2) * 2b$$

$$((a-1)/2 + 1/2) * 2b = ((a-1)/2) * 2b + b$$

Moltiplicazione egizia (cont.)

Algoritmo:

```
scanf("%d",&a); scanf("%d",&b); c = 0
while (a > 0){
    if (a è dispari) {c = c + b;}
    a = a/2;
    b = b*2;}
printf("%d",c);
```

- $\{a = 16, b = 26, c = 0\}$, $\{a = 8, b = 52, c = 0\}$, $\{a = 4, b = 104, c = 0\}$, $\{a = 2, b = 208, c = 0\}$, $\{a = 1, b = 416, c = 0\}$ $\{a = 0, b = 416, c = 416\}$
- $\{a = 15, b = 26, c = 0\}$, $\{a = 7, b = 52, c = 26\}$, $\{a = 3, b = 104, c = 78\}$, $\{a = 1, b = 208, c = 182\}$ $\{a = 0, b = 416, c = 390\}$

Problema: Calcolare quoziente e resto della divisione tra due numeri naturali non nulli.

Specifica:

Stato iniziale: $\{x \rightsquigarrow A,\ y \rightsquigarrow B\}$ con $A,B > 0$

Stato finale: $\{x \rightsquigarrow A,\ y \rightsquigarrow B,\ q \rightsquigarrow Q,\ r \rightsquigarrow R\}$

con $A = Q \cdot B + R$ e $0 \leq R < B$

Supponiamo che l'esecutore non sappia eseguire direttamente la divisione, ma solo la somma e la sottrazione.

Algoritmo:

```
q = 0;
r = x;
while (r ≥ y)
{
    q = q + 1;
    r = r - y;
}
```

Calcolare il MCD con l'algoritmo di Euclide

- Dati due naturali non nulli si calcola il MCD utilizzando le seguenti proprietà:

$$
\begin{aligned}
MCD(x,x) &= x \\
MCD(x,y) &= MCD(x-y,y) \qquad \text{se } x > y \\
MCD(x,y) &= MCD(x,y-x) \qquad \text{se } y > x
\end{aligned}
$$

Specifica:

Stato iniziale: $\{x \rightsquigarrow A,\ y \rightsquigarrow B\}$ con $A,B > 0$

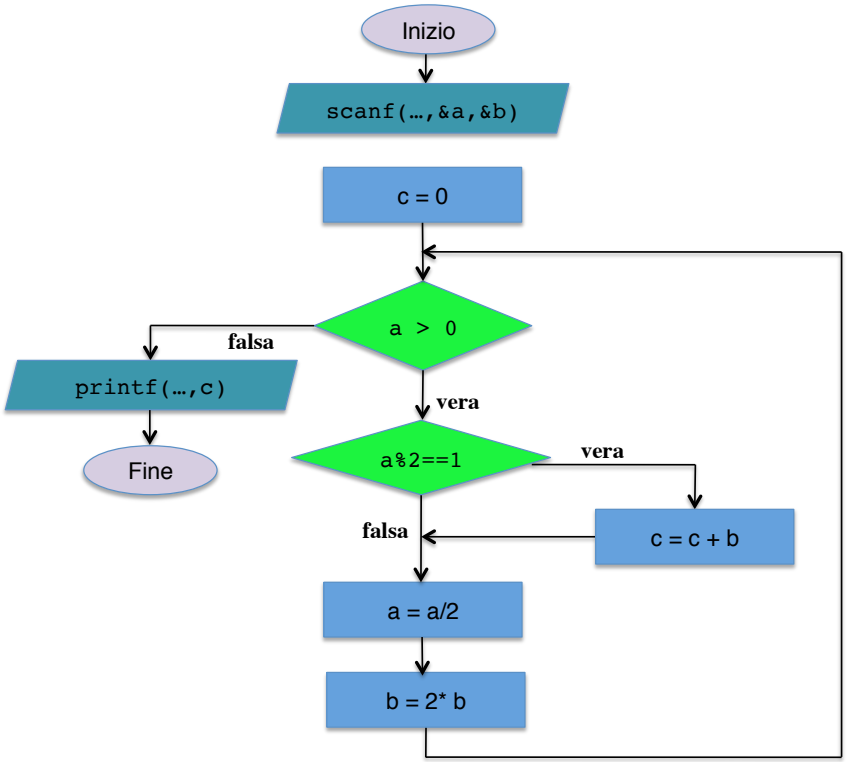
Stato finale: $\{x \rightsquigarrow MCD(A,B)\}$

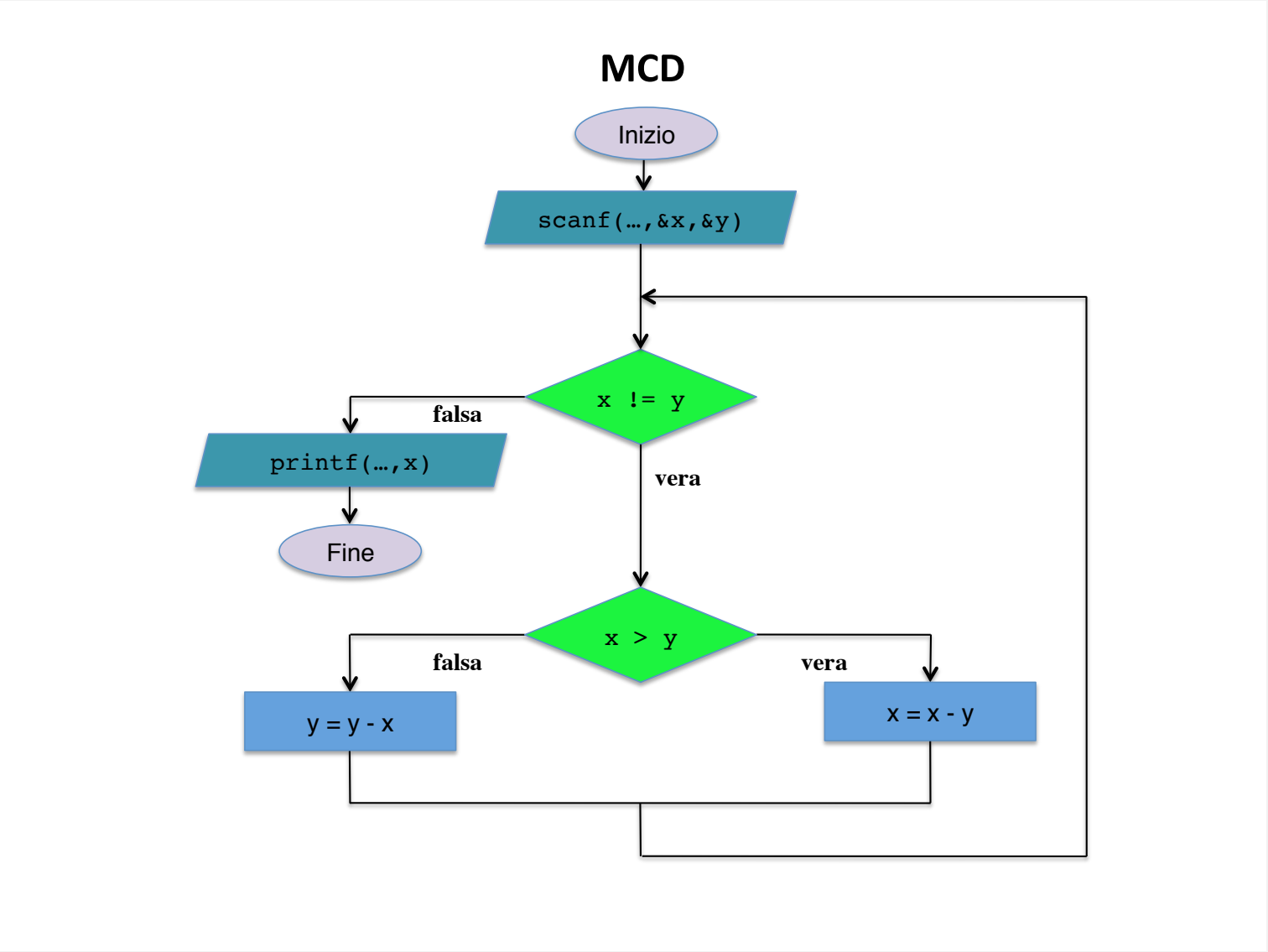
Calcolare il MCD con l'algoritmo di Euclide (cont.)

Algoritmo:

```
while (x != y)
{
    if (x > y)
        x = x - y;
    else
        y = y - x;
}
```

Moltiplicazione Egizia




MCD
Inizio
scanf(…,&x,&y)
x != y
falsa
printf(…,x)
Fine
vera
x > y
falsa
y = y - x
vera
x = x - y

Tipi di dato

- per tipo di dato si intende un insieme di valori e di operazioni che si possono ad essi applicare.
- Alcuni tipi di dato (numeri interi, caratteri, ecc.), detti tipi primitivi sono forniti direttamente dal linguaggio.
- Si possono tuttavia introdurre nuovi tipi di dato adatti al problema da affrontare.
- Nel caso servano dati aggregati (ad es. una data, un vettore, ecc.), si può infine ricorrere ai tipi di dato strutturati. Si deve poter accedere ai singoli elementi.

Esempi di algoritmi

Vediamo dapprima due esercizi, in cui ci concentriamo sullo stato.

Esempio: Ordinare tre valori interi distinti tra loro

Stato iniziale: $\{x \rightsquigarrow A,\ y \rightsquigarrow B,\ z \rightsquigarrow C\}$ con A,B,C distinti tra loro

Stato finale: $\{\ x \rightsquigarrow \max(\{A,B,C\}),\ y \rightsquigarrow \max([\{A,B,C\} \setminus \{\max(\{A,B,C\})\}]),\ z \rightsquigarrow \min(\{A,B,C\})\}$

Algoritmo

Passo 1. "Ordiniamo" x e y, portandoci nello stato intermedio

Stato 1: $\{\ x \rightsquigarrow \max(\{A,B\}),\ y \rightsquigarrow \min(\{A,B\}),\ z \rightsquigarrow C\ \}$

Passo 2. "Ordiniamo" x e z, portandoci nello stato intermedio

Stato 2: $\{\ x \rightsquigarrow \max(\{A,B,C\}),\ y \rightsquigarrow \min(\{A,B\})\ \ z \rightsquigarrow \min(\max(\{A,B\}),\ C)\ \}$

Passo 3. "Ordiniamo" y e z, portandoci nello stato finale desiderato

Cuore della soluzione

```
if (x < y)
{
    temp = x;
    x = y;
    y = temp;
}
```

$\{ x \rightsquigarrow \max(\{A,B\}), y \rightsquigarrow \min(\{A,B\}), z \rightsquigarrow C \}$

```
if (x < z)
{
    temp = x;
    x = z;
    z = temp;
}
```

$\{ x \rightsquigarrow \max(\{A,B,C\}), y \rightsquigarrow \min(\{A,B\}), z \rightsquigarrow \min(\max(\{A,B\}), C) \}$

```
if (y < z)
{
    temp = y;
    y = z;
    z = temp;
}
```

Stato finale

Esempio: Calcolare il massimo di tre interi.

Stato iniziale: $\{x \rightsquigarrow A,\ y \rightsquigarrow B,\ z \rightsquigarrow C\}$

Stato finale: $\{massimo \rightsquigarrow max(\{A,B,C\})\}$

Cuore della soluzione 1 (migliore)

```
if (x > y)
    massimo = x;
else
    massimo = y;
  { max ⇝ max({A,B})}
if (z > massimo)
    massimo = z;
  Stato finale
```

Cuore della soluzione 2

```
if (x > y)
  if (x > z)
    massimo = x;
  else
    massimo = z;
else
  if (y > z)
    massimo = y;
  else
    massimo = z;
```

Vediamo ora due esercizi e la loro soluzione con tutti i dettagli. **Esempio:** Calcolare il massimo di N interi dati da linea di comando, con N dato da linea di comando

- Dobbiamo calcolare il massimo mano a mano che gli interi vengono inseriti
- Utilizziamo un ciclo controllato da una variabile di controllo i che assume tutti i valori fino a N.
- Facciamo in modo che, ad ogni iterazione, il massimo tra i valori fino a quel punto inseriti. Per ottenere questo a ogni iterazione confrontiamo l'elemento corrente con il "massimo in carica".
- Alla fine del ciclo abbiamo il massimo tra tutti i valori.

Soluzione:

```
include <stdio.h>
int main()
{
int n,corr;
int max;
printf("Introduci il numero di interi\n");
scanf("%d", &n);
printf("Introduci il primo intero della sequenza\n");
scanf("%d", &max);
while (n - 1 > 0)
  {
   printf("Introduci il nuovo intero della sequenza\n");
   scanf("%d", &corr);
   if (corr > max)
     max = corr;
   n = n-1;
  }
printf("Il massimo e': %.d\n",max);
return 0; }
```

Per calcolare il massimo di K elementi devo fare K-1 confronti, dato che sono proprio K-1 gli elementi che devono uscire perdenti.

Esempio: Calcolo del fattoriale di un numero naturale. Ricordiamo che:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n & \text{se } n > 0 \end{cases}$$

Specifica:

Stato iniziale: $\{\mathsf{n} \rightsquigarrow \mathsf{N}\}$ con $\mathsf{N} \geq 0$

Stato finale: $\{\mathsf{n} \rightsquigarrow \mathsf{N}, \mathsf{fatt} \rightsquigarrow \mathsf{N!}\}$

- Attenzione: la specifica impone che il valore di n non deve cambiare!

Soluzione:

```
fatt =  1;
i =  1;
while  (i <= n)
{
    fatt =  fatt * i;
    i =  i + 1;
}
```

- i viene chiamata variabile di controllo del ciclo, dato che l'esecuzione di ogni iterazione dipende dal valore di i, o contatore dato che viene incrementata di 1 alla fine di ogni iterazione

  $\Longrightarrow$ per ogni valore di i $\in$ [1,N] .....

- Ad ogni iterazione del ciclo, vale la seguente proprietà:

  $$\texttt{fatt} = \prod_{j=1}^{i-1} j = \text{(i-1)!} \qquad (\bullet)$$

- Al termine del ciclo, quando cioè i=N+1, la ($\bullet$) diventa

  fatt = N!, ovvero lo stato finale desiderato.

Gnocchi in brodo (Pellegrino Artusi, 1891)

... è una minestra da farsene onore; ma se non volete consumare appositamente per lei un petto di pollastra o di cappone, aspettate che vi capiti d'occasione. Cuocete nell'acqua, o meglio a vapore, grammi 200 di patate grosse e farinacee e passatele per istaccio. A queste unite il petto di pollo lesso tritato finissimo colla lunetta, grammi 40 di parmigiano grattato, due rossi d'uovo, sale quanto basta e odore di noce moscata. Mescolate e versate il composto sulla spianatoia sopra a grammi 30 o 40 (che tanti devono bastare) di farina per legarlo, e poterlo tirare a bastoncini grossi quanto il dito mignolo. Tagliate questi a tocchetti e gettateli nel brodo bollente ove una cottura di cinque o sei minuti sarà sufficiente. Questa dose potrà bastare per sette od otto persone. Se il petto di pollo è grosso, due soli rossi non saranno sufficienti...

La ricetta è piacevole da leggere, ma non è facile da seguire.

Gnocchi in brodo (cont.)

DOSE: 7-8 persone
INGREDIENTI: 200 di patate grosse e farinacee, 40 g parmigiano, 2 rossi di uovo, sale e noce moscata q.b., un petto di pollo, 30-40 g farina

- Cuocete nell'acqua o a vapore le patate
- passatele al setaccio
- unite pollo lesso tritato, parmigiano, uova, sale e noce moscata
- versate il composto sulla spianatoia, mescolandolo con la farina
- tirare il composto a bastoncini dello spessore del dito mignolo
- tagliare i bastoncini a tocchetti
- gettare i tocchetti nell'acqua bollente e cuocerli per 5-6 minuti.

La ricetta non è altrettanto piacevole da leggere, ma è facile da seguire, anche se presenta delle ambiguità.

Introduzione al linguaggio C

- Abbiamo già visto come un programma non sia altro che un algoritmo codificato in un linguaggio di programmazione e che
- I linguaggi di programmazione ad alto livello si prestano a codificare algoritmi, pur rimanendo comprensibili, ponendosi quindi ad un livello intermedio tra il linguaggio naturale e il linguaggio macchina
- I compilatori sono programmi che traducono programmi scritti nel linguaggio di più alto livello in programmi equivalenti nel linguaggio macchina.

- Vedremo il cosidetto ANSI C (standard del 1989, con successive aggiunte)
- Il primo programma C: ciao mondo

```
#include <stdio.h>
int main()
  /* Stampa un messaggio sullo schermo. */
{
  printf("Ciao mondo!\n");
  return 0;
}
```

- Questo programma stampa sullo schermo una riga di testo:

```
Ciao mondo!
>
```

- Vediamo in dettaglio ogni riga del programma.

```
/* Stampa un messaggio sullo schermo. */
```

- testo racchiuso tra "/*" e "*/" è un commento
- i commenti sono a "uso umano", cioè servono a chi scrive o legge il programma, per renderlo più comprensibile
- il compilatore ignora i commenti
- attenzione a non dimenticare di chiudere i commenti con */, altrimenti tutto il resto del programma viene ignorato

`int main()`

- è una parte presente in tutti i programmi C
- le parentesi "`(`" e "`)`" dopo main indicano che main è una funzione
- i programmi C sono composti da una o più funzioni, tra le quali ci deve essere la funzione `main`, che è quella principale
  $\Longrightarrow$ `main` è una funzione speciale, perché l'esecuzione del programma comincia l'esecuzione all'inizio di `main`
- la parentesi "{" apre il corpo della funzione e "}" lo chiude
  - la coppia di parentesi e la parte racchiusa da esse costituiscono un blocco
  - il corpo della funzione contiene le istruzioni (e dichiarazioni) che costituiscono la funzione

```
printf("Ciao mondo!\n");
```

- è un'istruzione semplice (ordina al computer di eseguire un'azione) in questo caso visualizzare (stampare) sullo schermo la sequenza di caratteri tra apici
- ogni istruzione semplice deve terminare con ";"
- oltre alle istruzioni semplici, esistono anche istruzioni composte (che non devono necessariamente terminare con ";")
- la parte racchiusa in una coppia di doppi apici è una stringa (di caratteri)
- "\n" non viene visualizzato sullo schermo, ma provoca la stampa di un carattere di fine riga
  - "\" è un carattere di escape e, insieme al carattere che lo segue, assume un significato particolare (sequenza di escape)
- in realtà anche `printf` è una funzione, e l'istruzione di sopra è un'attivazione di funzione (le vedremo più avanti)

`#include <stdio.h>`

- è una direttiva di compilazione
- viene interpretata dal compilatore durante la compilazione
- la direttiva "`#include`" dice al compilatore di includere il contenuto di un file nel punto corrente
- `<stdio.h>` è un file che contiene i riferimenti alla libreria standard di input/output (dove è definita la funzione di stampa `printf`)
- il linguaggio C non prevede istruzioni esplicite di input/output. Queste operazioni sono definite tramite funzioni nella libreria standard di input/output.

Note:

- è importante distinguere i caratteri maiuscoli da quelli minuscoli `Main`, `MAIN`, `Printf`, `PRINTF` non andrebbero bene
- si è usata l'indentazione per mettere in evidenza la struttura del programma ▸)

Alcune varianti del programma

```
#include <stdio.h>
int main()
  /* Stampa un messaggio sullo schermo. */
{
  printf("Ciao");
  printf(" mondo!\n");
  return 0;
}
```

- produce lo stesso effetto del programma precedente
- la seconda invocazione di `printf` incomincia a stampare dal punto in cui aveva smesso la prima
- Cosa viene stampato se usiamo

```
printf("Ciao");
printf("mondo!\n");
```

```
printf("Ciao\n");
printf("mondo!\n");
```

Un altro programma: area di un rettangolo

```
#include <stdio.h>

int main() {
 int base;
 int altezza;
 int area;

 base = 3;
 altezza = 4;
 area = base * altezza;

 printf("Area: %d\n", area);
 return 0;
}
```

Quando viene eseguito stampa:

```
Area: 12
>
```

Le variabili (di programma)

Lo stato rappresenta il contenuto (modificabile) della memoria.

Una posizione della memoria, destinata a contenere dati, si identifica con una variabile Le variabili dunque rappresentano, nei programmi, le associazioni (modificabili) dello stato
$\Longrightarrow$ cf. $x \rightsquigarrow val$ nello pseudo-linguaggio

Una variabile è caratterizzata dalle seguenti proprietà:

1. nome: serve a identificarla — esempio: `area`
2. valore: valore associato allo stato corrente — Esempio: 4 (può cambiare durante l'esecuzione)
3. tipo: specifica l'insieme dei possibili valori assunti durante l'esecuzione — Esempio: `int` (numeri interi) e implicitamente le operazioni possibili.
4. indirizzo: della cella di memoria a partire dal quale è memorizzato il valore.

Nome, tipo e indirizzo non possono cambiare durante l'esecuzione.

Le variabili (cont.)

- Il nome di una variabile è un identificatore C
  $\Longrightarrow$ sequenza di lettere, cifre, e _ che inizia con una lettera o con _
  Esempio: `Numero_elementi`, `x1`, ma non `1_posto`
  - può avere lunghezza qualsiasi, ma solo i primi 31 caratteri sono significativi
  - lettere minuscole e maiuscole sono considerate distinte
- Ad ogni variabile è associata una cella di memoria o più celle consecutive, a seconda del suo tipo. Il suo indirizzo è quello della prima cella.
- Analogia con una scatola di scarpe etichettata in uno scaffale
  - nome $\Longrightarrow$ etichetta
  - valore $\Longrightarrow$ scarpa che c'è nella scatola
  - tipo $\Longrightarrow$ capienza (che tipo di scarpe ci metto dentro)
  - indirizzo $\Longrightarrow$ posizione nello scaffale (la scatola è incollata)

  N.B.
  - non tutte le variabili sono denotate da un identificatore (strut. din.)
  - non tutti gli identificatori sono identificatori di variabile (ad es. funzioni, tipi, parole riservate, ...)

Area del rettangolo

- `int base;` — è una dichiarazione di variabile
  - viene creata la scatola e incollata allo scaffale
  - ha tipo `int` $\Longrightarrow$ può contenere interi
  - ha nome `base`
  - ha un indirizzo (posizione nello scaffale), che è quello della cella di memoria associata alla variabile
  - ha un valore iniziale, che però non è significativo (è casuale)
    $\Longrightarrow$ la scatola viene creata piena, però con una scarpa scelta a caso, ovvero
    $\Longrightarrow$ l'associazione nello stato è del tipo nome $\rightsquigarrow$?

- `int altezza;`
  `int area;`
  $\Longrightarrow$ come per `base`

Variabili numeriche

Variabili intere

- per dichiarare variabili intere si può usare il tipo `int`
- i valori di tipo `int` sono rappresentati in C con almeno 16 bit
- il numero effettivo di bit dipende dal compilatore
  Esempio: 32 bit per il compilatore gcc (usato in ambiente Unix)
- in C esistono altri tipi per variabili intere (`short`, `long`) — li vedremo più avanti

Variabili reali

- per dichiarare variabili reali si può usare il tipo `float`
  Esempio: `float temperatura;`

Area del rettangolo

`base = 3;`
è un'istruzione di assegnamento (come nello pseudo-linguaggio)

- in C l'operatore di assegnamento è denotato dal simbolo "="
- come già sappiamo, l'effetto è di modificare una associazione nello stato
  $\Longrightarrow$ in questo caso il valore 3 viene associato a base, come?
  $\Longrightarrow$ il nuovo valore viene scritto nella spazio associato alla variabile
- a questo punto la variabile `base` ha un valore significativo
  $\Longrightarrow$ da base $\rightsquigarrow$? a base $\rightsquigarrow$ 3

`altezza = 4;` $\Longrightarrow$ come sopra

`area = base * altezza;`
a destra di "=" possono comparire espressioni $\Longrightarrow$ il valore assegnato è quello dell'espressione calcolata nello stato corrente

- una variabile all'interno di una espressione sta per il valore ad essa associato in quel momento (cf. pseudo-linguaggio)

Nota: operatori aritmetici tra interi del C sono: `+`, `-`, `*`, `/`, `%`, ...

Area del rettangolo

```
printf("Area: %d\n", area);
```

- è un'istruzione di stampa
- il primo argomento è la stringa di formato che può contenere specificatori di formato
- lo specificatore di formato `%d` indica che deve essere stampato un intero in notazione decimale (`d` per decimal)
- ad ogni specificatore di formato nella stringa deve corrispondere un valore che deve seguire la stringa di formato tra gli argomenti di `printf`

  Esempio: `printf(" %d %d··· %d", i1, i2, ..., in );`
- nel caso di `printf("Ciao mondo!\n");` la stringa di formato non conteneva specificatori e quindi non vi erano altri argomenti.

Struttura dei programmi C

- Nel semplice programma che abbiamo appena analizzato possiamo già vedere la struttura generale di un programma C.

```
/* DIRETTIVE DI COMPILAZIONE */
#include <stdio.h>
 main() {

 /* PARTE DICHIARATIVA */
  int base;
  int altezza;
  int area;

 /* PARTE ESECUTIVA */
  base = 3;
  altezza = 4;
  area = base * altezza;
  printf("Area: %d\n", area);
}
```

Un programma C deve contenere nell'ordine:

- Una parte contenente direttive per il compilatore. Nel nostro programma la direttiva

```
#include <stdio.h>
```

- l'identificatore predefinito `main` seguito dalle parentesi `()`.
- due parti racchiuse tra parentesi graffe
  - la parte dichiarativa. Nell'esempio:

```
int base;
int altezza;
int area;
```

  - la parte esecutiva. Nell'esempio:

```
base = 3;
altezza = 4;
area = base * altezza;
printf("Area: %d\n", area);
```

La parte dichiarativa

- Posta prima della codifica dell'algoritmo, obbliga a dichiarare i nomi simbolici che saranno presenti nello stato e di cui farà uso nella parte esecutiva. Prima si dichiara e poi si usa. Contiene i seguenti elementi:
  - la sezione delle dichiarazioni di costanti
  - la sezione delle dichiarazioni di variabili;
- Le dichiarazioni: rendono più pesante la fase di costruzione dei programmi, ma consentono di individuare e segnalare errori in fase di compilazione.

  Esempio:

```
int x;
int alfa;
alfa = 0;
x=alfa;
alba=alfa+1;
```

- Il compilatore vede `alba` come variabile non dichiarata.

Dichiarazioni di variabili

- Abbiamo già visto esempi di dichiarazioni di variabili.

```
float x;
int base;
int altezza;
```

- Ad ogni variabile viene attribuito, al momento della dichiarazione, un tipo

  $\Longrightarrow$ specifica l'insieme dei valori che la variabile può assumere

- La dichiarazione può anche attribuire un valore iniziale alla variabile (inizializzazione)

```
int x = 0;
```

- Variabili dello stesso tipo possono essere dichiarate contemporaneamente

```
int base, altezza, area;
```

(ma inizializzate singolarmente)

Esempio: `int x, y, z=0;` solo `z` è inizializzata a `0`.

Dichiarazioni di costanti (variabili *read-only*)

- Una dichiarazione di costante[2] crea un'associazione non modificabile $\Longrightarrow$ associa in modo permanente un valore ad un identificatore. Di solito si estende il campo dei valori rappresentabili.

  Esempio:

  ```
  const float PiGreco=3.14;
  const int N=100;
  ```

- L'associazione tra il nome `PiGreco` ed il valore `3.14` non può essere modificata durante l'esecuzione.
- Come per le dichiarazioni di variabili, più costanti dello stesso tipo possono essere dichiarate insieme

  Esempio:

  ```
  const float PiGreco=3.14, e=2.718;
  const int N=100, M=200;
  ```

- **N.B.** cosa succede se si modifica una costante non è specificato dallo standard ANSI C, dipende dal compilatore.

[2] cf. uso diverso di `#define`, vedi pagina successiva.

Digressione sulle costanti: la direttiva `#define`

- La dichiarazione di costante, ad esempio

  ```
  const int Nmax = 10;
  ```

  causa l'allocazione di memoria (si tratta di una dichiarazione di variabile *read only*)
- C'è un altro modo per ottenere un identificatore costante, che utilizza la direttiva **#define**.

  ```
  #define Nmax 10
  ```
- **#define** è una direttiva di compilazione
- dice al compilatore di sostituire ogni occorrenza di `Nmax` con `10` prima di compilare il programma
- a differenza di **const** non alloca memoria

Uso di costanti

- Con la dichiarazione `const float PiGreco=3.14;`
  l'istruzione
  `AreaCerchio=PiGreco*RaggioCerchio*RaggioCerchio;`
  è equivalente a
  `AreaCerchio=3.14*RaggioCerchio*RaggioCerchio`
- Maggiore leggibilità dei programmi, dovuta all'uso di nomi simbolici
- Maggiore adattabilità dei programmi che usano costanti

  Esempio:
  Per aumentare la precisione, basta cambiare la dichiarazione in
  `const float PiGreco = 3.1415;`

  Senza l'uso della costante si dovrebbero rimpiazzare nel codice tutte le occorrenze di `3.14` in `3.1415` ...
  NOTA: anche con la `#define` abbiamo gli stessi vantaggi

Area di un rettangolo di dimensioni lette da tastiera

```
#include <stdio.h>

int main()
{
  int base, altezza, area;

  printf("Immetti base del rettangolo e premi INVIO\n");
  scanf("%d", &base);
  printf("Immetti altezza del rettangolo e premi INVIO\n");
  scanf("%d", &altezza);

  area = base * altezza;

  printf("Area: %d\n", area);
  return 0;
}
```

Nuova istruzione: `scanf("%d", &base);`

- `scanf` è la funzione duale di `printf`
- legge da input (tastiera) un valore intero e lo assegna alla variabile `base`
- `"%d"` è la stringa di controllo del formato (in questo caso viene letto un intero in formato decimale)
- "`&`" è l'operatore di indirizzo
  - `&base` indica (l'indirizzo del)la locazione di memoria associata a `base`
  - `scanf` memorizza in tale locazione il valore letto
- quando viene eseguita `scanf` il programma si mette in attesa che l'utente immetta un valore. Quando l'utente digita Invio
  1. la sequenza di caratteri immessa viene convertita in un intero (formato `%d`) e
  2. l'intero ottenuto viene assegnato alla variabile `base` (viene cioè scritto nella/e cella/e di memoria a partire dall'indirizzo passato a `scanf`)

N.B. il precedente valore della variabile `base` va perduto

Esempio di esecuzione

- Vediamo cosa avviene durante l'esecuzione (indichiamo in rosso ciò che l'utente digita e in particolare con $\hookleftarrow$ il tasto Invio).

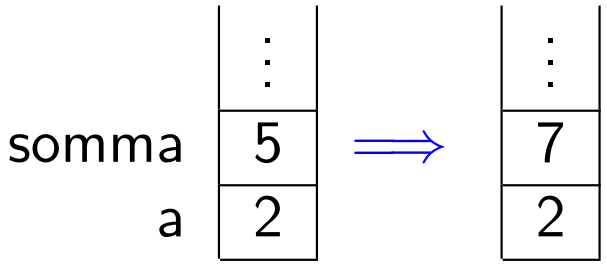
```
Immetti base del rettangolo e premi INVIO
5 ↩
Immetti altezza del rettangolo e premi INVIO
4 ↩
Area: 20
```

Assegnamento

- Ricordiamo che l'esecuzione di `x = exp` corrisponde a:
  1. valutare il valore dell'espressione `exp` a destra di "=" (usando i valori correnti delle variabili);
  2. assegnare poi tale valore alla variabile `x` a sinistra di "=".

Esempio:

```
somma = 5;
a = 2;
somma = somma + a;
```



Esempio:

	a	b
`int a, b;`	?	?
`a = 2;`	2	?
`b = 3;`	2	3
`a = b;`	3	3
`a = a + b;`	6	3
`b = a + b;`	6	9

Osservazioni sull'assegnamento

- **Attenzione:** A sinistra di "=" ci deve essere un identificatore di variabile

  $\Longrightarrow$ denota la corrispondente associazione modificabile nello stato.

  Esempio: Quali istruzioni sono corrette e quali no?

  `a = a;` SI corretta (anche se poco significativa ...)

  `a = 2 * a;` SI corretta (il valore associato ad `a` si raddoppia)

  `5 = a;` NO, `5` non denota una associazione modificabile nello stato ma un valore costante

  `a + b = c;` NO, `a+b` è un'espressione, non una variabile!

Tipi di dato semplici

- Abbiamo visto nei primi esempi che il C tratta vari tipi di dato $\Longrightarrow$ le dichiarazioni associano variabili e costanti al corrispondente tipo
- Per tipo di dato si intende un insieme di valori e un insieme di operazioni che possono essere applicate ad essi.

  Esempio:
  I numeri interi `{…, -2, -1, 0, 1, 2, …}` e le usuali operazioni aritmetiche (somma, sottrazione, …)
- Ogni tipo di dato ha una propria rappresentazione in memoria (codifica binaria) che utilizza un certo numero di celle di memoria.
- Il meccanismo dei tipi ci consente di trattare le informazioni in maniera astratta, cioè prescindendo dalla loro rappresentazione concreta.

L'uso di variabili con tipo ha importanti conseguenze quali:

- per ogni variabile è possibile determinare a priori l'insieme dei valori ammissibili e l'insieme delle operazioni ad essa applicabili
- per ogni variabile è possibile determinare a priori la quantità di memoria necessaria per la sua rappresentazione
- è possibile rilevare a priori (a tempo di compilazione) errori nell'uso delle variabili all'interno di operazioni non lecite per il tipo corrispondente

  Esempio: Nell'espressione `y + 3` se la variabile `y` non è stata dichiarata di tipo numerico si ha un errore (almeno dal punto di vista concettuale) rilevabile a tempo di compilazione (cioè senza eseguire il programma).

Classificazione dei tipi

- Tipi semplici: per rappresentare informazioni semplici
  Esempio: una temperatura, una misura, una velocità, ecc.
- Tipi strutturati: per rappresentare informazioni costituite dall'aggregazione di varie componenti
  Esempio: una data, una matrice, una fattura, ecc.
- Un valore di un tipo semplice è logicamente indivisibile, mentre un valore di un tipo strutturato può essere scomposto nei valori delle sue componenti
  Esempio: un valore di tipo data è costituito da tre valori
- Il C mette a disposizione un insieme di tipi predefiniti (tipi built-in) e dei meccanismi per definire nuovi tipi (tipi user-defined)

  Nota: con `T` identificatore di tipo, nel seguito indichiamo con `sizeof(T)` lo spazio (in byte) necessario per la memorizzazione di valori di tipo `T` (vedremo che `sizeof` è una funzione C).

Tipi semplici built-in

- interi
- reali
- caratteri

Per ciascun tipo consideriamo i seguenti aspetti:

1. intervallo di definizione (se applicabile)
2. notazione (sintassi) per le costanti
3. operatori
4. predicati (operatori di confronto)
5. formati di ingresso/uscita

Tipi interi: interi con segno

- `int`: un intero rappresentabile sulla macchina (segnaposto `%d`)
- dimensione e gamma di valori rappresentabile dipende dalla macchina su cui viene compilato il programma.
- la funzione predefinita `sizeof()` fornisce la lunghezza in byte di un qualsiasi tipo o variabile C. Controllare sul proprio con il comando: `printf("%d\n",sizeof(int));`

Tipi interi: interi con segno (cont.)

- 3 tipi:
  `short`
  `int`
  `long`
- **Intervallo di definizione:** da -2^{n-1} a $2^{n-1}-1$, dove n dipende dal compilatore
- I valori limite sono contenuti nel file limits.h (da includere), che definisce le costanti:
  `SHRT_MIN`, `SHRT_MAX`, `INT_MIN`, `INT_MAX`, `LONG_MIN`, `LONG_MAX`
- Vale: `sizeof(short)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long)`
  `sizeof(short)` $\geq$ 2 (ovvero, almeno 16 bit)
  `sizeof(long)` $\geq$ 4 (ovvero, almeno 32 bit)
- Compilatore gcc: `short`: 16 bit, `int`: 32 bit, `long`: 32 bit

Notazione per le costanti: in decimale: `0`, `10`, `-10`, ...

- Per distinguere `long` (solo nel codice): `10L` (oppure `10l`, ma `l` sembra `1`).

Operatori: `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `>`, `<=`, `>=`

N.B.: l'operatore di uguaglianza si rappresenta con `==` (mentre `=` è utilizzato per il comando di assegnamento!)

Ingresso/uscita: tramite `printf` e `scanf`, con i seguenti specificatori di formato (dove `d` indica "decimale"):

`%hd` per `short`
`%d` per `int`
`%ld` per `long` (con `l` minuscola)

Tipi interi: interi senza segno

- 3 tipi:
  `unsigned short`
  `unsigned int`
  `unsigned long`
- **Intervallo di definizione:** da 0 a 2^n-1, dove n dipende dal compilatore.
  Il numero n di bit è lo stesso dei corrispondenti interi con segno.
- Le costanti definite in limits.h sono:
  `USHRT_MAX`, `UINT_MAX`, `ULONG_MAX` (n.b. il minimo è sempre 0)

Notazione per le costanti:

- decimale: come per interi con segno
- esadecimale: `0xA`, `0x2F4B`, ...
- ottale: `012`, `027513`, ...

- Nel codice si possono far seguire le cifre del numero dallo specificatore `u` (ad esempio `10u`).

Ingresso/uscita: tramite `printf` e `scanf`, con i seguenti specificatori di formato:

`%u`	per numeri in decimale
`%o`	per numeri in ottale
`%x`	per numeri in esadecimale con cifre `0`, ..., `9`, `a`, ..., `f`
`%X`	per numeri in esadecimale con cifre `0`, ..., `9`, `A`, ..., `F`

Per interi `short` si antepone `h`
`long` si antepone `l` (minuscola)

Operatori: tutte le operazioni vengono fatte modulo 2^n.

Caratteri

- Servono per rappresentare caratteri alfanumerici attraverso opportuni codici, tipicamente il codice ASCII (American Standard Code for Information Interchange).
- Un codice associa ad ogni carattere un intero:
  Esempio: Codice ASCII:

carattere:	'0'	···	'9'	':'	';'	'<'
intero (in decimale):	48	···	57	58	59	60

carattere:	'a'	···	'z'	'{'	'|'	'}'
intero (in decimale):	97	···	122	123	124	125

carattere:	'A'	···	'Z'	'['	'\'	']'
intero (in decimale):	65	···	90	91	92	93

Caratteri (cont.)

- un singolo byte, in grado di contenere un carattere (codifica ASCII). Segnaposto %c.

```
char a='a'; // i caratteri si indicano tra apici
```

- In C i caratteri possono essere usati come gli interi (un carattere coincide con il codice che lo rappresenta).
  Infatti:

```
int a='a';
printf("%c\n",a);
printf("%d\n",a);
```

  stampa prima a e poi 97, che corrisponde alla codifica ASCII di a

- Nessuna relazione tra un carattere per rappresentare una cifra e la cifra stessa: '2' non è 2.

Intervallo di definizione: dipende dal compilatore

- Vale: `sizeof(char)` $\leq$ `sizeof(int)`
  Tipicamente i caratteri sono rappresentati con 8 bit.

  Operatori: sono gli stessi di `int` (operazioni effettuate utilizzando il codice del carattere).

  Costanti: `'A'`, `'#'`, ...

  Esempio:

```
char x, y, z;
x = 'A';
y = '\n';
z = '#';
```

- In C l'apice singolo ' delimita singoli caratteri, mentre l'apice doppio " delimita stringhe di caratteri.

Come non va usato il codice

- Confrontiamo:

```
char x, y, z;
x = 'A';
y = '\n';
z = '#';
```

```
char x, y, z;
x = 65;  /* codice ASCII di 'A' */
y = 10;  /* codice ASCII di '\n' */
z = 35;  /* codice ASCII di '#' */
```

- Non è sbagliato, però è pessimo stile di programmazione.
- Non è detto che il codice dei caratteri sia quello ASCII.
  $\Longrightarrow$ Il programma non sarebbe portabile.

Ingresso/uscita: tramite `printf` e `scanf`, con specificatore di formato `%c`

Attenzione: in ingresso non vengono saltati gli spazi bianchi e gli a capo

Esempio:

```
int i, j;
printf("Immetti due interi\n");
scanf("%d%d", &i, &j);
printf("%d %d\n", i, j);
```

Immetti due interi
> 18 25↩
18 25

```
int i, j;
char c;
printf("Immetti due interi\n");
scanf("%d%c%d", &i, &c, &j);
printf("%d %d %d\n", i, c, j);
```

Immetti due interi
> 18 25↩
18 32 25

- 32 è il codice ASCII del carattere ' ' (spazio)

Attenzione:

- è necessario specificare i salti degli spazi o di altri caratteri simili solo nel caso si leggano caratteri (ovvero quando si usa lo specificatore '%c'). Ad esempio in scanf("%c % c", &x, &y) dove voglio leggere due caratteri separati da uno spazio bianco non significativo.
- non è necessario se si desidera leggere interi, reali o stringhe. Ad esempio in scanf("%d%d", &i, &j)

- Funzioni per la stampa e la lettura di un singolo carattere (da usare in alternativa a `printf` e `scanf`):

  `putchar(c);` ... stampa il carattere memorizzato in `c`

  `c = getchar();` ... legge un carattere e lo assegna alla variabile `c`

Esempio:

```
char c;
putchar('A');
putchar('\n');
c = getchar();
putchar(c);
```

Tipi reali

- `float`, `double`,`long double`: sono i tipi usati per rappresentare i reali, con diversi gradi precisione. Si usa la notazione in virgola mobile (floating point).

```
float x=123.34;
double y=100.1e5; //anche notazione scientifica
```

- segnaposto `%f` e `%L`;

Tipi reali (cont.)

I reali vengono rappresentati in virgola mobile (floating point).

- 3 tipi:
  `float`
  `double`
  `long double`
- **Intervallo di definizione:**

	sizeof	cifre significative	min esp.	max esp.
`float`	4	6	−37	38
`double`	8	15	−307	308
`long double`	12	18	−4931	4932

- Le grandezze precedenti dipendono dal compilatore e sono definite nel file `float.h`.
- Deve comunque valere la relazione:
  `sizeof(float)` $\leq$ `sizeof(double)` $\leq$ `sizeof(long double)`

Costanti: con punto decimale o notazione esponenziale

Esempio:

```
double x, y, z, w;
x = 123.45;
y = 0.0034;     /* oppure y = .0034 */
z = 34.5e+20;   /* oppure z = 34.5E+20 */
w = 5.3e-12;
```

- Nei programmi, per denotare una costante di tipo
  - `float`, si può aggiungere `f` o `F` finale
    Esempio: `float x = 2.3e5f;`
  - `long double`, si può aggiungere `L` o `l` finale
    Esempio: `long double x = 2.34567e520L;`

Operatori: come per gli interi (tranne "%")

Ingresso/uscita: tramite `printf` e `scanf`, con diversi specificatori di formato

Output con printf (per `float`):

- `%f` ... notazione in virgola fissa
  `%8.3f` ... 8 cifre complessive, di cui 3 cifre decimali

  Esempio:

```
float x = 123.45;
printf("|%f| |%8.3f| |%-8.3f|\n", x, x, x);
```

```
|123.449997| | 123.450| |123.450 |
```

  dove il `-` indica l'allineamento a sinistra

- `%e` (oppure `%E`) ... notazione esponenziale
  `%10.3e` ... 10 cifre complessive, di cui 3 cifre decimali

  Esempio:

```
double x = 123.45;
printf("|%e| |%10.3e| |%-10.3e|\n", x, x, x);
```

```
|1.234500e+02| | 1.234e+02| |1.234e+02 |
```

Input con scanf (per `float`):
si può usare indifferentemente `%f` o `%e`.

Riassunto degli specificatori di formato per i tipi reali:

	float	double	long double
printf	`%f`, `%e`	`%f`, `%e`	`%Lf`, `%Le`
scanf	`%f`, `%e`	`%lf`, `%le`	`%Lf`, `%Le`

Booleani ed operatori

- In C non esiste un tipo Booleano. Si usa il tipo `int`:
  - `0` rappresenta FALSO;
  - `diverso da 0` (di solito 1) rappresenta VERO.
- Operatori logici:
  - `!`: NOT (operatore unario). Esempio: `!a`;
  - `&&`: AND (operatore binario). Esempio: `a && b`;
  - `||`: OR (operatore binario). Esempio: `a || b`;

  Restituiscono un valore intero pari a 0 o 1 a seconda del valore (falso/vero) dell'espressione.
  Altri operatori lavorano sui singoli bit: ad esempio operatori di shift (<<, >>), AND (&), OR (|), XOR (^) ...

- Scrivere un programma che simuli il gioco della morra cinese fra due giocatori, ovvero che legge due valori v1 e v2 che rappresentano uno dei tre valori “forbici”, “carta” e “sasso” e quindi stampa il vincitore oppure indicare che il gioco è pari. Si ricorda che “forbici” batte “carta”, “carta” batte “sasso” e, infine,“sasso” batte “forbici”.

Soluzione 1:

Supponiamo si tratti di caratteri.

```
#include <stdio.h>
int main()
{
char v1, v2;
printf("Digitare i valori di v1 e v2 \n");
scanf("%c %c", &v1, &v2);
if (v1 == v2)
  printf("Il gioco  pari \n");
else
  if ((v1 == 'f' && v2 == 'c')||(v1 == 'c' && v2 == 's')||
    (v1 == 's' && v2 == 'f'))
    printf("Il vincitore  il primo giocatore \n");
  else
    printf("Il vincitore  il secondo giocatore \n");
return 0; }
```

Soluzione 2: Codificando i tre valori con tre interi, cioè 'f' = 3, 'c' = 2 e 's' = 1, si vede che esiste una sorta di ordinamento modulo 3: 3 vince su 2, 2 vince su 1 e 1 vince su 3.

```
#include <stdio.h>
int main()
{
int v1, v2,v;
printf("Digitare i valori di v1 e v2 \n");
scanf("%d %d", &v1, &v2);
if (v1 == v2)
  printf("Il gioco  pari \n");
else
  v = v1-v2 % 3;
  printf("%d \n",v);
  if (v == 1)
    printf("Il vincitore è il primo giocatore \n");
  else
    printf("Il vincitore è il secondo giocatore \n");
return 0; }
```

Conversioni di tipo nelle espressioni

- Le espressioni aritmetiche hanno un valore ed un tipo dettato da quello delle variabili in gioco.
- In un'operazione gli operandi di tipo diverso vengono convertiti nello stesso tipo applicando alcune regole automatiche. In genere si amplia il campo dei valori rappresentabili.

Esempio

Se `x` ha tipo `int` e `y` ha tipo `float`, il risultato dell'espressione `x+y` viene automaticamente convertito a `float`

Conversioni di tipo

Situazioni in cui si hanno conversioni di tipo

- quando in un'espressione compaiono operandi di tipo diverso
- durante un'assegnamento `x = y`, quando il tipo di `y` è diverso da quello di `x`
- esplicitamente, tramite l'operatore di cast
- nel passaggio dei parametri a funzione (più avanti)
- attraverso il valore di ritorno di una funzione (più avanti)

Una conversione può o meno coinvolgere un cambiamento nella rappresentazione del valore.

da `short` a `long` (dimensioni diverse)
da `int` a `float` (anche se stessa dimensione)

Conversioni implicite tra operandi di tipo diverso nelle espressioni

Quando un'espressione del tipo x *op* y coinvolge operandi di tipo diverso, avviene una conversione implicita secondo le seguenti regole:

1. ogni valore di tipo `char` o `short` viene convertito in `int`
2. se dopo il passo 1. l'espressione è ancora eterogenea si converte l'operando di tipo inferiore facendolo divenire di tipo superiore secondo la seguente gerarchia:

   `int` → `long` → `float` → `double` → `long double`

Esempio: `int x; double y;`
Nel calcolo di `(x+y)`:

1. `x` viene convertito in `double`
2. viene effettuata la somma tra valori di tipo `double`
3. il risultato è di tipo `double`

Conversioni nell'assegnamento

Si ha in `x = exp` quando i tipi di `x` e `exp` non coincidono.

- La conversione avviene sempre a favore del tipo della variabile a sinistra:

  se si tratta di una promozione non si ha perdita di informazione

  se si ha una retrocessione si può avere perdita di informazione

  Esempio:

```
int i;
float x = 2.3, y = 4.5;
i = x + y;
printf("%d", i); /* stampa 6 */
```

- Se la conversione non è possibile si ha errore.

Conversioni esplicite (operatore di cast)

Sintassi: `(tipo) espressione`

- Converte il valore di `espressione` nel corrispondente valore del `tipo` specificato.

  Esempio:

```
int somma, n;
float media;
...
media = somma / n;          /* divisione tra interi */
media = (float)somma / n;   /* divisione tra reali */
```

- L'operatore di cast "`(tipo)`" ha precedenza più alta degli operatori binari e associa da destra a sinistra. Dunque
  `(float) somma / n`
  equivale a
  `((float) somma) / n`

Input/output

- Come già detto, input e output non sono parte integrante del C
- L'interazione con l'ambiente è demandato alla libreria standard
  $\Longrightarrow$ un insieme di funzioni a uso dei programmi C
- La libreria `stdio.h` implementa un semplice modello di ingresso e uscita di dati testuali
- un testo è trattato come un successione (stream) di caratteri, ovvero
  $\Longrightarrow$ una sequenza di caratteri organizzata in righe, ciascuna terminata da "`\n`"
- al momento dell'esecuzione, al programma vengono connessi automaticamente 3 stream:
  - standard input: di solito la tastiera
  - standard output: di solito lo schermo
  - standard error: di solito lo schermo

Input/output (cont.)

- Compito della libreria è fare in modo che tutto il trattamento dei dati in ingresso e uscita si conformi a questo modello

  $\Longrightarrow$ il programmatore non si deve preoccupare di come ciò sia effettivamente realizzato
- Ogni volta che si effettua una operazione di lettura attraverso `getchar` viene acquisito il prossimo carattere dallo standard input e viene restituito il suo valore

  (analogamente per `scanf` che comporta l'acquisizione di uno o più caratteri a seconda delle specifiche di formato presenti ...)
- Ogni volta che si effettua una operazione di scrittura (attraverso `putchar` o `printf`) tutti i valori coinvolti vengono convertiti in sequenze di caratteri e queste ultime vengo accodate allo standard output.
- Tipicamente il sistema operativo consente di redirigere gli stream standard, ad esempio su uno o più file.

Funzioni `printf` e `scanf`

- Le funzioni `printf` e `scanf` sono le funzioni di libreria per l'output e per l'input. La lettera `f` alla fine dei nomi delle due funzioni sta per "formatted", ovvero "con formato"
- Sia `printf` che `scanf` ricevono una stringa di controllo, che può contenere le specifiche di conversione indicate con il simbolo % (segnaposto), e una serie di parametri, che possono essere ad esempio le variabili da stampare o leggere.

```
printf(stringa di controllo, lista di variabili)
scanf(stringa di controllo, lista di variabili)
```

- Per stampare delle variabili di un determinato tipo dobbiamo utilizzare i segnaposto relativi (`%d` per `int`,`%c` per `char`,...)
- NOTA: come per i comandi della shell, anche per le funzioni di libreria standard possiamo usare il comando `man`.

Formattazione dell'output con `printf`

- Riepilogo specificatori di formato principali:
  - interi: `%d`, `%o`, `%u`, `%x`, `%X`
    per `short`: si antepone `h`
    per `long`: si antepone `l` (minuscola)
  - reali: `%e`, `%f`, `%g`
    per `double`: non si antepone nulla
    per `long double`: si antepone `L`
  - caratteri: `%c`
  - stringhe: `%s` (le vedremo più avanti)
  - puntatori: `%p` (li vedremo più avanti)
- Flag: messi subito dopo il "`%`"
  - "`-`": allinea a sinistra
  - altri flag (non ci interessano)
- Sequenze di escape: `\%`, `\'`, `\"`, `\\`, `\a`, `\b`, `\n`, `\t`, ...
- Per stampare il carattere `'%'` è necessario raddoppiarlo: `%%`

Formattazione dell'input con `scanf`

- Specificatori di formato: come per l'output, tranne che per i reali
  - `double`: si antepone `l`
  - `long double`: si antepone `L`
- **Soppressione dell'input:** mettendo "`*`" subito dopo "`%`"
  Non ci deve essere un argomento corrispondente allo specificatore di formato: il campo deve essere letto, ma il valore non deve essere assegnato ad alcuna variabile.

  Esempio: Lettura di una data in formato gg/mm/aaaa oppure gg-mm-aaaa.

```
int g, m, a;
scanf("%d%*c%d%*c%d%*c", &g, &m, &a);
```

Espressioni booleane

- Il linguaggio deve consentire di descrivere espressioni booleane (guardie di condizionali e iterazione).
- Come già sappiamo, in C non esiste un tipo Booleano $\Longrightarrow$ si usa il tipo **int** :

  falso $\Longleftrightarrow$ 0

  vero $\Longleftrightarrow$ 1 (in realtà qualsiasi valore diverso da 0)

  Esempio: `2 > 3` ha valore 0 (ossia falso)

  `5 > 3` ha valore 1 (ossia vero)

- **Operatori relazionali del C**
  - `<`, `>`, `<=`, `>=` (minore, maggiore, minore o uguale, maggiore o uguale) — priorità alta
  - `==`, `!=` (uguale, diverso) — priorità bassa

  Esempio: `temperatura <= 0` `velocita > velocita_max`
  `voto == 30` `anno != 2000`

Operatori logici

- In ordine di priorità:
  - `!` (negazione) — priorità alta
  - `&&` (congiunzione)
  - `||` (disgiunzione) — priorità bassa

Semantica:

a	b	!a	a && b	a || b
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

0 ...falso

1 ...vero (o qualsiasi valore $\neq 0$)

Esempio:

`(a >= 10) && (a <= 20)` vero (`1`) se $10 \leq a \leq 20$

`(b <= -5) || (b >= 5)` vero se $|b| \geq 5$

- Le espressioni booleane vengono valutate da sinistra a destra:
  - con `&&`, appena uno degli operandi è falso, restituisce falso senza valutare il secondo operando
  - con `||`, appena uno degli operandi è vero, restituisce vero senza valutare il secondo operando
- **Priorità** tra operatori di diverso tipo:
  - not logico — priorità alta
  - aritmetici
  - relazionali
  - booleani (and e or logico) — priorità bassa

Esempio:

```
a+2 == 3*b || !trovato && c < a/3
```

è equivalente a

```
((a+2) == (3*b)) || ((!trovato) && (c < (a/3)))
```

Selezione doppia: istruzione **if-else**

Sintassi:

```
if     (espressione)
       istruzione1
else   istruzione2
```

- `espressione` è un'espressione booleana
- `istruzione1` rappresenta il ramo then (deve essere un'unica istruzione)
- `istruzione2` rappresenta il ramo else (deve essere un'unica istruzione)

Semantica:

1. viene prima valutata `espressione`
2. se `espressione` è vera viene eseguita `istruzione1` altrimenti (ovvero se `espressione` è falsa) viene eseguita `istruzione2`

```c
int temperatura;

printf("Quanti gradi ci sono? "); scanf("%d", &temperatura);
if (temperatura >= 25)
  printf("Fa caldo\n");
else
  printf("Si sta bene\n");

printf("Arrivederci\n");
```

```
=> Quanti gradi ci sono? 30 ↩
Fa caldo
Arrivederci
=>
```

```
=> Quanti gradi ci sono? 18 ↩
Si sta bene
Arrivederci
=>
```

Selezione singola: istruzione **if**

- È un'istruzione **if-else** in cui manca la parte else.

Sintassi:

```
if (espressione)
   istruzione
```

Semantica:

1. viene prima valutata `espressione`
2. se `espressione` è vera viene eseguita `istruzione` altrimenti non si fa alcunché

Esempio:

```
int temperatura;
scanf("%d", &temperatura);
if (temperatura >= 25)
  printf("Fa caldo\n");
printf("Arrivederci\n");
```

```
=> 18 ↩
Arrivederci
```

```
=> 30 ↩
Fa caldo
Arrivederci
```

Blocco

- La sintassi di **if-else** consente di avere un'unica istruzione nel ramo then (o nel ramo else).
- Se in un ramo vogliamo eseguire più istruzioni dobbiamo usare un blocco.

  Sintassi:

```
{
    istruzione-1
    …
    istruzione-n
}
```

- Come già sappiamo e come rivedremo più avanti, un blocco può contenere anche dichiarazioni.

Esempio: Dati mese ed anno, calcolare mese ed anno del mese successivo.

```
int mese, anno, mesesucc, annosucc;

if (mese == 12)
{
  mesesucc = 1;
  annosucc = anno + 1;
}
else
{
  mesesucc = mese + 1;
  annosucc = anno;
}
```

Esempio: Dato un anno, controllare se si tratta di un anno bisestile. Ricordiamo che un anno è *bisestile* se lo si può dividere per 4, ma non per 100, ad eccezione degli anni divisibili per 400, che sono bisestili.

```
int anno;

if ((anno%4 == 0 && anno%100 ! = 0) || (anno%400 == 0))
printf(%d e' un anno bisestile \n, anno);
else
printf(%d e' un anno bisestile \n, anno);
```

Esempio: Dato un anno, controllare se si tratta di un anno bisestile. Ricordiamo che un anno è *bisestile* se lo si può dividere per 4, ma non per 100, ad eccezione degli anni divisibili per 400, che sono bisestili.

```
int anno;

if ((anno%4 == 0 && anno%100 ! = 0) || (anno%400 == 0))
printf(%d e' un anno bisestile \n, anno);
else
printf(%d e' un anno bisestile \n, anno);
```

Selezione multipla: If annidati (in cascata)

- Quando l'istruzione del ramo then o else è un'istruzione **if o if-else.**

  Esempio: Data una temperatura, stampare un messaggio secondo la seguente tabella:

temperatura t	messaggio
$30 < t$	Molto caldo
$20 < t \leq 30$	Caldo
$10 < t \leq 20$	Gradevole
$0 < t \leq 10$	Freddo
$t \leq 0$	Molto freddo

```
 if (temperatura > 30)
   printf("Molto caldo\n");
else if (temperatura > 20)
   printf("Caldo\n");
else if (temperatura > 10)
   printf("Gradevole\n");
else if (temperatura > 0)
   printf("Freddo\n");
else printf("Molto freddo\n");
```

Osservazioni:

- si tratta di un'unica istruzione **if-else**

```
if (temperatura > 30)
  printf("Molto caldo\n");
else ...
```

- non serve che la seconda condizione sia composta

```
if (temperatura > 30) printf("Molto caldo\n");
else /* il valore di temperatura e' <= 30 */
    if (temperatura > 20)
```

  Non c'è bisogno di una congiunzione del tipo

```
(t <= 30) && (t > 20)
```

  (analogamente per gli altri casi).

- **Attenzione**: il seguente codice

```
if (temperatura > 30) printf("Molto caldo\n");
if (temperatura > 20) printf("Caldo\n");
```

  ha ben altro significato (quale?)

Ambiguità dell'else

```
if (a >= 0) if (b >= 0) printf("b positivo");
else printf("???");
```

- `printf("???")` può essere la parte **else**
  - del primo **if** $\Longrightarrow$ `printf("a negativo");`
  - del secondo **if** $\Longrightarrow$ `printf("b negativo");`
- L'ambiguità sintattica si risolve considerando che un **else** fa sempre riferimento all'**if** più vicino, dunque

```
if (a > 0)
  if (b > 0)
    printf("b positivo");
  else
    printf("b negativo");
```

- Perché un **else** si riferisca ad un **if** precedente, bisogna inserire quest'ultimo in un blocco

```
if (a > 0)
 { if (b > 0) printf("b positivo"); }
else
  printf("a negativo");
```

Esercizio

Leggere un reale e stampare un messaggio secondo la seguente tabella:

gradi alcolici g	messaggio
$40 < g$	superalcolico
$20 < g \leq 40$	alcolico
$15 < g \leq 20$	vino liquoroso
$12 < g \leq 15$	vino forte
$10.5 < g \leq 12$	vino normale
$g \leq 10.5$	vino leggero

Esempio: Dati tre valori che rappresentano le lunghezze dei lati di un triangolo, stabilire se si tratti di un triangolo equilatero, isoscele o scaleno.

Algoritmo: `determina tipo di triangolo`

```
leggi i tre lati
confronta i lati a coppie, fin quando non
  hai raccolto una quantità di informazioni
  sufficiente a prendere la decisione
stampa il risultato
```

```
main()   {
float primo, secondo, terzo;

printf("Lunghezze lati triangolo ? ");
scanf("%f%f%f", &primo, &secondo, &terzo);

if (primo == secondo) {
  if (secondo == terzo)
    printf("Equilatero\n");
  else
    printf("Isoscele\n");
}
else {
  if (secondo == terzo)
    printf("Isoscele\n");
  else if (primo == terzo)
    printf("Isoscele\n");
  else
    printf("Scaleno\n");
}     }
```

Esercizio

Risolvere il problema del triangolo utilizzando il seguente algoritmo:

```
Algoritmo: determina tipo di triangolo con conteggio
leggi i tre lati
confronta i lati a coppie contando
  quante coppie sono uguali
if le coppie uguali sono 0
  è scaleno
  else if le coppie uguali sono 1
    è isoscele
    else è equilatero
```

Selezione a più vie: istruzione **switch**

- **Sintassi:**

```
switch (espressione) {
  case valore-1: istruzioni-1
                 break;
  ...
  case valore-n: istruzioni-n
                 break;
  default : istruzioni-default
}
```

Semantica:

1. viene valutata `espressione`
2. viene cercato il primo `i` per cui il valore di `espressione` è uguale a `valore-i`
3. se si trova tale `i`, allora vengono eseguite `istruzioni-i` altrimenti vengono eseguite `istruzioni-default`

Equivale a `if (espressione == valore-1) istruzioni-1 else if (espressione == valore-2) istruzione-2 else... istruzioni-default`

Esempio:

```
int giorno;
...
switch (giorno) {
  case 1: printf("Lunedi'\n");
          break;
  case 2: printf("Martedi'\n");
          break;
  case 3: printf("Mercoledi'\n");
          break;
  case 4: printf("Giovedi'\n");
          break;
  case 5: printf("Venerdi'\n");
          break;
  default : printf("Week end\n");
}
```

- Se abbiamo più valori a cui corrispondono le stesse istruzioni, possiamo raggrupparli come segue:

```
case valore-1: case valore-2:
                    istruzioni
                    break;
```

Esempio:

```
int giorno;
...
switch (giorno) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5: printf("Giorno lavorativo\n");
          break;
  case 6:
  case 7: printf("Week end\n");
  default : printf("Giorno non valido\n");
}
```

Osservazioni sull'istruzione **switch**

- L'`espressione` usata per la selezione può essere una qualsiasi espressione C che restituisce un valore intero.
- I valori specificati nei vari **case** devono invece essere costanti (o meglio valori noti a tempo di compilazione). In particolare, non possono essere espressioni in cui compaiono variabili.

  Esempio: Il seguente frammento di codice è sbagliato:

```
int a;
switch (a) {
  case a<0: printf("negativo\n");
            /* ERRORE: a<0 non è una costante*/
  case 0: printf("nullo\n");
  case a>0: printf("positivo\n");
            /* ERRORE: a>0 non è una costante*/
}
```

- In realtà il C non richiede che nei **case** di un'istruzione **switch** l'ultima istruzione sia **break**.

  Quindi, in generale la **sintassi** di un'istruzione **switch** è:

```
switch (espressione) {
  case valore-1: istruzioni-1
  ...
  case valore-n: istruzioni-n
  default : istruzioni-default
}
```

Semantica:

1. viene prima valutata `espressione`
2. viene cercato il primo `i` per cui il valore di `espressione` è pari a `valore-i`
3. se si trova tale `i`, allora si eseguono in sequenza `istruzioni-i`, `istruzioni-(i+1)`, ..., fino a quando non si incontra **break** o è terminata l'istruzione **switch**,
   altrimenti vengono eseguite `istruzioni-default`

Esempio: più **case** di uno **switch** eseguiti in sequenza (corretto)

```
int lati;
printf("Immetti il massimo numero di lati del poligono (al piu` 6): ");
scanf("%d", &lati);
printf("Poligoni con al piu` %d lati: ", lati);

switch (lati) {
  case 6: printf("esagono, ");
  case 5: printf("pentagono, ");
  case 4: printf("rettangolo, ");
  case 3: printf("triangolo\n");
          break;
  case 2: case 1: printf("nessuno\n");
                  break;
  default : printf("\nErrore: valore immesso > 6.\n");
}
```

- N.B. Quando si omettono i **break**, diventa rilevante l'ordine in cui vengono scritti i vari **case**. Questo può essere facile causa di errori.

  È buona norma mettere **break** come ultima istruzione di ogni **case**

Esempio: più **case** eseguiti in sequenza (scorretto)

```
int b;
printf("Immetti un numero tra 1 e 6: ");
scanf("%", &b);
switch (b) {
  case 1: case 2: case 3: case 5: printf("Numero primo\n");
  case 4: case 6:                  printf("Numero non primo\n");
  default :                         printf("Valore non valido!\n");
}
```

=> 3↩
Numero primo
Numero non primo
Valore non valido!
=>

=> 4 ↩
Numero non primo
Valore non valido!
=>

Istruzioni iterative

Esempio: Leggere 5 interi, calcolarne la somma e stamparli.

- Variante non accettabile: 5 variabili, 5 istruzioni di lettura, 5 ...

```
int i1, i2, i3, i4, i5;
scanf("%d", &i1):
...
scanf("%d", &i5);
printf("%d", i1 + i2 + i3 + i4 + i5);
```

- Variante migliore che utilizza solo 2 variabili:

```
int somma, i;
somma = 0;
scanf("%d", &i);
somma = somma + i;
...          /* per 5 volte */
scanf("%d", &i);
somma = somma + i;
printf("%d", somma);
```

$\Longrightarrow$ conviene però usare un'istruzione iterativa

Iterazione determinata e indeterminata

- Le istruzioni iterative permettono di ripetere determinate azioni più volte:
  - un numero di volte fissato $\Longrightarrow$ iterazione determinata
    Esempio:
    ```
    fai un giro del parco di corsa per 10 volte
    ```
  - finchè una condizione rimane vera $\Longrightarrow$ iterazione indeterminata
    Esempio:
    ```
    finché non sei sazio
    prendi una ciliegia dal piatto e mangiala
    ```

Istruzione **while**

Permette di realizzare l'iterazione in C.

Sintassi:

```
while (espressione)
    istruzione
```

- `espressione` è la guardia del ciclo
- `istruzione` è il corpo del ciclo (può essere un blocco)

Semantica:

1. viene valutata l'`espressione` (condizione di continuazione)
2. se è vera si esegue `istruzione` e si torna ad eseguire l'intero `while`
3. se è falsa si termina l'esecuzione del `while`

- Nota: se `espressione` è falsa all'inizio, il ciclo non fa nulla.

Iterazione determinata

Esempio: Stampa 100 asterischi.

- Si utilizza un contatore per contare il numero di asterischi stampati.

  Algoritmo: `stampa di 100 asterischi`
  `inizializza il contatore a 0`
  while `il contatore è minore di 100`
  `{ stampa un ``*''`
  `incrementa il contatore di 1 }`

- Implementazione:

```
int i;
i = 0;
while (i < 100) {
  putchar('*');
  i = i + 1;
}
```

- come già sappiamo, la variabile `i` viene detta variabile di controllo del ciclo.

Iterazione determinata

Esempio: Leggere 10 interi, calcolarne la somma e stamparla.

- Si utilizza un contatore per contare il numero di interi letti.

```
int conta, dato, somma;
printf("Immetti 10 interi: ");
somma = 0;
conta = 0;
while (conta < 10) {
  scanf("%d", &dato);
  somma = somma + dato;
  conta = conta + 1;
}
printf("La somma è %d\n", somma);
```

Esempio: Leggere un intero N seguito da N interi e calcolare la somma di questi ultimi.

- Simile al precedente: il numero di ripetizioni necessarie non è noto al momento della scrittura del programma ma lo è al momento dell'esecuzione del ciclo.

```
int lung, conta, dato, somma;
printf("Immetti la lunghezza della sequenza ");
printf("seguita dagli elementi della stessa: ");
scanf("%d", &lung);
somma = 0;
conta = 0;
while (conta < lung) {
  scanf("%d", &dato);
  somma = somma + dato;
  conta = conta + 1;
}
printf("La somma è%d\n", somma);
```

Esempio: Leggere 10 interi positivi e stamparne il massimo.

- Si utilizza un massimo corrente con il quale si confronta ciascun numero letto.

```
int conta, dato, massimo;
printf("Immetti 10 interi: ");
massimo = 0;
conta = 0;
while (conta < 10) {
  scanf("%d", &dato);
  if (dato > massimo)
    massimo = dato;
  conta = conta + 1;
}
printf("Il massimo è %d\n", massimo);
```

Esercizio

Leggere 10 interi arbitrari e stamparne il massimo.

Istruzione **for**

- I cicli visti fino ad ora hanno queste caratteristiche comuni:
  - utilizzano una variabile di controllo
  - la guardia verifica se la variabile di controllo ha raggiunto un limite prefissato
  - ad ogni iterazione si esegue un'azione
  - al termine di ogni iterazione viene incrementato (decrementato) il valore della variabile di controllo

  Esempio: Stampare i numeri pari da 0 a N.

```
i = 0; /* Inizializzazione della var. di controllo */
while (i <= N) { /* guardia */
  printf("%d ", i); /* Azione da ripetere */
  i=i+2; /* Incremento var. di controllo */
}
```

- L'istruzione **for** permette di gestire direttamente questi aspetti:

```
for (i = 0; i <= N; i=i+2)
  printf("%d", i);
```

Sintassi:

```
for (istr-1; espr-2; istr-3)
  istruzione
```

- `istr-1` serve a inizializzare la variabile di controllo
- `espr-2` è la verifica di fine ciclo
- `istr-3` serve ad aggiornare la variabile di controllo alla fine del corpo del ciclo
- `istruzione` è il corpo del ciclo

Semantica: l'istruzione **for** precedente è equivalente a

```
istr-1;
while (espr-2) {
  istruzione
  istr-3
}
```

Esempio:

```
for (i = 1; i <= 10; i=i+1)      ⟹   i: 1, 2, 3, ..., 10
for (i = 10; i >= 1; i=i-1)      ⟹   i: 10, 9, 8, ..., 2, 1
for (i = -4; i <= 4; i = i+2)    ⟹   i: -4, -2, 0, 2, 4
for (i = 0; i >= -10; i = i-3)   ⟹   i: 0, -3, -6, -9
```

- In realtà, la sintassi del **for** è

  for `(espr-1; espr-2; espr-3)`
  `istruzione`

  dove `espr-1`, `espr-2` e `espr-3` sono delle espressioni qualsiasi (in C anche l'assegnamento è un'espressione ...).
- È buona prassi:
  - usare ciascuna `espr-i` in base al significato descritto prima
  - non modificare la variabile di controllo nel corpo del ciclo
- Ciascuna delle tre `espr-i` può anche mancare:
  - i ";" vanno messi lo stesso
  - se manca `espr-2` viene assunto il valore vero
- Se manca una delle tre `espr-i` è meglio usare un'istruzione **while**

Esempio: Leggere 10 interi positivi e stamparne il massimo.

```
int conta, dato, massimo;
printf("Immetti 10 interi: ");
massimo = 0;
for (conta=0; conta<10; conta=conta+1)
{
  scanf("%d", &dato);
  if (dato > massimo)
    massimo = dato;
}
printf("Il massimo è %d\n", massimo);
```

Cicli annidati

- Il corpo di un ciclo può contenere a sua volta un ciclo.

  Esempio: Stampa della Tavola Pitagorica

  Algoritmo

  ```
  for ogni riga tra 1 e 10
    { for ogni colonna tra 1 e 10
      stampa riga * colonna
    stampa un a capo }
  ```

- Traduzione in C

```
int riga, colonna;
const int Nmax = 10; /* indica il numero di righe e di colonne
*/
for (riga = 1; riga <= Nmax; riga=riga+1) {
  for (colonna = 1; colonna <= Nmax; colonna=colonna+1)
    printf("%d ", riga * colonna);
  putchar('\n'); }
```

Iterazione indefinita

- In alcuni casi il numero di iterazioni da effettuare non è noto prima di iniziare il ciclo, perché dipende dal verificarsi di una condizione.

  Esempio: Leggere una sequenza di interi che termina con 0 e calcolarne la somma.

  Input: $n_1, \ldots, n_k, 0$ (con $n_i \neq 0$)

  Output: $\sum_{i=1}^{k} n_i$

```
int dato, somma = 0;
scanf("%d", &dato);
while (dato != 0) {
  somma = somma + dato;
  scanf("%d", &dato);
}
printf("%d", somma);
```

Istruzione **do-while**

- Nell'istruzione **while** la condizione viene controllata all'inizio di ogni iterazione.
- L'istruzione **do-while** è simile all'istruzione **while**, ma la condizione viene controllata alla fine di ogni iterazione

Sintassi:

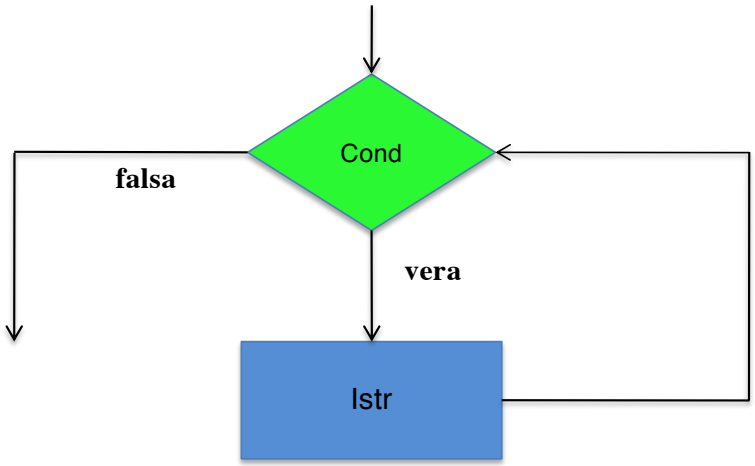
```
do
  istruzione
while (espressione);
```

Semantica: è equivalente a

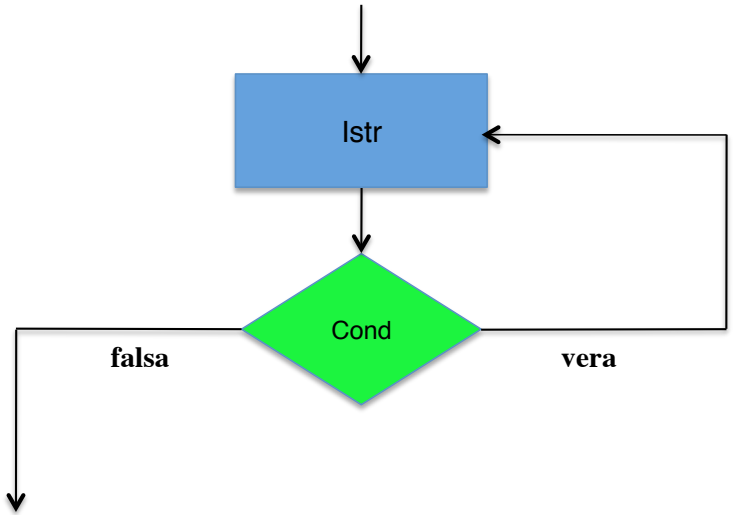
```
istruzione
while (espressione)
  istruzione
```

$\Longrightarrow$ una iterazione viene eseguita comunque. Ha quindi senso usare **do-while** quando sappiamo che deve essere eseguita **almeno una** iterazione.

while cond istr



do istr **while** cond



Esempio: Lunghezza di una sequenza di interi terminata da 0, usando **do-while**.

```
main() {
int lunghezza = 0; /* lunghezza della sequenza */
int dato; /* dato letto di volta in volta */
printf("Inserisci una sequenza di interi (0 fine seq.)\n");
do {
  scanf("%d", &dato);
  lunghezza=lunghezza+1;
} while (dato != 0);
printf("La sequenza è lunga %d\n", lunghezza - 1);
}
```

- Nota: lo 0 finale non è conteggiato (non fa parte della sequenza, fa da terminatore)

Assegnamento e altri operatori

- In C, l'operazione di assegnamento `x = exp` è un'espressione
  - il valore dell'espressione è il valore di `exp` (che è a sua volta un'espressione)
  - la valutazione dell'espressione `x = exp` ha un side-effect: quello di assegnare alla variabile `x` il valore di `exp`
- Dunque in realtà, "=" è un operatore (associativo a destra).
  Esempio: Qual è l'effetto di `x = y = 4` ?
  - È equivalente a: `x = (y = 4)`
  - `y = 4` ... espressione di valore `4` con modifica (side-effect) di `y`
  - `x = (y = 4)` ... espressione di valore `4` con ulteriore modifica su `x`
- L'eccessivo uso di assegnamenti come espressioni rende il codice difficile da comprendere e quindi anche da correggere/modificare.

Operatori di incremento e decremento

- Assegnamenti del tipo: `i = i + 1` / `i = i - 1` sono molto comuni.
  - operatore di incremento: `++`
  - operatore di decremento: `--`
- In realtà `++` corrisponde a due operatori:
- postincremento: `i++`
  - il valore dell'espressione è il valore di `i`
  - side-effect: incrementa `i` di `1`
- L'effetto di

```
int i,j;
i=6;
j=i++;
```

è j=6, i=7.

- preincremento: `++i`
  - il valore dell'espressione è il valore di `i+1`
  - side-effect: incrementa `i` di `1`
- L'effetto di

```
int i,j;
i=6;
j=++i;
```

è j=7, i=7.

(analogamente per `i--` e `--i`)

- Nota sull'uso degli operatori di incremento e decremento

Esempio:

	Istruzione	x	y	z
1	`int x, y, z;`	?	?	?
2	`x = 4;`	4	?	?
3	`y = 2;`	4	2	?
4a	`z = (x + 1) + y;`	4	2	7
4b	`z = (x++) + y;`	5	2	6
4c	`z = (++x) + y;`	5	2	7

- N.B.: Non usare mai in questo modo!
  In un'istruzione di assegnamento non ci devono essere altri side-effect (oltre a quello dell'operatore di assegnamento) !!!
- Riscrivere, ad esempio, come segue:

4b: `z = (x++) + y;` $\Longrightarrow$

```
z = x + y;
x++;
```

4c: `z = (++x) + y;` $\Longrightarrow$

```
x++;
z = x + y;
```

Ordine di valutazione degli operandi

- In generale il C non stabilisce quale è l'ordine di valutazione degli operandi nelle espressioni. Dipende quindi dal compilatore.

  Esempio: `int x, y, z;`

```
x = 2;
y = 4;
z = x++ + (x * y);
```

- Quale è il valore di `z`?
  - se viene valutato prima `x++`: $2 + (3 * 4) = 14$
  - se viene valutato prima `x*y`: $(2 * 4) + 2 = 10$

Forme abbreviate dell'assegnamento

```
a = a + b;   ⟹   a += b;
a = a - b;   ⟹   a -= b;
a = a * b;   ⟹   a *= b;
a = a / b;   ⟹   a /= b;
a = a % b;   ⟹   a %= b;
```

Espressioni condizionali

Al posto di

```
int x, y, z;
if (x > y)
z = x;
else
z = y;
```

possiamo scrivere, parafrasando quanto scritto, come:

```
int x, y, z;
z = (x > y)? x : y;
```

Abbiamo usato quelle che vengono definite espressioni condizionali, la cui sintassi è:

```
(espr-1)? espr-2 : espr-3
```

A seconda dell'esito della valutazione della prima espressione, si valuta la seconda oppure la terza.

Sequenze: nomi simbolici con indice

Consentiamo anche l'uso di nomi simbolici con indice, per rappresentare sequenze come quelle degli array.

$$\mathrm{c[i]} \rightsquigarrow 5$$

dove i è un valore naturale. Ad esempio:

$$\{\mathrm{i} \rightsquigarrow \mathrm{K},\ \mathrm{c[0]} \rightsquigarrow 0,\ \ldots,\ \mathrm{c}[K-1] \rightsquigarrow 0\} \qquad \mathrm{K} \geq 0$$

Nello stato sono presenti:

- un'associazione per il nome simbolico i, al quale è associato un valore naturale K.
- K associazioni per i nomi simbolici c[0], c[1], ..., c[K-1], a ciascuno dei quali è associato il valore 0.

All'interno degli algoritmi, consentiamo di riferire nomi simbolici con indice nella forma c[*exp*] dove *exp* deve essere una espressione a valori naturali. Anche in questo caso *exp* non deve contenere valori generici (nell'esempio, non è lecito un assegnamento del tipo c[K-1] = 5, mentre è lecito c[i-1] = 5).

Esempio: Calcolare il massimo di una sequenza data di interi.

Stato iniziale: $\{\text{dim} \rightsquigarrow \text{K}, \text{c[0]} \rightsquigarrow \text{V}_0, \ldots, \text{c[K-1]} \rightsquigarrow \text{V}_{\text{K-1}}\}$ con $\text{K} > 0$

Stato finale: $\{\text{massimo} \rightsquigarrow \max(\{\text{V}_0, \ldots, \text{V}_{\text{K-1}}\})\}$

- Dobbiamo calcolare

$$max\{\text{c[j]} \mid \text{j} \in [0, \text{dim-1}]\}$$

- Di nuovo, utilizziamo un ciclo controllato da una variabile di controllo i che assume tutti i valori nell'intervallo [0,dim-1].
- Facciamo in modo che, ad ogni iterazione, valga la seguente proprietà

$$\text{massimo} = max\{\text{c[j]} \mid \text{j} \in [0, \text{i-1}]\}$$

Per ottenere questo ad ogni iterazione confrontiamo l'elemento corrente con il "massimo in carica".

- Se, alla fine del ciclo, i=dim, abbiamo quanto desiderato e cioè

$$\text{massimo} = max\{\text{c[j]} \mid \text{j} \in [0, \text{dim-1}]\}$$

Soluzione:

```
massimo = c[0];
i = 1;
while (i <= dim-1)
{           qui massimo = max{c[0], …, c[i-1]}
    if (c[i] > massimo)
        massimo = c[i];
    i = i + 1;  anche qui massimo = max{c[0], …, c[i-1]}
}
```

- Inizializzare la variabile massimo a 0 sarebbe sbagliato: non è detto che tutti i numeri della sequenza siano negativi.
- Attenzione: l'istruzione i = i + 1 viene eseguita dopo l'istruzione if, indipendentemente dalla valutazione della condizione. Comunque dobbiamo scorrere tutta la sequenza.
- Per calcolare il massimo di K elementi devo fare K-1 confronti, dato che sono proprio K-1 gli elementi che devono uscire perdenti.

Esempio: Calcolare il numero di elementi pari in una sequenza data.

Stato iniziale: $\{\text{dim} \rightsquigarrow K,\ c[0] \rightsquigarrow V_0,\ \ldots,\ c[K\text{-}1] \rightsquigarrow V_{K\text{-}1}\}$ con $K > 0$

Stato finale: $\{\text{count} \rightsquigarrow \#\{\ j \mid j \in [0,K\text{-}1] \wedge\ V_j \text{ pari}\}\ \}$

- Dobbiamo calcolare

$$\sum_{\substack{0 \leq j \leq \text{dim-}1 \\ c[j] \text{ pari}}} 1$$

- Di nuovo, utilizziamo un ciclo controllato da una variabile di controllo i che assume tutti i valori nell'intervallo [0,dim-1].
- Facciamo in modo che, ad ogni iterazione, valga la seguente proprietà

$$\text{count} = \sum_{\substack{0 \leq j \leq \text{i-}1 \\ c[j] \text{ pari}}} 1$$

- Se, alla fine del ciclo, i=dim, abbiamo quanto desiderato e cioè

$$\text{count} = \sum_{\substack{0 \leq j \leq \text{dim-}1 \\ c[j] \text{ pari}}} 1$$

Soluzione:

```
count =  0;
i =  0;                          punto 1
while (i <= dim-1)
{
    if (c[i] % 2 == 0)
        count =  count + 1;
    i =  i + 1;
}
```

- Attenzione: l'istruzione i = i + 1 viene eseguita dopo l'istruzione if, indipendentemente dalla valutazione della condizione.
- Nello stato iniziale del while, al punto (1), count $\rightsquigarrow$ 0, i $\rightsquigarrow$ 0 e dunque

$$\sum_{\substack{0\leq j\leq \text{i-1}\\ c[j] \text{ pari}}} 1 = \sum_{\substack{1\leq j\leq 0\\ c[j] \text{ pari}}} 1 = 0 = \text{count}$$

- il corpo del while mantiene vera la proprietà

$$\text{count} = \sum_{\substack{0\leq j\leq \text{i-1}\\ c[j] \text{ pari}}} 1$$

Considerazioni generali

- In molti problemi è necessario operare allo stesso modo su tutti gli elementi di un intervallo dato

  per ogni $j \in [a,b]$ $\quad OP_j$

  esempio: $\displaystyle\sum_{j=a}^{b} exp_j$

- in tutti questi casi si ricorre a cicli in cui una variabile di controllo permette di scandire tutto l'intervallo di interesse

```
i = a;
while (i <= b)
{
    ⟨operazione in funzione di i⟩
    i = i+1;
}
```

Esercizio Controllare se un valore dato appare in una sequenza data

Stato iniziale: $\{\text{dim} \rightsquigarrow \text{K}, \text{val} \rightsquigarrow \text{V}, \text{c[0]} \rightsquigarrow \text{V}_0, \ldots, \text{c[K-1]} \rightsquigarrow \text{V}_{\text{K-1}}\}$

Stato finale: $\{\text{trovato} \rightsquigarrow \text{b} = 1 \text{ se } \exists\ j \in [0,\text{K-1}] \wedge \text{V}_\text{j} = \text{V}, \text{b} = 0 \text{ altr.}\}$

Breve introduzione alla Semantica

- Per fornire la semantica formale a un linguaggio di programmazione serve un modello matematico, come base per comprendere e ragionare su come si comportano i programmi.
- Utile per vari tipi di analisi e verifiche, ma anche perché definire con precisione il significato delle costruzioni di programmi ci rende consapevoli di molti tipi di sottigliezze.
- Storicamente esistono tre approcci, non in opposizione tra loro: quelli della semantica operazionale, della semantica denotazionale e della semantica assiomatica.

Breve introduzione alla Semantica

- La **semantica operazionale** descrive il significato di un linguaggio di programmazione specificando come viene eseguito su una macchina astratta [Gordon, Plotkin]. Utile per l'implementazione. Approccio più concreto.
- La **semantica denotazionale** è una tecnica per definire il significato dei linguaggi di programmazione che utilizza i concetti matematici più astratti di ordini parziali completi, funzioni continue e punti fissi [Christopher Strachey, Dana Scott]. Approccio più astratto.
- La **semantica assiomatica** cerca di fissare il significato di un costrutto di programmazione fornendo regole di dimostrazione all'interno di una logica di programma. [R.W.Floyd, C.A.R.Hoare] Approccio adatto alla verifica di proprietà.

Breve introduzione alla Semantica Operazionale*

Quello che abbiamo visto può essere visto come un piccolo linguaggio **imperativo**, come quello che Winskel chiama **IMP**.

- Il comportamento di **IMP** viene descritto formalmente da un insieme di regole che specificano come valutare le espressioni e come eseguire i comandi.
- Le regole danno una *semantica operazionale* al linguaggio, operazionale proprio perché vicina all'implementazione del linguaggio. Inoltre offrono la base per semplici dimostrazioni di equivalenze tra comandi.
- La semantica operazionale, che fornisce un modello matematico, l'**astrazione**, dell'esecuzione di un interprete, viene usata per scrivere compilatori e interpreti ci descrive l'effetto di ogni comando sullo stato.

* Dal Capitolo 2 di "The Formal Semantics of Programming Languages: An Introduction" di Glynn Winskel. - Edizione italiana: "La semantica formale dei linguaggi di programmazione" di G. Winskel, F. Turini, P. Baldan e A. Bracciali

Categorie sintattiche

Cominciamo con l'introduzione delle categorie sintattiche del nostro linguaggio:

- numeri $\mathbf{N} \ni n$
- valori di verità $\mathbf{T} = \{\mathbf{true}, \mathbf{false}\}$
- locazioni $\mathbf{Loc} \ni X$ (*variabili* che possono essere modificate)
- espressioni aritmetiche $\mathbf{Aexp} \ni a$
- espressioni booleane $\mathbf{Bexp} \ni b$
- istruzioni o comandi $\mathbf{Com} \ni c$
- L'insieme degli *stati* Σ è dato dalle funzioni $\sigma : \mathbf{Loc} \to \mathbf{N}$, t.c. $\sigma(X)$ rappresenta il valore o il contenuto della locazione X nello stato σ.
- Adesso vedremo come costruire gli elementi di questi insiemi.

Categorie sintattiche, cont.

Per quanto riguarda la definizione delle ultime tre categorie sintattiche (**Aexp**,**Bexp**, e **Com**) daremo una *definizione induttiva*, ricorrendo ad apposite regole di formazione, con le quali esprimeremo cose del tipo seguente.

- Se a_0 e a_1 sono espressioni aritmetiche, lo è anche l'espressione composta $a_0 + a_1$.
- Analogamente, se b_0 e b_1 sono espressioni booleane, lo è anche l'espressione composta $b_0 \wedge b_1$.
- Analogamente, se c_0 e c_1 sono comandi, lo è anche il comando composto $c_0; c_1$.
- I simboli a_i, b_i e c_i sono usati per rappresentare una qualsiasi espressione aritmetica (espressione booleana, o comando). Più precisamente, si tratta di *metavariabili* che variano all'interno degli insiemi **Aexp**,**Bexp**, e **Com**.
- Diamo ora le regole di formazione delle espressioni (usando una notazione il più vicina possibile a quella già vista per il C)

Espressioni Aritmetiche

$$a ::= n | X | a_0 + a_1 | a_0 - a_1 | a_0 \times a_1$$

dove " :: =" deve essere letto come "può essere" e il simbolo "|" va inteso come "oppure".

- La regola $a ::= n$ ci dice che il numero $n \in \mathbf{N}$ è considerato un'espressione aritmetica (elementare).
- La regola $a ::= X$ ci dice che la variabile $X \in \mathbf{Loc}$ è considerata un'espressione aritmetica (elementare).
- La regola $a ::= a_0 + a_1$ ci dice che se a_0 e a_1 sono espressioni aritmetiche, allora lo è anche $a_0 + a_1$, dove le a_i sono meta-variabili. Ad es. se $a_0 = X$ and $a_1 = 5$ allora $X + 5$ è ancora un'espressione aritmetica.
- Le altre regole vanno interpretate in modo analogo.
- Per indicare che due espressioni sono equivalenti usiamo il simbolo $\equiv$.

Espressioni Booleane

$$b ::= \textbf{true} | \textbf{false} | a_0 == a_1 | a_0 <= a_1 | \neg b | b_0 \vee b_1 | b_0 \wedge b_1$$

- La regola $b ::= \textbf{true}$ ci dice che la costante **true** è considerata un'espressione booleana (elementare), analogamente per **false**.
- La regola $b ::= a_0 == a_1$ ci dice che se a_0 e a_1 sono espressioni aritmetiche, allora $a_0 == a_1$ è un'espressione booleana, analogamente per $a_0 <= a_1$.
- La regola $b ::= \neg b$ ci dice che se b è un'espressione booleana, lo è anche $\neg b$.
- La regola $b ::= b_0 \vee b_1$ ci dice che se b_0 e b_1 sono espressioni booleane, allora lo è anche $b_0 \vee b_1$
- Le altre regole vanno interpretate in modo analogo.
- Per indicare che due espressioni sono equivalenti usiamo il simbolo $\equiv$.

Comandi

$$c ::= \textbf{skip} | X = a | c_0; c_1 | \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 | \textbf{while } b \textbf{ do } c$$

- La regola $c ::= \textbf{skip}$ ci dice che **skip** è considerato un comando (elementare).
- La regola $c ::= X = a$ ci dice che se X è una variabile, ed a è un'espressione aritmetica, allora l'assegnamento $X = a$ è un comando.
- La regola $c ::= c_0; c_1$ ci dice che se c_0 e c_1 sono comandi, lo è anche il comando composto $c_0; c_1$.
- La regola $c ::= \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1$ ci dice che se b è un'espressione booleana e c_0 e c_1 sono comandi, lo è anche il comando composto **if** b **then** c_0 **else** c_1.
- La regola $c ::= \textbf{while } b \textbf{ do } c$ ci dice che se b è un'espressione booleana e c è un comando, lo è anche il comando composto **while** b **do** c.
- Per indicare che due espressioni sono equivalenti usiamo il simbolo $\equiv$.

Valutazione delle Espressioni Aritmetiche

- Il significato di un'espressione **dipende** dal valore delle variabili, ovvero da quello che chiamiamo **stato**
- L'insieme degli stati Σ consiste nelle funzioni $\sigma : \mathbf{Loc} \to \mathbf{N}$ da locazioni a numeri, per cui $\sigma(X)$ rappresenta il valore associato a X, ovvero il contenuto della locazione X nello stato σ (nella nostra notazione corrisponde all'associazione $X \rightsquigarrow \sigma(X)$)
- La valutazione di un'espressione viene fatta dunque in un certo stato σ: abbiamo cioè *configurazioni* (coppie) della forma $\langle a, \sigma \rangle$ in attesa di essere valutate, arrivando al valore n.

$$\langle a, \sigma \rangle \to n$$

  Vedremo un insieme di regole da applicare per riscrivere le configurazioni fino a quando non raggiungiamo il valore finale n.
- La semantica operazionale si dice **strutturata**, quando usa la sintassi per guidare il processo di valutazione.

Regole di inferenza logica

- Regola di inferenza $\frac{\mathbf{A_1}, ..., \mathbf{A_n}}{\mathbf{B}}$: se sono vere le affermazioni sopra la riga (premesse), allora è vera l'affermazione sotto la riga (conclusione). Equivale a $A_1 \wedge ... \wedge A_n \Rightarrow B$
- $$\frac{\text{piove} \ \wedge \ \text{non ho l'ombrello}}{\text{mi bagno}}$$
- Un'inferenza è logicamente valida (le premesse implicano logicamente la conclusione) quando è impossibile che la conclusione sia falsa se le premesse sono vere. Altrimenti non è valida.
- $A \Rightarrow B$ è sempre vero, tranne nel caso in cui A è vero e B è falso.
- $A \Rightarrow B$ equivale a $\neg A \vee B$.

Regole di inferenza logica (cont.)

- Regola di inferenza $\frac{\mathbf{A_1}, ..., \mathbf{A_n}}{\mathbf{B}}$: se sono vere le affermazioni sopra la riga (premesse), allora è vera l'affermazione sotto la riga (conclusione). Equivale a $A_1 \wedge ... \wedge A_n \Rightarrow B$
- Assioma $\frac{}{\mathbf{B}}$: regola di inferenza senza premesse, B quindi la conclusione è sempre vera
- Una derivazione di B prende la forma di un albero (detto di dimostrazione o derivazione) che può essere una istanza di un assioma

  $\frac{}{\mathbf{B}}$ o includere le derivazioni delle premesse di B:

  $\frac{\frac{\mathbf{D_1}, ..., \mathbf{D_n}}{\mathbf{A_1}}, ..., \frac{\mathbf{E_1}, ..., \mathbf{E_n}}{\mathbf{A_n}}}{\mathbf{B}}$: la radice è l'asserzione da dimostrare, le foglie sono assiomi e i nodi intermedi sono ottenuti applicando regole

Valutazione delle Espressioni Aritmetiche (cont.)

- La valutazione di un'espressione non richiede di modificare la memoria.
- $\langle n, \sigma \rangle \to n$ **assioma** (premessa vuota)
- $\langle X, \sigma \rangle \to \sigma(X)$
- $$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n_0 + n_1}$$ **premessa** / **conclusione**
- $$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 - a_1, \sigma \rangle \to n_0 - n_1}$$
- $$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 \times a_1, \sigma \rangle \to n_0 \times n_1}$$
- N.B. Gli operatori in nero ($+$,$-$, $\times$) rappresentano simboli della sintassi, mentre i corrispondenti operatori in blu l'effettiva implementazione dell'operazione. Ad es. $n_0 + n_1$ sta per la somma di n_0 e di n_1.

Valutazione delle Espressioni Aritmetiche (esempio)

- Per valutare l'espressione aritmetica $a = (X + 5)$ nello stato σ_0, nel quale $\sigma_0(X) = 0$ si deve istanziare la regola della somma, cioè sostituire i valori concreti su cui fare la valutazione alle metavariabili della regola, X ad a_0, 5 ad a_1 e σ_0 a σ.

$$\frac{\langle \underline{a_0}, \underline{\sigma} \rangle \to n_0 \quad \langle \underline{a_1}, \underline{\sigma} \rangle \to n_1}{\langle \underline{a_0} + \underline{a_1}, \underline{\sigma} \rangle \to n_0 + n_1} \qquad \frac{\langle X, \sigma_0 \rangle \to n_0 \quad \langle 5, \sigma_0 \rangle \to n_1}{\langle X + 5, \sigma_0 \rangle \to n_0 + n_1}$$

- A questo punto il problema ci si riconduce alla valutazione delle sotto-espressioni $X \in \mathbf{Loc}$ e $5 \in \mathbf{N}$. Essendo espressioni aritmetiche semplici, ognuna ha il suo assioma da istanziare:

$$\langle X, \underline{\sigma} \rangle \to \underline{\sigma}(X) \quad \langle \underline{n}, \underline{\sigma} \rangle \to \underline{n} \quad \langle X, \sigma_0 \rangle \to \sigma_0(X) \quad \langle 5, \sigma_0 \rangle \to 5$$

- Adesso posso completare l'albero di derivazione

$$\frac{\dfrac{}{\langle X, \sigma_0 \rangle \to 0} \qquad \dfrac{}{\langle 5, \sigma_0 \rangle \to 5}}{\langle (X + 5), \sigma_0 \rangle \to 5}$$

Valutazione delle Espressioni Aritmetiche (esempio)

Consideriamo adesso la valutazione dell'espressione aritmetica più complessa $a_0 = (X + 5) + (7 + 9)$ sempre nello stato σ_0, dove $\sigma_0(X) = 0$
Procedendo analogamente otteniamo il seguente albero di derivazioni.

$$\dfrac{\dfrac{\dfrac{}{\langle X, \sigma_0\rangle \to 0} \quad \dfrac{}{\langle 5, \sigma_0\rangle \to 5}}{\langle (X + 5), \sigma_0\rangle \to 5} \quad \dfrac{\dfrac{}{\langle 7, \sigma_0\rangle \to 7} \quad \dfrac{}{\langle 9, \sigma_0\rangle \to 9}}{\langle (7 + 9), \sigma_0\rangle \to 16}}{\langle (X + 5) + (7 + 9), \sigma_0\rangle \to 21}$$

Valutazione delle Espressioni Booleane

- $\langle \mathbf{true}, \sigma \rangle \to \mathbf{true} \qquad \langle \mathbf{false}, \sigma \rangle \to \mathbf{false}$
- $$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 == a_1, \sigma \rangle \to \mathbf{true}/\mathbf{false}} \quad \text{se } n_0 = n_1 / n_0 \neq n_1$$
- $$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 <= a_1, \sigma \rangle \to \mathbf{true}/\mathbf{false}} \quad \text{se } n_0 \leq n_1 / n_0 \not\leq n_1$$
- $$\frac{\langle b, \sigma \rangle \to \mathbf{true}}{\langle \neg b, \sigma \rangle \to \mathbf{false}} \qquad \frac{\langle b, \sigma \rangle \to \mathbf{false}}{\langle \neg b, \sigma \rangle \to \mathbf{true}}$$
- $$\frac{\langle b_0, \sigma \rangle \to t_0 \quad \langle b_1, \sigma \rangle \to t_1}{\langle b_0 \vee b_1, \sigma \rangle \to t} \quad \text{con } t \text{ a seconda dei valori } t_0 \text{ e } t_1$$
- $$\frac{\langle b_0, \sigma \rangle \to t_0 \quad \langle b_1, \sigma \rangle \to t_1}{\langle b_0 \wedge b_1, \sigma \rangle \to t} \quad \text{con } t \text{ a seconda dei valori } t_0 \text{ e } t_1$$

Valutazione "lazy" di $\wedge$

- In C, la valutazione della congiunzione *and* è chiamata *lazy* o pigra, dato che inizialmente si rimanda la valutazione della seconda espressione e poi la si fa solo se necessario.
- In particolare, prima si controlla se la prima espressione è falsa, nel qual caso si evita di valutare la seconda parte dell'espressione, dato che la valutazione complessiva sarà comunque falsa.
- Questo tipo di valutazione può essere reso con le regole della semantica operazionale seguenti:

$$\frac{\langle b_0, \sigma\rangle \to \textbf{false}}{\langle b_0 \wedge b_1, \sigma\rangle \to \textbf{false}} \qquad \frac{\langle b_0, \sigma\rangle \to \textbf{true} \quad \langle b_1, \sigma\rangle \to t}{\langle b_0 \wedge b_1, \sigma\rangle \to t}$$

Esecuzione dei comandi

- L'esecuzione di un comando porta ad un nuovo stato: $\langle c, \sigma \rangle \to \sigma'$, ammesso che l'esecuzione termini!
- Si suppone che lo *stato inziale* σ_0 abbia la proprietà che $\sigma_0(X) = 0$ per tutte le locazioni X.
- $\langle \textbf{skip}, \sigma \rangle \to \sigma$
- $\dfrac{\langle a, \sigma \rangle \to m}{\langle X = a, \sigma \rangle \to \sigma[m/X]}$ $\sigma[m/X]$ è lo stato dove m è associato a X^*
- $\dfrac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}$ prima va eseguito il primo comando
- $\dfrac{\langle b, \sigma \rangle \to \textbf{true}/\textbf{false} \quad \langle c_i, \sigma \rangle \to \sigma_i}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma \rangle \to \sigma_0/\sigma_1}$

$$^*\sigma[m/X](Y) = \begin{cases} m & \text{if } Y = X; \\ \sigma(Y) & \text{if } Y \neq X. \end{cases}$$

Modifiche dello stato

- L'insieme degli *stati* Σ è dato dalle funzioni σ : **Loc** $\rightarrow$ **N**, t.c. $\sigma(X)$ rappresenta il valore o il contenuto della locazione X nello stato σ.
- Quando si esegue un **assegnamento** $X = a$ nello stato σ si va a modificare lo stato limitatamente a X.
- Formalmente la modifica dello stato in solo punto si scrive: $\sigma[m/X]$, che indica che nello stato modificato il valore m sostituirà il valore precedente associato a X.
- I valori associati alle altre locazioni non cambiano: nel nuovo stato, otterrò il nuovo valore m se la locazione è X, i vecchi valori in tutti gli altri casi.

$$\sigma[m/X](Y) = \begin{cases} m & \text{if } Y = X; \\ \sigma(Y) & \text{if } Y \neq X. \end{cases}$$

- Se si esegue $X = 3$ in σ_0, ad es. si otterrà un nuovo stato σ_1 t.c. $\sigma_1(X) = 3$, mentre $\sigma_1(Y) = \sigma_0(Y) = 0$ per ogni $Y \neq X$.

Valutazione dei comandi (cont.)

- $$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma}$$
- $$\frac{\langle b, \sigma\rangle \to \textbf{true}\ \langle c; \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}$$

Completezza e Correttezza

- Vorremmo che la nostra semantica ci permettesse di dimostrare tutto ciò che è vero e niente di falso. La semantica è
  - **completa** se può dimostrare ogni *asserzione* vera. Ad es., con la regola seguente, *non* potrei dimostrare che $\langle 5-3, \sigma \rangle \to 2$ (vero).

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to 0}{\langle a_0 - a_1, \sigma \rangle \to n_0}$$

  - **corretta** se ogni *asserzione* dimostrabile è vera. Ad es., con la regola seguente, *potrei* dimostrare che $\langle 5-3, \sigma \rangle \to 7$ (falso)

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 - a_1, \sigma \rangle \to n_0 + 2}$$

Equivalenze

- Due espressioni aritmetiche a_0 e a_1 sono *equivalenti* $a_0 \sim a_1$ se e solo se

$$\forall n \in \mathbf{N}, \forall \sigma \in \Sigma. \langle a_0, \sigma \rangle \to n \Leftrightarrow \langle a_1, \sigma \rangle \to n$$

- Due espressioni booleane b_0 e b_1 sono *equivalenti* $b_0 \sim b_1$ se e solo se

$$\forall t, \forall \sigma \in \Sigma. \langle b_0, \sigma \rangle \to t \Leftrightarrow \langle b_1, \sigma \rangle \to t$$

- Due comandi c_0 e c_1 sono *equivalenti* $c_0 \sim c_1$ se e solo se

$$\forall \sigma, \sigma' \in \Sigma. \langle c_0, \sigma \rangle \to \sigma' \Leftrightarrow \langle c_1, \sigma \rangle \to \sigma'$$

Es. di dimostrazione sui comandi

Proposizione
Sia $w \equiv$ **while** b **do** c. Allora

$$w \sim \textbf{if } b \textbf{ then } c; w \textbf{ else skip}$$

Dimostrazione
Si deve dimostrare che per ogni scelta di σ, σ':

$$\langle w, \sigma \rangle \to \sigma' \text{ sse } \langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma \rangle \to \sigma'$$

Lo si fa usando le derivazioni delle regole

Dimostrazione $\Leftarrow$

Siano $\sigma, \sigma' \in \Sigma$ e si supponga che $\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma \rangle \to \sigma'$.
Allora deve esistere una derivazione con una delle seguenti forme

- $$\dfrac{\dfrac{\cdots}{\langle b, \sigma \rangle \to \textbf{false}} \quad \dfrac{}{\langle \textbf{skip}, \sigma \rangle \to \sigma}}{\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma \rangle \to \sigma}$$ $(1 \Leftarrow)$ (qui $\sigma' = \sigma$)
- $$\dfrac{\dfrac{\cdots}{\langle b, \sigma \rangle \to \textbf{true}} \quad \dfrac{}{\langle c; w, \sigma \rangle \to \sigma'}}{\langle \textbf{if } b \textbf{ then } c; w \textbf{ else skip}, \sigma \rangle \to \sigma'}$$ $(2 \Leftarrow)$

Da ognuna si può partire per costruire la derivazione $\langle w, \sigma \rangle \to \sigma'$.

Dimostrazione caso (2 $\Leftarrow$)

La seconda derivazione deve contenere, per qualche stato σ'', qualcosa del tipo

$$\dfrac{\dfrac{\cdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\cdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle c; w, \sigma\rangle \to \sigma'}$$

Combinando con il resto:

$$\dfrac{\dfrac{\cdots}{\langle b, \sigma\rangle \to \mathbf{true}} \quad \dfrac{\cdots}{\langle c, \sigma\rangle \to \sigma''} \quad \dfrac{\cdots}{\langle w, \sigma''\rangle \to \sigma'}}{\langle w, \sigma\rangle \to \sigma'}$$

Analogamente per il caso (1). Si può quindi concludere che

$$\langle \mathbf{if}\ b\ \mathbf{then}\ c; w\ \mathbf{else\ skip}, \sigma\rangle \to \sigma' \Rightarrow \langle w, \sigma\rangle \to \sigma'$$

Esercizio Controllare se un valore dato appare in una sequenza data

Stato iniziale: $\{\text{dim} \rightsquigarrow \text{K}, \text{val} \rightsquigarrow \text{V}, \text{c[0]} \rightsquigarrow \text{V}_0, \ldots, \text{c[K-1]} \rightsquigarrow \text{V}_{\text{K-1}}\}$

Stato finale: $\{\text{trovato} \rightsquigarrow \text{b} = 1 \text{ se } \exists\ j \in [0,\text{K-1}] \wedge \text{V}_j = \text{V}, \text{ b}= 0 \text{ altr.}\}$

- Di nuovo, utilizziamo un ciclo controllato da una variabile di controllo i che assume tutti i valori nell'intervallo [0,dim-1].
- Devo usare anche una variabile di controllo trovato che mi dica se l'elemento è stato trovato oppure no.
- Dal ciclo si esce quindi quando:
  - o ho appena trovato l'elemento cercato
  - o sono arrivato alla fine della sequenza.

```
i = 0; trovato = 0;
while ((i <= dim-1) && (trovato == 0))
{
    if (c[i] == val)
        trovato = 1;
    i = i + 1;
}
```

Caveat

- La doppia condizione ((i <= dim-1) && (trovato == 0)) corrisponde al fatto che si può uscire dal ciclo:
  - se sono arrivato alla fine della sequenza,
  - o se ho appena trovato l'elemento cercato.
- usare solo la prima non è scorretto **ma** risulta *inefficiente*. Ad es., in una sequenza di 10.000 elementi in cui il valore si trova nella prima posizione, non fermarsi appena lo si è trovato comporta che si facciano comunque tutti i 9999 confronti.
- uscire non appena si trova l'elemento cercato si può ottenere anche in altri modi, come ad esempio
  - forzando il valore del contatore i ad assumere il valore dim per falsificare la prima condizione

  In questi i casi tuttavia, viene pregiudicata la leggibilità del codice ed anche minata la prossibilità di fare analisi di correttezza. Per questo motivo, sono **fortemente sconsigliati**.

Esercizi proposti

Problema 1: Calcolare il numero di occorrenze di un valore dato in una sequenza data

Stato iniziale: $\{\text{dim} \rightsquigarrow \text{K}, \text{val} \rightsquigarrow \text{V}, \text{c[0]} \rightsquigarrow \text{V}_0, \ldots, \text{c[K-1]} \rightsquigarrow \text{V}_{\text{K-1}}\}\ \text{K} > 0$

Stato finale: $\{\text{occ} \rightsquigarrow \#\{\ \text{j} \mid \text{j} \in [0,\text{K-1}] \wedge \text{V}_\text{j} = \text{V}\}\ \}$

Problema 2: Calcolare contemporaneamente il massimo e il minimo di una sequenza data di interi

Stato iniziale: $\{\text{dim} \rightsquigarrow \text{K}, \text{c[0]} \rightsquigarrow \text{V}_0, \ldots, \text{c[K-1]} \rightsquigarrow \text{V}_{\text{K-1}}\}$ con $\text{K} > 0$

Stato finale: $\{\text{massimo} \rightsquigarrow \max(\{\text{V}_0,\ldots,\text{V}_{\text{K-1}}\}),$
$\text{minimo} \rightsquigarrow \min(\{\text{V}_0,\ldots,\text{V}_{\text{K-1}}\})\}$

Problema 3: Calcolare il numero di occorrenze del valore massimo in una sequenza di interi

Stato iniziale: ? Stato finale: ?

Problema 4: Calcolare la media di una sequenza data di interi

Stato iniziale: ? Stato finale: ?

Fondamenti di Programmazione: AUTOMI E LINGUAGGI FORMALI Corso di Laurea in MATEMATICA a.a. 2023/2023

Chiara Bodei, Roberta Gori, Damiano De Francesco Maesa

Dipartimento di Informatica
chiara.bodei@unipi.it

Testi di riferimento

- **Libro di testo principale**: *Automi, linguaggi e calcolabilità*, J. E. Hopcroft, R. Motwani, and J. D. Ullman, Addison-Wesley, 2003.
- **Altro libro**: *Compilers: principles, techniques, and tools*, A.H. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Pearson/Addison-Wesley, 2007 (seconda edizione).
- I lucidi su questa parte del corso sono basati su quelli della Prof. Francesca Rossi (basati a loro volta su quelli di Gösta Grahne e David Ford).

Perché studiare gli automi e i linguaggi formali?

- Le espressioni regolari sono usate in molti sistemi, ad esempio in UNIX: `rm hello*.c`
- Gli automi sono utilizzati per modellare protocolli e circuiti elettronici.
- Le grammatiche sono usate per descrivere la sintassi dei linguaggi di programmazione.

Cosa studieremo

- Linguaggi regolari
  - Loro descrizione tramite automi finiti deterministici e non deterministici, espressioni regolari.
  - Proprietà di decisione e di chiusura
- Linguaggi liberi
  - Loro descrizione tramite grammatiche libere
  - Proprietà di decisione e di chiusura

Inoltre impareremo a trattare in modo formale i sistemi discreti e a familiarizzare con i modelli astratti.

Qual è il pattern?

Un robot lancia in aria una moneta e occorre indovinare se viene testa o croce. Sembra esserci tuttavia una sequenza, un pattern, ripetuta nel modo in cui la moneta appare. È un gioco truccato? Dati i seguenti risultati relativi ai tiri della moneta (t=testa, c=croce):

ttcttctttccttttcctccctttttcttt
ccctttccctttttcctcccccctcctccc
tttcctttctttttttttccttttccccttt
ttcccccccc

qual è il pattern ricorrente dei risultati dei tiri?

Qual è il pattern?

ttc/ttc/ttt/cct/ttt/cct/ccc/ttt/ttc/ttt
ccc/ttt/ccc/ttt/ttt/cct/ccc/cct/cct/ccc
ttt/cct/ttc/ttt/ttt/ttt/cct/ttc/ccc/ttt
ttc/ccc/ccc

I primi due tiri ogni tre danno sempre lo stesso risultato

Automi a stati finiti

Gli automi a stati finiti sono usati come modello per

- Software per la progettazione di circuiti digitali.
- Analizzatori lessicali di un compilatore.
- Applicazioni per l'elaborazione di testi, ad esempio per la ricerca di parole chiave in un file o sul web.
- Software per verificare sistemi a stati finiti, come protocolli di comunicazione.

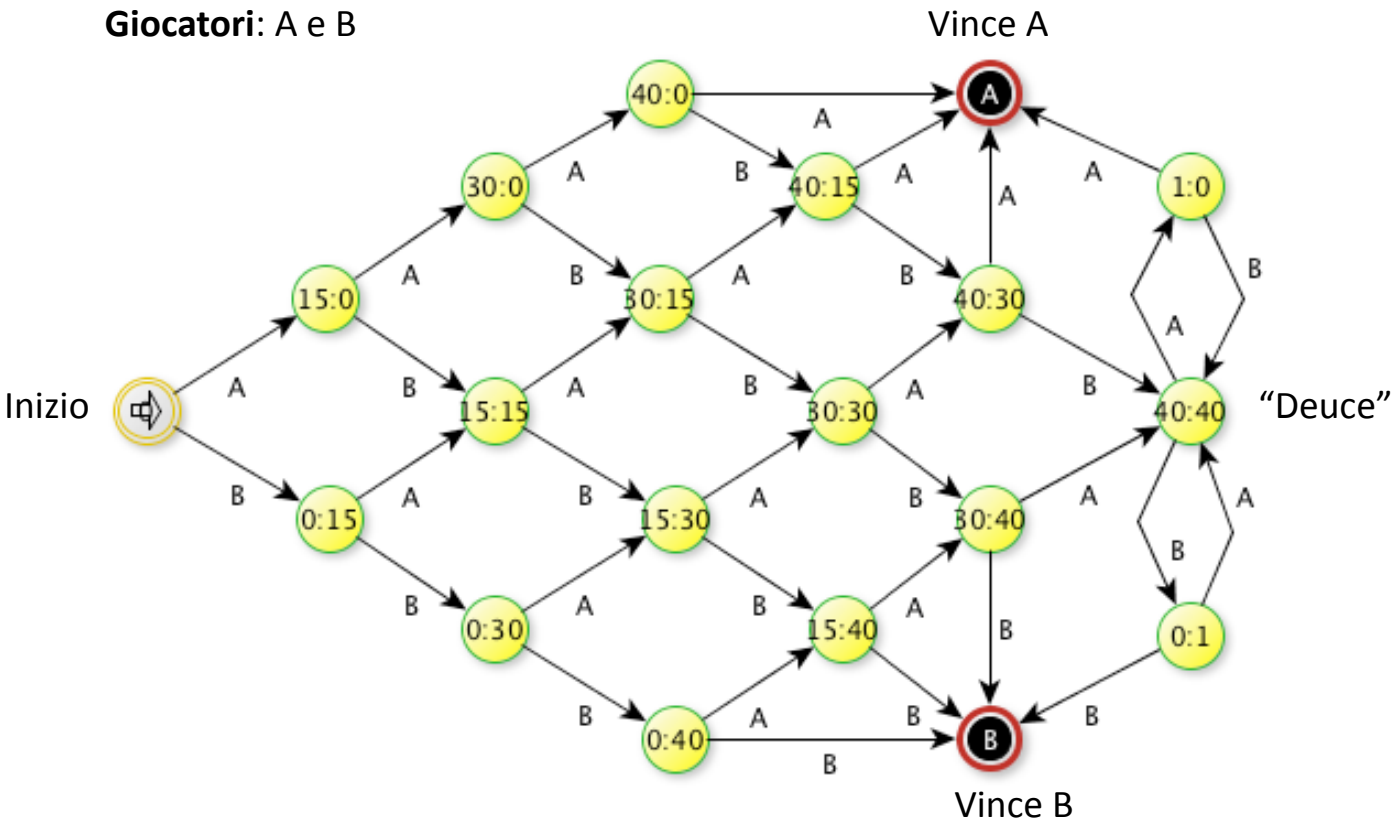
Automi a stati finiti (cont.)

Che cosa è un automa a stati finiti?

- È un sistema **formale**
- È un sistema in grado di ricordare solo un quantitativo finito di informazione.
- L'informazione viene rappresentata da stati.
- Gli stati cambiano in risposta agli input.
- È un sistema costituito da un insieme finito di stati e da regole che dicono come passare da uno stato all'altro, in risposta all'input.

Esempi

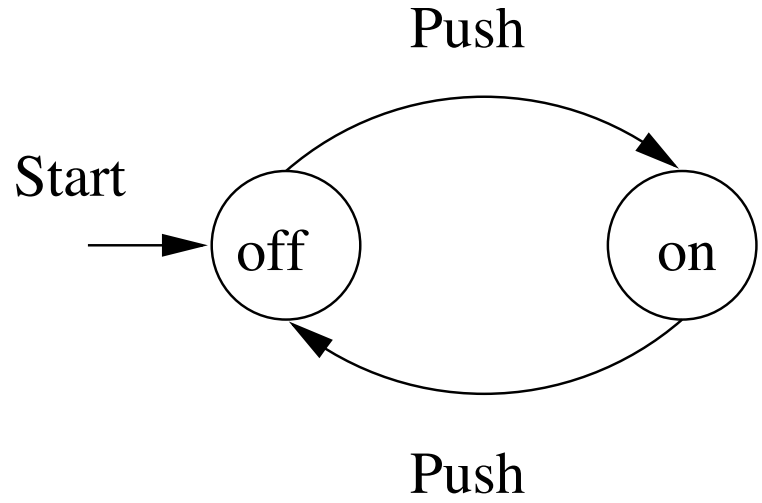
Esempio: automa a stati finiti per il game del tennis*



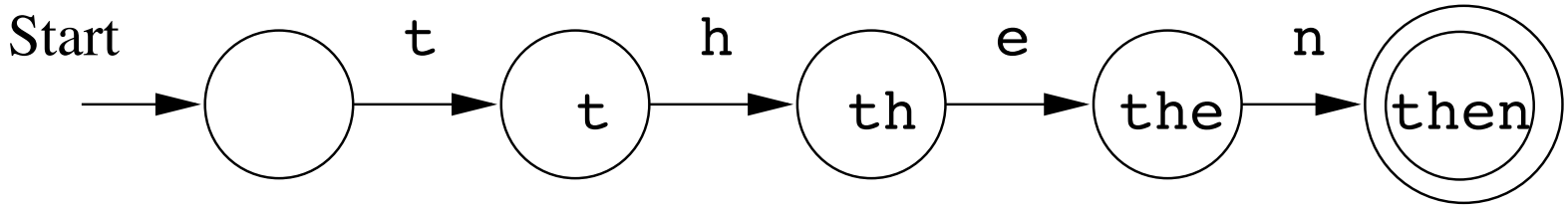
* Da "Automatic Java Code Generator for Regular Expression and Finite Automata" di Suejb Memeti, riprendendo un esempio di J. D. Ullman

Esempi

- Esempio: automa a stati finiti per un interruttore on/off



- Esempio: automa a stati finiti che riconosce la stringa `then`



Accettare input

- Data una sequenza (o stringa) di input, si comincia dallo stato iniziale e si seguono gli archi delle transizioni di simbolo in simbolo.
- L"input viene accettato quando dopo aver letto tutti i simboli in input si finisce in uno stato finale. Nell'esempio di prima, se la stringa in ingresso è proprio `then`, l'input è accettato, mentre non lo è se la stringa è `ten`.

AUTOMI: dall'analisi di un esempio alla loro formalizzazione

Fondamenti di Programmazione: AUTOMI E LINGUAGGI FORMALI
Corso di Laurea in MATEMATICA
a.a. 2022/2023

PROBLEMA 1

Un robot lancia in aria una moneta. I primi due tiri su tre danno lo stesso risultato (**T**esta, **T**esta o **C**roce**, C**roce).

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

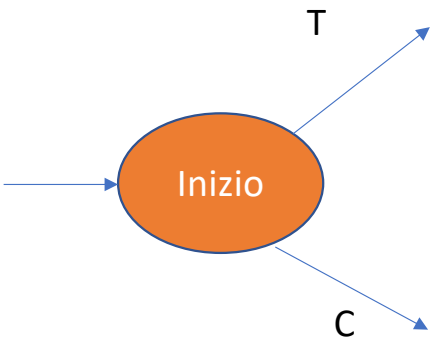
L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}

Stringhe w composte da elementi in {T,C}

PROBLEMA 1

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

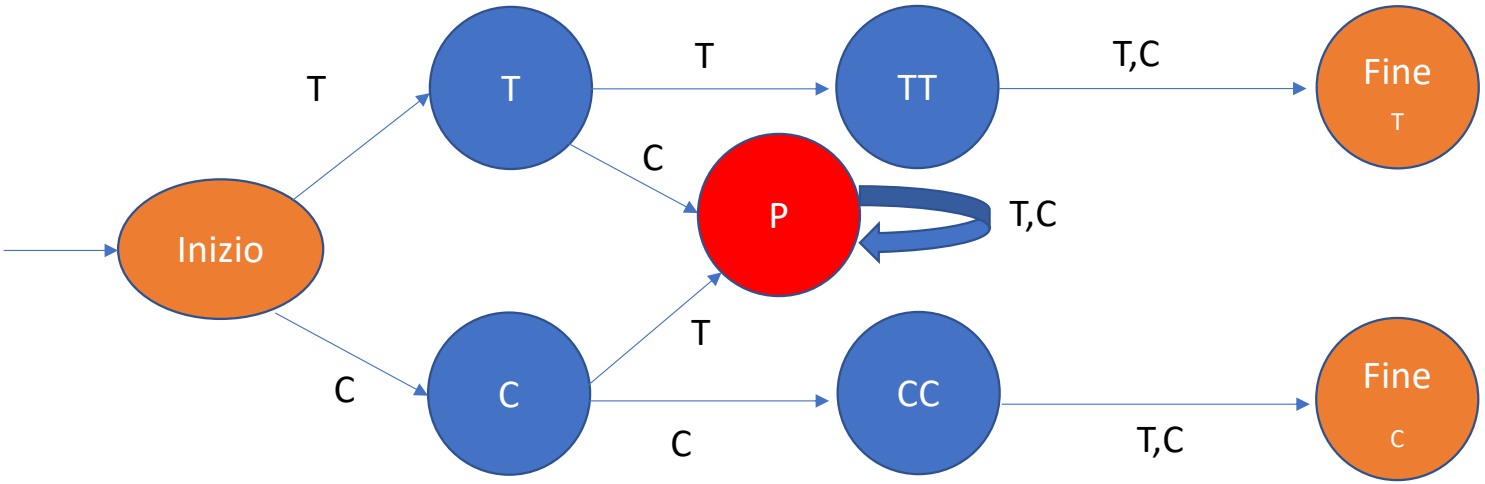
L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}



PROBLEMA 1

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

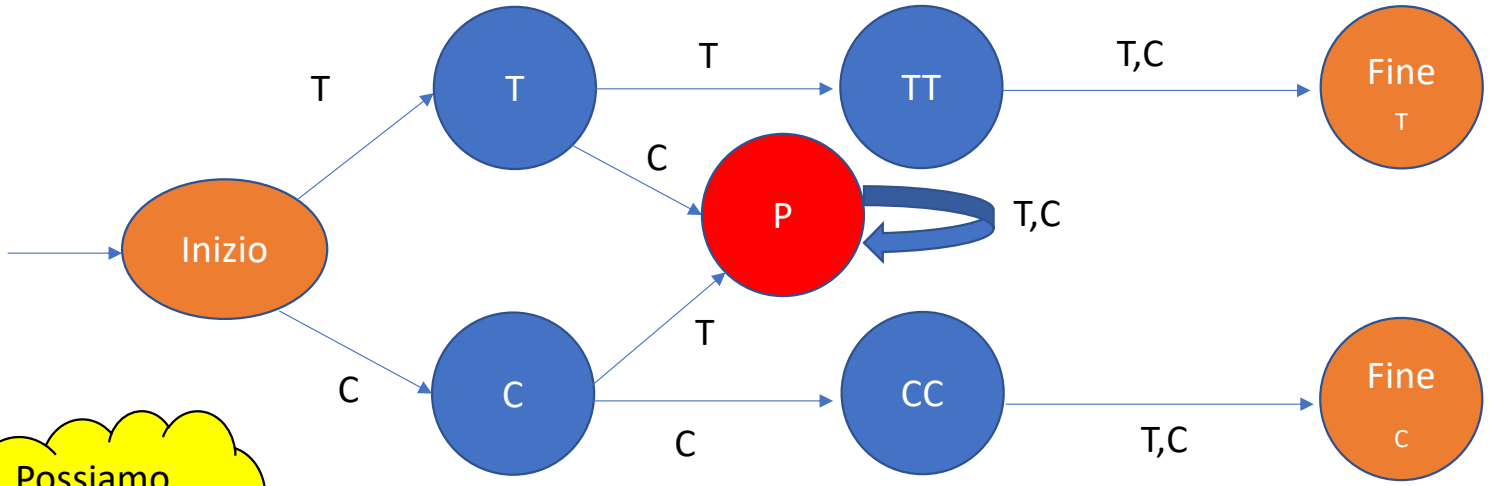
L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}



PROBLEMA 1

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}

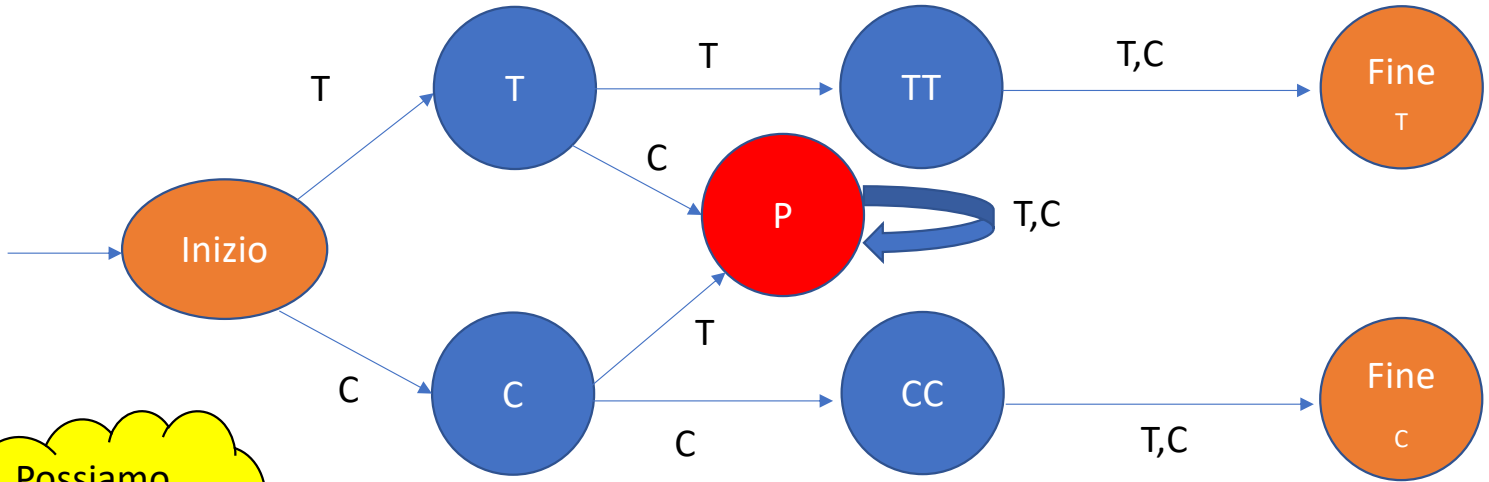


Possiamo avere un solo stato finale?

PROBLEMA 1

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}



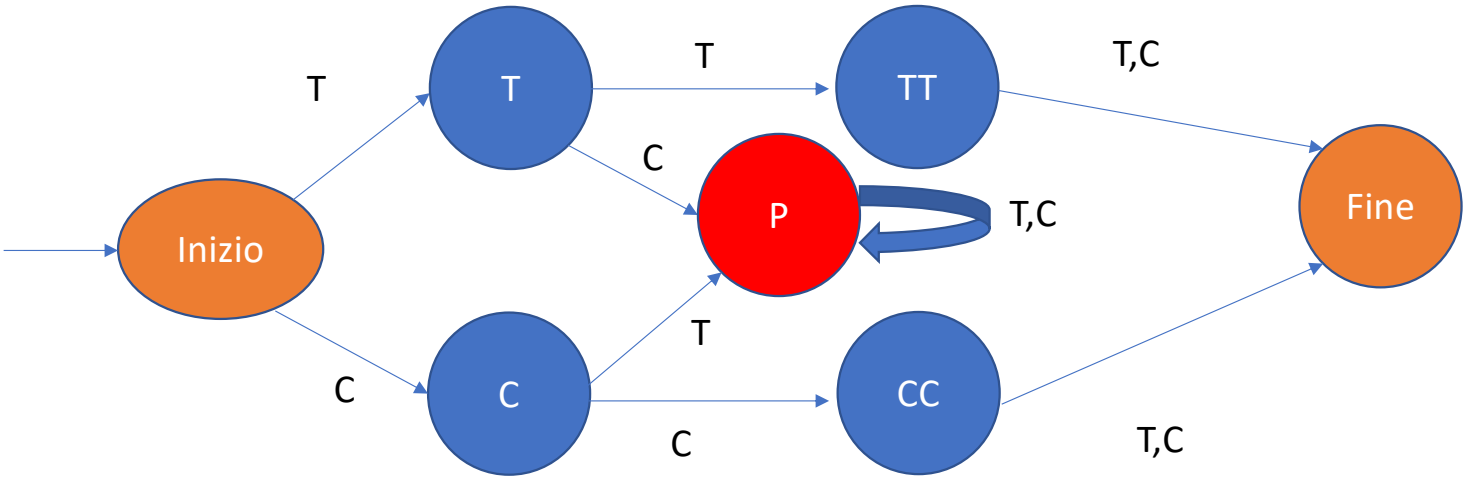
Possiamo avere un solo stato finale?

Sì, se non voglio ricordare se avevo TT o CC

PROBLEMA 1

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

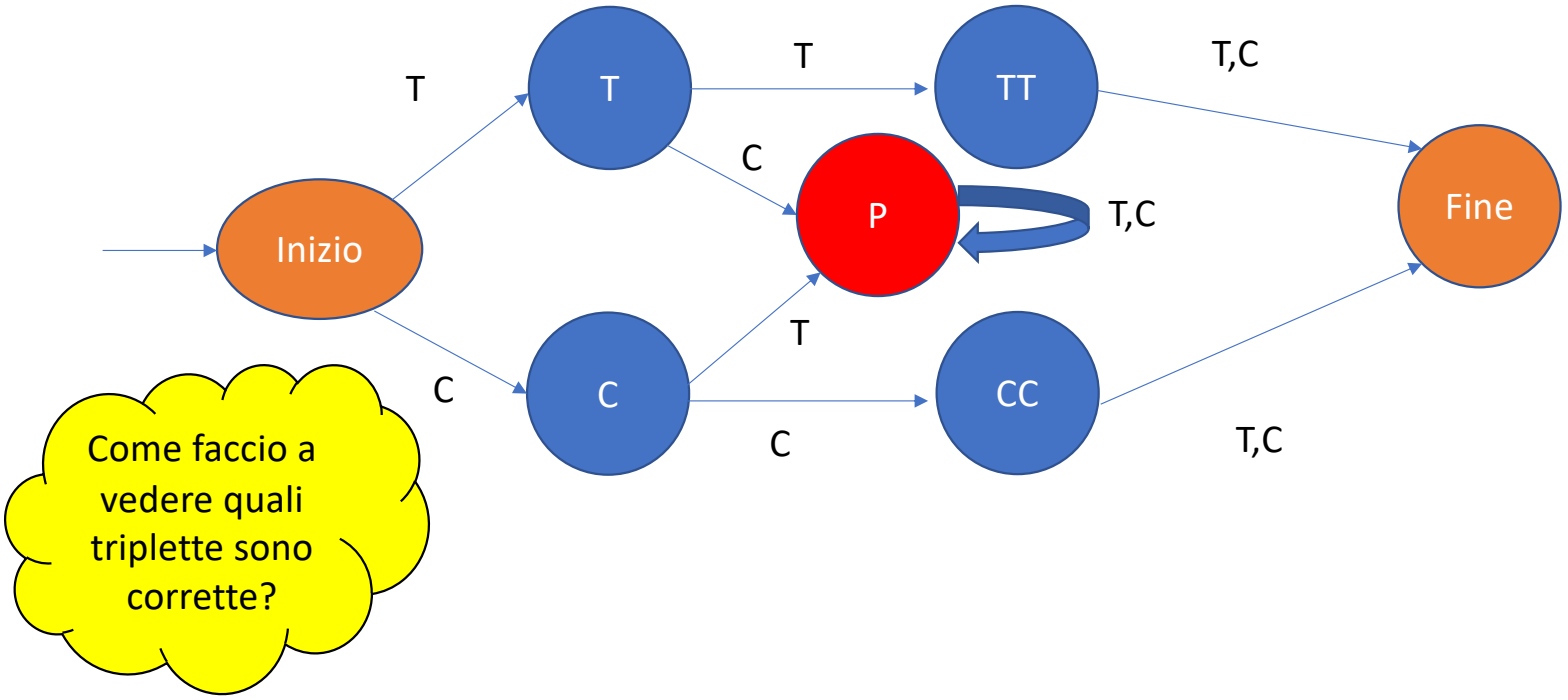
L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}



PROBLEMA 1

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

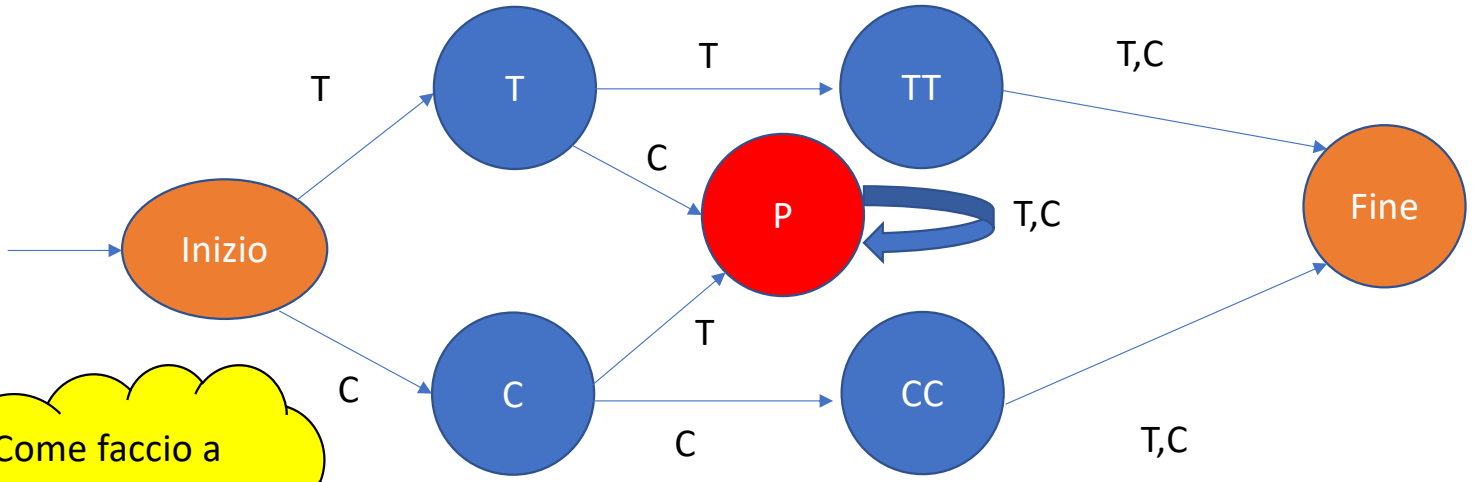
L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}



PROBLEMA 1

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

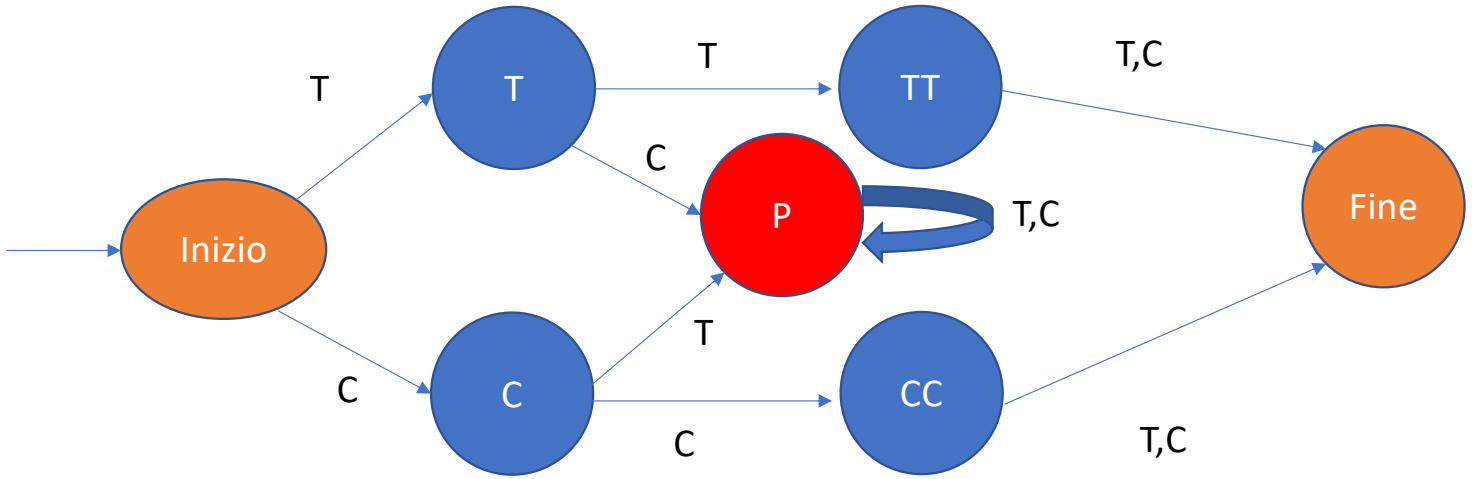
L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}

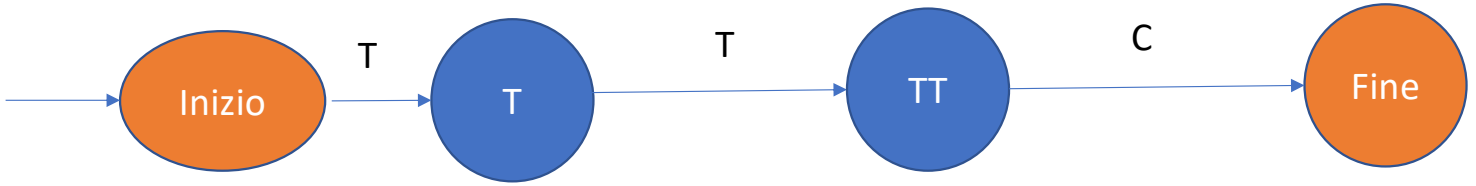


PROBLEMA 1

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}



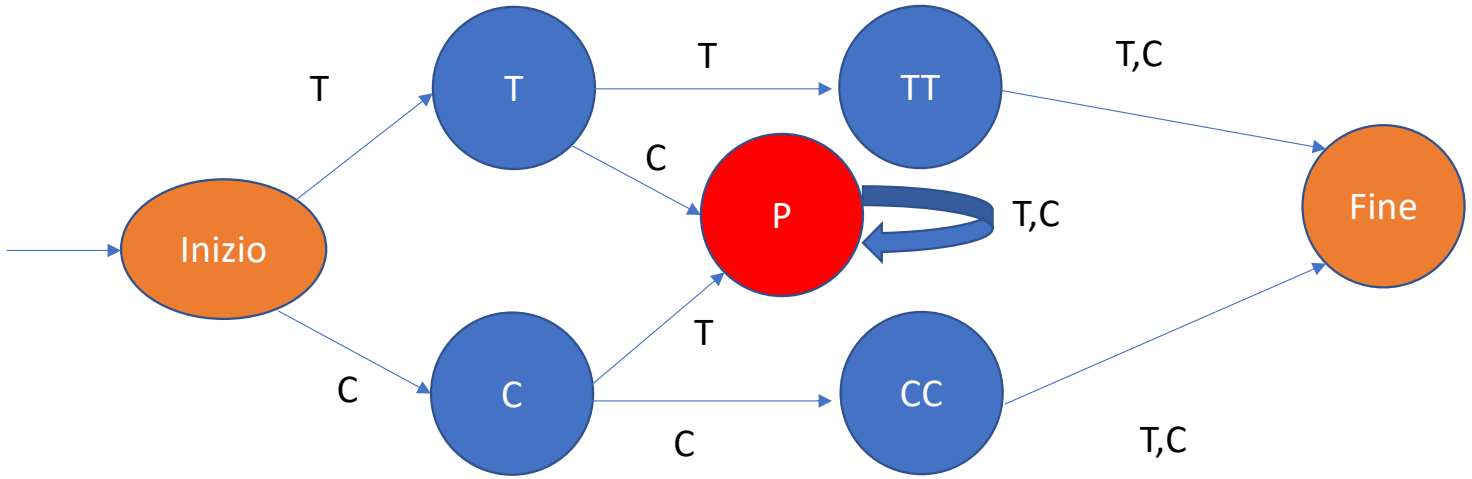


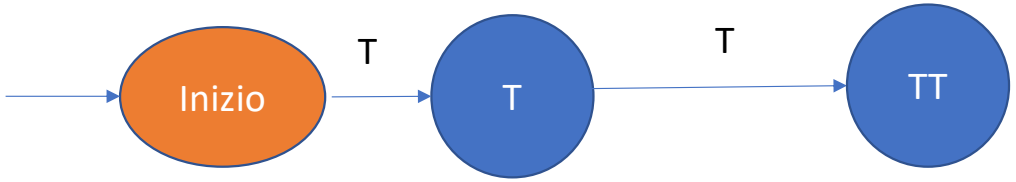
Sequenza accettata
TTC

PROBLEMA 1

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}



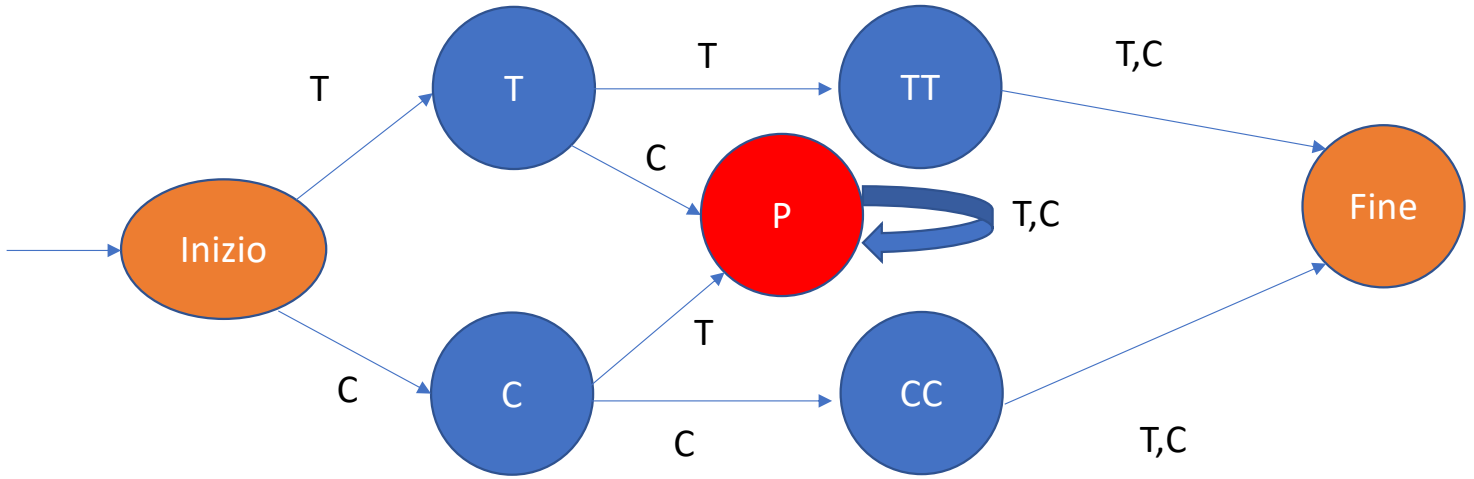


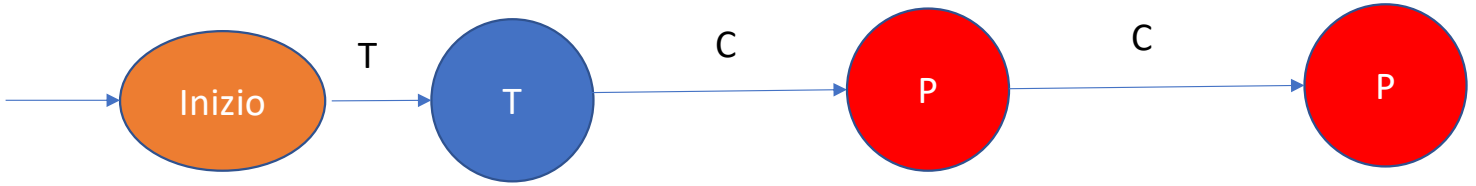
Sequenza non accettata
TT

PROBLEMA 1

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}



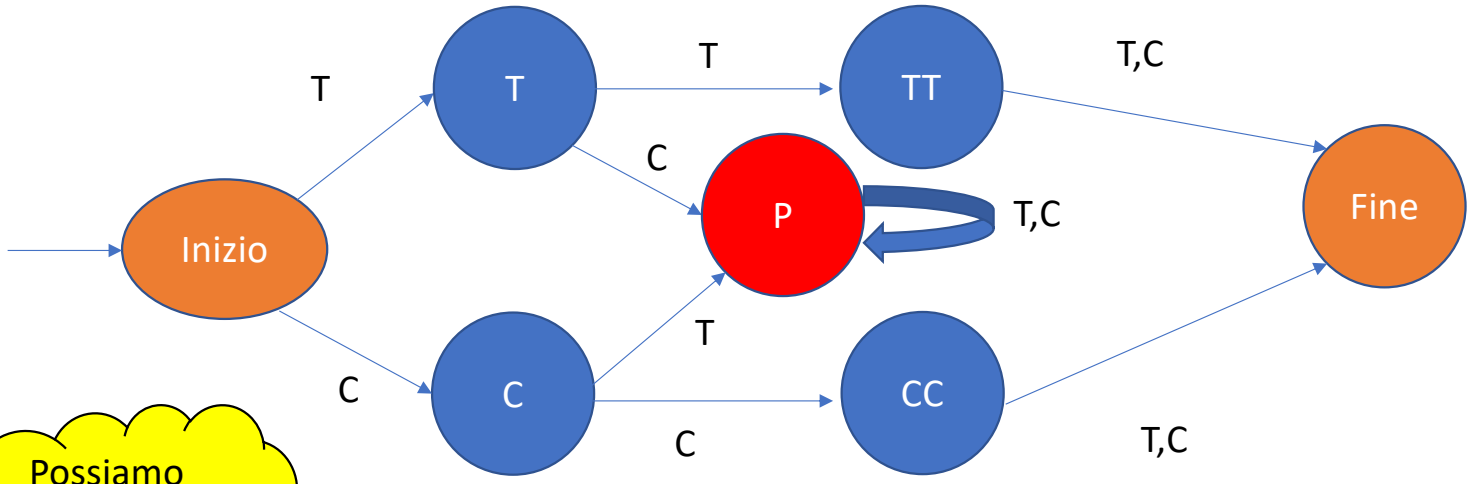


Sequenza rifiutata
TCC

PROBLEMA 1

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}

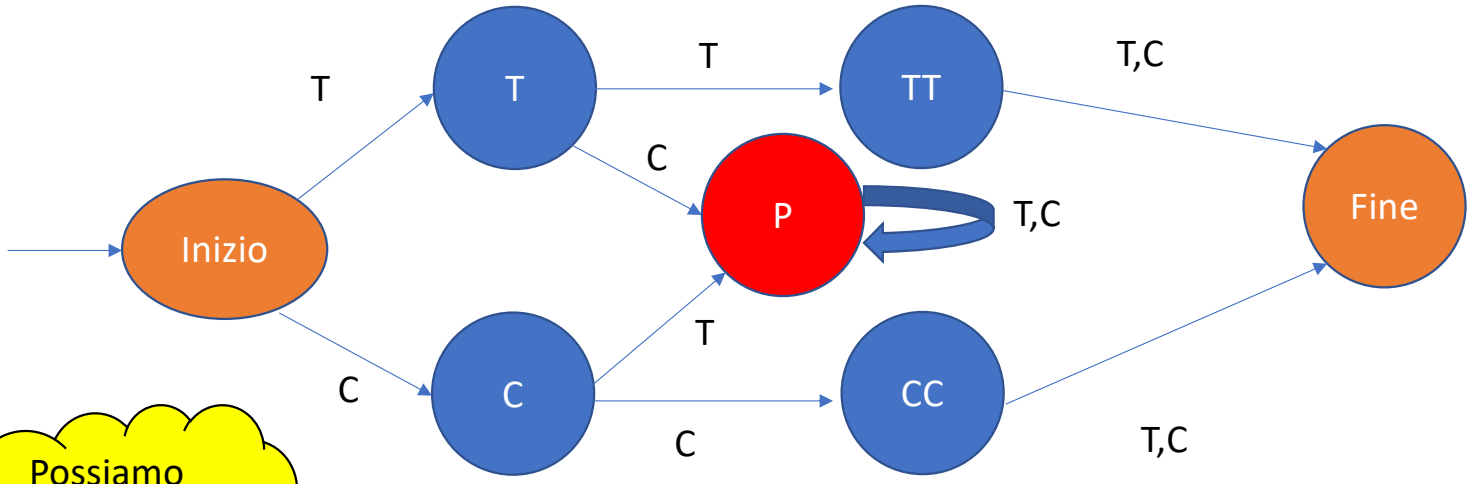


Possiamo riconoscere sequenze di triplette?

PROBLEMA 1

Trovare l'automa A che riconosce se una singola tripletta w è corretta rispetto alle regola:

L = {w in {T,C}* è OK sse w = ε oppure w = xxy, con x,y in {T,C}}



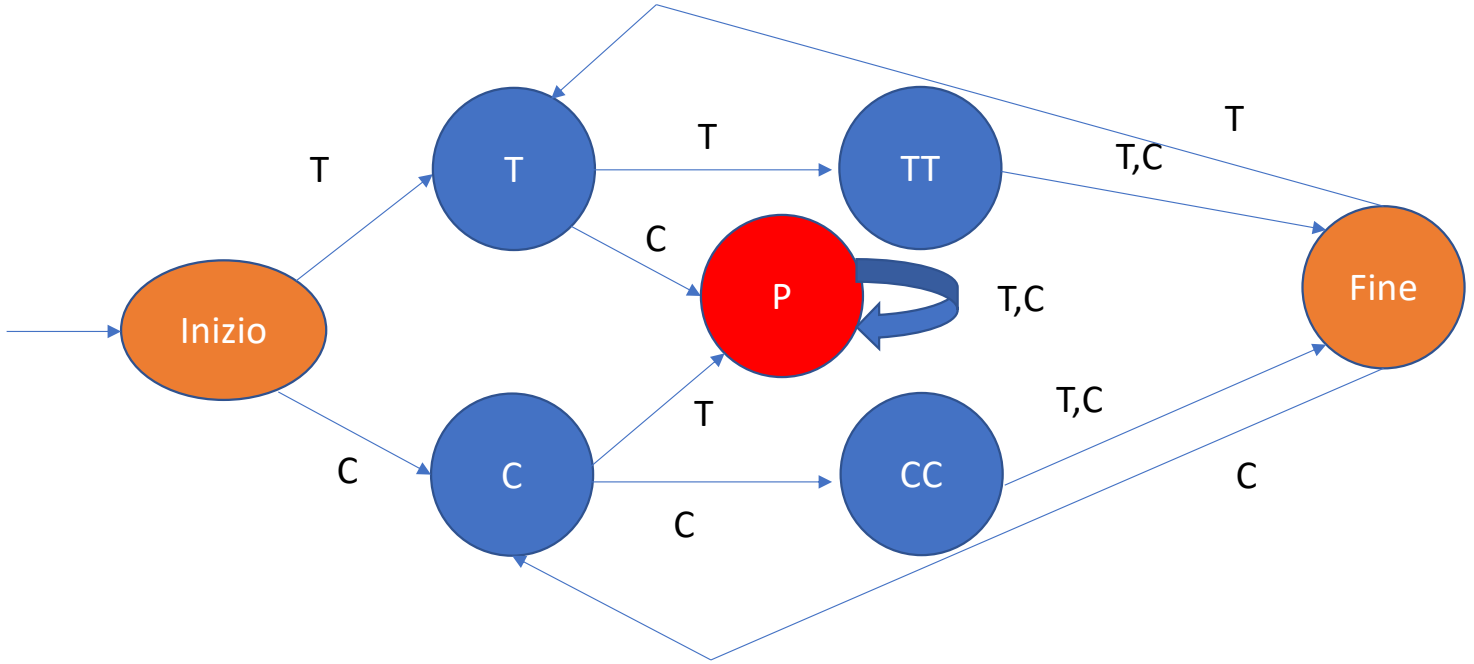
Possiamo riconoscere sequenze di triplette?

Sì, se ogni volta posso iniziare ancora

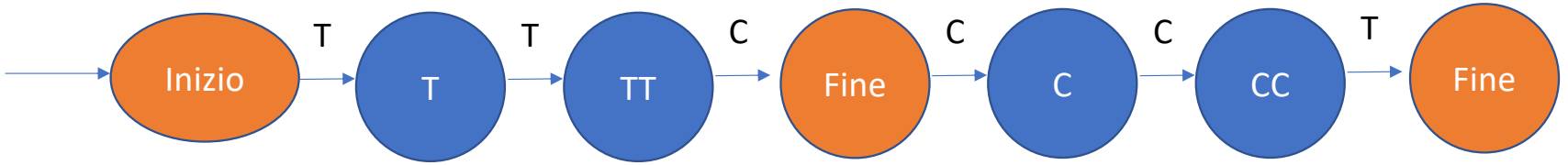
PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



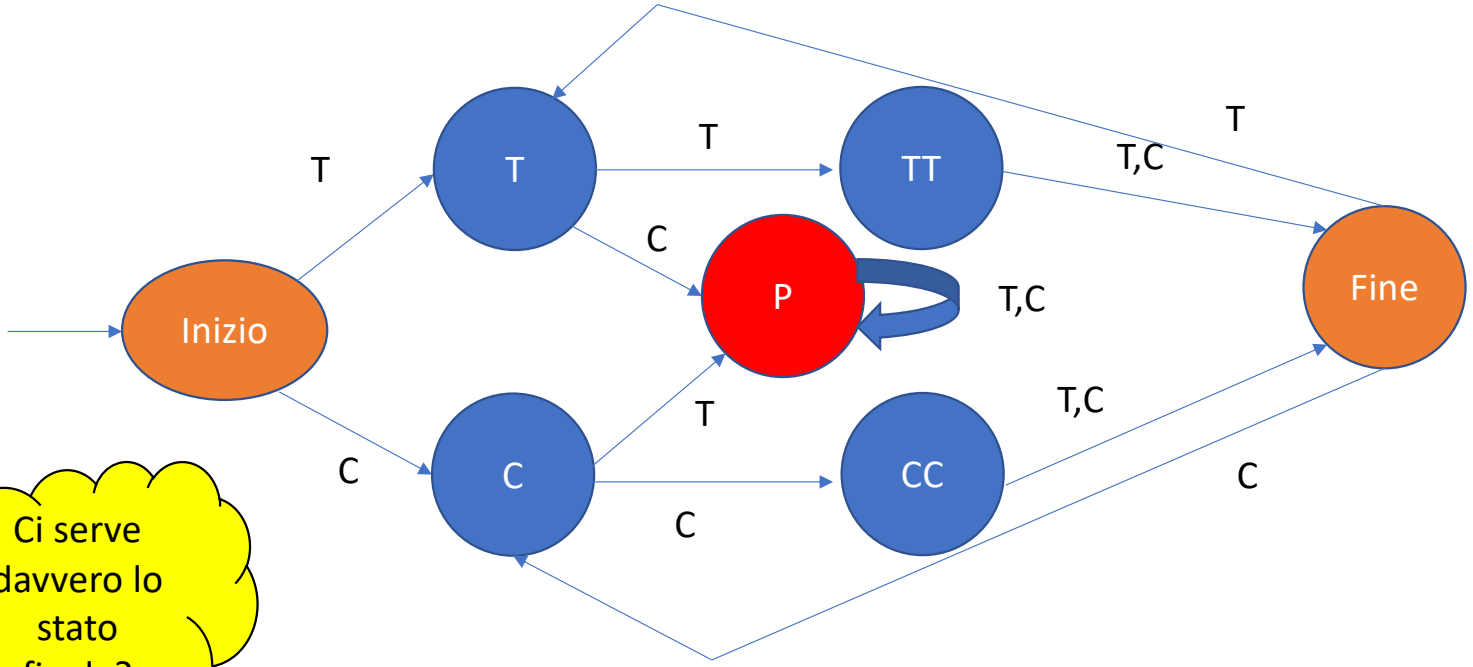
Sequenza accettata
TTC CCT



PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

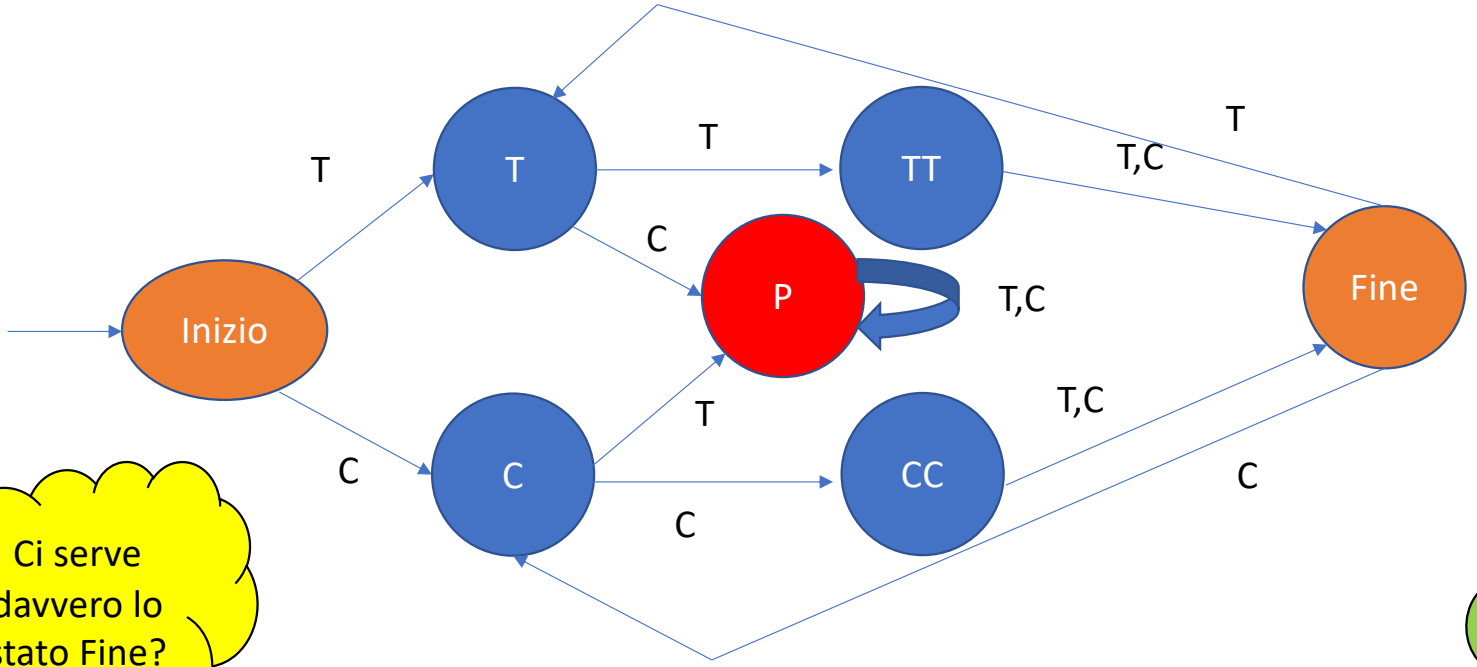
L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



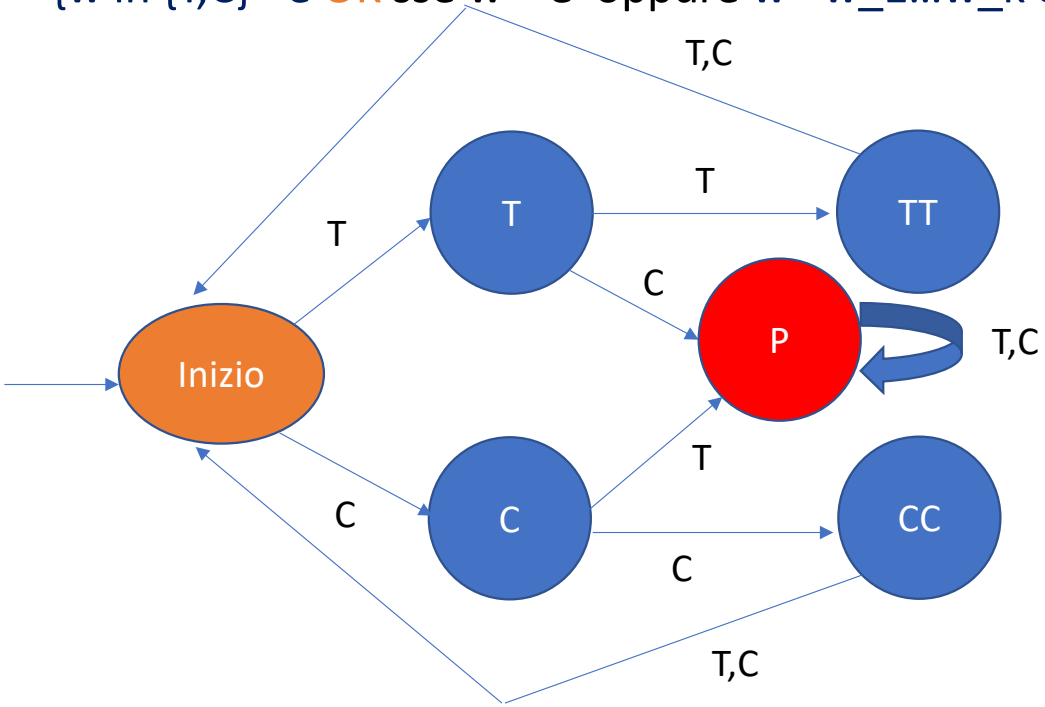
Ci serve davvero lo stato Fine?

No, perché fa quello che fa lo stato Inizio

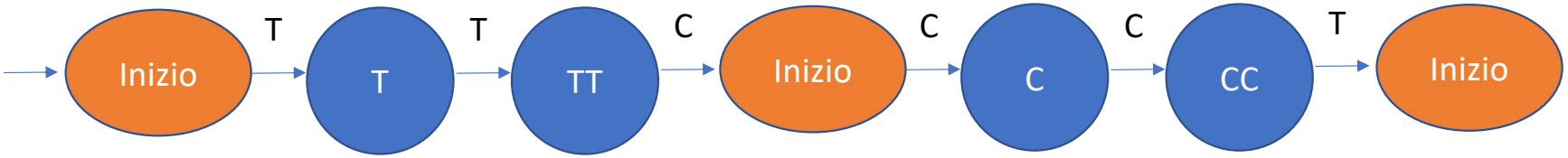
PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



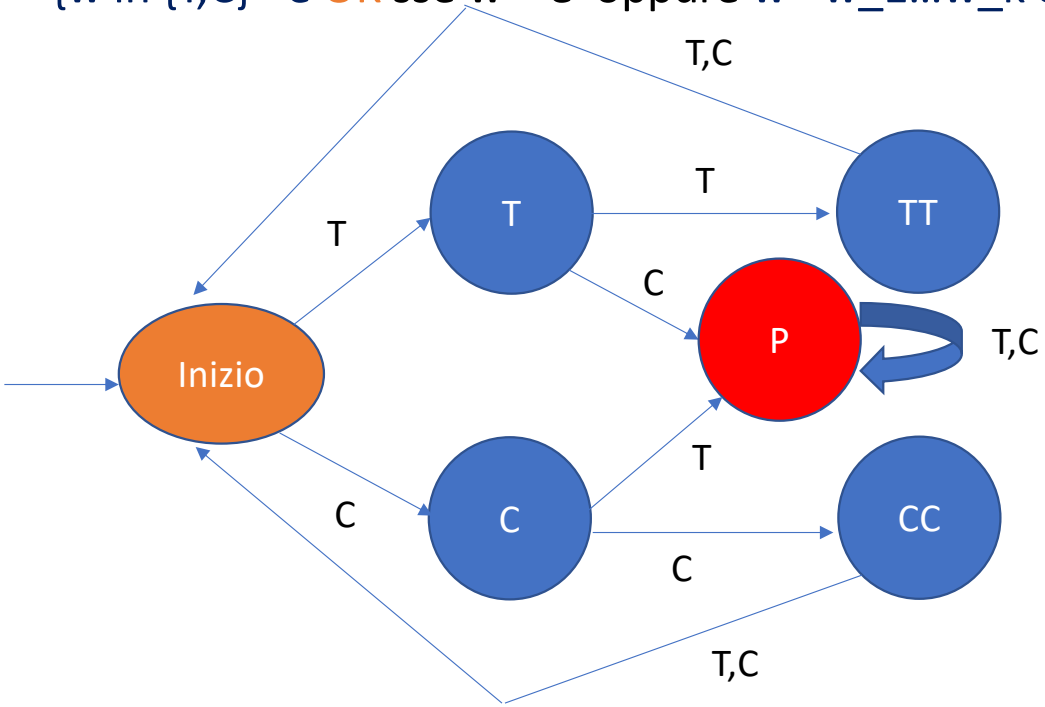
Sequenza accettata
TTC CCT



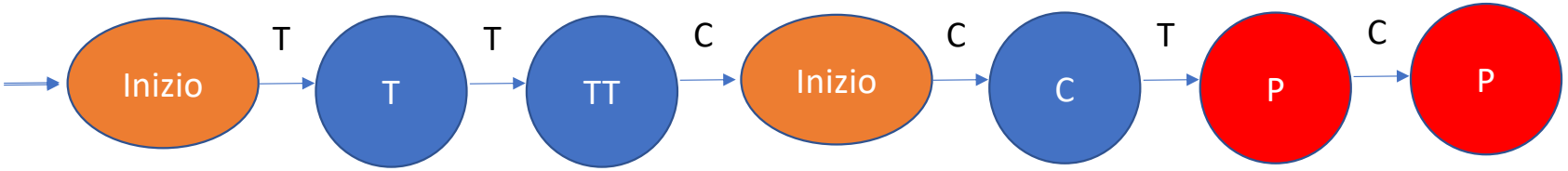
PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



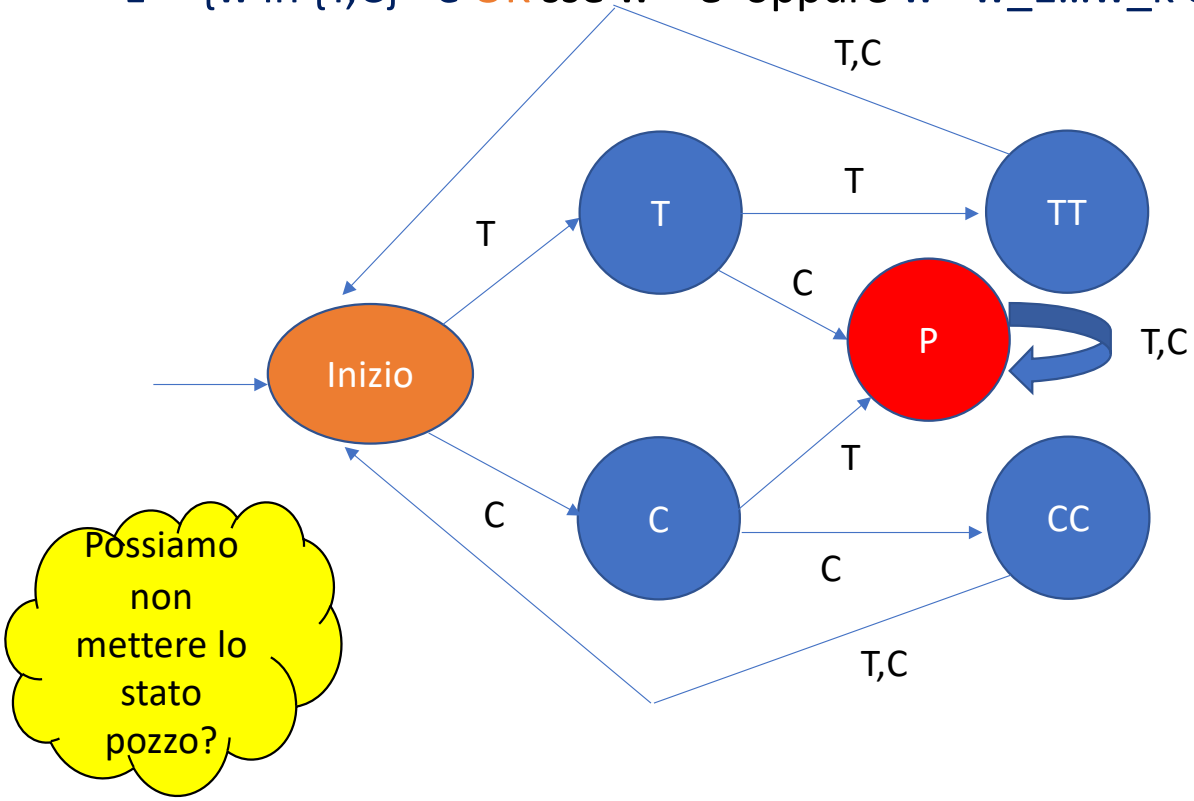
Sequenza rifiutata
TTC CTC



PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

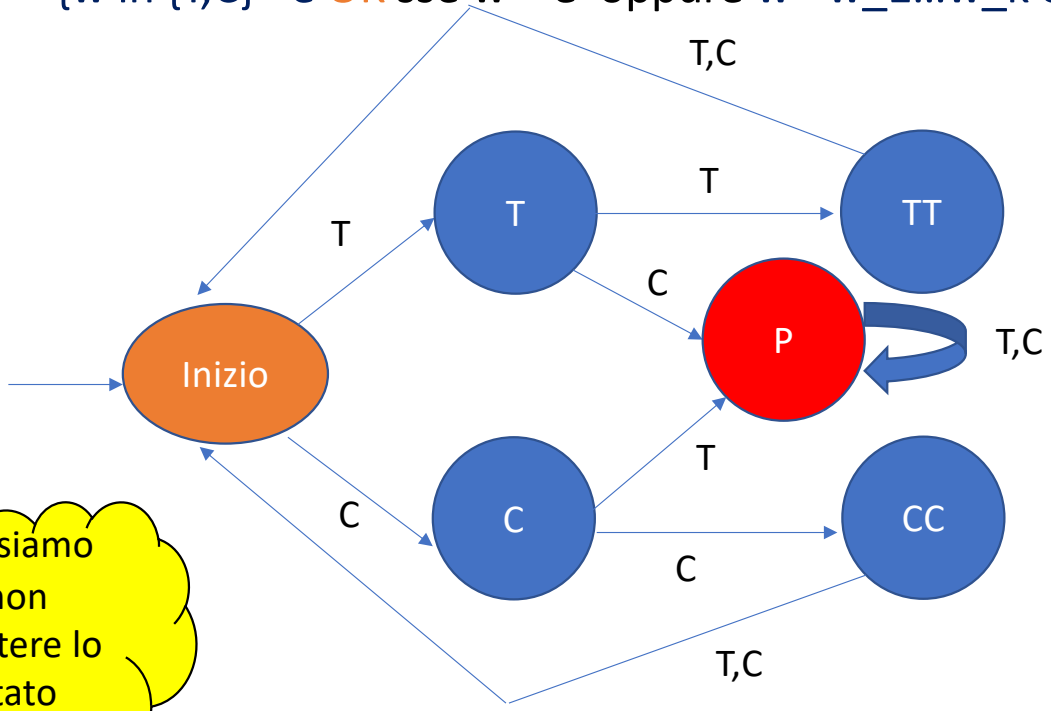
L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



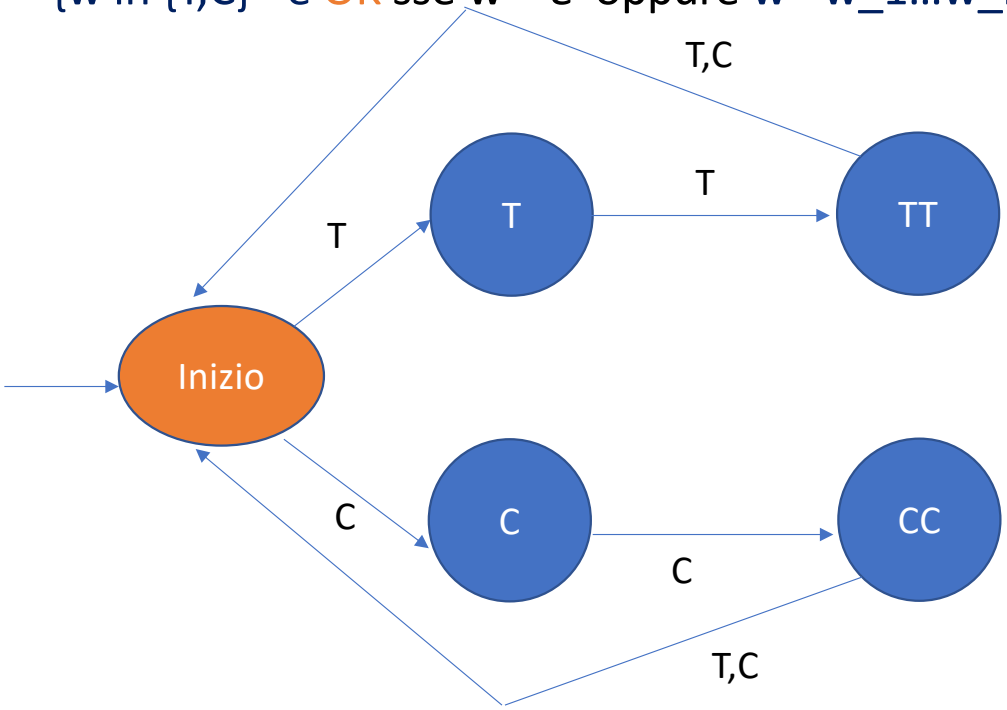
Possiamo non mettere lo stato pozzo?

Sì, possiamo considerarlo implicitamente

PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

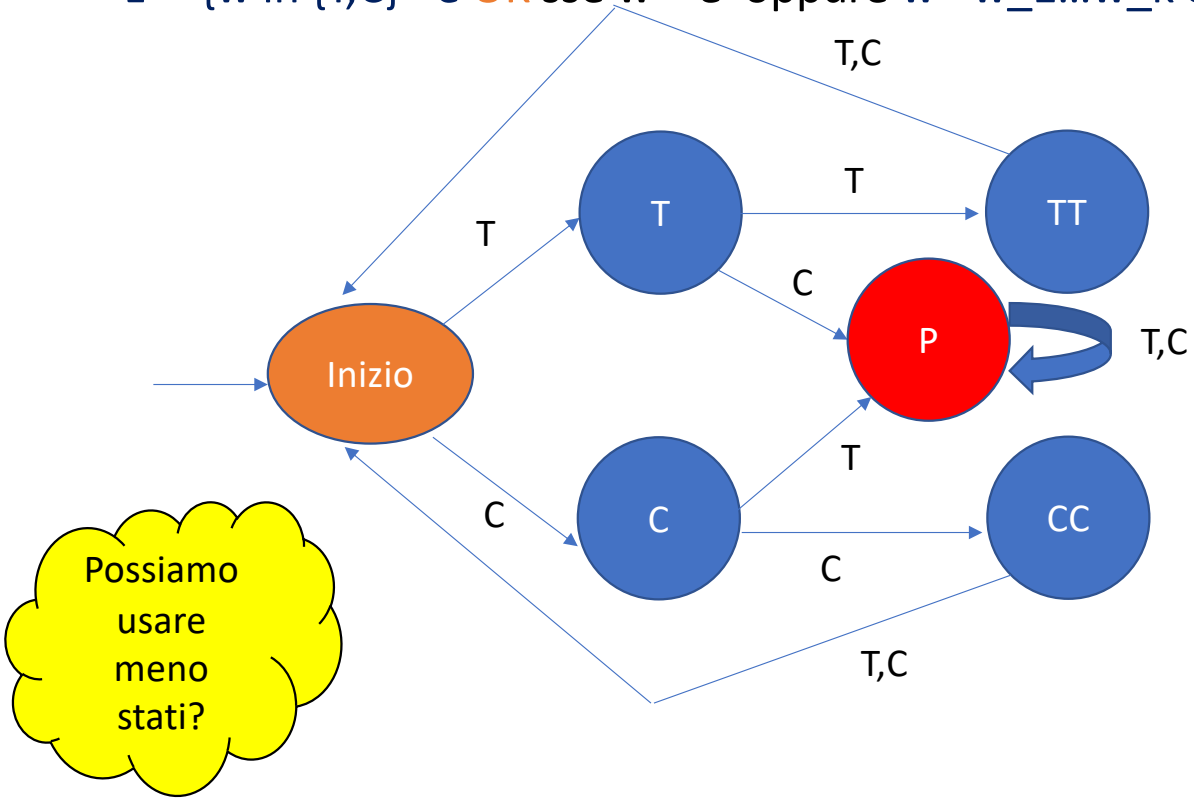
L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

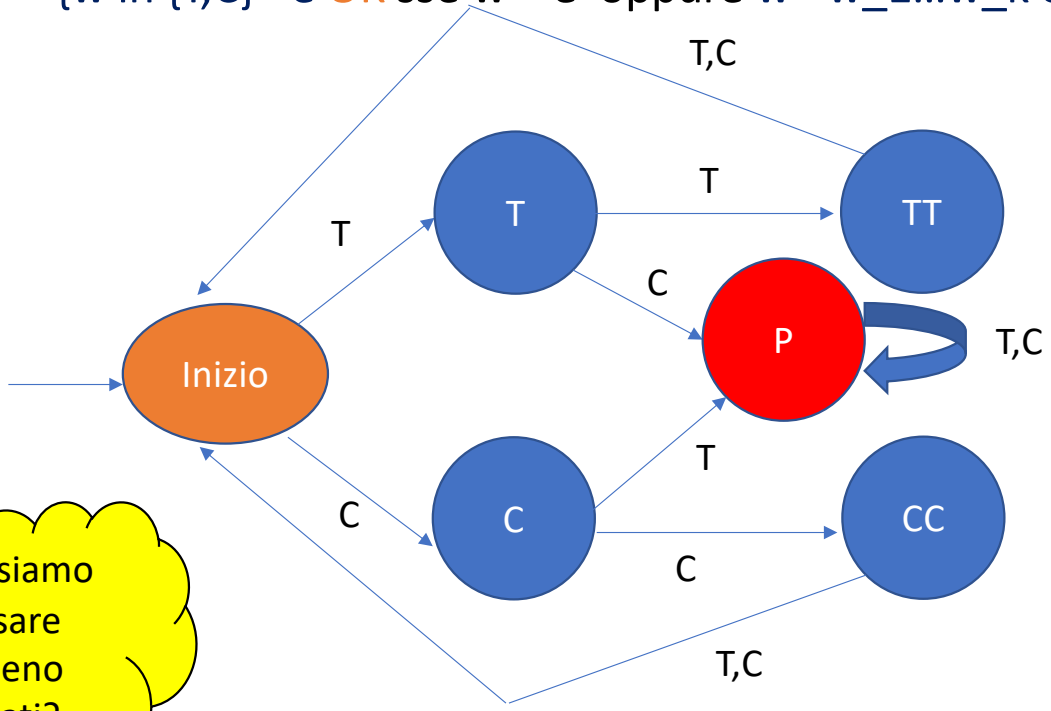
L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



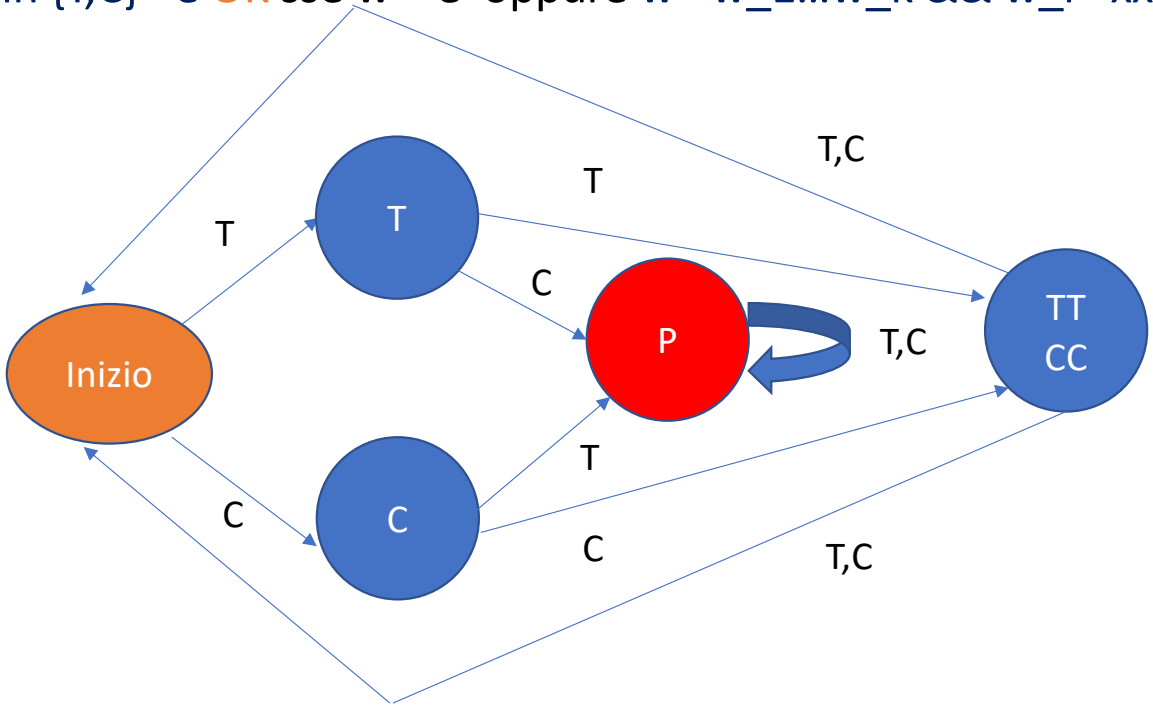
Possiamo usare meno stati?

Sì, gli stati **TT** e **CC** possono collassare in uno stato solo

PROBLEMA 2

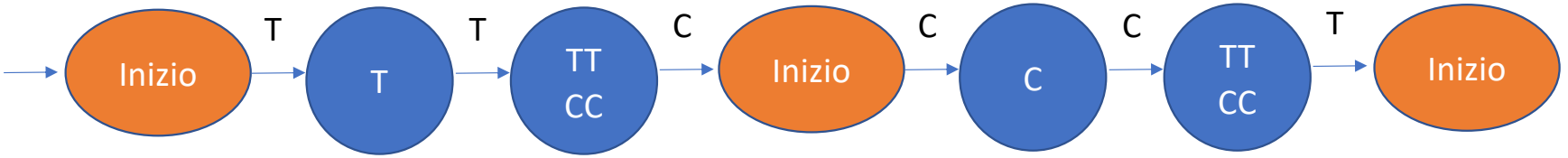
Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



Bastano meno stati

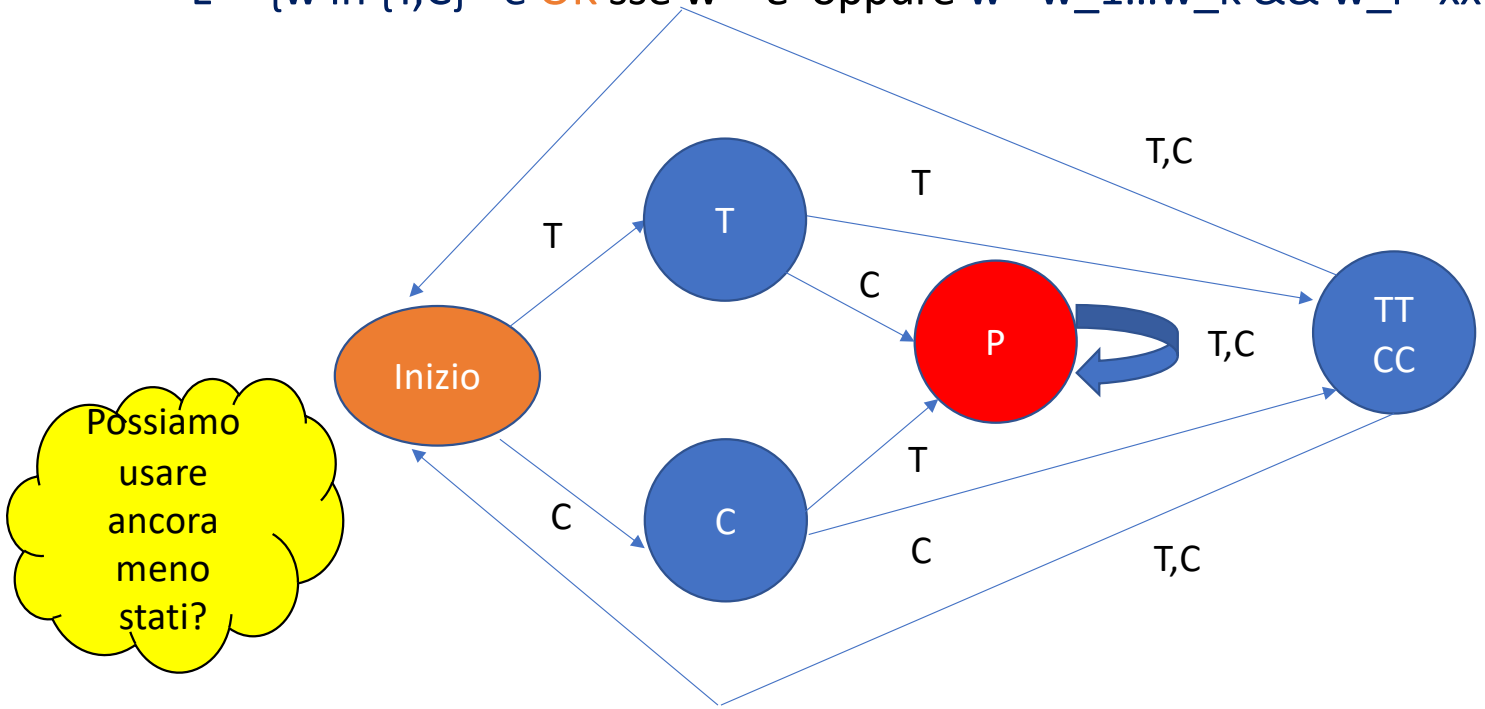
Sequenza accettata
TTC CCT



PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

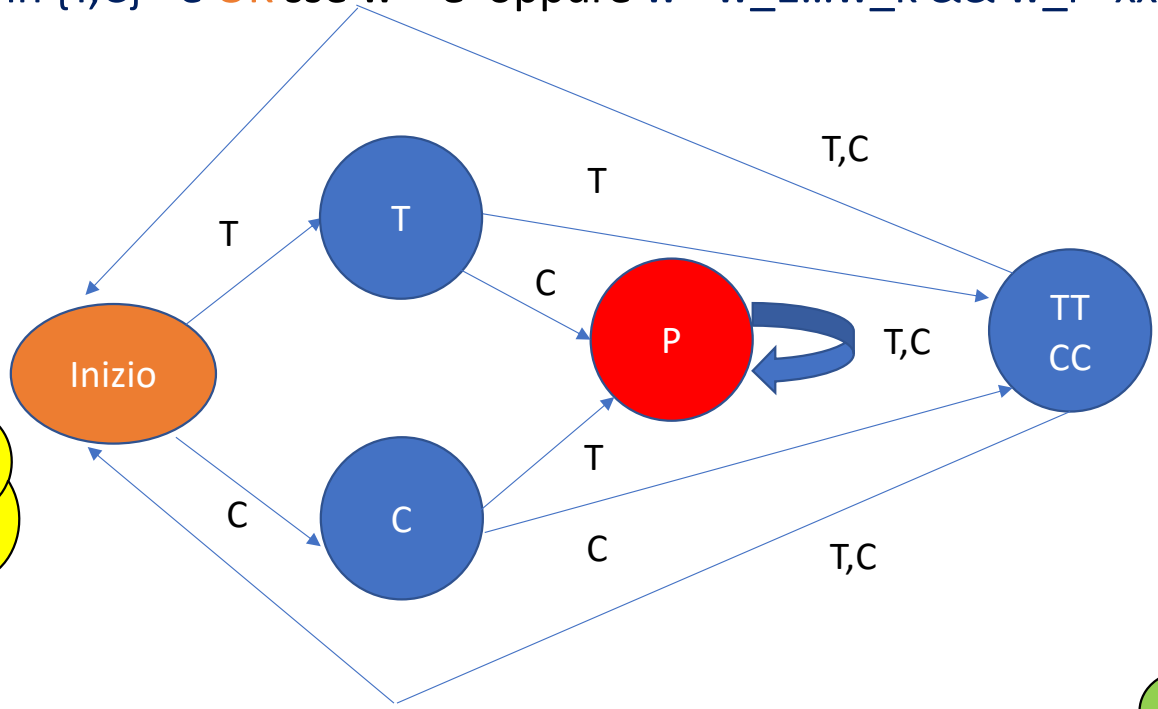
L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



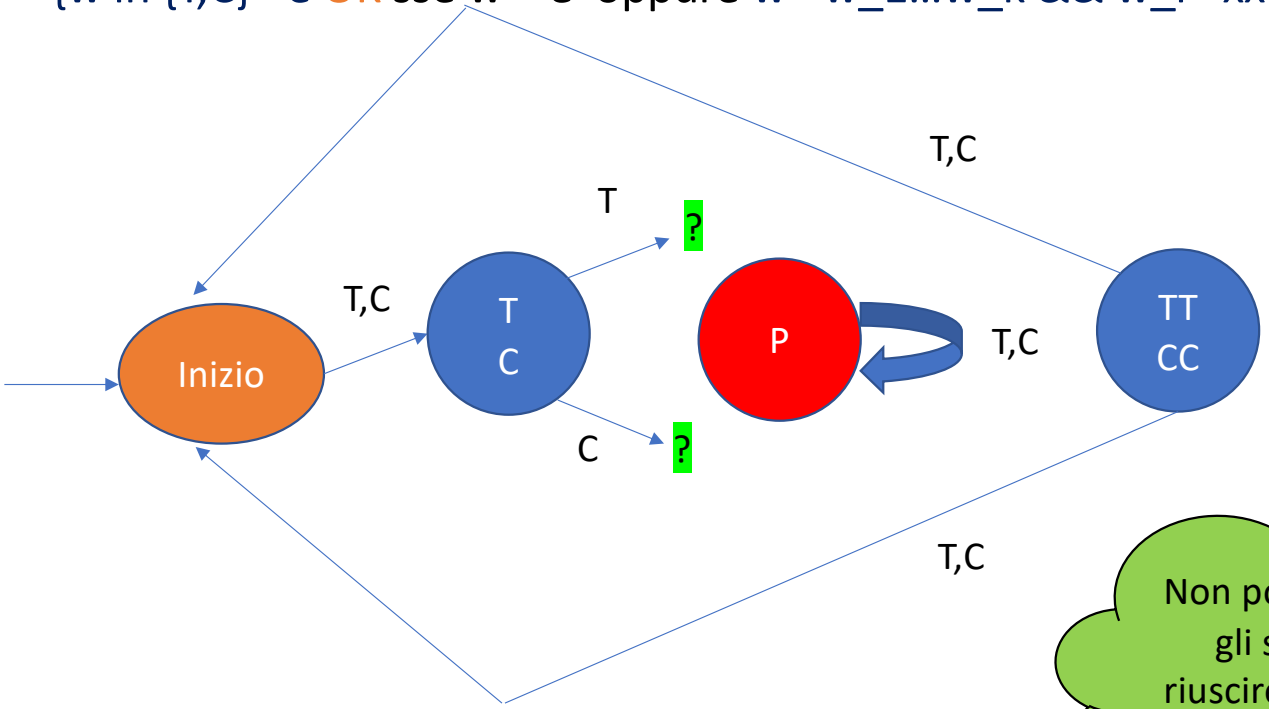
Possiamo usare ancora meno stati?

Proviamo a far collassare gli stati **T** e **C** in uno stato solo

PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



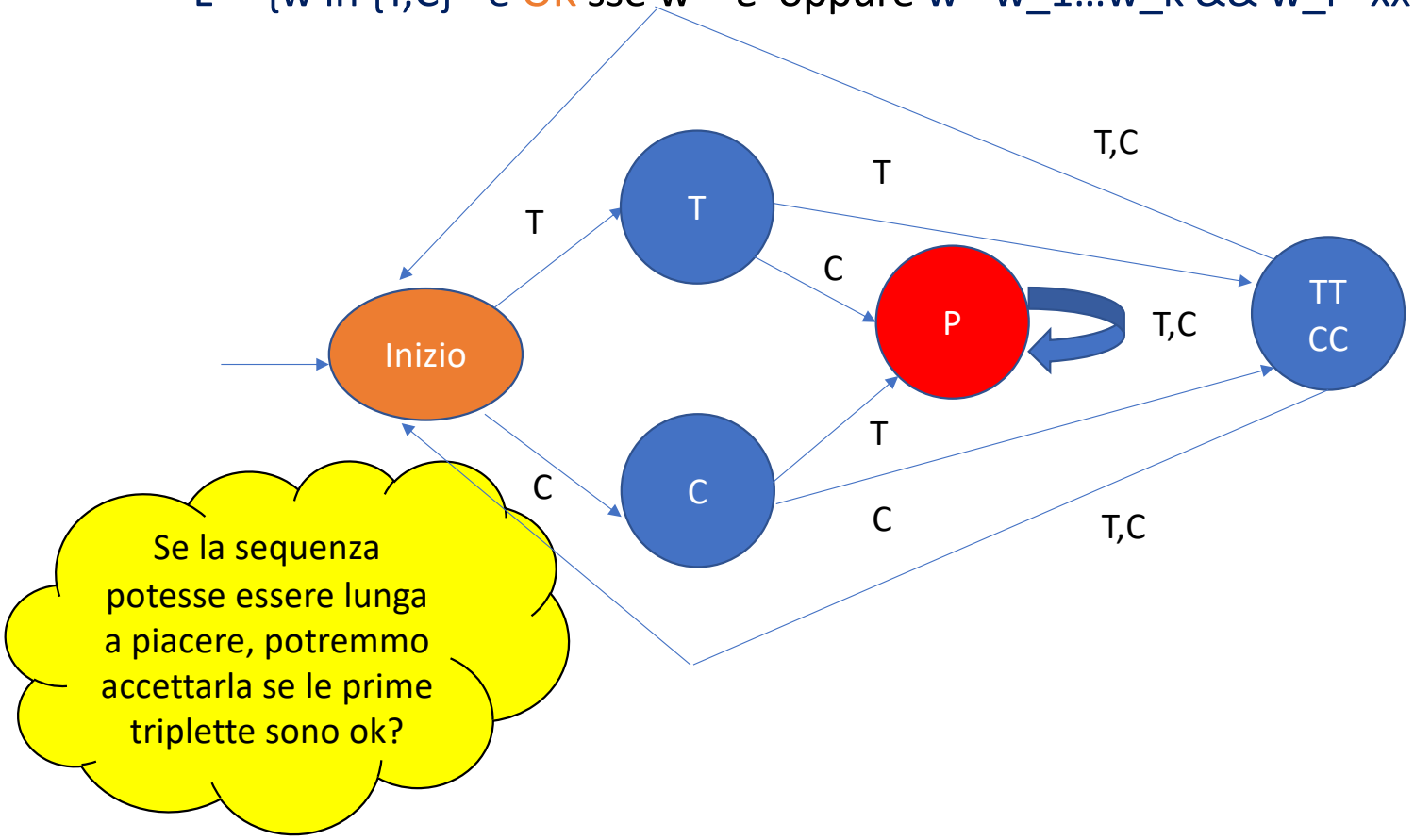
- Se entra T e dopo C? Dovrei andare nello stato **P**
- Se entra C e dopo C? Dovrei andare nello stato **TT/CC**

Non posso far collassare gli stati **T** e **C**. Non riuscirei a distinguere se ho letto T oppure C

PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

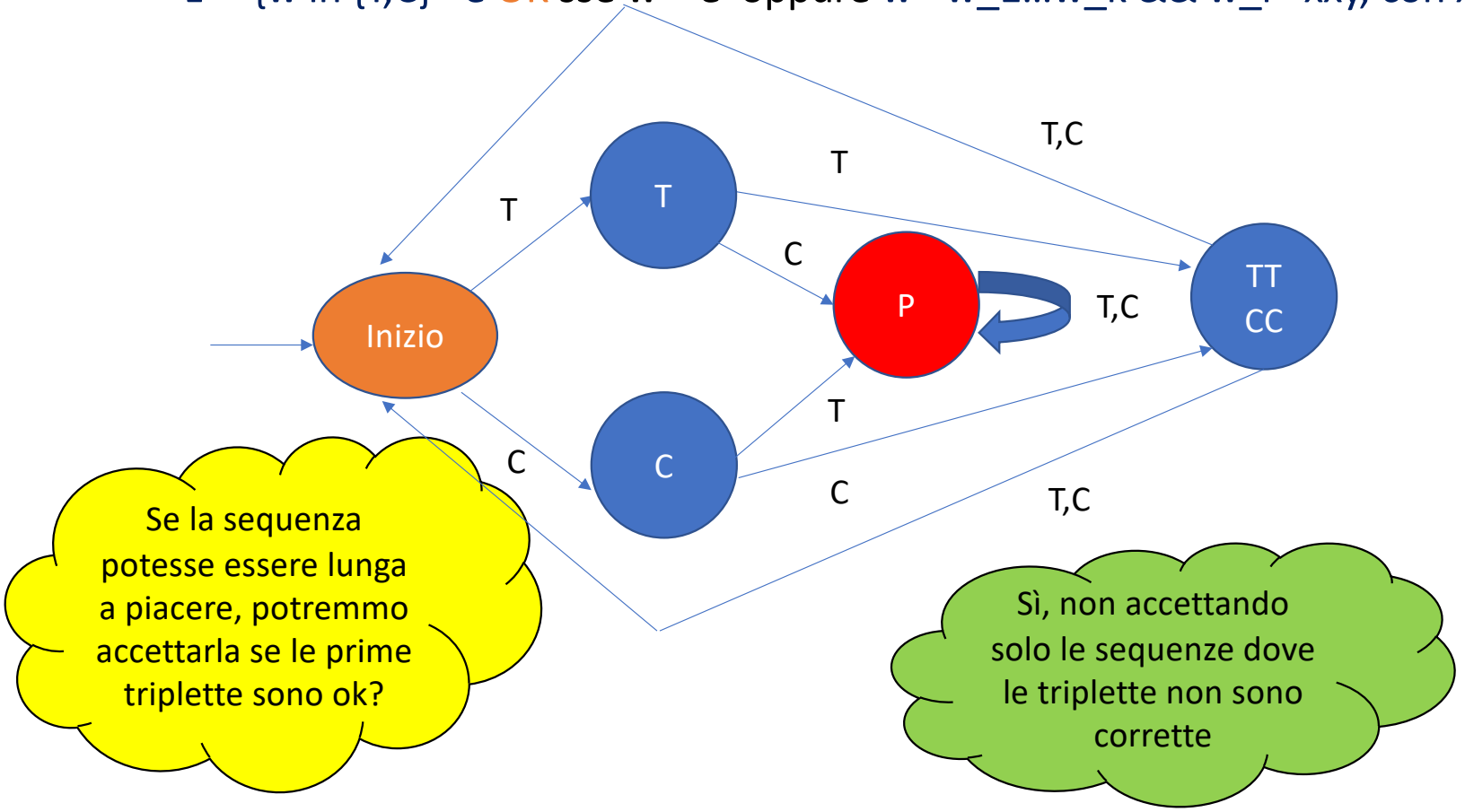
L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

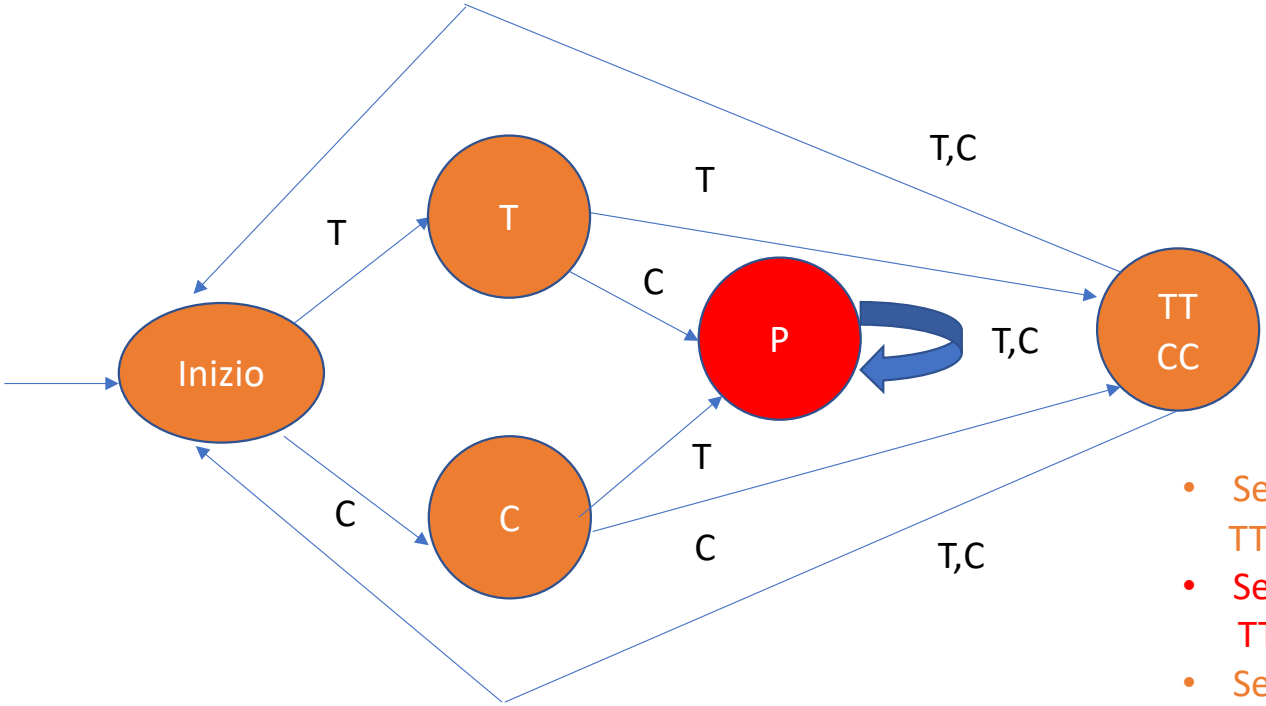
L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



PROBLEMA 3

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

L'' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k z && w_i =xxy, con x,y in {T,C} e z in $\{T,C\}^0 \cup \{T,C\}^1 \cup \{T,C\}^2$}

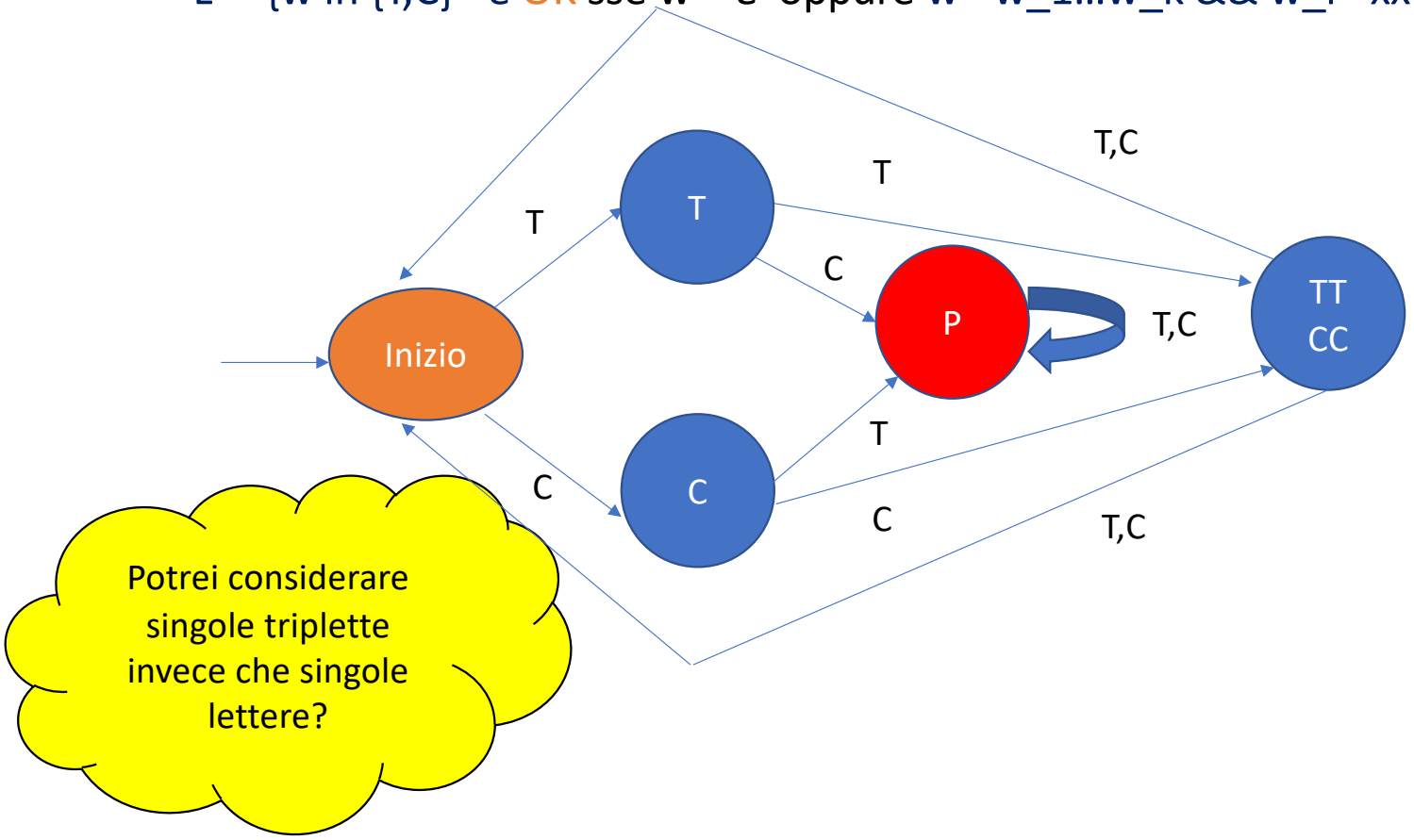


- Sequenza accettata
  TTC CCT TTC
- Sequenza non accettata
  TTC CTC TTC
- Sequenza ~~incompleta~~ accettata
  TTC CC

PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

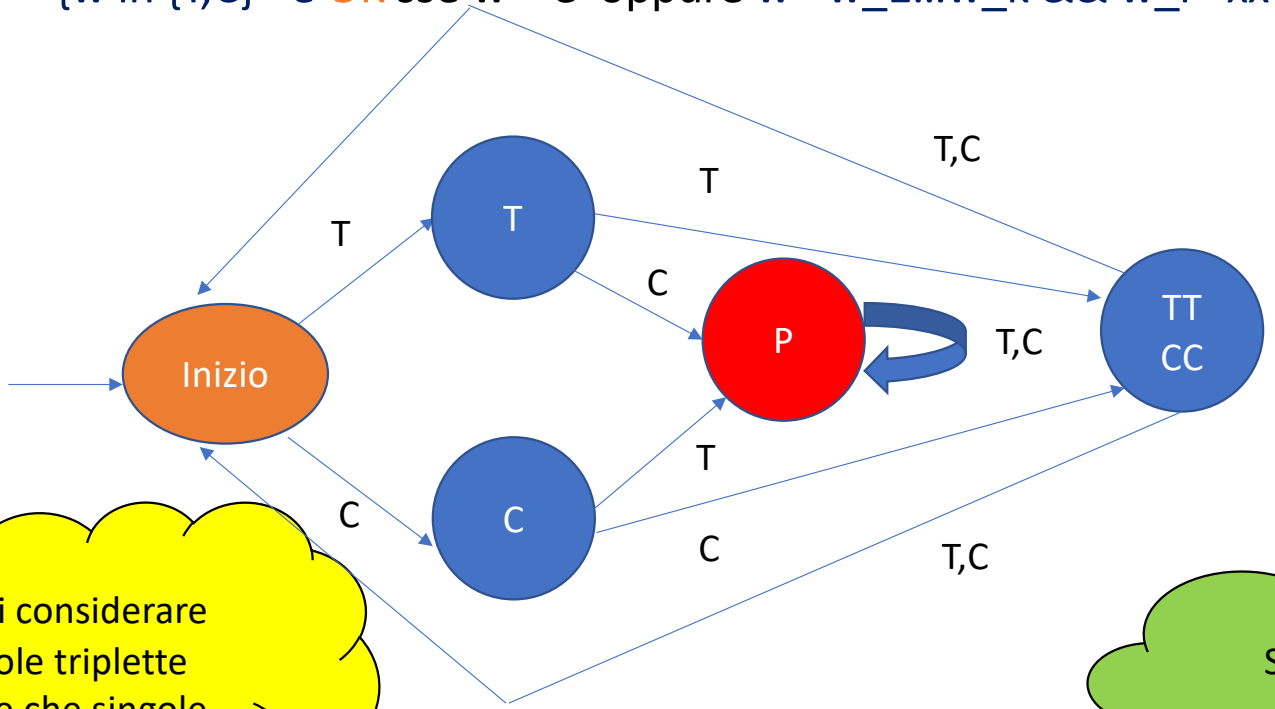
L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



PROBLEMA 2

Trovare l'automa A che riconosce se una sequenza di triplette w abbia tutte le triplette in regola:

L' = {w in {T,C}* è OK sse w = ε oppure w =w_1…w_k && w_i =xxy, con x,y in {T,C}}



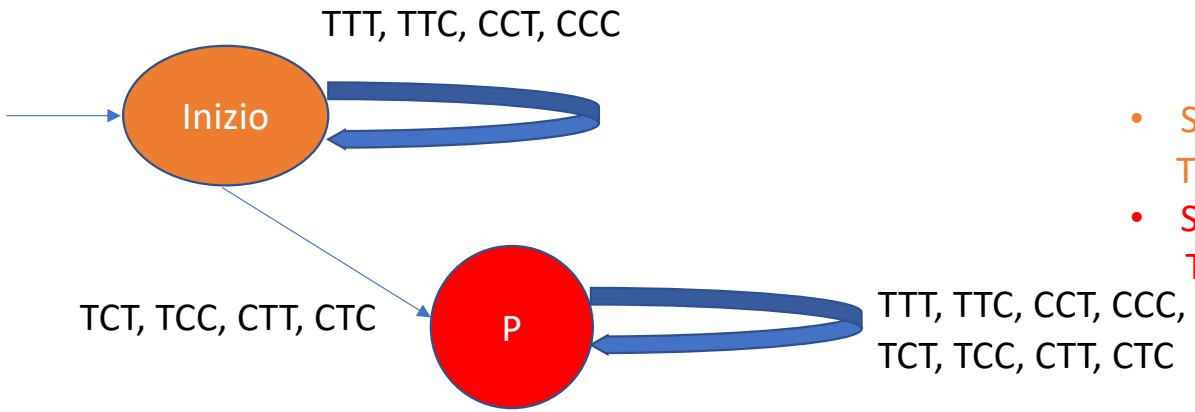
Potrei considerare singole triplette invece che singole lettere?

Sì, cambiando l'alfabeto

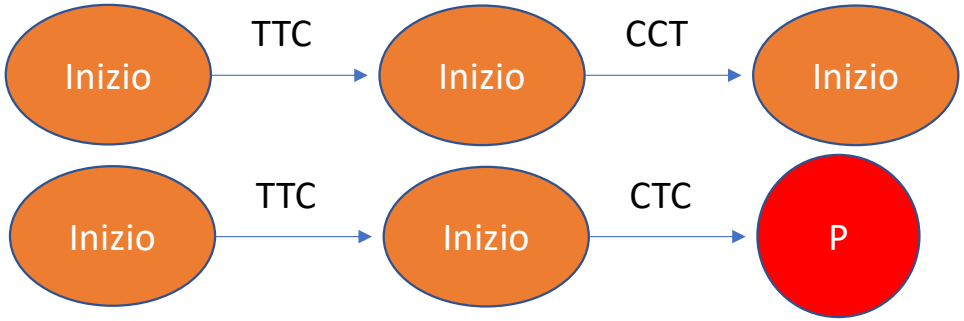
PROBLEMA 3

Trovare l'automa A che riconosce se una **sequenza di triplette** w abbia tutte le triplette in regola:

L'' = {w in {TTT,TTC,TCT,TCC,CCC,CCT,CTC,CTT}* è OK sse w = ε oppure w in {TTT, TTC, CCT, CCC}}

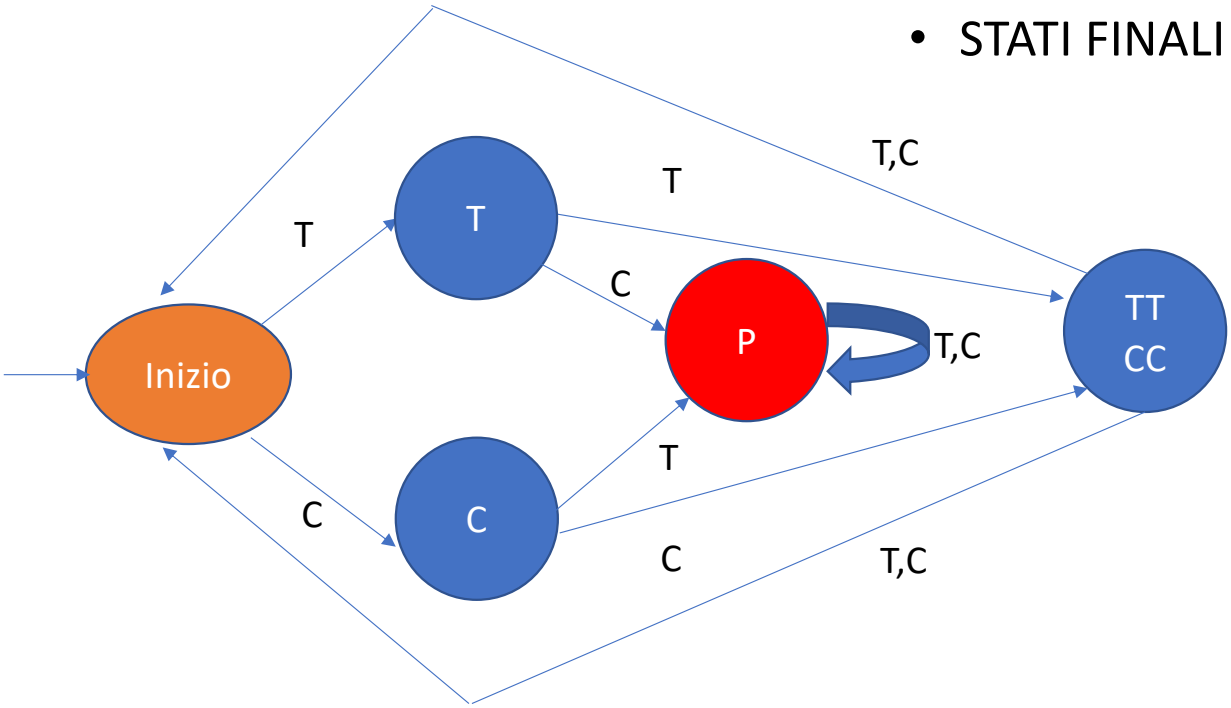


- Sequenza accettata
  TTC CCT
- Sequenza non accettata
  TTC CTC



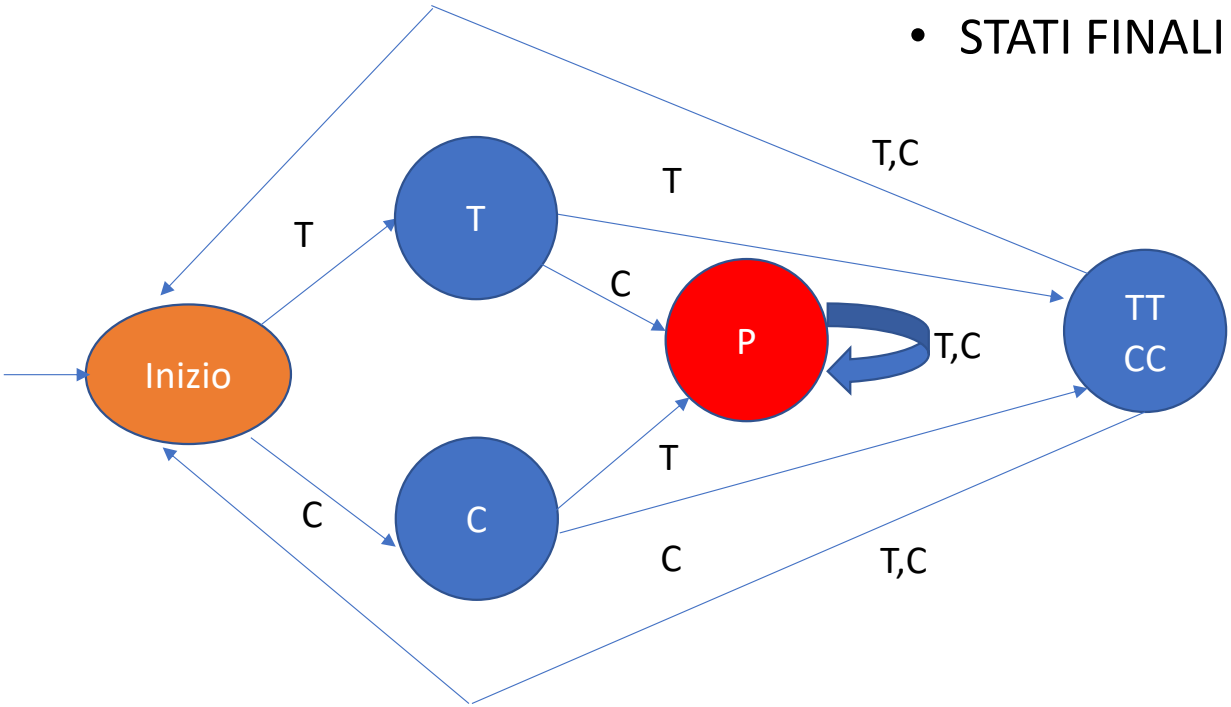
COSA CI SERVE PER DEFINIRE UN AUTOMA?

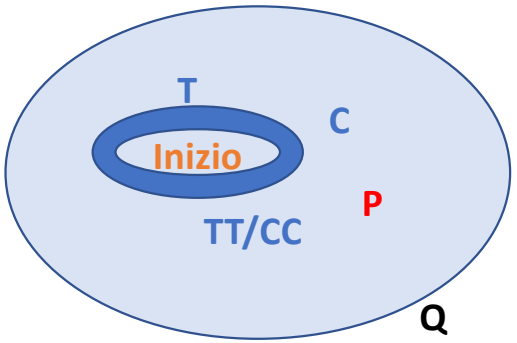
- STATI
- ALFABETO FINITO di simboli
- FUNZIONE (stato,lettera) |--> stato’
- STATO Iniziale
- STATI FINALI



COSA CI SERVE PER DEFINIRE UN AUTOMA: $(Q, \Sigma, \delta, q_0, F)$

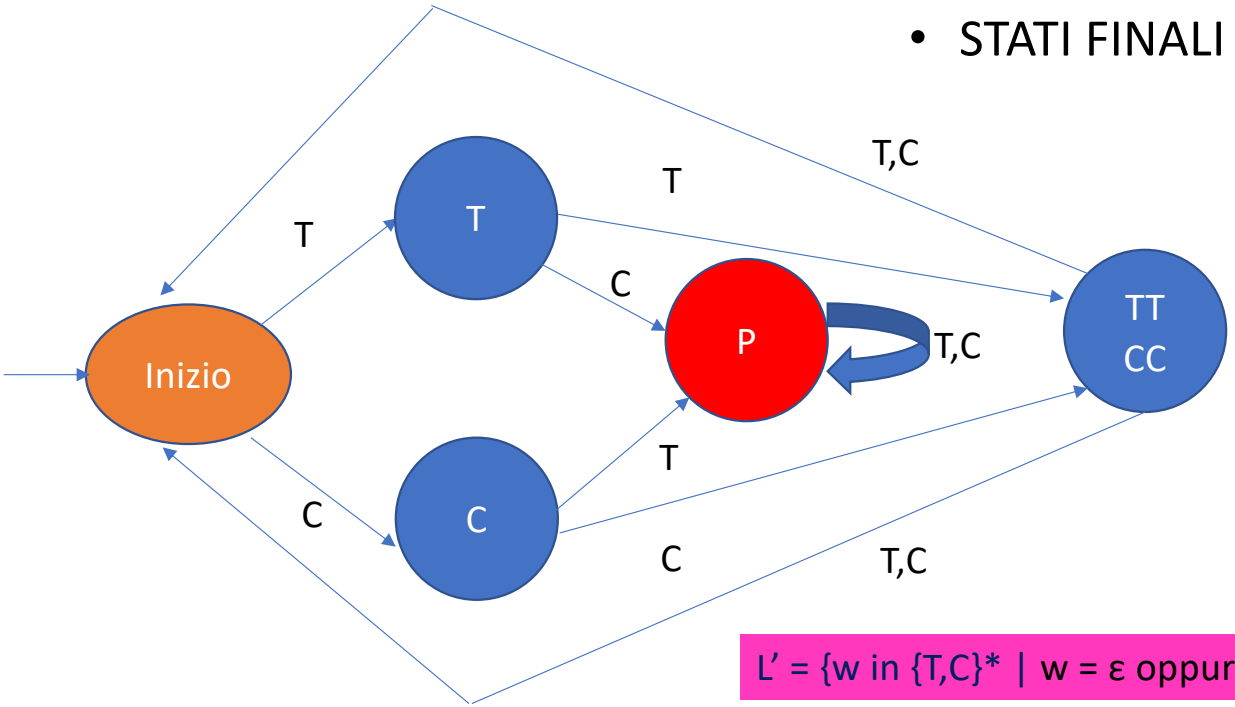
- STATI **Q** = {Inizio, T,C,TT/CC,P}
- ALFABETO FINITO **Σ** = {T,C}
- FUNZIONE: **δ:** Q x Σ → Q $\delta(q, a) = q'$
- STATO Iniziale $\mathbf{q_0} \in Q$ = Inizio
- STATI FINALI **F** $\subseteq$ Q= {Inizio}





COSA CI SERVE PER DEFINIRE UN AUTOMA: (Q, Σ, δ, q_0 , F)

- STATI **Q** = {Inizio, T,C,TT/CC,P}
- ALFABETO FINITO **Σ** = {T,C}
- FUNZIONE: **δ:** Q x Σ → Q (***) δ(q, a) = q
- STATO Iniziale q_0 ∈ Q = Inizio
- STATI FINALI **F** ⊆ Q= {Inizio}

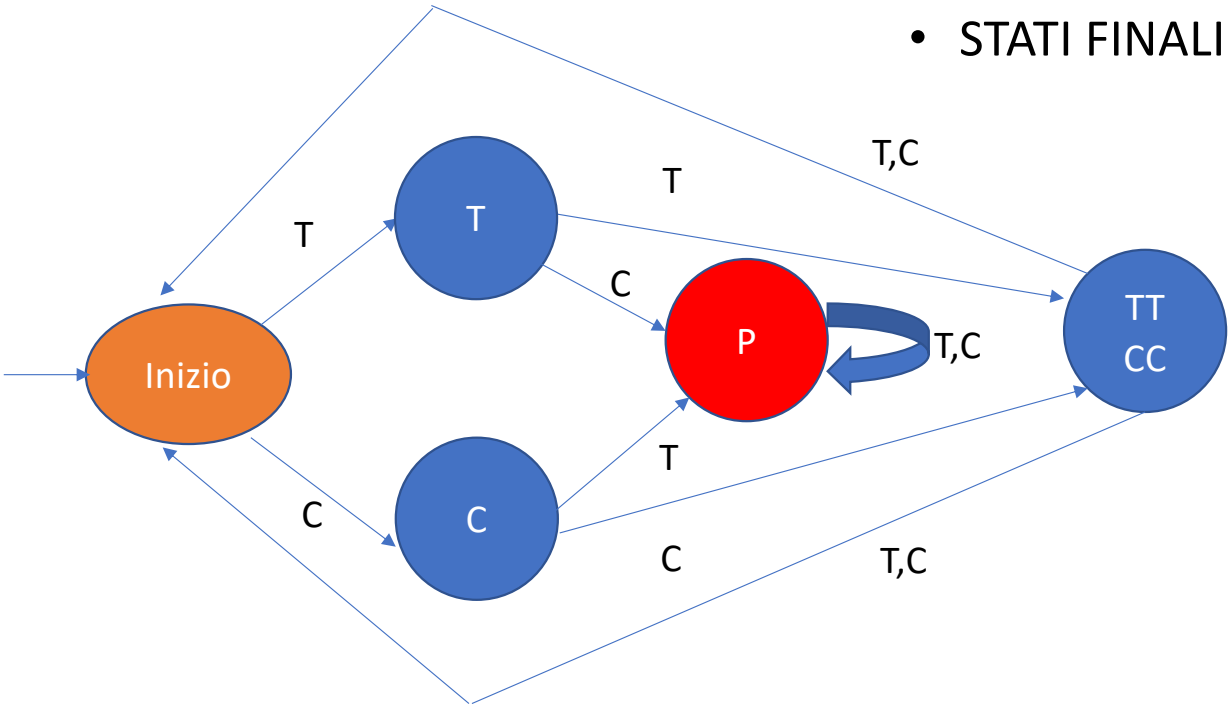


Gli automi riconoscono
linguaggi L ⊆ **Σ***
Nel nostro esempio il linguaggio L'
delle sequenze
di triplette corrette

L' = {w in {T,C}* | w = ε oppure w =w_1…w_k con w_i =xxy e x,y in {T,C}}

COSA CI SERVE PER DEFINIRE UN AUTOMA: (Q, Σ, δ, q_0 , F)

- STATI **Q** = {Inizio, T,C,TT/CC,P}
- ALFABETO FINITO **Σ** = {T,C}
- FUNZIONE: **δ:** Q x Σ → Q
- STATO Iniziale $\mathbf{q_0} \in$ Q = Inizio
- STATI FINALI **F** ⊆ Q= {Inizio}

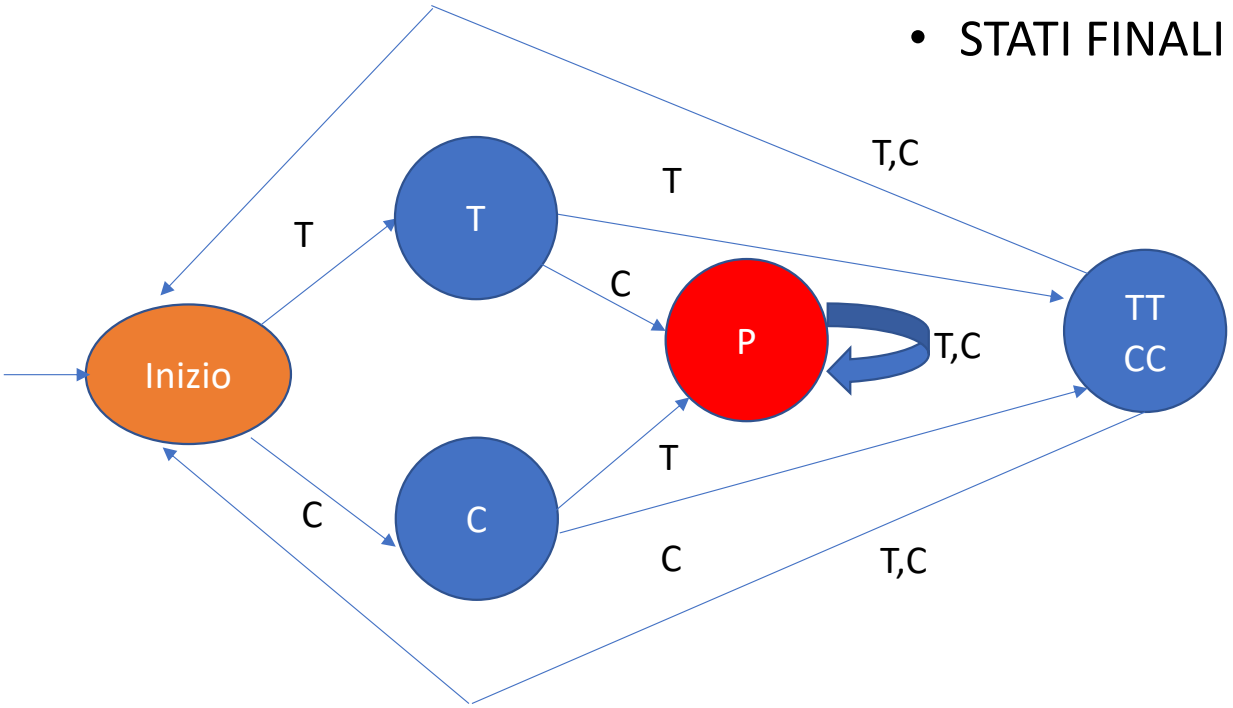


δ: Q x Σ → Q **FUNZIONE DI TRANSIZIONE**

- δ(Inizio, T) = T
- δ(Inizio, C) = C
- δ(T, T) = TT/CC
- δ(T, C) = P
- δ(C, C) = TT/CC
- δ(C, T) = P
- δ(TT/CC, T) = Inizio
- δ(TT/CC, C) = Inizio
- δ(P, C) = P
- δ(P, T) = P

COSA CI SERVE PER DEFINIRE UN AUTOMA: (Q, Σ, δ, q_0 , F)

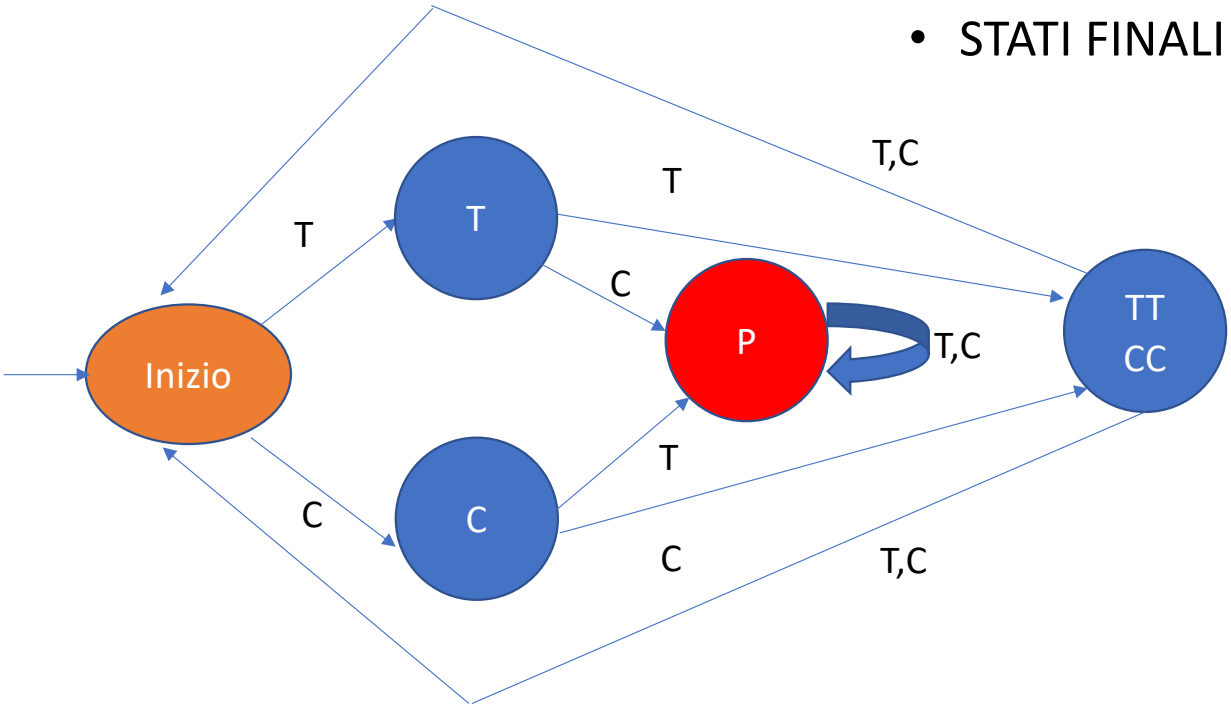
- STATI **Q** = {Inizio, T,C,TT/CC,P}
- ALFABETO FINITO **Σ** = {T,C}
- FUNZIONE: **δ:** Q x Σ → Q
- STATO Iniziale $\mathbf{q_0} \in Q$ = Inizio
- STATI FINALI **F** $\subseteq$ Q= {Inizio}



*	δ	T	C
→	Inizio	T	C
	T	TT/CC	P
	C	P	TT/CC
	TT/CC	Inizio	Inizio
	P	P	P

COSA CI SERVE PER DEFINIRE UN AUTOMA: (Q, Σ, δ, q_0 , F)

- STATI **Q** = {Inizio, T,C,TT/CC,P}
- ALFABETO FINITO **Σ** = {T,C}
- FUNZIONE: **δ:** Q x Σ → Q
- STATO Iniziale $\mathbf{q_0} \in$ Q = Inizio
- STATI FINALI **F** ⊆ Q= {Inizio}

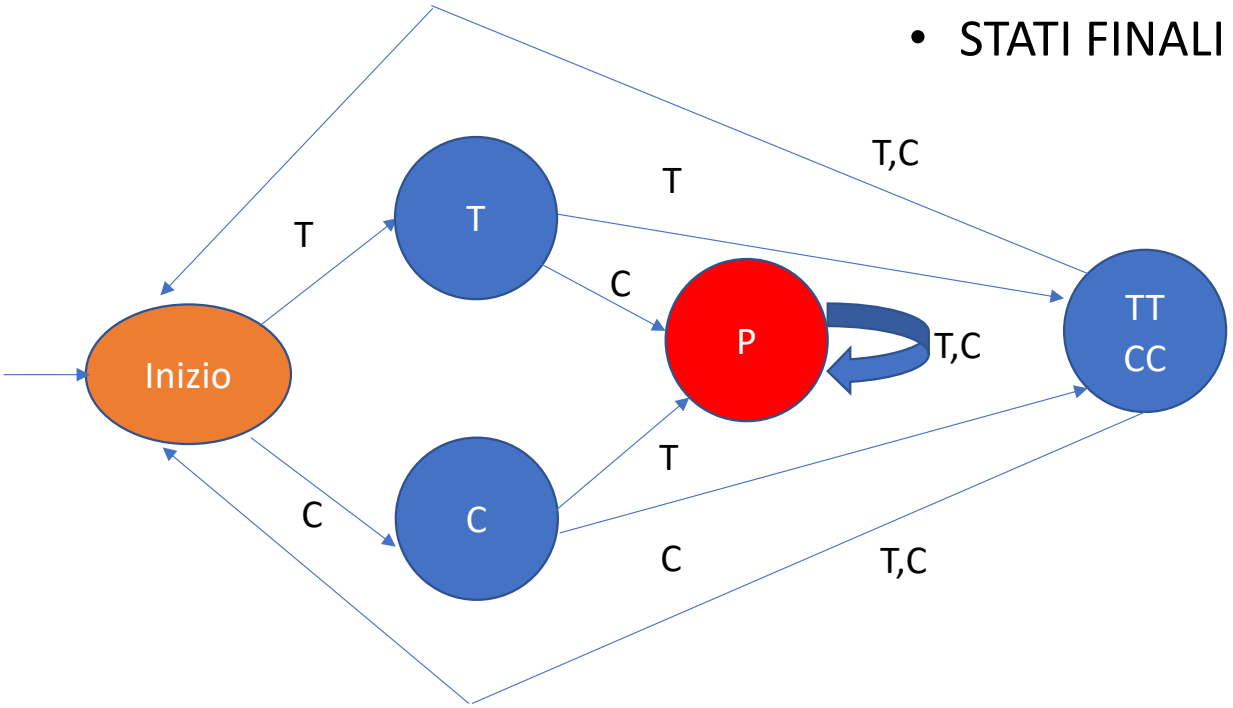


δ: Q x Σ → Q **FUNZIONE DI TRANSIZIONE**

- δ(Inizio, T) = T
- δ(Inizio, C) = C
- δ(T, T) = TT/CC
- δ(T, C) = P
- δ(C, C) = TT/CC
- δ(C, T) = P
- δ(TT/CC, T) = Inizio
- δ(TT/CC, C) = Inizio
- δ(P, C) = P
- δ(P, T) = P

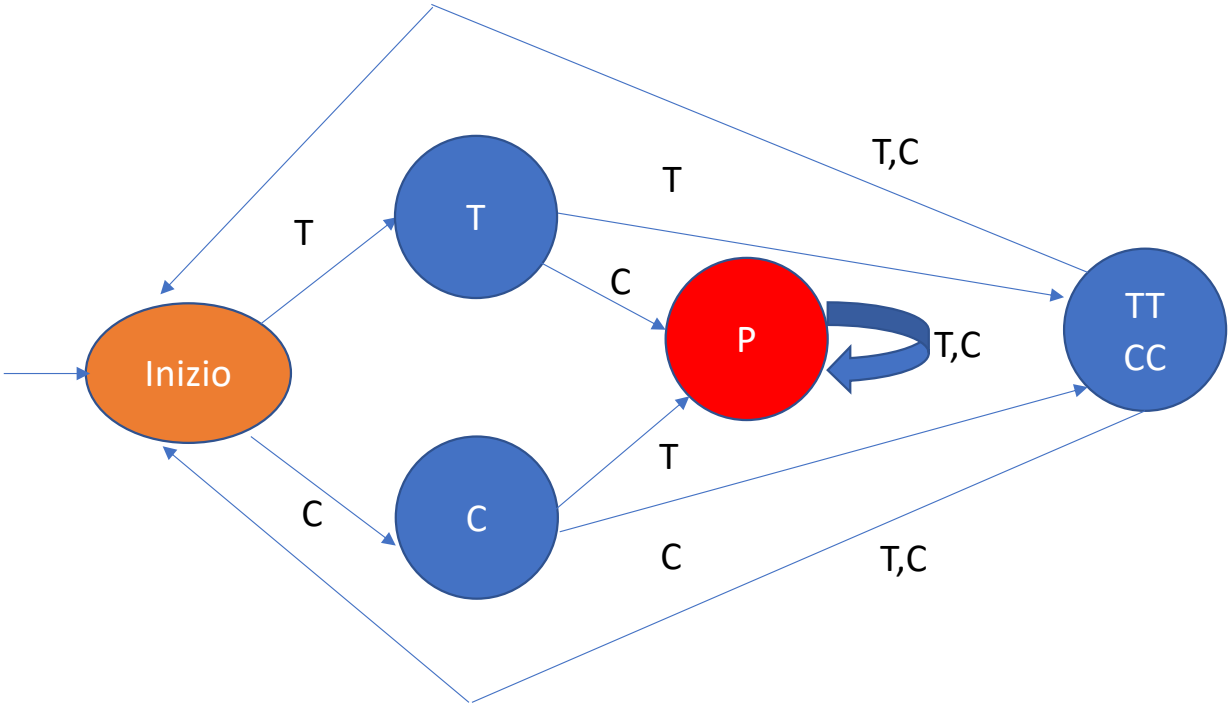
COSA CI SERVE PER DEFINIRE UN AUTOMA: (Q, Σ, δ, q_0 , F)

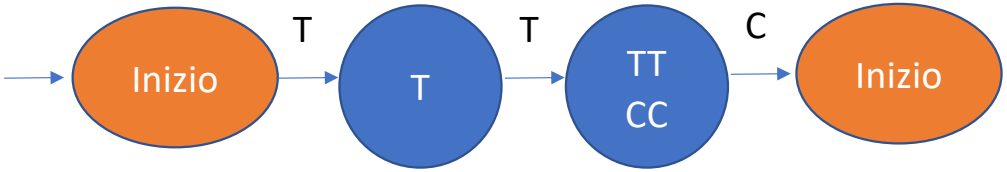
- STATI **Q** = {Inizio, T,C,TT/CC,P}
- ALFABETO FINITO **Σ** = {T,C}
- FUNZIONE: **δ:** Q x Σ → Q
- STATO Iniziale $\mathbf{q_0} \in Q$ = Inizio
- STATI FINALI **F** $\subseteq$ Q= {Inizio}



*	δ	T	C
→	Inizio	T	C
	T	TT/CC	P
	C	P	TT/CC
	TT/CC	Inizio	Inizio
	P	P	P

FUNZIONE DI TRANSIZIONE ESTESA: definizione induttiva





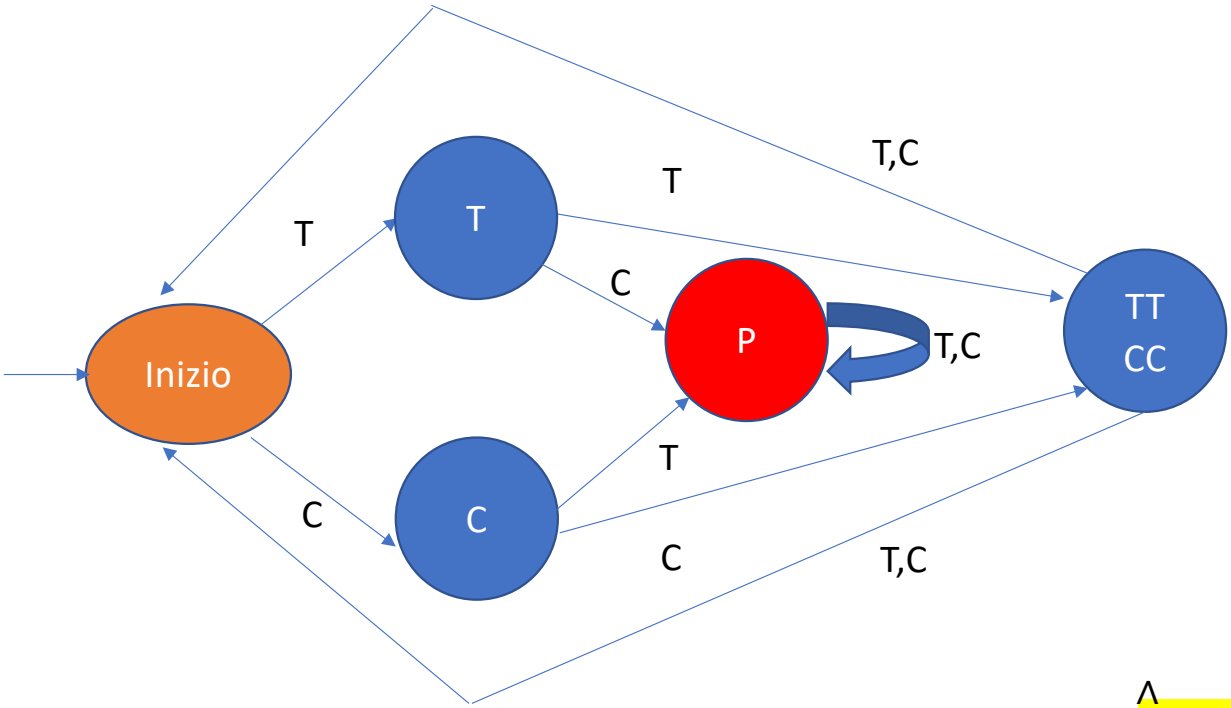
$\hat{\delta}: Q \times \Sigma^* \rightarrow Q$

Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$
$x \text{ in } \Sigma^*, a \text{ in } \Sigma$

- La funzione di transizione estesa $\hat{\delta}$ opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(q,w) = p$ denota che p è lo stato raggiunto a partire da q quando si riceve in input uno a uno i simboli di w

FUNZIONE DI TRANSIZIONE ESTESA: definizione induttiva



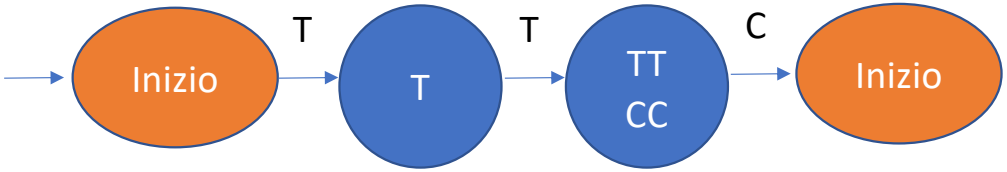
$\hat{\delta}$: Q x Σ* → Q

Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

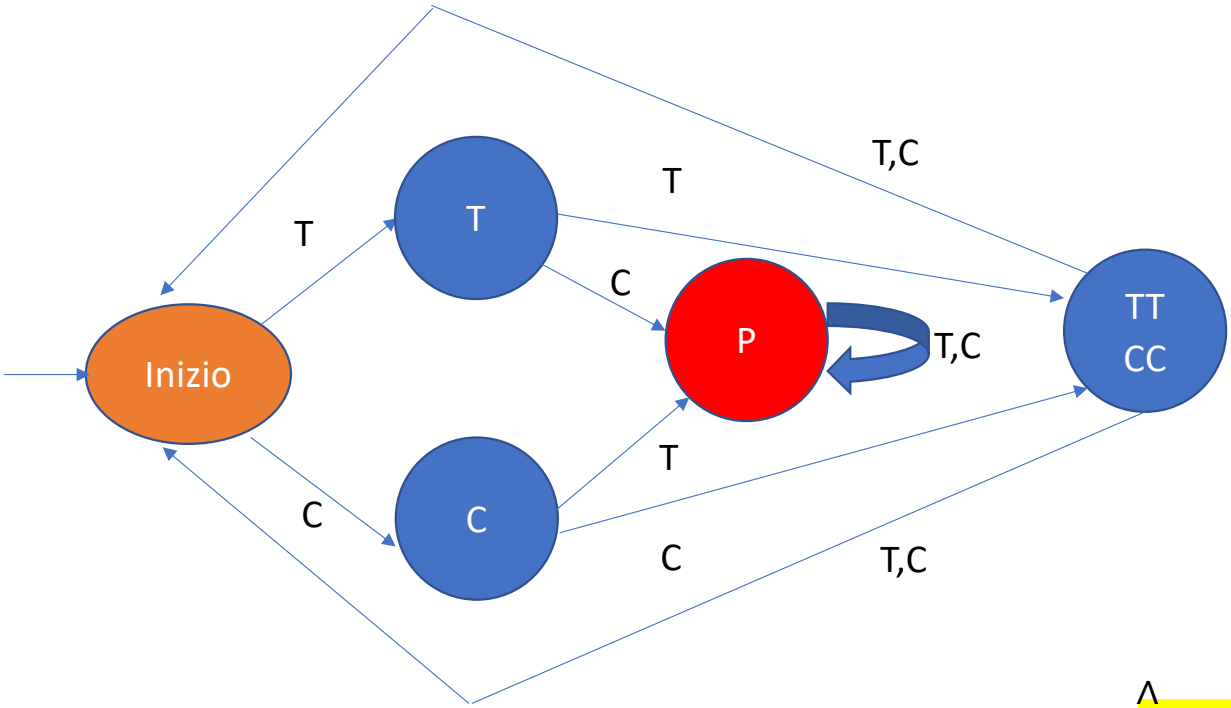
Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$
x in Σ*, a in Σ

- La funzione di transizione estesa $\hat{\delta}$ opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(1,w) = p$ denota che p è lo stato raggiunto a partire da q quando si riceve in input uno a uno i simboli di w

$\hat{\delta}$(Inizio, TTC) =



FUNZIONE DI TRANSIZIONE ESTESA: definizione induttiva



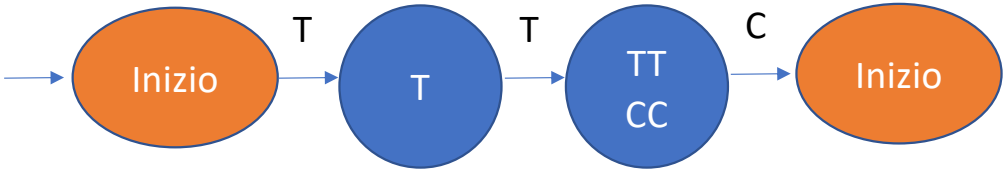
$\hat{\delta}: Q \times \Sigma^* \rightarrow Q$

Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

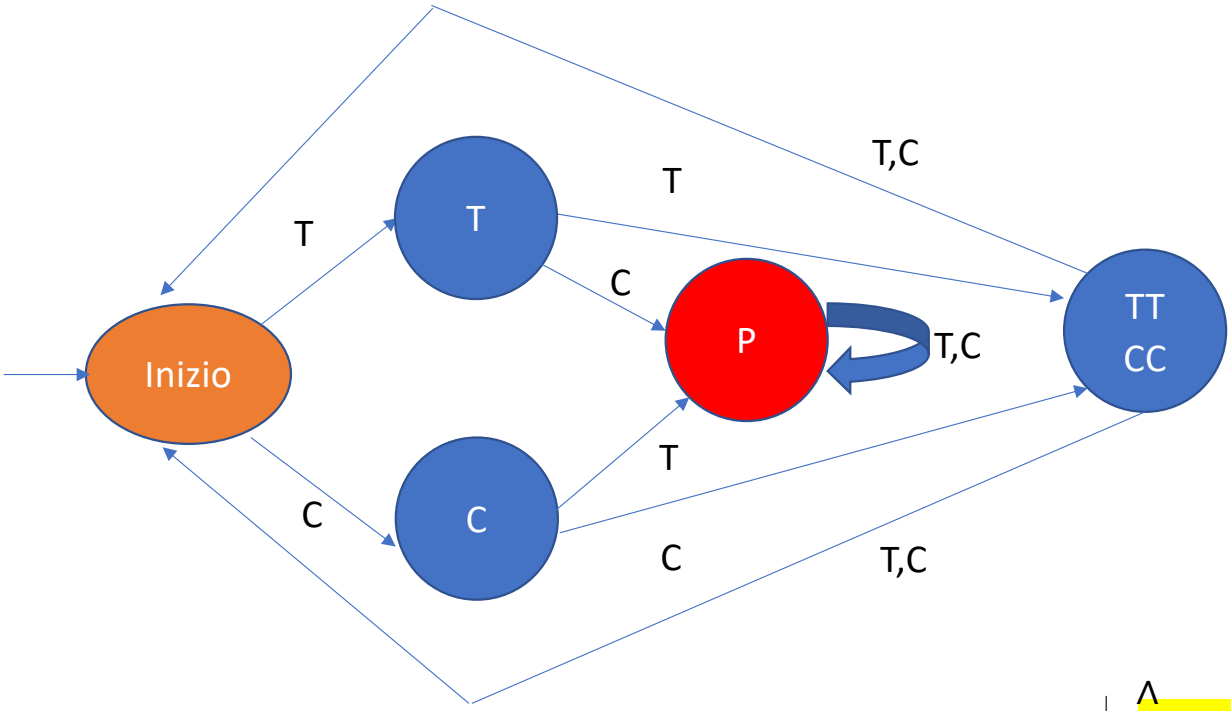
Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$
$x \text{ in } \Sigma^*, a \text{ in } \Sigma$

- La funzione di transizione estesa $\hat{\delta}$ opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(1,w) = p$ denota che p è lo stato raggiunto a partire da q quando si riceve in input uno a uno i simboli di w

$\hat{\delta}(\text{Inizio}, TTC) = \delta(\hat{\delta}(\text{Inizio}, TT), C)$



FUNZIONE DI TRANSIZIONE ESTESA: definizione induttiva



$\hat{\delta}: Q \times \Sigma^* \rightarrow Q$

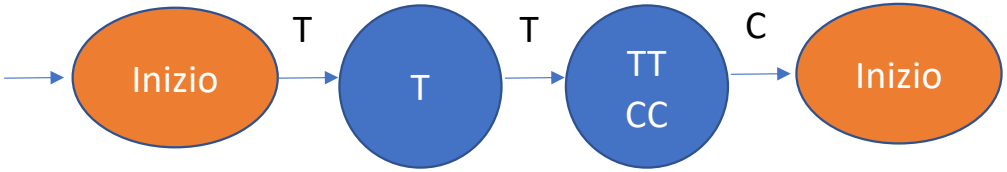
Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$
$x \text{ in } \Sigma^*, a \text{ in } \Sigma$

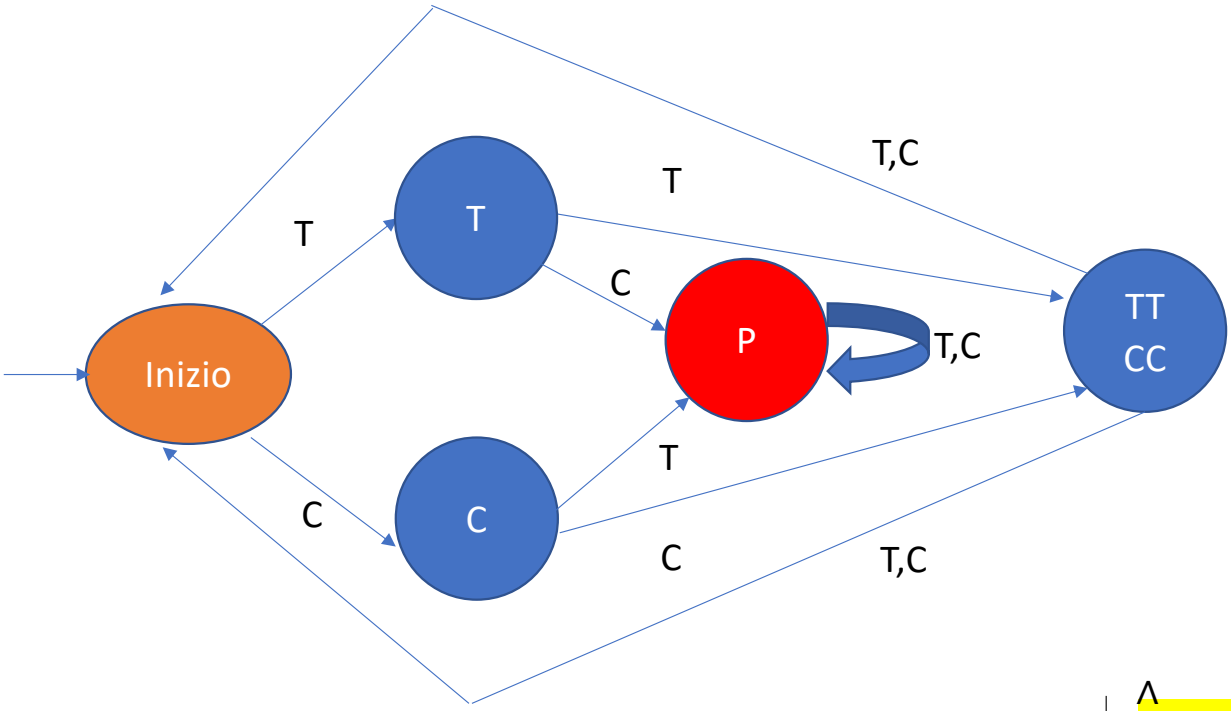
- La funzione di transizione estesa $\hat{\delta}$ opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(1,w) = p$ denota che p è lo stato raggiunto a partire da q quando si riceve in input uno a uno i simboli di w

$\hat{\delta}(\text{Inizio}, TTC) = \delta(\hat{\delta}(\text{Inizio}, TT), C)$

$\hat{\delta}(\text{Inizio}, TT) =$



FUNZIONE DI TRANSIZIONE ESTESA: definizione induttiva



$\hat{\delta}$: Q x Σ* → Q

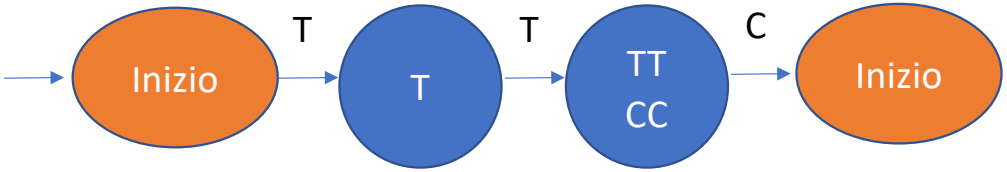
Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$
x in Σ*, a in Σ

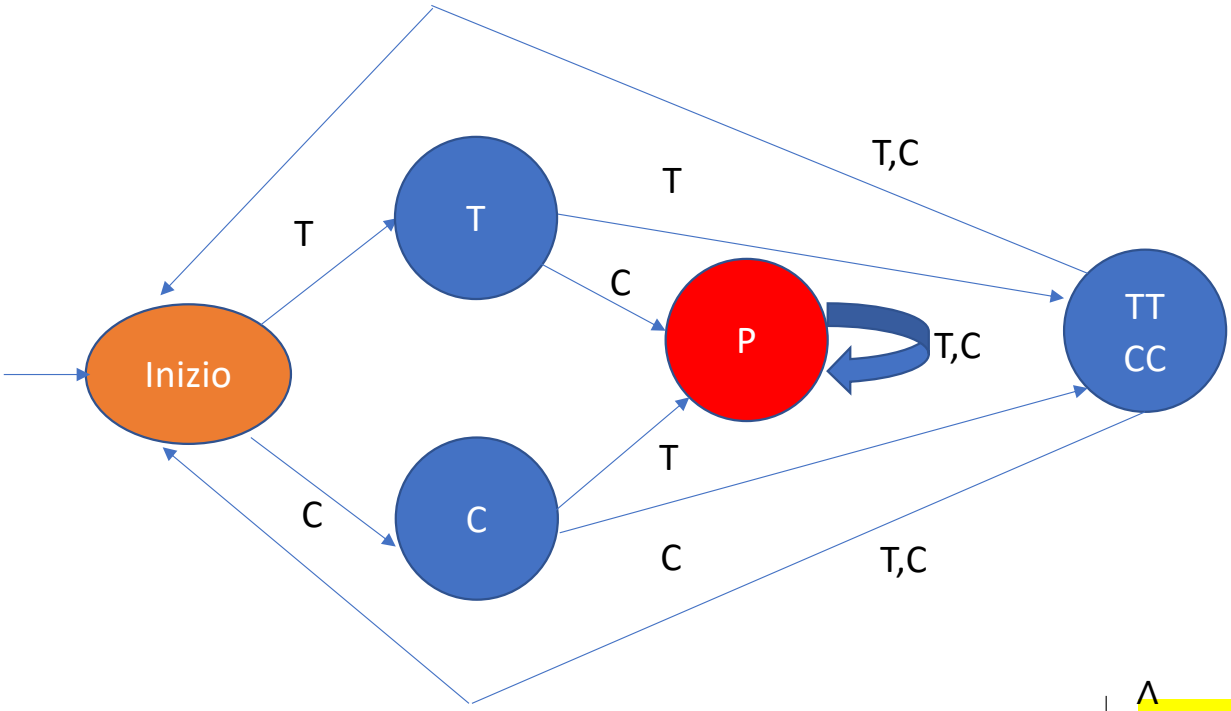
- La funzione di transizione estesa $\hat{\delta}$ opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(1,w) = p$ denota che p è lo stato raggiunto a partire da q quando si riceve in input uno a uno i simboli di w

$\hat{\delta}$(Inizio, TTC) = $\delta(\hat{\delta}$(Inizio, TT),C)

$\hat{\delta}$(Inizio, TT) = $\delta(\hat{\delta}$(Inizio, T),T)



FUNZIONE DI TRANSIZIONE ESTESA: definizione induttiva

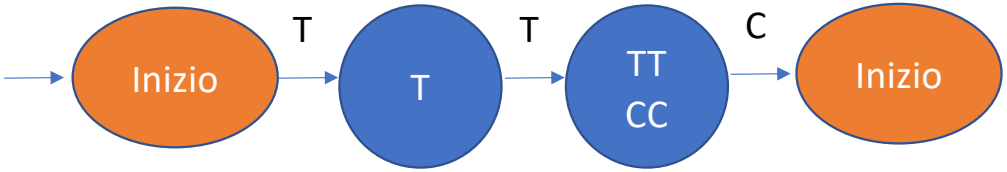


$\hat{\delta}: Q \times \Sigma^* \rightarrow Q$

Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$
$x \text{ in } \Sigma^*, a \text{ in } \Sigma$

- La funzione di transizione estesa $\hat{\delta}$ opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(1,w) = p$ denota che p è lo stato raggiunto a partire da q quando si riceve in input uno a uno i simboli di w

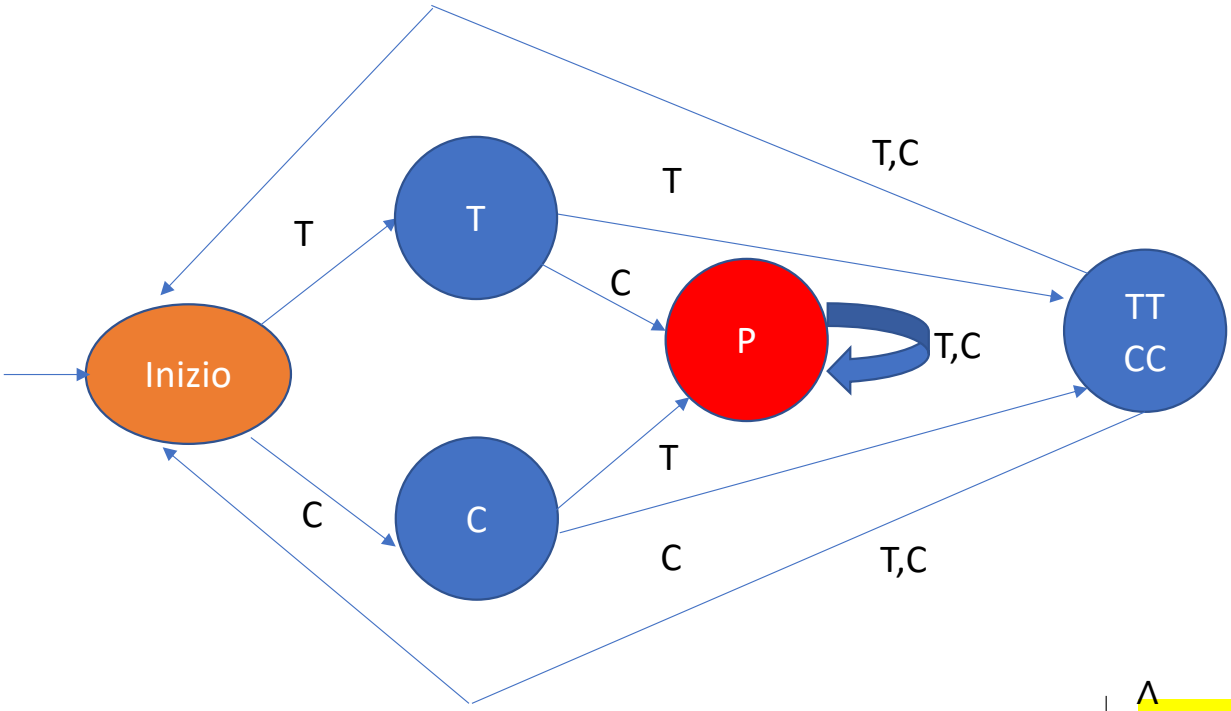


$\hat{\delta}(\text{Inizio}, TTC) = \delta(\hat{\delta}(\text{Inizio}, TT), C)$

$\hat{\delta}(\text{Inizio}, TT) = \delta(\hat{\delta}(\text{Inizio}, T), T)$

$\hat{\delta}(\text{Inizio}, T) =$

FUNZIONE DI TRANSIZIONE ESTESA: definizione induttiva

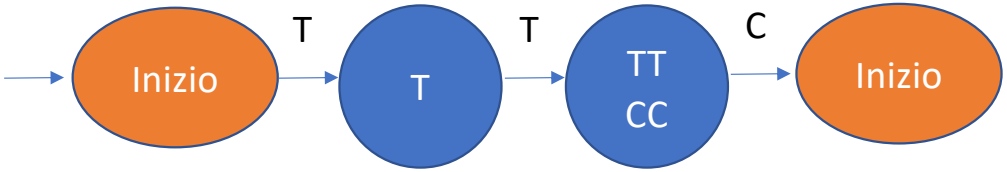


$\hat{\delta}: Q \times \Sigma^* \rightarrow Q$

Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$
$x \text{ in } \Sigma^*, a \text{ in } \Sigma$

- La funzione di transizione estesa $\hat{\delta}$ opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(1,w) = p$ denota che p è lo stato raggiunto a partire da q quando si riceve in input uno a uno i simboli di w

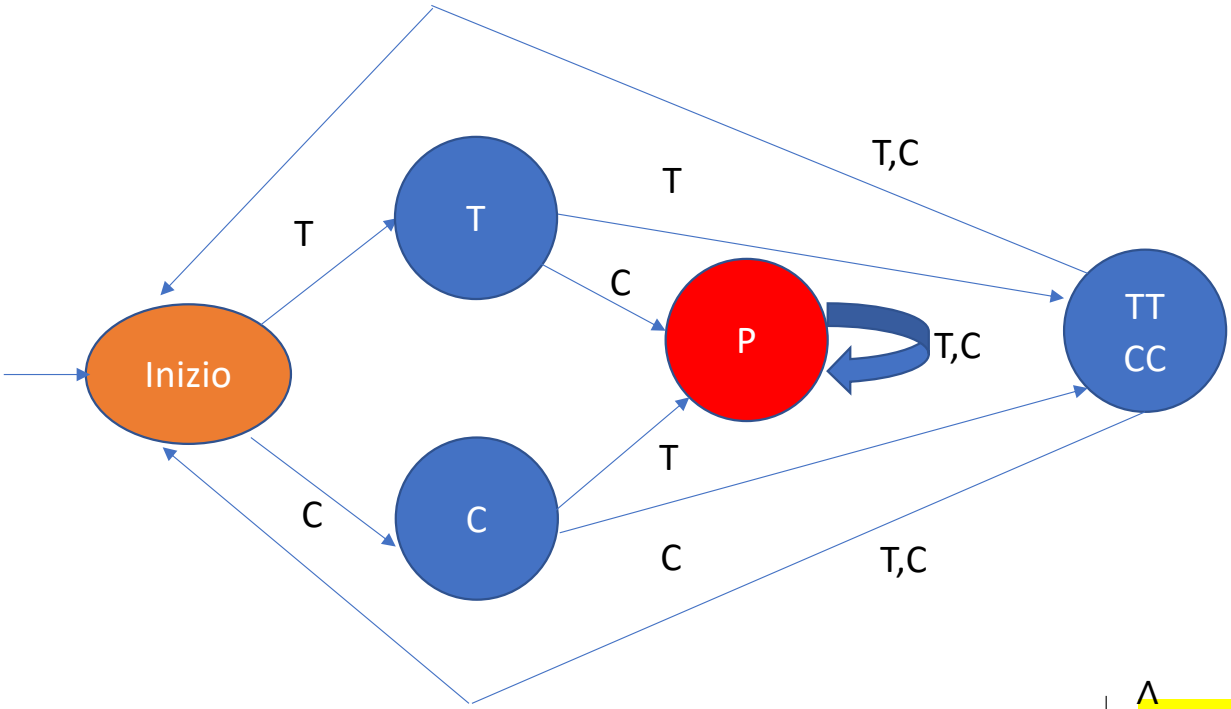


$\hat{\delta}(\text{Inizio}, TTC) = \delta(\hat{\delta}(\text{Inizio}, TT), C)$

$\hat{\delta}(\text{Inizio}, TT) = \delta(\hat{\delta}(\text{Inizio}, T), T)$

$\hat{\delta}(\text{Inizio}, T) = \delta(\hat{\delta}(\text{Inizio}, \varepsilon), T)$

FUNZIONE DI TRANSIZIONE ESTESA: definizione induttiva

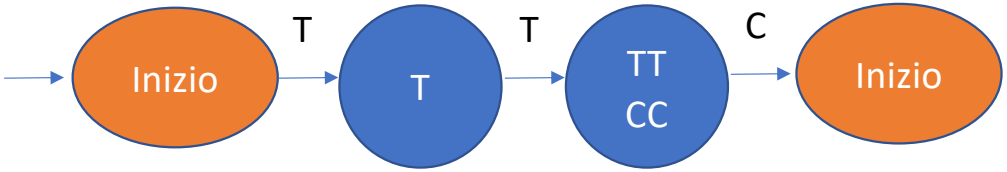


$\hat{\delta}: Q \times \Sigma^* \rightarrow Q$

Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$
$x \text{ in } \Sigma^*, a \text{ in } \Sigma$

- La funzione di transizione estesa $\hat{\delta}$ opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(1,w) = p$ denota che p è lo stato raggiunto a partire da q quando si riceve in input uno a uno i simboli di w



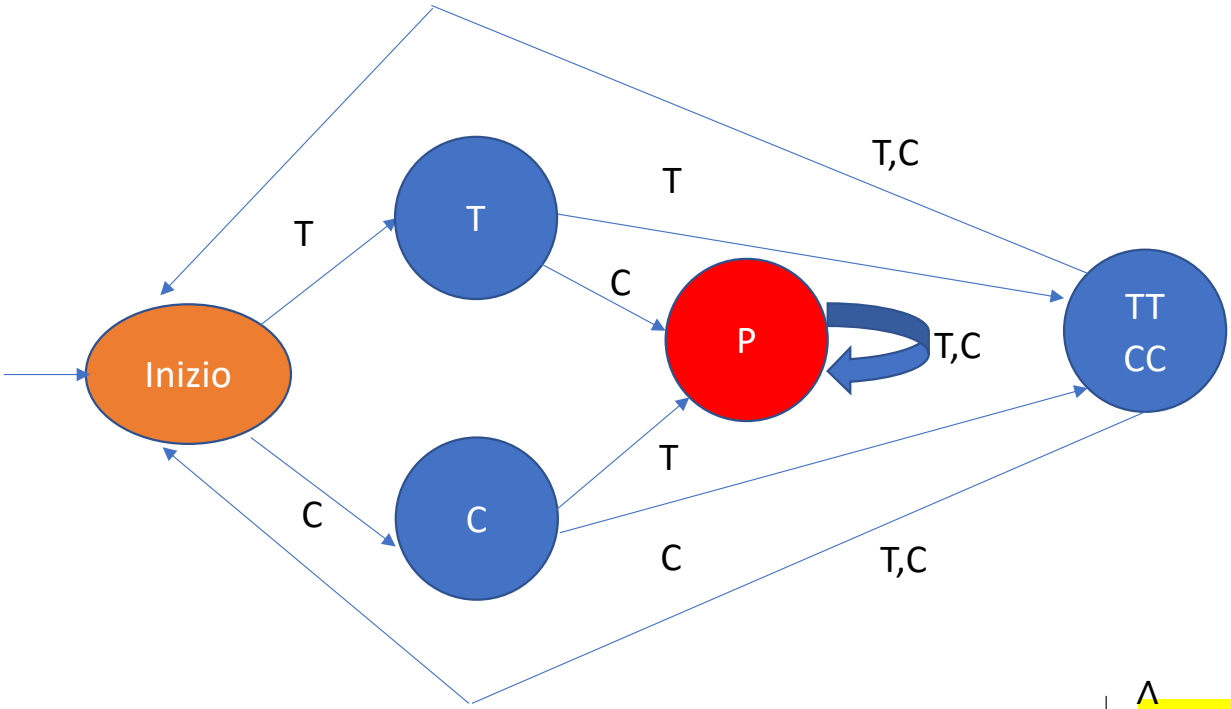
$\hat{\delta}(\text{Inizio}, TTC) = \delta(\hat{\delta}(\text{Inizio}, TT), C)$

$\hat{\delta}(\text{Inizio}, TT) = \delta(\hat{\delta}(\text{Inizio}, T), T)$

$\hat{\delta}(\text{Inizio}, T) = \delta(\hat{\delta}(\text{Inizio}, \varepsilon), T)$

$\hat{\delta}(\text{Inizio}, \varepsilon) =$

FUNZIONE DI TRANSIZIONE ESTESA: definizione induttiva

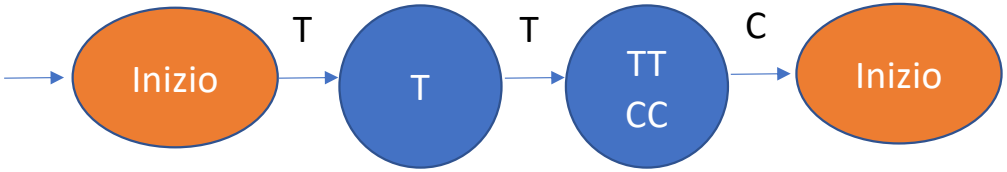


$\hat{\delta}: Q \times \Sigma^* \rightarrow Q$

Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$
$x \text{ in } \Sigma^*, a \text{ in } \Sigma$

- La funzione di transizione estesa $\hat{\delta}$ opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(1,w) = p$ denota che p è lo stato raggiunto a partire da q quando si riceve in input uno a uno i simboli di w



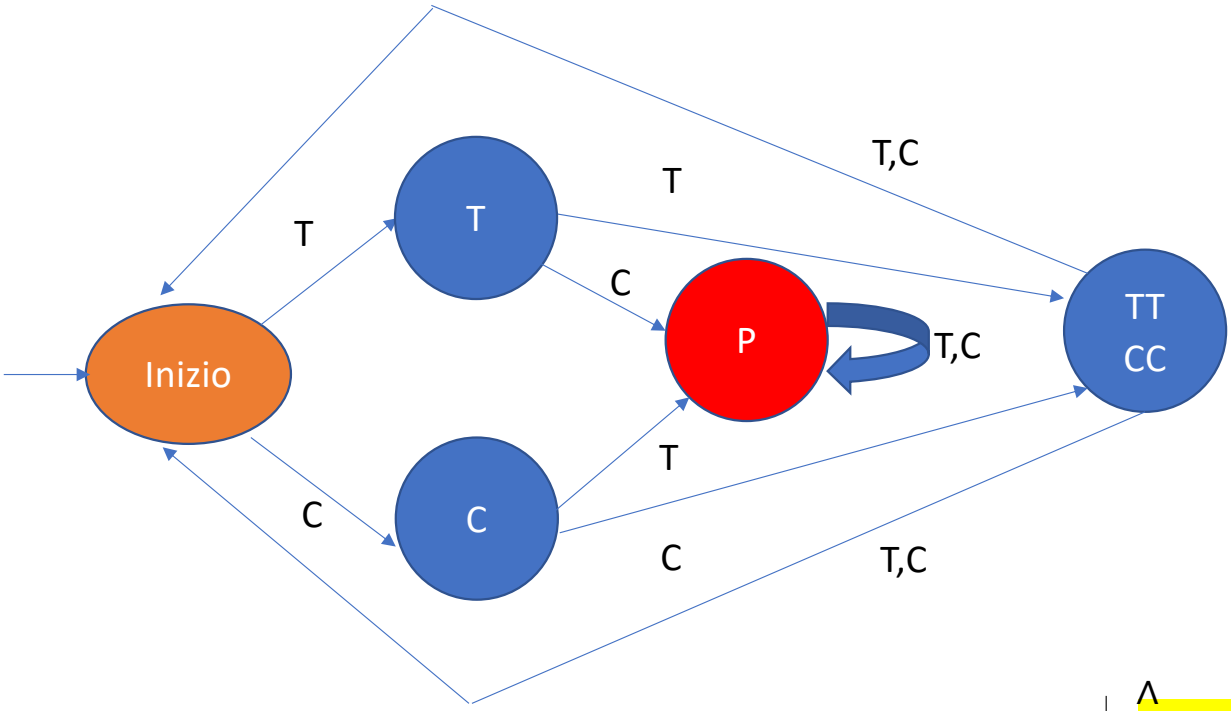
$\hat{\delta}(\text{Inizio}, TTC) = \delta(\hat{\delta}(\text{Inizio}, TT), C)$

$\hat{\delta}(\text{Inizio}, TT) = \delta(\hat{\delta}(\text{Inizio}, T), T)$

$\hat{\delta}(\text{Inizio}, T) = \delta(\hat{\delta}(\text{Inizio}, \varepsilon), T)$

$\hat{\delta}(\text{Inizio}, \varepsilon) = \text{Inizio}$

FUNZIONE DI TRANSIZIONE ESTESA: definizione induttiva

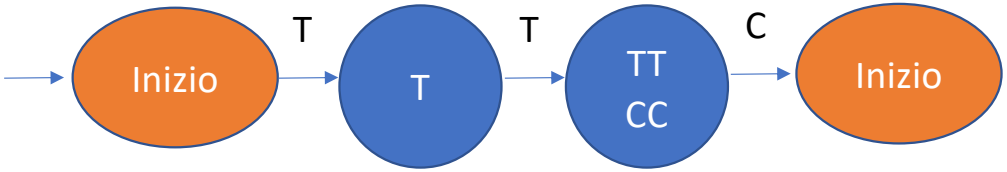


$\hat{\delta}: Q \times \Sigma^* \rightarrow Q$

Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$
$x \text{ in } \Sigma^*, a \text{ in } \Sigma$

- La funzione di transizione estesa $\hat{\delta}$ opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(1,w) = p$ denota che p è lo stato raggiunto a partire da q quando si riceve in input uno a uno i simboli di w



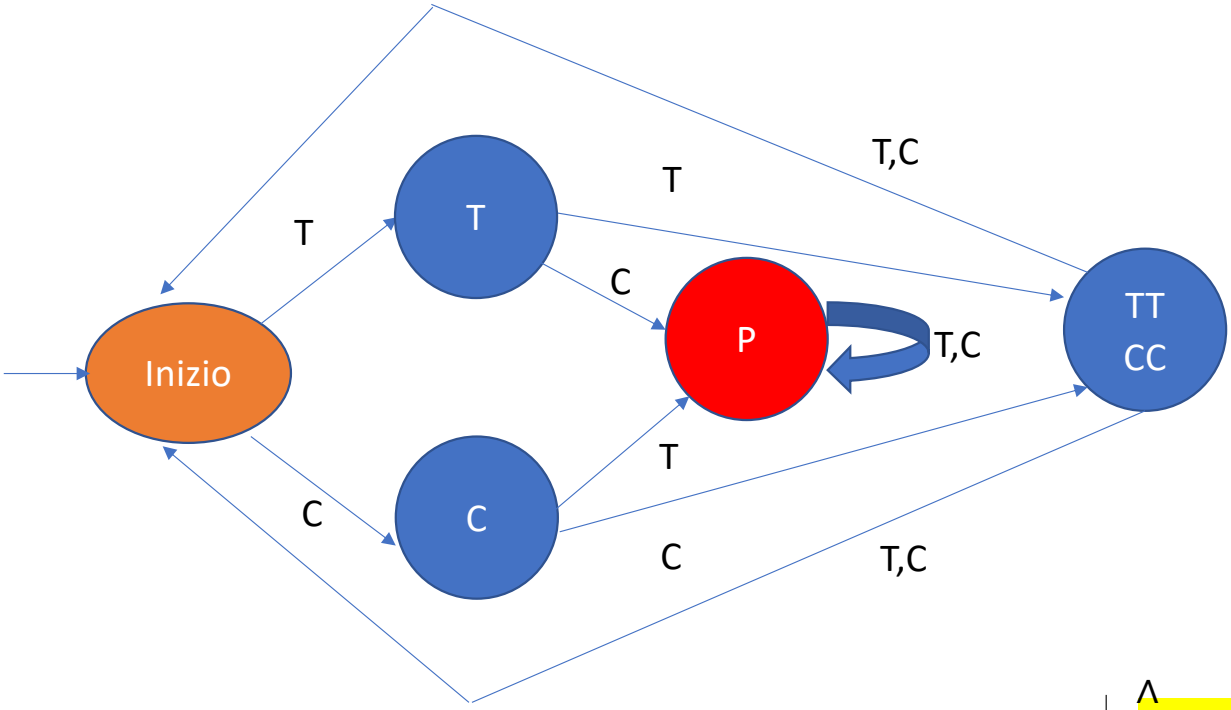
$\hat{\delta}(\text{Inizio}, TTC) = \delta(\hat{\delta}(\text{Inizio}, TT), C)$

$\hat{\delta}(\text{Inizio}, TT) = \delta(\hat{\delta}(\text{Inizio}, T), T)$

$\hat{\delta}(\text{Inizio}, T) = \delta(\hat{\delta}(\text{Inizio}, \varepsilon), T) = \delta(\text{Inizio}, T) = T$

$\hat{\delta}(\text{Inizio}, \varepsilon) = \text{Inizio}$

FUNZIONE DI TRANSIZIONE ESTESA: definizione induttiva

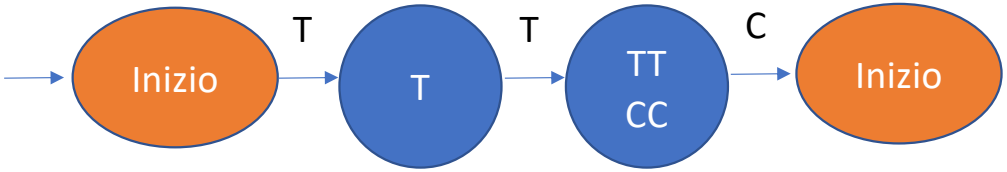


$\hat{\delta}: Q \times \Sigma^* \rightarrow Q$

Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$
$x \text{ in } \Sigma^*, a \text{ in } \Sigma$

- La funzione di transizione estesa $\hat{\delta}$ opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(1,w) = p$ denota che p è lo stato raggiunto a partire da q quando si riceve in input uno a uno i simboli di w



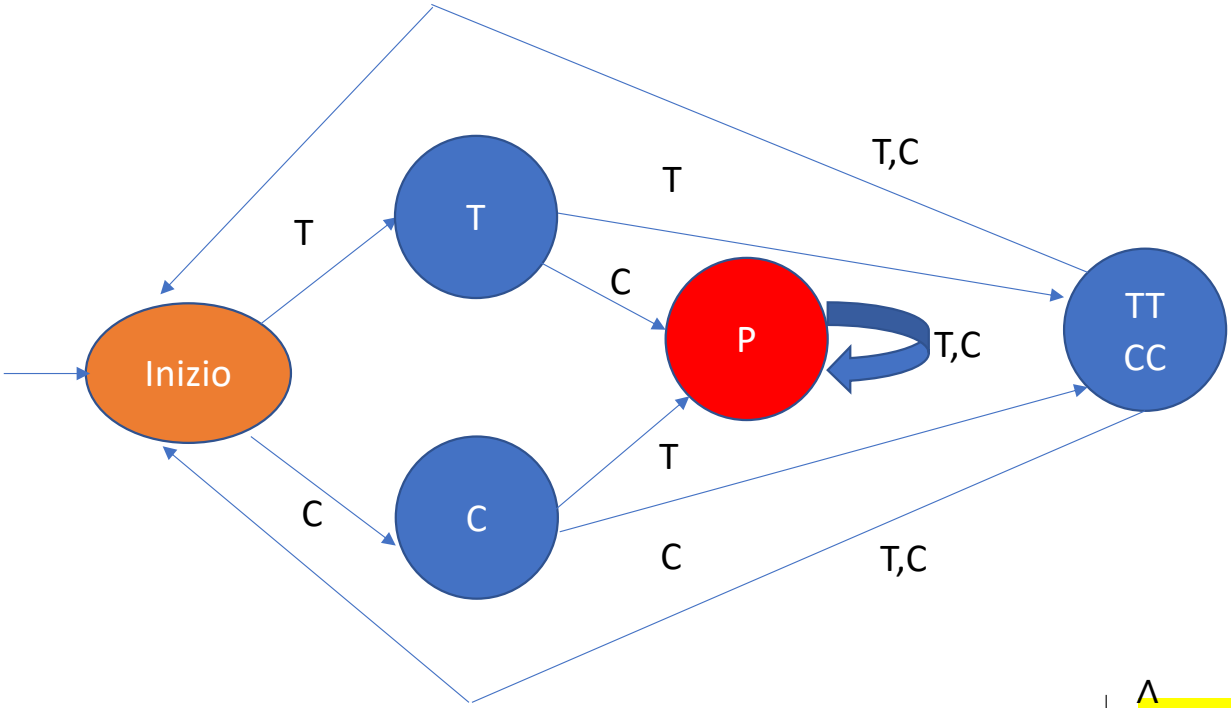
$\hat{\delta}(\text{Inizio}, TTC) = \delta(\hat{\delta}(\text{Inizio}, TT), C)$

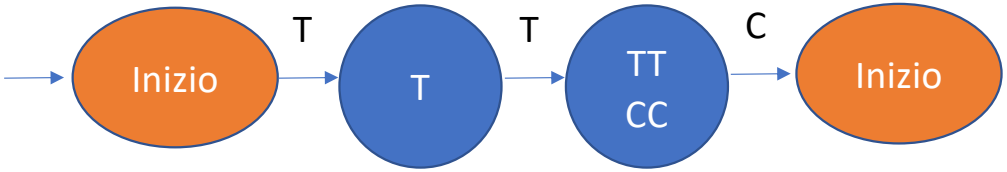
$\hat{\delta}(\text{Inizio}, TT) = \delta(\hat{\delta}(\text{Inizio}, T), T) = \delta(T, T) = TT/CC$

$\hat{\delta}(\text{Inizio}, T) = \delta(\hat{\delta}(\text{Inizio}, \varepsilon), T) = \delta(\text{Inizio}, T) = T$

$\hat{\delta}(\text{Inizio}, \varepsilon) = \text{Inizio}$

FUNZIONE DI TRANSIZIONE ESTESA: definizione induttiva





$\hat{\delta}: Q \times \Sigma^* \rightarrow Q$

Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$
$x \text{ in } \Sigma^*, a \text{ in } \Sigma$

- La funzione di transizione estesa $\hat{\delta}$ opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(1,w) = p$ denota che p è lo stato raggiunto a partire da q quando si riceve in input uno a uno i simboli di w

$\hat{\delta}(\text{Inizio}, \text{TTC}) = \delta(\hat{\delta}(\text{Inizio}, \text{TT}), \text{C}) = \delta(\text{TT/CC}, \text{T}) = \text{Inizio}$

$\hat{\delta}(\text{Inizio}, \text{TT}) = \delta(\hat{\delta}(\text{Inizio}, \text{T}), \text{T}) = \delta(\text{T}, \text{T}) = \text{TT/CC}$

$\hat{\delta}(\text{Inizio}, \text{T}) = \delta(\hat{\delta}(\text{Inizio}, \varepsilon), \text{T}) = \delta(\text{Inizio}, \text{T}) = \text{T}$

$\hat{\delta}(\text{Inizio}, \varepsilon) = \text{Inizio}$

Definizioni induttive

$\hat{\delta}$

Base: $\hat{\delta}(\mathbf{q}, \varepsilon) = \mathbf{q}$

Induzione: $\hat{\delta}(\mathbf{q}, xa) = \delta(\hat{\delta}(\mathbf{q}, x), a)$

$x \text{ in } \Sigma^*, a \text{ in } \Sigma$

$\hat{\delta}(\text{Inizio}, TTC) = \delta(\hat{\delta}(\text{Inizio}, TT), C) = \delta(\text{TT/CC}, T) = \text{Inizio}$

$\hat{\delta}(\text{Inizio}, TT) = \delta(\hat{\delta}(\text{Inizio}, T), T) = \delta(T, T) = \text{TT/CC}$

$\hat{\delta}(\text{Inizio}, T) = \delta(\hat{\delta}(\text{Inizio}, \varepsilon), T) = \delta(\text{Inizio}, T) = T$

$\hat{\delta}(\text{Inizio}, \varepsilon) = \text{Inizio}$

Fattoriale

Base: 0! = 1

Induzione: n! = n x (n-1)!

$3! = 3 \times 2! = 3 \times 2 = 6$

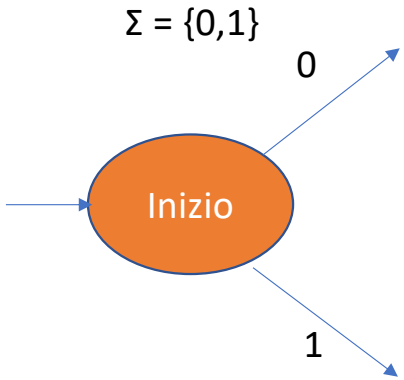
$2! = 2 \times 1! = 2 \times 1 = 2$

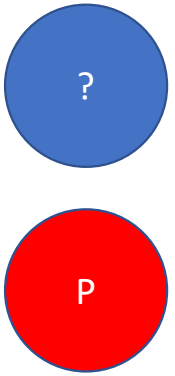
$1! = 1 \times 0! = 1 \times 1 = 1$

$0! = 1$

PROBLEMA 4 Trovare il DFA che accetta tutte e sole le stringhe con un numero pari di zeri e un numero pari di uni (compresa quindi la stringa vuota)

$$L_{pari} = \{w \text{ in } \{0,1\}^* \mid w = \varepsilon \text{ oppure } |w|_0 \text{ e } |w|_1 \text{ sono pari}\}$$

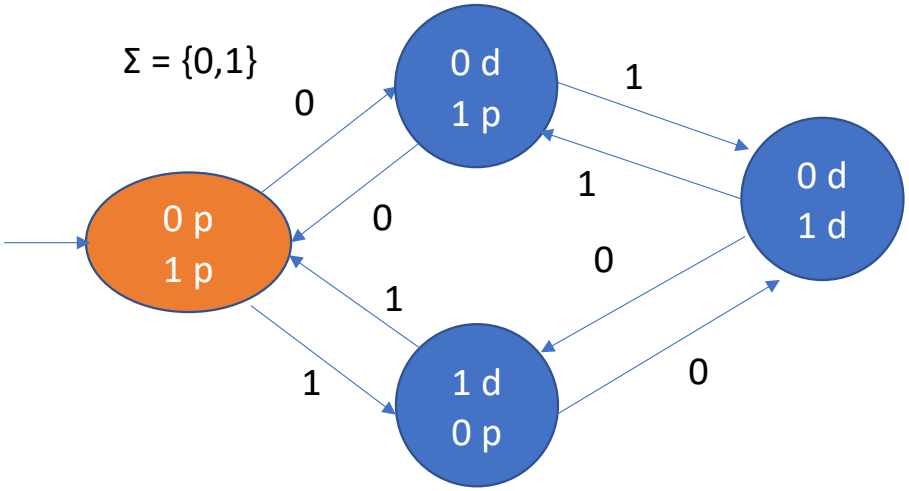




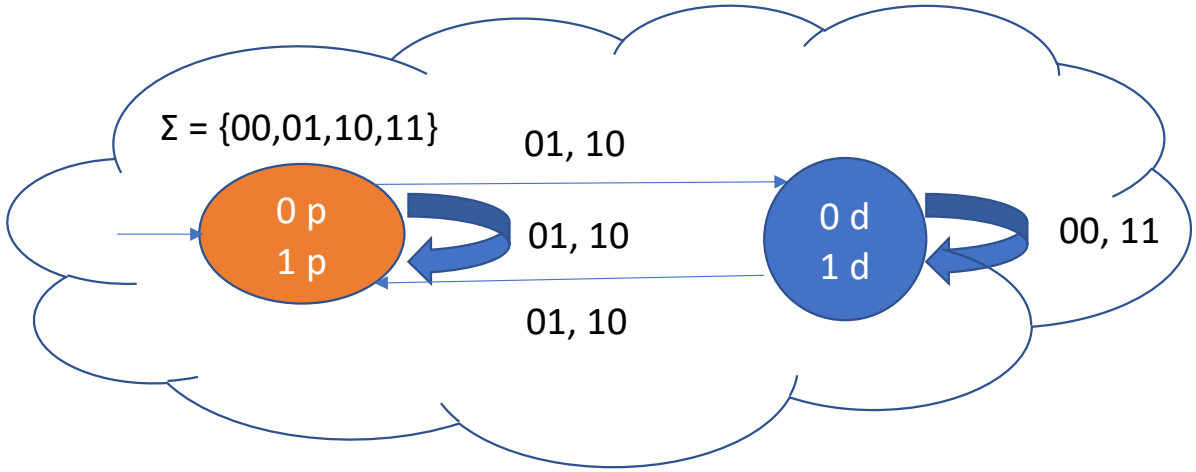
w = 0101 viene accettata

PROBLEMA 4 Trovare il DFA che accetta tutte e sole le stringhe con un numero pari di zeri e un numero pari di uni (compresa quindi la stringa vuota)

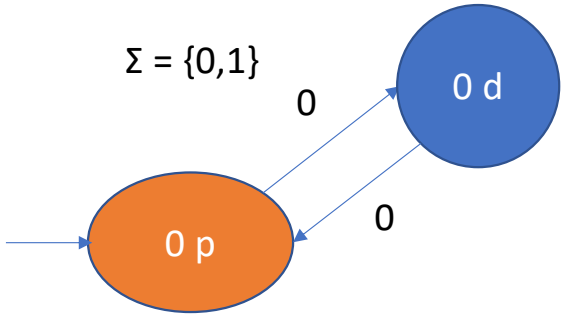
$L_{pari} = \{w \text{ in } \{0,1\}^* \mid w = \varepsilon \text{ oppure } |w|_0 \text{ e } |w|_1 \text{ sono pari}\}$



w = 0101 viene accettata



PROBLEMA 5 Trovare il programma C che (i) legge una sequenza di 0 fino ad un numero che indica la fine (ad esempio 3) e (ii) restituisce **true** se la sequenza ha un numero pari di zeri e **false** altrimenti.



w = 0101

Confrontate le due soluzioni

```
…
{
PariZeri = 0;
scanf("%d",&corrente);
while (corrente != 3)
 {
   scanf(…, &corrente);
   if corrente == 0
   PariZeri = !PariZeri;
  }
return PariZeri;
}
```

```
…
{
somma = 0;
scanf("%d",&corrente);
while (corrente != 3)
  {
  scanf(…, &corrente);
   somma == somma + corrente;
  }
PariZeri = pari(somma);
return PariZeri;
}
```

Il problema è aritmetico o booleano?

Concetti di base

- **Alfabeto**: Insieme finito e non vuoto di simboli
  - Esempio: $\Sigma = \{0, 1\}$ alfabeto binario
  - Esempio: $\Sigma = \{a, b, c, \ldots, z\}$ insieme di tutte le lettere minuscole
  - Esempio: Insieme di tutti i caratteri ASCII
- **Stringa:** Sequenza finita di simboli presi da un alfabeto Σ, per es. 0011001 (i simboli li scriviamo di seguito)
- **Stringa vuota:** La stringa con zero occorrenze di simboli da Σ
  - La stringa vuota è denotata con ϵ

Lunghezza di una stringa: Numero di posizioni per i simboli nella stringa.

$|w|$ denota la lunghezza della stringa w

$|0110| = 4, |\epsilon| = 0$

Potenze di un alfabeto: Σ^k = insieme delle stringhe di lunghezza k con simboli da Σ

Esempio: $\Sigma = \{0, 1\}$ $\Leftarrow$ 0 rappresenta un simbolo (in C '0')

$\Sigma^1 = \{0, 1\}$ $\Leftarrow$ 0 (in C "0")

$\Sigma^2 = \{00, 01, 10, 11\}$

$\Sigma^0 = \{\epsilon\}$

Domanda: Quante stringhe ci sono in Σ^3?

L'insieme di tutte le stringhe su Σ è denotato da Σ^*

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$

Ad esempio, $\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001...\}$ è l'insieme di tutte le stringhe composte di 0 e di 1.

Anche:

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$

$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

Se Σ è finito, Σ^* è *numerabile*, ovvero esiste una corrispondenza biunivoca tra Σ^* e $\mathbb{N}$

Concatenazione: Se x e y sono stringhe, allora xy è la stringa ottenuta rimpiazzando una copia di y immediatamente dopo una copia di x

$x = a_1 a_2 \ldots a_i, y = b_1 b_2 \ldots b_j$

$xy = a_1 a_2 \ldots a_i b_1 b_2 \ldots b_j$

Esempio: $x = 01101, y = 110, xy = 01101110$

Nota: Per ogni stringa x

$$x\epsilon = \epsilon x = x$$

Linguaggi

Definizione

Se Σ è un alfabeto (finito), e $L \subseteq \Sigma^*$, allora L è un **linguaggio**

Esempi di linguaggi:

- L'insieme delle parole italiane legali
- L'insieme dei programmi C legali
- L'insieme delle stringhe che consistono di n zeri seguiti da n uni

$$\{\epsilon, 01, 0011, 000111, \ldots\}$$

Altri esempi

- L'insieme delle stringhe con un numero uguale di zeri e di uni

$$\{\epsilon, 01, 10, 0011, 0101, 1001, \ldots\}$$

- L_P = insieme dei numeri binari il cui valore è primo

$$\{10, 11, 101, 111, 1011, \ldots\}$$

- Il linguaggio vuoto $\emptyset$
- Il linguaggio $\{\epsilon\}$ consiste della stringa vuota

Nota: $\emptyset \neq \{\epsilon\}$

Nota: L'alfabeto Σ è sempre finito

Tipi di dato strutturati: Array

- I tipi di dato visti finora sono tutti semplici: `int`, `char`, `float`, ...
- ma i dati manipolati nelle applicazioni reali sono spesso complessi (o strutturati)
- Gli array sono uno dei tipi di dato strutturati
  - sono composti da elementi omogenei (tutti dello stesso tipo)
  - ogni elemento è identificato all'interno dell'array da un numero d'ordine detto indice dell'elemento
  - il numero di elementi dell'array è prefissato e detto lunghezza (o dimensione) dell'array
- Consentono di rappresentare tabelle, matrici, matrici n-dimensionali, ...

Array monodimensionali (o vettori)

- Supponiamo di dover rappresentare e manipolare la classifica di un campionato cui partecipano 16 squadre.
- È del tutto naturale pensare ad una tabella

Classifica

Squadra A	Squadra B	...	Squadra C
1^o posto	2^o posto		16^o posto

che si evolve con il procedere del campionato

Classifica

Squadra B	Squadra A	...	Squadra C
1^o posto	2^o posto		16^o posto

Sintassi: dichiarazione di variabile di tipo vettore

```
tipo-elementi nome-array [lunghezza];
```

Esempio: `int vet[6];`

dichiara un vettore di 6 elementi, ciascuno di tipo intero.

- All'atto di questa dichiarazione vengono riservate (allocate) 6 locazioni di memoria consecutive, ciascuna contenente un intero. La lunghezza del vettore è 6.
- La lunghezza di un vettore deve essere costante (nota a tempo di compilazione). Non si può ad esempio chiedere all'utente di immettere il valore della lunghezza.
- Ogni elemento del vettore è una variabile identificata dal nome del vettore e da un indice, ad esempio `vet[i]`

  Sintassi: elemento di array `nome-array[espressione];`

  Attenzione: `espressione` deve essere di tipo intero ed il suo valore deve essere compreso tra 0 a lunghezza-1.

- **Esempio:**

indice	elemento	variabile
0	?	`vet[0]`
1	?	`vet[1]`
2	?	`vet[2]`
3	?	`vet[3]`
4	?	`vet[4]`
5	?	`vet[5]`

- `vet[i]` è l'elemento del vettore `vet` di indice `i`.
  Ogni elemento del vettore è una variabile.

```
int vet[6], a;
vet[0] = 15; // Inizializzazione implicita
a = vet[0];
vet[1] = vet[0] + a; // Inizializzazione implicita
printf("%d", vet[0] + vet[1]);
```

- `vet[0]`, `vet[1]`, ecc. sono variabili intere come tutte le altre e dunque possono stare a sinistra dell'assegnamento (es. `vet[0] = 15`), come all'interno di espressioni (es. `vet[0] + a`).
- Come detto, l'indice del vettore è un'espressione.

```
index = 2;
vet[index+1] = 23;
```

Inizializzazione di vettori

- Gli elementi del vettore possono essere inizializzati con valori costanti (valutabili a tempo di compilazione) contestualmente alla dichiarazione del vettore. [Inizializzazione esplicita]

  Esempio: `int n[4] = {11, 22, 33, 44};`

- l'inizializzazione deve essere contestuale alla dichiarazione

  Esempio:
  ```
  int n[4];
  n = {11, 22, 33, 44};
  ```
  $\Longrightarrow$ errore!

- se i valori iniziali sono meno degli elementi, i rimanenti vengono posti a 0

`int n[10] = {3};`	azzera i rimanenti 9 elementi del vettore
`float af[5] = {0.0};`	pone a 0.0 i 5 elementi
`int x[5] = {};`	errore!

- se ci sono più inizializzatori di elementi, si ha un errore a tempo di compilazione

  Esempio: `int v[2] = {1, 2, 3};` errore!

- se si mette una sequenza di valori iniziali, si può omettere la lunghezza (viene presa la lunghezza della sequenza)

  Esempio: `int n[] = {1, 2, 3};` equivale a `int n[3] = {1, 2, 3};`

Altri errori tipici

- Che cosa accade se scriviamo o leggiamo un indice fuori dall'array?

```
int vet[6];

vet[6] = 15;
```

- Errore di segmentazione o Segmentation Fault a tempo di esecuzione: tentativo di accedere ad una posizione di memoria alla quale non è permesso accedere
- Che succede se leggiamo un valore da un array non inizializzato?

```
int vet[6];

x =vet[1];
```

- Il risultato della lettura non è un valore affidabile.

Manipolazione di vettori

- avviene solitamente attraverso cicli **for**
- l'indice del ciclo varia in genere da 0 a lunghezza-1
- spesso conviene definire la lunghezza come una costante attraverso la direttiva `#define`

Esempio: Lettura e stampa di un vettore.

```
#include <stdio.h>
#define LUNG 5

main ()
{
int v[LUNG]; /* vettore di LUNG elementi, indicizzati da 0 a LUNG-1 */
int i;

for (i = 0; i < LUNG; i++) {
  printf("Inserisci l'elemento di indice %d: ", i);
  scanf("%d", &v[i]);
}
printf("Indice Elemento\n");
for (i = 0; i < LUNG; i++) {
  printf("%6d %8d\n", i, v[i]); }

}
```

- In C l'unica operazione possibile sugli array è l'accesso ai singoli elementi.
- Ad esempio, non si possono effettuare direttamente delle assegnazioni tra vettori.

Esempio:

```
int a[3] = {11, 22, 33};
int b[3];
b = a;                          errore!
```

Esempi

- Calcolo della somma degli elementi di un vettore.

```
int a[10], i, somma = 0;
...
for (i = 0; i < 10; i++)
  somma += a[i];
printf("%d", somma);
```

- Leggere N interi e stampare i valori maggiori di un valore intero y letto in input.

```
#include <stdio.h>
#define N 4
main()   {
int ris[N];
int y, i;
printf("Inserire i %d valori:\n", N);
for (i = 0; i < N; i++) {
  printf("Inserire valore n. %d: ", i+1);
  scanf("%d", &ris[i]);     }
printf("Inserire il valore y:\n");
scanf("%d", &y);
printf("Stampa i valori maggiori di %d:\n", y);
for (i = 0; i < N; i++)
  if (ris[i] > y)
    printf("L'elemento %d: %d è maggiore di %d\n",
           i+1, ris[i],y);
}
```

- Leggere una sequenza di caratteri terminata dal carattere `\n` di fine linea e stampare le frequenze delle cifre da `'0'` a `'9'`.

Altri esempi

- L'insieme delle stringhe con un numero uguale di zeri e di uni

$$\{\epsilon, 01, 10, 0011, 0101, 1001, \ldots\}$$

- L_P = insieme dei numeri binari il cui valore è primo

$$\{10, 11, 101, 111, 1011, \ldots\}$$

- Il linguaggio vuoto $\emptyset$
- Il linguaggio $\{\epsilon\}$ consiste della stringa vuota

Nota: $\emptyset \neq \{\epsilon\}$

Nota: L'alfabeto Σ è sempre finito

Problemi

- La stringa w è un elemento di un linguaggio L?
- Esempio: Dato un numero binario, è primo = è un elemento di L_P?
- È $11101 \in L_P$? Che risorse computazionali sono necessarie per rispondere a questa domanda?
- Di solito non pensiamo ai problemi come delle decisioni sì/no, ma come qualcosa che trasforma un input in un output.
- Esempio: Fare il parsing di un programma C = controllare se il programma è corretto, e se lo è, produrre un albero di parsing.

Automi a stati finiti deterministici

Un DFA (Deterministic Finite Automaton, in italiano ASFD) è un formalismo per definire linguaggi che consiste di:

- Q è un insieme finito di *stati*
- Σ è un *alfabeto finito* ($=$ simboli in input)
- δ è una *funzione di transizione* $(q, a) \mapsto p$
- $q_0 \in Q$ è lo *stato iniziale*
- $F \subseteq Q$ è un insieme di *stati finali*

Un DFA è quindi una quintupla

$$A = (Q, \Sigma, \delta, q_0, F)$$

Funzione di transizione

- La funzione di transizione δ prende in ingresso uno stato e un simbolo e restituisce uno stato
- $\delta(q, a) = p$ denota che p è lo stato raggiunto a partire dallo stato q quando si riceve in input il simbolo a
- Per ogni stato deve essere sempre definito uno stato successivo per ogni simbolo di input (esistono anche stati pozzo).

Esempio

Esempio: Un automa che accetta

$$L = \{x01y : x, y \in \{0,1\}^*\}$$

è l'automa $A = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_1\})$, dove

q_0 rappresenta lo stato in cui non si è ancora visto 01

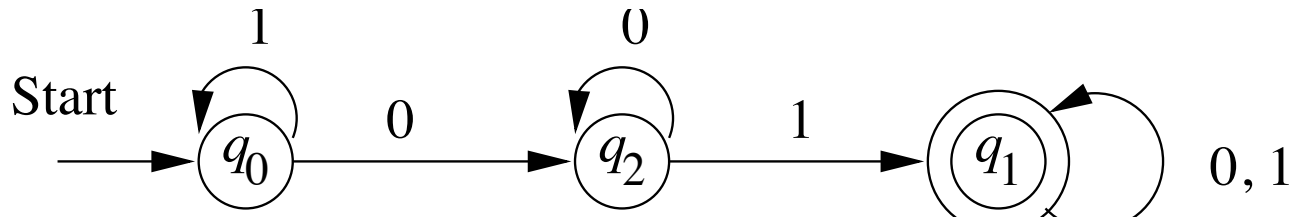
q_2 rappresenta lo stato in cui non si è visto 01, ma si è appena visto 0

q_1 rappresenta lo stato in cui si è visto 01

- L'automa come una *tabella di transizione*(stati/simboli di input):

	0	1
$\rightarrow q_0$	q_2	q_0
$\star q_1$	q_1	q_1
q_2	q_2	q_1

- L'automa come un *diagramma di transizione*:

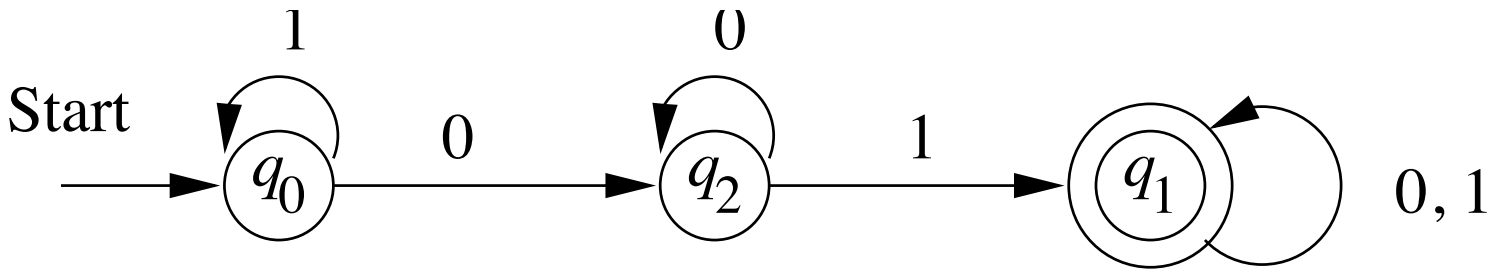


Accettazione

Un automa a stati finiti (FA) *accetta* una stringa $w = a_1 a_2 \cdots a_n$ se esiste un cammino nel diagramma di transizione che

1. Inizia nello stato iniziale
2. Finisce in uno stato finale (di accettazione)
3. Ha una sequenza di etichette $a_1 a_2 \cdots a_n$

Esempio: L'automa a stati finiti



accetta ad esempio la stringa 01101

Funzione di transizione estesa $\hat{\delta}$ e il linguaggio accettato

- La funzione di transizione δ può essere estesa a $\hat{\delta}$ che opera su stati e stringhe (invece che su stati e simboli)
- $\hat{\delta}(q, w) = p$ denota che p è lo stato raggiunto a partire dallo stato q quando si riceve in input uno ad uno i simboli $a_1 ... a_n$ che formano la stringa w

Definizione

Base: $\hat{\delta}(q, \epsilon) = q$

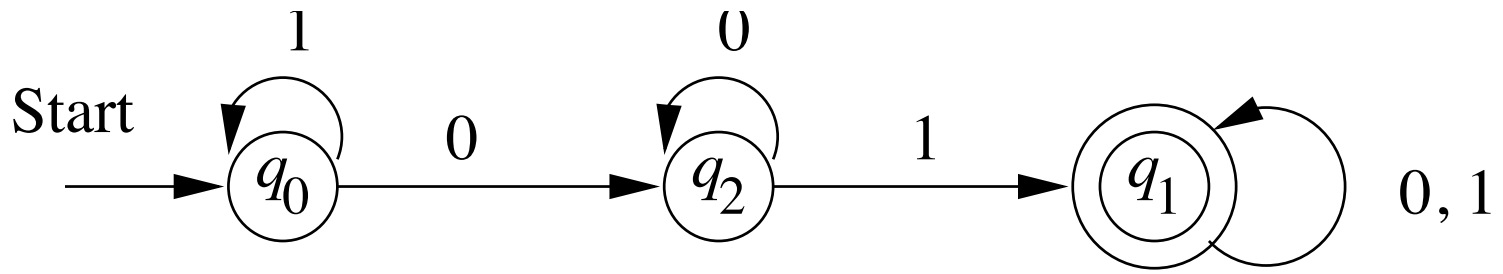
Induzione: $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$

Definizione

Il *linguaggio accettato da* A è $L(A) = \{w : \hat{\delta}(q_0, w) \in F\}$

- Insiemi diversi di stati finali portano a linguaggi diversi.
- I linguaggi accettati da automi a stati finiti sono detti linguaggi regolari

Esempio di stringa accettata



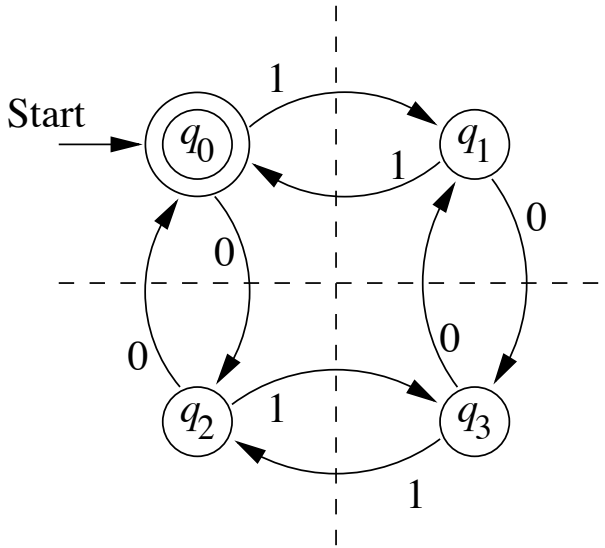
Applichiamo la funzione di transizione estesa $\hat{\delta}$ all'input 01101:

1. $\hat{\delta}(q_0, \epsilon) = q_0$
2. $\hat{\delta}(q_0, 0) = q_2$
3. $\hat{\delta}(q_0, 01) = \delta(\hat{\delta}(q_0, 0), 1) = \delta(q_2, 1) = q_1$
4. $\hat{\delta}(q_0, 011) = \delta(\hat{\delta}(q_0, 01), 1) = \delta(q_1, 1) = q_1$
5. $\hat{\delta}(q_0, 0110) = \delta(\hat{\delta}(q_0, 011), 0) = \delta(q_1, 1) = q_1$
6. $\hat{\delta}(q_0, 01101) = \delta(\hat{\delta}(q_0, 0110), 1) = \delta(q_1, 1) = q_1$

con $q_1 \in F$

Esempio

Esempio: DFA che accetta tutte e sole le stringhe con un numero pari di zeri e un numero pari di uni (compresa quindi la stringa ϵ)



Esempio

Rappresentazione tabulare dell'automa

		0	1
$\star \rightarrow$	q_0	q_2	q_1
	q_1	q_3	q_0
	q_2	q_0	q_3
	q_3	q_1	q_2

q_0 rappresenta lo stato in cui sia il numero di 0 che di 1 è pari
q_1 rappresenta lo stato in cui il numero di 0 è pari e quello di 1 è dispari
q_2 rappresenta lo stato in cui il numero di 0 è dispari e quello di 1 è pari
q_3 rappresenta lo stato in cui sia il numero di 0 che di 1 è dispari

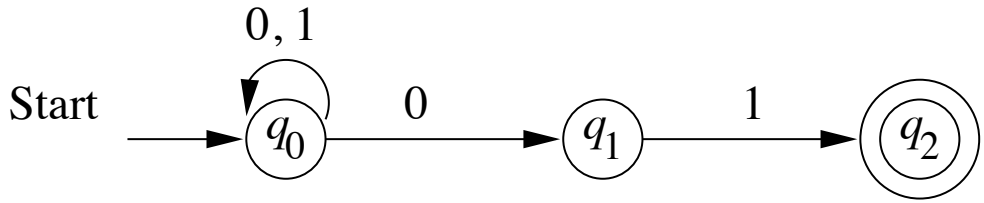
Alcuni esercizi

- DFA per i seguenti linguaggi sull'alfabeto $\{0, 1\}$:
  - Insieme di tutte le stringhe che finiscono con 00
  - Insieme di tutte le stringhe con tre zeri consecutivi
  - Insieme delle stringhe con 011 come sotto-stringa
  - Insieme delle stringhe che cominciano o finiscono (o entrambe le cose) con 01 [Provare a fare prima un automa per le stringhe che cominciano con 01 e uno per quelle che finiscono con 01]

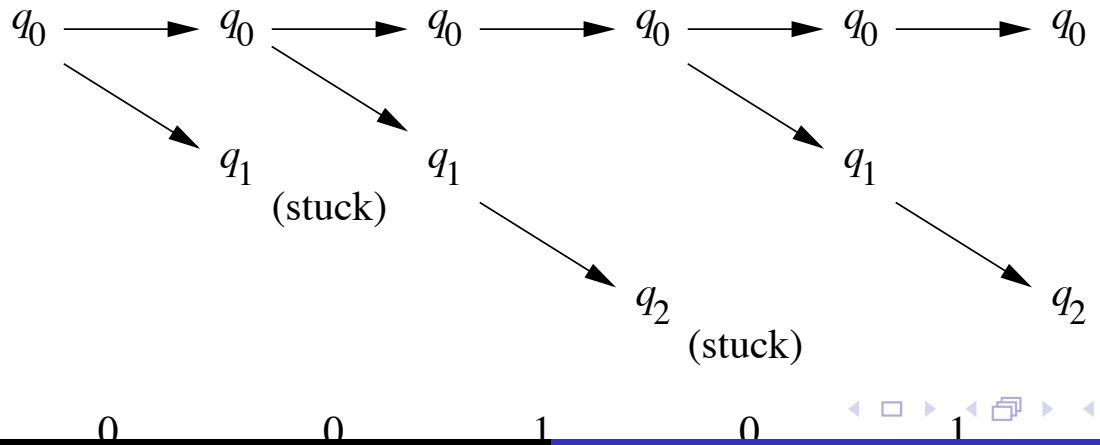
Automi a stati finiti non deterministici (NFA)

- Un NFA può essere in vari stati nello stesso momento, intuitivamente può "scommettere" su quale sarà il prossimo stato.
- Da uno stato, dato un certo input, posso infatti raggiungere un insieme di stati.

Es: automa che accetta tutte e solo le stringhe che finiscono in 01.



Ecco cosa succede quando l'automa elabora l'input 00101 con l'*albero delle alternative*



Definizione formale di NFA

Formalmente, un NFA è una quintupla

$$A = (Q, \Sigma, \delta, q_0, F)$$

- Q è un insieme finito di stati
- Σ è un alfabeto finito
- δ è una funzione di transizione da $Q \times \Sigma$ all'insieme dei sottoinsiemi di Q
- $q_0 \in Q$ è lo *stato iniziale*
- $F \subseteq Q$ è un insieme di *stati finali*

Funzione di transizione

- La funzione di transizione δ prende in ingresso uno stato e un simbolo e restituisce un insieme di stati (che può essere anche vuoto).
- Lo stato successivo non è quindi univocamente determinato dallo stato corrente e dall'input. L'automa può *scegliere* tra stati diversi.
- $\delta(q, a) = \{p_1, ..., p_n\}$ denota che ogni stato p_i può essere raggiunto dallo stato q quando si riceve in input il simbolo a

$$q \rightarrow p_i \Leftrightarrow p_i \in \delta(q, a)$$

- Non è detto che per ogni stato sia sempre definito uno stato successivo per ogni simbolo di input.

Esempio

L' NFA di due pagine fa è

$$(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$$

dove δ è la funzione di transizione

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$\star q_2$	$\emptyset$	$\emptyset$

Funzione di transizione estesa $\hat{\delta}$ e il linguaggio accettato

Definizione induttiva di $\hat{\delta}$.

Definizione

Base: $\hat{\delta}(q, \epsilon) = \{q\}$

Induzione:

$$\hat{\delta}(q, xa) = \bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a)$$

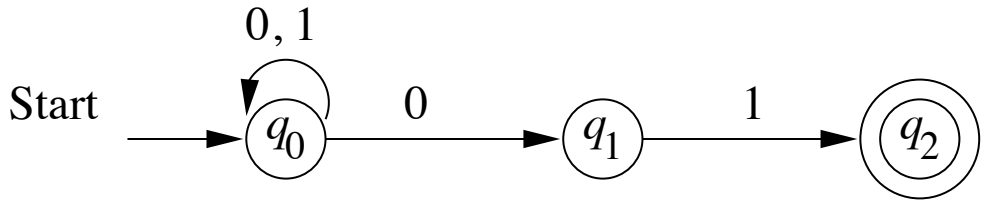
cioè l'unione di tutti gli stati $\delta(p, a)$ per p che appartiene a $\hat{\delta}(q, x)$

- Un NFA *accetta* una stringa w se $\hat{\delta}(q_0, w) \in F$ contiene almeno uno stato finale

Definizione

Il *linguaggio accettato da* A è $L(A) = \{w : \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$

Automi a stati finiti non deterministici (NFA)



Applichiamo la funzione di transizione estesa $\hat{\delta}$ all'input 00101:

1. $\hat{\delta}(q_0, \epsilon) = \{q_0\}$
2. $\hat{\delta}(q_0, 0) = \{q_0, q_1\}$
3. $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
4. $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
5. $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
6. $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

con $\{q_0, q_2\} \cap F \neq \emptyset$

Dimostrazioni di equivalenza

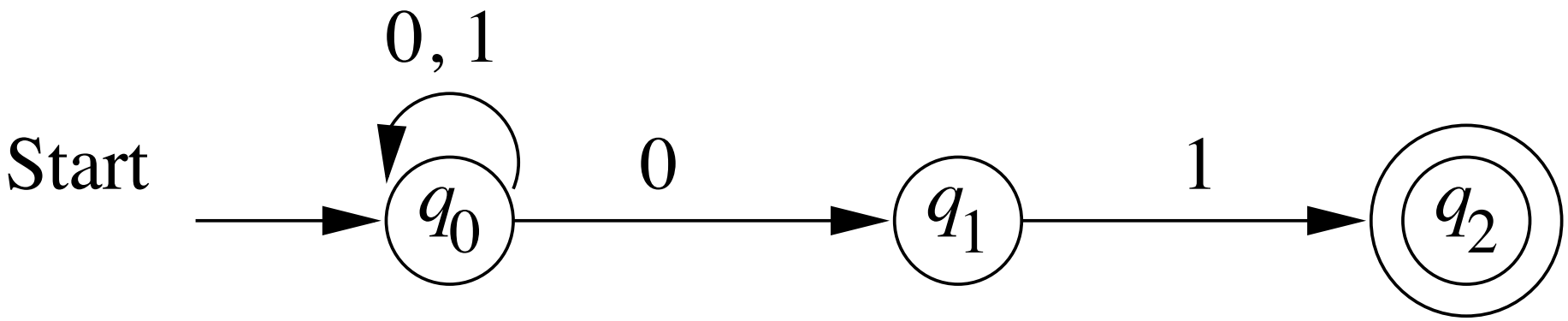
- Spesso ci servirà dimostrare che due descrizioni di insiemi, descrivono lo stesso insieme, ad esempio che il linguaggio accettato dall'automa visto prima coincide con il linguaggio $\{x01 : x \in \Sigma^*\}$
- Per dimostrare che due insiemi S e T sono uguali, si deve dimostrare che $S \subseteq T$ e che $T \subseteq S$:
  - $w \in S \Rightarrow w \in T$, and
  - $w \in T \Rightarrow w \in S$
- Nel nostro esempio dobbiamo dimostrare che

$$w \in L(A) \quad \Leftrightarrow \quad w \in \{x01 : x \in \Sigma^*\}$$

ovvero

$$q_2 \in \hat{\delta}(q_0, w) \quad \Leftrightarrow \quad w \in \{x01 : x \in \Sigma^*\}$$

Per dimostrare formalmente che l'NFA



accetta il linguaggio $\{x01 : x \in \Sigma^*\}$, ci conviene espandere le ipotesi. La dimostrazione si fa per mutua induzione, sulla lunghezza della stringa w, sui tre enunciati seguenti

1. $w \in \Sigma^* \Rightarrow q_0 \in \hat{\delta}(q_0, w)$
2. $q_1 \in \hat{\delta}(q_0, w) \Leftrightarrow w = x0$
3. $q_2 \in \hat{\delta}(q_0, w) \Leftrightarrow w = x01$

Dimostrazione

1. $w \in \Sigma^* \Rightarrow q_0 \in \hat{\delta}(q_0, w)$
2. $q_1 \in \hat{\delta}(q_0, w) \Leftrightarrow w = x0$
3. $q_2 \in \hat{\delta}(q_0, w) \Leftrightarrow w = x01$

Base: Se $|w| = 0$ allora $w = \epsilon$. Allora l'enunciato (1) segue dalla definizione. Per (2) e (3) le ipotesi di entrambi i lati sono false per ϵ e quindi entrambe le implicazioni sono banalmente vere, dato che in logica:

$$(\mathit{False} \Rightarrow \mathit{False}) \equiv \mathit{True}$$

Induzione: Ipotizziamo che $w = xa$, dove $a \in \{0, 1\}$, $|x| = n$ e gli enunciati (1)–(3) valgono per x. Si mostra che gli enunciati valgono per xa (che è lunga $n + 1$).

Dimostrazione (cont.)

Induzione: Ipotizziamo che $w = xa$, dove $a \in \{0, 1\}$, $|x| = n$ e gli enunciati (1)–(3) valgono per x. Si mostra che gli enunciati valgono per xa (che è lunga $n + 1$).

1. $w \in \Sigma^* \Rightarrow q_0 \in \hat{\delta}(q_0, w)$
   Per ipotesi induttiva $q_0 \in \hat{\delta}(q_0, x)$. Dato che $\forall a \in \Sigma . q_0 \in \delta(q_0, a)$, allora $\hat{\delta}(q_0, w) = \delta(\hat{\delta}(q_0, x), a) \ni q_0$

Dimostrazione (cont.)

Induzione: Ipotizziamo che $w = xa$, dove $a \in \{0, 1\}$, $|x| = n$ e gli enunciati (1)–(3) valgono per x. Si mostra che gli enunciati valgono per xa (che è lunga $n + 1$).

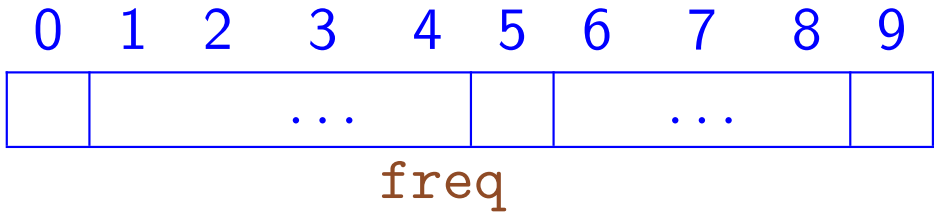
2.
   - (**se**) $q_1 \in \hat{\delta}(q_0, w) \Leftarrow w = x0$
     Supponiamo che $w = xa$ finisca per 0 (quindi $a = 0$).
     Per l'enunciato (1) sappiamo che $q_0 \in \hat{\delta}(q_0, x)$ e
     dato che $q_1 \in \delta(q_0, 0)$, possiamo concludere che $q_1 \in \hat{\delta}(q_0, w)$.
   - (**solo se**) $q_1 \in \hat{\delta}(q_0, w) \Rightarrow w = x0$
     Supponiamo che $q_1 \in \hat{\delta}(q_0, w)$. Dal diagramma vediamo che l'unico modo di raggiungere q_1 è che $w = x0$.

Dimostrazione (cont.)

3.
   - (**se**) $q_2 \in \hat{\delta}(q_0, w) \Leftarrow w = x01$
     Supponiamo che $w = xa$ finisca per 01 (quindi $a = 1$ e x finisce per 0).
     Per l'enunciato (2) sappiamo che $q_1 \in \hat{\delta}(q_0, x)$ e
     dato che $q_2 \in \delta(q_1, 1)$, possiamo concludere che $q_2 \in \hat{\delta}(q_0, w)$.
   - (**solo se**) $q_2 \in \hat{\delta}(q_0, w) \Rightarrow w = x01$
     Supponiamo che $q_2 \in \hat{\delta}(q_0, w)$. Dal diagramma vediamo che
     l'unico modo di raggiungere q_2 è che $w = x1$, dove $q_1 \in \hat{\delta}(q_0, x)$.
     Per l'enunciato (2), x finisce per 0 e quindi w finisce per 01.

- Leggere una sequenza di caratteri terminata dal carattere `\n` di fine linea e stampare le frequenze delle cifre da `'0'` a `'9'`.

- Leggere una sequenza di caratteri terminata dal carattere `\n` di fine linea e stampare le frequenze delle cifre da `'0'` a `'9'`.
- utilizziamo un vettore `freq` di 10 elementi nel quale memorizziamo le frequenze dei caratteri da `'0'` a `'9'`



`freq[0]` conta il numero di occorrenze di `'0'`

...

`freq[9]` conta il numero di occorrenze di `'9'`

- utilizziamo un ciclo per l'acquisizione dei caratteri in cui aggiorniamo una delle posizioni dell'array tutte le volte che il carattere letto è una cifra

```
int i; char ch;
int freq[10] = {0};
do {
  ch = getchar();
  switch (ch) {
    case '0': freq[0]++; break;
    case '1': freq[1]++; break;
    case '2': freq[2]++; break;
    case '3': freq[3]++; break;
    case '4': freq[4]++; break;
    case '5': freq[5]++; break;
    case '6': freq[6]++; break;
    case '7': freq[7]++; break;
    case '8': freq[8]++; break;
    case '9': freq[9]++; break;
    }
} while (ch != '\n');
printf("Le frequenze sono:\n");
for (i = 0; i < 10; i++)

  printf("Freq. di %d: %d\n", i, freq[i]);
```

- Nel ciclo **do-while**, il comando `switch` può essere rimpiazzato da un **if** come segue

```
if (ch >= '0' && ch <= '9')
  freq[ch - '0']++;
```

  Infatti:

- i codici dei caratteri da `'0'` a `'9'` sono consecutivi
- dato un carattere `ch`, l'espressione intera `ch - '0'` è la distanza del codice di `ch` dal codice del carattere `'0'`. In particolare:
  - `'0' - '0' = 0`
  - `'1' - '0' = 1`
  - ...
  - `'9' - '0' = 9`

- Leggere da tastiera e memorizzare una sequenza di interi di lunghezza prefissata `DIM`. Stampare quindi la sequenza in ordine inverso.
- Leggere da tastiera i risultati (double) di 20 esperimenti. Stampare il numero d'ordine ed il valore degli esperimenti per i quali il risultato è minore del 50% della media.

Array multidimensionali

Sintassi: dichiarazione

`tipo-elementi nome-array` $[\text{lung}_1][\text{lung}_2]\cdots[\text{lung}_n]$`;`

Esempio: **int** `mat[3][4];` $\Longrightarrow$ matrice 3×4

- Per ogni dimensione i l'indice va da 0 a lung_i-1.

		colonne 0	1	2	3
	0	?	?	?	?
righe	1	?	?	?	?
	2	?	?	?	?

Esempio: **int** `marketing[10][5][12]`

(indici potrebbero rappresentare: prodotti, venditori, mesi dell'anno)

Accesso agli elementi di una matrice

```
int i, mat[3][4];
...
i = mat[0][0];        elemento di riga 0 e colonna 0 (primo elemento)
mat[2][3] = 28;       elemento di riga 2 e colonna 3 (ultimo elemento)
mat[2][1] = mat[0][0] * mat[1][3];
```

- Come per i vettori, l'unica operazione possibile sulle matrici è l'accesso agli elementi tramite l'operatore `[]`.

Esempio: Lettura e stampa di una matrice.

```
#include <stdio.h>
#define RIG 2
#define COL 3
main()
{
int mat[RIG][COL];
int i, j;
/* lettura matrice */
printf("Lettura matrice %d x %d;\n", RIG, COL);
for (i = 0; i < RIG; i++)
  for (j = 0; j < COL; j++)
    scanf("%d", &mat[i][j]);
/* stampa matrice */
printf("La matrice è:\n");
for (i = 0; i < RIG; i++) {
   for (j = 0; j < COL; j++)
     printf("%6d ", mat[i][j]);
   printf("\n");     }   /* a capo dopo ogni riga */
}
```

Esempio: Programma che legge due matrici $M \times N$ (ad esempio 4×3) e calcola la matrice somma.

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    c[i][j] = a[i][j] + b[i][j];
```

Inizializzazione di matrici

```
int mat[2][3] = {{1,2,3}, {4,5,6}};
int mat[2][3] = {1,2,3,4,5,6};
```

1	2	3
4	5	6

```
int mat[2][3] = {{1,2,3}};
int mat[2][3] = {1,2,3};
```

1	2	3
0	0	0

```
int mat[2][3] = {{1}, {2,3}};
```

1	0	0
2	3	0

Esercizio

Programma che legge una matrice A (M×P) ed una matrice B (P×N) e calcola la matrice C prodotto di A e B

- La matrice C è di dimensione M×N.
- Il generico elemento C_{ij} di C è dato da:

$$C_{ij} = \sum_{k=0}^{P-1} A_{ik} \cdot B_{kj}$$

Soluzione

```
#define M 3
#define P 4
#define N 2
int a[M][P], b[P][N], c[M][N];
...
/* calcolo prodotto */
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++) {
    c[i][j] = 0;
    for (k = 0; k < P; k++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
  }
```

- Tutti gli elementi di c possono essere inizializzati a 0 al momento della dichiarazione:

```
int a[M][P], b[P][N], c[M][N] = {0};
...
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < P; k++)
      c[i][j] += a[i][k] * b[k][j];
```

Equivalenza di DFA e NFA

- Gli NFA sono di solito più facili da "programmare".
- Sorprendentemente, per ogni NFA N c'è un DFA D, tale che $L(D) = L(N)$, e viceversa.
- Questo comporta una *costruzione per sottoinsiemi*, un esempio importante di come un automa B può essere costruito a partire da un altro automa A.
- Dato un NFA

$$N = (Q_N, \Sigma, \delta_N, q_0, F_N)$$

costruiremo un DFA

$$D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$$

tali che

$$L(D) = L(N).$$

I dettagli della costruzione per sottoinsiemi:

- $Q_D = \{S : S \subseteq Q_N\}$.

Nota: $|Q_D| = 2^{|Q_N|}$, anche se la maggior parte degli stati in Q_D sono "garbage", cioè non raggiungibili dallo stato iniziale.

- $F_D = \{S \subseteq Q_N : S \cap F_N \neq \emptyset\}$
- Per ogni $S \subseteq Q_N$ e $a \in \Sigma$,

$$\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$$

Ad esempio l'insieme di stati $\{q_0, q_1\}$ nel nostro esempio diventa un singolo stato del DFA corrispondente, raggiungibile dallo stato $\{q_0\}$.

Costruiamo δ_D dall'NFA già visto, quello cioè che accetta le stringhe che terminano con 01.

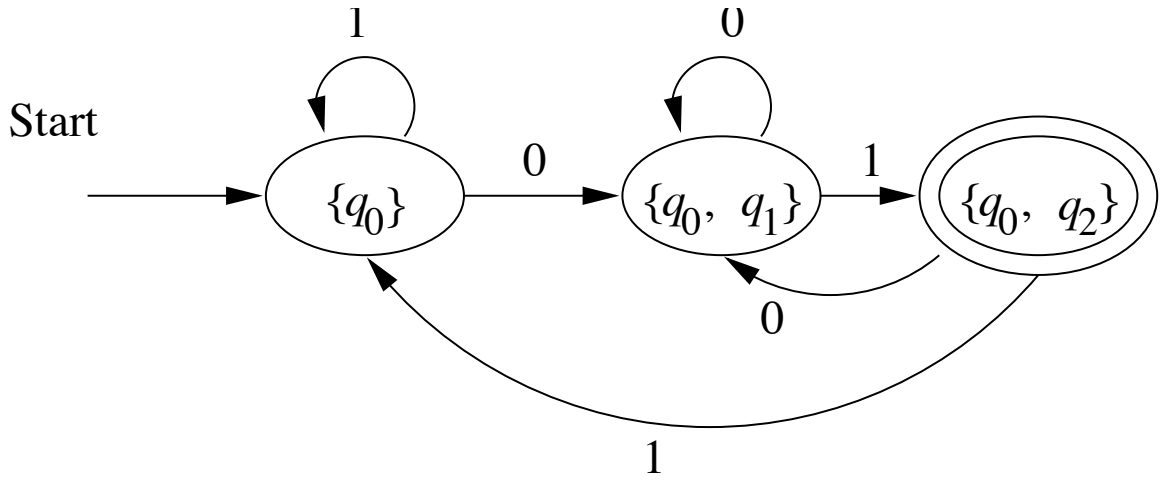
	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$\star\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\star\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\star\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$\star\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Nota: Gli stati di D corrispondono a sottoinsiemi di stati di N, ma potevamo denotare gli stati di D in un altro modo, per esempio $A - F$.

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$\star D$	A	A
E	E	F
$\star F$	E	B
$\star G$	A	D
$\star H$	E	F

Per evitare la crescita esponenziale degli stati può essere utile costruire la tabella di transizione per D solo per gli stati raggiungibili (o accessibili) S come segue:
Base: $S = \{q_0\}$ è raggiungibile in D
Induzione: Se lo stato S è raggiungibile, lo sono anche gli stati in $\bigcup_{a \in \Sigma} \delta_D(S, a)$ raggiungibile a partire da S.

Esempio: Il "sottoinsieme" DFA con i soli stati raggiungibili: B ($\{q_0\}$), E ($\{q_0, q_1\}$) e F ($\{q_0, q_2\}$).



Teorema 2.11:

Teorema

Sia D il DFA ottenuto da un NFA N con la costruzione a sottoinsiemi. Allora $L(D) = L(N)$.

Dimostrazione: Prima mostriamo per induzione su $|w|$ che

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

N.B. Entrambe le funzioni restituiscono un insieme di stati di Q_N.
Base: $w = \epsilon$. Per definizione $\hat{\delta}_D(\{q_0\}, \epsilon) = \hat{\delta}_N(q_0, \epsilon) = \{q_0\}$.

Induzione: $w = xa$, con $|x| = n$ e $a \in \Sigma$. Per ipotesi induttiva $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$ che supponiamo uguali a $\{p_1, ..., p_k\}$.

$$\begin{aligned}
\hat{\delta}_N(q_0, xa) &\stackrel{\text{def}}{=} \delta_N(\hat{\delta}_N(q_0, x), a) \\
&\stackrel{\text{i.h.}}{=} \delta_N(\{p_1, ..., p_k\}, a) \\
&\stackrel{\text{def}}{=} \bigcup_{p_i \in \hat{\delta}_N(q_0, x)} \delta_N(p_i, a)
\end{aligned}$$

$$\begin{aligned}
\hat{\delta}_D(\{q_0\}, xa) &\stackrel{\text{def}}{=} \delta_D(\hat{\delta}_D(\{q_0\}, x), a) \\
&\stackrel{\text{i.h.}}{=} \delta_D(\{p_1, ..., p_k\}, a) \\
&\stackrel{\text{cst}}{=} \bigcup_{p_i \in \hat{\delta}_N(q_0, x)} \delta_N(p_i, a)
\end{aligned}$$

- Quindi $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$.
- Osservando inoltre che sia D che N accettano w se e solo se $\hat{\delta}_D(\{q_0\}, w)$ e $\hat{\delta}_N(q_0, w)$ contengono uno stato in F_N, ne segue che $L(D) = L(N)$.

Teorema 2.12: Un linguaggio L è accettato da un DFA se e solo se L è accettato da un NFA.

Dimostrazione: La parte "se" è il Teorema 2.11 più la costruzione per sottoinsiemi.

Per la parte "solo se" notiamo che un qualsiasi DFA può essere convertito in un NFA equivalente modificando la δ_D in δ_N secondo la regola seguente:

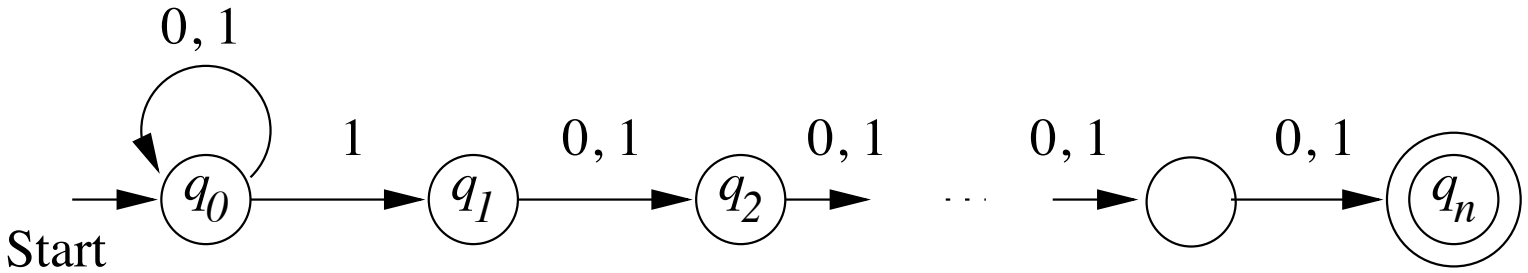
- Se $\delta_D(q, a) = p$, allora $\delta_N(q, a) = \{p\}$.

Per induzione su $|w|$ si può facilmente mostrare che se $\hat{\delta}_D(q_0, w) = p$, allora $\hat{\delta}_N(q_0, w) = \{p\}$.

Di conseguenza la stringa w viene accettata da D se e solo se viene accettata da N e l'enunciato del teorema segue.

Crescita esponenziale degli stati

Esiste un NFA N con $n+1$ stati che non ha nessun DFA equivalente con meno di 2^n stati



$$L(N) = \{x1c_1c_2 \cdots c_{n-1} : x \in \{0,1\}^*, c_i \in \{0,1\}\}$$

D deve ricordare gli ultimi n simboli che ha letto.
Dato che ci sono 2^n sequenze di n bit, se esistesse un DFA equivalente con meno di 2^n stati, allora esisterebbe uno stato q e due sequenze $a_1a_2 \cdots a_n$ e $b_1b_2 \cdots b_n$, con almeno $i : a_i \neq b_i$: $q \in \hat{\delta}_N(q_0, a_1a_2 \cdots a_n), q \in \hat{\delta}_N(q_0, b_1b_2 \cdots b_n), a_1a_2 \cdots a_n \neq b_1b_2 \cdots b_n$

Caso 1: $i = 1$

$1 a_2 \cdots a_n$

$0 b_2 \cdots b_n$

Allora q deve essere sia uno stato di accettazione che uno stato di non accettazione.

Caso 2: $i > 1$

$a_1 \cdots a_{i-1} 1 a_{i+1} \cdots a_n$

$b_1 \cdots b_{i-1} 0 b_{i+1} \cdots b_n$

Consideriamo ora lo stato p raggiunto dopo aver letto $i-1$ simboli 0 a partire da q.

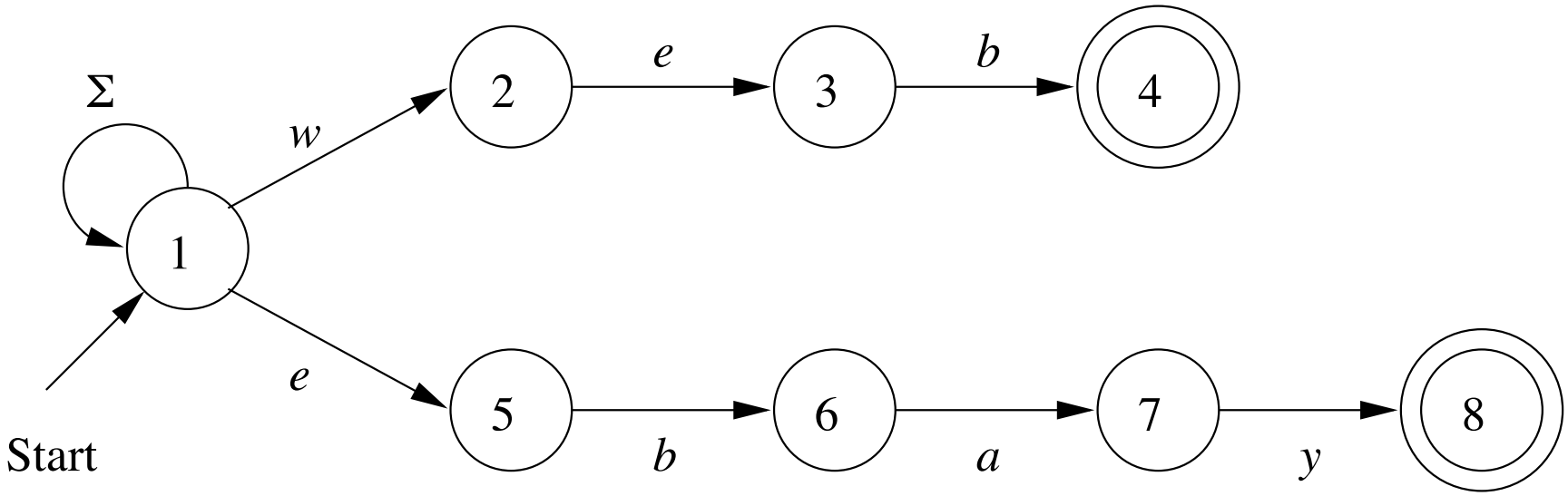
$\hat{\delta}_N(q_0, a_1 \cdots a_{i-1} 1 a_{i+1} \cdots a_n 0^{i-1}) = \hat{\delta}_N(q_0, b_1 \cdots b_{i-1} 0 b_{i+1} \cdots b_n 0^{i-1})$

Allora p deve essere sia uno stato di accettazione che uno stato di non accettazione.

$$\hat{\delta}_N(q_0, a_1 \cdots a_{i-1} 1 a_{i+1} \cdots a_n 0^{i-1}) \in F_D$$
$$\hat{\delta}_N(q_0, b_1 \cdots b_{i-1} 0 b_{i+1} \cdots b_n 0^{i-1}) \notin F_D$$

NFA per ricerche testuali

Un NFA che accetta l'insieme di parole chiave $\{\texttt{ebay}, \texttt{web}\}$



- Naturalmente l'NFA va poi implementato, passando all'equivalente DFA (che in questo caso ha gli stessi stati).
- Esempio di come derivare un programma (che implementa il DFA) a partire dalla specifica (quella dell'NFA)

Ancora esercizi sui DFA

- Definire un DFA che accetti, sull'alfabeto $\Sigma = \{0, 1\}$, i linguaggi dati dai seguenti insiemi:
  - quello di tutte le stringhe che contengono un numero pari di 0;
  - quello di tutte le stringhe che contengono un numero pari di 1;
  - quello di tutte le stringhe che non contengono mai più di due 0 consecutivi (ovvero che non contengono mai 000 come sottostringa);
- Dimostrare che $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$ per qualunque stato q e qualunque coppia di stringhe x e y.
  Si consiglia di usare l'induzione sulla lunghezza della stringa y.
- Dimostrare che $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$ per qualunque stato q, qualunque simbolo a e qualunque stringa x.
  Si consiglia di usare l'esercizio precedente.
- Definire un DFA (con $\Sigma = \{a, b, c, ..., z\}$) che accetti tutte le stringhe in cui le cinque vocali appaiono nell'ordine *a e i o u*
- Per questo come per gli altri argomenti attingere anche agli esercizi sul libro.

Esercizi sugli NFA

- Definire un NFA che accetti, sull'alfabeto $\Sigma = \{a, e, i, o, u\}$, i linguaggi dati dai seguenti insiemi:
  - quello di tutte le stringhe tali che la vocale finale sia apparsa in precedenza;
  - quello di tutte le stringhe tali che la vocale finale non sia apparsa in precedenza
- Definire un NFA che accetti sull'alfabeto $\Sigma = \{a, b\}$, il seguente insieme di stringhe: *abab*, *bab* e *abb*
- Convertire l'NFA ottenuto nel DFA corrispondente
- Definire un NFA che, sull'alfabeto $\Sigma = \{0, 1\}$, riconosca tutte le stringhe in cui compare la sequenza 011
- Dimostrare che l'NFA così definito accetta il linguaggio richiesto

Parametri di tipo vettore

- Il meccanismo del passaggio per valore di un indirizzo consente il passaggio di vettori come parametri di funzioni/procedure.
- Quando si passa un vettore come parametro ad una funzione, in realtà si sta passando l'indirizzo dell'elemento di indice 0.
- Il parametro formale deve essere di tipo puntatore (al tipo degli elementi del vettore)
- di solito si passa anche la dimensione del vettore in un ulteriore parametro.

Esempio:

```
void stampaVettore(int *v, int dim)
{ int i;
  for (i = 0; i < dim; i++)
    printf("v[%d]: %d\n", i, v[i]);
}

main()
{ int vet[5] = {1, 2, 3, 4, 5};
  ...
  stampaVettore(vet, 5);    ...  }
```

- Per evidenziare che il parametro formale è un vettore (ovvero l'indirizzo dell'elemento di indice 0), si può utilizzare la notazione `nome-parametro[]` invece di `*nome-parametro`.
  Esempio: `void stampa(int v[], int dim) { ... }`
- Si può anche specificare la dimensione nel parametro, ma questa viene ignorata.
  Esempio: `void stampa(int v[5], int dim) { ... }`
- Come al solito, nel prototipo della funzione il nome del parametro (vettore) può anche mancare.
  Esempio: `void stampa(int [], int);`

- Il passaggio di un vettore è un passaggio per indirizzo.

  $\Longrightarrow$ La funzione può modificare gli elementi del vettore passato.

  Esempio: Lettura di un vettore.

```
void leggiVettore(int v[], int dim)
{
  int i;
  for (i = 0; i < dim; i++)
  {
    printf("Immettere l'elemento di indice %d: ", i);
    scanf("%d", &v[i]);
  }
}
```

Esempio: Programma che legge, inverte e stampa un vettore di interi

```
#include <stdio.h>
#define LUNG 5

void leggiVettore(int [], int);
void stampaVettore(int [], int);
void invertiVettore(int [], int);

main()
{
  int vett[LUNG];

  leggiVettore(vett, LUNG);
  printf("Vettore prima dell'inversione\n");
  stampaVettore(vett, LUNG);

  invertiVettore(vett, LUNG);
  printf("Vettore dopo l'inversione\n");
  stampaVettore(vett, LUNG);
}
```

- La definizione della procedura `void invertiVettore(int [], int);` è lasciata per esercizio.

Bigné alla crema

- Come si preparano i bigné alla crema?
- si prepara la pasta choux
- si depositano piccole quantità di impasto sulla teglia
- si fanno cuocere per 7-8 minuti, ottenendo i bigné
- si prepara la crema
- si farciscono i bigné con la crema
- Già, ma per preparare pasta choux e crema ho bisogno delle relative ricette!

Prodotto matriciale

- Data una matrice A (M×P) ed una matrice B (P×N), come si calcola la matrice C prodotto di A e B?
- ogni elemento C_{ij} si ottiene dal prodotto scalare della riga i di A e della colonna j di B: $\mathbf{a_i}^\mathbf{T} \times \mathbf{b_j}$
- Adesso ho tuttavia bisogno di un sotto-programma per calcolare il prodotto scalare!

Modularizzazione

- Quando abbiamo a che fare con un problema complesso spesso lo suddividiamo in problemi più semplici che risolviamo separatamente, per poi combinare insieme le soluzioni dei sottoproblemi al fine di determinare la soluzione del problema di partenza.
- Questo procedimento è applicabile anche alla programmazione.
  - si suddivide un problema complesso in problemi di volta in volta più semplici
  - una volta individuati (sotto)problemi sufficientemente elementari si risolvono questi ultimi direttamente
  - si combinano le soluzioni dei sottoproblemi per ottenere la soluzione del problema di partenza

- Approccio top-down: si parte dall'alto, considerando il problema nella sua interezza e si procede verso il basso per raffinamenti successivi fino a ridurlo ad un insieme di sottoproblemi elementari
- Approccio bottom-up: ci si occupa prima di risolvere singole parti del problema, per poi risalire procedendo per aggiustamenti successivi fino ad ottenere la soluzione globale.
- I linguaggi di programmazione mettono a disposizione dei meccanismi di astrazione che favoriscono un approccio modulare

  Astrazione sui dati - si possono definire nuovi tipi di dato specifici per il particolare problema (tipi di dato astratti)
  - collezioni di valori + relative operazioni

  Astrazione funzionale - si possono definire sottoprogrammi per (sotto)problemi specifici.
  - i sottoprogrammi sono di solito parametrici e in C si realizzano attraverso le funzioni
  - possono essere (ri)usati alla stessa stregua delle operazioni built-in del linguaggio

Funzioni

- $y = f(x)$

  In una funzione matematica, ad ogni valore della variabile indipendente o argomento x corrisponde uno e un solo valore della variabile dipendente y.

- Compito dell'informatica è quello di trovare delle tecniche per calcolare le funzioni alla base dei problemi da risolvere.
  In C esiste il concetto di funzione:
  - le variabili indipendenti sono chiamate parametri formali
  - il risultato viene restituito attraverso il comando `return`
- Un programma C è definito come un insieme di funzioni (una obbligatoria, il `main`).

Funzioni

- Una funzione può essere vista come una scatola nera:

$$\text{parametri di ingresso} \longrightarrow \boxed{\text{F}} \longrightarrow \text{valore calcolato}$$

  - risolve un sottoproblema specifico
  - attraverso i parametri e il risultato scambia informazioni con il `main` e con altre funzioni

Esempio:

$$x \longrightarrow \boxed{\text{abs}} \longrightarrow |\,x\,|$$

$$x, y \longrightarrow \boxed{\text{mcd}} \longrightarrow mcd(x, y)$$

$$b, e \longrightarrow \boxed{\text{exp}} \longrightarrow b^e$$

$$x_1, \ldots, x_n \longrightarrow \boxed{\text{sum}} \longrightarrow \sum_{i=1}^{n} x_i$$

Esempio: Definizione di abs in C

```
int abs(int x)
{
int ris;
if (x<0)
   ris = -x;
else
   ris = x;
return ris;  }
```

- Uso della funzione

```
main()
{
int x1, x2, z, w;
  ...
z = abs(x1);
  ...
printf("%d\n", w + abs(x2));
  ...
}
```

Funzioni: perché?

La stesura di un programma riflette l'analisi funzionale del problema da risolvere. L'uso delle funzioni nasconde al resto del programma i dettagli implementativi, ponendo l'accento su cosa il programma fa rispetto a come lo fa, consentendo:

- la modularità
- la chiarezza e leggibilità
- la fattorizzazione del codice
- la separazione di ciò che cambia da ciò che resta uguale:
  - posso apportare modifiche alla funzione in un punto solo, senza rischiare modifiche parziali
  - posso anche cambiare l'implementazione della funzione senza cambiare il programma che la usa

Scrivi una volta, usa tutte le volte che vuoi

Ogni funzione rappresenta un'unità indipendente. Di conseguenza:

- chi scrive la funzione può non coincidere con chi la usa.
- la stessa funzione può essere usata in altri programmi (riuso del codice)
- ogni funzione può essere trattata separatamente dal resto

- Il linguaggio deve mettere a disposizione strumenti per
  - definire nuove operazioni astratte (funzioni)
  - usare le nuove operazioni definite
- Distinguiamo due momenti diversi:
  - la definizione della funzione
    definisce il codice che realizza l'operazione astratta
  - e la chiamata della funzione
    corrisponde all'utilizzo della funzione

- Ad una stessa definizione possono corrispondere diverse chiamate (come `z = abs(x1)` e `w + abs(x2)` nell'esempio precedente).
- Nella definizione della funzione, il codice fa riferimento agli argomenti o parametri formali della funzione (nell'esempio `x`)

  $\Longrightarrow$ un parametro formale non corrisponde ad un valore vero e proprio: è semplicemente un riferimento simbolico (ad un argomento della funzione)

Esempio:

```
int exp(int base, int esponente)
{
int ris = 1;
while (esponente > 0)
  {
    ris = ris * base;
    esponente = esponente - 1;
  }
return ris;
}
```

- La definizione di una funzione consta di due parti fondamentali:
  - Intestazione
  - Blocco (come nel `main()`).
- I parametri formali sono `base` ed `esponente`

- Al momento della chiamata, alla funzione vengono forniti i valori degli argomenti, o parametri attuali, rispetto ai quali effettuare il calcolo

```
int exp(int base, int esponente)
{...}

main()  {
int b, e, r1, r2;
  ...
r1 = exp(2,5);
  ...
scanf("%d  %d", &b, &e);
r2 = exp(b, e);
  ... }
```

- Prima chiamata `exp(2,5)`
  - `2` è il parametro attuale corrispondente a `base`
  - `5` è il parametro attuale corrispondente a `esponente`
- Seconda chiamata `exp(b,e)`
  - `b` è il parametro attuale corrispondente a `base`
  - `e` è il parametro attuale corrispondente a `esponente`

Funzioni: definizione

Come le variabili, anche le funzioni vanno dichiarate.

Sintassi:

`intestazione blocco`

dove

- `blocco` è il corpo della funzione
- `intestazione` è l'intestazione della funzione ed ha la seguente forma:
  `id-tipo identificatore (parametri-formali)`
- `id-tipo` specifica il tipo del risultato calcolato dalla funzione
- `identificatore` specifica il nome della funzione ed è un qualsiasi identificatore C valido
- `parametri-formali` è una sequenza (eventualmente vuota) di dichiarazioni di parametro (tipo e nome) separate da virgola

Esempi: intestazioni di funzione

- `int abs (int x)`
  `int MassimoComunDivisore(int a, int b)`
- `double Potenza(double x, double y)`
  `float media (int vet[], int lung)`

Funzioni: chiamata (invocazione, attivazione)

Sintassi:

`identificatore` (`parametri-attuali`)

- `identificatore` è il nome della funzione
- `lista-parametri-attuali` è una lista di espressioni separate da virgola
- i parametri attuali devono corrispondere in numero e tipo ai parametri formali, altrimenti sarà rilevato un errore di sintassi

Esempi: chiamate di funzioni

```
int mcd, x, y1, y2;
double exp, w, v, z;
  ...
mcd = MassimoComunDivisore(x+1, y1+y2);
exp = Potenza(z, 3.0);
  ...
exp = Potenza(z, Potenza(v,w));
```

Semantica (informale) di una chiamata di funzione

- Dentro il corpo di una funzione F compare una chiamata di un'altra funzione G
  - F viene detta funzione chiamante
  - G viene detta funzione chiamata

    Esempio: nel `main` c'è un assegnamento `x = abs(x);`

    $\Longrightarrow$ `main` è il chiamante, `abs` il chiamato
- Una chiamata di funzione è un'espressione, la cui valutazione avviene come segue:
  - viene sospesa l'esecuzione di F e viene "ceduto il controllo" a G, dopo aver opportunamente associato i parametri attuali ai parametri formali (passaggio dei parametri, fra poco ...)
  - vengono eseguite le istruzioni di G, a partire dalla prima
  - l'esecuzione di G termina con l'esecuzione di un'istruzione speciale (istruzione `return`) che calcola il risultato della chiamata (è il valore dell'espressione corrispondente alla chiamata)
  - al termine dell'esecuzione di G il controllo ritorna a F, che prosegue l'esecuzione a partire dal punto in cui G era stata attivata

Valore di ritorno di una funzione: istruzione `return`

- **Esempio:** Funzione che restituisce il massimo tra due interi.

```
int max(int m, int n) {
   if (m >= n)
     return m;
   else
     return n;   }
```

- Chiamata di `max`, ad esempio da `main`:

```
main() {
   int i, j, massimo;
   scanf("%d%d", &i, &j);
   massimo = max(i,j);
   printf("massimo = %d\n", massimo);   }
```

- La funzione `main` tramite i parametri attuali comunica alla funzione `max` i valori (di `i` e `j`) sui quali calcolare la funzione.
- La funzione `max` tramite il valore di ritorno comunica il risultato al `main`.

- Nel corpo deve esserci l'istruzione `return espressione;` la cui esecuzione comporta:
  - il calcolo del valore di `espressione`: questo valore viene restituito al chiamante come risultato dell'esecuzione della funzione
  - la cessione del controllo alla funzione chiamante

Osservazioni

- in `return espressione`, il tipo di `espressione` deve essere lo stesso del tipo del risultato della funzione dichiarato nella definizione
- l'esecuzione di `return espressione` comporta la terminazione dell'esecuzione della funzione

  Esempio:

```
int max(int m, int n) {
 if (m >= n)
   return m;
 else
   return n;
 printf("pippo");   /* non viene mai eseguita */  }
```

Dichiarazioni di funzione (o prototipi)

- I parametri attuali nella chiamata di una funzione devono corrispondere in numero e tipo (in ordine) ai parametri formali.
- Dobbiamo permettere al compilatore di fare questo controllo
  $\Longrightarrow$ prima della chiamata deve essere nota l'intestazione
- Due possibilità:
  1. la funzione è stata definita prima
  2. la funzione è stata dichiarata prima

Sintassi della dichiarazione di funzione (o prototipo)

```
intestazione;
```

ovvero:

```
id-tipo identificatore (parametri-formali);
```

- c'e' un ";" finale al posto del blocco
- nella lista di parametri formali può anche mancare il nome dei parametri — interessa solo il tipo
- il compilatore usa la dichiarazione per controllare che l'attivazione sia corretta
- dopo deve esserci una definizione della funzione coerente con la dichiarazione

Ordine di dichiarazioni e funzioni

- Bisogna dichiarare o definire ogni funzione prima di usarla (chiamarla)
- È pratica comune specificare in quest'ordine:
  1. dichiarazioni (prototipi) di tutte le funzioni (tranne `main`)
  2. definizione di `main`
  3. definizioni delle funzioni
- In questo modo ogni funzione è stata dichiarata prima di essere usata
- L'ordine in cui mettiamo le definizioni non deve necessariamente corrispondere a quello delle dichiarazioni.

Esempio:

```
int max(int, int);

int foo(char, int);

main()  ...

int max(int m, int n)  ...  /* OK. definizione coerente
 con il prototipo */

int foo (int z, char c)  ...  /* NO! definizione non coerente
 con il prototipo */
```

Nella definizione di `foo` i parametri formali non sono nell'ordine specificato dal prototipo.

Passaggio dei parametri

- Abbiamo visto che le funzioni utilizzano parametri
  - permettono uno scambio di dati tra chiamante e chiamato
  - nell'intestazione/prototipo: lista di parametri formali (con tipo associato) – sono delle variabili
  - nell'attivazione: lista di parametri attuali — possono essere delle espressioni
- Al momento della chiamata ogni parametro formale viene inizializzato al valore del corrispondente parametro attuale.
- Il valore del parametro attuale viene copiato nella locazione di memoria del corrispondente parametro formale.
- Questo meccanismo di passaggio dei parametri viene comunemente detto passaggio per valore.

Esempio:

```
int succ (int);   /* prototipo di succ */

   main()                                  int succ (int x)
   {  int z, w;                               { x = x + 1;
      z = 10;                                   return x;
      w = succ(z);                           }
      printf("%d", w);
   }
```

- L'effetto della chiamata `succ(z)` può essere simulato dall'esecuzione della seguente porzione di codice:

```
x = 10;          /* il parametro formale si inizializza con
                    il valore del parametro attuale */
x = x + 1;       /* esecuzione del corpo della funzione
return x;           succ */
```

- Chiamate diverse corrispondono ad inizializzazioni diverse delle variabili corrispondenti ai parametri formali

```
w = succ(20);          x = 20;
                  ⟹    x = x + 1;
                       return x;
```

- In questo caso il valore assegnato alla variabile `w` è `21`.

```
z = 10;
w = succ(z+3);          x = 13;
                   ⟹    x = x + 1;
                        return x;
```

- In questo caso il valore assegnato alla variabile `w` è `14`.
  - Se non vi è corrispondenza perfetta tra il tipo del parametro formale e quello del parametro attuale, viene effettuata una conversione implicita di tipo secondo le regole già viste.
  - Il passaggio dei parametri di tipo array non comporta la copia dei valori dell'array (fra poco ...)

Procedure

- Non sempre le operazioni astratte di cui abbiamo bisogno possono essere descritte in modo naturale come funzioni matematiche.

  Esempio: progettare un'interfaccia utente per la stampa di figure geometriche, in cui l'utente può scegliere:

  1. la forma della figura
  2. la dimensione
  3. il carattere di riempimento
  4. ...

- In questo caso il compito dell'operazione astratta non è (o non è soltanto) produrre un valore, ma è produrre effetti di altro tipo, tipicamente modifiche di stato.

  $\Longrightarrow$ in questi casi possiamo utilizzare procedure

  - le procedure sono un'astrazione delle istruzioni
  - le funzioni sono un'astrazione delle espressioni

Le procedure in C

- Una procedura è una funzione avente come tipo del risultato il tipo speciale `void`.
- La definizione/dichiarazione di procedure e la loro chiamata è analoga al caso delle funzioni

Esempio:

```
void emoticon (int n)
{  /* stampa n volte la sequenza -:) */
   int i;
   for (i=0; i<n; i++)
     {putchar('-');  putchar(':'); putchar(')'); putchar(' ');}
}
main()  {
   ...;
   emoticon(3);
   ...  }
```

- Le procedure non contengono di solito un'istruzione `return` (se la contengono è del tipo `return;` che non comporta il calcolo di alcun valore, ma solo la cessione del controllo al chiamante)

- La semantica di una chiamata di procedura P da una funzione/procedura F è analoga a quella della chiamata di funzione, ma una chiamata di procedura è un'istruzione
- In particolare, il passaggio dei parametri avviene per valore come nel caso delle funzioni
- il controllo viene restituito al chiamante al termine dell'esecuzione del blocco che costituisce il corpo della procedura (o in corrispondenza dell'esecuzione di un'istruzione del tipo `return;`)
- Il C non distingue tra funzioni e procedure (queste ultime sono casi particolari di funzioni)
  $\Longrightarrow$ concettualmente, però, è bene vedere le funzioni come astrazioni di espressioni e le procedure come astrazioni di istruzioni.

Esempio:

Procedura che stampa una cornice di asterischi di "altezza" parametrica

```
void stampaCornice(int altezza)
{
  int i;
  printf("**********************\n");
  for (i=1; i<=altezza; i++)
     printf("*                    *\n");
  printf("**********************\n");
  return;
}
```

- Come astrazione delle istruzioni, le procedure possono dover modificare lo stato.

  Esempio: Procedura `abs` che assegna ad una variabile intera x il suo valore assoluto
  - il chiamante deve comunicare alla procedura la variabile x
  - la procedura deve analizzare il valore della variabile e, se necessario, effettuare il rimpiazzamento
- La seguente realizzazione della procedura non è corretta

```
void abs(int x)
{    if (x < 0)
        x = -x;   }
```

- Simuliamo il comportamento di una chiamata della procedura (come visto in precedenza)

```
int z = -5;
abs(z);        ⟹    x = -5;
                     if (x < 0) x = -x;
```

- La modifica del parametro formale non si ripercuote sul parametro attuale (il passaggio è per valore).

NFA per riconoscere numeri decimali

Vogliamo un NFA che accetta numeri decimali. Un numero decimale consiste di:

1. Un segno + o -, **opzionale**
2. Una stringa di cifre decimali
3. un punto decimale
4. un'altra stringa di cifre decimali

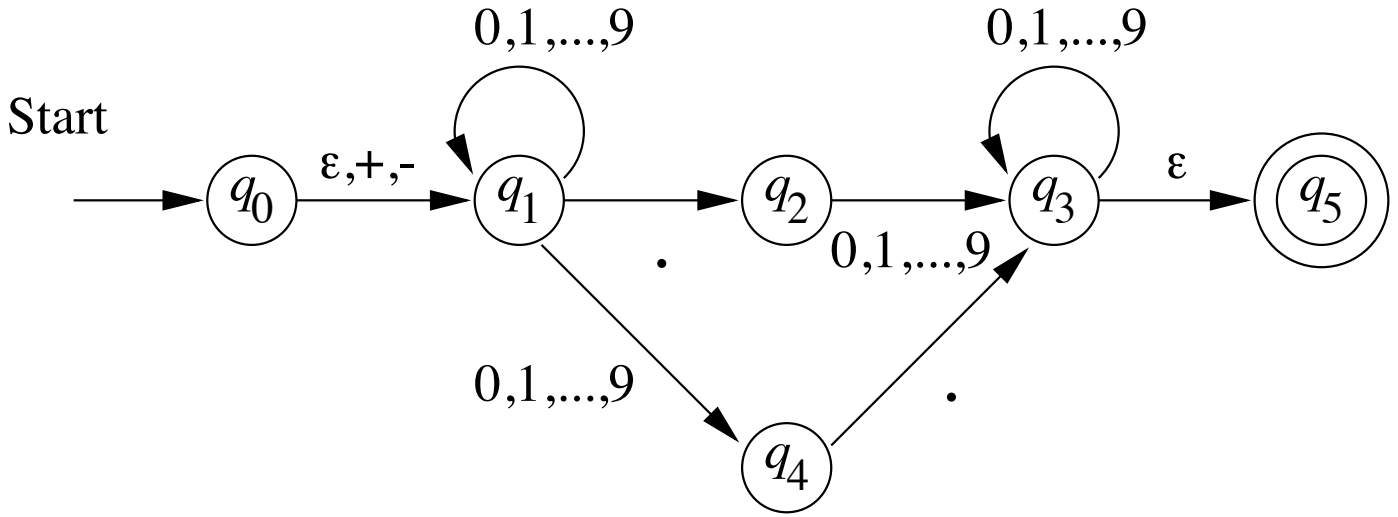
Una delle stringhe (2) e (4), ma non entrambi, può essere vuota.
Cosa succede se non compare il segno?

NFA per riconoscere numeri decimali

Vogliamo un NFA che accetta numeri decimali consiste di:

1. Un segno + o -, **opzionale**
2. Una stringa di cifre decimali
3. un punto decimale
4. un'altra stringa di cifre decimali

Una delle stringhe (2) e (4), ma non entrambi, può essere vuota.
Cosa succede se non compare il segno? Uso un ϵ-NFA.



Definizione ed esempio

Un ϵ-NFA è una quintupla $(Q, \Sigma, \delta, q_0, F)$ dove:

- $\delta : Q \times \Sigma \cup \{\epsilon\} \to 2^Q$ è una funzione da $Q \times \Sigma \cup \{\epsilon\}$ all'insieme dei sottoinsiemi di Q.

Esempio: L'ϵ-NFA della pagina precedente

$$E = (\{q_0, q_1, \ldots, q_5\}, \{., +, -, 0, 1, \ldots, 9\}\ \delta, q_0, \{q_5\})$$

dove la tabella delle transizioni (con una colonna per ϵ) per δ è

	ϵ	+,-	.	$0, \ldots, 9$
$\to q_0$	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
$\star q_5$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Epsilon-chiusura

ECLOSE(q) = gli stati raggiungibili da q tramite una sequenza di ϵ-transizioni

Definizione induttiva di ECLOSE(q)

Base:

$q \in \text{ECLOSE}(q)$

Induzione:

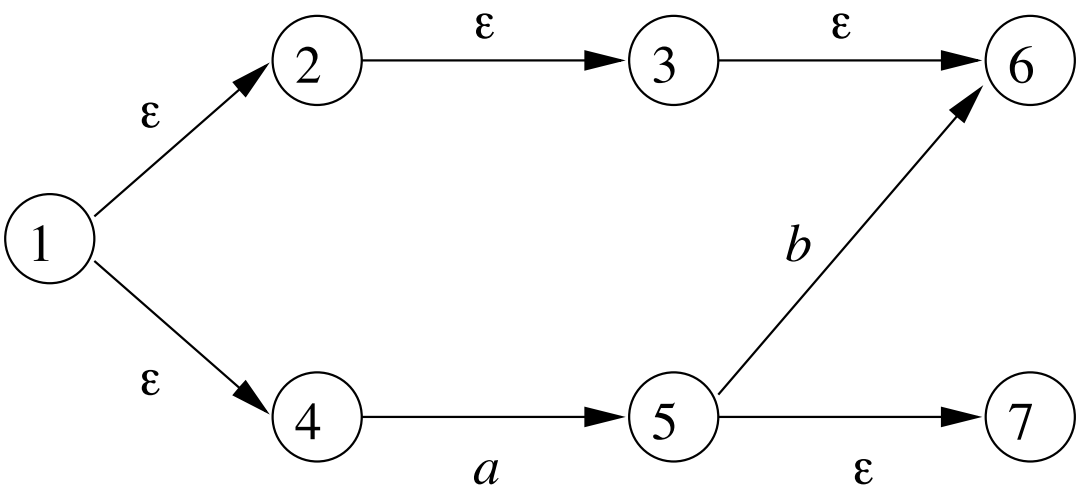
$p \in \text{ECLOSE}(q)$ and $r \in \delta(p, \epsilon) \Rightarrow r \in \text{ECLOSE}(q)$

Nell'esempio di prima ogni stato coincide con la propria epsilon-chiusura, tranne $\text{ECLOSE}(q_0) = \{q_0, q_1\}$ e $\text{ECLOSE}(q_3) = \{q_3, q_5\}$ che corrispondono alle due ϵ-transizioni.

La stessa operazione si può applicare ad un insieme di stati:

$$\text{ECLOSE}(S) = \bigcup_{q \in S} \text{ECLOSE}(q)$$

Esempio di epsilon-chiusura



Per esempio,

$$\text{ECLOSE}(1) = \{1, 2, 3, 4, 6\}$$

Ognuno di questi stati può essere raggiunto a partire da 1 attraverso ϵ-transizioni.

Funzione di transizione estesa $\hat{\delta}$ e il linguaggio accettato

Intuitivamente $\hat{\delta}(q, w)$ è l'insieme degli stati raggiungibili da q seguendo il percorso w (con possibili ϵ-transizioni)

- Definizione induttiva di $\hat{\delta}$ per automi ϵ-NFA

Base:

$$\hat{\delta}(q, \epsilon) = \text{ECLOSE}(q)$$

Induzione: $a \neq \epsilon$

Se

- $\hat{\delta}(q, x) = \{p_1, ..., p_k\}$;
- $\bigcup_{p_i \in \hat{\delta}(q,x)} \delta(p_i, a) = \{r_1, ..., r_m\}$

$$\hat{\delta}(q, xa) = \text{ECLOSE}(\bigcup_{p_i \in \hat{\delta}(q,x)} \delta(p_i, a)) = \text{ECLOSE}\{r_1, ..., r_m\}$$

Il linguaggio di un ϵ-NFA E è dato da:

$$L(E) = \{w | \hat{\delta}(q_0, w) \cap F_E \neq \emptyset\}$$

Da ϵ-NFA a DFA

Procedimento simile alla costruzione per sottoinsiemi: vanno incorporate le ϵ-transizioni, usando le ϵ-chiusure.
Dato un ϵ-NFA

$$E = (Q_E, \Sigma, \delta_E, q_0, F_E)$$

costruiremo un DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

tale che

$$L(D) = L(E)$$

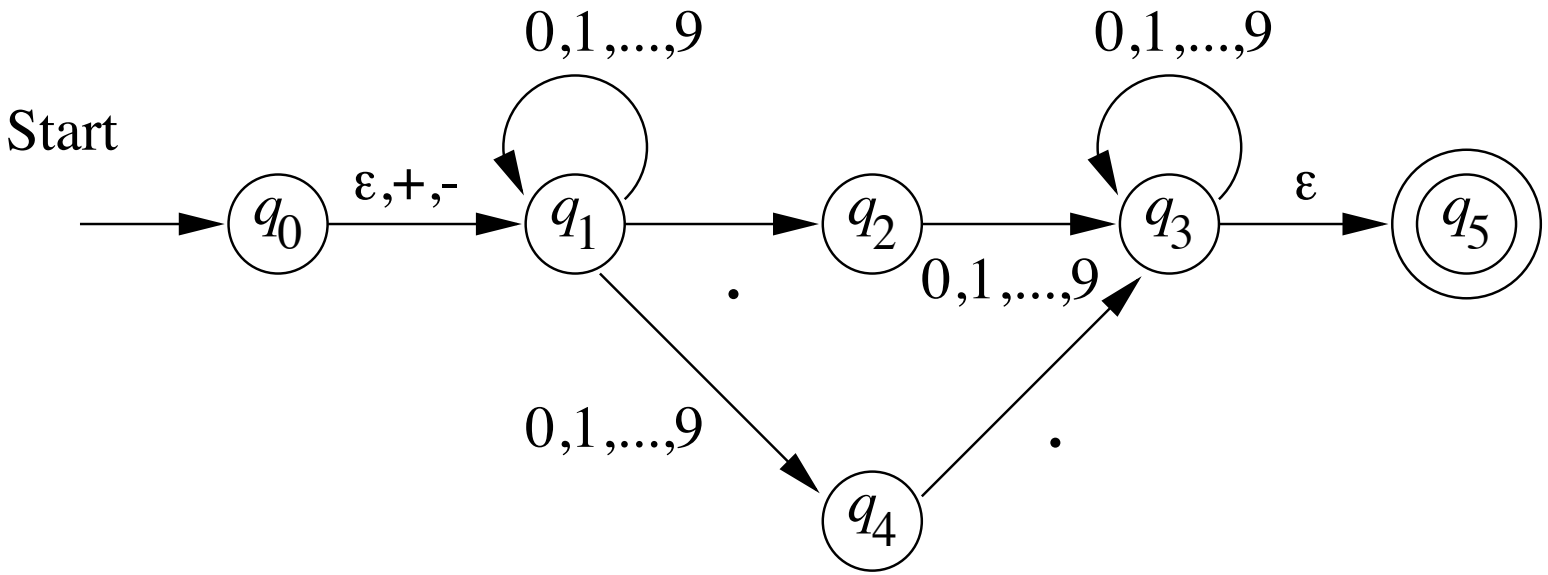
Dettagli della costruzione:

- $Q_D = \{S : S \subseteq Q_E \text{ e } S = \text{ECLOSE}(S)\}$
- $q_D = \text{ECLOSE}(q_0)$
- $F_D = \{S : S \in Q_D \text{ e } S \cap F_E \neq \emptyset\}$
- $\delta_D(S, a) =$

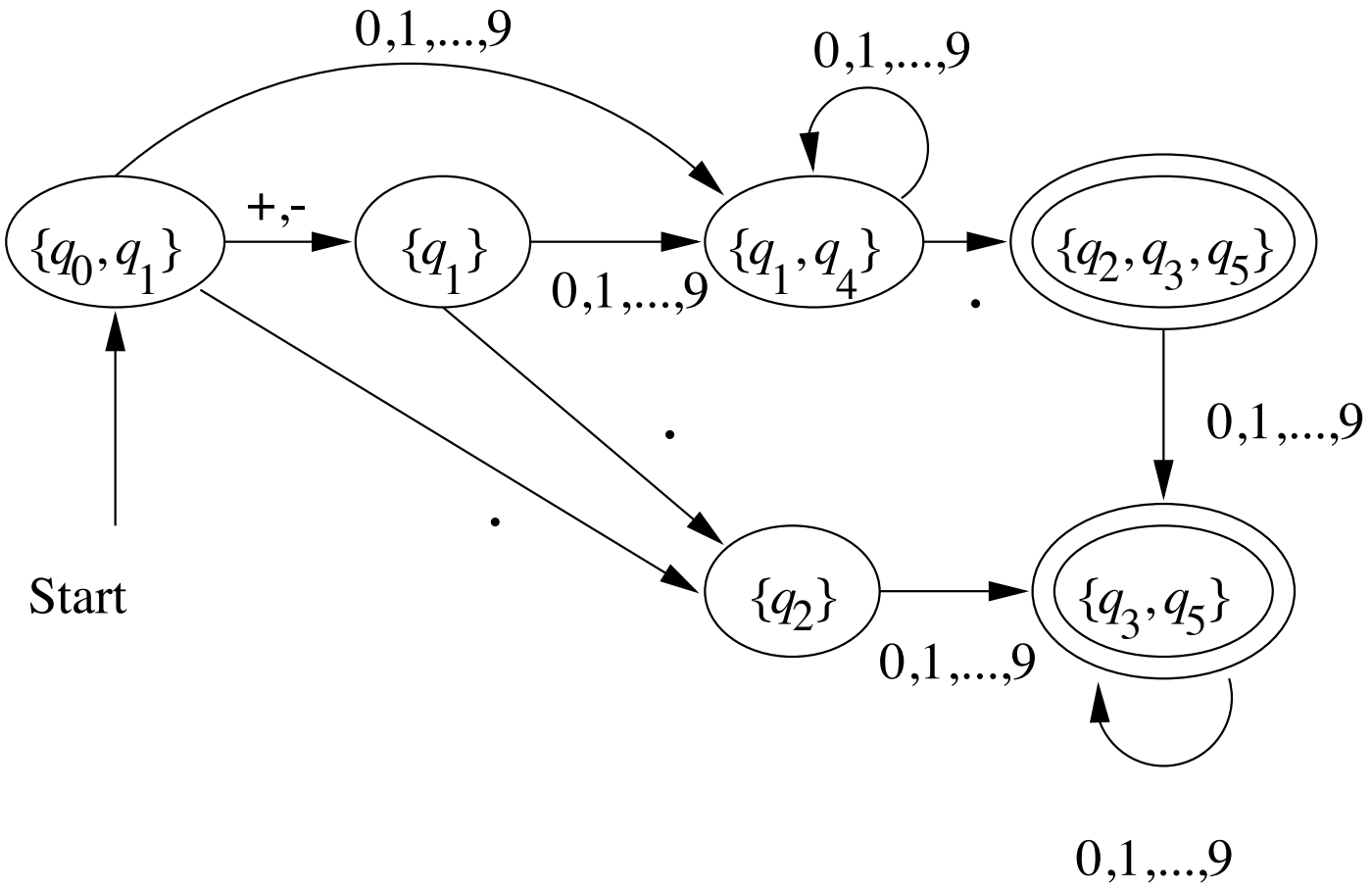
$$\bigcup \{\text{ECLOSE}(p) : p \in \delta(t, a) \text{ per alcuni } t \in S\}$$

Esempio

ϵ-NFA E per riconoscere numeri decimali in notazione anglo-sassone



DFA D corrispondente ad E



Teorema 2.22: Un linguaggio L è accettato da un ϵ-NFA E se e solo se L è accettato da un DFA.
Dimostrazione: Usiamo D costruito come sopra e mostriamo per induzione che $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$

Base: $\hat{\delta}_E(q_0, \epsilon) = \text{ECLOSE}(q_0) = q_D = \hat{\delta}(q_D, \epsilon)$

Induzione: $a \neq \epsilon$ Sia $w = xa$ con l'ipotesi valida per x, ovvero $\hat{\delta}_E(q_0, x) = \hat{\delta}_D(q_0, x) = \{p_1, ..., p_k\}$. Per la definizione di $\hat{\delta}_E$, $\hat{\delta}_E(q_0, w)$ si calcola come segue:

- sia $\bigcup_{p_i \in \hat{\delta}_E(q_0,x)} \delta(p_i, a) = \{r_1, ..., r_m\}$,
- allora

$$\hat{\delta}_E(q, xa) = \text{ECLOSE}(\bigcup_{p_i \in \hat{\delta}(q,x)} \delta(p_i, a)) = \text{ECLOSE}\{r_1, ..., r_m\}$$

Ricapitolando

- Tutti i tipi di automi visti, DFA, NFA e ϵ-NFA, accettano lo stesso insieme di linguaggi: i linguaggi regolari.
- Solo i DFA possono tuttavia essere implementati!

Variabili locali

- Il blocco che costituisce il corpo di una funzione/procedura può contenere dichiarazioni di variabili.

  Esempio:

  - sono variabili proprie della funzione
  - hanno tempo di vita limitato alla durata della chiamata
  - più in generale: un identificatore dichiarato nel corpo di una funzione è detto locale alla funzione e non è visibile all'esterno della funzione (ad esempio nel `main`), ma solo nel corpo della stessa
- In realtà, ciò non è altro che un caso particolare di regole generali che governano la visibilità e il tempo di vita degli identificatori di un programma.

Struttura generale di un programma C

- parte direttiva
- parte dichiarativa globale che comprende (ricordarsi l'acrostico La Cosa Tre Volte Più Facile):
  - dichiarazioni di Costanti
  - dichiarazioni di Tipi (li vedremo ...)
  - dichiarazioni di Variabili (variabili globali)
  - prototipi di Procedure/Funzioni
- il programma principale (`main`)
- le definizioni di funzioni/procedure

Esempio

```
#include <stdio.h>        /* parte direttiva */
#define LUNG 10

int i = 1;               /* variabili globali */
int j = 2;

int Q(int);              /* prototipi di funzioni e procedure */
void R(float);

main()                  /* programma principale */
{
int x = 10;
float y = 3.4;
char c = 'a';
x = Q(x);
R(y);
}

int Q(int v) { ... }    /* definizioni di funzioni e procedure */
void R(float z) { ... }
```

Blocchi

- il corpo di una funzione/procedura, come il corpo del programma principale, è un blocco.
- In C un blocco è costituito da
  - una parte dichiarativa (può non esserci)
  - una parte esecutiva (sequenza di istruzioni)
- Nel `main` o nel corpo delle funzioni possono comparire diversi blocchi, che possono essere
  - annidati: un blocco è una delle istruzioni di un altro blocco
  - paralleli: blocchi che fanno parte della medesima sequenza di istruzioni

```
{
   int x;
   x = 10;
   {
     int z;
     z = 20 ;
     ...
   }
   ...
}
```

```
{
   int x;
   x = 10;
   ...
}
{
   int z;
   z = 20;
   ...
}
```

- Anche la parte esecutiva del programma principale e di una funzione/procedura è un blocco
- Gli identificatori dichiarati nella parte dichiarativa di un blocco sono detti nomi locali del blocco e devono essere tutti diversi tra loro
  - nel caso di una funzione/procedura, fanno parte dei nomi locali anche gli identificatori utilizzati per i parametri formali

Esempio:

```
{
int x;   /* NO! identificatore x dichiarato   */
char x;  /* due volte nello stesso blocco     */
...
}

void p(int x, char y)
{
int x;      /* NO! identificatore x gia' usato per un parametro formale */
...
}
```

- In blocchi diversi possono essere utilizzati gli stessi identificatori

Esempio:

```
main()
{
int x;      /* x, y: variabili locali del main */
int y;
...
   {
     char x;     /* x: variabile locale del blocco annidato */
     ...
   }
...
}

void p(int x)
{
int y;          /*x,y: variabili locali della procedura p  */
...
}
```

- Un programma C può avere una struttura molto complessa a seguito dell'uso di funzioni, procedure e blocchi.
- È necessario definire regole precise per regolamentare l'uso dei nomi utilizzati all'interno di un programma.
- A questo scopo introduciamo alcune definizioni utili.

  Ambiente globale: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa globale del programma

  Ambiente locale di una funzione: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa della funzione e nella sua intestazione

  Ambiente locale di un blocco: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa del blocco

- Quanto detto informalmente in precedenza può essere meglio precisato:

  $\Longrightarrow$ è possibile dichiarare più volte lo stesso identificatore (anche con significati diversi) purché in ambienti diversi
- Se ciò evita il proliferare di identificatori, causa il problema di stabilire il significato di un riferimento ad un identificatore in un generico punto del programma

Esempio: Riprendiamo l'esempio precedente

```
main()
{
int x;      /* x, y: variabili locali del main */
int y;
...
   {
     char x;    /* x: variabile locale del blocco annidato */
     ...
   }
...
}

void p(int x)
{
int y;          /*x,y: variabili locali della procedura p  */
...
}
```

- Se in un punto del programma viene eseguita l'istruzione `x = ...`, a quale delle tre dichiarazioni di `x` ci si riferisce?
- Dipende dal punto in cui si trova tale assegnamento e dalle regole di visibilità (o regole di scoping).

Regole di visibilità

Gli identificatori presenti nell'ambiente

- globale sono visibili in tutte le funzioni e in tutti i blocchi del programma.
  N.B. Gli identificatori predefiniti del linguaggio si intendono parte dell'ambiente globale.
- locale di una funzione sono visibili nel corpo della funzione (ivi compresi eventuali blocchi in esso contenuti).
- locale di un blocco sono visibili nella parte esecutiva del blocco (ivi compresi eventuali blocchi in essa contenuti).

Se un identificatore è definito in più punti, la definizione valida è quella dell'ambiente più vicino al punto di utilzzo. In particolare, una variabile locale "nasconde" una eventuale variabile omonima definita nell'ambiente superiore (shadowing o oscuramento) per tutto il blocco.

- Detto altrimenti, l'ambito di visibilità di un identificatore è determinato dalla posizione della sua dichiarazione:
  - gli identificatori dichiarati all'interno di un blocco hanno ambito di visibilità a livello di blocco
    $\Longrightarrow$ una variabile dichiarata in un blocco è visibile solo in quel blocco (compresi eventuali blocchi annidati)
  - gli identificatori dichiarati all'interno di una funzione (compresi quelli nell'intestazione) hanno ambito di visibilità a livello di funzione
    $\Longrightarrow$ una variabile dichiarata in una funzione è visibile solo nel corpo della funzione (compresi eventuali blocchi annidati)
  - gli identificatori dichiarati all'esterno delle funzioni e del main hanno ambito di visibilità a livello di programma
    $\Longrightarrow$ una variabile globale è visibile ovunque nel programma

Esempio:

```
int x1=10, x2=20;
char c='a';

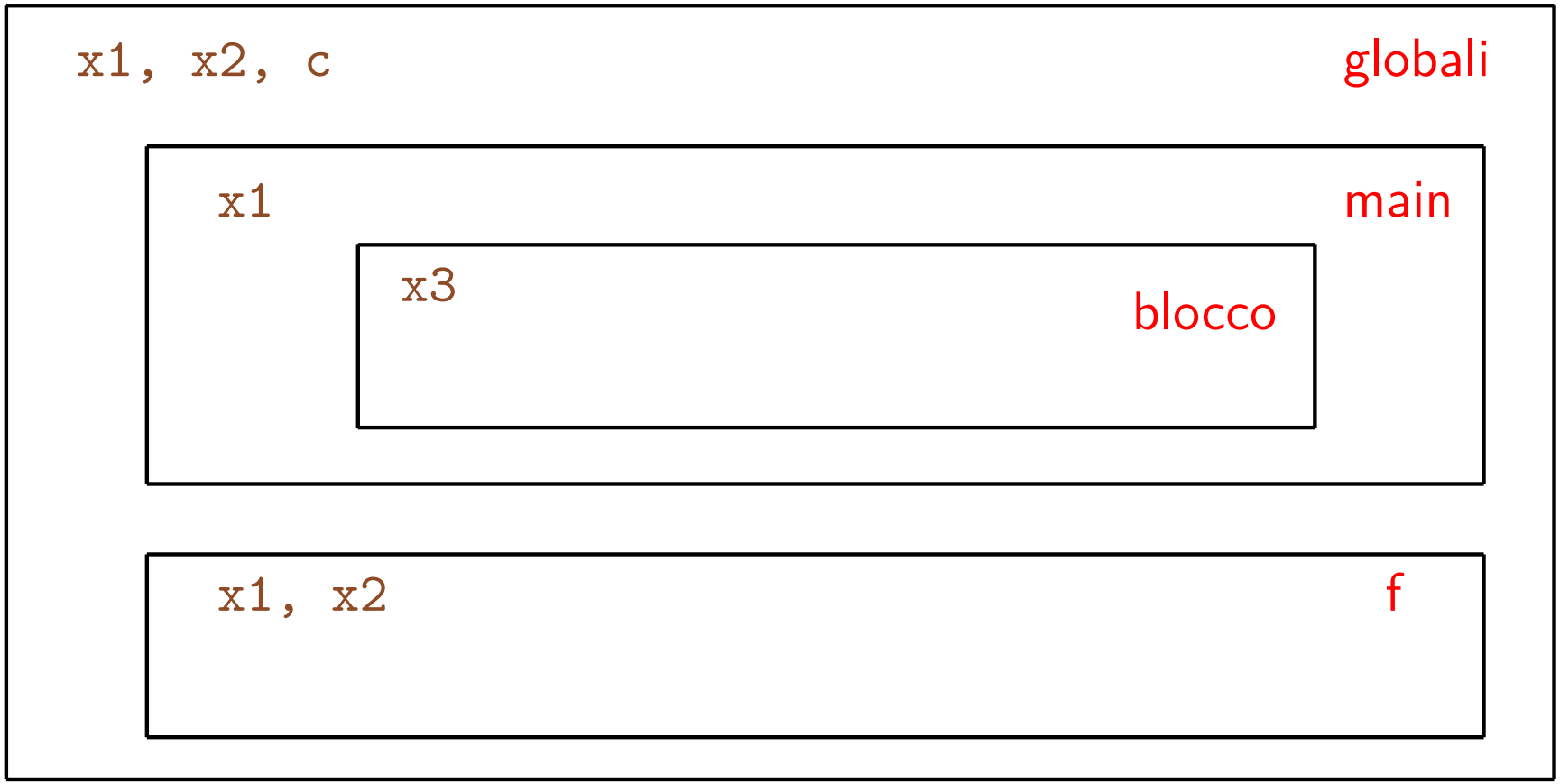
int f(int);

main()
{
int x1=30;    /* nasconde la variabile globale x1 */
x2 = x1+x2;   /* x1 e' quella locale, x2 e' globale */
printf("x1=%d   x2=%d\n", x1, x2);   /* stampa x1=30  x2=50  */
  { int x3=50;
    x1=f(x3); /* x1 e' quella locale al primo blocco */
    printf("x1=%d   x2=%d\n", x1, x2);   /* stampa x1=150  x2=50 */
  }
}

int f(int x1)  /* nasconde la variabile globale x1 */
{ int x2;       /* nasconde la variabile globale x2 */
  x2 = x1 + 100;  /* x1 e' il parametro formale, x2 la var. locale */
  return x2;
}
```

Rappresentazione Grafica: Modello a contorni

- Si rappresenta ogni ambiente mediante un rettangolo con gli identificatori in esso contenuti.



Durata delle variabili

- Una variabile ha un suo tempo di vita.
  - viene creata (ovvero ad essa viene riservata uno spazio di memoria)
  - viene (o può essere) distrutta (ovvero viene rilasciato il corrispondente spazio di memoria).
- Si distinguono due classi di variabili:
  - variabili automatiche: vengono create ogni volta che si entra nel loro ambiente di visibilità e vengono distrutte all'uscita di tale ambiente
    - es. variabili locali di un blocco: vengono create all'ingresso del blocco, {, e distrutte all'uscita dal blocco, }.
    - es. variabili locali di una funzione: vengono create al momento della chiamata e distrutte all'uscita
  - variabili statiche: vengono create una sola volta e vengono distrutte solo al termine dell'esecuzione del programma (non ne faremo uso ...)

Durata delle variabili (cont.)

Nel caso di funzioni/blocchi eseguiti più volte (es. funzione chiamata in punti diversi, blocco all'interno di un ciclo):

- le variabili automatiche corrispondenti possono essere associate di volta in volta a locazioni di memoria diverse, quindi
- il loro valore non persiste tra una esecuzione e la successiva

Gestione della memoria a tempo di esecuzione (run-time)

- La memoria per
  - il codice macchina è fissata a tempo di compilazione
  - i dati (in particolare per le variabili automatiche) cresce e decresce dinamicamente durante l'esecuzione: viene gestita a pila
- Ricordiamo che una pila (o stack) è una struttura dati con accesso LIFO: Last In First Out (ultimo entrato, primo servito), come ad es. la pila di piatti da lavare o di pratiche da svolgere.

Pila dei record di attivazione

- Il sistema gestisce in memoria la pila dei record di attivazione (RDA)
  - per ogni chiamata di funzione viene creato (e allocato) un nuovo RDA in cima alla pila
  - al termine della chiamata della funzione il RDA viene rimosso dalla pila
- Ogni RDA contiene:
  - le locazioni di memoria per i parametri formali (se presenti)
  - le locazioni di memoria per le variabili locali (se presenti)
  - altre informazioni che non analizziamo
- Anche gli ambienti locali dei blocchi vengono allocati/deallocati sulla pila.

Pila dei record di attivazione: come funzionano

- Il primo record di attivazione (RDA) è per il `main()`
- Ad ogni attivazione viene allocato un RDA
- Al termine dell'attivazione il record viene rilasciato (liberando la memoria che occupava)
- La dimensione dell'RDA è nota in fase di compilazione
- Il numero di attivazioni della funzione non è noto

Esempio:

```
int f(int);
main()
{
  int x, y, z;
  x=10;
  y=20;          /* blocco principale */                          PUNTO 1
  z = f(x);      /* prima chiamata di f */                        PUNTO 2
    {
    int x=50;    /* uscita da f e ingresso nel blocco annidato*/  PUNTO 3
    y=f(x);      /* seconda chiamata di f */                      PUNTO 4
    z=y;         /* uscita da f */                                PUNTO 5
    }
  ...            /* uscita dal blocco */                          PUNTO 6
}
int f(int a)
{
  int z;
  z = a + 1;
  return z;
}
```

Evoluzione della pila

x	10
y	20
z	?

▸ PUNTO 1

Evoluzione della pila

a	10
z	?

x	10
y	20
z	?

▸ PUNTO 2

Evoluzione della pila

x	50

x	10
y	20
z	11

▸ PUNTO 3

Evoluzione della pila

a	50
z	?

x	50

x	10
y	20
z	11

▸ PUNTO 4

Evoluzione della pila

x	50

x	10
y	51
z	11

▸ PUNTO 5

Evoluzione della pila

x	10
y	51
z	51

▸ PUNTO 6

Variabili statiche: un esempio d'uso

- Una variabile statica, una volta creata, rimane in vita per tutto il tempo di esecuzione del programma.

  Esempio: `f(void) { static int x; ... }`

  - la variabile viene inizializzata alla prima attivazione della funzione
  - conserva il suo valore tra attivazioni successive
  - è locale, quindi visibile solo all'interno della funzione in cui è dichiarata

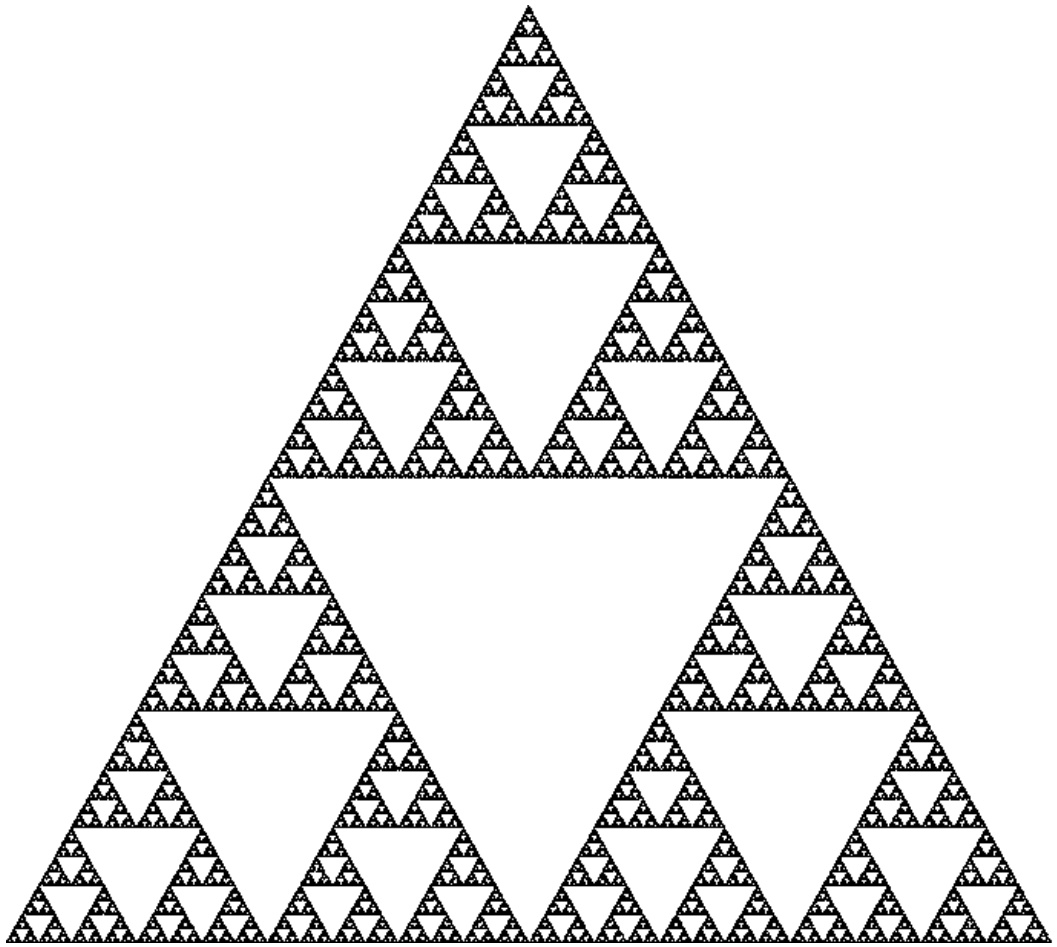
  Esempio: Funzione che restituisce il numero di volte che è stata attivata.

```
int fun1(void) {
   static int conta = 0;
       /* variabile locale statica visibile solo in fun1;
          contatore del numero di attivazioni di fun1    */
   .....
   conta++;
   return conta;
   }
```

Ricorsione: C'era una volta un Re

- C'era una volta un Re
  seduto sul sofà
  che disse alla sua serva
  raccontami una storia
  e la serva incominciò:
  - C'era una volta un Re
    seduto sul sofà
    che disse alla sua serva
    raccontami una storia
    e la serva incominciò:
    - C'era una volta un Re
      seduto sul sofà...

Il triangolo di Sierpinskyi

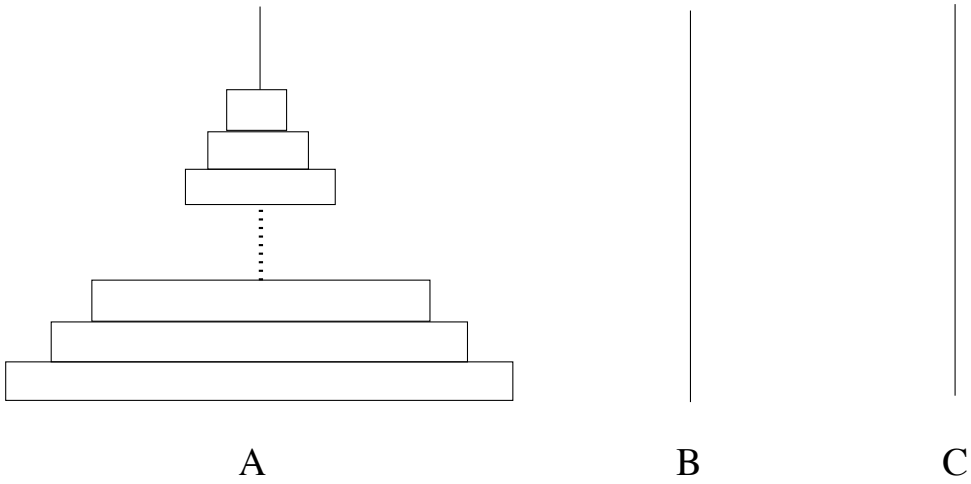


Programmazione ricorsiva: cenni

- In quasi tutti i linguaggi di programmazione evoluti è ammessa la possibilità di definire funzioni/procedure ricorsive: durante l'esecuzione di una funzione F è possibile chiamare la funzione F stessa.
- Ciò può avvenire
  - direttamente: il corpo di F contiene una chiamata a F stessa.
  - indirettamente: F contiene una chiamata a G che a sua volta contiene una chiamata a F.
- Questo può sembrare strano: se pensiamo che una funzione è destinata a risolvere un sottoproblema $\mathcal{P}$, una definizione ricorsiva sembra indicare che per risolvere $\mathcal{P}$ dobbiamo ...saper risolvere $\mathcal{P}$!

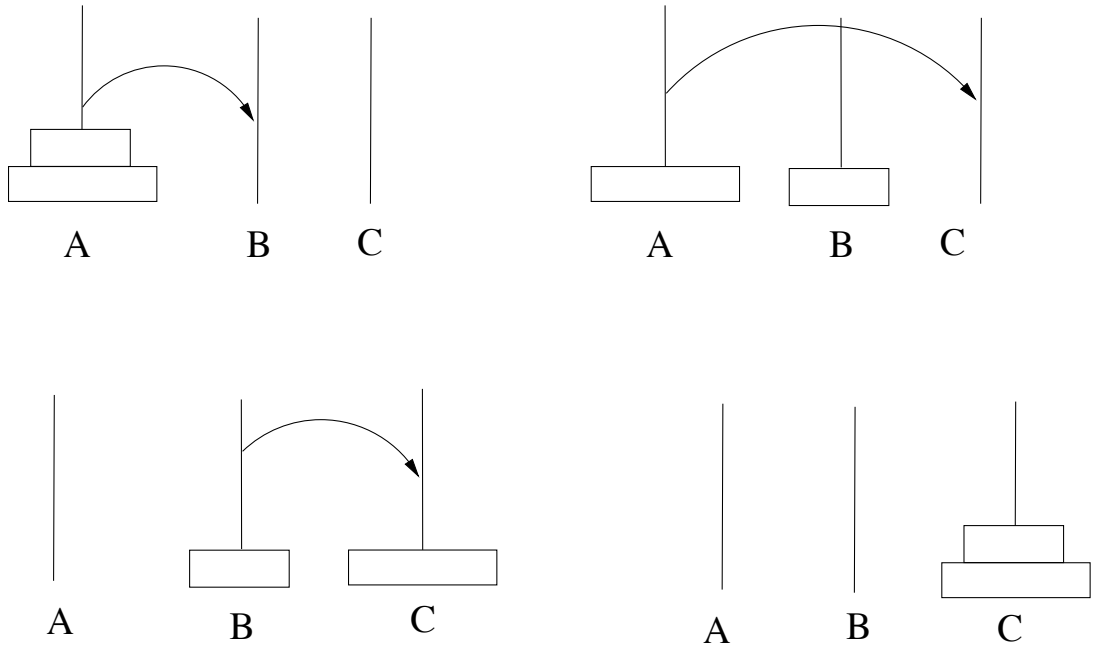
- In realtà, la programmazione ricorsiva si basa sull'osservazione che per molti problemi la soluzione per un caso generico può essere ricavata sulla base della soluzione di un altro caso, generalmente più semplice, dello stesso problema.
- La programmazione ricorsiva trova radici teoriche nel principio di induzione ben fondata che può essere visto come una generalizzazione del principio di induzione sui naturali
- La soluzione di un problema viene individuata supponendo di saperlo risolvere su casi più semplici.
- Bisogna poi essere in grado di risolvere direttamente il problema sui casi più semplici di qualunque altro.

Esempio: Torre di Hanoi (leggenda Vietnamita).



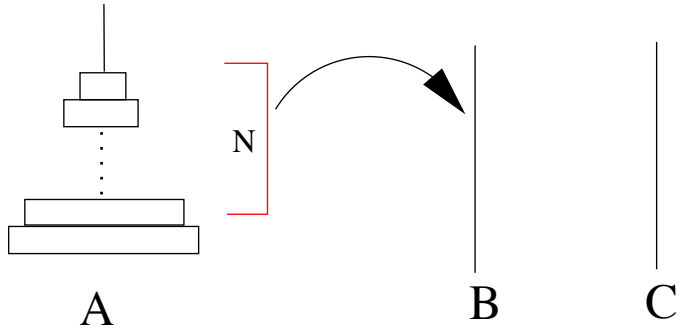
- pila di dischi di dimensione decrescente su un perno A
- vogliamo spostarla sul perno C, usando un perno di appoggio B
- vincoli:
  - possiamo spostare un solo disco alla volta
  - un disco più grande non può mai stare su un disco più piccolo
- secondo la leggenda: i monaci stanno spostando 64 dischi: quando avranno finito, ci sarà la fine del mondo

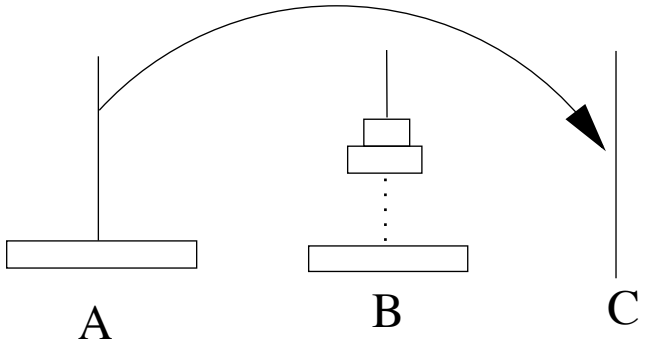
- Come individuare una soluzione per un numero N di dischi arbitrario?
  - per N=1 la soluzione è immediata: spostiamo l'unico disco da A a C
  - se sappiamo risolvere il problema per N=1 lo sappiamo risolvere anche per N=2: come?

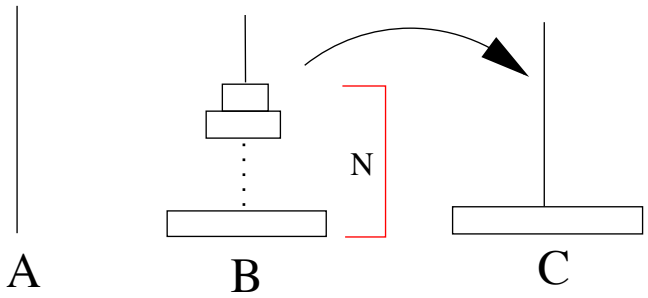


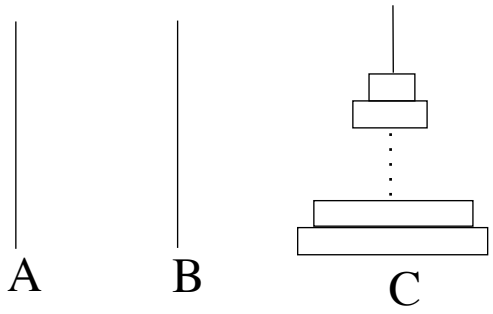
- Notiamo l'utilizzo del perno ausiliario B

- Possiamo generalizzare il ragionamento? Se sappiamo risolvere il problema per N dischi, possiamo individuare una soluzione per lo stesso problema ma con N+1 dischi?









- Formalizziamo il ragionamento
- Indichiamo con hanoi(N, P1, P2, P3) il problema:"spostare N dischi dal perno P1 al perno P2 utilizzando P3 come perno d'appoggio".

```
hanoi(N, P1, P2, P3)
     if (N=1)
         sposta  da P1 a P2;
     else
       {
         hanoi(N-1, P1, P3, P2);
         sposta da P1 a P2;
         hanoi(N-1, P3, P2, P1);
       }
```

Esempio: Soluzione di hanoi(3,A,C,B)

hanoi(3,A,C,B) =
- hanoi(2,A,B,C) =
  - hanoi(1,A,C,B) = sposta(A,C)
  - sposta(A,B)
  - hanoi(1,C,B,A) = sposta(C,B)
- sposta(A, C)
- hanoi(2,B,C,A) =
  - hanoi(1,B,A,C) = sposta(B,A)
  - sposta(B,C)
  - hanoi(1,A,C,B) = sposta(A,C)

Quante mosse per N dischi?

Si può dimostrare, per induzione sul numero di dischi N, che il numero di mosse è:

$$Mosse(N) = \begin{cases} 1 & \text{se } N = 1 \quad \text{(caso base)} \\ 2^N - 1 & \text{se } N > 1 \quad \text{(caso induttivo)} \end{cases}$$

Supponendo che ogni mossa duri un secondo, i monaci avrebbero da lavorare per più di 580 miliardi di anni.
Sapendo che l'universo ha circa una ventina di miliardi di anni, possiamo dire che i monaci ne avranno ancora per molto ☺

- Le funzioni ricorsive sono convenienti per implementare funzioni matematiche definite in modo induttivo.

  Esempio: Definizione induttiva di somma tra due interi non negativi:

$$somma(x, y) = \begin{cases} x & \text{se } y{=}0 \\ 1 + (somma(x, y-1)) & \text{se } y > 0 \end{cases}$$

  - La somma di x con 0 viene definita in modo immediato;
  - la somma di x con il successore di y viene definita come il successore della somma tra x e y.

- **Esempio:** somma di 3 e 2:

$$\begin{aligned} somma(3,2) = \quad & 1 + (\underline{somma(3,1)}) = \\ & 1 + (1 + (\underline{somma(3,0)})) = \\ & 1 + (1 + (3)) = \\ & 1 + 4 = \\ & 5 \end{aligned}$$

Esempio: Funzione fattoriale.

- definizione iterativa: $fatt(n) = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$
- definizione induttiva:

$$fatt(n) = \begin{cases} 1 & \text{se } n = 0 \quad \text{(caso base)} \\ n \cdot fatt(n-1) & \text{se } n > 0 \quad \text{(caso induttivo)} \end{cases}$$

- È essenziale il fatto che, applicando ripetutamente il caso induttivo, ci riconduciamo prima o poi al caso base.

$$\begin{aligned} fatt(3) = \quad & 3 \cdot \underline{fatt(2)} = \\ & 3 \cdot (2 \cdot \underline{fatt(1)}) = \\ & 3 \cdot (2 \cdot (1 \cdot \underline{fatt(0)})) = \\ & 3 \cdot (2 \cdot (1 \cdot 1)) = \\ & 3 \cdot (2 \cdot 1) = \\ & 3 \cdot 2 = \\ & 6 \end{aligned}$$

Il codice delle due diverse versioni

- definizione iterativa:

```
int fatt(int n) {
    int i,ris;

    ris=1;
    for (i=1;i<=n;i++)
      ris=ris*i;
    return ris;
}
```

- definizione ricorsiva:

```
int fattric(int n) {
  if (n == 0)
    return 1;
  else
    return n * fattric(n-1);
 }
```

Esempio: Programma che usa una funzione ricorsiva.

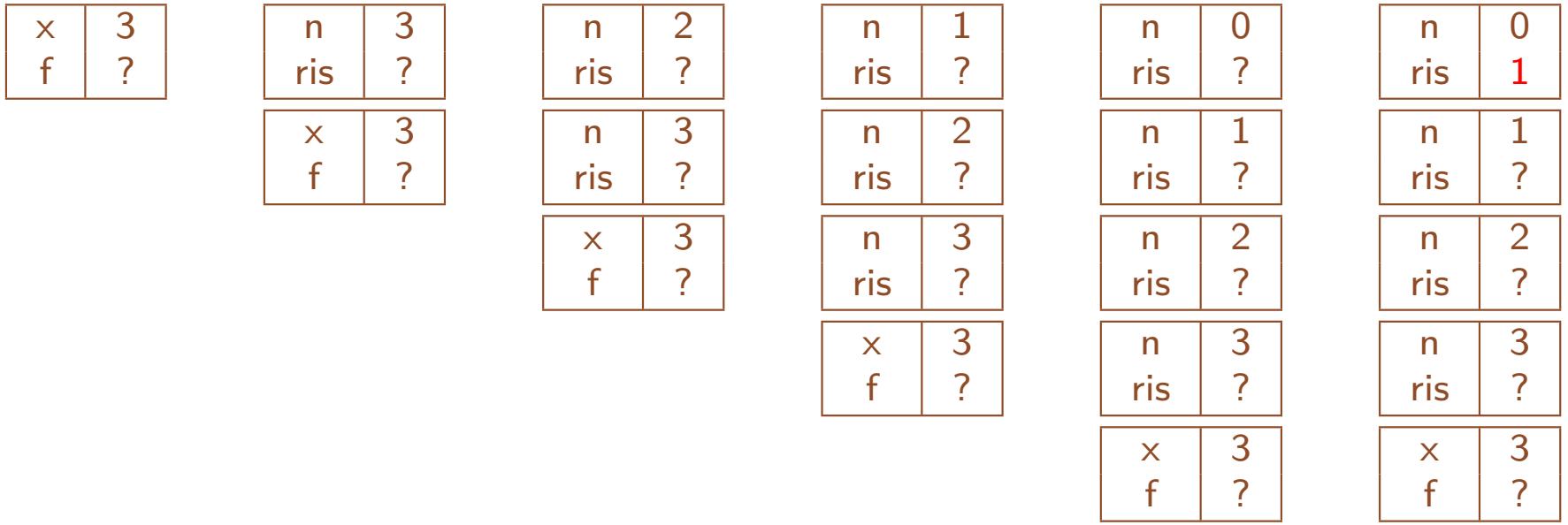
```
#include <stdio.h>

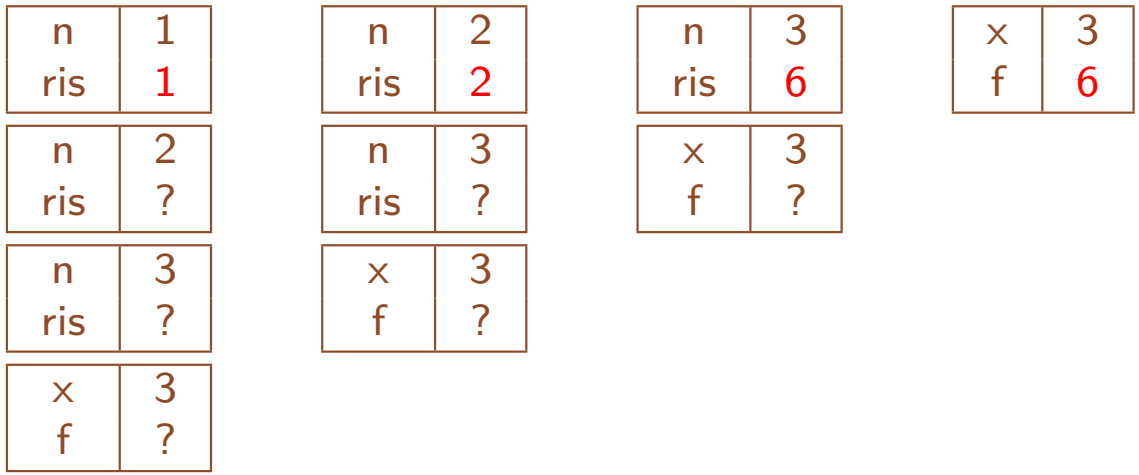
int fattric (int);

main()
{
int x, f;
scanf("%d", &x);
f = fattric(x);
printf("Fattoriale di %d:  %d\n", x, f);
}

int fattric(int n) {
  int ris;
  if (n == 0)
    ris = 1;
  else
    ris = n * fattric(n-1);
  return ris;
}
```

Evoluzione della pila (supponendo x=3).





Esempio: Leggere una sequenza di caratteri terminata da `'\n'` e stamparla invertita. Ad esempio: `rosa` $\Longrightarrow$ `asor`

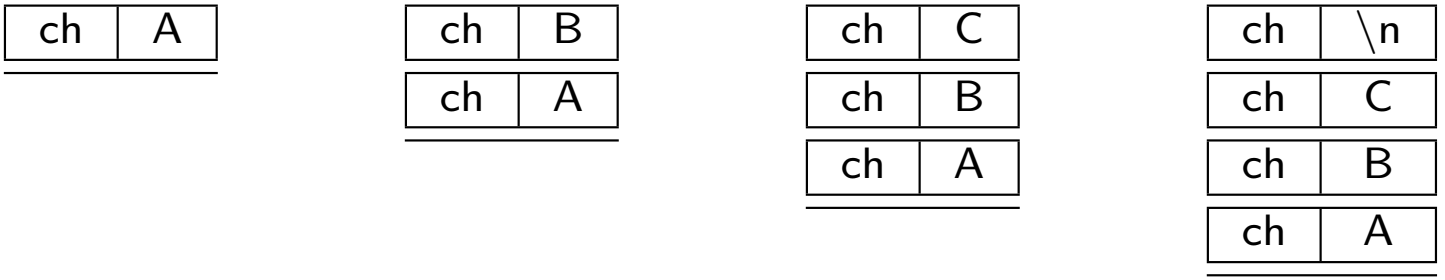
- Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
  1. usando una struttura dati opportuna ma dinamica (liste, le vedremo più avanti)
  2. usando un procedimento ricorsivo.
     - leggiamo un carattere della sequenza, `c1`, leggiamo e stampiamo ricorsivamente il resto della sequenza `c2...cn` e infine stampiamo `c1`;
     - il caso base è la lettura del carattere di fine sequenza.

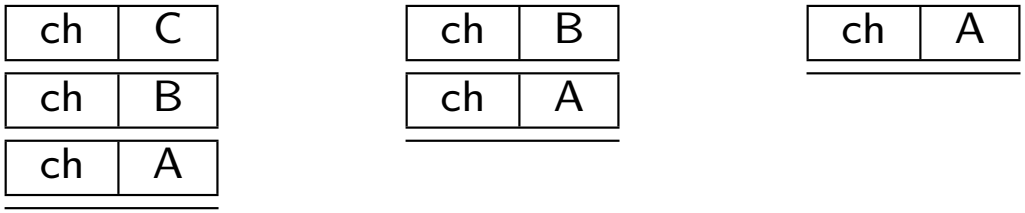
```
void invertInputRic()
 {  char ch;

  ch = getchar();
  if (ch != '\n')
    {
    invertInputRic();
    putchar(ch);
    }
  else
    printf("Sequenza invertita: ");
}
```

```
 main()
{
  printf("Immetti una sequenza di caratteri\n");
  invertInputRic();
  printf("\n");
}
```

Vediamo come si evolve la pila per l'input `ABC\n`





L'output prodotto è il seguente

```
Sequenza invertita: CBA
```

Ricorsione multipla

- Si ha ricorsione multipla quando un'attivazione di una funzione può causare più di una attivazione ricorsiva della stessa funzione (es. torre di Hanoi)

  Esempio: Definizione induttiva dei numeri di Fibonacci.

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-2) + F(n-1) \qquad \text{se } n > 1 \end{aligned}$$

- $F(0)$, $F(1)$, $F(2)$, ... è detta sequenza dei numeri di Fibonacci: $0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ \ldots$

```
 #include <stdio.h>

int fibonacci (int);

main() {
  int n;

  printf("Inserire un intero >= 0: ");
  scanf("%d", &n);
  printf("Numero %d di Fibonacci: %d\n", n, fibonacci(n));
}

 int fibonacci(int i)
  {
  int ris;
  if (i == 0)
    ris = 0;
  else if (i == 1)
    ris = ;
  else
    ris =  fibonacci(i-1) + fibonacci(i-2);
  return ris;
  }
```

Esempi di funzioni ricorsive

- Tradurre in C la definizione induttiva già vista:

$$somma(x, y) = \begin{cases} x & \text{se } y = 0 \\ 1 + (somma(x, y-1)) & \text{se } y > 0 \end{cases}$$

```
int somma (int x, int y)
{
  int ris;
  if (y==0)
     ris = x;
  else
     ris = 1 + somma(x, y-1);
  return ris;
}
```

- Calcolo ricorsivo di x^y (si assume $y \geq 0$)

$$x^y = \begin{cases} 1 & \text{se } y = 0 \\ x \cdot x^{y-1} & \textit{altrimenti} \end{cases}$$

```
int exp (int x, int y)
{
  int ris;
  if (y==0)
     ris = 1;
  else
     ris = x * exp(x, y-1);
  return ris;
}
```

- Calcolare ricorsivamente la somma degli elementi nella porzione di un array v compresa tra gli indici from e to.
- Esprimiamo formalmente quanto richiesto:

$$sumVet(v, from, to) = \sum_{i=from}^{to} v[i]$$

- È evidente che:

$$\sum_{i=from}^{to} v[i] = \begin{cases} 0 & \text{se } from > to \\ v[from] + \sum_{i=from+1}^{to} v[i] & \text{se } from \leq to \end{cases}$$

- La traduzione in C è immediata.

```
int sumVet(int *v, int from, int to)
{
  if (from > to)
    return 0;
  else
    return v[from] + sumvet(v,from+1,to);
}
```

```
int sumVet(int *v, int from, int to)
{
  int somma;
  if (from > to)
    somma = 0;
  else
    somma = v[from] + sumvet(v,from+1,to);
  return somma;
}
```

Espressioni regolari

- Un FA (NFA o DFA) è un metodo per costruire una macchina che riconosce linguaggi regolari.
- Le *espressioni regolari* una notazione algebrica per descrivere, in modo dichiarativo, i linguaggi regolari.
- Esempio: $\mathbf{01}^* + \mathbf{10}^*$
- Le espressioni regolari sono usate, ad esempio,
  - nella descrizione di particolari tipi di sequenze (pattern) nei testi
  - nei comandi UNIX (`grep`)
  - negli strumenti per l'analisi lessicale di UNIX (Lex (Lexical analyzer generator) e Flex (Fast Lex)).

Operazioni sui linguaggi

Le espressioni regolari usano le seguenti operazioni.

- **Unione:**
  $L \cup M = \{w : w \in L \text{ o } w \in M\}$
  Esempio: $\{0, 10\} \cup \{1, 10, 011\} = \{0, 10, 1, 10, 011\}$
- **Concatenazione:**
  $L.M = \{w : w = xy, x \in L, y \in M\}$
  Esempio:
  $\{0, 10\} \cup \{1, 10, 011\} = \{01, 010, 0011, 101, 1010, 10011\}$
- **Potenze:**
  $L^0 = \{\epsilon\},\ L^1 = L,\ L^{k+1} = L.L^k$
- **Chiusura di Kleene:**
  $L^* = \bigcup_{i=0}^{\infty} L^i$
  Esempio: $\{0, 10\}^* = \{\epsilon, 0, 10, 00, 010, 100, 1010, ...\}$

Definizione induttiva di espressioni regolari

Se E è un'espressione regolare, $L(E)$ denota il linguaggio che E definisce.

- **Base:**
  - ϵ e $\emptyset$ sono espressioni regolari.
    $L(\epsilon) = \{\epsilon\}$ e $L(\emptyset) = \emptyset$.
  - Se a è un simbolo in Σ, allora $\boldsymbol{a}$ è un'espressione regolare.
    $L(\boldsymbol{a}) = \{a\}$.
- **Induzione:**
  - Se E e F sono espressioni regolari, allora $E + F$ è un'espressione regolare. $L(E + F) = L(E) \cup L(F)$.
  - Se E e F sono espressioni regolari, allora $E.F$ è un'espressione regolare. $L(E.F) = L(E).L(F)$.
  - Se E è un'espressione regolare, allora (E) è un'espressione regolare. $L((E)) = L(E)$.
  - Se E è un'espressione regolare, allora E^* è un'espressione regolare. $L(E^*) = (L(E))^*$.

Ordine di precedenza per gli operatori

1. Chiusura (*)
2. Concatenazione (.)
3. Più (+)

Esempio: $\mathbf{01}^* + \mathbf{1}$ è raggruppato in $(\mathbf{0}(\mathbf{1})^*) + \mathbf{1}$

Esempi

- $L(\boldsymbol{ab}) = L(\boldsymbol{a}).L(\boldsymbol{b}) = \{ab\}$
- $L(\boldsymbol{ab} + \boldsymbol{b}) = L(\boldsymbol{ab}) \cup L(\boldsymbol{b}) = \{ab, b\}$
- $L(\boldsymbol{a}(\boldsymbol{ab} + \boldsymbol{b})) = L(\boldsymbol{a}).(L(\boldsymbol{ab}) \cup L(\boldsymbol{b})) =$
  $L(\boldsymbol{a}).L(\boldsymbol{ab}) \cup L(\boldsymbol{a}).L(\boldsymbol{b}) = \{aab, ab\}$
- $L(\boldsymbol{ab}^*) = (L(\boldsymbol{a})).(L(\boldsymbol{b}))^* = \{a, ab, abb, abbb, ...\}$
- $L((\boldsymbol{ab})^*) = (L(\boldsymbol{ab}))^* = \{\epsilon, ab, abab, ababab, ...\}$

Esempio

Espressione regolare per
$L = \{w \in \{0,1\}^* : \text{ 0 e 1 alternati in } w\}$

$$(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^*$$

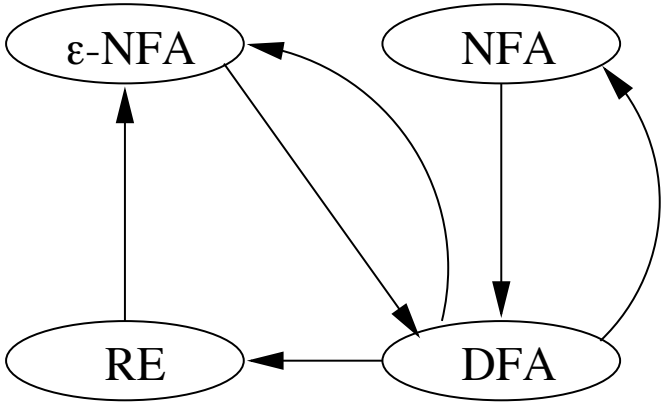
o, equivalentemente,

$$(\epsilon + \mathbf{1})(\mathbf{01})^*(\epsilon + \mathbf{0})$$

Notare che il linguaggio accetta anche le stringhe 0 e 1.

Equivalenza di FA e espressioni regolari

Abbiamo già mostrato che DFA, NFA, e ϵ-NFA sono tutti equivalenti.

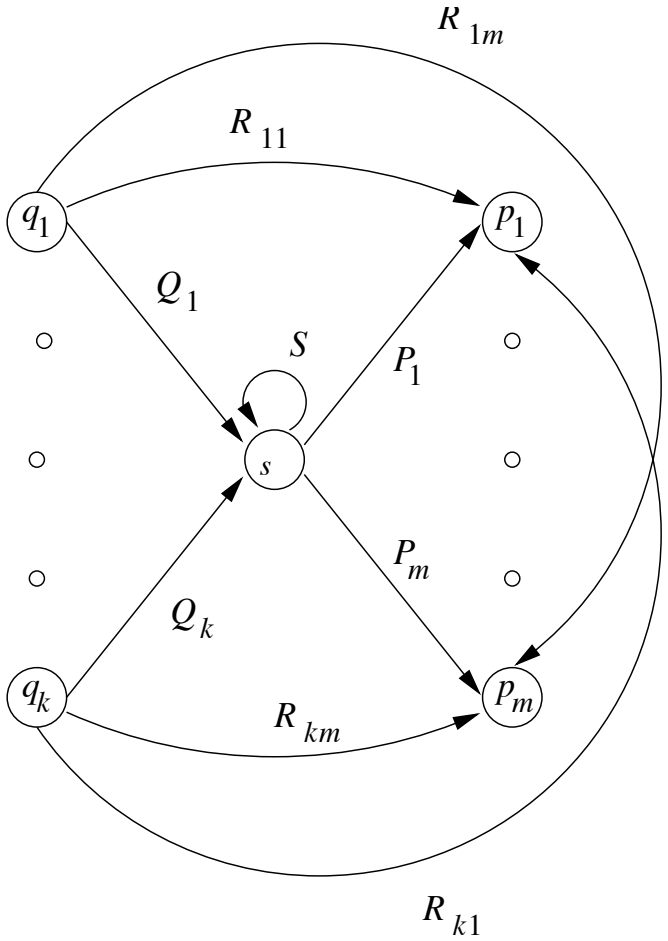


Per mostrare che gli FA sono equivalenti alle espressioni regolari, mostreremo che

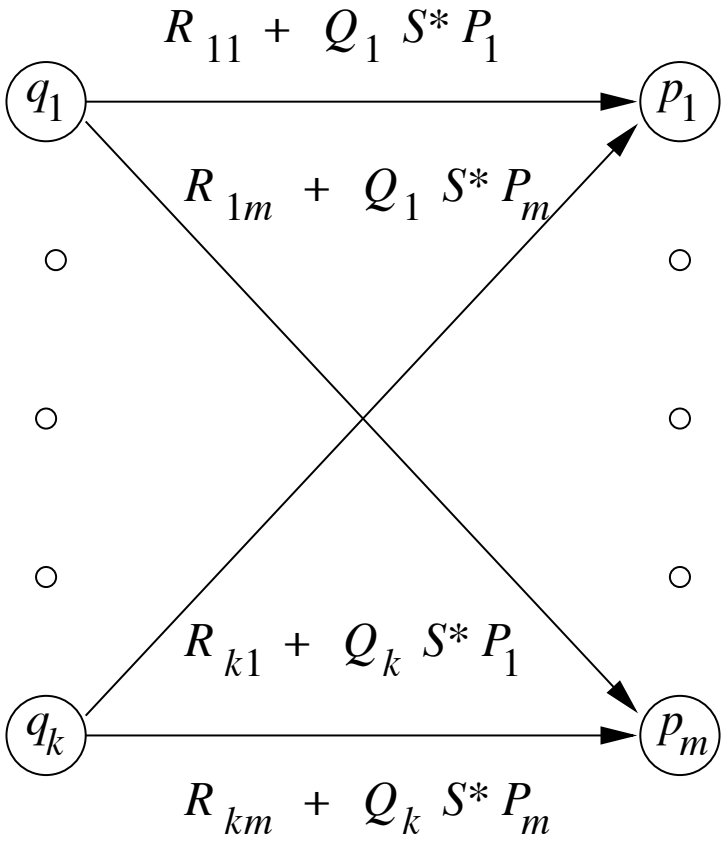
1. Per ogni DFA A possiamo trovare (costruire, in questo caso) un'espressione regolare R, tale che $L(R) = L(A)$.
2. Per ogni espressione regolare R esiste un ϵ-NFA A, tale che $L(A) = L(R)$.

La tecnica di eliminazione di stati

Etichettiamo gli archi con espressioni regolari di simboli

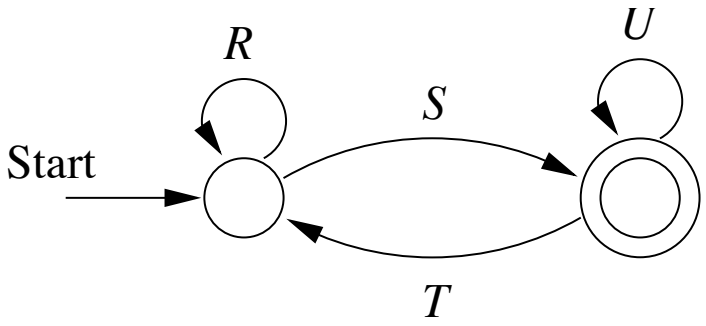


Ora eliminiamo lo stato s.

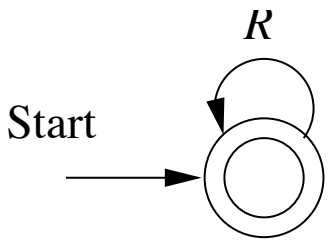


Per lo stato accettante q eliminiamo dall'automa originale tutti gli stati eccetto q_0 e q.

Per ogni $q \in F$ saremo rimasti con A_q della forma



che corrisponde all'espressione regolare $E_q = (R + SU^*T)^*SU^*$, se $q_0 \neq q$, o con A_q della forma



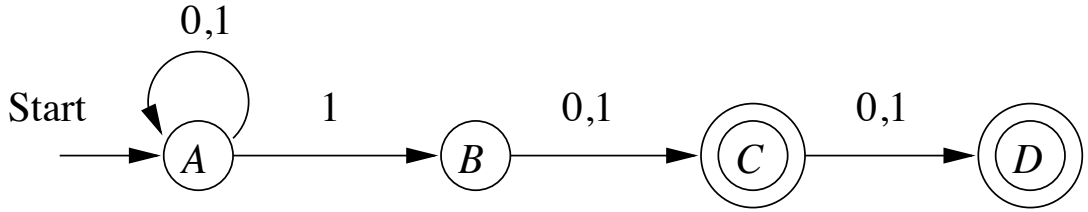
che corrisponde all'espressione regolare $E_q = R^*$, se $q_0 \in F$.

- L'espressione finale è

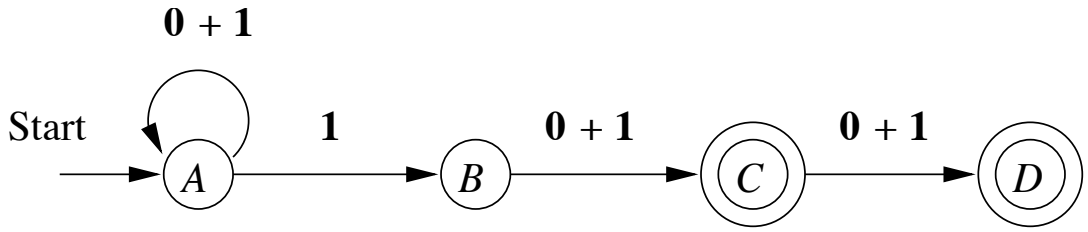
$$\bigoplus_{q \in F} E_q$$

Esempio

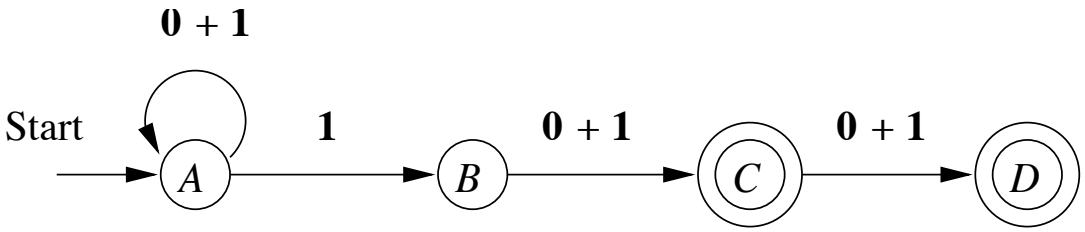
$\mathcal{A}$, dove $L(\mathcal{A}) = \{W : w = x1b,\ \text{o}\ w = x1bc,\ \ x \in \{0,1\}^*, \{b,c\} \subseteq \{0,1\}\}$



La trasformiamo in un automa con espressioni regolari come etichette

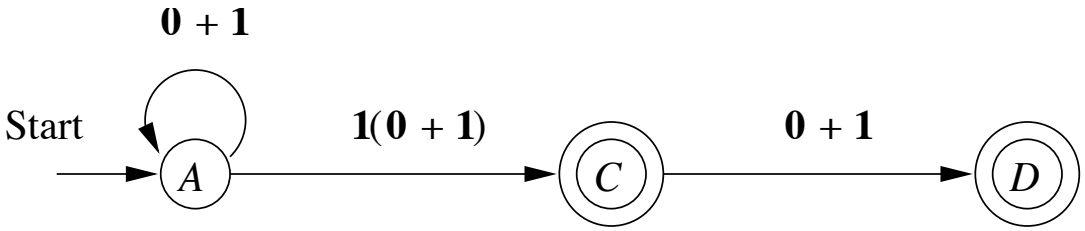


Esempio



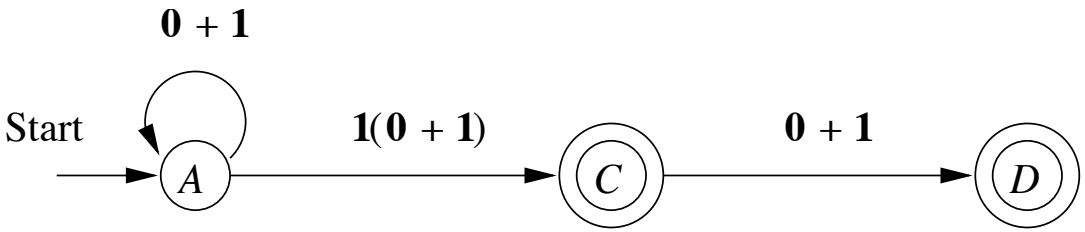
Eliminiamo lo stato B. L'espressione sul nuovo arco che va da A a C è $R_{11} + Q_1 S^* P_1$, dove $R_{11} = \emptyset$, $Q_1 = \mathbf{1}$, S $= \emptyset$, $P_1 = \mathbf{0} + \mathbf{1}$. Considerando che $\emptyset^* = \epsilon$, l'espressione è

$$\emptyset + \mathbf{1}\emptyset^*(\mathbf{0} + \mathbf{1}) = \mathbf{1}\emptyset^*(\mathbf{0} + \mathbf{1}) = \mathbf{1}(\mathbf{0} + \mathbf{1})$$

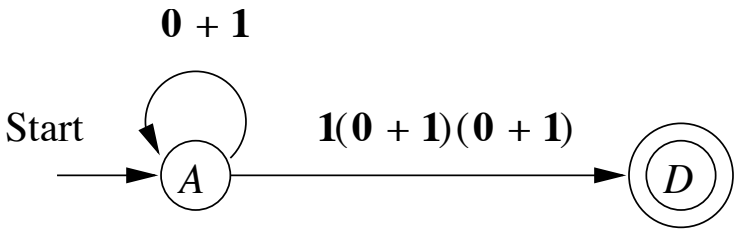


Esempio

Da



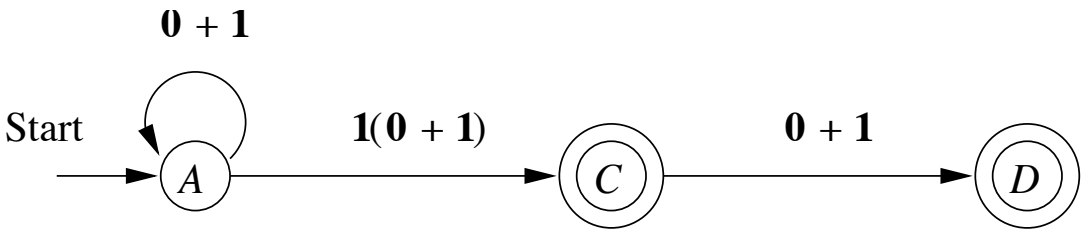
possiamo eliminare lo stato C e ottenere $\mathcal{A}_D$



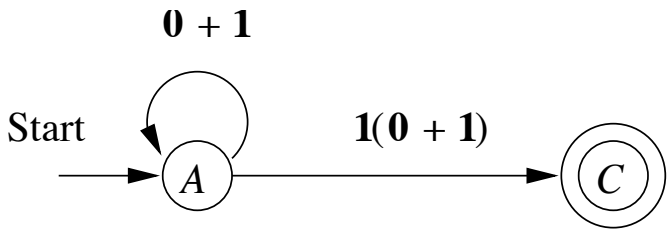
con espressione regolare $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$, dato che $R = \mathbf{0}+\mathbf{1}$, $S = \mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$ e $U = T = \emptyset$ e che quindi $(R + SU^*T)^*SU^* = R^*S$.

Esempio

Da



possiamo eliminare D (e con esso l'arco che va da C a D) e ottenere $\mathcal{A}_C$ con espressione regolare $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$



- L'espressione finale è l'unione delle due precedenti:

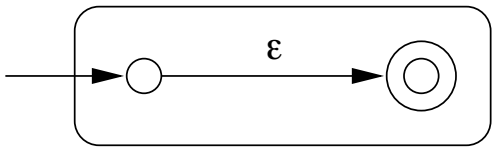
$$(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1}) + (\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$$

Da espressioni regolari a ϵ-NFA

Teorema 3.7: Per ogni espressione regolare R possiamo costruire un ϵ-NFA A, tale che $L(A) = L(R)$.

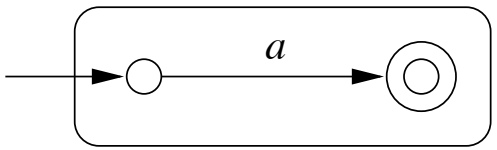
Dimostrazione: Per induzione strutturale:

Base: Automa per ϵ, $\emptyset$, e a.



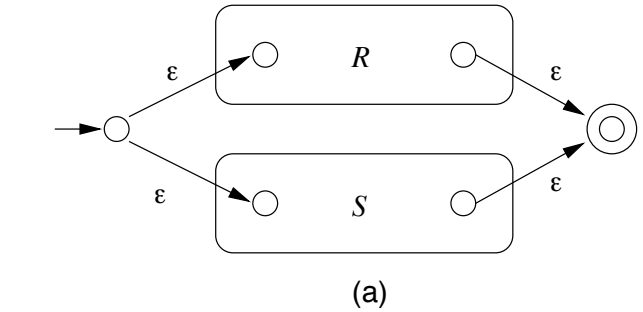
(a)

(b)

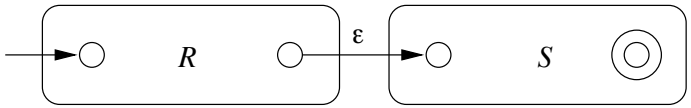


(c)

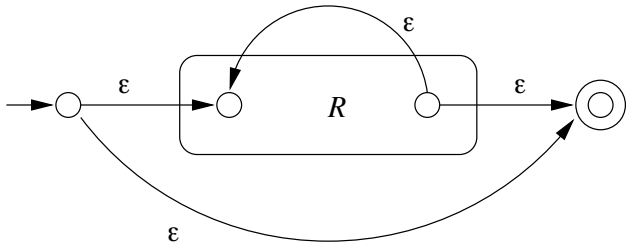
Induzione: Automa per $R + S$, RS, e R^* (quello per (R) coincide con quello per R)



(a)



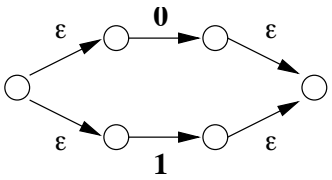
(b)



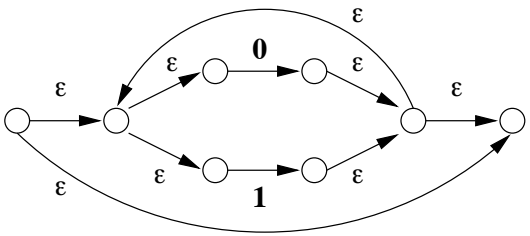
(c)

Esempio

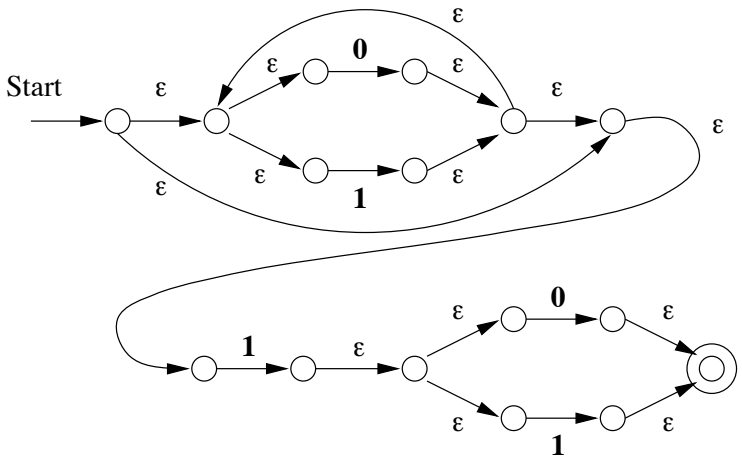
Trasformiamo $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$



(a)



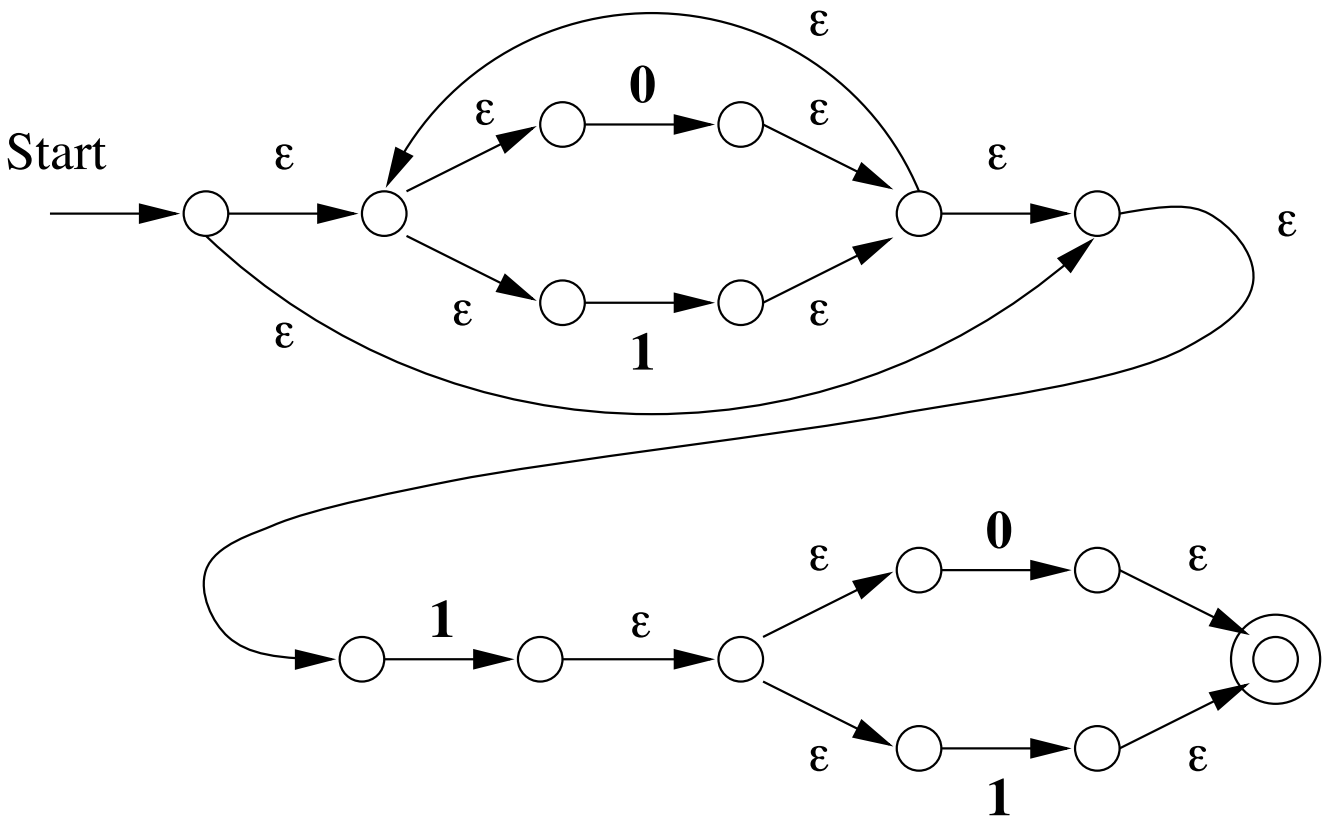
(b)



(c)

Esempio

(b)



(c)

Leggi algebriche per i linguaggi

- $L \cup M = M \cup L$.
  L'unione è *commutativa*.
- $(L \cup M) \cup N = L \cup (M \cup N)$.
  L'unione è *associativa*.
- $(LM)N = L(MN)$.
  La concatenazione è *associativa*.

Nota: La concatenazione non è commutativa, *cioè*, esistono L e M tali che $LM \neq ML$.

- $\emptyset \cup L = L \cup \emptyset = L$.
  $\emptyset$ è *l'identità* per l'unione.
- $\{\epsilon\}L = L\{\epsilon\} = L$.
  $\{\epsilon\}$ è l'identità *sinistra* e *destra* per la concatenazione.
- $\emptyset L = L\emptyset = \emptyset$.
  $\emptyset$ è l'annichilatore *sinistro* e *destro* per la concatenazione.

- $L(M \cup N) = LM \cup LN$.
  La concatenazione è *distributiva a sinistra* sull'unione.
- $(M \cup N)L = ML \cup NL$.
  La concatenazione è *distributiva a destra* sull'unione.
- $L \cup L = L$.
  L'unione è *idempotente*.
- $\emptyset^* = \{\epsilon\},\ \{\epsilon\}^* = \{\epsilon\}$.
- $L^+ = LL^* = L^*L,\ L^* = L^+ \cup \{\epsilon\}$

- $(L^*)^* = L^*$. La chiusura è *idempotente*.

Dimostrazione:

$$w \in (L^*)^* \iff w \in \bigcup_{i=0}^{\infty} \left(\bigcup_{j=0}^{\infty} L^j \right)^i$$

$$\iff \exists k, m \in \mathrm{N} : w \in (L^m)^k$$

$$\iff \exists p \in \mathrm{N} : w \in L^p$$

$$\iff w \in \bigcup_{i=0}^{\infty} L^i$$

$$\iff w \in L^*$$

Espressioni Regolari in UNIX

Le espressioni regolari fanno parte del sistema operativo UNIX fin dall'inizio e sono usate in vari comandi che elaborano testi (ad esempio `grep`, `sed`, `lex`, `vi`, `awk`, `ad`, `ex`, `pg`) per:

- elencare/eliminare righe che contengono un'espressione regolare
- sostituire un'espressione regolare (find/replace)
- ...

La sintassi è leggermente diversa da quella vista finora.

Espressioni Regolari in UNIX

- insiemi di caratteri: pattern elementari che specificano la presenza un carattere appartenente ad un certo insieme.
- ancore: legano il pattern a comparire una posizione specifica della riga (es. inizio, fine).
- gruppi: "racchiudono" l'espressione regolare che può quindi essere riferita come una singola entità.
- modificatori: specificano ripetizioni dell'espressione che precede il modificatore stesso.

Espressioni Regolari in UNIX: sintassi

- `[abc]` match any of the characters enclosed
- `[a-d]` match any character in the enclosed range
- `[set]` match any character not in the following set
- `.` match any single character except `<newline>`
- `[:alpha:]` some predefined character sets
- `[:digit:]`
- `\` treat the next char literally. Normally used to escape special characters such as "." and "*"

Proprietà dei Linguaggi regolari

- **Pumping Lemma.**
  Ogni linguaggio regolare soddisfa il pumping lemma. Se qualcuno vi presenta un falso linguaggio regolare, l'uso del pumping lemma mostrerà una contraddizione.
- **Proprietà di chiusura.**
  Come costruire automi da componenti usando delle operazioni, ad esempio dati L e M possiamo costruire un automa per $L \cap M$.
- **Proprietà di decisione.**
  Analisi computazionale di automi, cioè quanto costa controllare varie proprietà, come l'equivalenza di due automi.
- **Tecniche di minimizzazione.**
  Possiamo risparmiare costruendo automi più piccoli.

Il Pumping Lemma, informalmente

- Supponiamo che $L_{01} = \{0^n 1^n : n \geq 1\}$ sia regolare.
- Allora deve essere accettato da un qualche DFA A, con, ad esempio, k stati.
- Supponiamo che A legga 0^k. Avrà le seguenti transizioni:

ϵ	p_0
0	p_1
00	p_2
$\ldots$	$\ldots$
0^k	p_k

  $\Rightarrow \exists i < j : p_i = p_j$
- Chiamiamo q questo stato.

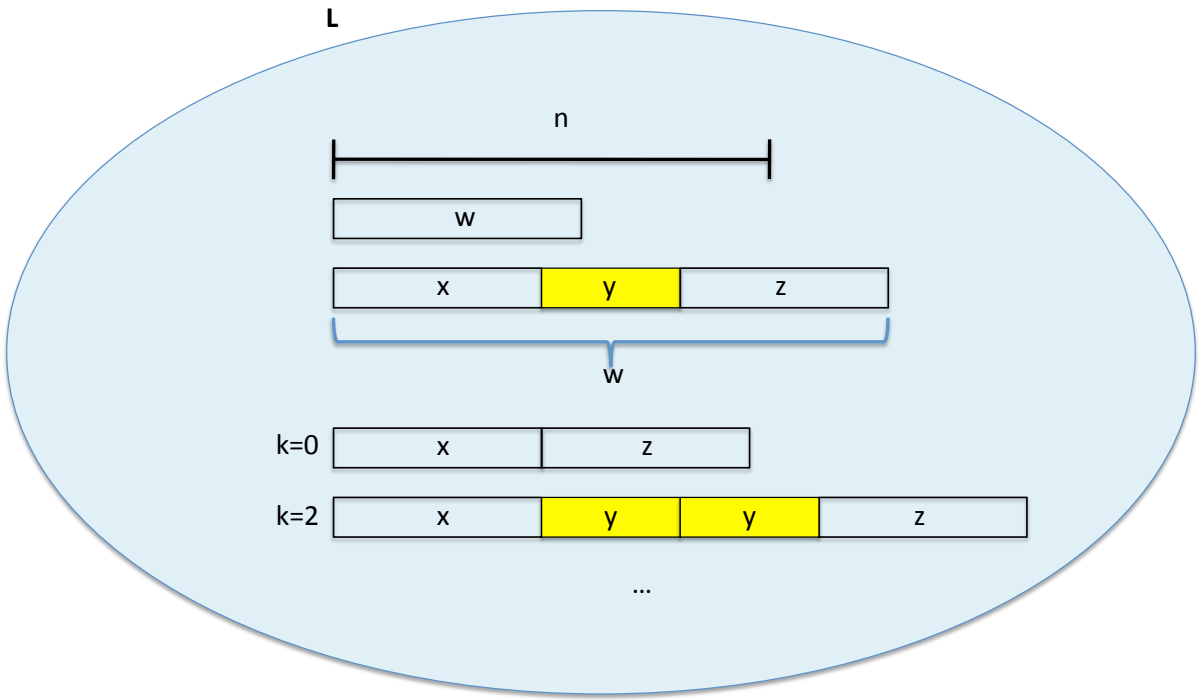
- Adesso possiamo ingannare A:
  - Se $\hat{\delta}(q, 1^i) \in F$ l'automa accetterà, sbagliando, $0^j 1^i$.
  - Se $\hat{\delta}(q, 1^i) \notin F$ l'automa rifiuterà, sbagliando, $0^i 1^i$.
- Quindi L_{01} non può essere regolare.

Teorema 4.1: Il Pumping Lemma per Linguaggi Regolari

Se L è un linguaggio regolare, per il Pumping Lemma, allora
Allora $\exists n, \forall w \in L : |w| \geq n \Rightarrow w = xyz$ tale che:

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. $\forall k \geq 0,\ xy^k z \in L$

Intuitivamente



Intuitivamente (2)

- Esiste una costante n dipendente dal linguaggio L tale che tutte le stringhe di lunghezza $\geq n$ possono essere scomposte in un dato modo
- È sempre possibile scegliere una stringa *non vuota* y da replicare, ovvero **cancellare** o **ripetere** k volte, pur rimanendo all'interno del linguaggio L

Ovvero un cammino più lungo di n deve contenere un ciclo ed è il ciclo a pompare.

Dimostrazione

- Supponiamo che L sia regolare.
- Allora L è riconosciuto da un DFA A con, ad esempio, n stati $Q = \{q_0, \ldots, q_{n-1}\}$.
- Prendiamo come costante il valore n, e consideriamo una generica stringa $w \in L$ più lunga di n. Avremo quindi $w = a_1 a_2 \ldots a_m \in L$ con $m \geq n$.

Dimostrazione (2)

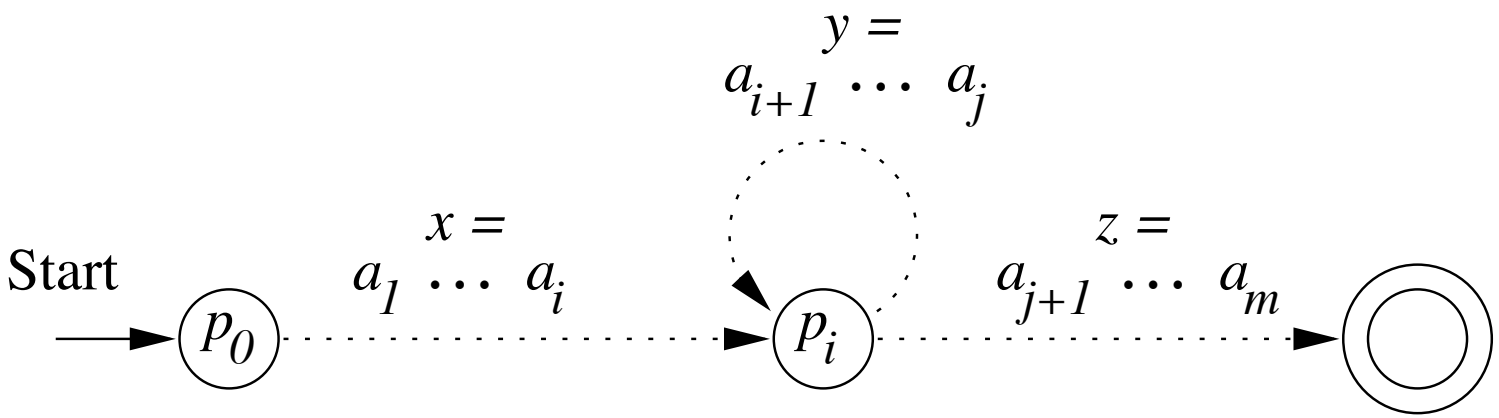
Chiamiamo p_i, per $i \in \{0, ..., m\}$, lo stato in cui si trova l'automa A dopo avere esaminato $a_1 a_2 ... a_i$ a partire dallo stato iniziale q_0.
Formalmente, utilizzando la funzione di transizione estesa:

- $p_0 = \hat{\delta}(q_0, \epsilon) = q_0$
- $p_i = \hat{\delta}(q_0, a_1 a_2 \cdots a_i)$.
- Dato che ci sono n stati distinti, gli $n+1$ stati p_i non possono essere tutti distinti: $\Rightarrow \exists i < j : p_i = p_j$

Dimostrazione (3)

Ora $w = xyz$, dove

1. $x = a_1 a_2 \cdots a_i$ (x porta a p_i la prima volta)
2. $y = a_{i+1} a_{i+2} \cdots a_j$ (y porta da p_i a p_i, dato che p_i e p_j coincidono)
3. $z = a_{j+1} a_{j+2} \ldots a_m$ (z conclude w)



Dimostrazione (4)

Notiamo che

- x può essere vuota (per $i = 0$) e anche z può essere vuota (per $j = n = m$). Invece
- $y \neq \epsilon$: la stringa y non è vuota, dato che $i < j$
- $|xy| \leq n$ dato che gli stati $p_0, ..., p_{j-1}$ sono tutti distinti (basta considerare il minimo indice che si ripete)

Data la forma dell'automa, è chiaro che, eseguendo $k \geq 0$ cicli in p_i, l'automa accetta ogni stringa $xy^k z$.

- per $k = 0$, l'automa passa dallo stato iniziale $q_0 = p_0$ a $p_i = p_j$ su input x. Allora passa da p_i allo stato accettante con input z. Quindi accetta xz.
- per $k > 0$, A va da q_0 a p_i su x, cicla su p_i per k volte su input y^k e passa allo stato accettante per z e accetta $xy^k z$

Quindi per $k \geq 0$, abbiamo che $xy^k z \in L(A)$

PL: una condizione necessaria per la regolarità

Il pumping lemma fornisce una condizione necessaria affinché un linguaggio sia regolare. Ovvero:

- L è regolare $\Rightarrow$ L soddisfa il Pumping Lemma
- L **non** soddisfa il Pumping Lemma $\Rightarrow$ L **non** è regolare

Il Pumping Lemma non dice che **solo** i linguaggi regolari possono godere della proprietà.

Dimostrare che un linguaggio non è regolare con il P.L.

L **non** soddisfa il Pumping Lemma $\Rightarrow$ L **non** è regolare

Non soddisfare il Pumping Lemma significa invertire l'implicazione, utilizzando il fatto che $A \Rightarrow B$ equivale a $\overline{B} \Rightarrow \overline{A}$. Con un po' di manipolazione algebrica possiamo passare quindi dalla formula:

$$\text{L reg.} \Rightarrow \left((\exists n \; \forall w \in L |w| \geq n \Rightarrow \left(\exists x, y, z \; t.c. \begin{cases} w = xyz \\ |xy| \leq n \\ y \neq \epsilon \end{cases} \wedge \; \forall k : xy^k z \in L \right) \right)$$

alla formula

$$\left(\forall n \; \exists w \in L |w| \geq n \wedge \left(\forall x, y, z \; t.c. \begin{cases} w = xyz \\ y \neq \epsilon \\ |xy| \leq n \end{cases} \Rightarrow \; \exists k : xy^k z \notin L \right) \right) \Rightarrow \overline{\text{L reg.}}$$

Esempio

- Sia $L_{01} = \{0^n 1^n\}$ il linguaggio delle stringhe formate da un certo numero di 0, seguiti dallo stesso numero di 1.
- Supponiamo che L_{01} sia regolare. Allora $w = 0^n 1^n \in L$ la stringa per n (infatti $|w| = 2n \geq n$)
- Per il pumping lemma, $w = xyz$, $|xy| \leq n$, $y \neq \epsilon$ e $xy^k z \in L_{01}$

$$w = \underbrace{000\ldots}_{x}\underbrace{\ldots000}_{y}\underbrace{0111\ldots11}_{z} \quad \begin{cases} x = 0^i & \\ y = 0^h & h \geq 1 \wedge i + h \leq n \\ z = 0^j 1^n & i + h + j = n \end{cases}$$

- Valgono (1) e (2), ma
- non vale (3): consideriamo $xy^0 z = xz = 0^{i+j} 1^n$ ha meno 0 che 1: xz non sta nel linguaggio.
- Ne segue che L **non** è regolare.

Esempio (cont.)

Anche non considerando l'ipotesi $|xy| \leq n$, potevamo anche scegliere

- $y = 0^h 1^j$ ($x = 0^{n-h}$, $z = 1^{n-j}$): è chiaro che ripetendo la stringa k volte, gli 0 e gli 1 vengono mescolati; quindi la stringa ottenuta non sta nel linguaggio 1
- $y = 1^h$ è formata solo da 1: basta considerare xz ha meno 0 che 1 e non sta nel linguaggio

Esempio

- Sia L_{eq} il linguaggio delle stringhe con ugual numero di zeri e di uni.
- Supponiamo che L_{eq} sia regolare. Allora $w = 0^n1^n \in L_{eq}$.
- Per il pumping lemma, $w = xyz$, $|xy| \leq n$, $y \neq \epsilon$ e $xy^kz \in L_{eq}$

$$w = \underbrace{000\cdots0}_{x}\underbrace{0}_{y}\underbrace{0111\cdots11}_{z}$$

- In particolare, $xz \in L_{eq}$, ma xz ha meno zeri di uni.
- In alternativa possiamo utilizzare la chiusura dei linguaggi regolari rispetto all'intersezione (vedi dopo) e procedere così:
  - Supponiamo che L_{eq} sia regolare
  - $\mathbf{0}^*\mathbf{1}^*$ sappiamo che è regolare
  - allora $L_{eq} \cap \mathbf{0}^*\mathbf{1}^* = L_{01}$ è regolare, ma questo non lo è (lo abbiamo dimostrato).
  - Quindi anche L_{eq} non è regolare.

Esempio

Supponiamo che $L_{pr} = \{1^p : p \text{ è primo}\}$ sia regolare.
Sia n dato dal pumping lemma.
Scegliamo un numero primo $p \geq n + 2$ e $w = 1^p$.

$$w = \overbrace{111 \cdots \cdots 11111 \cdots 11}^{p}{}_z$$

Esempio

Supponiamo che $L_{pr} = \{1^p : p \text{ è primo}\}$ sia regolare.
Sia n dato dal pumping lemma.
Scegliamo un numero primo $p \geq n + 2$ e $w = 1^p$.

$$w = \overbrace{\underbrace{111\cdots}_{x}\underbrace{\cdots 1}_{y}\underbrace{1111\cdots 11}_{z}}^{p}$$

$$|y|=m \ \wedge \ |xz|=p-m$$

Ora $xy^{p-m}z$ dovrebbe appartenere a L_{pr}

Esempio

Supponiamo che $L_{pr} = \{1^p : p \text{ è primo}\}$ sia regolare.
Sia n dato dal pumping lemma.
Scegliamo un numero primo $p \geq n+2$ e $w = 1^p$.

$$w = \overbrace{\underbrace{111\cdots1}_{x}\underbrace{1}_{y}\underbrace{1111\cdots11}_{z}}^{p}$$
$$|y|=m \ \wedge \ |xz|=p-m$$

Ora $xy^{p-m}z$ dovrebbe appartenere a L_{pr}
$|xy^{p-m}z| = |xz|+(p-m)|y| = p-m+(p-m)m = (1+m)(p-m)$
che non è primo a meno che uno dei fattori non sia 1.

- $y \neq \epsilon \Rightarrow 1+m > 1$
- $m = |y| \leq |xy| \leq n, \quad p \geq n+2$

$\Rightarrow p-m \geq n+2-n = 2.$

Dimostrare che un linguaggio non è regolare come gioco a due

Nel pumping lemma ci sono quattro quantificatori distinti.

- Il giocatore 1 sceglie il linguaggio L
- Il giocatore 2 (l'avversario) sceglie n senza dirlo a 1 (la strategia di 1 deve valere per qualsiasi n $[\forall n]$)
- Il giocatore 1 sceglie w tale che $|w| \geq n$ $[\exists w]$
- Il giocatore 2 sceglie come scomporre w rispettando i vincoli (1) e (2) $[\forall x, y, z]$
- Il giocatore 1 "vince" scegliendo k tale che $xy^k z \notin L$ $[\exists k]$

Se il linguaggio è regolare "vince" l'avversario

- $L = \emptyset$: il giocatore non può scegliere w dall'insieme vuoto
- $L = \{00, 11\}$: se l'avversario sceglie $n > 2$, il giocatore non può scegliere w. Analogo ragionamento vale per tutti gli insiemi finiti.
- $L = (\mathbf{00} + \mathbf{11})^*$: scelto n, qualsiasi w scelto dal giocatore è composto da coppie 00 o 11. L'avversario può scegliere una qualsiasi di queste coppie per y. Ma allora per qualsiasi i, xy^iz continua a rimanere dentro L.
- $L = \mathbf{10}^*\mathbf{1}^*\mathbf{0}$ scelto $n > 2$, qualsiasi w scelto dal giocatore è del tipo 10^i1^j0, con $|w| \geq 1$. Ognuna di queste stringhe w può essere pompata, prendendo $x = 1$, y come secondo simbolo della stringa e z come quel che rimane ($|xy| \leq n$ e $|y| \neq 0$), e rimanere dentro L.
  Per ogni stringa, l'avversario trova come decomporre per "vincere".

PL: non è una condizione sufficiente per la regolarità

Il pumping lemma fornisce **soltanto** una condizione necessaria affinché un linguaggio sia regolare. Ovvero:

- L è regolare $\Rightarrow$ L soddisfa il Pumping Lemma
- L **non** soddisfa il Pumping Lemma $\Rightarrow$ L **non** è regolare
- L soddisfa il Pumping Lemma $\not\Rightarrow$ L è regolare

Esistono linguaggi **non** regolari che soddisfano il Pumping Lemma:

$$\{ww^R v | w, v \in \{0,1\}^+\}$$

Per dimostrare la non regolarità dobbiamo usare altri sistemi.

Proprietà di chiusura dei linguaggi regolari

Siano L e M due linguaggi regolari. Allora i seguenti linguaggi sono regolari:

- *Unione:* $L \cup M$
- *Intersezione:* $L \cap M$
- *Complemento:* $\overline{N}$
- *Differenza:* $L \setminus M$
- *Inversione:* $L^R = \{w^R : w \in L\}$
- *Chiusura:* L^*.
- *Concatenazione:* $L.M$

Chiusura rispetto a unione e complemento

Teorema 4.4. Per ogni coppia di linguaggi regolari L e M, $L \cup M$ è regolare.

Chiusura rispetto a unione e complemento

Teorema 4.4. Per ogni coppia di linguaggi regolari L e M, $L \cup M$ è regolare.

Dimostrazione. Sia $L = L(E)$ e $M = L(F)$. Allora $L(E + F) = L \cup M$ per definizione.

Chiusura rispetto a unione e complemento

Teorema 4.4. Per ogni coppia di linguaggi regolari L e M, $L \cup M$ è regolare.

Dimostrazione. Sia $L = L(E)$ e $M = L(F)$. Allora $L(E + F) = L \cup M$ per definizione.

Teorema 4.5. Se L è un linguaggio regolare su Σ, allora che $\overline{L} = \Sigma^* \setminus L$ è regolare.

Chiusura rispetto a unione e complemento

Teorema 4.4. Per ogni coppia di linguaggi regolari L e M, $L \cup M$ è regolare.

Dimostrazione. Sia $L = L(E)$ e $M = L(F)$. Allora $L(E + F) = L \cup M$ per definizione.

Teorema 4.5. Se L è un linguaggio regolare su Σ, allora che $\overline{L} = \Sigma^* \setminus L$ è regolare.

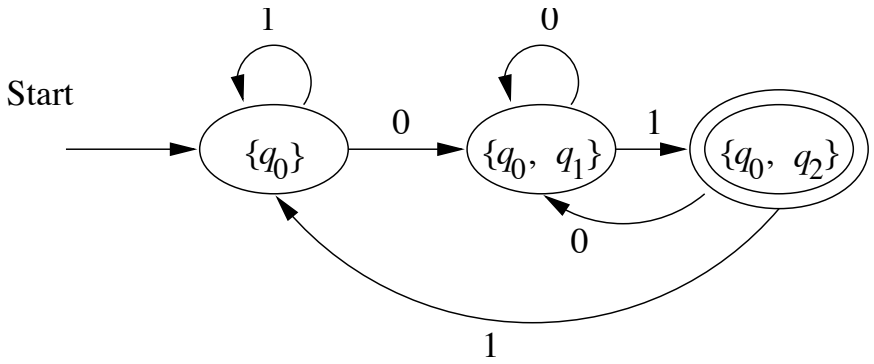
Dimostrazione. Sia L riconosciuto da un DFA

$$A = (Q, \Sigma, \delta, q_0, F).$$

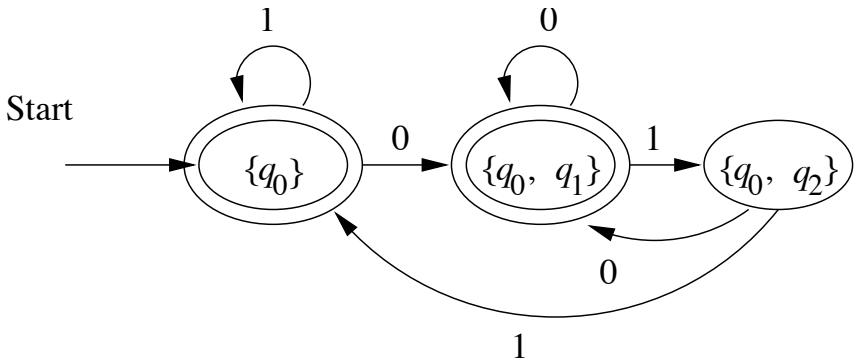
Sia $B = (Q, \Sigma, \delta, q_0, Q \setminus F)$. Allora $L(B) = \overline{L}$.

Esempio

Sia L riconosciuto dal DFA qui sotto:



Allora $\overline{L}$ è riconosciuto da:



Domanda: Quali sono le espressioni regolari per L e $\overline{L}$?

Chiusura rispetto all'intersezione

Teorema 4.8. Se L e M sono regolari, allora anche $L \cap M$ è regolare.

Chiusura rispetto all'intersezione

Teorema 4.8. Se L e M sono regolari, allora anche $L \cap M$ è regolare.

Dimostrazione 1. Per la legge di De Morgan, $L \cap M = \overline{\overline{L} \cup \overline{M}}$. Sappiamo già che i linguaggi regolari sono chiusi sotto il complemento e l'unione.

Chiusura rispetto all'intersezione: un'altra dimostrazione

Se L e M sono regolari, allora anche $L \cap M$ è regolare.
Dimostrazione 2. Sia L il linguaggio di

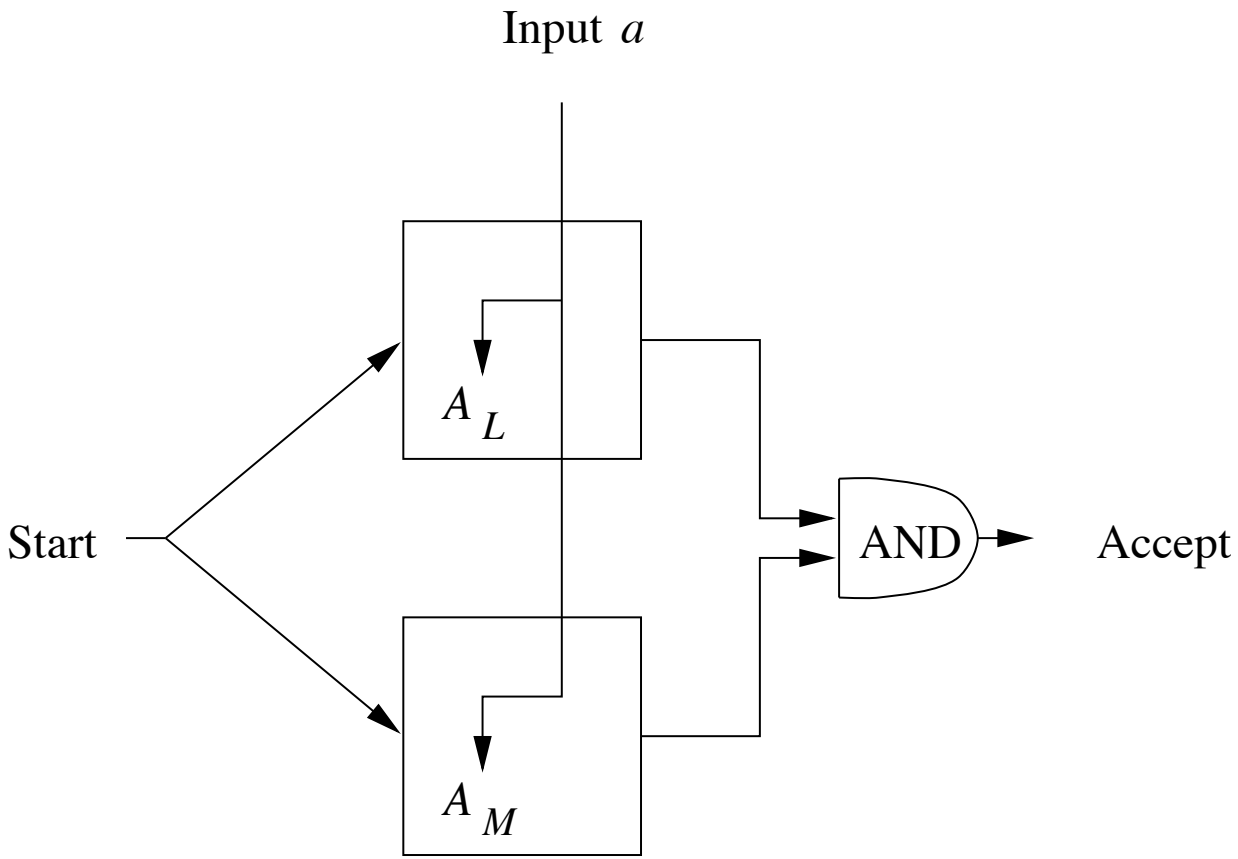
$$A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$$

e M il linguaggio di

$$A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$$

Supponiamo senza perdita di generalità che entrambi gli automi siano deterministici.
Costruiremo un automa che simula A_L e A_M in parallelo, e accetta se e solo se sia A_L che A_M accettano.

Se A_L va dallo stato p allo stato s leggendo a, e A_M va dallo stato q allo stato t leggendo a, allora $A_{L\cap M}$ andrà dallo stato (p, q) allo stato (s, t) leggendo a.



Formalmente

$$A_{L\cap M} = (Q_L \times Q_M, \Sigma, \delta_{L\cap M}, (q_L, q_M), F_L \times F_M),$$

dove

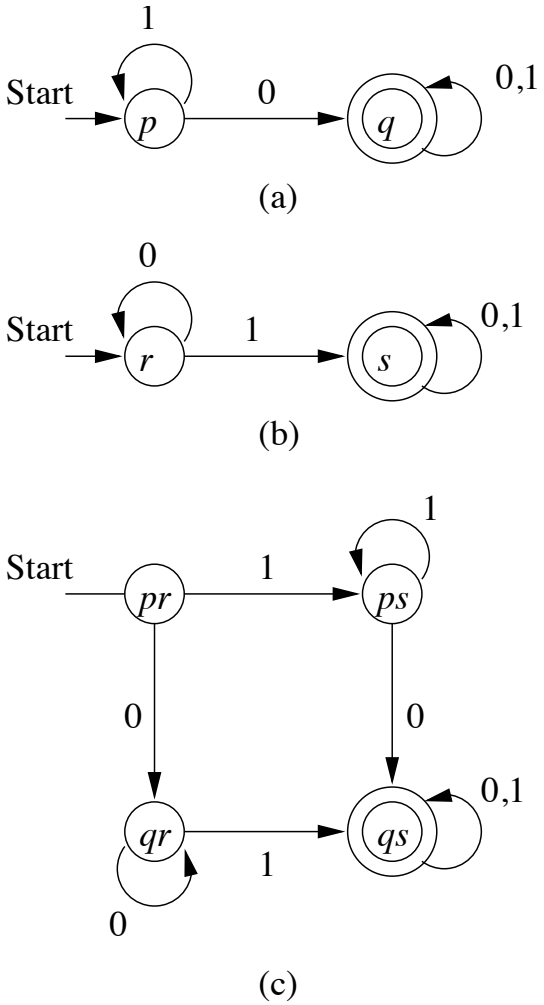
$$\delta_{L\cap M}((p, q), a) = (\delta_L(p, a), \delta_M(q, a))$$

Si può mostrare per induzione su $|w|$ che

$$\hat{\delta}_{L\cap M}((q_L, q_M), w) = \left(\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w)\right)$$

Esempio

$(c) = (a) \times (b)$



Chiusura rispetto alla differenza

Teorema 4.10 Se L e M sono linguaggi regolari, allora anche $L \setminus M$ è regolare.

Chiusura rispetto alla differenza

Teorema 4.10 Se L e M sono linguaggi regolari, allora anche $L \setminus M$ è regolare.

Dimostrazione. Osserviamo che $L \setminus M = L \cap \overline{M}$. Sappiamo già che i linguaggi regolari sono chiusi sotto il complemento e l'intersezione.

Chiusura rispetto al "reverse"

Teorema 4.11 Se L è un linguaggio regolare, allora anche L^R è regolare.

Chiusura rispetto al "reverse"

Teorema 4.11 Se L è un linguaggio regolare, allora anche L^R è regolare.

Dimostrazione 1: Sia L riconosciuto da un FA A. Modifichiamo A per renderlo un FA per L^R:

1. Giriamo tutti gli archi.
2. Rendiamo il vecchio stato iniziale l'unico stato finale.
3. Creiamo un nuovo stato iniziale p_0, con $\delta(p_0, \epsilon) = F$ (i vecchi stati finali).

Chiusura rispetto al "reverse": un'altra dimostrazione

Se L è un linguaggio regolare, allora anche L^R è regolare.
Dimostrazione 2: Sia L descritto da un'espressione regolare E.
Costruiremo un'espressione regolare E^R, tale che
$L(E^R) = (L(E))^R$.

Chiusura rispetto al "reverse": un'altra dimostrazione

Se L è un linguaggio regolare, allora anche L^R è regolare.
Dimostrazione 2: Sia L descritto da un'espressione regolare E.
Costruiremo un'espressione regolare E^R, tale che
$L(E^R) = (L(E))^R$.
Procediamo per induzione strutturale su E.
Base: Se E è ϵ, $\emptyset$, o $\boldsymbol{a}$, allora $E^R = E$.
Induzione:

1. $E = F + G$. Allora $E^R = F^R + G^R$
2. $E = F.G$. Allora $E^R = G^R.F^R$
3. $E = F^*$. Allora $E^R = (F^R)^*$

Proprietà di decisione

1. Convertire tra diverse rappresentazioni dei linguaggi regolari.
2. È $L = \emptyset$?
3. È $w \in L$?
4. Due descrizioni definiscono lo stesso linguaggio?

Da NFA a DFA

- Supponiamo che un ϵ-NFA abbia n stati.
- Per calcolare ECLOSE(p) seguiamo al più n^2 archi. Lo facciamo per n stati, quindi in totale sono n^3 passi.
- Il DFA ha 2^n stati, per ogni stato S e ogni $a \in \Sigma$ calcoliamo $\delta_D(S, a)$ in n^3 passi, consultando l'informazione sulle ϵ-chiusure e la tabella delle transizioni. In totale abbiamo $O(n^3 2^n)$ passi.
- Se calcoliamo δ solo per gli stati raggiungibili, dobbiamo calcolare $\delta_D(S, a)$ solo s volte, dove s è il numero di stati raggiungibili. In totale: $O(n^3 s)$ passi.

Da DFA a NFA

Dobbiamo solo mettere le parentesi graffe attorno agli stati.
Totale: $O(n)$ passi.

Da FA a espressione regolare

Dobbiamo calcolare n^3 cose di grandezza fino a 4^n. Totale: $O(n^3 4^n)$.
L'FA può essere un NFA. Se prima vogliamo convertire l'NFA in un DFA, il tempo totale sarà doppiamente esponenziale.

Da espressioni regolari a FA

Possiamo costruire un albero per l'espressione in n passi.
Possiamo costruire l'automa in n passi.
Eliminare le ϵ-transizioni ha bisogno di $O(n^3)$ passi.
Se si vuole un DFA, potremmo aver bisogno di un numero esponenziale di passi.

Controllare se un linguaggio è vuoto

- $L(A) \neq \emptyset$ per FA A se e solo se uno stato finale è raggiungibile dallo stato iniziale in in A. Totale: $O(n^2)$ passi.
- Oppure, possiamo guardare un'espressione regolare E e vedere se $L(E) = \emptyset$, considerando tutti i casi:
  - $E = F + G$. Allora $L(E)$ è vuoto se e solo se sia $L(F)$ che $L(G)$ sono vuoti.
  - $E = F.G$. Allora $L(E)$ è vuoto se e solo se o $L(F)$ o $L(G)$ sono vuoti.
  - $E = F^*$. Allora $L(E)$ non è mai vuoto, perché $\epsilon \in L(E)$.
  - $E = \epsilon$. Allora $L(E)$ non è vuoto.
  - $E = \boldsymbol{a}$. Allora $L(E)$ non è vuoto.
  - $E = \emptyset$. Allora $L(E)$ è vuoto.

Controllare l'appartenenza ad un linguaggio

- Per controllare se $w \in L(A)$ per DFA A, simuliamo A su w. Se $|w| = n$, questo prende $O(n)$ passi.
- Se A è un NFA e ha s stati, simulare A su w prende $O(ns^2)$ passi.
- Se A è un ϵ-NFA e ha s stati, simulare A su w prende $O(ns^3)$ passi.
- Se $L = L(E)$, per l'espressione regolare E di lunghezza s, prima convertiamo E in un ϵ-NFA con $2s$ stati. Poi simuliamo w su questo automa, in $O(ns^3)$ passi.

Pumping Lemma per la non regolarità: qualche esercizio

Dimostrare che i seguenti linguaggi non sono regolari.

- L'insieme delle stringhe di parentesi bilanciate.
- $L = \{0^n 1^m | n \leq m\}$
- $L = \{0^n 1^m 2^n | n, m \text{ interi}\}$
- $L = \{0^{n^2} | n \text{ intero}\}$
- $L = \{ww | w \in \{0,1\}^*\}$
- $L = \{ww^R | w \in \{0,1\}^*\}$
- $L = \{0^i 1^j | mcd(i,j) = 1\}$

Tipi user-defined

- Il `C` mette a disposizione un insieme di tipi di dato predefiniti (tipi built-in) e dei meccanismi per definire nuovi tipi (tipi user-defined)
- Vediamo le regole generali che governano la definizione di nuovi tipi e quindi i costrutti linguistici (costruttori) che il `C` mette a disposizione.
- Tutti i tipi non predefiniti utilizzati in un programma devono essere dichiarati come ogni altro elemento del programma. Una dichiarazione di tipo viene fatta di solito nella parte dichiarativa del programma.
  - parte dichiarativa globale:
    - dichiarazioni di costanti
    - dichiarazioni di tipi
    - dichiarazioni di variabili
    - prototipi di funzioni/procedure

Dichiarazione di tipo

- Una dichiarazione di tipo (type declaration) consiste nella parola chiave typedef seguita da:
  - la rappresentazione o costruzione del nuovo tipo (ovvero la specifica di come è costruito a partire dai tipi già esistenti)
  - il nome del nuovo tipo
  - il simbolo ; che chiude la dichiarazione

  Esempio: `typedef int anno;`
- Una volta definito e nominato un nuovo tipo, è possibile utilizzarlo per dichiarare nuovi oggetti (ad es. variabili) di quel tipo.

  Esempio:

```
float x;
anno a;
```

- **Nota:** In C si possono anche definire tipi senza usare `typedef`. Quest'ultima consente l'associazione di un nome (identificatore) a un nuovo tipo. Per uniformità e leggibilità del codice useremo spesso `typedef` per definire nuovi tipi.

Tipi semplici user-defined

Ridefinizione: Un nuovo tipo può essere definito rinominando un tipo già esistente (cioè creandone un alias)

typedef TipoEsistente NuovoTipo;

dove `TipoEsistente` può essere un tipo built-in o user-defined.

Esempio:

```
typedef int anno;
typedef int naturale;

typedef char carattere;
```

Enumerazione: Consente di definire un nuovo tipo enumerando i suoi valori, con la seguente sintassi

typedef enum {v1, v2, ... , vk} NuovoTipo;

Esempio:

```
typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
typedef enum {gen, feb, mar, apr, mag, giu,
              lug, ago, set, ott, nov, dic} Mese;

typedef enum {m, f} sesso;
```

- I valori elencati nella definizione di un nuovo tipo enumerato, sono identificatori che rappresentano costanti di quel tipo (esattamente come `0, 1, 2, ...` sono costanti del tipo `int`, o `'a', 'b', ...` sono costanti del tipo `char`).
- Dunque, se dichiariamo una variabile
  `Giorno g;`
  possiamo scrivere l'assegnamento
  `g = mar;`
- Le costanti dei tipi enumerati non vanno racchiuse tra virgolette o tra apici!

  N.B. Il compilatore associa ai nomi utilizzati per denotare le costanti dei tipi enumerati valori naturali progressivi.

  Esempio: il valore associato a `g` dopo l'assegnamento `g=mar` è il numero naturale (intero) `1`.

  $\Longrightarrow$ mancanza di astrazione: è possibile fare riferimento alla rappresentazione dei valori.

- La relazione tra interi e tipi enumerati consente di applicare a questi ultimi le seguenti operazioni:
  - operazioni aritmetiche: `+,-,*,/,%`
  - uguaglianza e disuguaglianza: `=, !=`
  - confronto: `<,<=,>,>=`
- Si noti che la relazione di precedenza tra i valori (che determina l'esito delle operazioni di confronto) dipende dall'ordine in cui vengono elencati i valori del tipo al momento della sua definzione.

  Esempio: Con le dichiarazioni viste in precedenza `lun < gio` è vero (un intero diverso da `0`) `apr <= feb` è falso (il valore intero `0`)
- Il C tratta questi tipi come ridefinizione di `int`

Tipi fai da te: i booleani

Soluzione 1

```
typedef int Boolean;
Boolean b; ...
```

Soluzione 2

```
#define FALSE 0;
#define TRUE 1;...
typedef int Boolean;
Boolean b;
...
```

Soluzione 3

```
typedef enum {FALSE, TRUE} Boolean;...
Boolean b;
...
```

N.B. I valori vanno elencati come sopra, rispettando la convenzione adottata dal `C`: il valore `0` rappresenta falso.

Esempio:

```
typedef enum {false, true} boolean;

boolean even (int n)
{
if (n % 2 == 0)
    return true;
else
    return false;
}


boolean implies (boolean p, boolean q)
{
    if (p)
        return q;
    else
        return true;
}
```

Esempio: Uso del costrutto `switch` con tipi enumerati

```
typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
Giorno g;
...
switch (g) {
case lun: case mar: case mer: case gio: case ven:
        printf("Giorno lavorativo");
        break;
case sab: case dom:
        printf("Week-end");
        break;
}
```

```
void stampaGiorno(Giorno g) {
switch (g) {
case lun: printf("lun");
          break;
...

case dom: printf("dom");
          break;
}
```

Tipi strutturati user-defined

- Il C non possiede tipi strutturati built-in, ma fornisce dei costruttori che permettono di definire tipi strutturati anche piuttosto complessi.
- Array e puntatori possono essere visti come costruttori di tipo (definiscono un tipo di dato non semplice a partire da tipi esistenti).

Uso di `typedef` con array e puntatori

- In generale, una dichiarazione di tipo mediante `typedef` ha la forma di una dichiarazione di variabile preceduta dalla parola chiave `typedef`, e con il nome di tipo al posto del nome della variabile.
- Nel caso di array e puntatori:

```
typedef TipoElemento TipoArray[Dimensione];
typedef TipoPuntato *TipoPuntatore;
```

Esempio:

```
typedef int ArrayDieciInteri[10];
typedef int MatriceTreXQuattro[3][4];
typedef int *PuntIntero;
ArrayDieciInteri vet;           /*  int vet[10]; */
PunIntero p;                    /*  int *p; */
MatriceTreXQuattro mat, mat1;   /*  int mat[3][4]; int mat1[3][4]; */
```

Il costruttore struct

- Una struttura è un'aggregazione di elementi che possono essere eterogenei (di tipo diverso).
  Esempio:

```
struct persona  {
      char nome[15];
      char cognome[20];
      int eta;
      sesso s;   }
```

  - la parola chiave `struct` introduce la definizione della struttura
  - `persona` è l'etichetta della struttura, attribuisce un nome alla definizione della struttura
  - `nome`, `cognome`, `eta`, `s` sono detti campi della struttura
- È anche possibile definire strutture con campi omogenei

```
struct complex  {
 double real;
 double imag;  }
```

Campi di una struttura

- devono avere nomi univoci all'interno di una struttura
- strutture diverse possono avere campi con lo stesso nome
- i nomi dei campi possono coincidere con altri nomi già utilizzati (es. per variabili o funzioni)

  Esempio:

```
int x;
struct a { char x; int y; };
struct b { int w; float x; };
```

- possono essere di tipo diverso (semplice o altre strutture)
- un campo di una struttura non può essere del tipo struttura che si sta definendo
- un campo può però essere di tipo puntatore alla struttura

  Esempio:

```
struct s { int a;
           struct s *p;  };
```

Dichiarazione di variabili di tipo struttura

- La definizione di una struttura non provoca allocazione di memoria, ma introduce un nuovo tipo di dato.

  Esempio: `struct persona tizio, docenti[10], *p;`

  - `tizio` è una variabile di tipo `struct persona`
  - `docenti` è un vettore di 10 elementi di tipo `struct persona`
  - `p` è un puntatore a una `struct persona`
  - N.B.: `persona tizio;` Errore!

- Una variabile di tipo struttura può essere dichiarata contestualmente alla definizione della struttura.

  Esempio:

```
struct studente {          |      struct  {
  char nome[20];           |        char nome[20];
  long matricola;          |        long matricola;
  struct data ddn;         |        struct data ddn;
} s1, s2;                  |      } s1, s2;
```

- In questo caso si può anche omettere l'etichetta di struttura.

Uso di typedef con strutture

- Attraverso `typedef` è possibile associare un nome ad un tipo definito mediante il costruttore `struct`.

  Esempio:

```
struct data { int giorno, mese, anno; };

typedef struct data Data;
```

- `Data` è un sinonimo di `struct data`, che può essere utilizzato nelle dichiarazioni di variabili.

```
Data d1, d2;
Data appelli[10], *pd;
```

Operazioni sulle strutture

- Si possono assegnare variabili di tipo struttura a variabili dello stesso tipo struttura.

  Esempio:

```
Data d1, d2;
...
d1 = d2;
```

- Non è possibile invece effettuare il confronto tra due variabili di tipo struttura.

  Esempio:

```
struct data d1, d2;
if (d1 == d2) ...          Errore!
```

- L'equivalenza di tipo tra strutture è per nome.

  Esempio:

```
struct s1 { int i; };
struct s2 { int i; };
struct s1 a, b;
struct s2 c;
```

`a = b;`	OK	`a` e `b` sono dello stesso tipo
`a = c;`	Errore!	`a` e `c` non sono dello stesso tipo

- Si può ottenere l'indirizzo di una variabile di tipo struttura tramite l'operatore `&`.
- Si può rilevare la dimensione di una struttura con `sizeof`.

  Esempio: `sizeof(struct data)`
- Attenzione: non è detto che la dimensione di una struttura sia pari alla somma delle dimensioni dei singoli campi.

Accesso ai campi di una struttura

- I campi di una struttura si comportano come variabili del tipo corrispondente. L'accesso avviene tramite l'operatore punto

```
Data oggi;
oggi.giorno = 26; oggi.mese = 4; oggi.anno = 2012;
printf("%d %d %d", oggi.giorno, oggi.mese, oggi.anno);
```

- Accesso tramite un puntatore alla struttura.

```
Data oggi, *pd;
pd = &oggi;
(*pd).giorno = 26; (*pd).mese = 4; (*pd).anno = 2012;
```

N.B. Ci vogliono le **()** perché "**.**" ha priorità più alta di "*****".

- Operatore freccia: combina il dereferenziamento e l'accesso al campo della struttura.

```
pd->giorno =26; pd->mese = 4; pd->anno = 2012;
```

- N.B.: `pd->giorno` è una abbreviazione per `(*pd).giorno`.

Esempio: Accesso al campo di una struttura che è a sua volta campo di un'altra struttura.

```
struct dipendente
   { Persona datiDip;
     Data dataAssunzione;
     int stipendio;
   };
typedef struct dipendente Dipendente;

Dipendente dip, *p;
...
dip.dataAssunzione.giorno = 3;
dip.dataAssunzione.mese = 4;
dip.dataAssunzione.anno = 1997;
...
(p->dataAssunzione).giorno = 5;
(p->stipendio) = (p->stipendio) + 120;
```

Inizializzazione di strutture

- Può avvenire, come per i vettori, con un elenco di inizializzatori.

  Esempio: `Data oggi = { 26, 4, 2012 }`

- Se ci sono meno inizializzatori di campi della struttura, i campi rimanenti vengono inizializzati a 0 (o al valore speciale `NULL`, se il campo è un puntatore).

Passaggio di parametri di tipo struttura

- È come per i parametri di tipo semplice:
  - il passaggio è per valore $\Longrightarrow$ viene fatta una copia dell'intera struttura dal parametro attuale a quello formale
  - è comunque possibile simulare il passaggio per indirizzo attraverso un puntatore

  Nota: per passare per valore ad una funzione un vettore (il vettore, non il puntatore al suo primo elemento) è sufficiente racchiuderlo in una struttura.

Esempio:

```
struct dipendente
   { Persona datiDip;
     Data dataAssunzione;
     int stipendio;
   };
typedef struct dipendente Dipendente;

void aumento(Dipendente *p, int percentuale)
{
    int incremento;
    incremento = (p -> stipendio) * percentuale / 100;
    p -> stipendio = p -> stipendio + incremento;
}
```

Allocazione Dinamica della memoria

- Il C mette a disposizione delle primitive per la gestione dinamica della memoria, grazie all'utilizzo di alcune funzioni messe a disposizione dalla libreria standard (standard library). Infatti è richiesta l'inclusione del file header `<stdlib.h>`
- Le primitive principali sono
  - malloc,calloc, e realloc: consentono di allocare dinamicamente memoria per una variabile di un tipo specificato o per array di dimensione non nota a priori;
  - free: consente di rilasciare dinamicamente memoria (precedentemente allocata con malloc)
- I tipi di dato sono ancora statici, ovvero hanno una dimensione fissata a priori. Le variabili di un certo tipo di dato possono invece essere create.

malloc

- La chiamata di funzione

  malloc(sizeof(TipoDato));

  crea in memoria una variabile di tipo TipoDato, e restituisce come risultato l'indirizzo della variabile creata (`NULL` se fallisce).
- Se p è una variabile di tipo puntatore a TipoDato, l'istruzione

  p=malloc(sizeof(TipoDato));

  assegna l'indirizzo restituito dalla funzione malloc a p che punta quindi alla nuova variabile (p già esiste).
- Una variabile creata dinamicamente è necessariamente anonima: a essa si può fare riferimento solo tramite un puntatore

  a differenza di una variabile dichiarata mediante un proprio identificatore, che può essere riferita sia direttamente sia tramite un puntatore

Allocazione dinamica di array

- Posso creare dinamicamente un array di interi la cui dimensione `N` viene immessa da tastiera? Lo si può fare nel modo seguente
- ```
  ptr = malloc(N*sizeof(int));
  ```
- Un array allocato dinamicamente può poi essere trattato come un array creato staticamente.
  ```

calloc

- Che assunzioni possiamo fare sul contenuto della memoria appena allocata con `malloc`?
- Nessuna, come nel caso della dichiarazione di una variabile non inizializzata. Però, esiste la funzione

```
calloc(size_t num, size_t size);
```

- Alloca dinamicamente `num` elementi di dimensione `size` (in bytes) e li inizializza a 0, ad esempio con

```
int * ptr = (int*) calloc(N,sizeof(int));
```

  si allocano `N` interi e li si inizializza a `0`.
- Restituisce `NULL` in caso di fallimento dell'allocazione

realloc

- Che succede se ci si accorge che l'array allocato è troppo piccolo e lo si vuole ingrandire dinamicamente senza perdere le informazioni memorizzate?

```
realloc(void *ptr, size_t size);
```

- `ptr` è un puntatore ad un'area di memoria precedentemente allocata
- `size` è la nuova dimensione (in bytes) dell'area di memoria
- Restituisce il puntatore al primo elemento dell'array ridimensionato o `NULL`

free

- In C, la gestione dello heap e la deallocazione dello spazio è lasciata al programmatore.
- Se p è l'indirizzo di una variabile allocata dinamicamente, la chiamata

  free(p);

  rilascia lo spazio di memoria puntato da p

  la corrispondente memoria fisica è resa disponibile per qualsiasi altro uso.
- free deve ricevere come parametro attuale (per valore) un puntatore al quale era stato assegnato come valore l'indirizzo restituito da una funzione di allocazione dinamica di memoria (come malloc), altrimenti il comportamentoè indefinito.
- Attenzione: La chiamata `free(p)` non può modificare il valore del puntatore, che quindi non diventa `NULL`, ma continua a puntare all'indirizzo della zona precedentemente allocata. Accedere tramite `p` a questa zona, adesso libera e riallocabile, è scorretto.

Heap

- Poiché le variabili dinamiche possono essere create e distrutte in un qualsiasi punto del programma esse non possono essere allocate sullo stack.
- Vengono allocate in un'altra zona di memoria chiamata heap (mucchio). La loro gestione risulta molto più inefficiente.

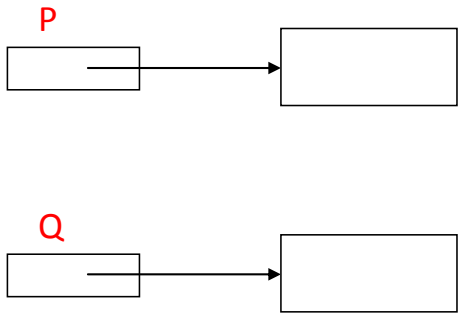
Produzione di garbage (spazzatura)

- Si verifica quando la memoria allocata dinamicamente risulta logicamente inaccessibile, e quindi sprecata, perché non esiste alcun riferimento ad essa.

  Esempio:

```
P=malloc(sizeof(TipoDato));
...
P=Q;
```

- In questo modo la cella puntata da `P` subito dopo l'assegnamento `P=Q` perde ogni possibilità di accesso (da cui il termine spazzatura).



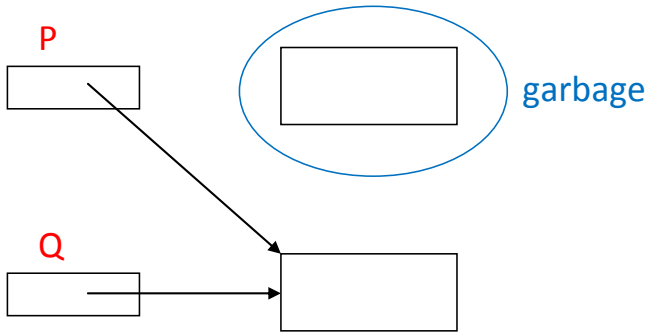
Produzione di garbage (spazzatura)

- Si verifica quando la memoria allocata dinamicamente risulta logicamente inaccessibile, e quindi sprecata, perché non esiste alcun riferimento ad essa.

  Esempio:

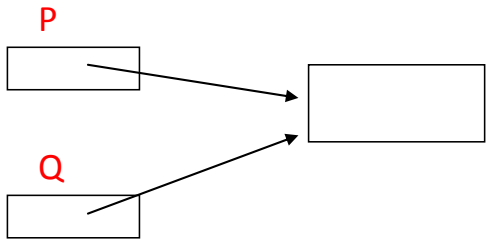
```
P=malloc(sizeof(TipoDato));
...
P=Q;
```

- In questo modo la cella puntata da `P` subito dopo l'assegnamento `P=Q` perde ogni possibilità di accesso (da cui il termine spazzatura).



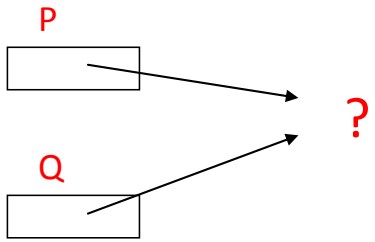
Riferimenti fluttuanti (dangling references)

- Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.
  Esempio:
  ```
  P=Q;
  free(Q);
  ```
- L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).
- `*P` comporterebbe l'accesso all'indirizzo fisico puntato da `P` e l'interpretazione del suo contenuto come un valore del tipo di `*P` con risultati imprevedibili.



Riferimenti fluttuanti (dangling references)

- Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.
  Esempio:
  ```
  P=Q;
  free(Q);
  ```
- L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).
- `*P` comporterebbe l'accesso all'indirizzo fisico puntato da `P` e l'interpretazione del suo contenuto come un valore del tipo di `*P` con risultati imprevedibili.



Riferimenti fluttuanti (dangling references) (cont.)

- È possibile che il blocco di memoria ridimensionato da una `realloc` venga allocato in una differente posizione dello heap. Il contenuto della precedente area di memoria viene quindi spostato dalla `realloc`. Però

  Esempio:

```
 int *ptr1 = (int*) malloc(5 * sizeof(int));
int *ptr2 ;
....
ptr2 = (int*) realloc(ptr1,10*sizeof(int));
```

  Il puntatore `ptr1` che puntava all'area di memoria prima della chiamata alla `realloc` dopo potrebbe puntare ad un'area non più valida.

- Produzione di garbage e riferimenti fluttuanti hanno svantaggi simmetrici:
  - la prima comporta spreco di memoria
  - la seconda comporta risultati imprevedibili e scorretti.
- La seconda è più pericolosa della prima e in alcuni linguaggi non è prevista l'istruzione `free`.
- Viene lasciato al supporto del linguaggio l'onere di effettuare garbage collection ("raccolta rifiuti").

Equivalenza e minimizzazione

Problema: due descrizioni di linguaggi regolari denotano lo stesso linguaggio?

- vedremo un algoritmo che permette di stabilire se due FA definiscono lo stesso linguaggio, sono equivalenti
- una conseguenza importante, è l'esistenza di un modo per minimizzare un automa, ovvero per trovare l'automa minimo (con il minor numero di stati) equivalente
- l'automa minimo è unico a meno di ridenominazione degli stati

Stati equivalenti

Sia $A = (Q, \Sigma, \delta, q_0, F)$ un DFA, e $\{p, q\} \subseteq Q$. Definiamo

$$p \equiv q \Leftrightarrow \forall w \in \Sigma^* : \hat{\delta}(p, w) \in F \text{ se e solo se } \hat{\delta}(q, w) \in F$$

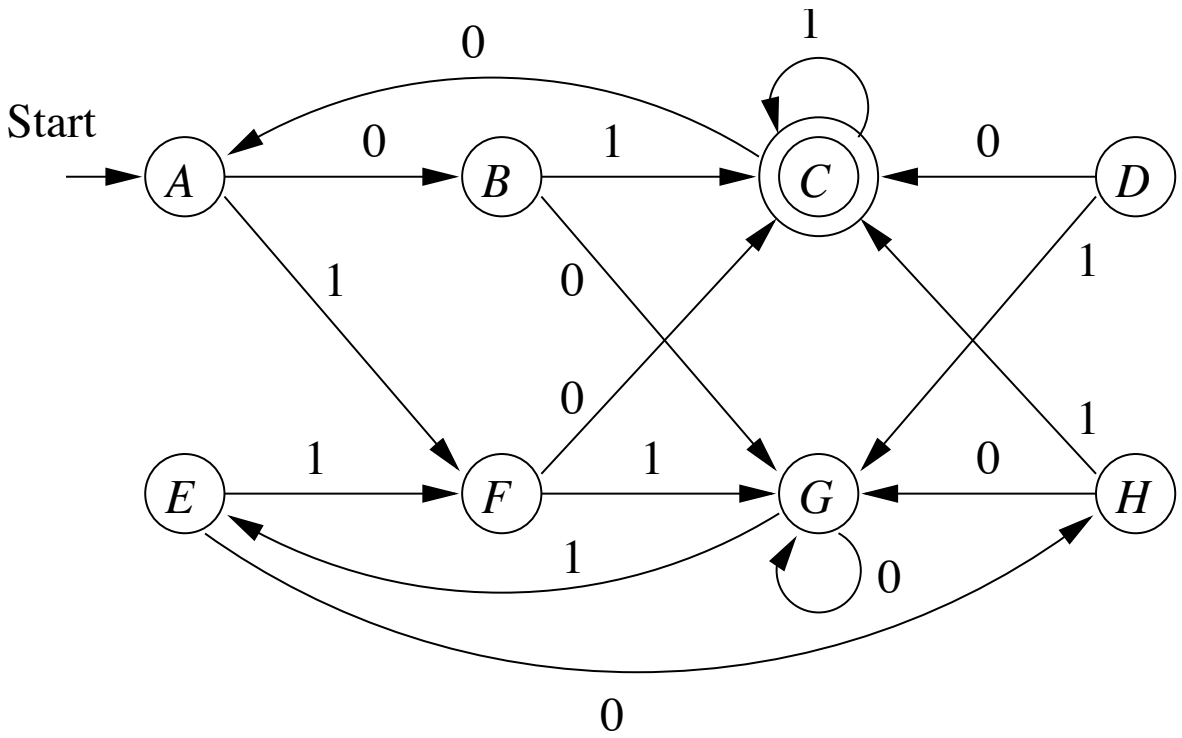
- Se $p \equiv q$ diciamo che p e q sono *equivalenti*
- Se $p \not\equiv q$ diciamo che p e q sono *distinguibili*

In altre parole: p e q sono distinguibili se e solo se

$$\exists w : \hat{\delta}(p, w) \in F \text{ e } \hat{\delta}(q, w) \notin F, \text{ o viceversa}$$

Ovvero se e solo se esiste almeno una stringa w che li differenzia, ovvero l'automa si muove da p in uno stato accettante e da q in uno stato non accettante (o viceversa).

Esempio



$\hat{\delta}(C, \epsilon) \in F, \hat{\delta}(G, \epsilon) \notin F \Rightarrow C \not\equiv G$
$\hat{\delta}(A, 01) = C \in F, \hat{\delta}(G, 01) = E \notin F \Rightarrow A \not\equiv G$

Stati distinguibili

Alcuni stati distinguibili:

- $C \neq G$ infatti C è accettante e G no. Formalmente, la stringa che li distingue è la stringa vuota,

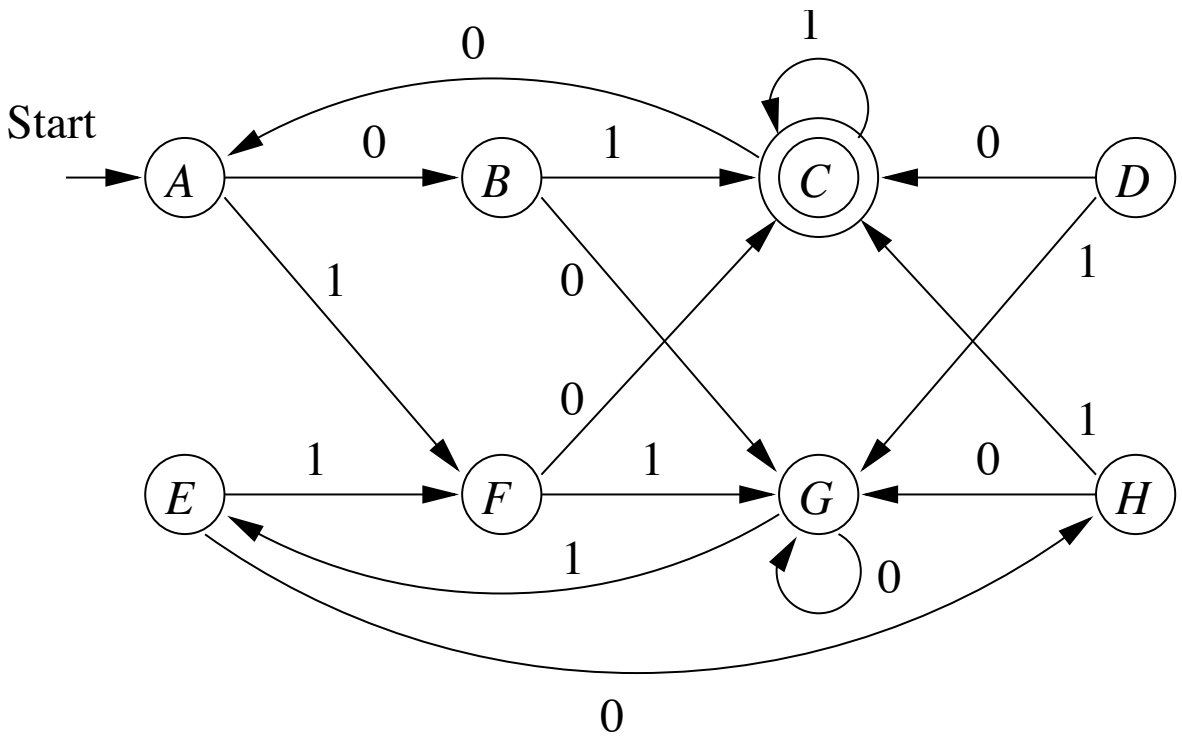
$$\hat{\delta}(C, \epsilon) \in F, \hat{\delta}(G, \epsilon) \notin F$$

- Consideriamo A e G. Non sono distinti da ϵ (entrambi non sono accettanti). Non sono distinti da 0 perche porta A in B e G in se stesso, ma entrambi non sono accettanti. Analogamente non sono distinti da 1 perche porta A in F e G in E, ma entrambi non sono accettanti. Abbiamo tuttavia:

$$\hat{\delta}(A, 01) = C \in F, \hat{\delta}(G, 01) = E \notin F \Rightarrow A \not\equiv G$$

Dato che C è accettante e E non lo è, la stringa 01 distingue A da G, ovvero $A \not\equiv G$.

Cosa si può dire su A e E?



$\hat{\delta}(A, \epsilon) = A \notin F, \hat{\delta}(E, \epsilon) = E \notin F$
$\hat{\delta}(A, 1) = F = \hat{\delta}(E, 1)$
Quindi $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x) = \hat{\delta}(F, x)$
$\hat{\delta}(A, 00) = G = \hat{\delta}(E, 00)$
$\hat{\delta}(A, 01) = C = \hat{\delta}(E, 01)$
Conclusione: $A \equiv E$.

Algoritmo induttivo

Possiamo calcolare *tutte e sole* le coppie di stati distinguibili con il seguente metodo induttivo (**algoritmo di riempimento-tabella**):
Formalmente, calcoliamo la minima relazione $R_A \subseteq Q \times Q$ tale che
Base: Se $p \in F$ e $q \notin F$, allora $(p, q) \in R_A$, ovvero $p \not\equiv q$.
Induzione: Se $\exists a \in \Sigma : (\delta(p, a), \delta(q, a)) \in R_A$, ovvero $\delta(p, a) \not\equiv \delta(q, a)$, allora $(p, q) \in R_A$, ovvero $p \not\equiv q$.

Algoritmo induttivo

Applichiamo l'algoritmo ad A:

	A	B	C	D	E	F	G
B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x

Le coppie *marcate* nella tabella stanno in R_A (sono distinguibili).
Le coppie (E, A), (B, H), $(F, D) \notin R_A$ sono le uniche che risultano non distinguibili, equivalenti.

Algoritmo: dettagli

1. Marcare tutte le coppie $(q_i, q_j).q_i \in F \wedge q_j \notin F$ o viceversa: dato che C è l'unico stato accettante, l'algoritmo parte con $(C, q) \in R_A$ per qualsiasi altro stato q
2. Marcare ogni coppia non ancora marcata (q_i, q_j) se $(\delta(q_i, a), \delta(q_j, a))$ $(a \in \Sigma)$ è già marcata: marchiamo gli stati (p, q) che si muovono con a in una coppia di stati del punto 1. Ad es. la coppia (E, F) viene aggiunta perché si muove in (H, C) con input 0. Analogamente, vengono aggiunte tutte le altre coppie, ad eccezione di (A, G) ed (E, G).
3. Ripetere il passo precedente fino a che non si possono marcare ulteriori stati: al passo successivo si aggiungono le coppie (A, G) ed (E, G). Infatti in entrambi i casi i due stati si muovono, leggendo 1, negli stati (E, F), che sono stati aggiunti al passo 2.
4. Le coppie rimaste $(q_i, q_j) \notin R_A$ sono di stati **indistinguibili**: non ci sono altre coppie da aggiungere, rimangono fuori solo le coppie (E, A), (B, H), (E, D)

Variante al modo di procedere (Algoritmo di Hopcroft)

1. Si inizia proponendo una prima partizione: gli stati in F e quelli $Q \setminus F$, in modo da non avere nella partizione finale nessuna classe con stati finale e non finali.
2. Poi iterativamente si affina questa partizione, fino ad arrivare alle classe finali: per ogni stato q di una classe si osserva dove si arriva con 0: se non si rimane nella stessa classe di q, allora q forma una nuova classe insieme agli stati che si comportano come q, ovvero che su 0 vanno nello stesso stato raggiunto da q su 0. Si ripete il procedimento su 1. A questo punto si procede con la intersezione delle partizioni ottenute e si ripete il procedimento a partire dalle nuove classi.

Altro modo di procedere (2)

1. Dividiamo gli stati in $\{C\}$ e $\{A, B, D, E, F, G, H\}$.
2. Osserviamo che con 0: da A si va in B (stessa classe), da B si va in G (stessa classe), mentre da D si va in C (l'altra classe), da E si va in H (stessa classe), da F si va in C (altra classe, come G), da G si va in G (stessa classe), infine da H si va in G (stessa classe). Nuova partizione della classe degli stati non finali: $\{A, B, E, G, H\}$ e $\{D, F\}$.
3. Osserviamo che con 1: da A si va in F (stessa classe), da B si va in C (altra classe), mentre da D si va in G (stessa classe), da E si va in F (stessa classe), da F si va in G (stessa classe), da G si va in E (stessa classe), infine da H si va in C (altra classe). Nuova partizione della classe degli stati non finali: $\{A, D, E, F, G\}$ e $\{B, H\}$.
4. Intersecando otteniamo: $\{A, E, G\}$, $\{B, H\}$, $\{C\}$, $\{D, F\}$.
5. Ripetiamo. e otteniamo: $\{A, E\}$ e $\{G\}$.

Correttezza dell'algoritmo

Teorema 4.20: Se p e q non sono distinguibili dall'algoritmo, allora $p \equiv q$. Ovvero

$$(p, q) \notin R_A \text{ sse } p \equiv q$$

Dimostrazione: Supponiamo per assurdo che esista una *coppia "sbagliata"* $\{p, q\}$, tale che

1. $\exists w : \hat{\delta}(p, w) \in F, \hat{\delta}(q, w) \notin F$, o viceversa.
2. L'algoritmo non distingue tra p e q, ovvero $(p, q) \notin R_A$.

Sia $w = a_1 a_2 \cdots a_n$ la stringa più corta che identifica la coppia "sbagliata" $\{p, q\}$, ovvero $\hat{\delta}(p, w) \in F$ and $\hat{\delta}(q, w) \notin F$.
Allora $w \neq \epsilon$ perché altrimenti l'algoritmo distinguerebbe p da q (caso base). Quindi $n \geq 1$ (deve esserci almeno un simbolo a_1).

- Consideriamo gli stati $r = \delta(p, a_1)$ e $s = \delta(q, a_1)$
- Notiamo che

  $$\hat{\delta}(p, w) = \hat{\delta}(r, a_2 \cdots a_n) \in F \text{ and } \hat{\delta}(q, w) = \hat{\delta}(s, a_2 \cdots a_n) \notin F$$

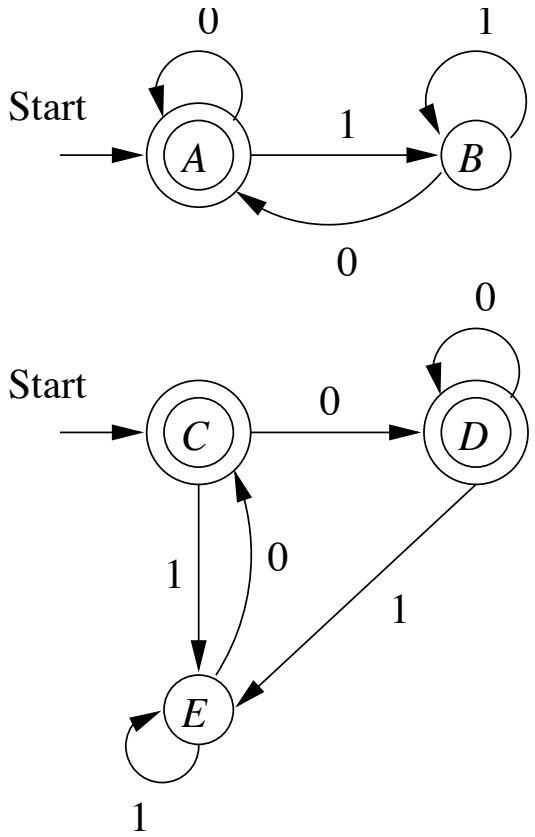
  Allora $\{r, s\}$ non può essere una coppia sbagliata perché $\{r, s\}$ sarebbe identificata da una stringa più corta di w.
- Quindi, l'algoritmo deve aver scoperto nel caso base che r and s sono distinguibili, ovvero $(r, s) \in R_A$.
- Ma allora l'algoritmo distinguerebbe p da q nella parte induttiva. Contraddizione.
- Quindi non ci sono coppie "sbagliate" e il teorema è vero.

Testare l'equivalenza di linguaggi regolari

Siano L e M linguaggi regolari (descritti in qualche forma).
Per testare se $L = M$

1. convertiamo sia L che M in DFA.
2. Immaginiamo il DFA che sia l'unione dei due DFA (non importa se ha due stati iniziali)
3. Se l'algoritmo dice che i due stati iniziali sono distinguibili, allora $L \neq M$, altrimenti $L = M$.

Esempio



Possiamo vedere che entrambi i DFA accettano $L(\epsilon + (\mathbf{0} + \mathbf{1})^*\mathbf{0})$.

Esempio

Il risultato dell'algoritmo è

B	x			
C		x		
D		x		
E	x		x	x
	A	B	C	D

La coppia di stati iniziali (A, C) non è marcata. Quindi i due automi sono equivalenti.

Minimizzazione di DFA

- Possiamo usare l'algoritmo per minimizzare un DFA mettendo insieme tutti gli stati equivalenti. Cioè rimpiazzando p by $p/_{\equiv}$.
- Il DFA unione di prima ha le seguenti classi di equivalenza: $\{\{A, C, D\}, \{B, E\}\}$.
- Notare: affinché $p/_{\equiv}$ sia una *classe di equivalenza*, la relazione $\equiv$ deve essere una *relazione di equivalenza* (riflessiva, simmetrica, e transitiva).

Classi di equivalenza

	A	B	C	D	E	F	G
B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x

Il primo DFA esaminato ha le seguenti classi di equivalenza:

$$\{\{A, E\}, \{B, H\}, \{C\}, \{D, F\}, \{G\}\}$$

Transitività

Teorema 4.23: Se $p \equiv q$ e $q \equiv r$, allora $p \equiv r$.

Dimostrazione: Supponiamo per assurdo che $p \not\equiv r$.

- Allora $\exists w$ tale che $\hat{\delta}(p, w) \in F$ e $\hat{\delta}(r, w) \notin F$, o viceversa.
- Lo stato $\hat{\delta}(q, w)$ o è di accettazione o no.
- *Caso 1*: $\hat{\delta}(q, w)$ è di accettazione. Allora $q \not\equiv r$.
- *Caso 2*: $\hat{\delta}(q, w)$ non è di accettazione. Allora $p \not\equiv q$.
- Il caso contrario può essere dimostrato simmetricamente.
- Quindi deve essere $p \equiv r$.

Minimizzazione di automi

Per minimizzare un DFA $A = (Q, \Sigma, \delta, q_0, F)$ costruiamo un DFA $B = (Q/_{\equiv}, \Sigma, \gamma, q_0/_{\equiv}, F/_{\equiv})$, dove

$$\gamma(p/_{\equiv}, a) = \delta(p, a)/_{\equiv},$$

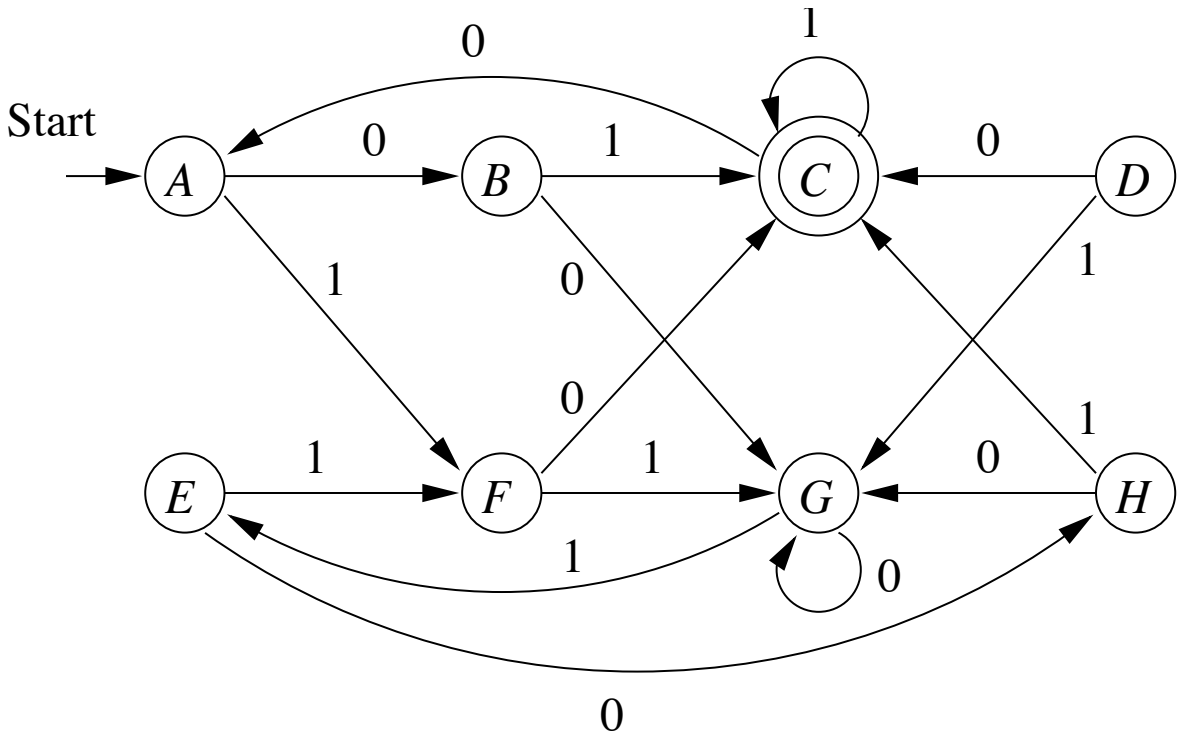
ovvero le mosse della classe di equivalenza dello stato p, per ogni $a \in \Sigma$ sono la classe di equivalenza trovata tramite δ per ogni stato $p \in p/_{\equiv}$.
Nota Affinché B sia ben definito, dobbiamo mostrare che

$$\text{Se } p \equiv q \text{ allora } \delta(p, a) \equiv \delta(q, a)$$

Se $\delta(p, a) \not\equiv \delta(q, a)$, allora l'algoritmo concluderebbe $p \not\equiv q$, quindi B è ben definito.
Analogamente, si può mostrare che $F/_{\equiv}$ contiene tutti e soli gli stati accettanti di A.

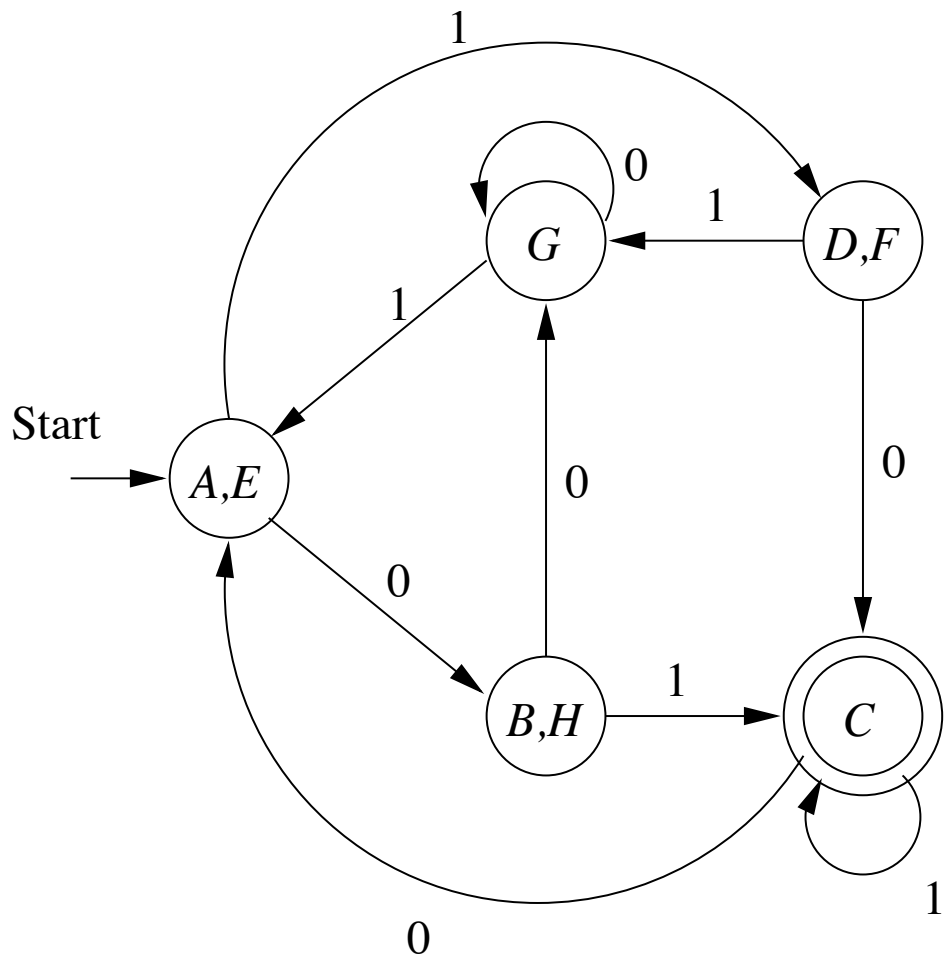
Esempio

Possiamo minimizzare



Esempio

Otteniamo:



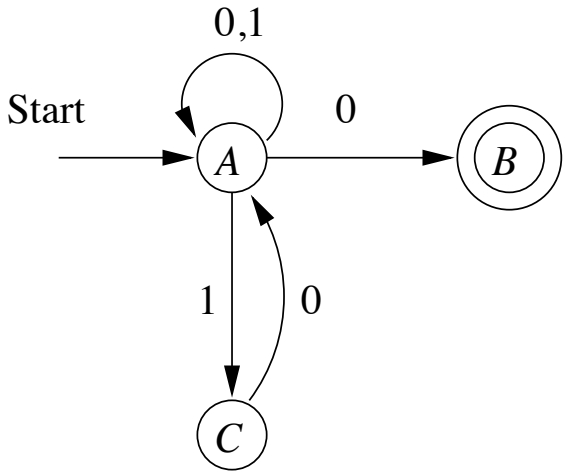
Commenti

- Gli stati sono le classi di equivalenza

$$\{\{A, E\}, \{B, H\}, \{C\}, \{D, F\}, \{G\}\}$$

- lo stato iniziale è la classe di equivalenza dello stato iniziale A, ovvero $\{A, E\}$
- gli stati accettanti sono le classi di equivalenza dell'unico stato accettante C, ovvero $\{C\}$
- per ogni classe di equivalenza e per ogni simbolo, tutta la classe si muove in modo uniforme. Esempio: $\{A, E\}$ leggendo 0 si muove in $\{B, H\}$.

Nota: Non possiamo applicare l'algoritmo agli NFA. Proviamo a minimizzare, ad esempio, l'NFA che riconosce il linguaggio $(0+1)^*(10)^*(0+1)^*0$



- Intuitivamente possiamo rimuovere lo stato C e le sue transizioni.
- L'algoritmo tuttavia non riporterebbe nessuna coppia di stati equivalenti
- B è finale e distinguibile da A e C
- Si può togliere lo stato C e il ciclo, ma $A \not\equiv C$, quindi l'automa non è minimo.

Perché non si può migliorare il DFA minimizzato

- Sia B il DFA minimizzato ottenuto applicando l'algoritmo al DFA A.
- Sappiamo già che $L(A) = L(B)$.
- Potrebbe esistere un DFA C, con $L(C) = L(B)$ e meno stati di B?
- Applichiamo l'algoritmo a B "unito con" C.
- Dato che $L(B) = L(C)$, abbiamo $q_0^B \equiv q_0^C$.
- Inoltre, $\delta(q_0^B, a) \equiv \delta(q_0^C, a)$, per ogni a.

- Per ogni stato p in B esiste almeno uno stato q in C, tale che $p \equiv q$.
- **Dimostrazione**:
  - Non ci sono stati inaccessibili, quindi $p = \hat{\delta}(q_0^B, a_1 a_2 \cdots a_k)$, per una qualche stringa $a_1 a_2 \cdots a_k$.
  - Allora $q = \hat{\delta}(q_0^C, a_1 a_2 \cdots a_k)$, e $p \equiv q$.
  - Dato che C ha meno stati di B, ci devono essere due stati r e s di B tali che $r \equiv t \equiv s$, per qualche stato t di C.
  - Ma allora $r \equiv s$ che è una contraddizione, dato che B è stato costruito dall'algoritmo.

Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP

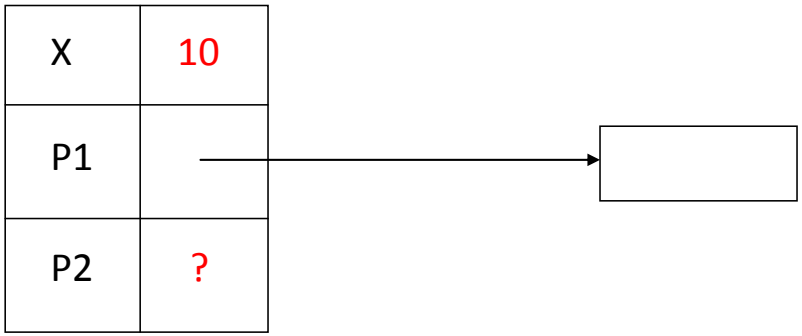
X	10
P1	?
P2	?

Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP

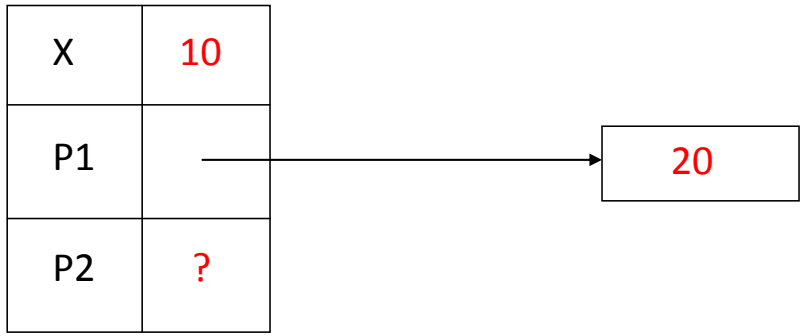


Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP

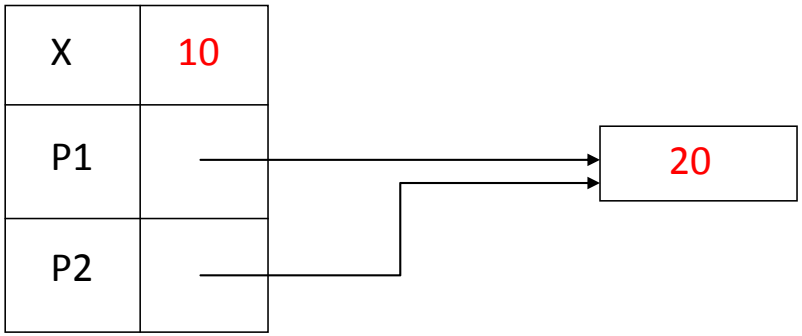


Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP

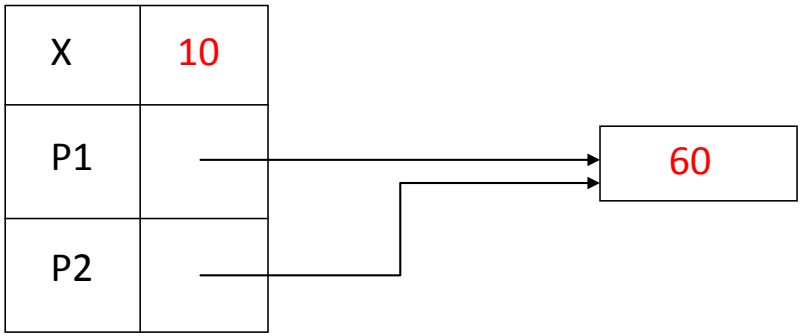


Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP

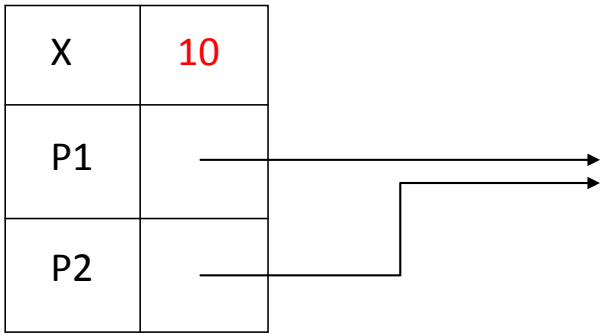


Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA HEAP



Liste

- È molto comune dover rappresentare sequenze di elementi tutti dello stesso tipo e fare operazioni su di esse.

  Esempi: sequenza di interi (23 46 5 28 3)
  sequenza di caratteri (`'x' 'r' 'f'`)
  sequenza di persone con nome e data di nascita

- Finora abbiamo usato gli array per realizzare tali strutture, nonostante ciò porti talvolta a un impiego inefficiente della memoria.
- Vediamo adesso un modo basato sull'allocazione dinamica di variabili, che ci permette di realizzare liste di elementi in maniera che la memoria fisica utilizzata corrisponda meglio a quella astratta, cioè al numero di elementi della sequenza che vogliamo rappresentare.

Diversi modi di rappresentare sequenze di elementi

1. Rappresentazione sequenziale: tramite array
   - Vantaggi:
     - l'accesso agli elementi è diretto (tramite indice) ed efficiente
     - l'ordine degli elementi è quello in memoria $\Longrightarrow$ non servono strutture dati addizionali
     - è semplice manipolare l'intera struttura (copia, ordinamento, ...)
   - Svantaggi:
     - dobbiamo avere un'idea precisa della dimensione della sequenza
     - inserire o eliminare elementi è complicato ed inefficiente (comporta un numero di spostamenti che nel caso peggiore può essere dell'ordine del numero degli elementi della struttura)

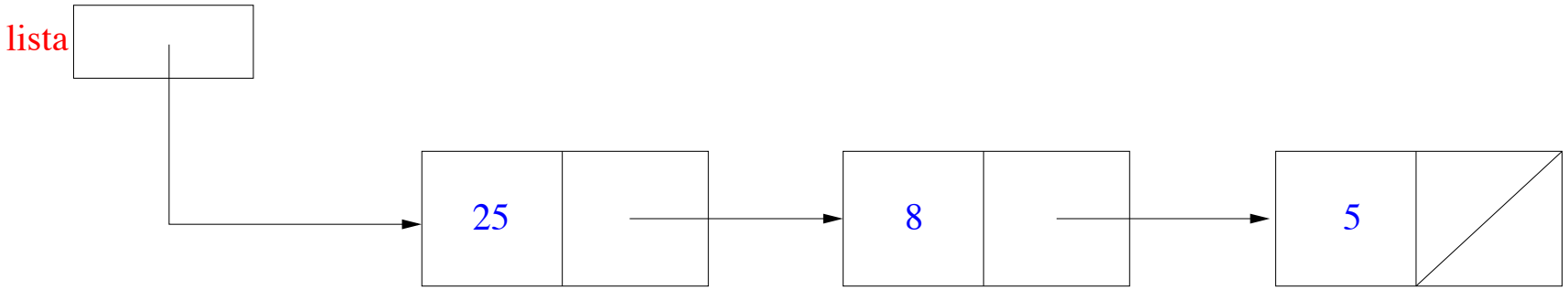
2. Rappresentazione collegata

- Una lista concatenata è una sequenza lineare di nodi, ciascuno dei quali memorizza un valore e contiene un riferimento (puntatore) al nodo successivo nella sequenza.
- Per aggiungere e cancellare nodi in qualunque posizione semplicemente aggiustando il sistema di puntatori senza operare sui nodi non interessati dalla aggiunta o dalla cancellazione.
- L'accesso agli elementi è di tipo sequenziale: per accedere al generico nodo, si deve scandire la lista, dato che l'accesso ad un elemento è possibile attraverso il puntatore contenuto nell'elemento precedente.

2. Rappresentazione collegata (continua)

- La sequenza di elementi viene rappresentata da una struttura di dati collegata, realizzata tramite strutture e puntatori.
- Ogni elemento è rappresentato con una struttura C:
  - un campo (o più campi se necessario) per l'elemento (ad es. `int`)
  - un campo puntatore alla struttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo identico a quello della struttura corrente)
- L'ultimo elemento non ha un elemento successivo
  - il campo puntatore ha valore `NULL` che assume quindi il significato di "fine lista".
- L'inizio della lista è individuato da una variabile del tipo dei puntatori ai vari elementi.
  - Sarà nostra abitudine attribuire a questa variabile il nome stesso della lista, identificando il concetto di "inizio lista" (o "testa della lista") con la lista stessa.
- l'accesso a una lista avviene attraverso il puntatore al primo elemento.

Graficamente



- La variabile lista, di tipo puntatore, è utilizzata per accedere alla sequenza.

Definizione ricorsiva di lista

- Possiamo definire ricorsivamente una lista come segue.
- Una lista è una struttura definita su un insieme di elementi che:
  - non contiene nessun elemento (lista vuota [ ]), oppure
  - contiene un elemento *EL* detto *testa* (head) della lista) seguito dal resto della lista *L*, detta *coda*: ([*EL*, *L*])
- La definizione usata dal C riflette proprio questa definizione. Una variabile di tipo lista può valere NULL (che rappresenta la lista vuota), oppure può essere un puntatore a una struttura che contiene un dato più un puntatore, che rappresenta un'altra lista.

Esempio: Sequenze di interi.

```
struct EL {
  int info;
  struct EL *next;
};
typedef struct EL ElementoLista;
typedef ElementoLista *ListaDiElementi;
```

1. La prima dichiarazione `struct EL` definisce un primo campo, `info`, di tipo `int` e permette di dichiarare il campo `next` come puntatore al tipo strutturato che si sta definendo;
2. la seconda dichiarazione utilizza `typedef` per ridenominare il tipo `struct EL` come `ElementoLista`;
3. la terza dichiarazione definisce il tipo `ListaDiElementi` come puntatore al tipo `ElementoLista`.

- A questo punto possiamo definire variabili di tipo lista:

```
ListaDiElementi Lista1, Lista2;
```

Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

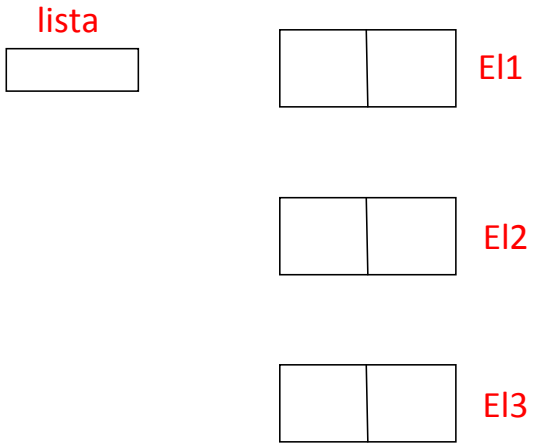
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

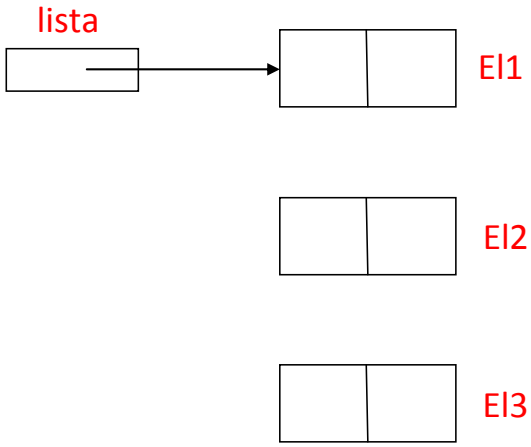
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

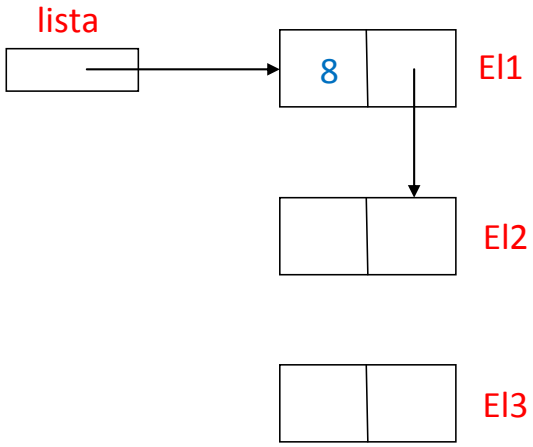
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

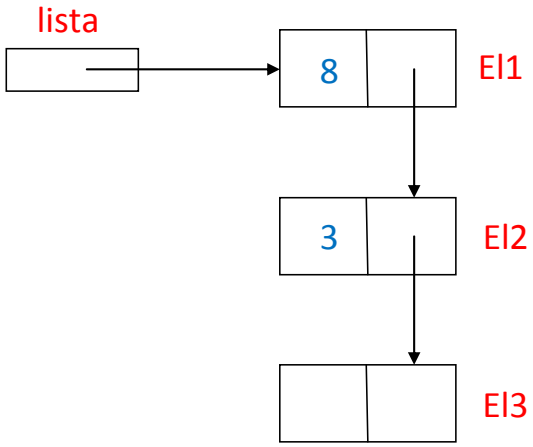
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

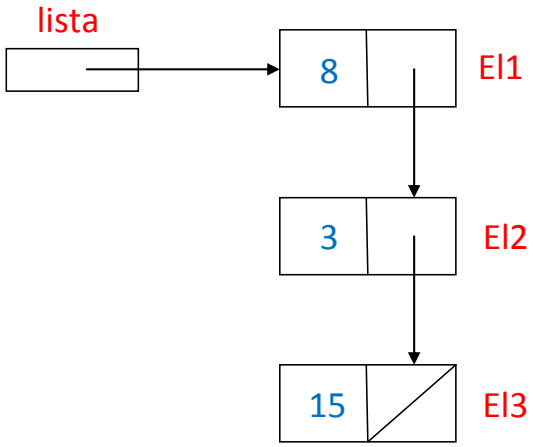
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Aliasing

- Si parla di aliasing quando si utilizzano due puntatori (alias) per far riferimento allo stesso valore.
- Se si modifica il valore puntato da uno dei due, implicitamente (come effetto collaterale) si modifica anche il valore puntato dall'altro, essendo lo stesso.
- Questo è un fenomeno particolarmente rilevante quando si manipolano liste.
- Nell'esempio visto prima, se avessi:
  `lista2=&El2;`
  `lista2 --> info = 9`
  allora avrei anche che la condizione
  `((El1-->next)-->info == 9)` sarebbe vera
  Ricordiamo che `lista2 --> info` equivale a `(*lista2).info`

- Nel programma che abbiamo appena visto per creare una lista di tre elementi dobbiamo dichiarare tre variabili di tipo `ElementoLista`.
- Il numero degli elementi della lista deve essere deciso a tempo di compilazione.
- Non è possibile, per esempio, creare una lista con un numero di elementi letti in ingresso. Ma allora qual è il vantaggio rispetto all'array?
- Quello che abbiamo visto non è l'unico modo. Possiamo ricorrere all'allocazione dinamica della memoria.

Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ListaDiElementi lista;            /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```

PILA HEAP

Creazione di una lista di tre interi fissati: (8, 3, 15)

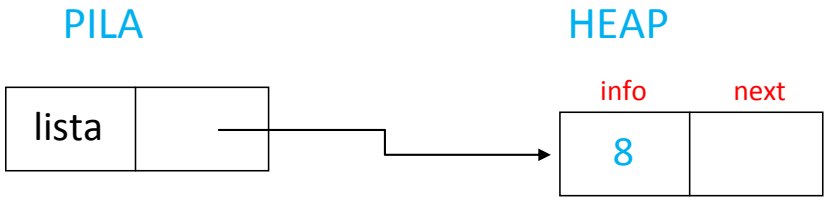
```
ListaDiElementi lista;            /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Creazione di una lista di tre interi fissati: (8, 3, 15)

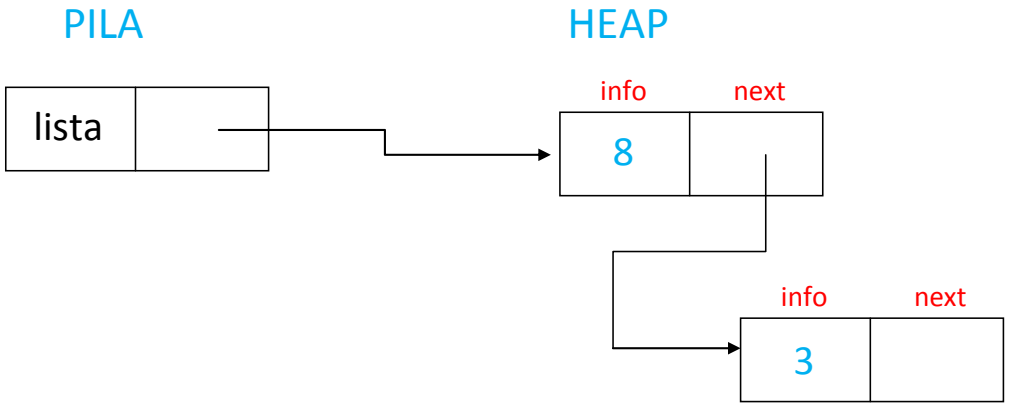
```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));       /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Creazione di una lista di tre interi fissati: (8, 3, 15)

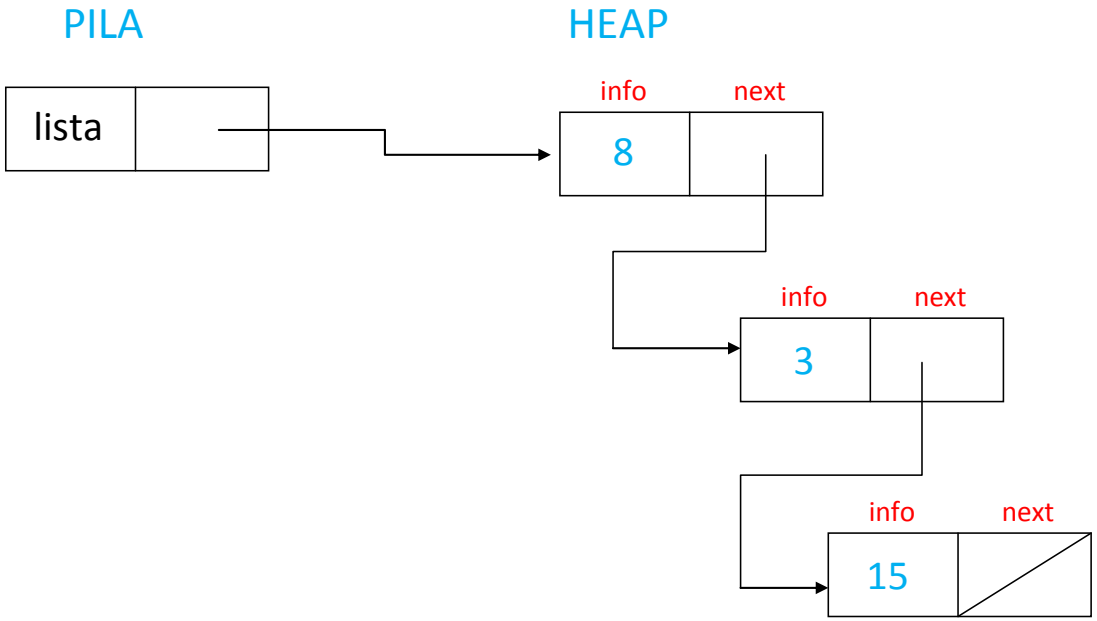
```
ListaDiElementi lista;            /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));       /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Osservazioni:

- `lista` è di tipo `ListaDiElementi`, quindi è un puntatore e non una struttura
- la zona di memoria per ogni elemento della lista (non per ogni variabile di tipo `ListaDiElementi`) deve essere allocata esplicitamente con `malloc`
- Esiste un modo più semplice di creare la lista di 3 elementi?
- Creiamo la lista a partire dal fondo!

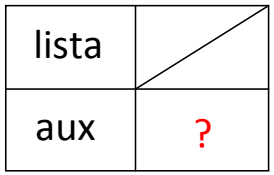
```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

PILA HEAP

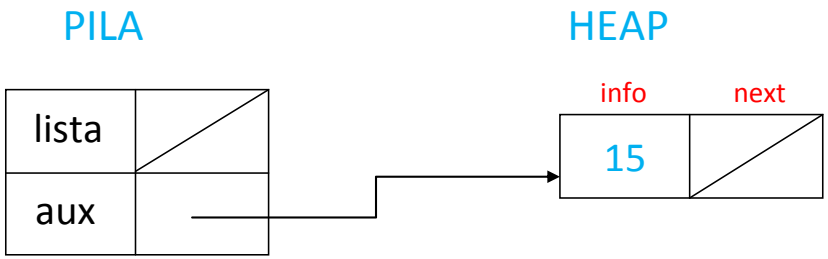


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

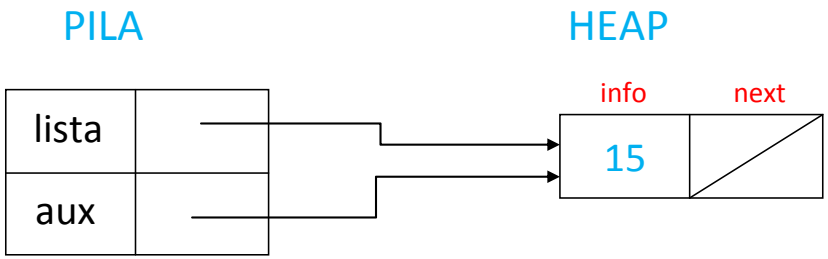


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

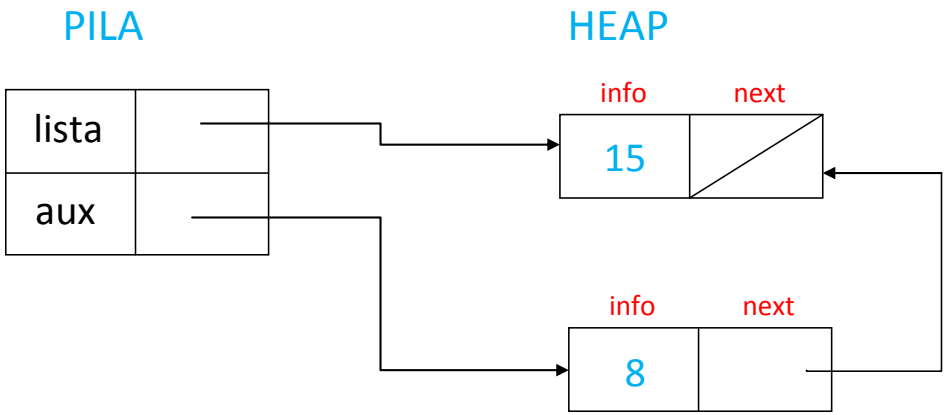


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

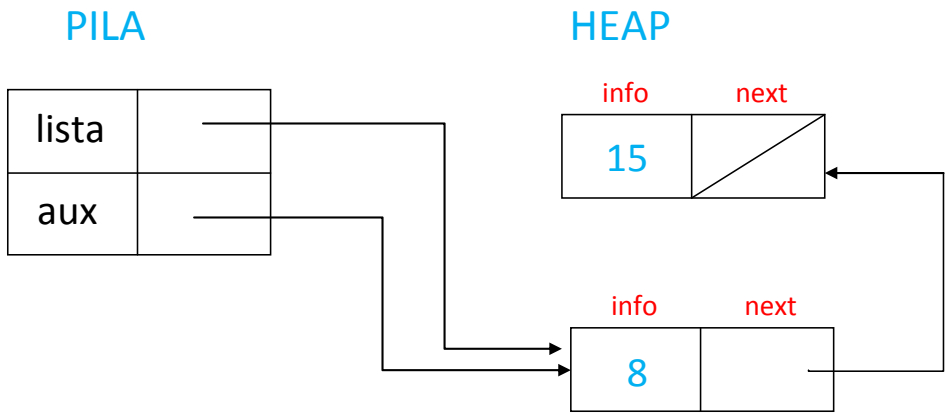


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

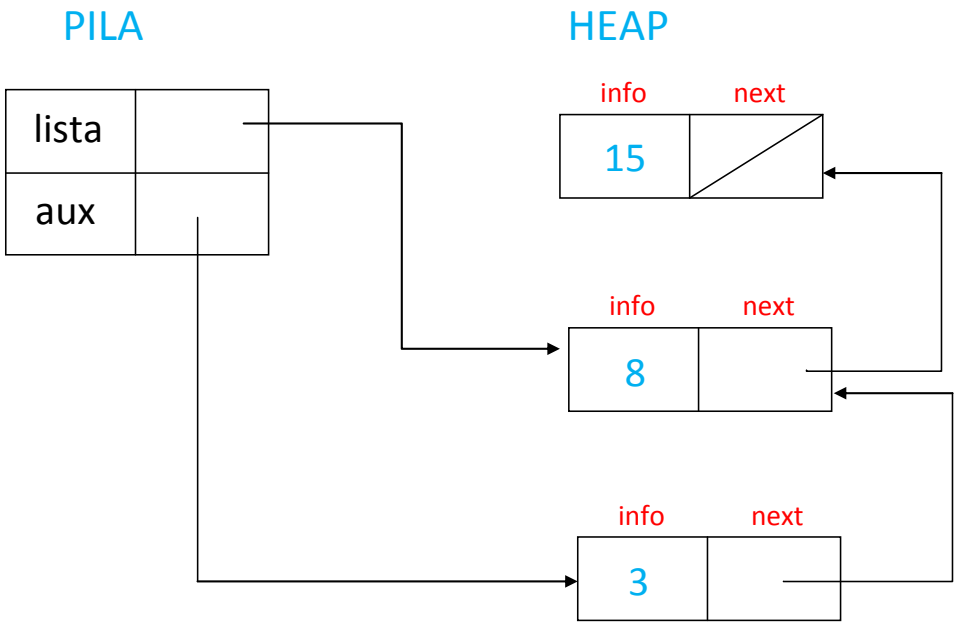


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

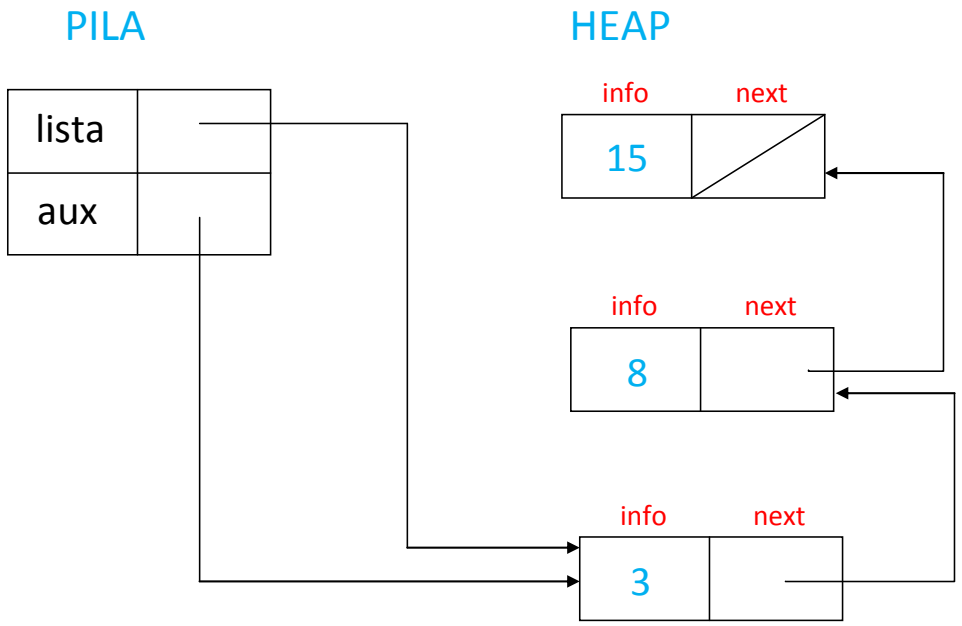


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```



Operazioni sulle liste

- Definiamo una serie di procedure e funzioni per operare sulle liste.
- Usiamo liste di interi per semplicità, ma tutte le operazioni sono realizzabili in modo del tutto analogo su liste di altro tipo (salvo rare eccezioni)
- Facciamo riferimento alle dichiarazioni dei tipi `ElementoLista` e `ListaDiElementi` viste in precedenza

Inizializzazione

- Definiamo una procedura che inizializza una lista assegnando il valore `NULL` alla variabile testa della lista.
- Tale variabile deve essere modificata e quindi passata per indirizzo.
- Ciò provoca, nell'intestazione della procedura, la presenza di un puntatore a puntatore.

```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

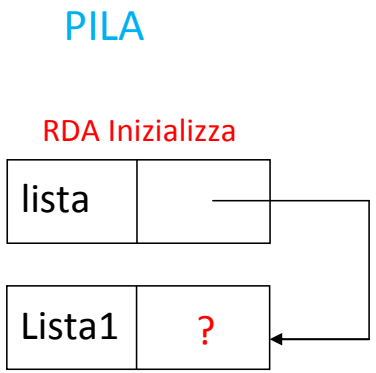
PILA

Lista1	?

```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

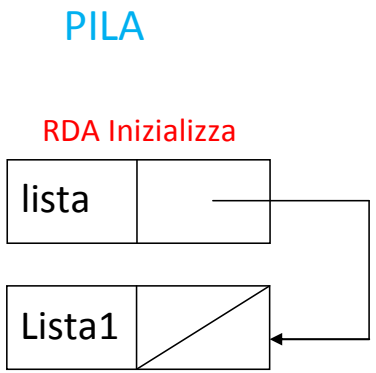
```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```



```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

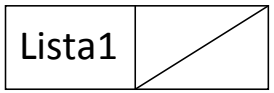


```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

PILA

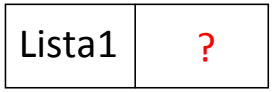


Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA



Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA

RDA Inizializza

lista	?

Lista1	?

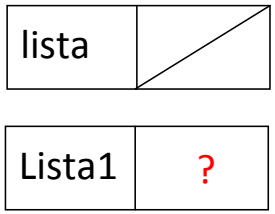
Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA

RDA Inizializza

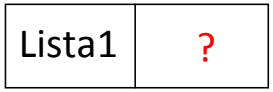


Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA



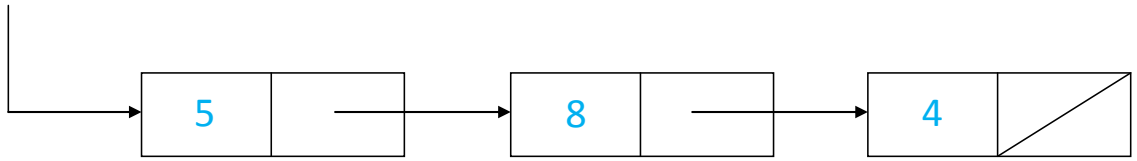
Controllo lista vuota

```
boolean ListaVuota(ListaDiElementi lista)
{
 return (lista==NULL);
}
```

A `lista` viene passato il valore contenuto nella variabile testa di lista e quindi punta al primo elemento della lista considerata.

Stampa degli elementi di una lista

- Data la lista



vogliamo che venga stampato:

5 –> 8 –> 4 –> //

Versione iterativa:

```
void StampaLista(ListaDiElementi lis)
{
   while (lis != NULL)
    {
     printf("%d -->", lis->info);
     lis = lis->next;
    }
printf("//");
}
```

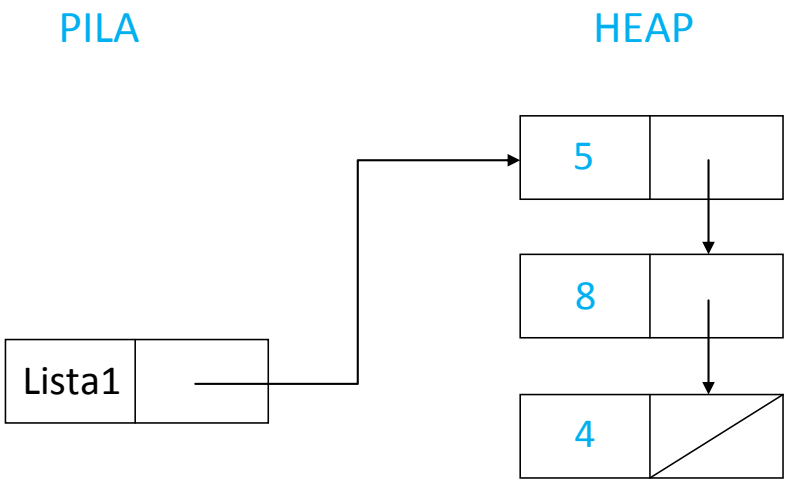
- N.B.: `lis = lis->next` fa puntare `lis` all'elemento successivo della lista
- Attenzione: Possiamo usare `lis` per scorrere la lista perché, avendo utilizzato il passaggio per valore, le modifiche a `lis` non si ripercuotono sul parametro attuale.

```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```

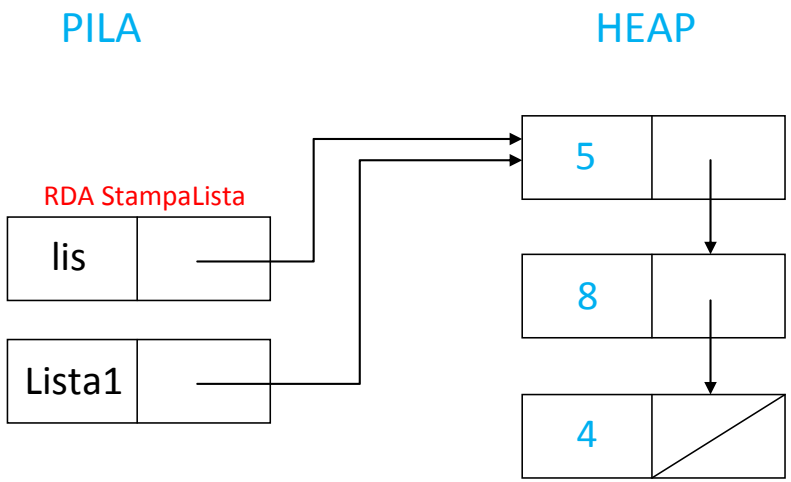
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



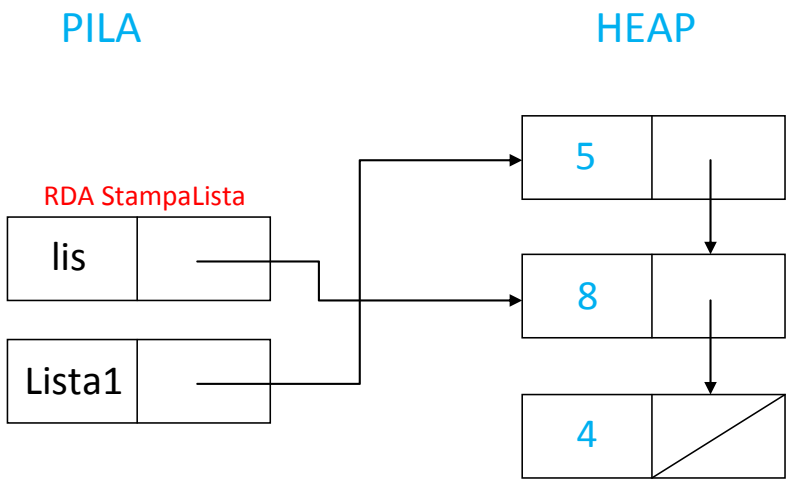
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

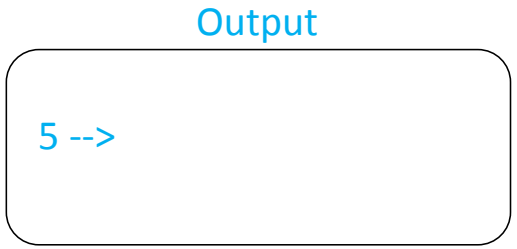
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

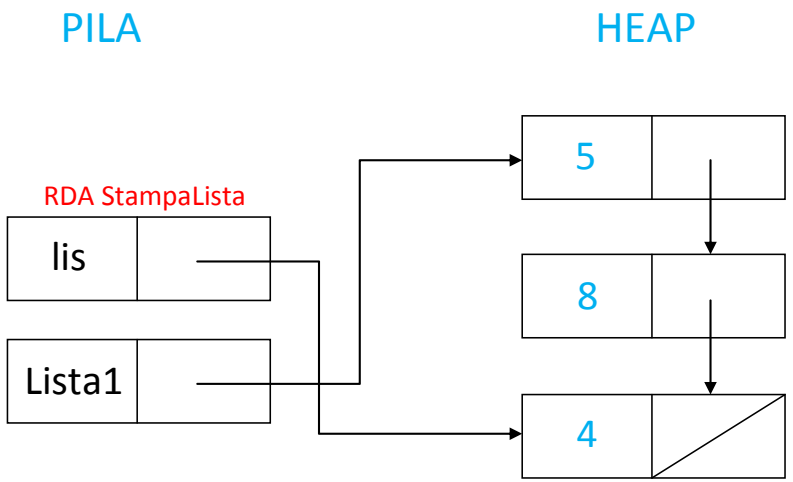
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```





```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```

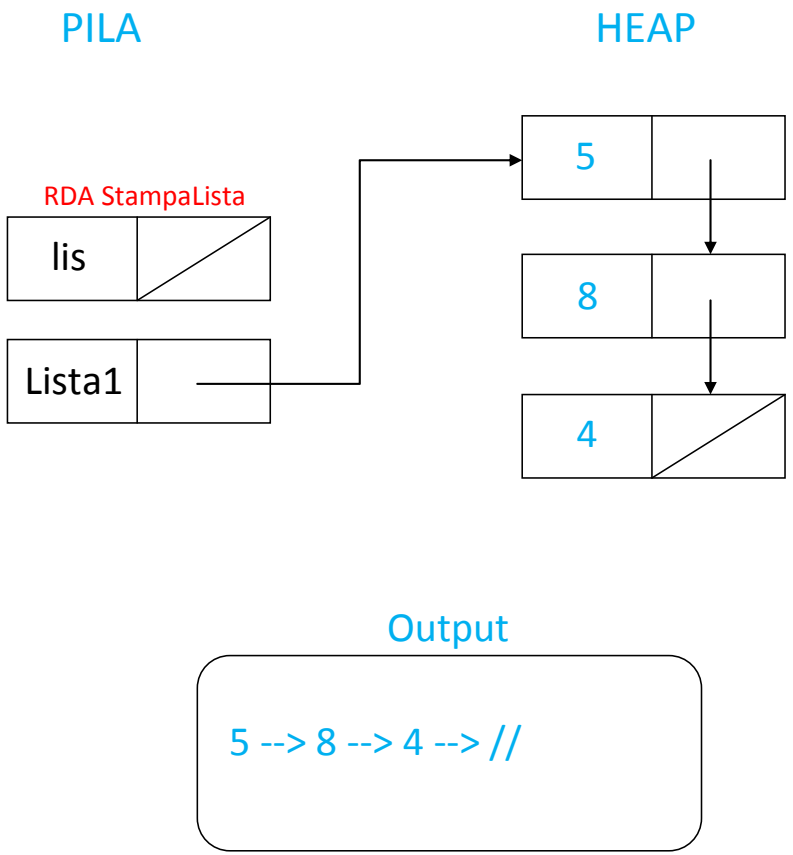


Output

5 --> 8 -->

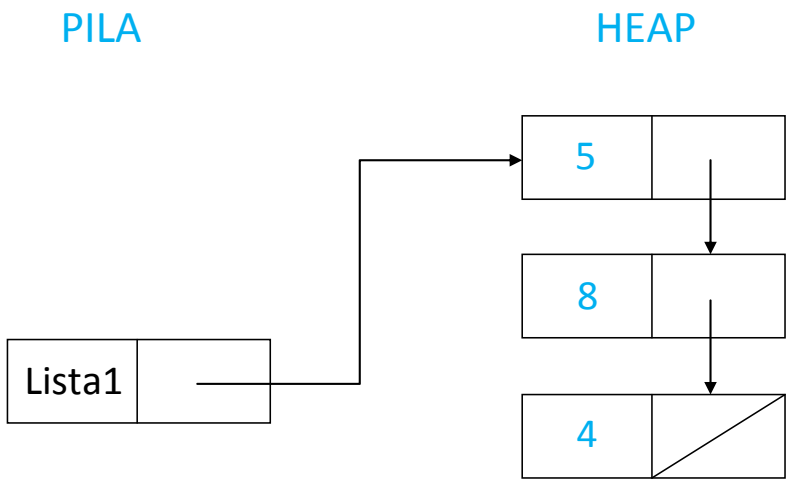
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



Output

```
5 --> 8 --> 4 --> //
```

Cosa sarebbe successo passando il parametro per indirizzo?

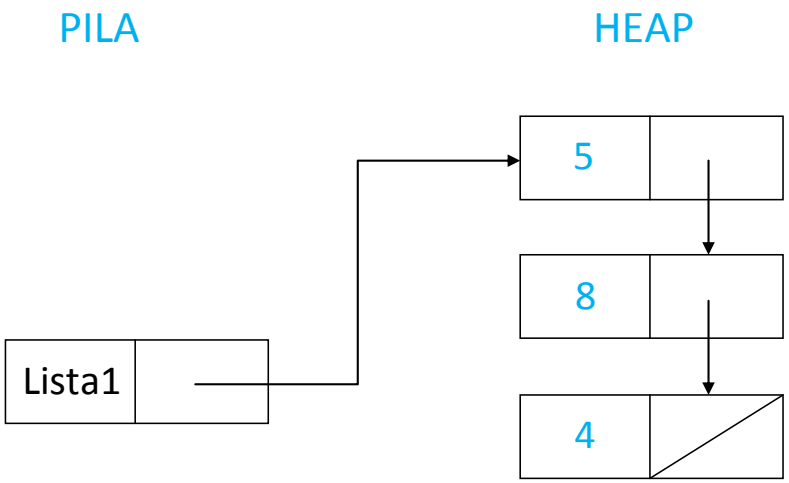
```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

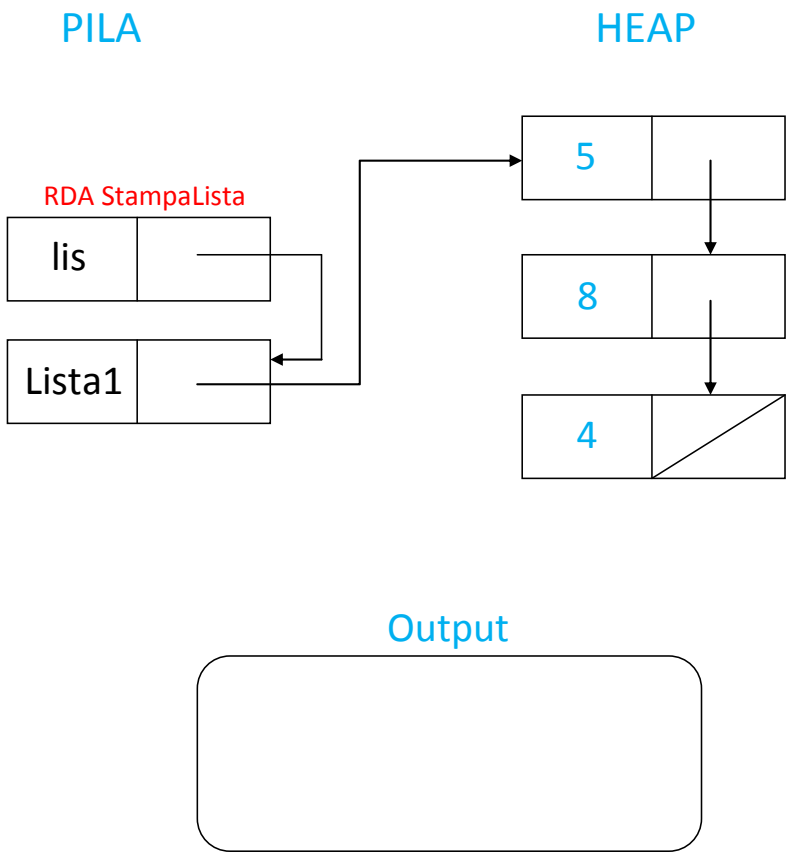
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

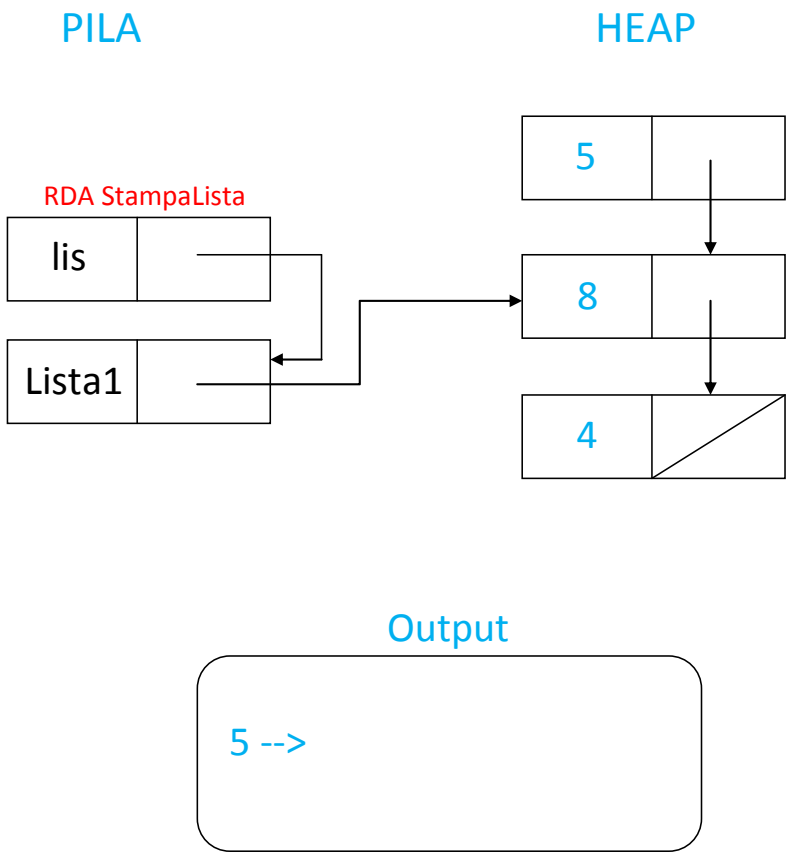
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

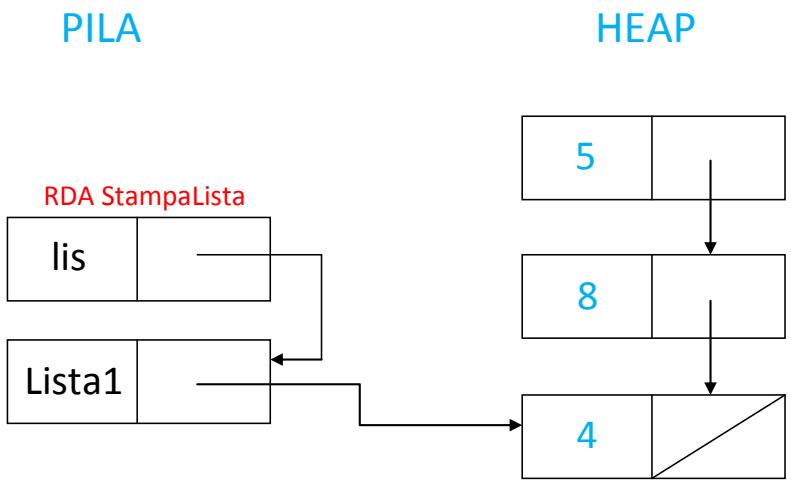
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



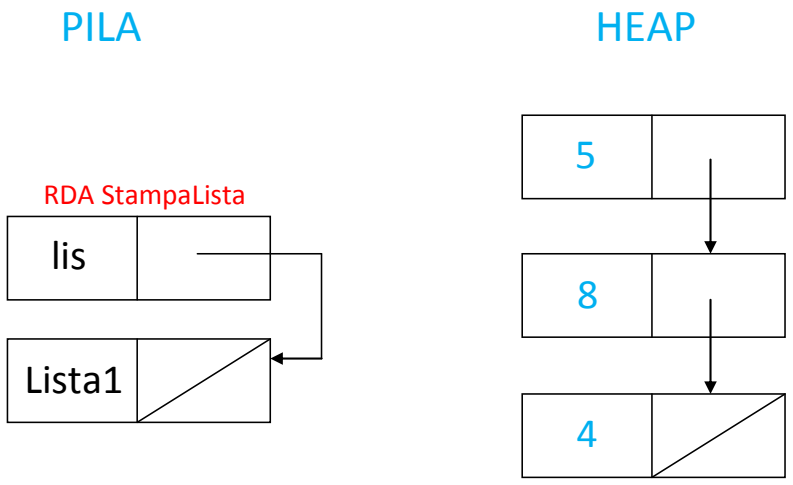
Output

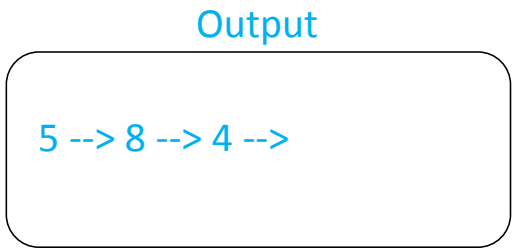
5 --> 8 -->

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```

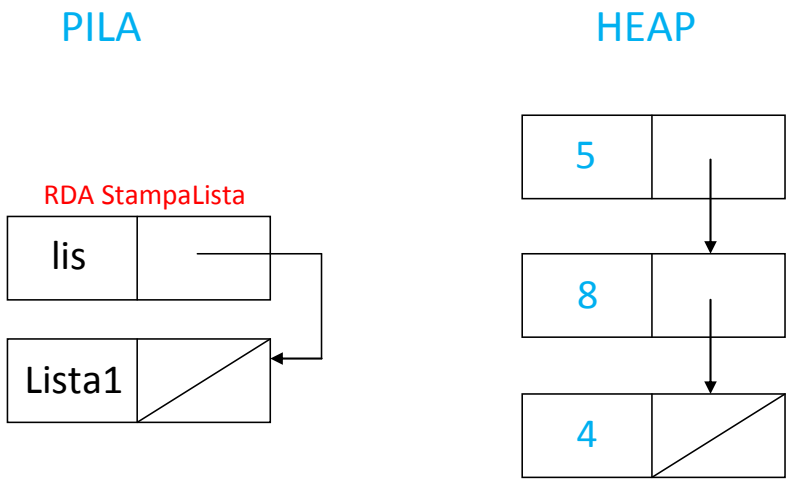




Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



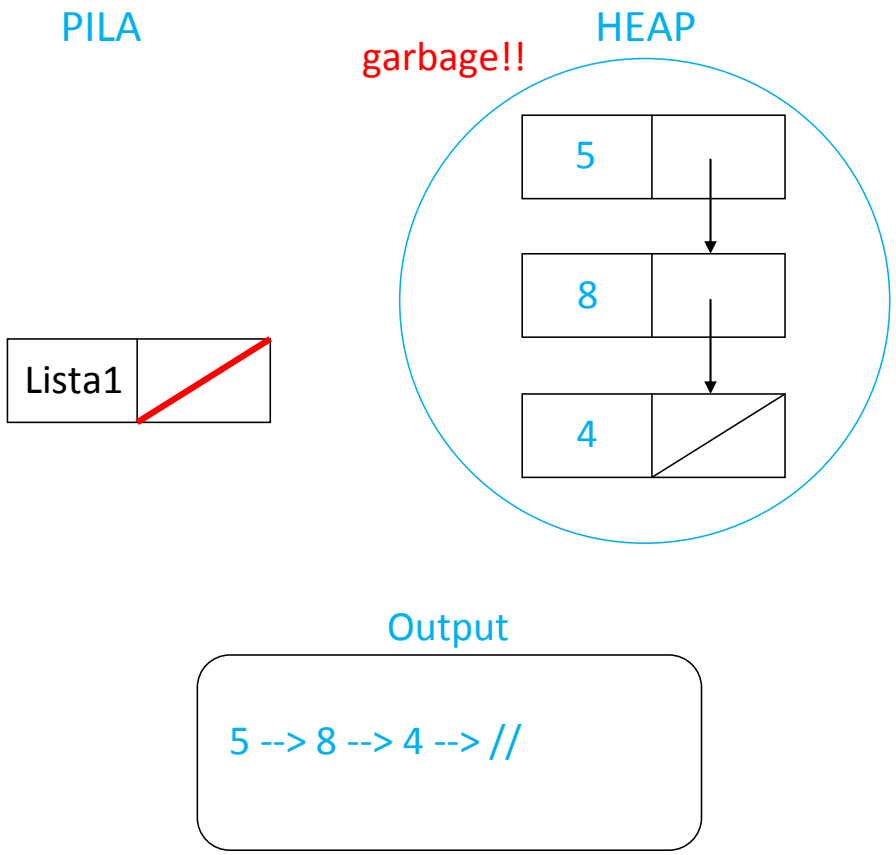
Output

5 --> 8 --> 4 --> //

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

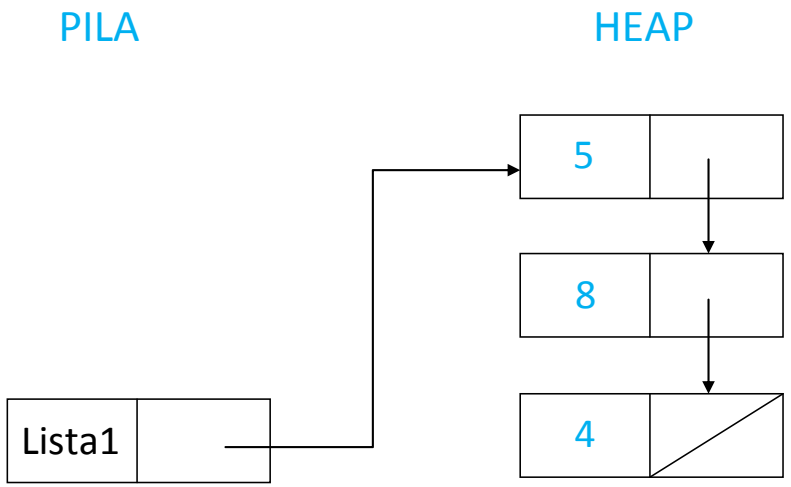
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```

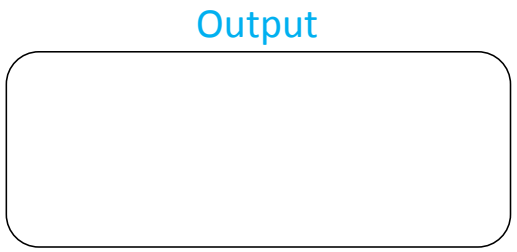
Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



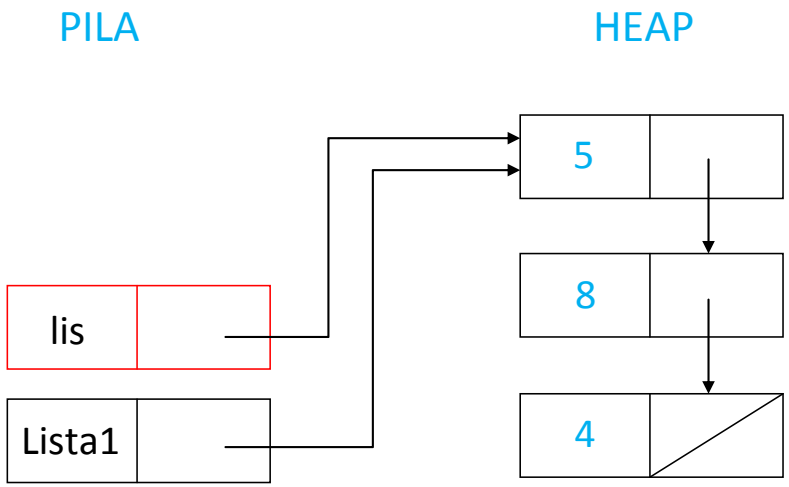


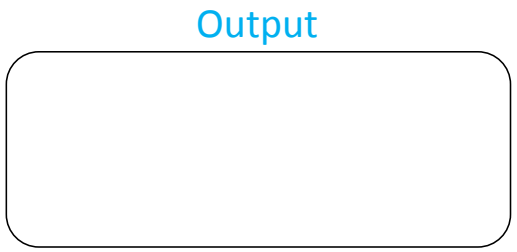
Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



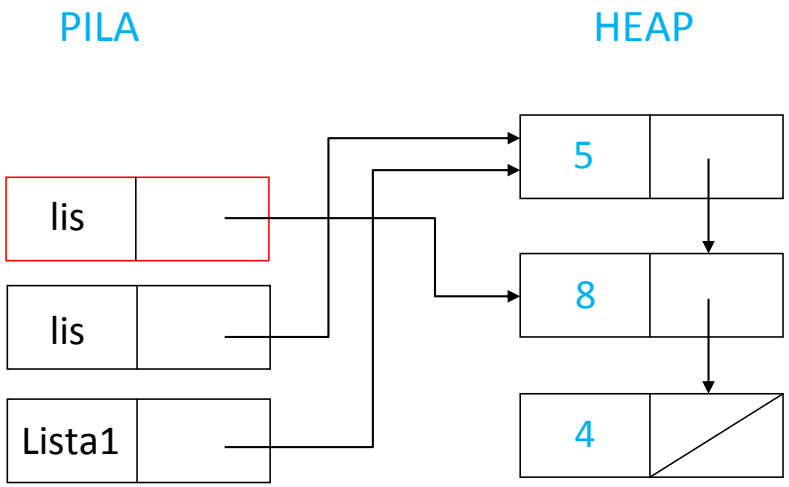


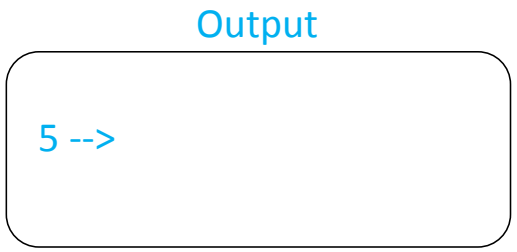
Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



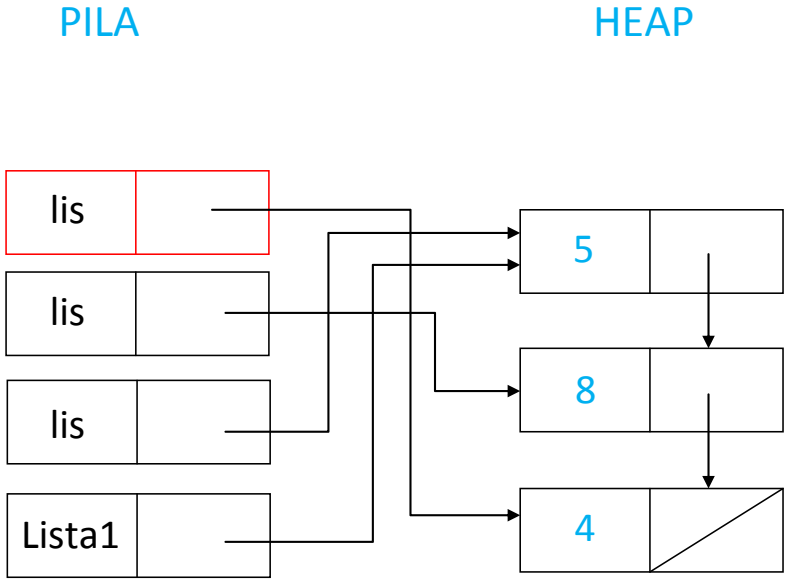


Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Output

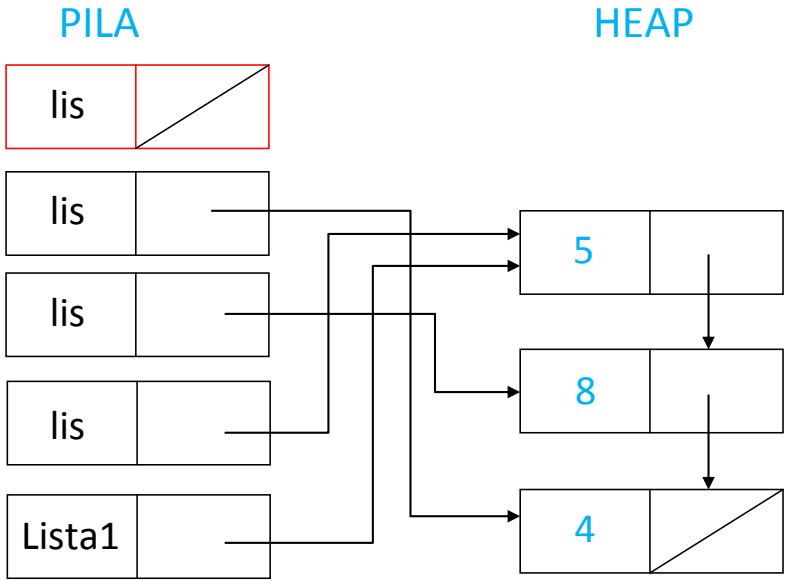
5 --> 8 -->

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Output

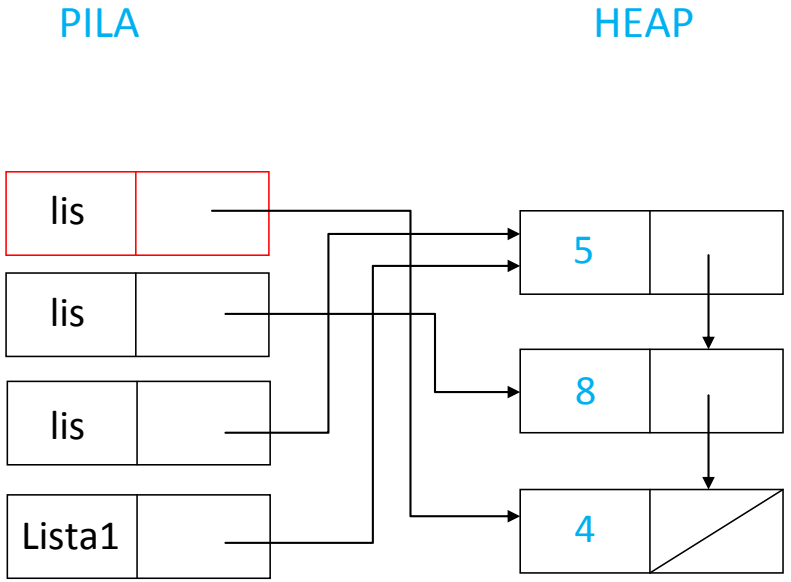
5 --> 8 --> 4 -->

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Output

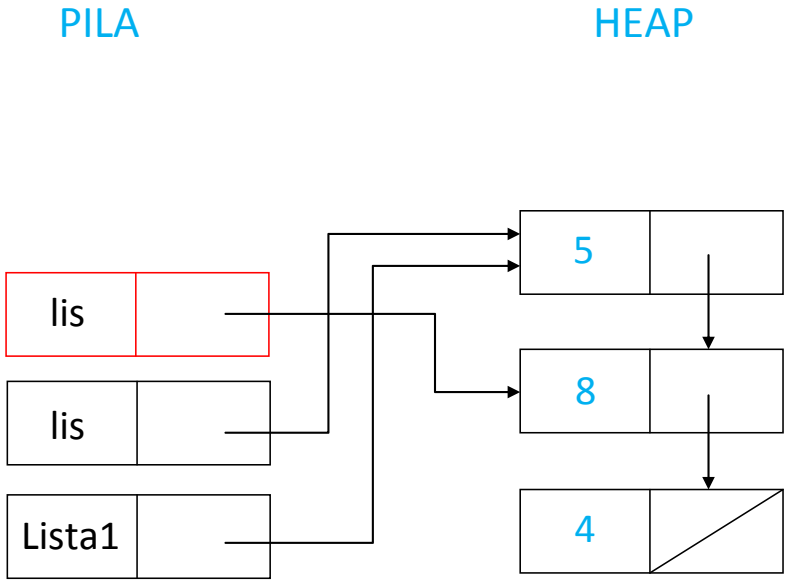
5 --> 8 --> 4 --> //

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Output

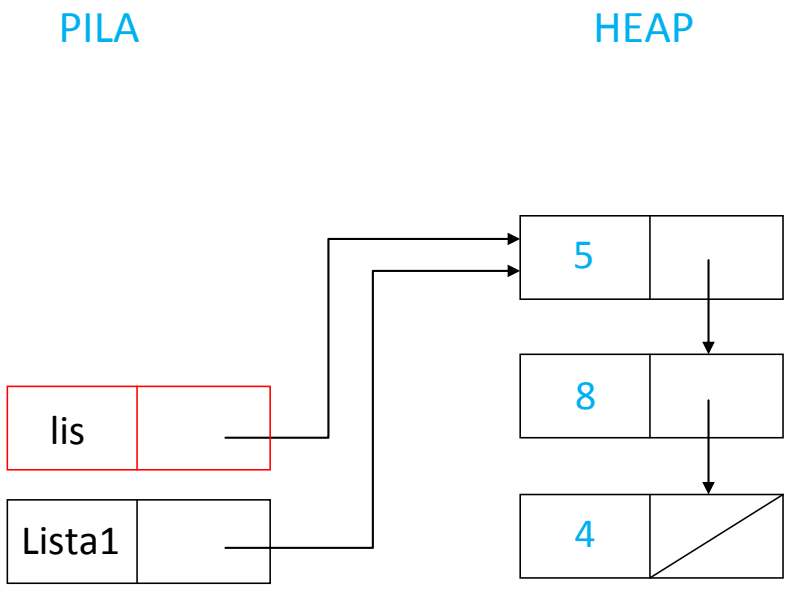
5 --> 8 --> 4 --> //

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Output

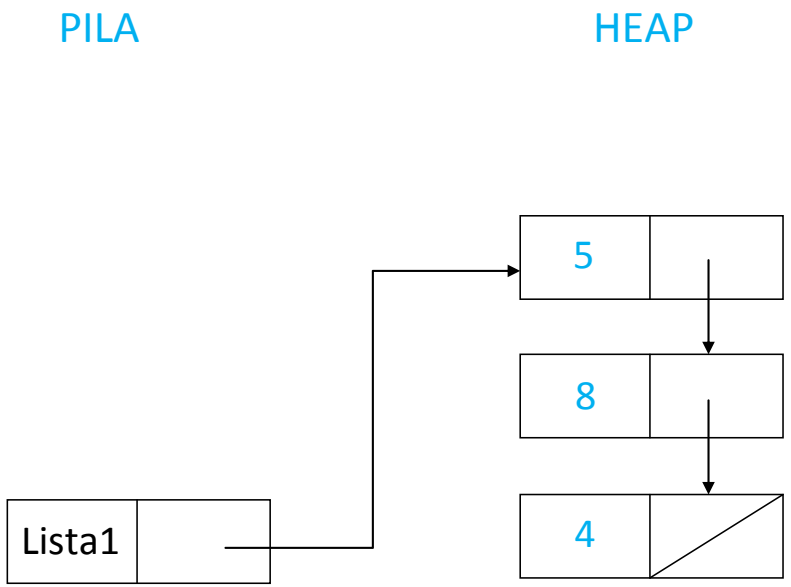
5 --> 8 --> 4 --> //

Versione ricorsiva

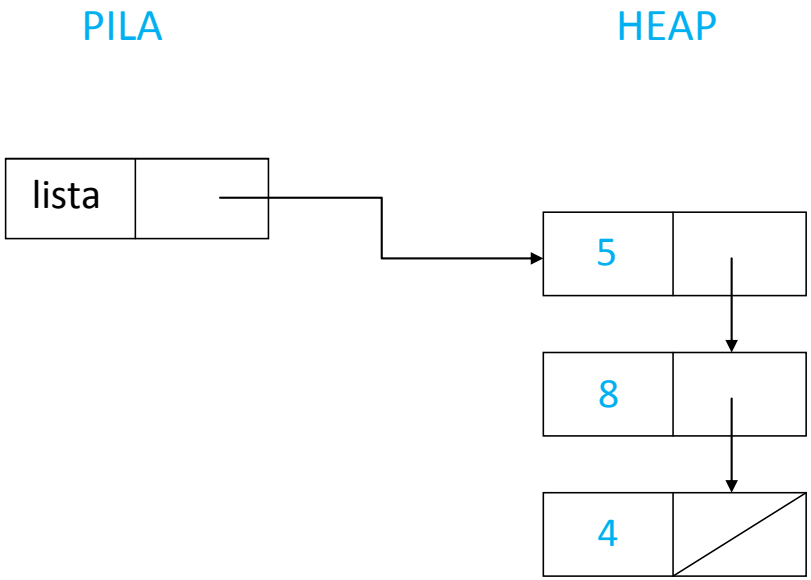
Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



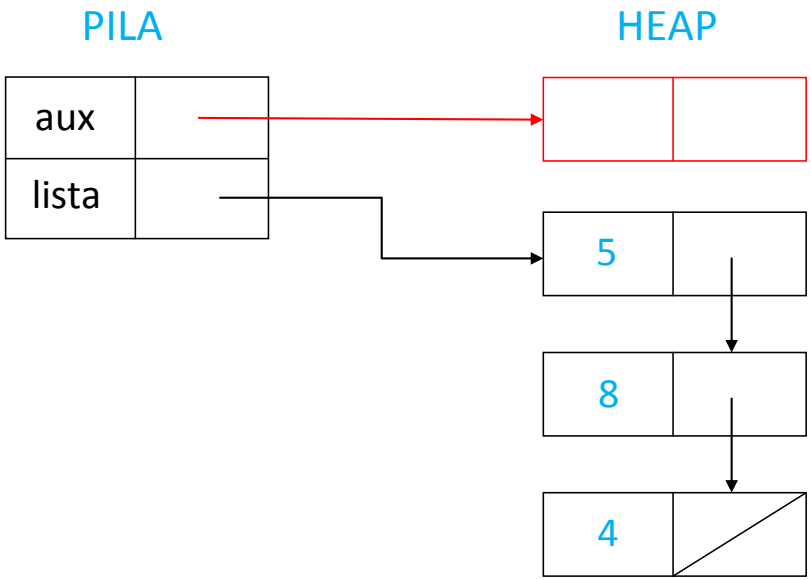
Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura

   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

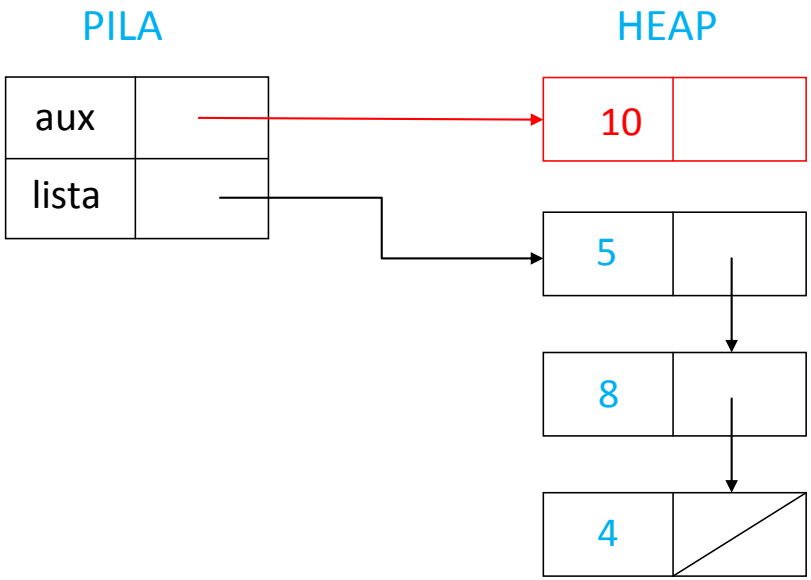
Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura

   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

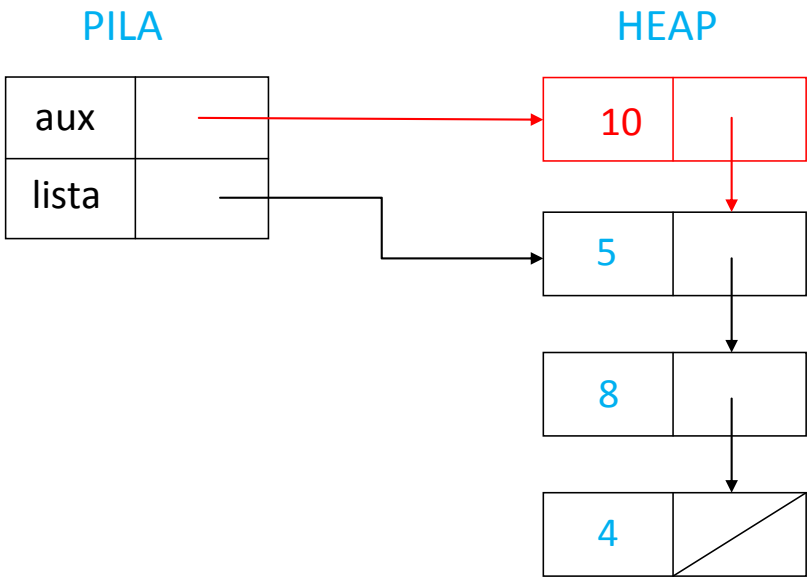
Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura

   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

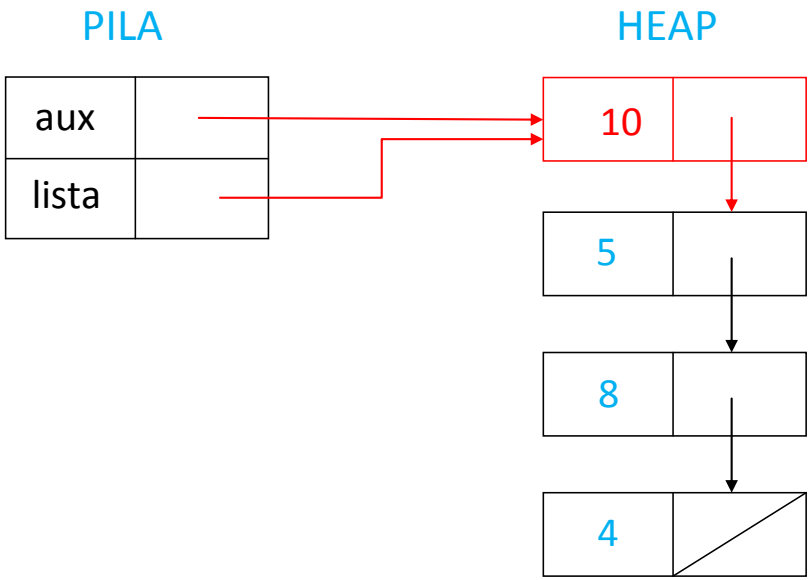
Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura

   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura

   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

```
void InserisciTestaLista(ListaDiElementi *lista, int elem)
{
  ListaDiElementi aux;

  aux = malloc(sizeof(ElementoLista));
  aux->info = elem;
  aux->next = *lista;
  *lista = aux;
}
```

- il primo parametro è la lista da modificare (passata per indirizzo)
- il secondo parametro è l'elemento da inserire (passato per indirizzo)

Esercizio

Impostare una chiamata alla procedura e tracciare l'evoluzione di pila e heap

```
void InserisciTestaLista(ListaDiElementi *lista, TipoElemLista elem)
{
  ListaDiElementi aux;

  aux = malloc(sizeof(ElementoLista));
  aux->info = elem;
  aux->next = *lista;
  *lista = aux;
}
```

- il primo parametro è la lista da modificare (passata per indirizzo)
- il secondo parametro è l'elemento da inserire (passato per indirizzo)

Esercizio

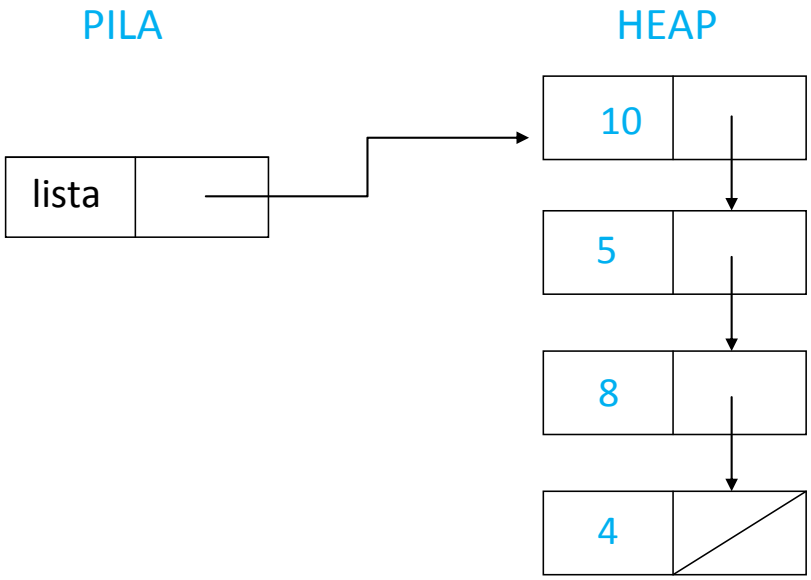
Impostare una chiamata alla procedura e tracciare l'evoluzione di pila e heap

- il primo parametro è la lista da modificare (passata per indirizzo)
- il secondo parametro è l'elemento da inserire (passato per indirizzo)
  - Attenzione: nel caso di liste di tipo `TipoElemLista` la procedura può essere generalizzata se su tale tipo è definito l'assegnamento

Esercizio

Impostare una chiamata alla procedura e tracciare l'evoluzione di pila e heap

Cancellazione del primo elemento



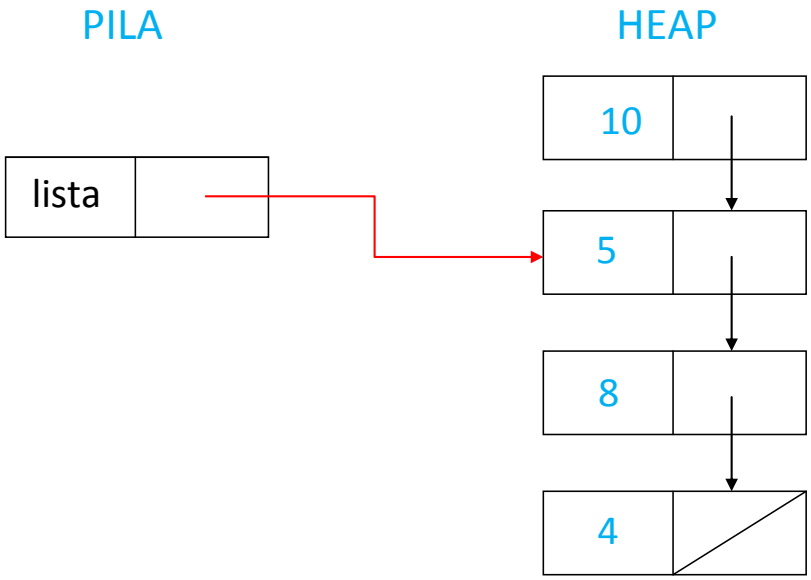
- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento

  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare

  $\Longrightarrow$ è necessario un puntatore ausiliario

Cancellazione del primo elemento



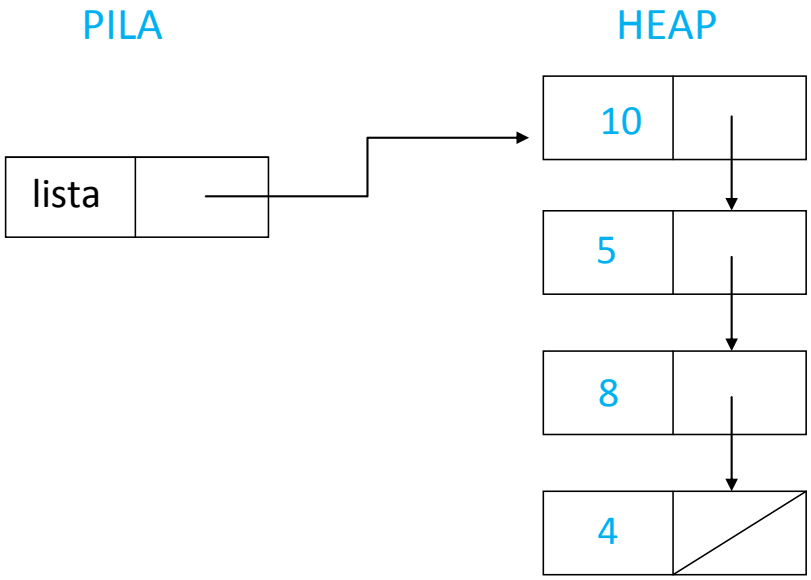
- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento

  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare

  $\Longrightarrow$ è necessario un puntatore ausiliario

Cancellazione del primo elemento



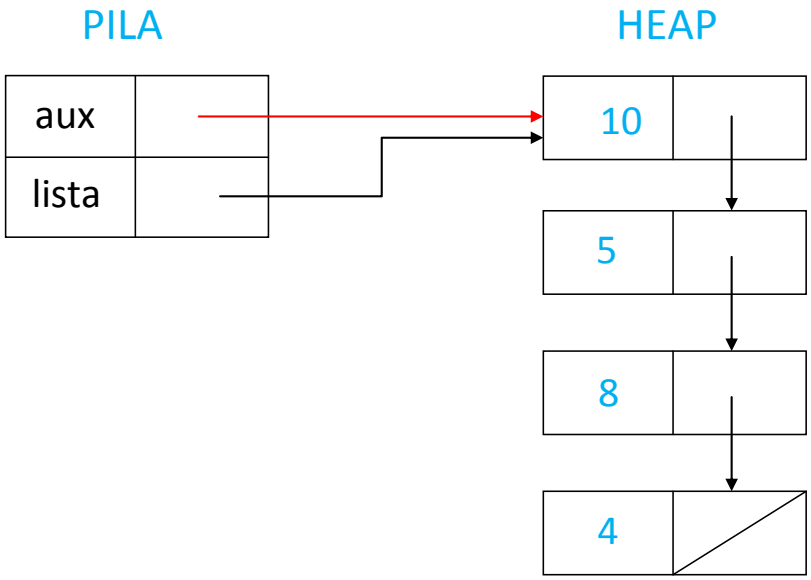
- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento

  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare

  $\Longrightarrow$ è necessario un puntatore ausiliario

Cancellazione del primo elemento



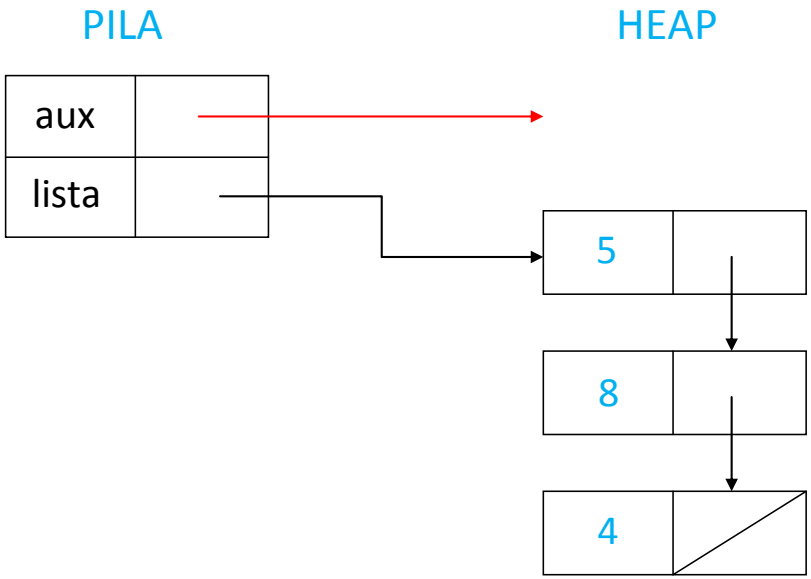
- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento

  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare

  $\Longrightarrow$ è necessario un puntatore ausiliario

Cancellazione del primo elemento



- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento

  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare

  $\Longrightarrow$ è necessario un puntatore ausiliario

```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}
```

Liste

- È molto comune dover rappresentare sequenze di elementi tutti dello stesso tipo e fare operazioni su di esse.

  Esempi: sequenza di interi (23 46 5 28 3)
  sequenza di caratteri (`'x' 'r' 'f'`)
  sequenza di persone con nome e data di nascita

- Finora abbiamo usato gli array per realizzare tali strutture, nonostante ciò porti talvolta a un impiego inefficiente della memoria.
- Vediamo adesso un modo basato sull'allocazione dinamica di variabili, che ci permette di realizzare liste di elementi in maniera che la memoria fisica utilizzata corrisponda meglio a quella astratta, cioè al numero di elementi della sequenza che vogliamo rappresentare.

Diversi modi di rappresentare sequenze di elementi

1. Rappresentazione sequenziale: tramite array
   - Vantaggi:
     - l'accesso agli elementi è diretto (tramite indice) ed efficiente
     - l'ordine degli elementi è quello in memoria $\Longrightarrow$ non servono strutture dati addizionali
     - è semplice manipolare l'intera struttura (copia, ordinamento, ...)
   - Svantaggi:
     - dobbiamo avere un'idea precisa della dimensione della sequenza
     - inserire o eliminare elementi è complicato ed inefficiente (comporta un numero di spostamenti che nel caso peggiore può essere dell'ordine del numero degli elementi della struttura)

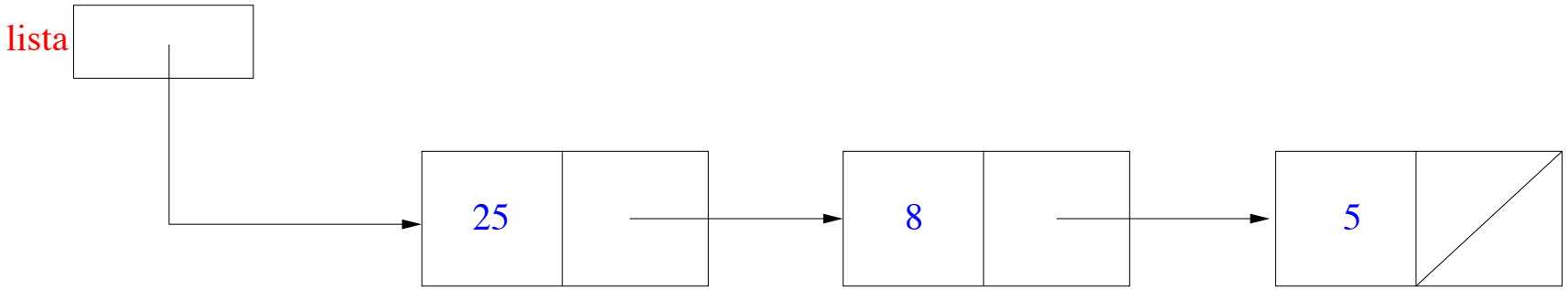
2. Rappresentazione collegata

- Una lista concatenata è una sequenza lineare di nodi, ciascuno dei quali memorizza un valore e contiene un riferimento (puntatore) al nodo successivo nella sequenza.
- Per aggiungere e cancellare nodi in qualunque posizione semplicemente aggiustando il sistema di puntatori senza operare sui nodi non interessati dalla aggiunta o dalla cancellazione.
- L'accesso agli elementi è di tipo sequenziale: per accedere al generico nodo, si deve scandire la lista, dato che l'accesso ad un elemento è possibile attraverso il puntatore contenuto nell'elemento precedente.

2. Rappresentazione collegata (continua)

- La sequenza di elementi viene rappresentata da una struttura di dati collegata, realizzata tramite strutture e puntatori.
- Ogni elemento è rappresentato con una struttura C:
  - un campo (o più campi se necessario) per l'elemento (ad es. `int`)
  - un campo puntatore alla struttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo identico a quello della struttura corrente)
- L'ultimo elemento non ha un elemento successivo
  - il campo puntatore ha valore `NULL` che assume quindi il significato di "fine lista".
- L'inizio della lista è individuato da una variabile del tipo dei puntatori ai vari elementi.
  - Sarà nostra abitudine attribuire a questa variabile il nome stesso della lista, identificando il concetto di "inizio lista" (o "testa della lista") con la lista stessa.
- l'accesso a una lista avviene attraverso il puntatore al primo elemento.

Graficamente



- La variabile lista, di tipo puntatore, è utilizzata per accedere alla sequenza.

Definizione ricorsiva di lista

- Possiamo definire ricorsivamente una lista come segue.
- Una lista è una struttura definita su un insieme di elementi che:
  - non contiene nessun elemento (lista vuota [ ]), oppure
  - contiene un elemento *EL* detto *testa* (head) della lista) seguito dal resto della lista *L*, detta *coda*: ([*EL*, *L*])
- La definizione usata dal C riflette proprio questa definizione. Una variabile di tipo lista può valere NULL (che rappresenta la lista vuota), oppure può essere un puntatore a una struttura che contiene un dato più un puntatore, che rappresenta un'altra lista.

Esempio: Sequenze di interi.

```
struct EL {
  int info;
  struct EL *next;
};
typedef struct EL ElementoLista;
typedef ElementoLista *ListaDiElementi;
```

1. La prima dichiarazione `struct EL` definisce un primo campo, `info`, di tipo `int` e permette di dichiarare il campo `next` come puntatore al tipo strutturato che si sta definendo;
2. la seconda dichiarazione utilizza `typedef` per ridenominare il tipo `struct EL` come `ElementoLista`;
3. la terza dichiarazione definisce il tipo `ListaDiElementi` come puntatore al tipo `ElementoLista`.

- A questo punto possiamo definire variabili di tipo lista:

```
ListaDiElementi Lista1, Lista2;
```

Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

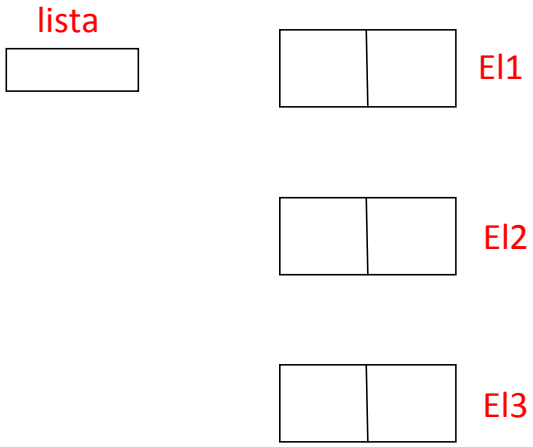
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

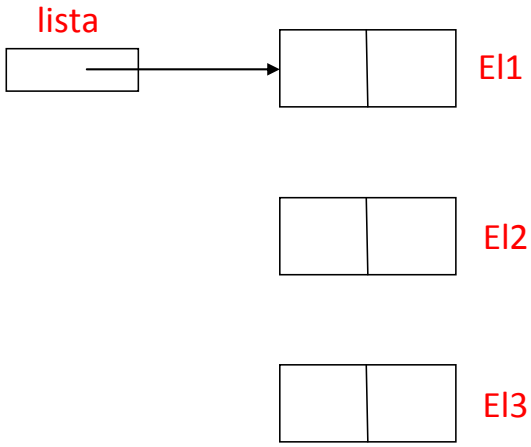
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

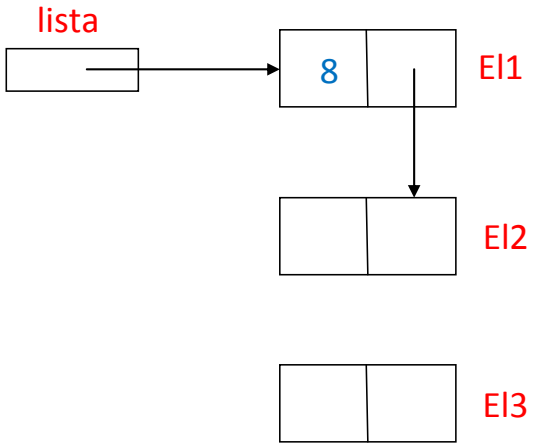
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

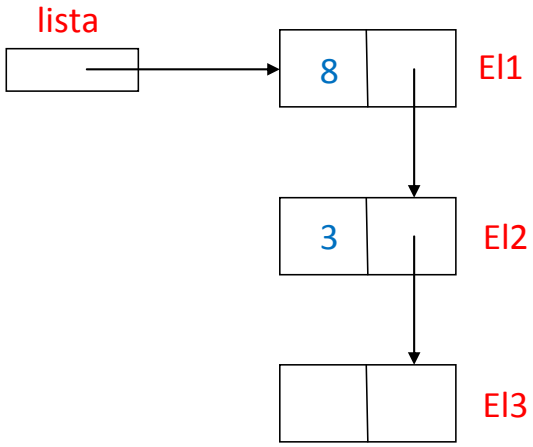
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

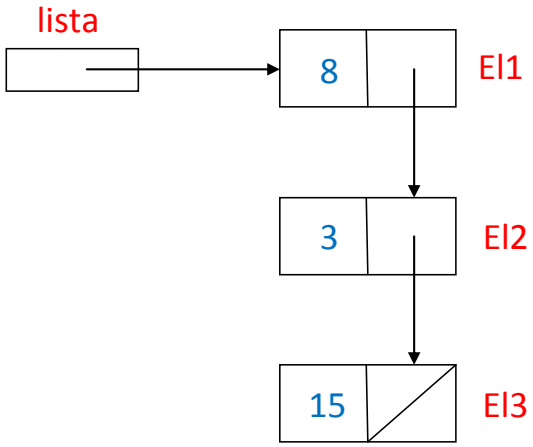
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Aliasing

- Si parla di aliasing quando si utilizzano due puntatori (alias) per far riferimento allo stesso valore.
- Se si modifica il valore puntato da uno dei due, implicitamente (come effetto collaterale) si modifica anche il valore puntato dall'altro, essendo lo stesso.
- Questo è un fenomeno particolarmente rilevante quando si manipolano liste.
- Nell'esempio visto prima, se avessi:
  `lista2=&El2;`
  `lista2 --> info = 9`
  allora avrei anche che la condizione
  `((El1-->next)-->info == 9)` sarebbe vera
  Ricordiamo che `lista2 --> info` equivale a `(*lista2).info`

- Nel programma che abbiamo appena visto per creare una lista di tre elementi dobbiamo dichiarare tre variabili di tipo `ElementoLista`.
- Il numero degli elementi della lista deve essere deciso a tempo di compilazione.
- Non è possibile, per esempio, creare una lista con un numero di elementi letti in ingresso. Ma allora qual è il vantaggio rispetto all'array?
- Quello che abbiamo visto non è l'unico modo. Possiamo ricorrere all'allocazione dinamica della memoria.

Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ListaDiElementi lista;            /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```

PILA HEAP

Creazione di una lista di tre interi fissati: (8, 3, 15)

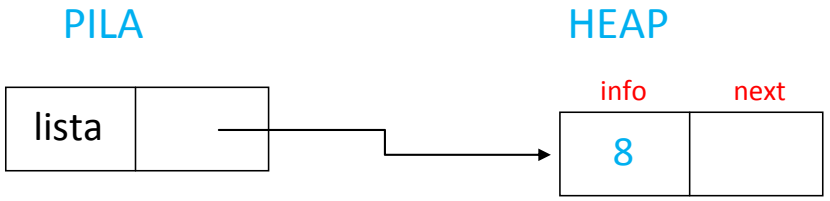
```
ListaDiElementi lista;            /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Creazione di una lista di tre interi fissati: (8, 3, 15)

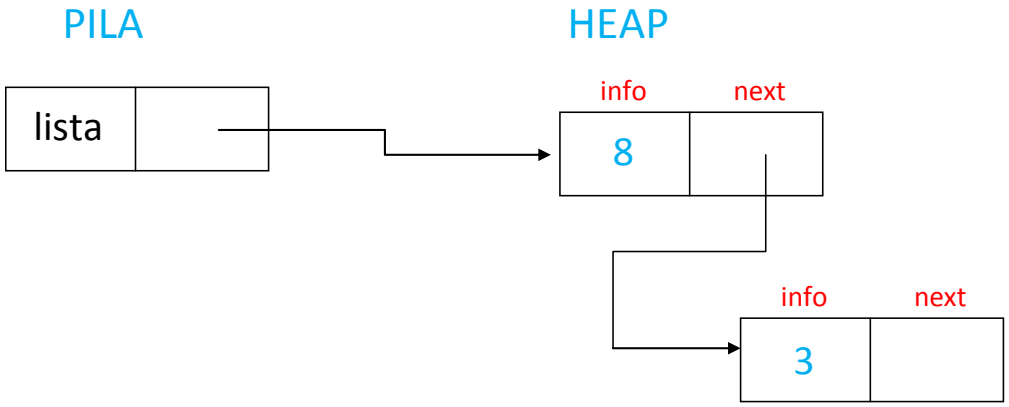
```
ListaDiElementi lista;            /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Creazione di una lista di tre interi fissati: (8, 3, 15)

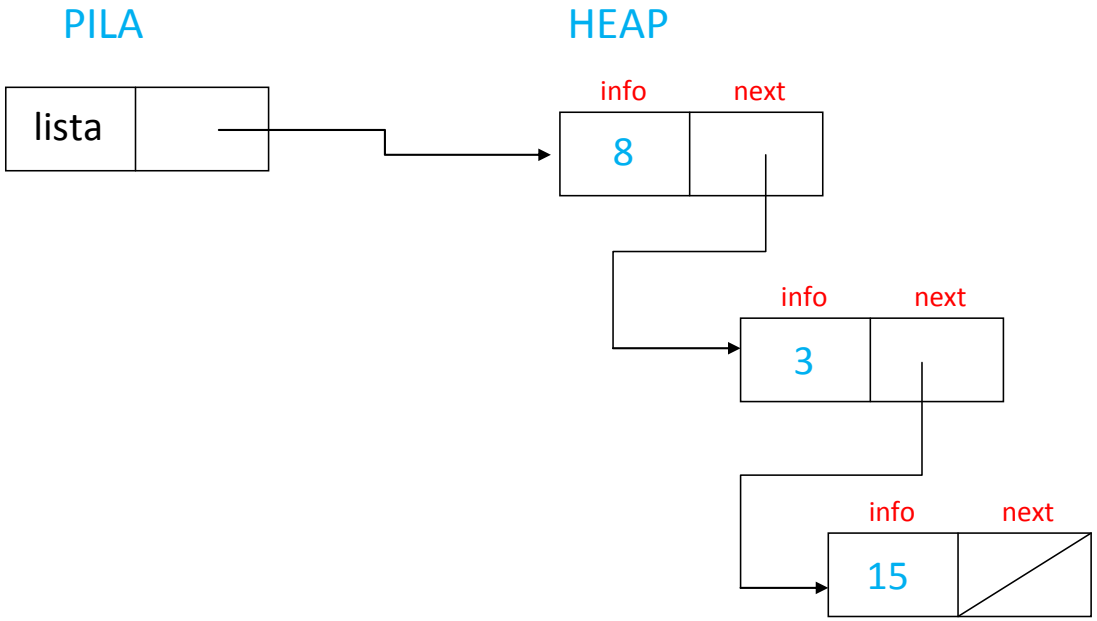
```
ListaDiElementi lista;              /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));        /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Osservazioni:

- `lista` è di tipo `ListaDiElementi`, quindi è un puntatore e non una struttura, così come, in **int** `*p`, `p` è un puntatore a intero e non un intero.
- la zona di memoria per ogni elemento della lista (non per ogni variabile di tipo `ListaDiElementi`) deve essere allocata esplicitamente con `malloc`
- Esiste un modo più semplice di creare la lista di 3 elementi?
- Creiamo la lista a partire dal fondo!

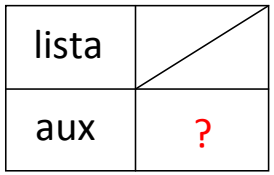
```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

PILA HEAP

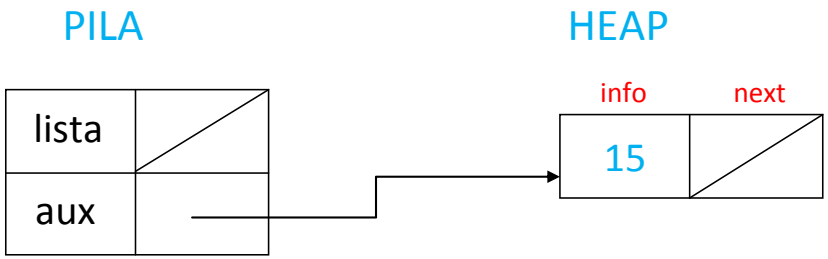


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

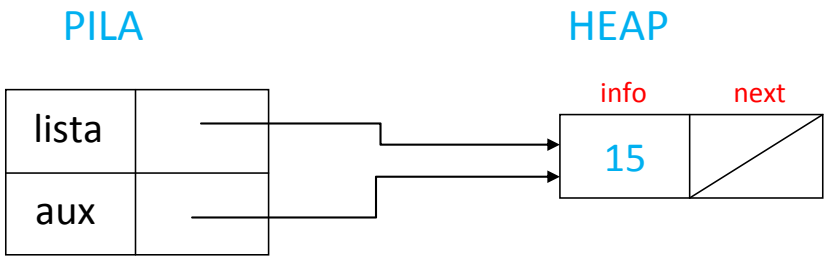


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

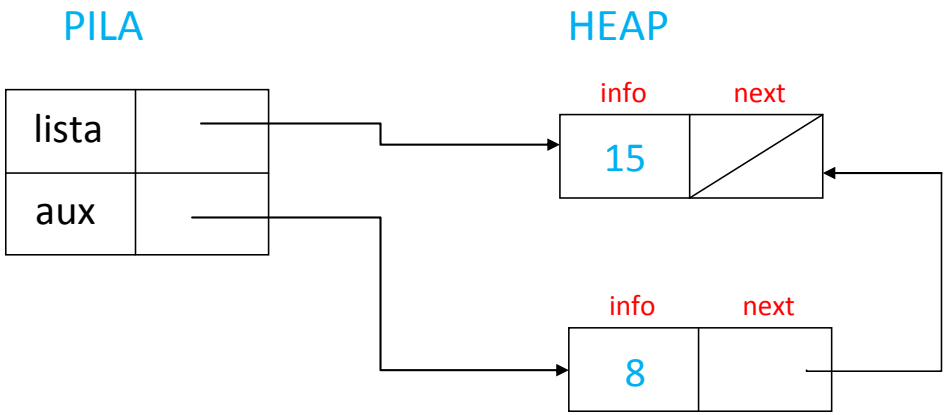


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

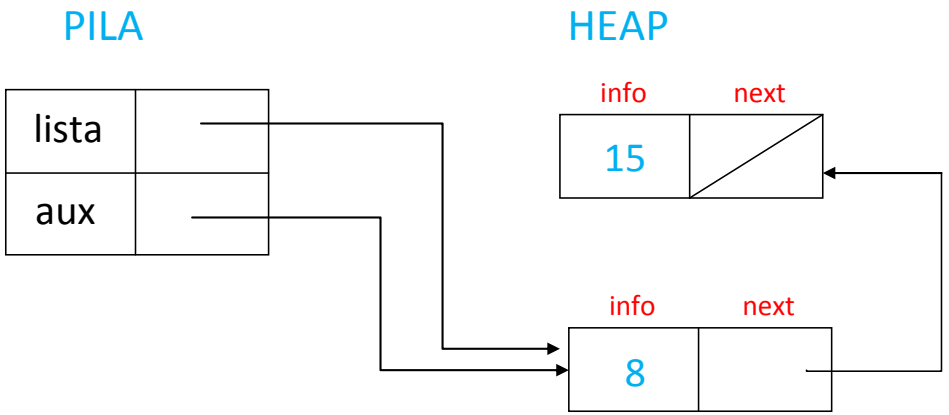


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

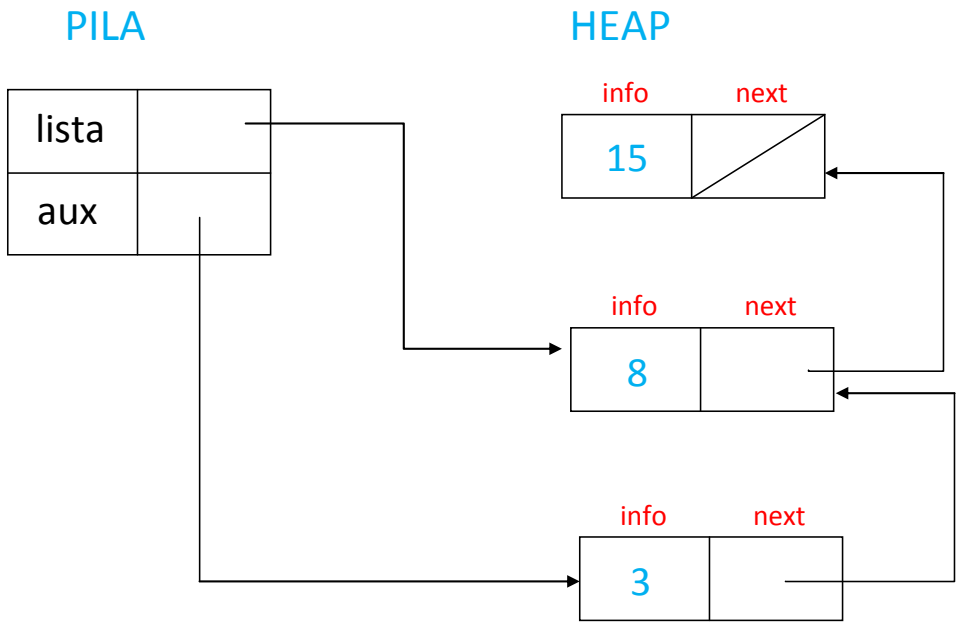


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

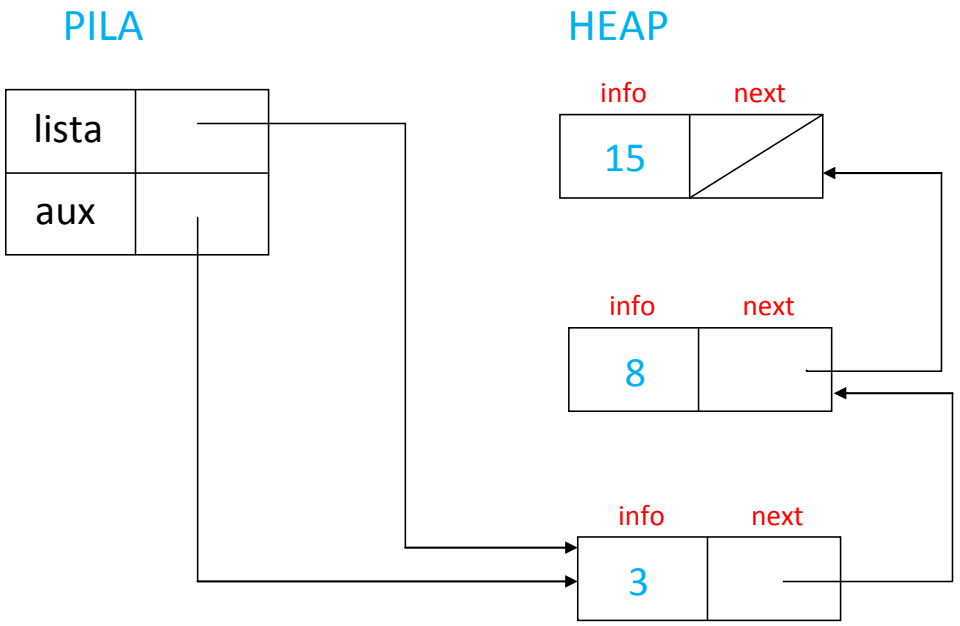


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```



Operazioni sulle liste

- Definiamo una serie di procedure e funzioni per operare sulle liste.
- Usiamo liste di interi per semplicità, ma tutte le operazioni sono realizzabili in modo del tutto analogo su liste di altro tipo (salvo rare eccezioni)
- Facciamo riferimento alle dichiarazioni dei tipi `ElementoLista` e `ListaDiElementi` viste in precedenza

Inizializzazione

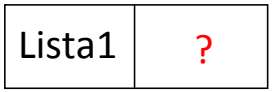
- Definiamo una procedura che inizializza una lista assegnando il valore `NULL` alla variabile testa della lista.
- Tale variabile deve essere modificata e quindi passata per indirizzo.
- Ciò provoca, nell'intestazione della procedura, la presenza di un puntatore a puntatore.

```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

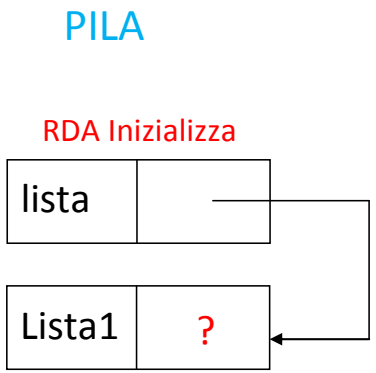
PILA



```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

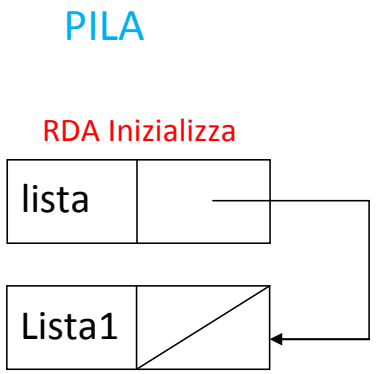
```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```



```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

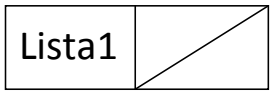


```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

PILA



Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA

Lista1	?

Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA

RDA Inizializza

lista	?

Lista1	?

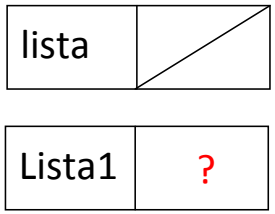
Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA

RDA Inizializza



Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA

Lista1	?

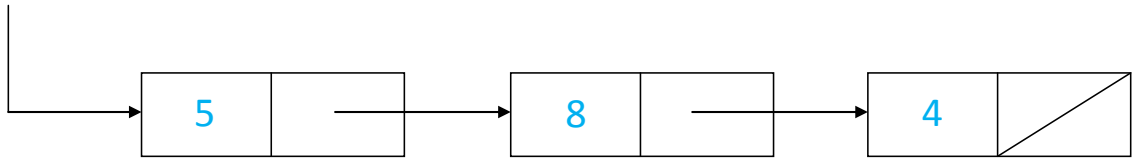
Controllo lista vuota

```
boolean ListaVuota(ListaDiElementi lista)
{
 return (lista==NULL);
}
```

A `lista` viene passato il valore contenuto nella variabile testa di lista e quindi punta al primo elemento della lista considerata.

Stampa degli elementi di una lista

- Data la lista



vogliamo che venga stampato:

5 –> 8 –> 4 –> //

Versione iterativa:

```
void StampaLista(ListaDiElementi lis)
{
   while (lis != NULL)
    {
     printf("%d -->", lis->info);
     lis = lis->next;
    }
printf("//");
}
```

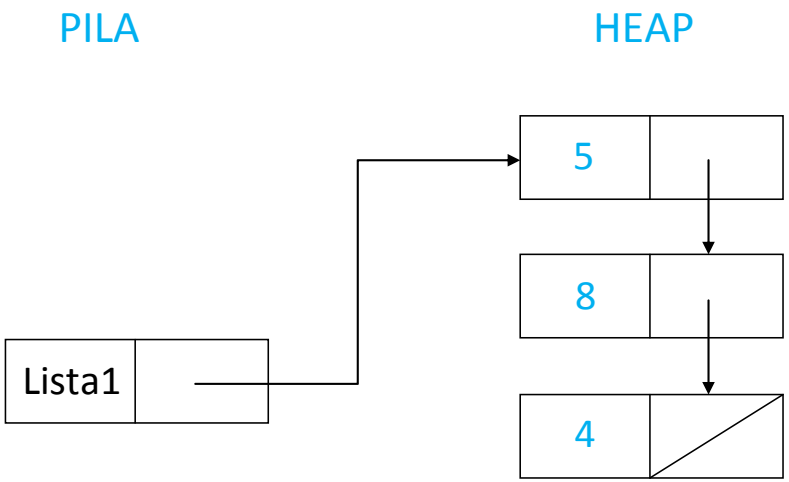
- `lis = lis->next` fa puntare `lis` all'elemento successivo della lista
- `(lis != NULL)` permette di scorrere fino alla fine della lista, di cui solitamente non sappiamo la lunghezza.
- Attenzione: Possiamo usare `lis` per scorrere la lista perché, avendo utilizzato il passaggio per valore, le modifiche a `lis` non si ripercuotono sul parametro attuale.

```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```

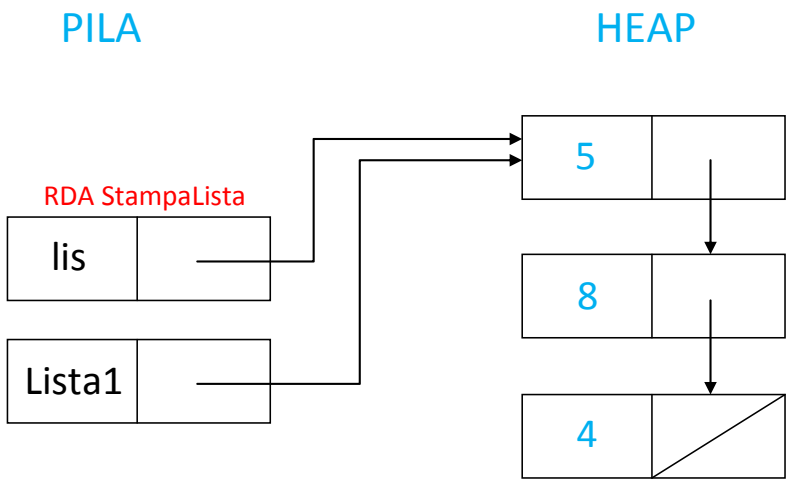
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



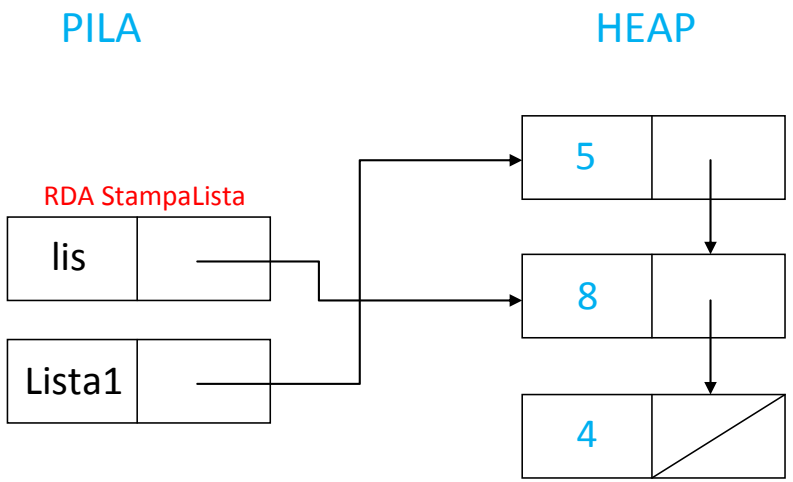
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

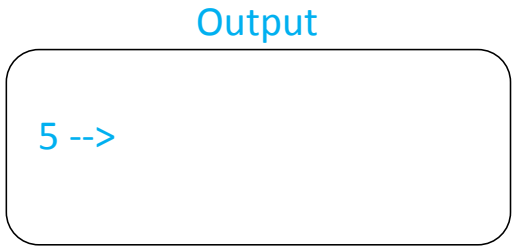
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

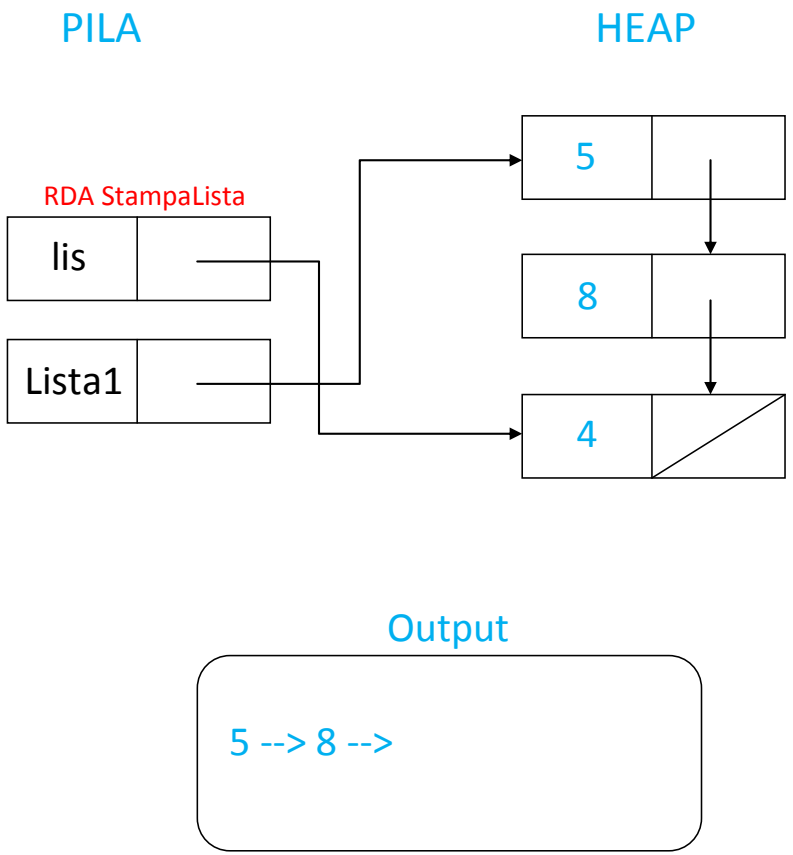
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```





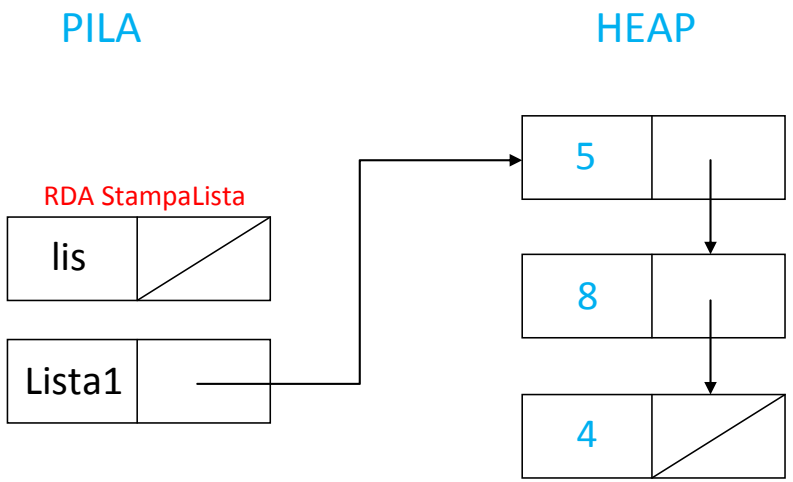
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```

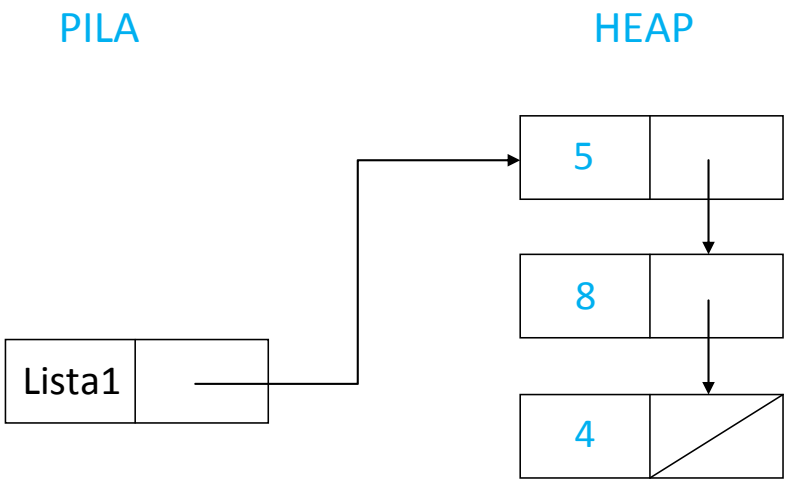


Output

5 --> 8 --> 4 --> //

```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(Lista1);
  ...
}
```



Output

```
5 --> 8 --> 4 --> //
```

Cosa sarebbe successo passando il parametro per indirizzo?

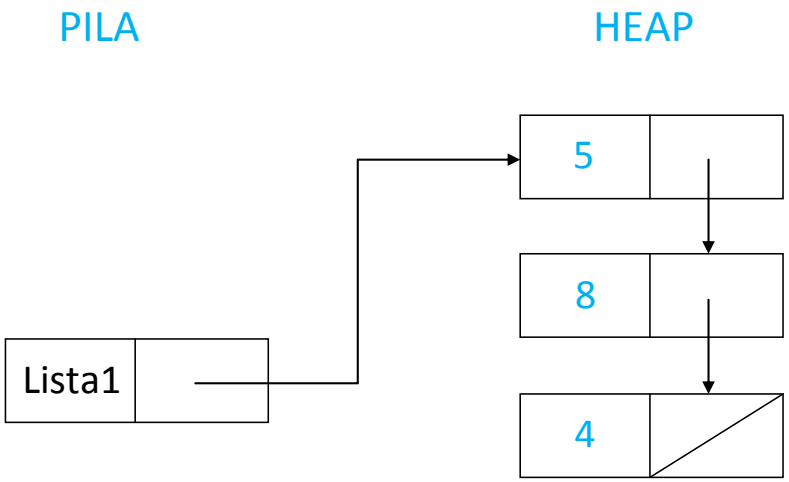
```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

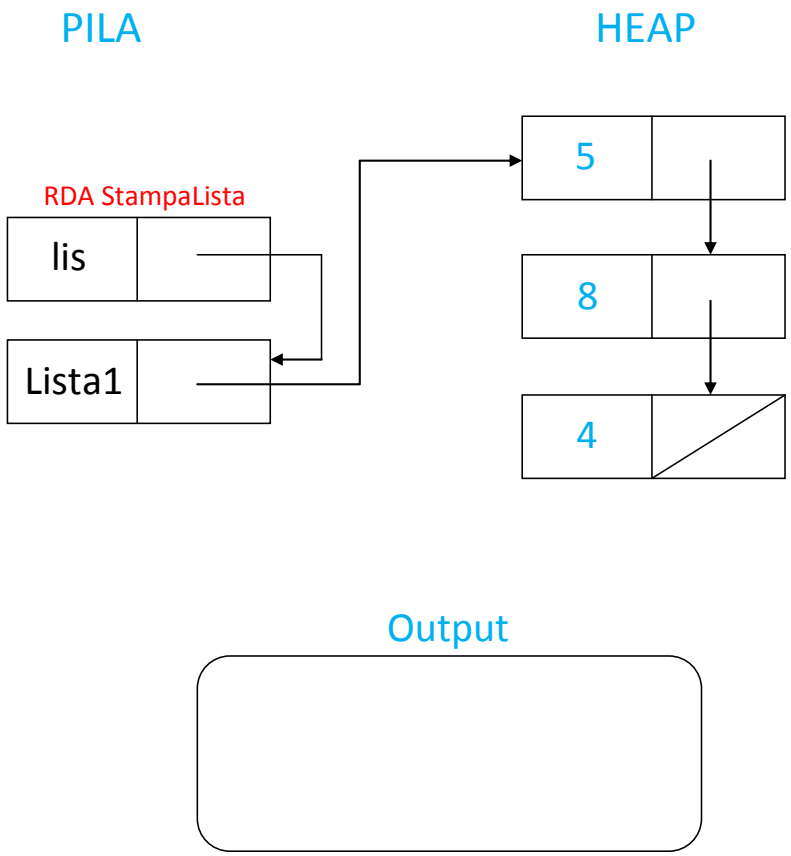
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

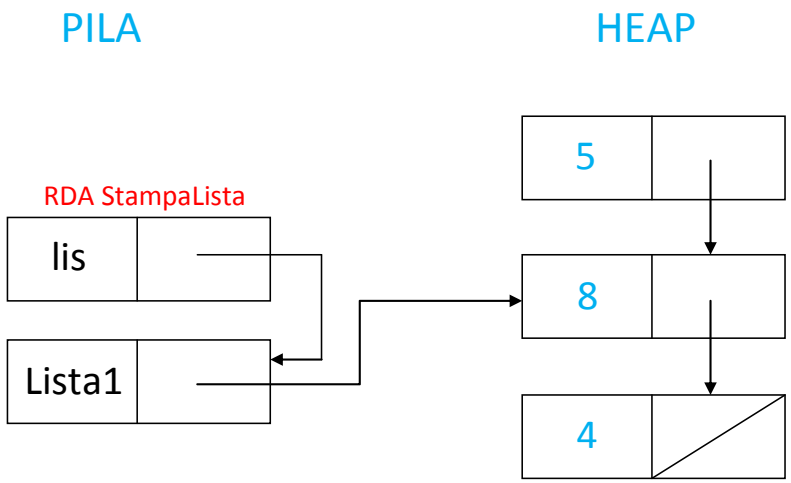
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```

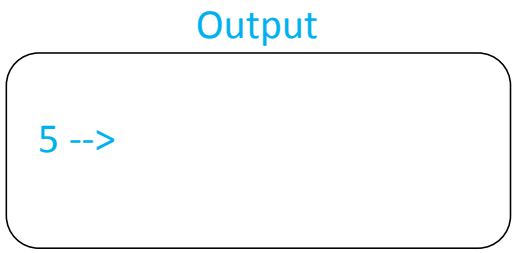


Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```

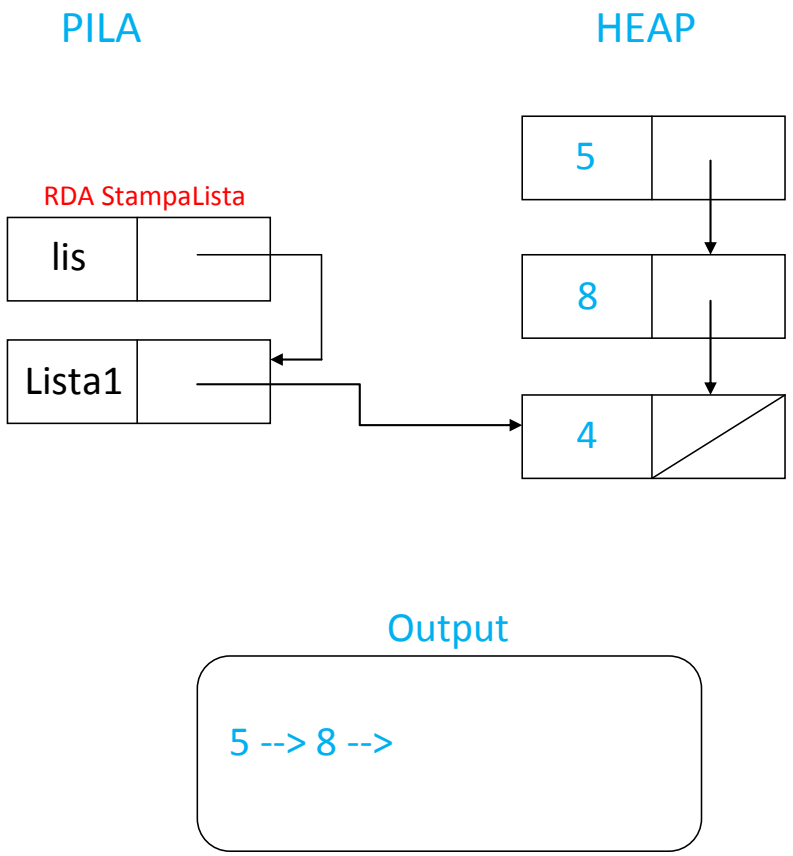




Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

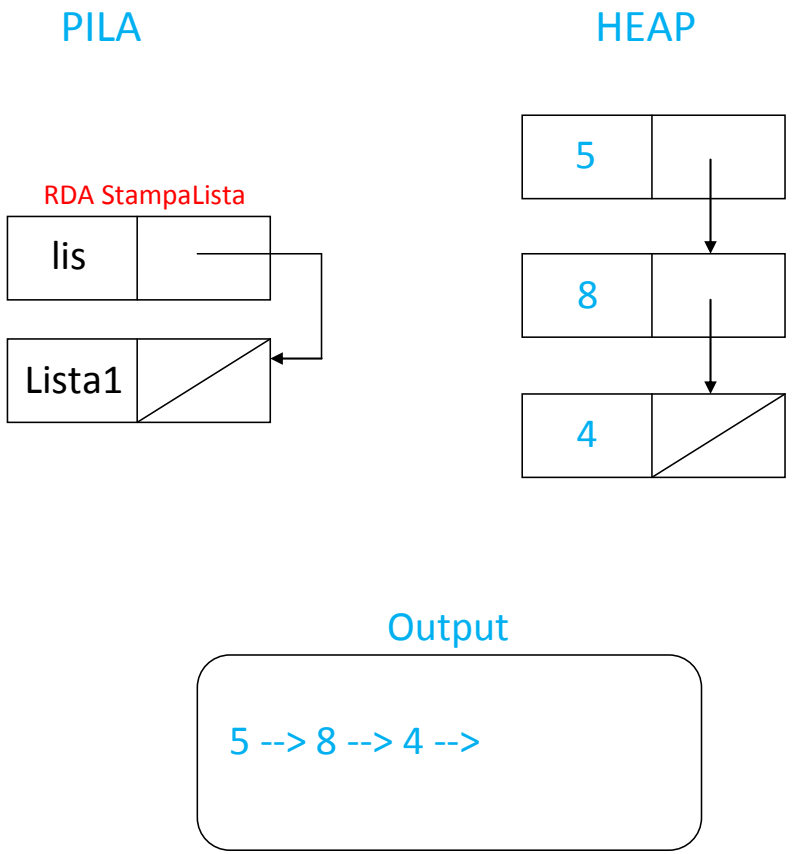
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

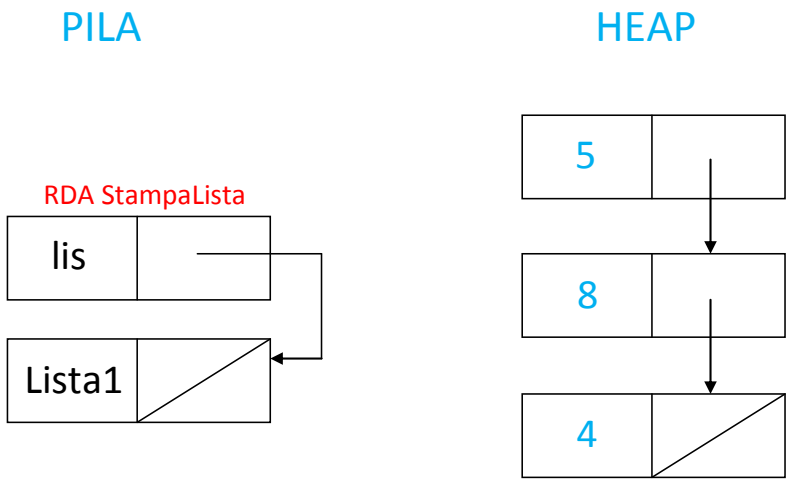
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



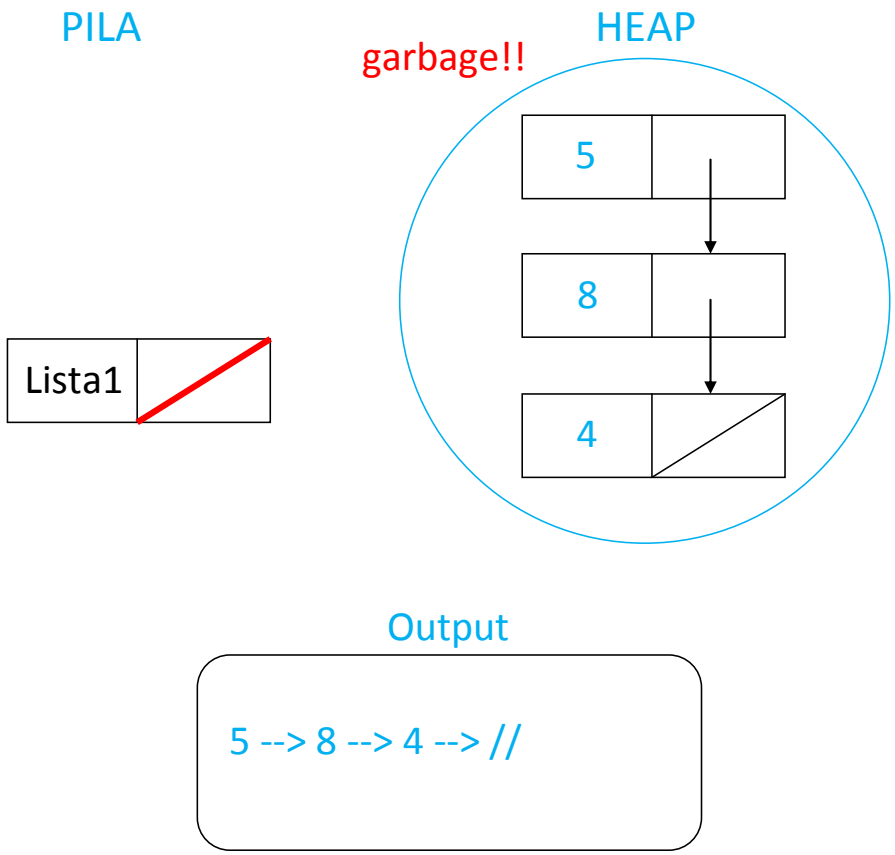
Output

5 --> 8 --> 4 --> //

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaLista(&Lista1);
  ...
}
```



Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

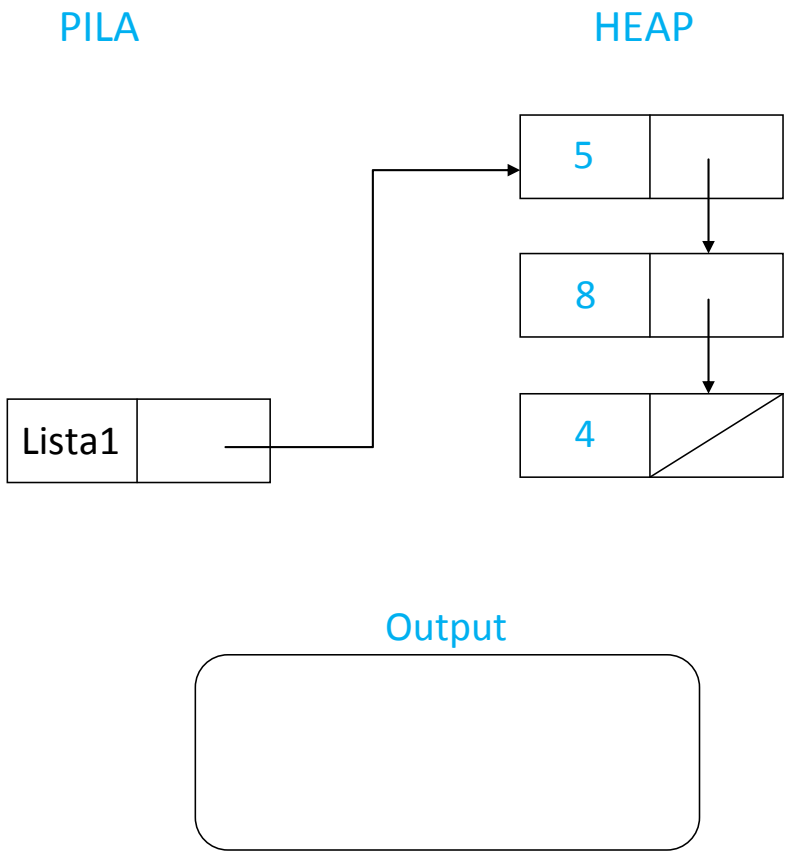
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```

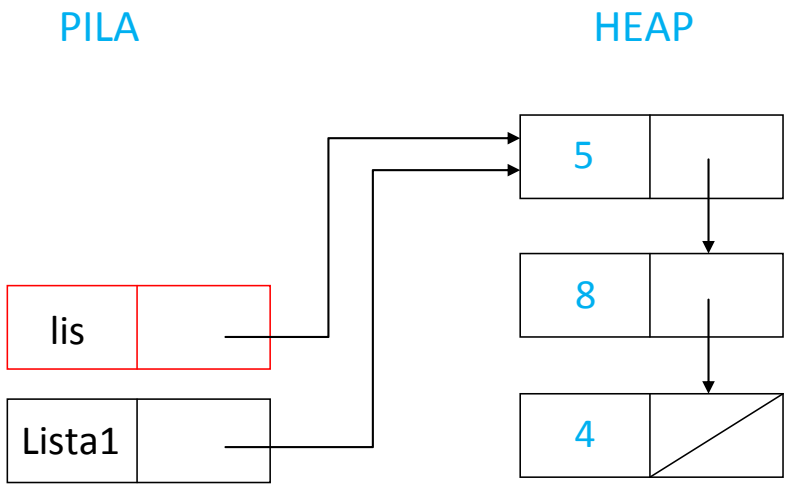


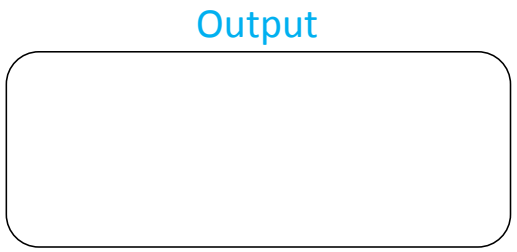
Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



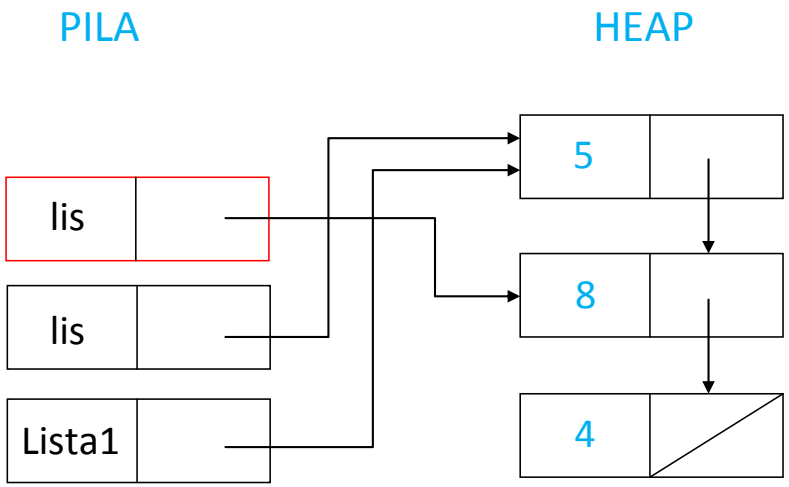


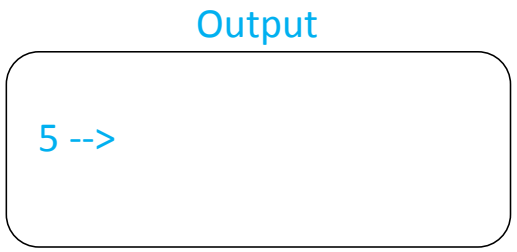
Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



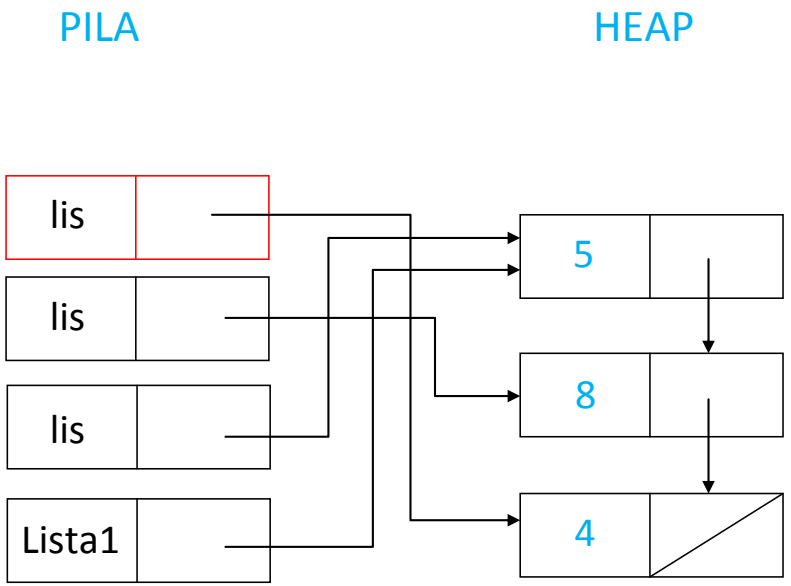


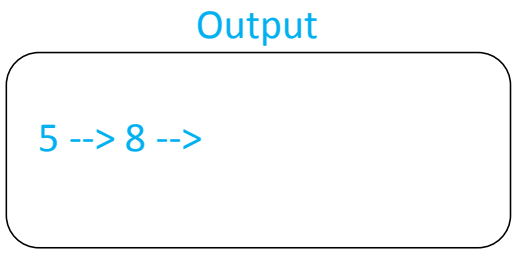
Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



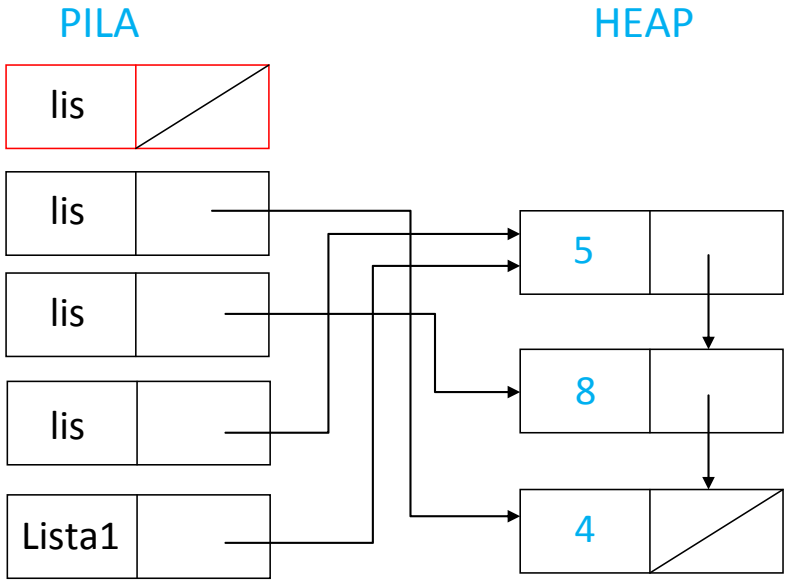


Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Output

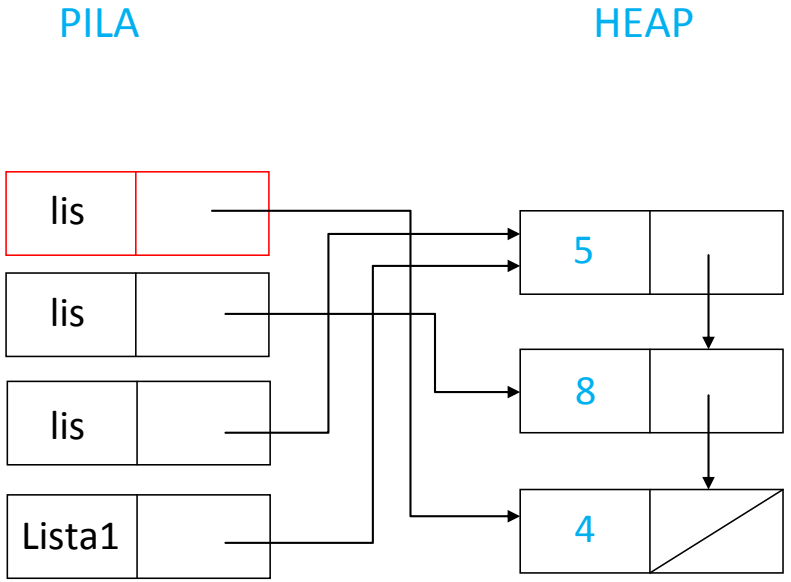
5 --> 8 --> 4 -->

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Output

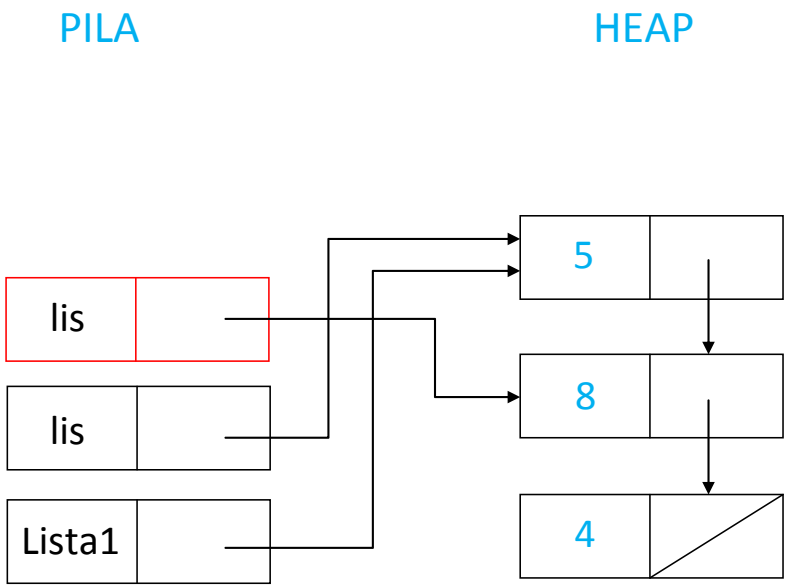
5 --> 8 --> 4 --> //

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Output

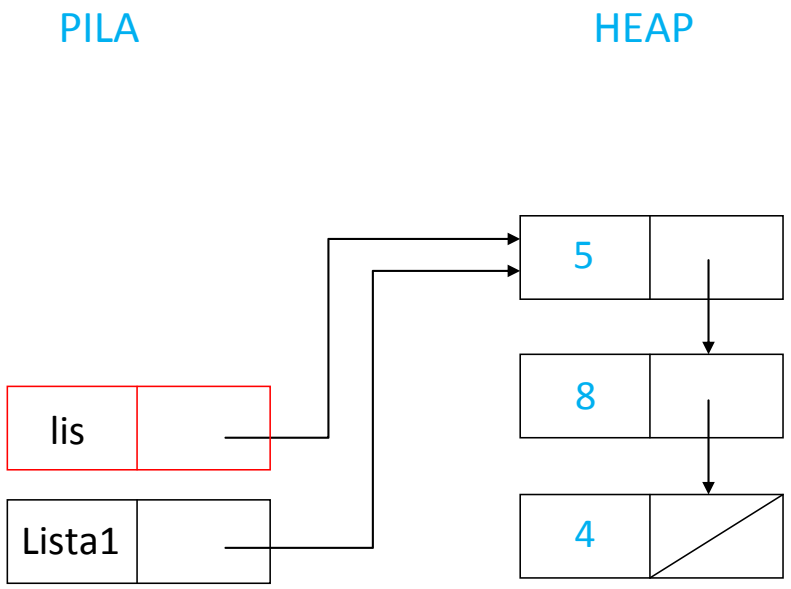
5 --> 8 --> 4 --> //

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Output

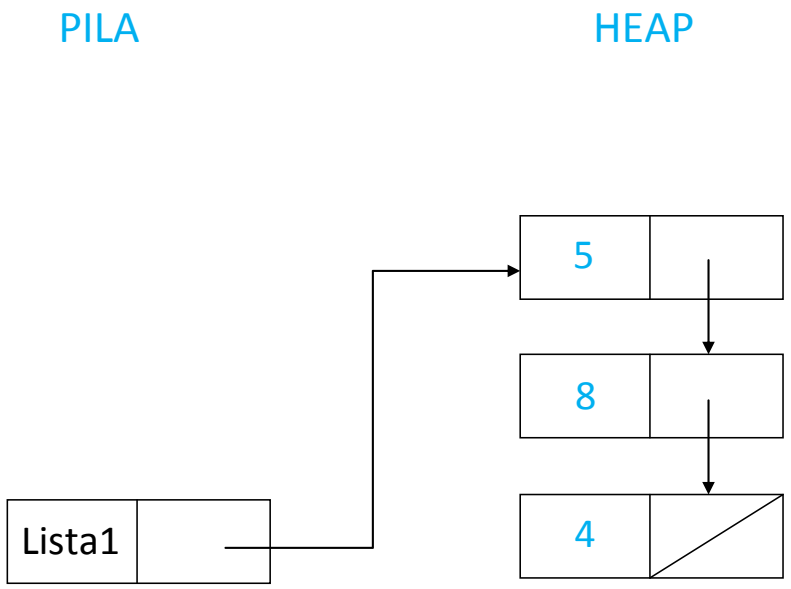
5 --> 8 --> 4 --> //

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

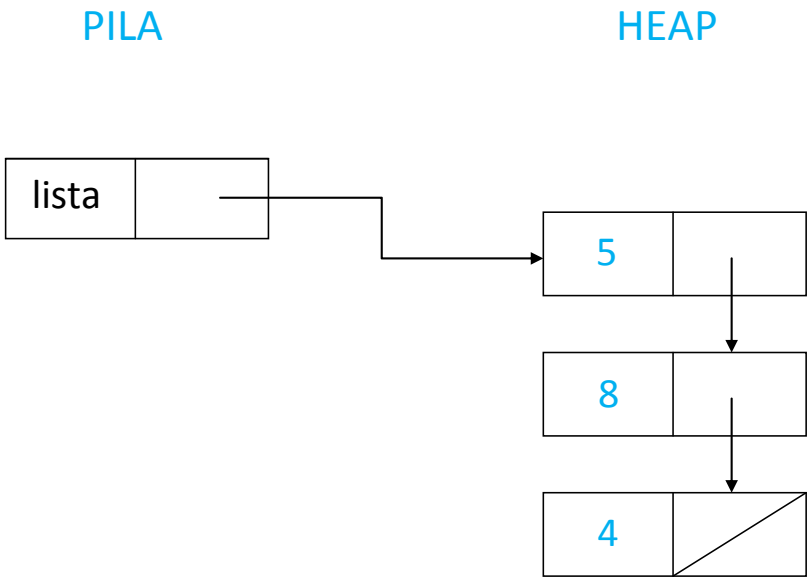
main()
{
  ListaDiElementi Lista1;
  ...
  /* costruzione lista 5 --> 8 --> 4  */
  ...
  StampaListaRic(Lista1);
  ...
}
```



Output

5 --> 8 --> 4 --> //

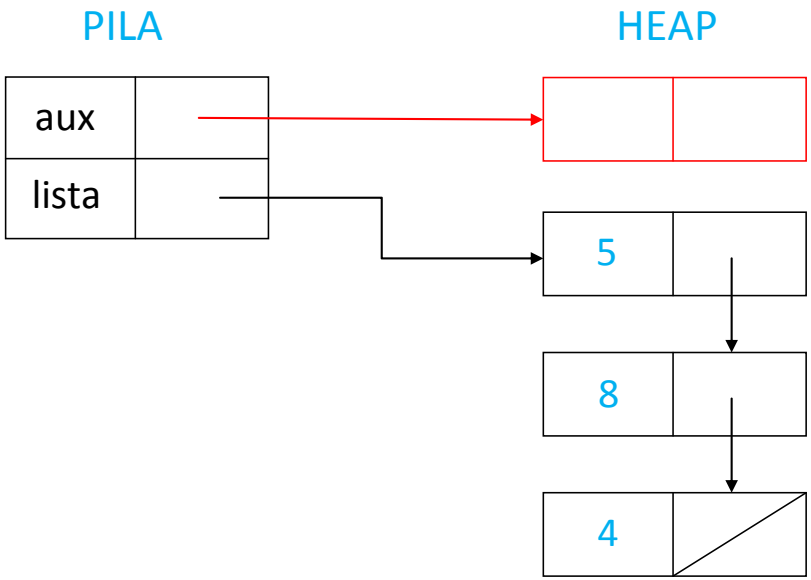
Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura

   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

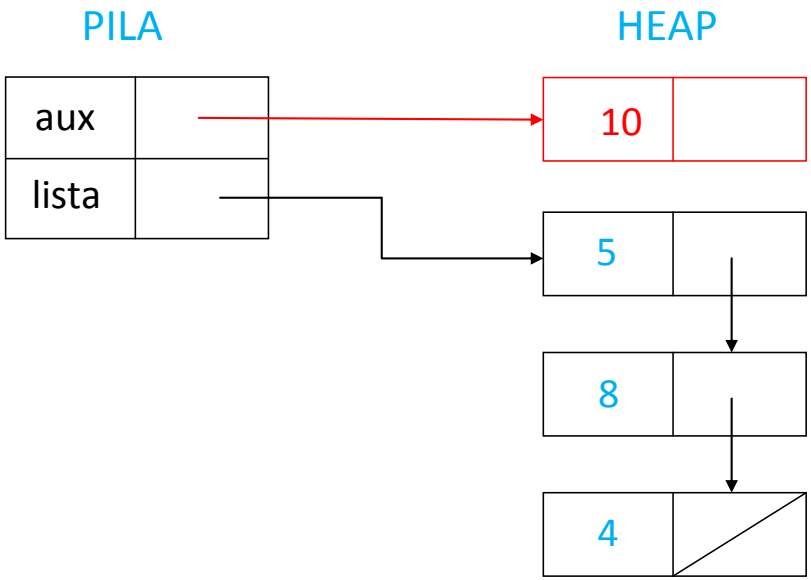
Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura

   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

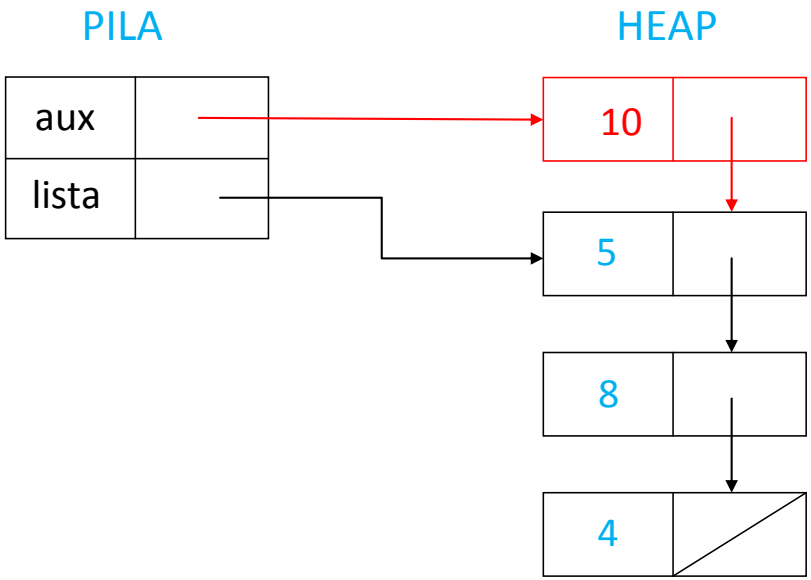
Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura

   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

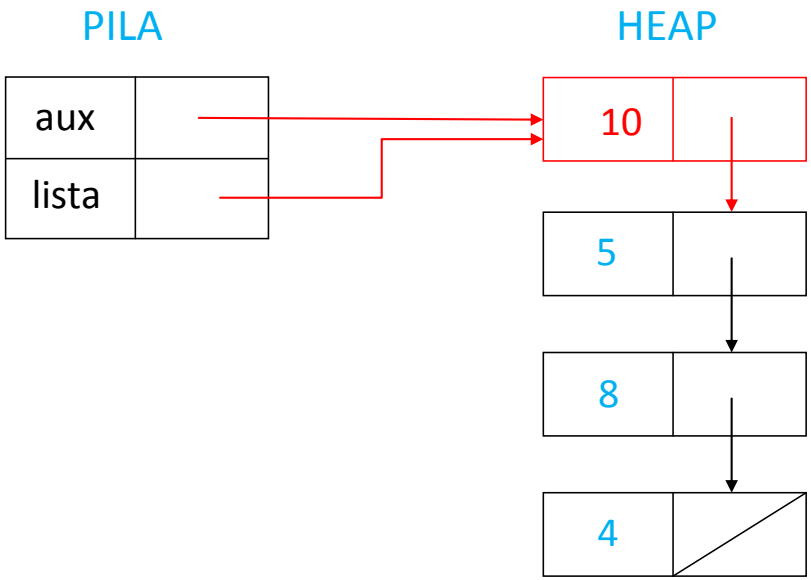
Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura

   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura

   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

```
void InserisciTestaLista(ListaDiElementi *lista, int elem)
{
  ListaDiElementi aux;

  aux = malloc(sizeof(ElementoLista));
  aux->info = elem;
  aux->next = *lista;
  *lista = aux;
}
```

- il primo parametro è la lista da modificare (passata per indirizzo)
- il secondo parametro è l'elemento da inserire (passato per indirizzo)

Esercizio

Impostare una chiamata alla procedura e tracciare l'evoluzione di pila e heap

```
void InserisciTestaLista(ListaDiElementi *lista, TipoElemLista elem)
{
  ListaDiElementi aux;

  aux = malloc(sizeof(ElementoLista));
  aux->info = elem;
  aux->next = *lista;
  *lista = aux;
}
```

- il primo parametro è la lista da modificare (passata per indirizzo)
- il secondo parametro è l'elemento da inserire (passato per indirizzo)

Esercizio

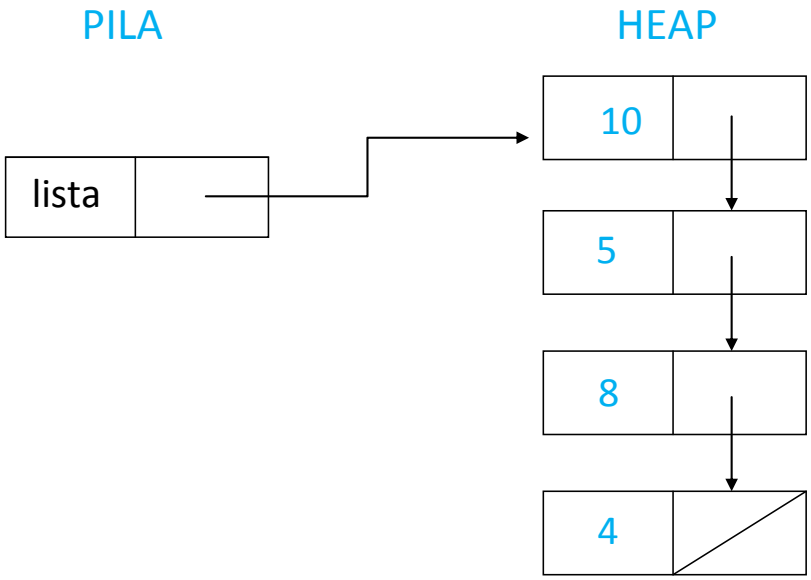
Impostare una chiamata alla procedura e tracciare l'evoluzione di pila e heap

- il primo parametro è la lista da modificare (passata per indirizzo)
- il secondo parametro è l'elemento da inserire (passato per indirizzo)
  - Attenzione: nel caso di liste di tipo `TipoElemLista` la procedura può essere generalizzata se su tale tipo è definito l'assegnamento

Esercizio

Impostare una chiamata alla procedura e tracciare l'evoluzione di pila e heap

Cancellazione del primo elemento



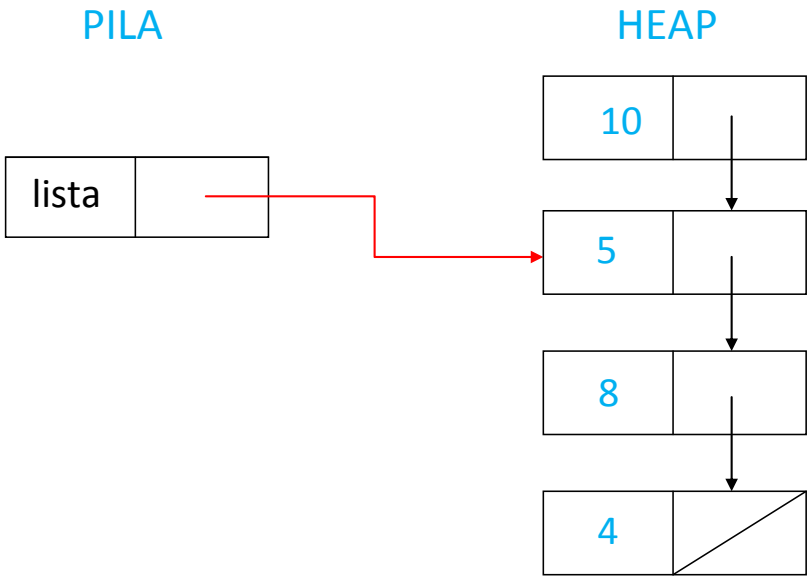
- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento

  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare

  $\Longrightarrow$ è necessario un puntatore ausiliario

Cancellazione del primo elemento



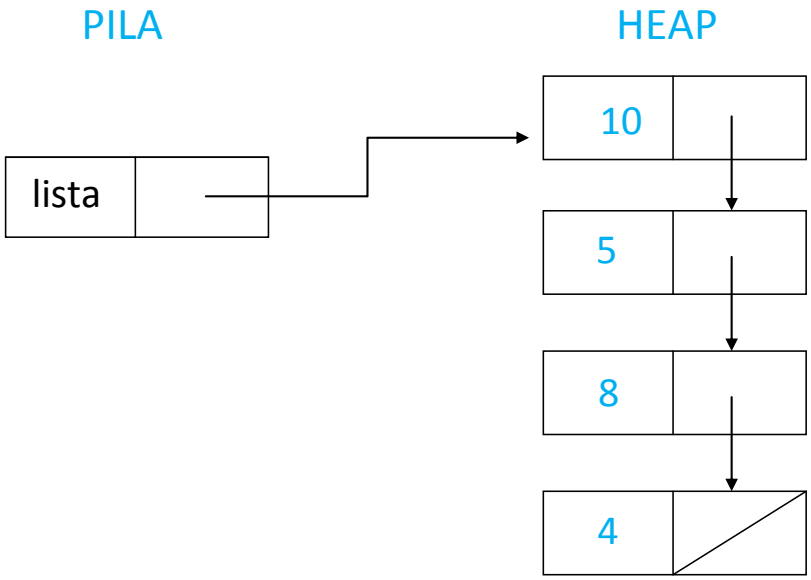
- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento

  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare

  $\Longrightarrow$ è necessario un puntatore ausiliario

Cancellazione del primo elemento



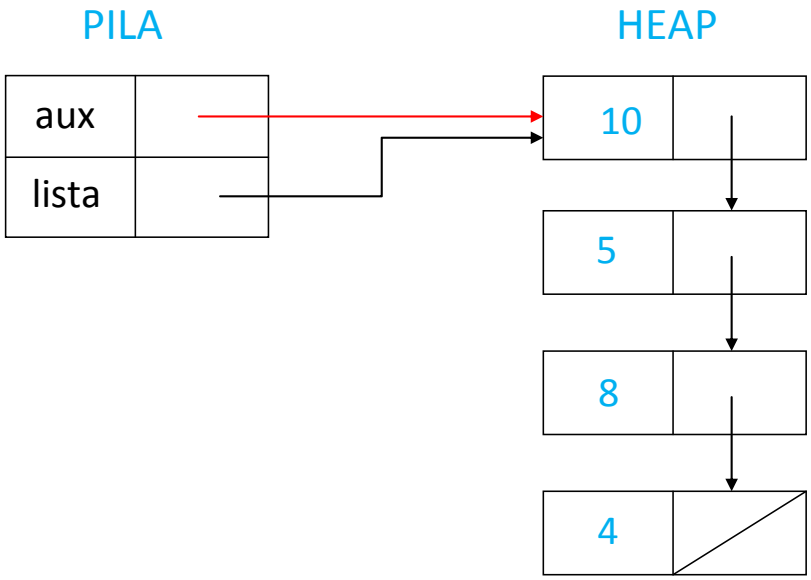
- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento

  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare

  $\Longrightarrow$ è necessario un puntatore ausiliario

Cancellazione del primo elemento



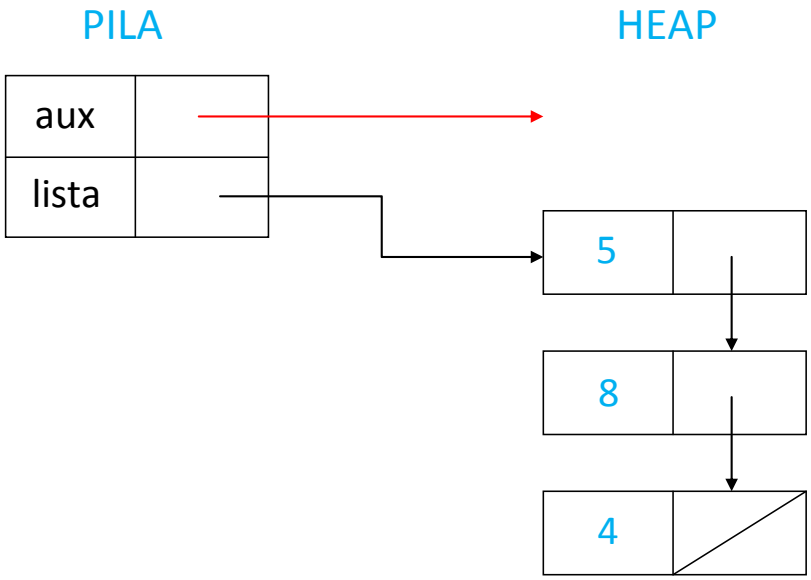
- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento

  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare

  $\Longrightarrow$ è necessario un puntatore ausiliario

Cancellazione del primo elemento



- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento

  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare

  $\Longrightarrow$ è necessario un puntatore ausiliario

```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}
```

Cancellazione di tutta una lista

```
void CancellaLista(ListaDiElementi *lista)
{
  ListaDiElementi aux;

  while (*lista != NULL) {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
  }
}
```

- Osserviamo che il corpo del ciclo corrisponde alle azioni della procedura `CancellaPrimo`. Possiamo allora scrivere:

```
void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}
```

- Si noti il parametro attuale della chiamata a `CancellaPrimo`, che è `lista` (di tipo `ListaDiElementi *`) e non `&lista` (v. animazione)

```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

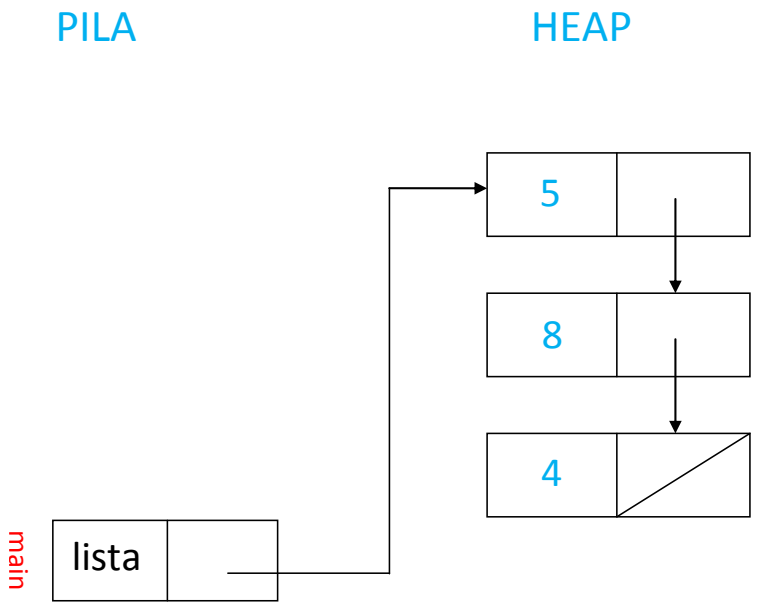
void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

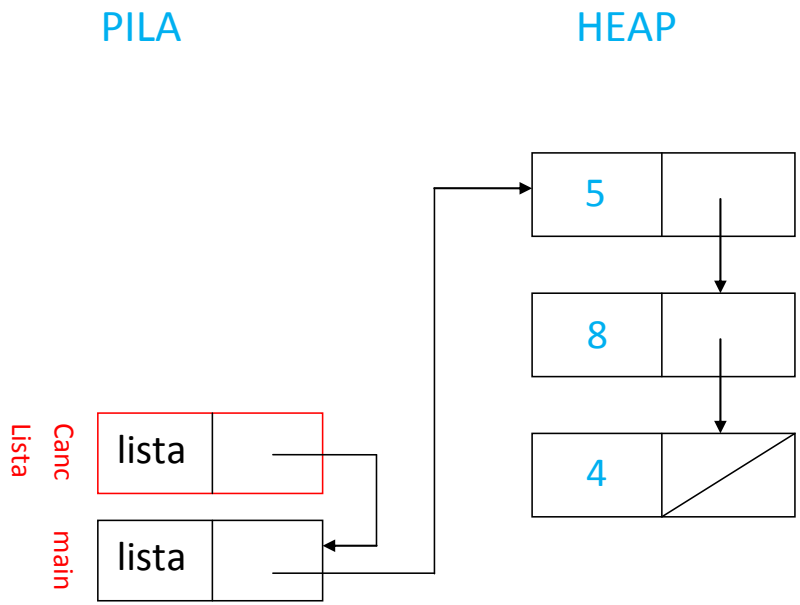
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

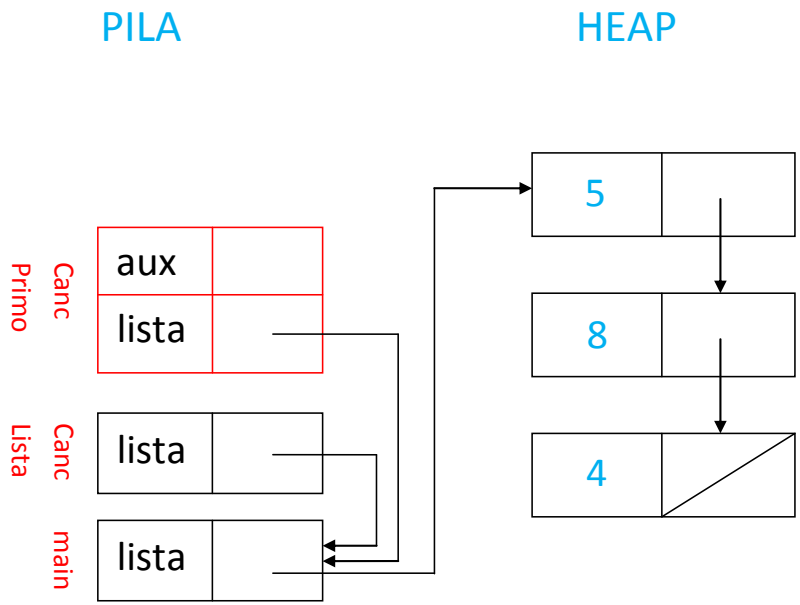
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

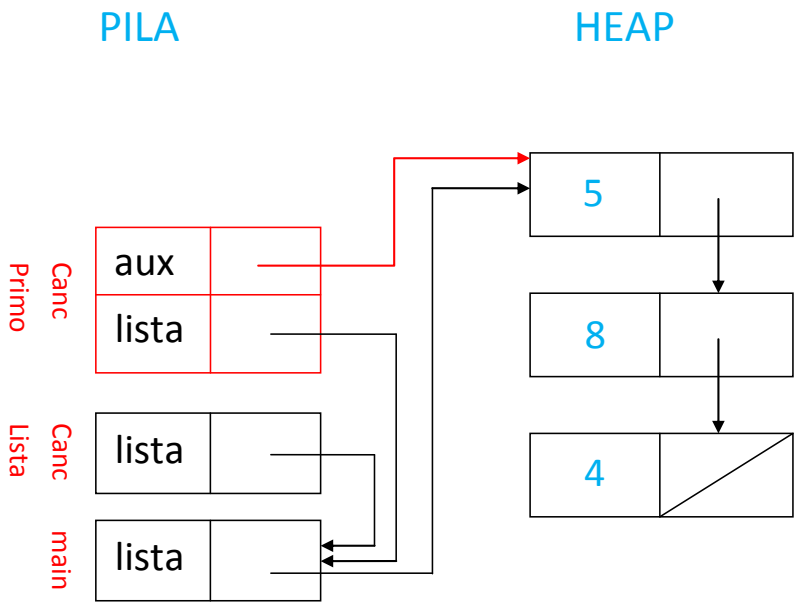
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

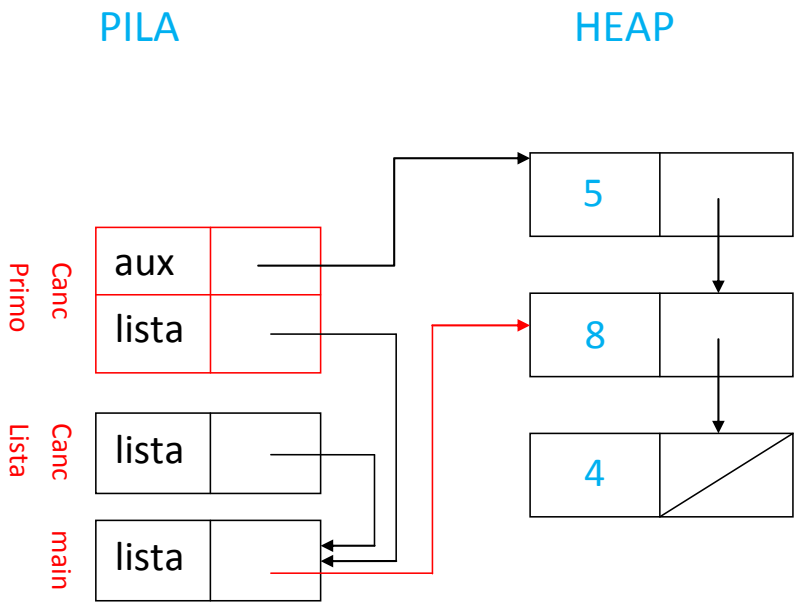
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

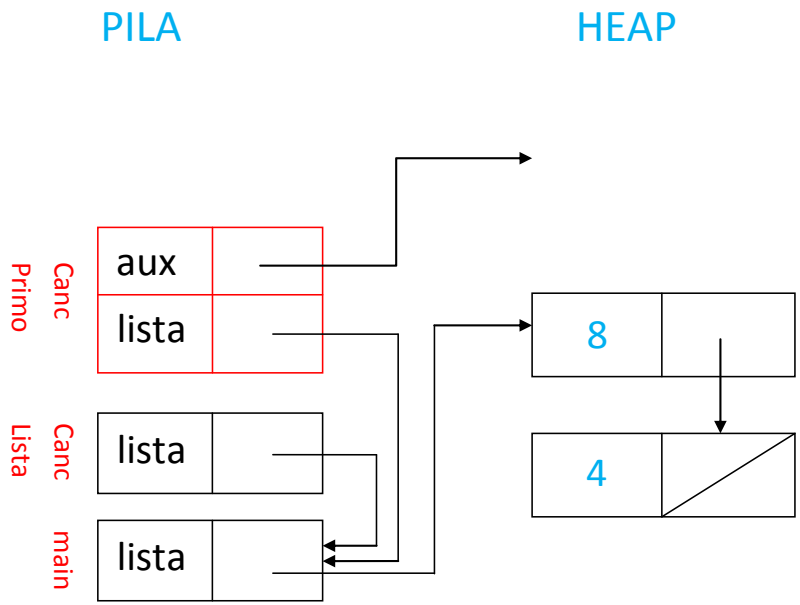
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

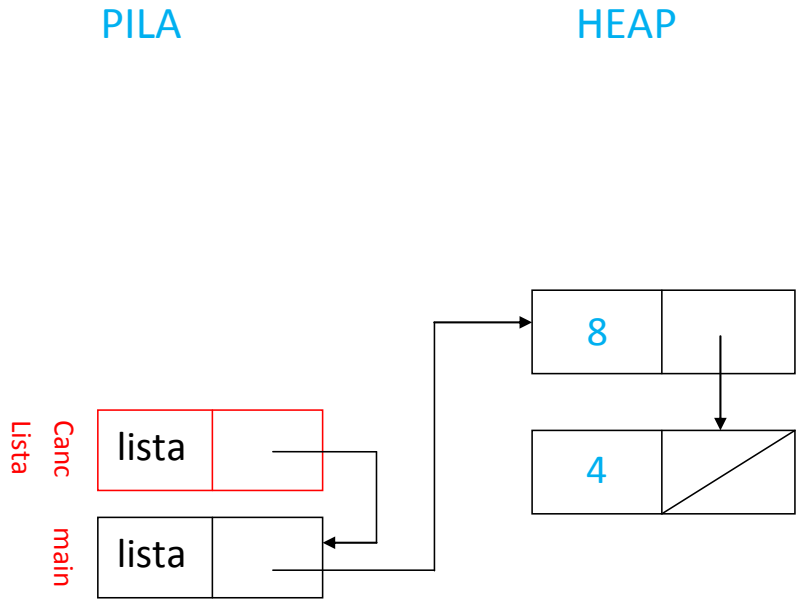
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

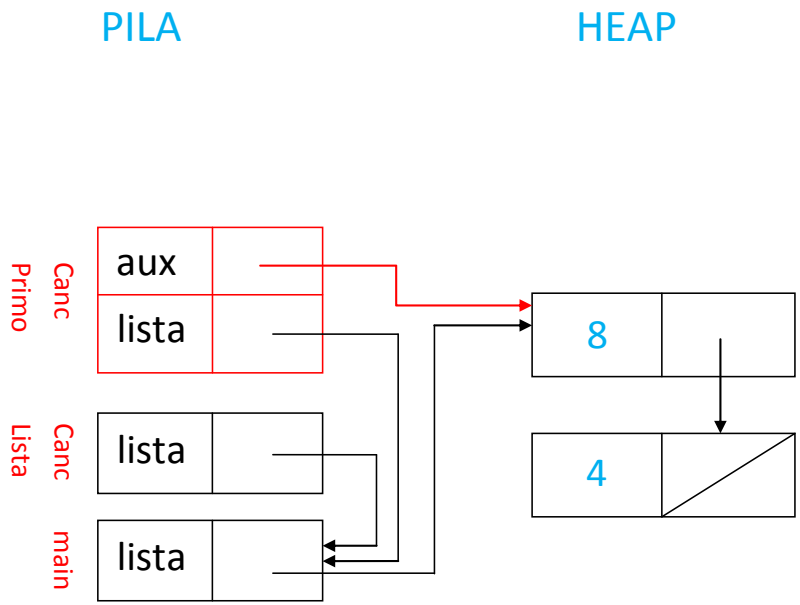
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

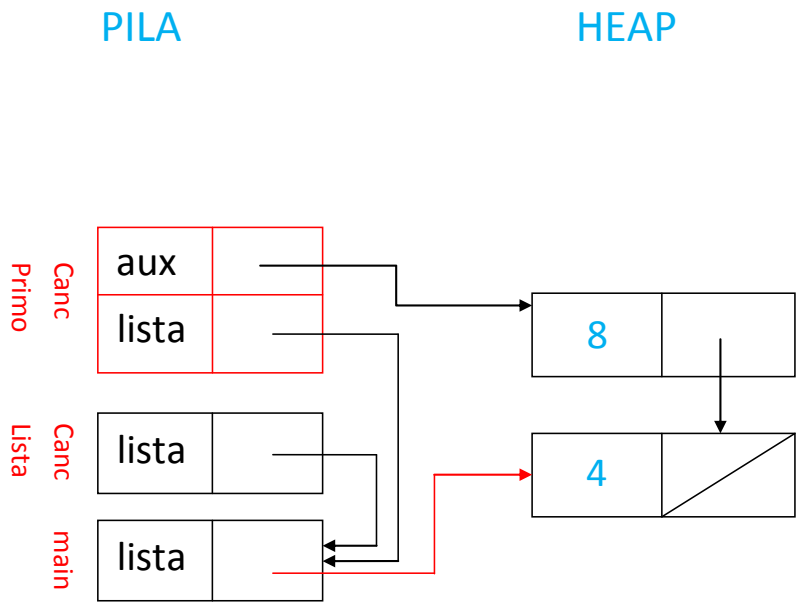
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

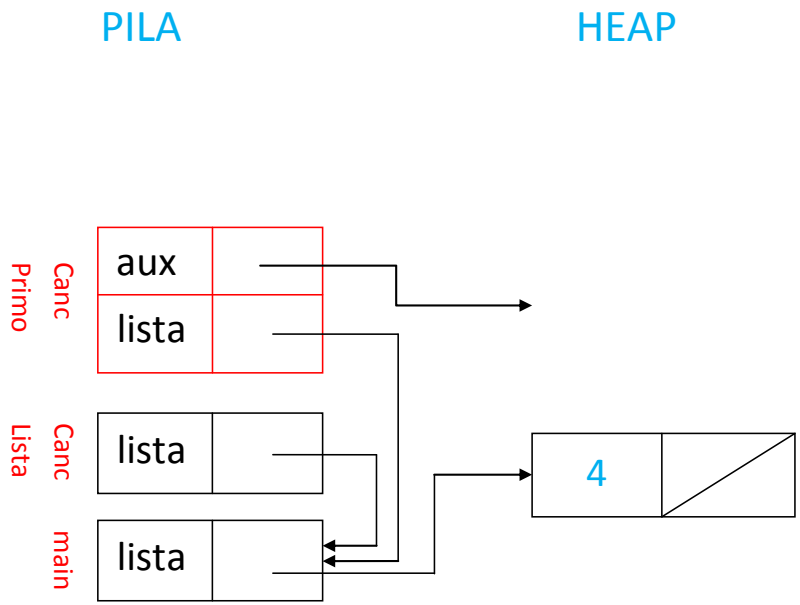
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

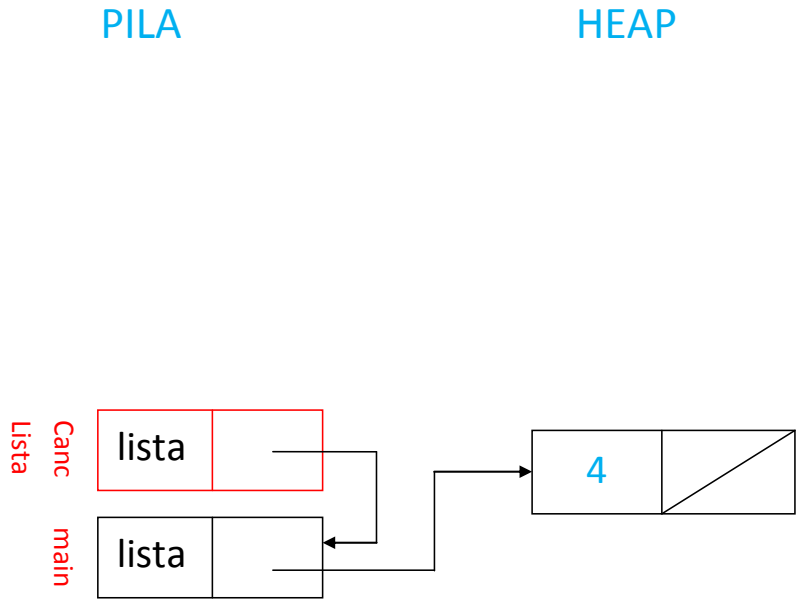
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

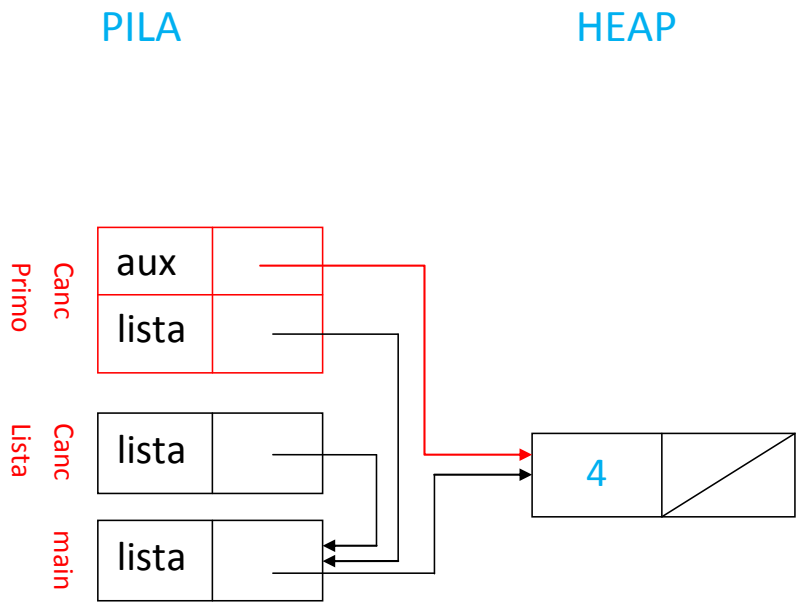
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

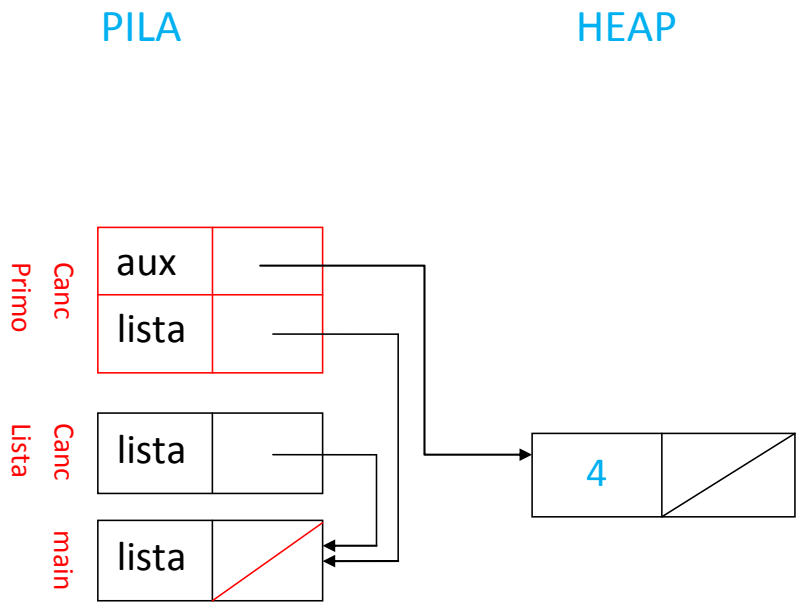
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

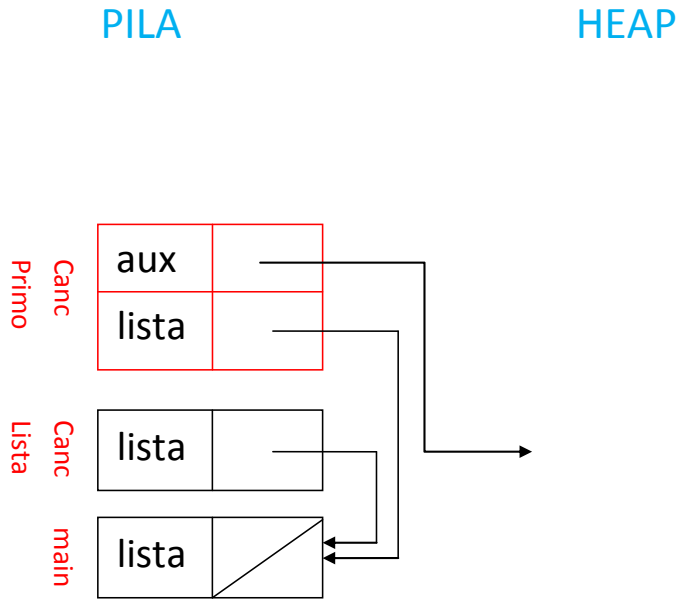
main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

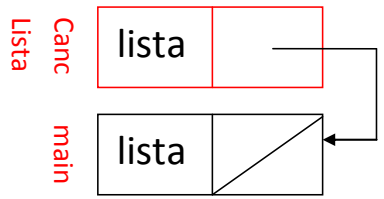


```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

PILA HEAP



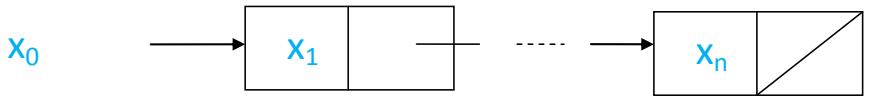
```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

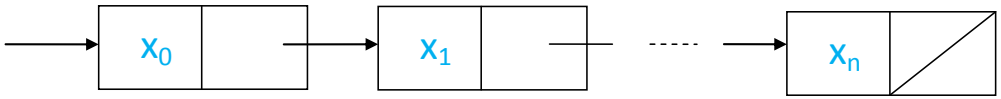
PILA HEAP

Visione ricorsiva delle liste



- Una lista di elementi è una struttura dati ricorsiva per sua natura
  1. data una lista L di elementi x_1, ..., x_n
  2. dato un ulteriore elemento x_0
  3. anche la concatenazione di x_0 e L è una lista
- Si noti che in 1. L può anche essere la lista vuota

Visione ricorsiva delle liste



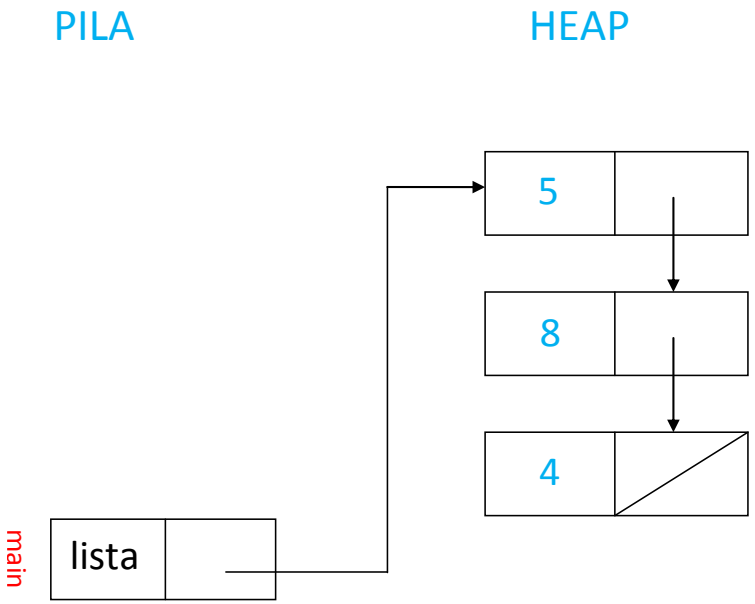
- Una lista di elementi è una struttura dati ricorsiva per sua natura
  1. data una lista L di elementi x_1, ..., x_n
  2. dato un ulteriore elemento x_0
  3. anche la concatenazione di x_0 e L è una lista
- Si noti che in 1. L può anche essere la lista vuota

Cancellazione lista: versione ricorsiva

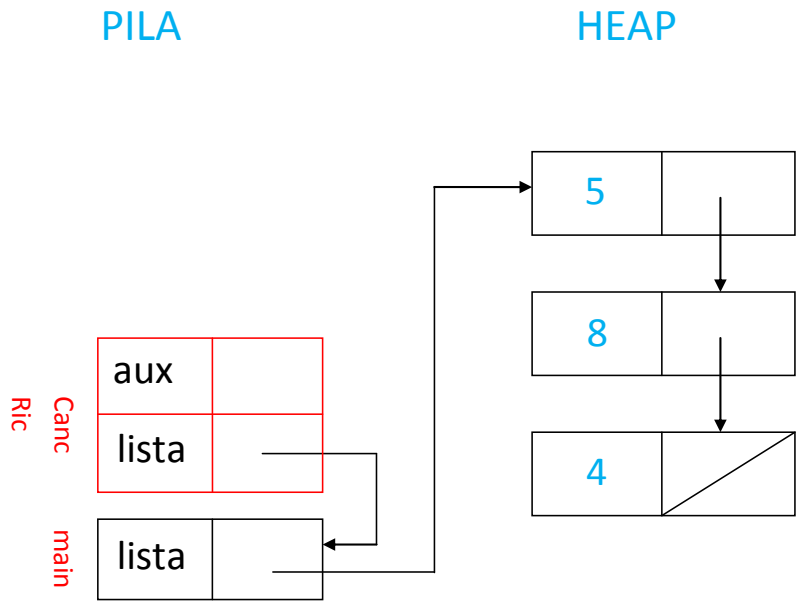
- Sfruttiamo la visione ricorsiva della struttura dati lista per realizzare la cancellazione in modo ricorsivo
  1. la cancellazione della lista vuota non richiede alcuna azione
  2. la cancellazione della lista ottenuta come concatenazione dell'elemento x e della lista L richiede l'eliminazione di x e la cancellazione di L
- la traduzione in C è immediata

```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

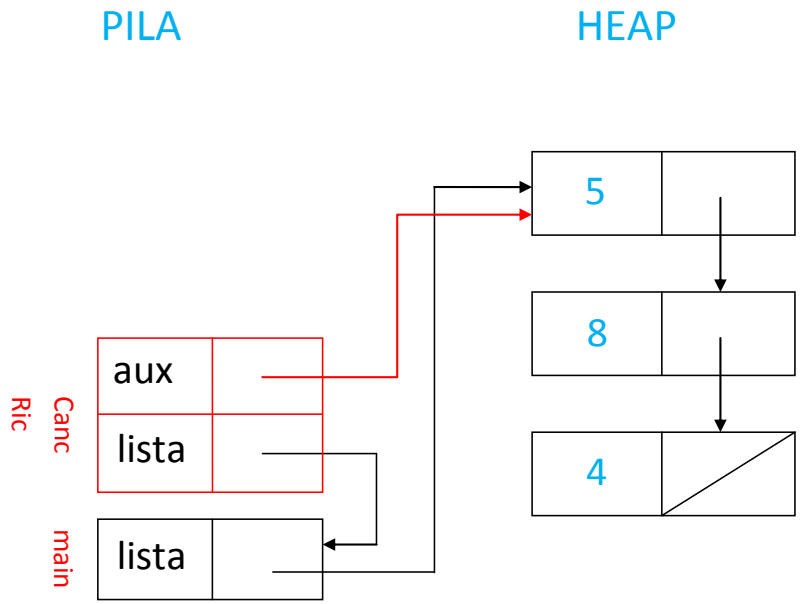
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



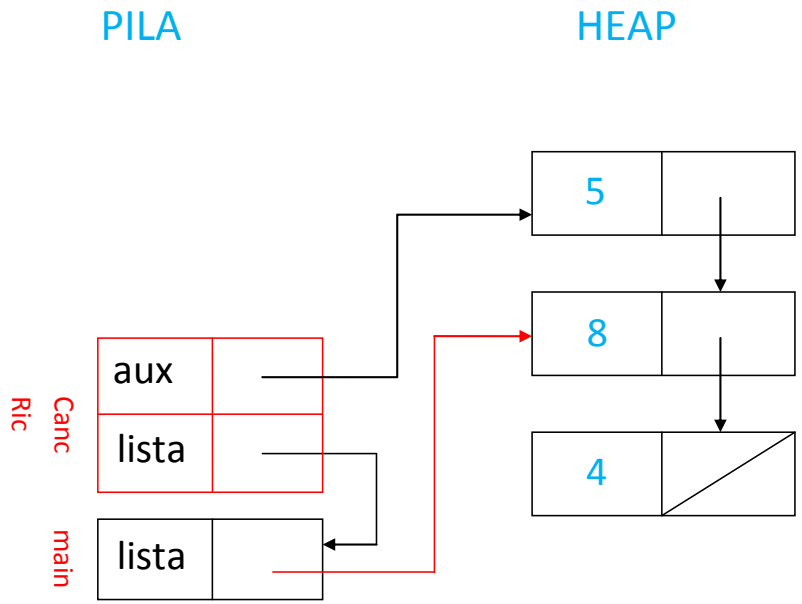
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



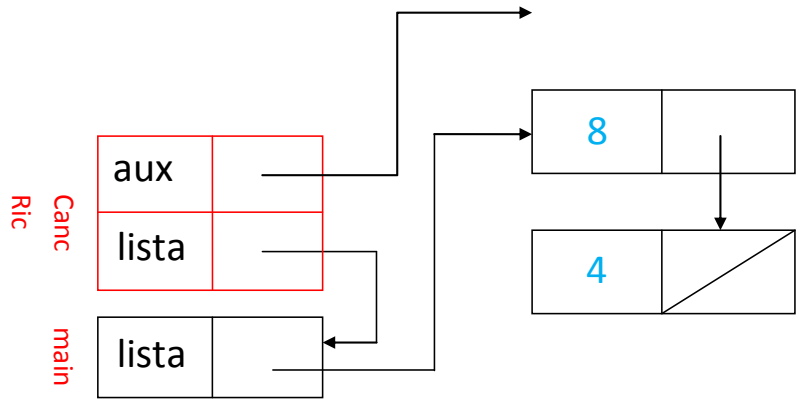
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



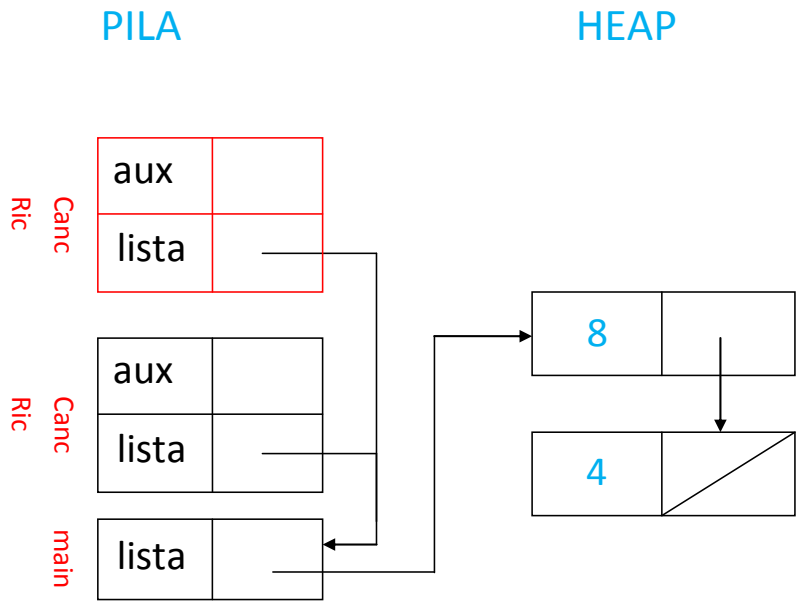
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



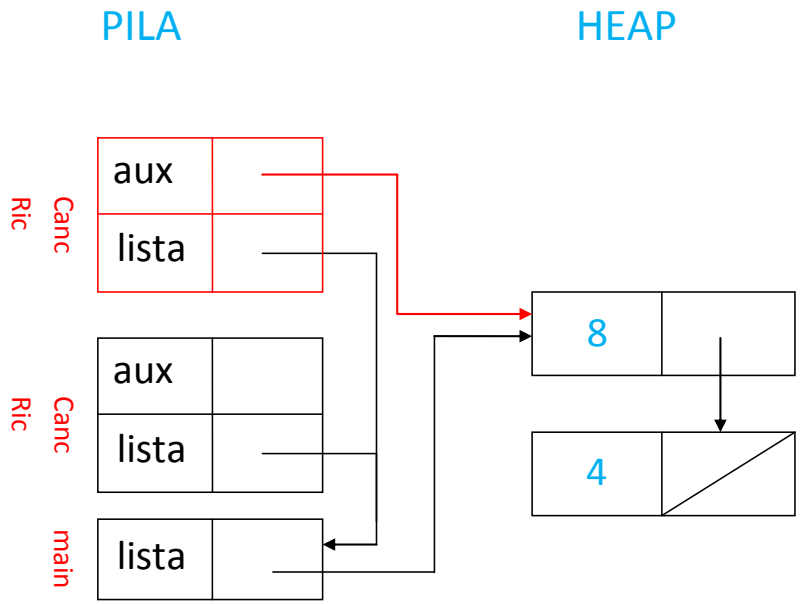
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



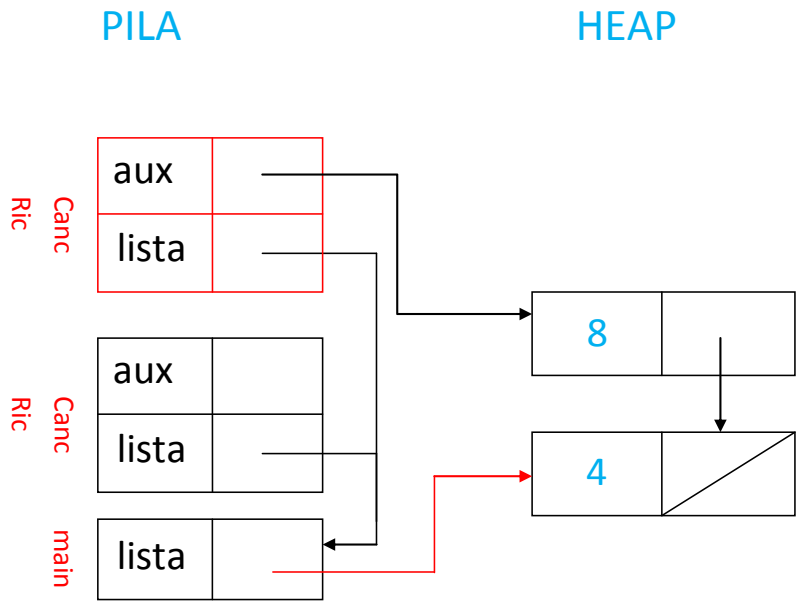
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



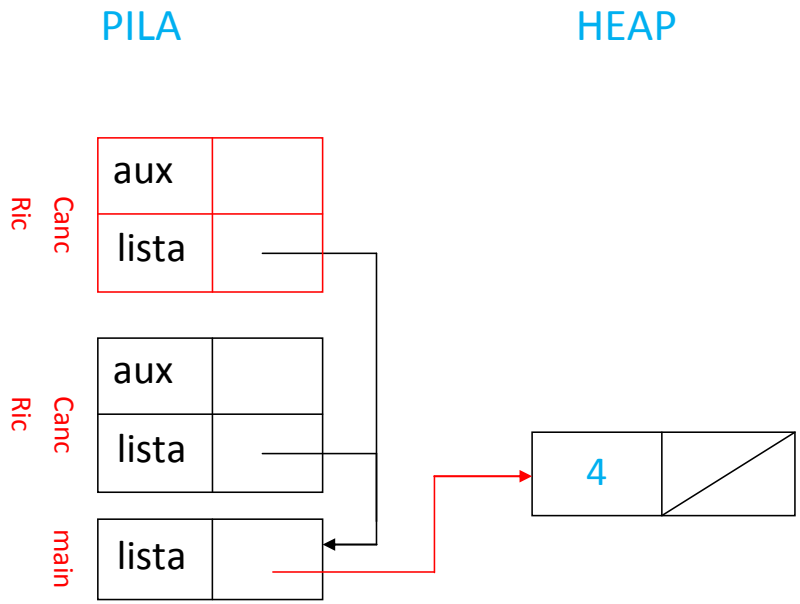
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

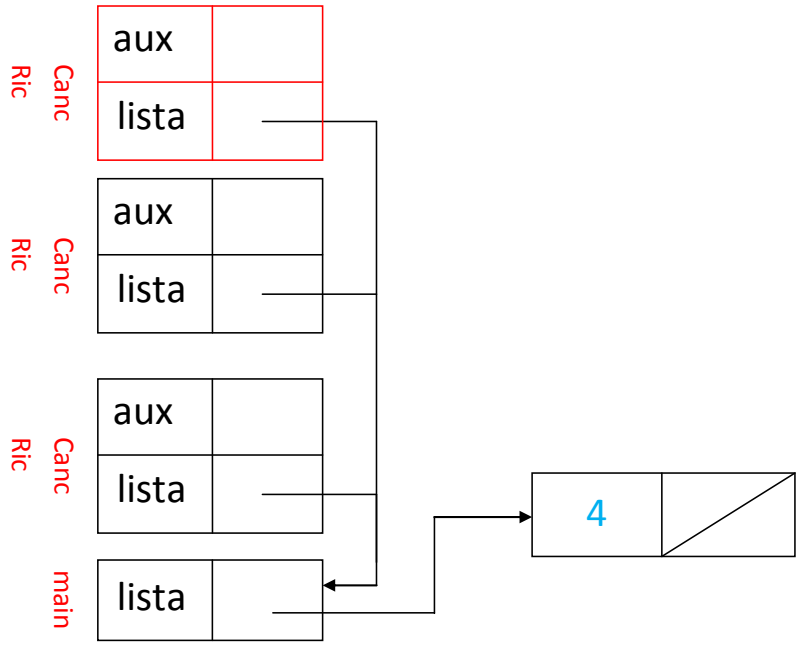


```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



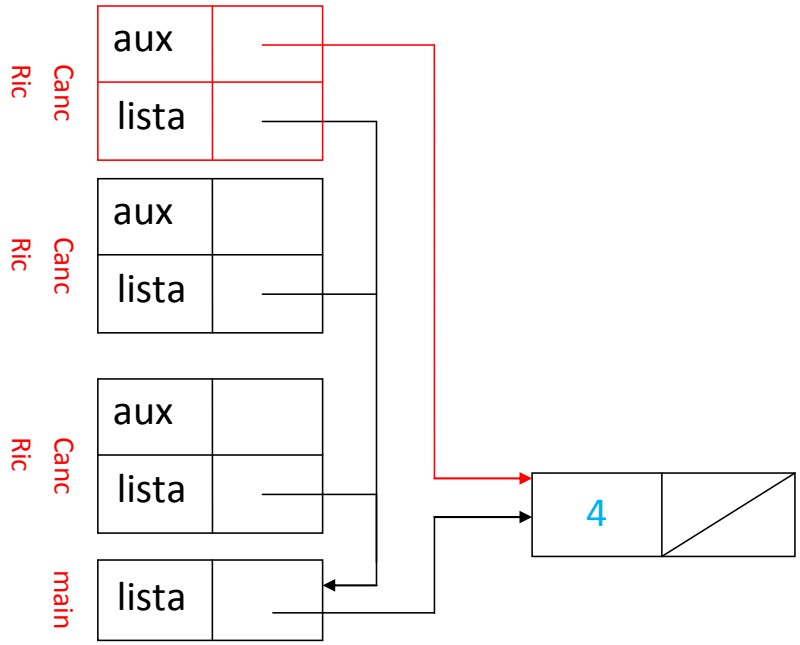
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP



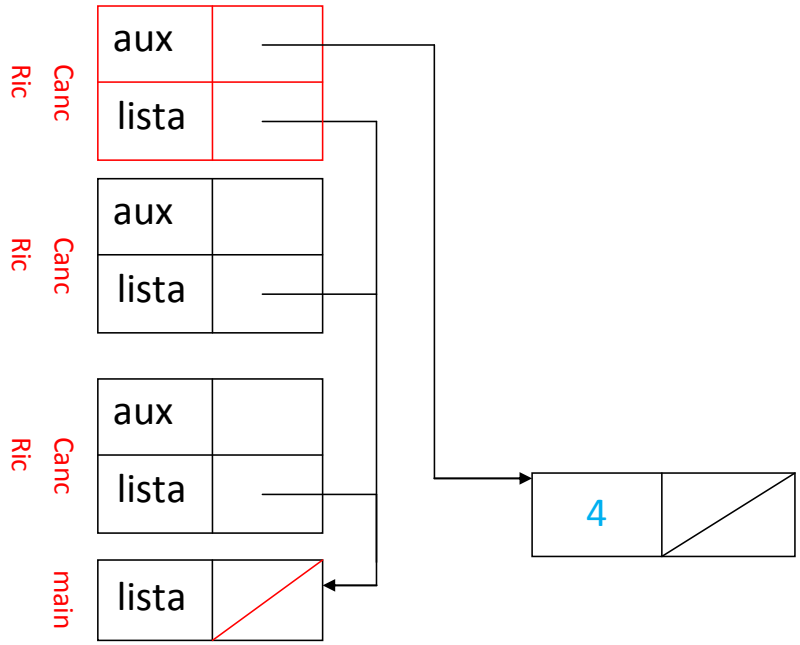
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP



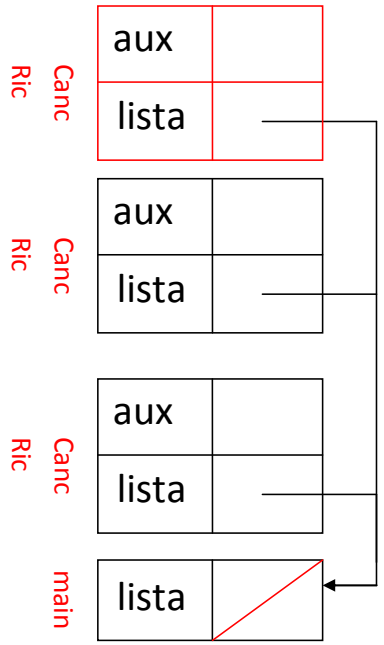
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP

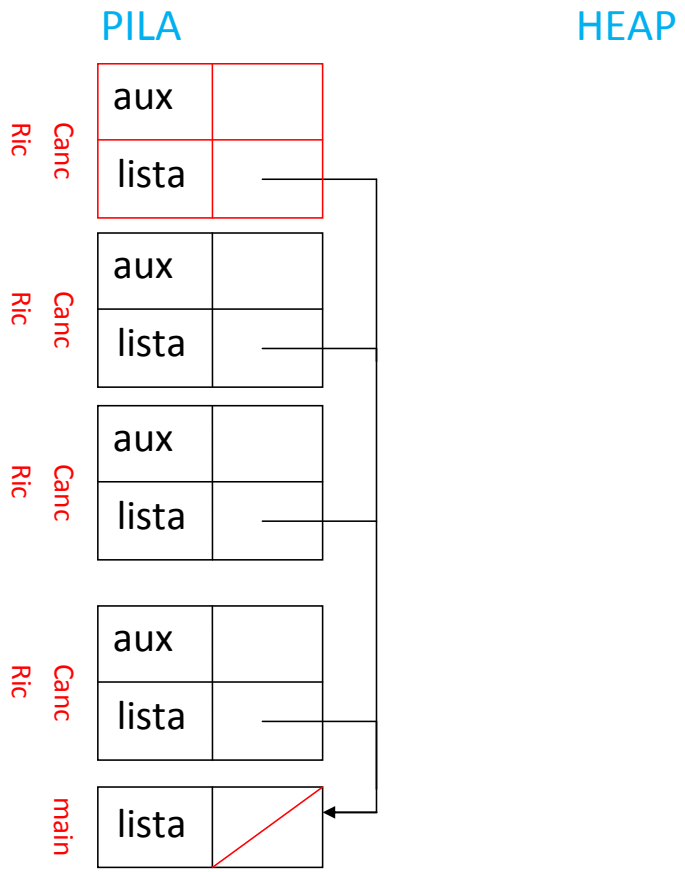


```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP

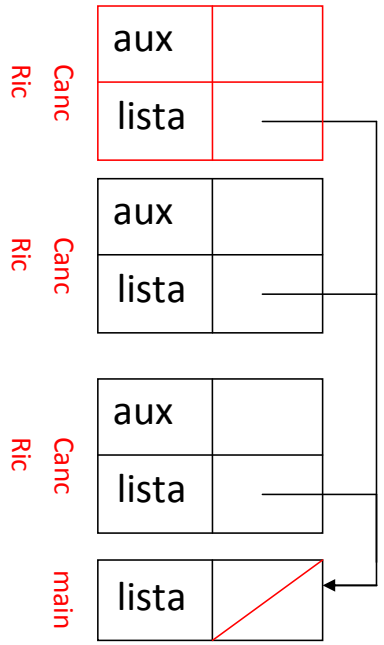


```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



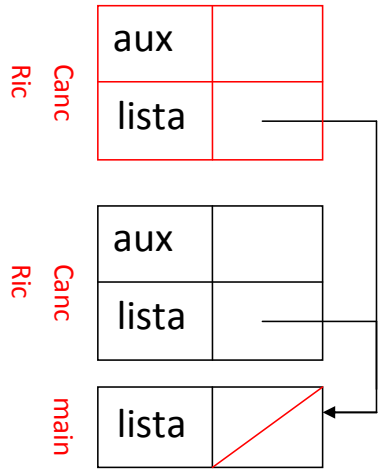
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP



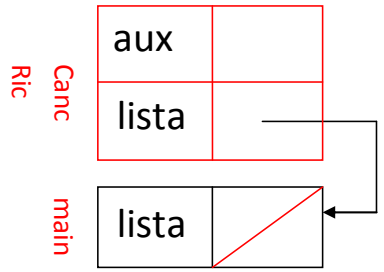
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP



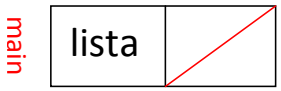
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP



```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP



Ricerca di un elemento in una lista

- Ricordiamo la ricerca lineare incerta su vettori

```
i = 0;      /* indice del primo elemento */
trovato = false;

while (i < DIM && ! trovato)
 {
  if (vet[i] == elem)   /* elemento corrente */
     trovato = true;
  else
     i = i + 1;
 }
```

- sostituiamo l'indice i con un puntatore alla lista che ci permette di scorrerla
- Incapsuliamo questo codice in una funzione a valori booleani

Ricerca di un elemento in una lista

Esempio: Versione Iterativa

```
boolean Ricerca(ListaDiElementi lis, TipoElementoLista  elem)
{
 boolean trovato = false;
  while (lis != NULL && ! trovato)
    {
     if  (lis->info == elem)
          trovato = true;
     else
     lis = lis->next;
    }
return trovato;
}
```

- Non c'è bisogno di un puntatore ausiliario per scorrere la lista
  $\Longrightarrow$ il passaggio per valore consente di scorrere utilizzando il parametro formale!
- Abbiamo assunto che sul tipo `TipoElementoLista` sia definito l'operatore di uguaglianza `==`

Ricerca di un elemento in una lista

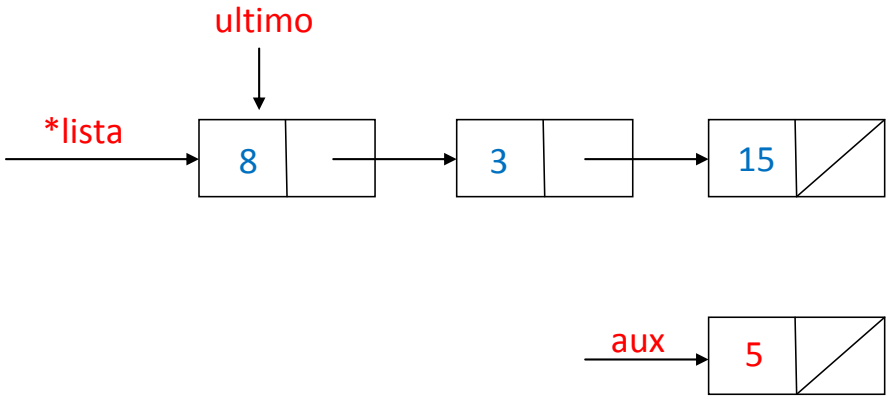
Esempio: Versione Ricorsiva

```
boolean RicercaRic(ListaDiElementi lis, TipoElementoLista  elem)
{
   if (lis == NULL)
      return false;
   else
        if (lis->info == elem) return true;
        else return RicercaRic(lis->next,elem);
  }
```

- Un elemento elem
  - non appartiene alla lista vuota
  - appartiene alla lista con testa x se elem coincide con x
  - appartiene alla lista con testa x diversa da elem e resto L se e solo se appartiene a L

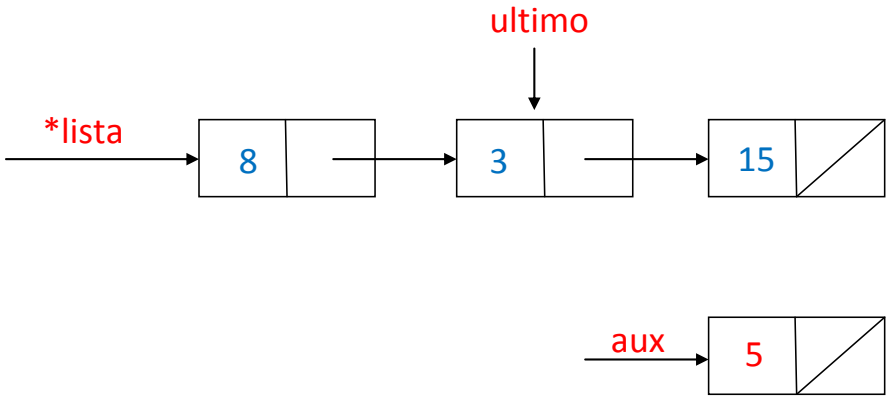
Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



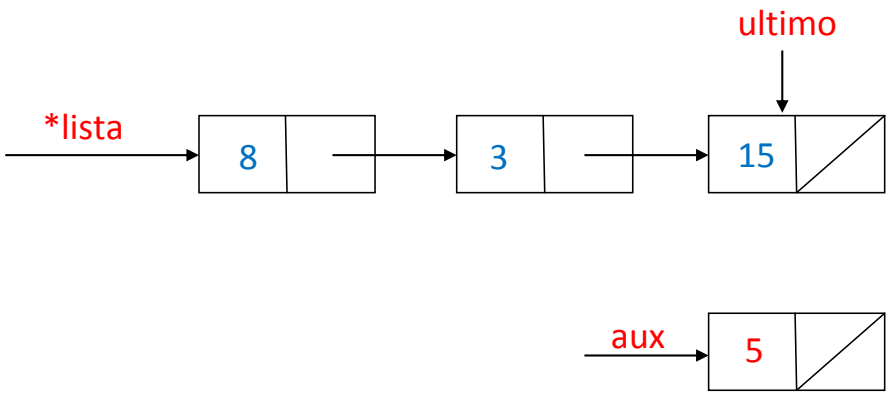
Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



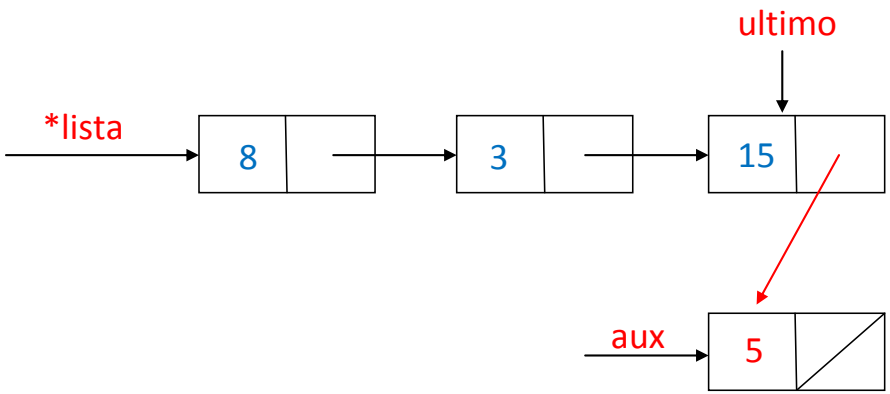
Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



Codice della versione iterativa

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi ultimo;   /* puntatore usato per la scansione */
  ListaDiElementi aux;
                            /* creazione del nuovo elemento */
  aux = malloc(sizeof(ElementoLista));
  aux->info = elem;
  aux->next = NULL;

  if (*lista == NULL)
    *lista = aux;
  else {
    ultimo = *lista;
    while (ultimo->next != NULL)
      ultimo = ultimo->next;
                        /* concatenazione del nuovo elemento in coda alla lista */
    ultimo->next = aux;
  }
}
```

Inserimento ricorsivo di un elemento in coda

- Caratterizzazione induttiva dell'inserimento in coda

  Sia nuovaLista la lista ottenuta inserendo elem in coda a lista.

  1. se lista è vuota, allora nuovaLista è costituita dal solo `elem` (caso base)
  2. altrimenti nuovaLista è ottenuta da lista facendo l'inserimento di elem in coda al resto di lista (caso ricorsivo)

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
  if (*lista == NULL)
  {
    *lista = malloc(sizeof(ElementoLista));
    (*lista)->info = elem;
    (*lista)->next = NULL;
  }
  else
    InserisciCodaLista(       ??       , elem);
}
```

Inserimento ricorsivo di un elemento in coda

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
  if (*lista == NULL)
  {
    *lista = malloc(sizeof(ElementoLista));
    (*lista)->info = elem;
    (*lista)->next = NULL;
  }
  else
    InserisciCodaLista(  (*lista)->next , elem);
}
```

Inserimento ricorsivo di un elemento in coda

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
  if (*lista == NULL)
  {
    *lista = malloc(sizeof(ElementoLista));
    (*lista)->info = elem;
    (*lista)->next = NULL;
  }
  else
    InserzioneInCoda(&((*lista)->next), elem);
}
```

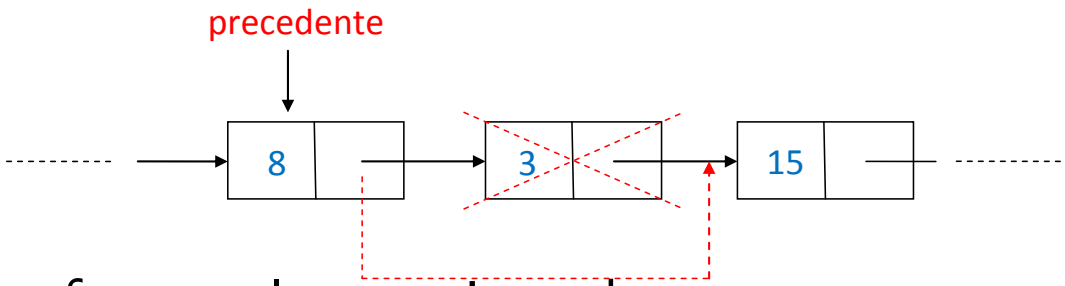
N.B.: Potremmo qui sostituire `*lista == NULL` con `ListaVuota(*lista)`

Cancellazione della prima occorrenza di un elemento

- si scandisce la lista alla ricerca dell'elemento
- se l'elemento non compare non si fa nulla
- altrimenti, a seconda di dove si trova l'elemento, si distinguono tre casi
  1. l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo
     $\Longrightarrow$ passaggio per indirizzo!!
  2. l'elemento non è né il primo né l'ultimo: si aggiorna il campo next dell'elemento che precede quello da cancellare in modo che punti all'elemento che segue
  3. l'elemento è l'ultimo: come (2), solo che il campo next dell'elemento precedente viene posto a `NULL`
- in tutti e tre i casi bisogna liberare la memoria occupata dall'elemento da cancellare

Osservazioni:

- per poter aggiornare il campo next dell'elemento precedente, bisogna fermare la scansione sull'elemento precedente (e non su quello da cancellare)



- per fermare la scansione dopo aver trovato e cancellato l'elemento, si utilizza una sentinella booleana
- Seguendo questa idea, fare per esercizio la versione iterativa della cancellazione.

Versione iterativa:

```
void CancellaElementoLista(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi prec;    /* puntatore all'elemento precedente */
  ListaDiElementi corr;    /* puntatore all'elemento corrente */
  boolean trovato;        /* usato per terminare la scansione */

  if (*lista != NULL)
    if ((*lista)->info==elem)
        { /* cancella il primo elemento */
            CancellaPrimo(lista);
        }
    else    /* scansione della lista e cancellazione dell'elemento */
            prec = *lista; corr = prec->next; trovato = false;
            while (corr != NULL && !trovato)
              if (corr->info == elem)
                {   /* cancella l'elemento */
                    trovato = true;          /* provoca l'uscita dal ciclo */
                    prec->next = corr->next;
                    free(corr);  }
              else  {
                    prec = prec->next;    /* avanzamento dei due puntatori */
                    corr = corr->next;   }
}
```

Versione ricorsiva:

```
void CancellaElementoLista(ListaDiElementi *lista, TipoElementoLista elem)
{
  if (*lista != NULL)
    if ((*lista)->info== elem)
        { /* cancella il primo elemento */
            CancellaPrimo(lista);
        }
    else   /* cancella elem dal resto */
        CancellaElementoLista(&((*lista)->next), elem);
}
```

Cancellazione di tutte le occorrenze di un elemento

Versione iterativa

- analoga alla cancellazione della prima occorrenza
- però, dopo aver trovato e cancellato l'elemento, bisogna continuare la scansione
- ci si ferma solo quando si è arrivati alla fine della lista

  $\Longrightarrow$ non serve la sentinella booleana per fermare la scansione

Cancellazione di tutte le occorrenze di un elemento

Caratterizzazione induttiva

Sia ris la lista ottenuta cancellando tutte le occorrenze di elem da lista. Allora:

1. se lista è la lista vuota, allora ris è la lista vuota (caso base)
2. altrimenti, se il primo elemento di lista è uguale ad elem, allora ris è ottenuta da lista cancellando il primo elemento e tutte le occorrenze di elem dal resto di lista (caso ricorsivo)
3. altrimenti ris è ottenuta da lista cancellando tutte le occorrenze di elem dal resto di lista (caso ricorsivo)

Esercizio

Implementare le due versioni

Versione iterativa

```
void CancellaTuttiLista(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi prec;    /* puntatore all'elemento precedente */
  ListaDiElementi corr;    /* puntatore all'elemento corrente */
  boolean trovato = false;
  while ((*lista != NULL) && ! trovato)  /* cancella le occorrenze */
    if ((*lista)->info!=elem)           /* di elem in testa       */
        trovato = true;
    else  CancellaPrimo(lista);

   if (*lista != NULL)
        {
            prec = *lista; corr = prec->next;
            while (corr != NULL)
              if (corr->info == elem)
                {   /* cancella l'elemento */
                    prec->next = corr->next;
                    free(corr);
                    corr = prec->next;}
              else  {
                    prec = prec->next;    /* avanzamento dei due puntatori */
                    corr = corr->next;   }
        }
}
```

Versione ricorsiva

```
void CancellaTuttiLista(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi aux;

  if (*lista != NULL)
    if ((*lista)->info==elem)
    {                                  /* cancellazione del primo elemento */
      CancellaPrimo(lista);
                                /* cancellazione di elem dal resto della lista */
      CancellaTuttiLista(lista, elem);
    }
    else
      CancellaTuttiLista(&((*lista)->next), elem);
}
```

Inserimento di un elemento in una lista ordinata

- Data una lista (ad es. di interi) già ordinata (in ordine crescente), si vuole inserire un nuovo elemento mantenendo l'ordinamento.

  Versione iterativa: per **esercizio**

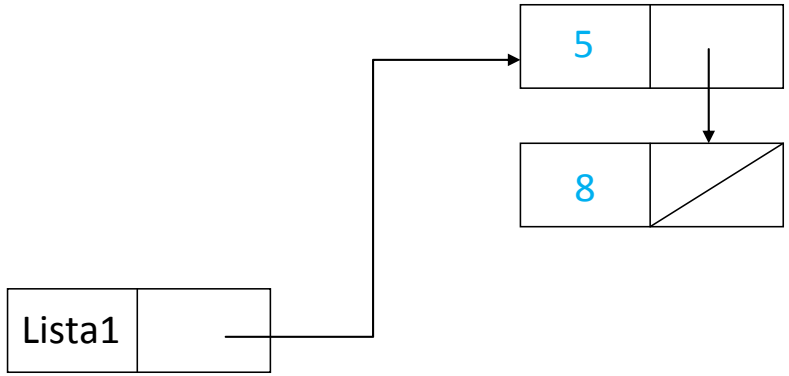
  Versione ricorsiva

- Caratterizziamo il problema induttivamente
- Sia ris la lista ottenuta inserendo l'elemento elem nella lista ordinata lista.
  1. se lista è la lista vuota, allora ris è costituita solo da elem (caso base)
  2. se il primo elemento di lista è maggiore o uguale a elem, allora ris è ottenuta da lista inserendo elem in testa (caso base)
  3. altrimenti ris è ottenuta da lista inserendo ordinatamente elem nel resto di lista (caso ricorsivo)

```
void InserzioneOrdinata(ListaDiElementi *lista, int elem)
{
if (*lista == NULL)
   InserisciTestaLista(lista, elem);
else
    if ((*lista) --> info >= elem)
        InserisciTestaLista(lista, elem);
    else
        InserzioneOrdinata(&((*lista)->next), elem);
}
```

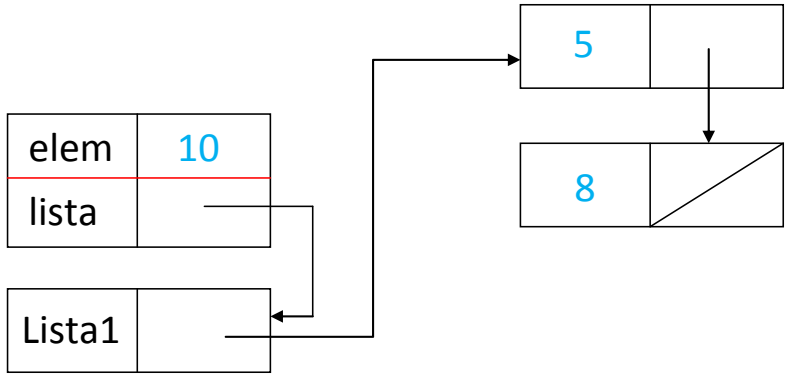
InserzioneOrdinata(&Lista1, 10)

PILA HEAP



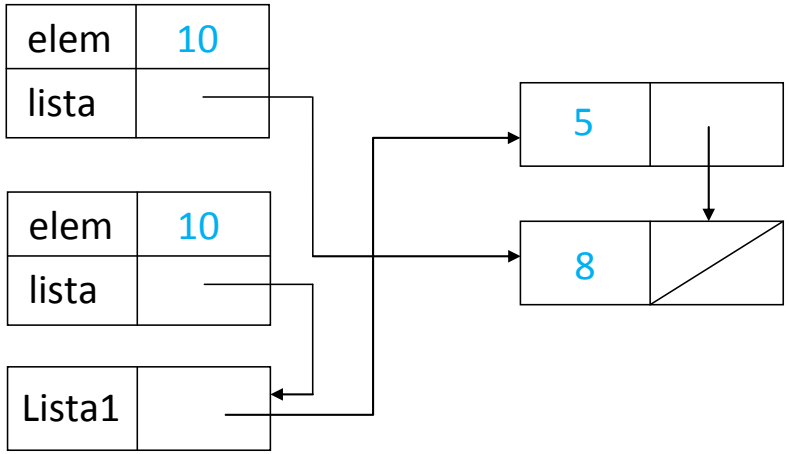
InserzioneOrdinata(&Lista1, 10)

PILA HEAP

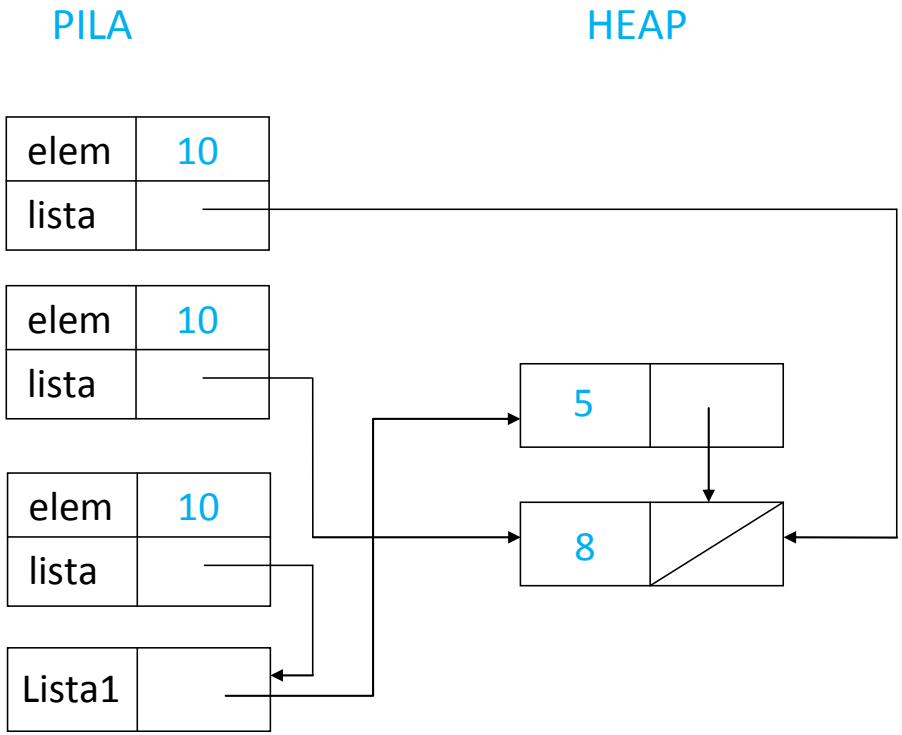


InserzioneOrdinata(&Lista1, 10)

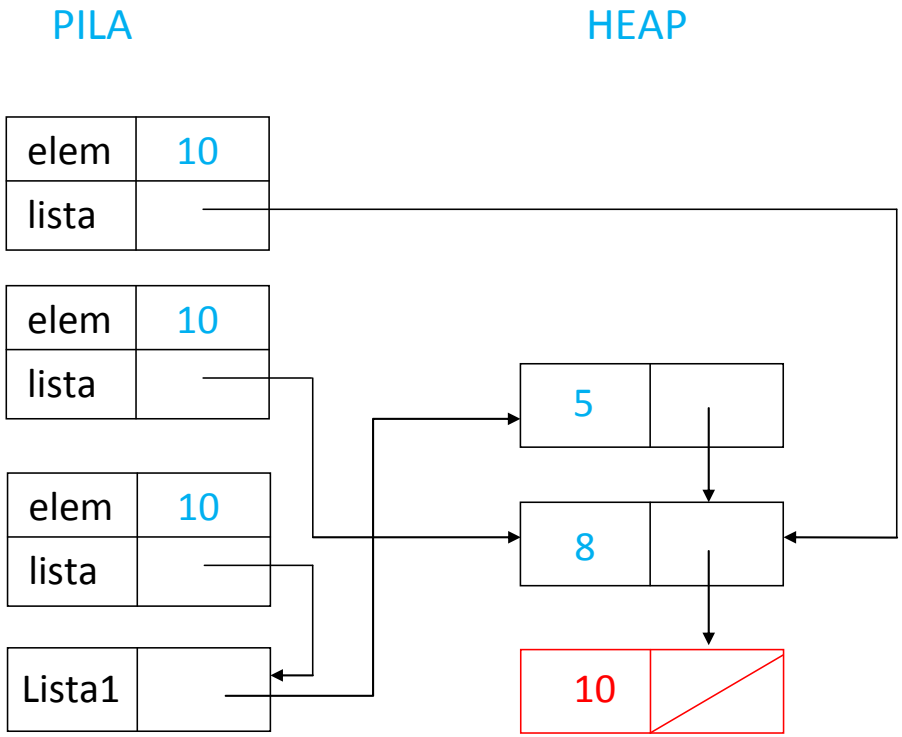
PILA HEAP



InserzioneOrdinata(&Lista1, 10)



InserzioneOrdinata(&Lista1, 10)



Grammatiche e Linguaggi Liberi da Contesto

- Abbiamo visto che molti linguaggi non sono regolari. Consideriamo allora classi più grandi di linguaggi.
- I **Linguaggi Liberi da Contesto** (CFL) sono stati usati nello studio dei linguaggi naturali dal 1950, e nello studio dei compilatori dal 1960.
- Le **grammatiche libere da contesto** (CFG) sono la base della sintassi BNF (Backus-Naur-Form).
- Oggi i CFL sono importanti per XML.
- Studieremo: CFG, i linguaggi che generano e gli alberi sintattici.

Esempio informale di CFG

- Consideriamo $L_{pal} = \{w \in \Sigma^* : w = w^R\}$
- Per esempio: `otto` $\in L_{pal}$, `madamimadam` $\in L_{pal}$.
- Sia $\Sigma = \{0, 1\}$ e supponiamo che L_{pal} sia regolare.

Esempio informale di CFG

- Consideriamo $L_{pal} = \{w \in \Sigma^* : w = w^R\}$
- Per esempio: `otto` $\in L_{pal}$, `madamimadam` $\in L_{pal}$.
- Sia $\Sigma = \{0, 1\}$ e supponiamo che L_{pal} sia regolare.
- Sia n dato dal pumping lemma. Allora $0^n10^n \in L_{pal}$. Nel leggere 0^n il FA deve passare per un loop. Se omettiamo il loop, contraddizione.
- Definiamo L_{pal} induttivamente:

Esempio informale di CFG

- Consideriamo $L_{pal} = \{w \in \Sigma^* : w = w^R\}$
- Per esempio: `otto` $\in L_{pal}$, `madamimadam` $\in L_{pal}$.
- Sia $\Sigma = \{0, 1\}$ e supponiamo che L_{pal} sia regolare.
- Sia n dato dal pumping lemma. Allora $0^n 1 0^n \in L_{pal}$. Nel leggere 0^n il FA deve passare per un loop. Se omettiamo il loop, contraddizione.
- Definiamo L_{pal} induttivamente:
  - **Base:** $\epsilon, 0$, e 1 sono palindromi.
  - **Induzione:** Se w è palindroma, anche $0w0$ e $1w1$ lo sono.
  - Nessun altra stringa è palindroma.

Le CFG sono un modo formale per definire linguaggi come L_{pal}.

1. $P \to \epsilon$
2. $P \to 0$
3. $P \to 1$
4. $P \to 0P0$
5. $P \to 1P1$

- 0 e 1 sono *terminali*
- P è una *variabile* (o *nonterminale*, o *categoria sintattica*)
- P è in questa gramatica anche il *simbolo iniziale*.
- 1–5 sono *produzioni* (o *regole*)

Definizione formale di CFG

Una *grammatica libera da contesto* è una quadrupla

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

dove

- V è un insieme finito di *variabili* o *[simboli] non terminali* o *categorie sintattiche*. (Rappresentano insiemi di stringhe)
- T è un insieme finito di *[simboli] terminali*.
- P è un insieme finito di *produzioni* (regole ricorsive che definiscone le variabili) della forma $A \to \alpha$, dove A è una variabile, la *testa della produzione*, e $\alpha \in (V \cup T)^*$ è il *corpo della produzione*. Possono esserci regole alternative per la stessa variabile.
- S è una variabile distinta chiamata il *simbolo iniziale*.

Esempi

- $G_{pal} = (\{P\}, \{0,1\}, A, P)$, dove $A = \{P \to \epsilon, P \to 0, P \to 1, P \to 0P0, P \to 1P1\}$.
- A volte raggruppiamo le produzioni con la stessa testa: $A = \{P \to \epsilon | 0 | 1 | 0P0 | 1P1\}$.
- Le espressioni regolari su $\{0,1\}$ possono essere definite dalla grammatica

$$G_{regex} = (\{E\}, \{0,1\}, A, E)$$

dove A corrisponde a

$$\{E \to \mathbf{0}, E \to \mathbf{1}, E \to E.E, E \to E + E, E \to E^{\star}, E \to (E)\}$$

Esempio

Espressioni (semplici) in un tipico linguaggio di programmazione.
Gli operatori sono $+$ e *, e gli operandi sono identificatori, cioè stringhe in $L((\boldsymbol{a}+\boldsymbol{b})(\boldsymbol{a}+\boldsymbol{b}+\boldsymbol{0}+\boldsymbol{1})^*)$
Usiamo la grammatica $G = (\{E, I\}, T, P, E)$ dove
$T = \{+, *, (,), a, b, 0, 1\}$ e P è il seguente insieme di produzioni:

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

Derivazioni usando le grammatiche

- *Inferenza ricorsiva*, usando le produzioni dal corpo alla testa
- *Derivazioni*, usando le produzioni dalla testa al corpo

Esempio di inferenza ricorsiva:

	Stringa	Ling.	Prod.	Stringhe usate
(i)	a	I	$5. I \to a$	-
(ii)	b	I	$6. I \to b$	-
(iii)	$b0$	I	$9. I \to I0$	*(ii)*
(iv)	$b00$	I	$9. I \to I0$	*(iii)*
(v)	a	E	$1. E \to I$	*(i)*
(vi)	$b00$	E	$1. E \to I$	*(iv)*
(vii)	$a + b00$	E	$2. E \to E + E$	*(v)*, *(vi)*
(viii)	$(a + b00)$	E	$4. E \to (E)$	*(vii)*
(ix)	$a * (a + b00)$	E	$3. E \to E * E$	*(v)*, *(viii)*

Derivazioni

- Sia $G = (V, T, P, S)$ una CFG, $A \in V$, $\{\alpha, \beta\} \subset (V \cup T)^*$, e $A \to \gamma \in P$.
- Allora scriviamo

$$\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$$

  o, se è ovvia la G,

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

  e diciamo che da $\alpha A \beta$ **si deriva** $\alpha \gamma \beta$.
- Definiamo $\overset{*}{\Rightarrow}$ la chiusura riflessiva e transitiva di $\Rightarrow$, cioe':
  - **Base:** Sia $\alpha \in (V \cup T)^*$. Allora $\alpha \overset{*}{\Rightarrow} \alpha$.
  - **Induzione:** Se $\alpha \overset{*}{\Rightarrow} \beta$, e $\beta \Rightarrow \gamma$, allora $\alpha \overset{*}{\Rightarrow} \gamma$.

Esempio

Derivazione di $a*(a+b00)$ da E nella grammatica delle espressioni:

$$E \Rightarrow E*E \Rightarrow I*E \Rightarrow a*E \Rightarrow a*(E) \Rightarrow$$
$$a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(a+E) \Rightarrow a*(a+I) \Rightarrow$$
$$a*(a+I0) \Rightarrow a*(a+I00) \Rightarrow a*(a+b00)$$

- Ad ogni passo potremmo avere varie regole tra cui scegliere, ad esempio
  $I*E \Rightarrow a*E \Rightarrow a*(E)$, oppure
  $I*E \Rightarrow I*(E) \Rightarrow a*(E)$.
- Non tutte le scelte portano a derivazioni di una particolare stringa, per esempio

  $$E \Rightarrow E+E$$

  non ci fa derivare $a*(a+b00)$.

Derivazioni a sinistra e a destra

- **Derivazione a sinistra** $\underset{lm}{\Rightarrow}$: rimpiazza sempre la variabile più e a sinistra con il corpo di una delle sue regole.
- **Derivazione a destra** $\underset{rm}{\Rightarrow}$: rimpiazza sempre la variabile più a destra con il corpo di una delle sue regole.
- Der. a sinistra: quella del lucido precedente.
- A destra:

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow}$$

$$E * (E) \underset{rm}{\Rightarrow} E * (E + E) \underset{rm}{\Rightarrow} E * (E + I) \underset{rm}{\Rightarrow} E * (E + I0)$$

$$\underset{rm}{\Rightarrow} E * (E + I00) \underset{rm}{\Rightarrow} E * (E + b00) \underset{rm}{\Rightarrow} E * (I + b00)$$

$$\underset{rm}{\Rightarrow} E * (a + b00) \underset{rm}{\Rightarrow} I * (a + b00) \underset{rm}{\Rightarrow} a * (a + b00)$$

Possiamo concludere che $E \overset{*}{\underset{rm}{\Rightarrow}} a * (a + b00)$

Il linguaggio di una grammatica

- Se $G(V, T, P, S)$ è una CFG, allora il **linguaggio di** G è

$$L(G) = \{w \in T^* : S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

  cioè l'insieme delle stringhe su T^* derivabili dal simbolo iniziale.
- Se G è una CFG, chiameremo $L(G)$ un **linguaggio libero da contesto**.
- Esempio: $L(G_{pal})$ è un linguaggio libero da contesto.
- Il linguaggio è visto come l'insieme delle stringhe generate dalla grammatica (approccio *generativo-sintetico*), mentre finora come l'abbiamo visto come l'insieme delle stringhe riconosciute o accettate dagli automi (approccio *riconoscitivo-analitico*).

Teorema 5.7:

$$L(G_{pal}) = \{w \in \{0,1\}^* : w = w^R\}$$

Dimostrazione: (direzione $\supseteq$) Supponiamo $w = w^R$. Mostriamo per induzione su $|w|$ che $w \in L(G_{pal})$.

- **Base:** $|w| = 0$, o $|w| = 1$. Allora w è $\epsilon, 0$, o 1. Dato che $P \to \epsilon, P \to 0$, e $P \to 1$ sono produzioni, concludiamo che $P \overset{*}{\underset{G}{\Rightarrow}} w$ in tutti i casi base.
- **Induzione:** Supponiamo $|w| \geq 2$. Dato che $w = w^R$, abbiamo $w = 0x0$, o $w = 1x1$, e $x = x^R$.
  Se $w = 0x0$ sappiamo che per l'ipotesi induttiva $P \overset{*}{\Rightarrow} x$.
  Allora
  $$P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$$
  Quindi $w \in L(G_{pal})$.
  Il caso di $w = 1x1$ è simile.

(direzione $\subseteq$) Supponiamo che $w \in L(G_{pal})$ e dobbiamo mostrare che $w = w^R$.
Dato che $w \in L(G_{pal})$, abbiamo $P \stackrel{*}{\Rightarrow} w$.
Faremo un'induzione sulla lunghezza di $\stackrel{*}{\Rightarrow}$.

- **Base:** La derivazione $P \stackrel{*}{\Rightarrow} w$ ha 1 passo.
  Allora w deve essere $\epsilon, 0$, o 1, tutte palindromi.
- **Induzione:** Sia $n \geq 1$, e supponiamo che la derivazione abbia $n+1$ passi e che l'enunciato sia vero per tutte le derivazioni di n passi (se $P \stackrel{*}{\Rightarrow} x$ in n passi, allora $x = x^R$). Allora una derivazione di $n+1$ passi deve essere del tipo

$$w = 0x0 \stackrel{*}{\Leftarrow} 0P0 \rightarrow P \text{ oppure } w = 1x1 \stackrel{*}{\Leftarrow} 1P1 \rightarrow P$$

  dove la seconda derivazione di x ha n passi. Infatti $n+1 > 1$ e le produzioni $P \Leftarrow 0P0$ e $P \Leftarrow 1P1$ sono le uniche che permettono passi aggiuntivi.
  Per l'ipotesi induttiva, x è palindroma. Lo sarà quindi anche la stringa w.

Forme sentenziali

- Sia $G = (V, T, P, S)$ una CFG, e $\alpha \in (V \cup T)^*$.
- Se $S \stackrel{*}{\Rightarrow} \alpha$ diciamo che α è una **forma sentenziale** (*sentential form*).
- Se $S \underset{lm}{\Rightarrow} \alpha$ diciamo che α è una **forma sentenziale sinistra**,
- Se $S \underset{rm}{\Rightarrow} \alpha$ diciamo che α è una **forma sentenziale destra**
- Nota: $L(G)$ contiene le forme sentenziali (o *sentence*) che sono in T^*.

Esempi

- Prendiamo la G delle espressioni. Allora $E * (I + E)$ è una forma sentenziale perché

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$$

  Questa derivazione non è né a sinistra né a destra

- $a * E$ è una forma sentenziale sinistra, perché

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E$$

- $E * (E + E)$ è una forma sentenziale destra, perché

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E)$$

Alberi sintattici

- Se $w \in L(G)$, per una CFG, allora w ha un **albero sintattico**, che ci dice la struttura (sintattica) di w
- w potrebbe essere un programma, una query SQL, un documento XML, ...
- Gli alberi sintattici sono una rappresentazione alternativa alle derivazioni e alle inferenze ricorsive.
- Ci possono essere diversi alberi sintattici per la stessa stringa
- Idealmente ci dovrebbe essere solo un albero sintattico (la "vera" struttura), cioè il linguaggio dovrebbe essere non ambiguo.
- Sfortunatamente, non sempre possiamo rimuovere l'ambiguità.

Costruzione di un albero sintattico

Sia $G = (V, T, P, S)$ una CFG. Un albero è un **albero sintattico** per G se:

1. Ogni nodo interno è etichettato con una variabile in V.
2. Ogni foglia è etichettata con un simbolo in $V \cup T \cup \{\epsilon\}$. Ogni foglia etichettata con ϵ deve essere l'unico figlio del suo genitore.
3. Se un nodo interno è etichettato A, e i suoi figli (da sinistra a destra) sono etichettati

$$X_1 X_2 \ldots X_k$$

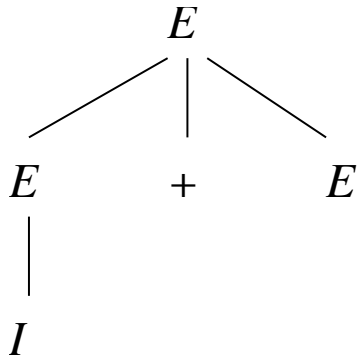
allora $A \to X_1 X_2 \ldots X_k \in P$.

Esempio

Nella grammatica

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$

il seguente è un albero sintattico:



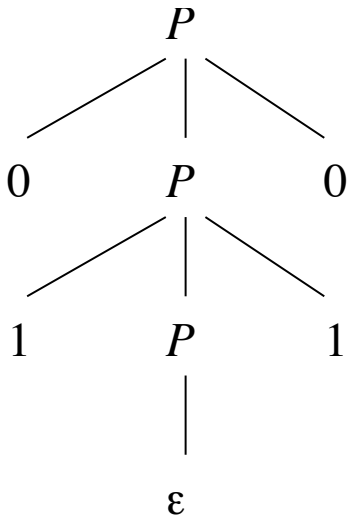
Questo albero sintattico mostra la derivazione $E \overset{*}{\Rightarrow} I + E$

Esempio

Nella grammatica

1. $P \to \epsilon$
2. $P \to 0$
3. $P \to 1$
4. $P \to 0P0$
5. $P \to 1P1$

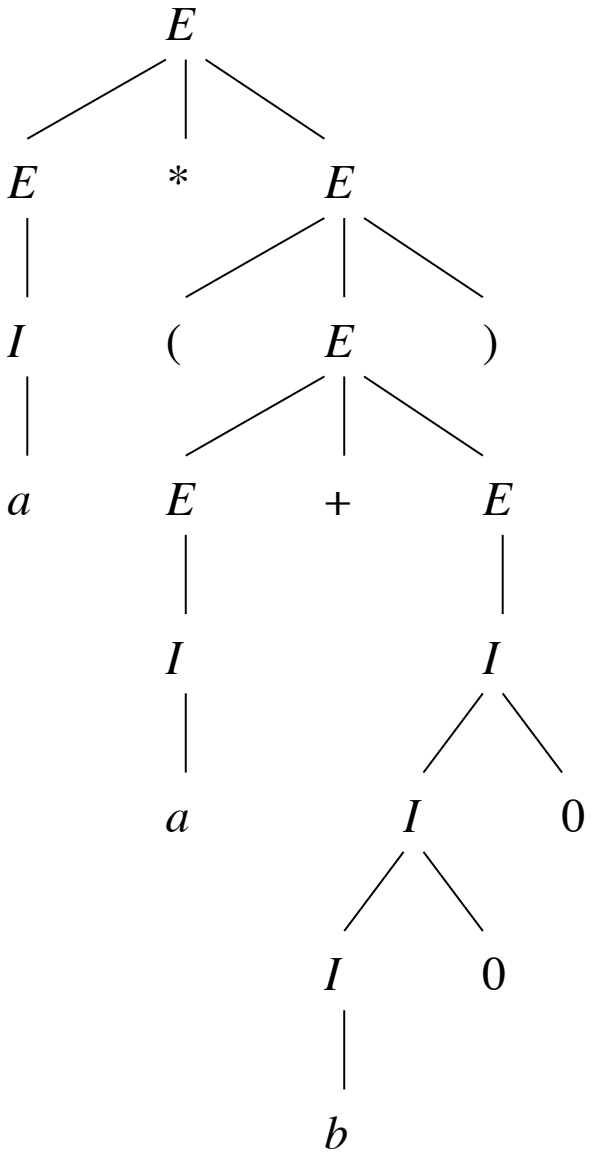
il seguente è un albero sintattico:



Il prodotto di un albero sintattico

- Il **prodotto** di un albero sintattico è la stringa di foglie da sinistra a destra.
- Sono importanti quegli alberi sintattici dove:
  1. Il prodotto è una stringa terminale.
  2. La radice è etichettata dal simbolo iniziale.
- L'insieme dei prodotti di questi alberi sintattici è il linguaggio della grammatica.

Esempio



Gerarchia di Chomsky

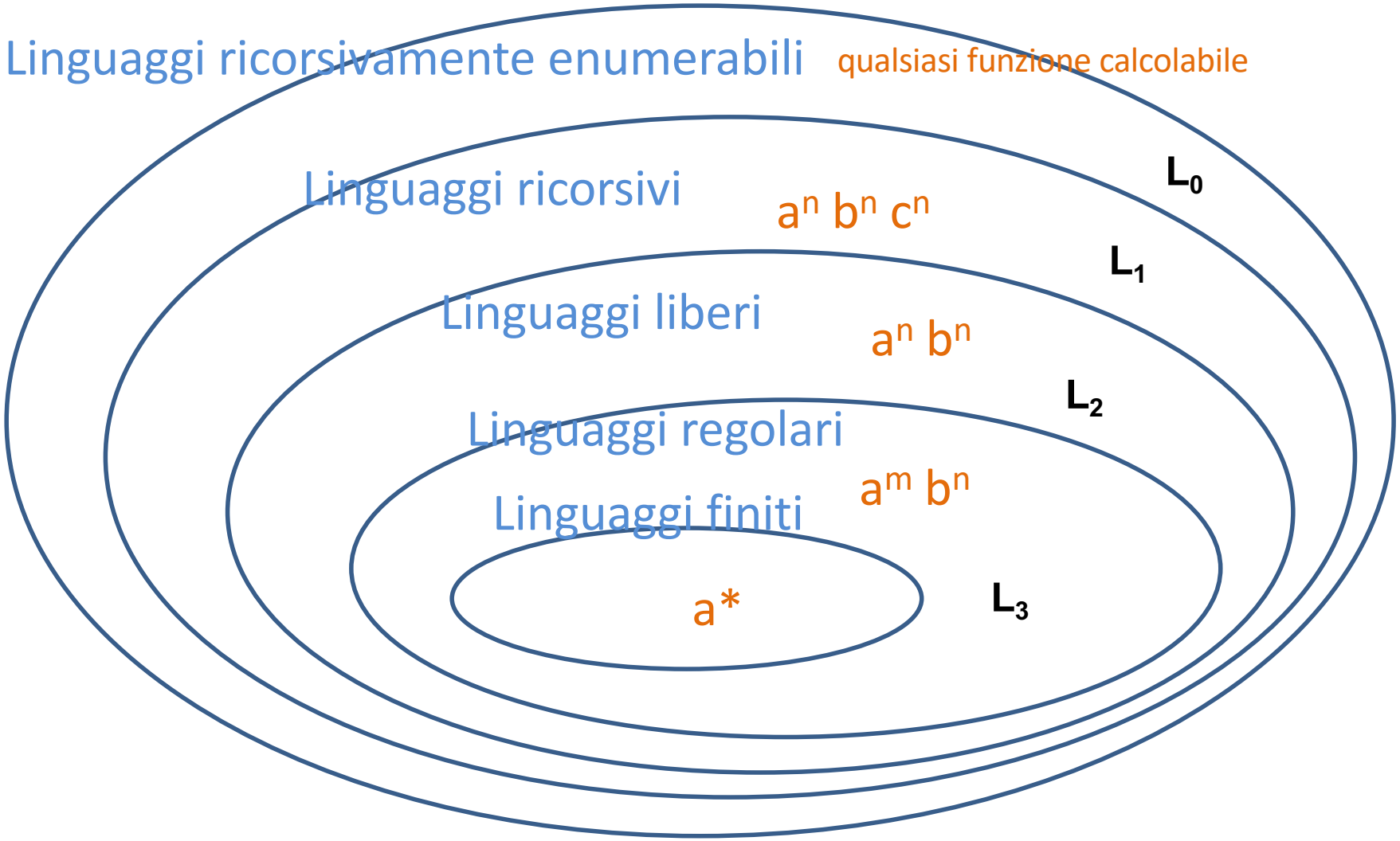
Gerarchia di Chomsky

Type	Language	Grammar	Automaton
0	Recursively Enumerable	Unrestricted	DTM - NTM
1	Context Sensitive	Context Sensitive	Linearly Bounded Automaton
2	Context Free	Context Free	NPDA
3	Regular	Right Linear, Left Linear	DFA, NFA

Abbiamo visto solo
i linguaggi di tipo 2 e 3 e
non abbiamo visto gli NPDA

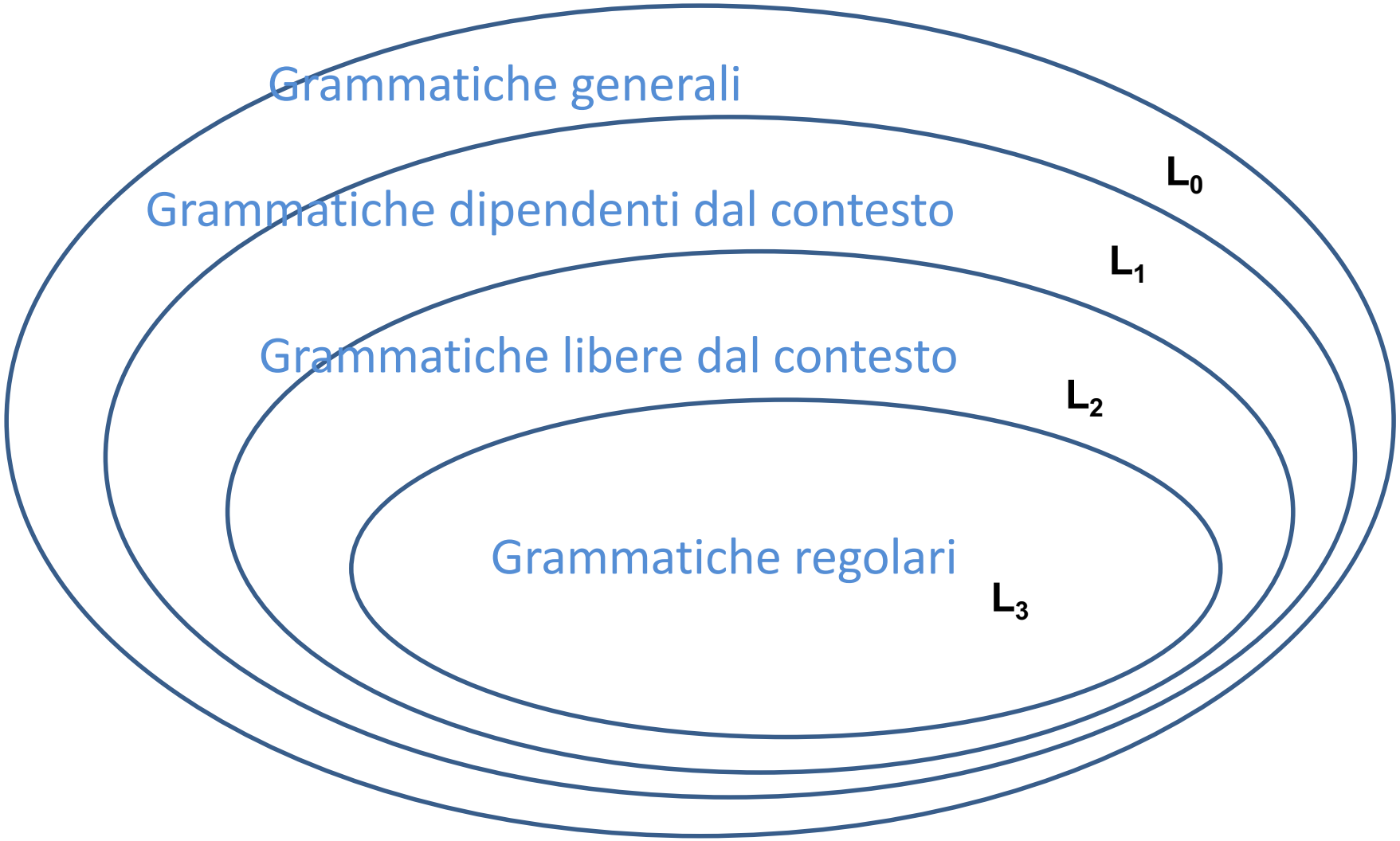
$$\text{Type } 3 \subset \text{Type } 2 \subset \text{Type } 1 \subset \text{Type } 0$$

Gerarchia di Chomsky



$L0 \supset L1 \supset L2 \supset L3 \supset L$ finiti

Gerarchia di grammatiche



Grammatiche Generali*

Produzioni: α --> β, dove α, β sono stringhe di terminali e non terminali

Ex:

- S --> aBc
- aB --> cA
- Ac --> d

Un linguaggio generato da una grammatica di questo genere è detto ricorsivamente enumerabile

* O non ristrette o a struttura di frase

Grammatiche dipendenti dal contesto

Produzioni: α --> β, dove α, β sono stringhe di terminali e non terminali e | α | ≤ | β |

Ex:

- S --> abc | aAbc
- aB --> bA
- Ac --> Bbcc
- bB --> Bb
- aB --> aa | aaA

Genera $a^n b^n c^n$

Grammatiche libere dal contesto

Produzioni: S --> β, dove S è un simbolo non terminale e β è una stringa di terminali e non terminali

Ex:

- S --> ab | aSb

Genera $a^n b^n$

Grammatiche regolari

Produzioni: S --> v, dove S è un simbolo non terminale e v è una stringa composta da un terminale o da un terminale e un non terminale

Ex:

- S --> a | aS

Genera a^*

- Se la produzione è del tipo A --> aB oppure A --> a si chiama **lineare destra o regolare**
- Se la produzione è del tipo A --> Ba oppure A --> a si chiama **lineare sinistra**

Per ogni grammatica lineare a destra esiste una grammatica lineare a sinistra equivalente