Linguaggi, Grammatiche, Parser

Un libro su: Linguaggi, Grammatiche, Parser

28 settembre 2025

Tutto il materiale si trova al link `https://github.com/gdv/linguaggi-formali`. Tutti sono invitati a contribuire a questo libro.

Indice

Introduzione

1 Introduzione

Questo testo si prefigge di introdurre i linguaggi formali e descrivere la rilevanza che questi hanno nel disegno di un compilatore.

In sintesi, un compilatore è un programma che riceve il codice sorgente di un programma e produce un programma equivalente, detto codice oggetto, in un altro linguaggio di programmazione: spesso in assembler.

L'anatomia di un compilatore individua tre componenti principali [1]:

1. Frontend: trasforma il codice sorgente in una rappresentazione intermedia (IR) indipendente dal linguaggio di programmazione.
2. Optimizer: manipola la rappresentazione intermedia per migliorarne le caratteristiche (rendere più veloce il programma, usare meno memoria, ecc.)
3. Backend: riceve la rappresentazione intermedia ed emette il corrispondente codice oggetto.

Vedremo che i linguaggi formali sono il fondamento di ogni frontend.

[1]: Cooper et al. (2022), *Engineering a compiler*

Linguaggi e grammatiche

Linguaggi e Grammatiche 2

2.1 Linguaggi

iIniziamo con le definizioni di base che saranno fondamentali in tutto il testo.

Un **simbolo** o **carattere** è un qualsiasi oggetto.

Definizione 2.1 (Alfabeto) Un **alfabeto**, normalmente indicato con Σ, è un insieme finito e non vuoto di simboli.

alfabeto

Esempi di alfabeti sono `abcdefghijklmnopqrstuvwxyz`, e l'insieme di cifre `0123456789`. Siccome un alfabeto è un insieme, l'ordine dei simboli non è rilevante. In altre parole `9876543210` è sempre l'alfabeto delle cifre. Nel seguito useremo Σ_L per indicare l'alfabeto delle lettere minuscole, Σ_C per l'alfabeto delle cifre, Σ_B per l'alfabeto delle cifre binarie.

La giustapposizione (o concatenazione) di simboli permette di creare parole o stringhe, quali ad esempio `parola` o `2301`. La concatenazione di due simboli viene rappresentata semplicemente scrivendo un simbolo dopo l'altro.

Definizione 2.2 (Parola) Sia Σ un alfabeto. Allora una **parola** è una sequenza $\sigma_1 \cdots \sigma_n$ di simboli, non necessariamente distinti, dell'alfabeto Σ. Una parola viene normalmente rappresentata tramite la giustapposizione dei simboli che compongono la sequenza, rispettando l'ordine.

parola

Normalmente usiamo le ultime lettere dell'alfabeto latino (ad esempio w, x, y, z) per rappresentare parole. La **lunghezza** di una parola è il numero di simboli nella sequenza. La lunghezza della parola z viene indicata con $|z|$.

lunghezza

Alcuni simboli sono speciali perchè hanno un significato particolare e non fanno parte di nessun alfabeto Σ. Il primo simbolo speciale che vediamo è ϵ e rappresenta la stringa vuota, formata dalla sequenza di zero simboli.

Definizione 2.3 (Linguaggio) Dato un alfabeto Σ, un **linguaggio** L è un insieme di parole su Σ.

linguaggio

Sebbene l'alfabeto sia finito, un linguaggio potrebbe contenere un numero infinito di parole. Diventa quindi importante capire quando un linguaggio infinito ha una rappresentazione finita. Notare che praticamente tutti i linguaggi rilevanti sono infiniti, ma con rappresentazione finita. Prendiamo ad esempio JSON: abbiamo una specifica formale (quindi una sequenza di caratteri corrisponde ad un documento JSON se e solo se soddisfa la specifica), e l'insieme dei documenti JSON è infinito.

Esempio 2.1 Un numero è formato da una parte intera che consiste unicamente di cifre e, opzionalmente, da una parte frazionaria che consiste unicamente di cifre. Se la parte frazionaria esiste, allora la parte intera e la parte frazionaria sono separate da un punto. La parte intera non può iniziare con `0` e la parte frazionaria non può finire con `0`. L'insieme di tutti i numeri è un linguaggio (infinito).

La descrizione nell'Esempio 2.1 è di natura insiemistica: il linguaggio è visto come l'insieme delle parole che lo compongono. Quando vogliamo studiare un linguaggio, esistono tre punti di vista alternativi: (1) considerare come sia possibile generare tutte le parole di un linguaggio, (2) descrivere una procedura che determini se una parola appartiene al linguaggio, (3) fornire una proprietà che è soddisfatta da tutte e sole le parole del linguaggio. Si noti che la definizione di linguaggio non cambia. Nel seguito vedremo come questi tre punti di vista si complementino e interagiscono fra loro.

L'approccio, basato sulla procedura per determinare se una parola appartiene al linguaggio, ci porta a definire il primo fondamentale problema computazionale, detto problema dell'**appartenenza** ad un linguaggio.

appartenenza

Problema 1 (Appartenenza) Siano L un linguaggio e z una parola, entrambi sull'alfabeto Σ. Determinare se z **appartiene** a L o, in altre parole, se z sia una parola del linguaggio L.

appartiene

Il problema dell'appartenenza è un **problema di decisione**, perchè ammette solo due risposte: sì o no. La nozione di concatenazione non si applica solo ai caratteri, ma anche a stringhe e linguaggi.

problema di decisione

Definizione 2.4 (Concatenazione) Siano w e x due parole. La loro **concatenazione** wx è ottenuta prendendo w e facendo seguire a questa la stringa x. Più formalmente, se $w = a_1 \cdots a_{|w|}$ e $x = b_1 \cdots b_{|w|}$, allora la loro concatenazione wx è la stringa $a_1 \cdots a_{|w|} b_1 \cdots b_{|w|}$.

concatenazione

Definizione 2.5 (Concatenazione di linguaggi) Siano L_1 e L_2 due linguaggi, rispettivamente su alfabeto Σ_1 e Σ_2. Allora la loro concatenazione L_1L_2 è uguale alla concatenazione di tutte le coppie ordinate di parole di L_1 e L_2. Formalmente, $L_1L_2 = \{w_1w_2 : w_1 \in L_1, w_2 \in L_2\}$.

Una definizione equivalente di concatenazione di linguaggi è $L_1L_2 = \{w_1w_2 : (w_1, w_2) \in L_1 \times L_2\}$. Si noti anche che l'alfabeto del linguaggio L_1L_2 è $\Sigma_1 \cup \Sigma_2$.

Definizione 2.6 Sia L un linguaggio e sia k un numero intero strettamente positivo. Allora la k-esima **potenza** di L, denotata con L^k, è l'insieme di tutte le stringhe ottenute concatenando k parole, non necessariamente distinte, del linguaggio L.

potenza

Per convenzione, si indica con $L^0 = \{epsilon\}$. Quindi possiamo vedere $L^1 = L$, mentre $L^k = L^{k-1}\Sigma$ se $k > 1$. Una variante della Definizione 2.5 consiste nella definizione di potenza di un alfabeto.

Definizione 2.7 Sia Σ un alfabeto e sia k un numero intero strettamente positivo. Allora la k-esima **potenza** di Σ, denotata con Σ^k, è l'insieme di tutte le stringhe di lunghezza k sull'alfabeto Σ.

potenza

Per convenzione, si indica con $\Sigma^0 = \{\epsilon\}$, esattamente come nel caso di potenza di un linguaggio. Vediamo Σ^1 come il linguaggio formato da tutte le parole di lunghezza 1 prese dall'alfabeto Σ, mentre $\Sigma^k = \Sigma^{k-1}\Sigma$ se $k > 1$. Non possiamo dire $\Sigma^1 = \Sigma$ perchè il primo è un insieme di parole e il secondo è un insieme di simboli.

Esempio 2.2 L'alfabeto Σ_B^3 è formato dalle stringhe `000`, `001`, `010`, `011`, `100`, `101`, `110`, `111`.

La nozione di chiusura permette di generalizzare la potenza al caso $k = \infty$.

Definizione 2.8 Sia A un insieme non vuoto e sia ϕ una funzione con mappa insiemi in insiemi. Allora A è un **punto fisso** per ϕ se $\phi(A) \subseteq A$.

punto fisso

In altre parole, A è un punto fisso se l'applicazione di ϕ non "cambia" A. Ovviamente una funzione $\phi : X \mapsto Y$ potrebbe non avere punti fissi e potrebbe anche avere più di un punto fisso. Noi siamo interessati ai *minimi punti fissi* che contengono un insieme A: i più piccoli insiemi $B \supseteq A$ tali che B sia un punto fisso per ϕ.

Definizione 2.9 Sia Σ un alfabeto. Allora la sua **chiusura di Kleene**, denotata con Σ^*, è l'unione infinita di potenze $\bigcup_{k=0}^{\infty} \Sigma^k = \Sigma^0 \cup \Sigma^1 \cup \cdots$.

chiusura di Kleene

Proposizione 2.1 Sia Σ un alfabeto. Allora Σ^* è il minimo punto fisso di $\Sigma \cup \{\epsilon\}$ rispetto alla funzione chiusura di Kleene come minimo punto fisso di $L \cup \{\epsilon\}$ rispetto alla funzione ϕ che mappa ogni linguaggio L in $\phi(L) = L^2$.

Esempio 2.3 $\Sigma_B^* = \{, \mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{100}, \mathtt{000}, \ldots\}$.

Adesso possiamo introdurre la notazione più utilizzata per specificare un linguaggio costruito su un determinato alfabeto.

Nota 2.1 Un linguaggio L su alfabeto Σ è un sottoinsieme di stringhe in Σ^*, quindi $L \subseteq \Sigma^*$.

Siccome ogni linguaggio è un insieme, il simbolo $\emptyset$ denota anche il **linguaggio vuoto** (notare che $\emptyset \in \Sigma^k$, $|\emptyset| = 0$), che è diverso dal linguaggio $\{\epsilon\}$ che consiste esattamente della stringa vuota. Infatti $|\{\varepsilon\}| = 1$. Infine si noti che Σ^* è sempre un insieme infinito. Vediamo adesso alcuni esempi di linguaggi sull'alfabeto Σ_B.

linguaggio vuoto

Esempio 2.4 Le stringhe che consistono in n `0` seguiti da n `1`: $\{\varepsilon, \mathtt{01}, \mathtt{0011}, \mathtt{000111}, \ldots\}$.

Esempio 2.5 le stringhe con un uguale numero di 0 e di 1: $\{\varepsilon, \mathtt{01}, \mathtt{10}, \mathtt{0011}, \mathtt{0110}, \mathtt{0101}, \mathtt{1001}, \ldots\}$.

Esempio 2.6 le stringhe che codificano un numero primo: $\{\mathtt{10}, \mathtt{11}, \mathtt{101}, \mathtt{111}, \ldots\}$.

Esempio 2.7 le stringhe che sono una codifica unaria di un numero primo, senza usare `0`: $\{\mathtt{11}, \mathtt{111}, \mathtt{11111}, \mathtt{1111111}, \ldots\}$.

L'Esempio 2.7 è interessante perchè l'alfabeto effettivamente usato ha un solo simbolo. In altre parole, non sono necessari due simboli per le nozioni di alfabeto e linguaggio.

Nota 2.2 Sia Σ un alfabeto. Allora $\emptyset$, $\{\varepsilon\}$, e Σ^k per ogni intero $k \geq 1$ sono tutti linguaggi per l'alfabeto Σ.

Notiamo che tutti gli esempi di linguaggi che abbiamo dato in questa sezione sono basati sulla descrizione di una proprietà che è vera per tutte e sole le parole del linguaggio. Questo vuole dire che stiamo considerando una visione *insiemistica* dei linguaggi. Vedremo presto che questa non è l'unica visione possibile.

2.2 Grammatiche

Mentre la 2.3 vede un linguaggio come un insieme, in questa sezione andiamo ad introdurre il concetto di grammatica: ciò ci porterà a vedere come sia possibile *generare* tutte e sole le parole appartenenti ad un linguaggio.

Definizione 2.10 (Grammatica) Una **grammatica** G è una quadrupla $G = \langle V, T, P, \mathrm{S}\rangle$ dove:

- V è un insieme finito di **variabili** o non terminali;
- T è un insieme finito di simboli **terminali**, ovvero i simboli dell'alfabeto di riferimento, disgiunto dalle variabili. Quindi $V \cap T = \emptyset$;
- P è un insieme finito di **produzioni** o regole;
- $\mathrm{S} \in V$ è il **simbolo iniziale** (S rappresenta "start").

grammatica

variabili

terminali

simbolo iniziale

Dobbiamo ancora fornire la definizione di produzione.

Definizione 2.11 Una **produzione** $\alpha \to \beta$ formata da tre parti:

- una stringa non vuota α, detta **testa**, sull'alfabeto $V \cup T$,
- il simbolo $\to$ della produzione;
- una stringa β, detta **corpo**, sull'alfabeto $V \cup T$.

produzione

testa

corpo

Notiamo che sia la testa che il corpo di una produzione possono contenere sia terminali che variabili, senza alcuna restrizione: possono quindi esserci solo terminali, solo variabili, oppure entrambi. Anche l'ordine con cui appaiono non ha vincoli. L'unica restrizione è che la testa non può essere la stringa vuota, mentre il corpo potrebbe essere la stringa vuota ($\beta = \epsilon$). Convenzionalmente, per evitare ambiguità, il simbolo di produzione $\to$ non appartiene a $V \cup T$.

Ad ogni grammatica $G = \langle V, T, P, S\rangle$ possiamo associare il linguaggio generato da G. Intuitivamente, si parte dal simbolo iniziale S e si applica ripetutamente una produzione. L'applicazione di una produzione $\alpha \to \beta$ consiste nell'individuare α nella sottostringa attuale e sostituirla con β. Dopo ogni applicazione si ottiene una nuova stringa attuale e il processo può ripetersi, anche applicando una produzione diversa da quella utilizzata nel passo precedente. Quando la stringa attuale è formata da soli simboli terminale, tale stringa è una delle parole del linguaggio. Il processo di ripetute applicazioni di produzioni si chiama derivazione.

Per semplificare alcune parti, introduciamo la definizione di forma sentenziale.

Definizione 2.12 (Forma sentenziale) Sia $G = \langle V, T, P, S\rangle$ una grammatica. Una **forma sentenziale** è una stringa sull'alfabeto $V \cup T$.

forma sentenziale

Esempio 2.8 Consideriamo il linguaggio L su alfabeto Σ_B delle parola che contengono esattamente un `1` e questo carattere is deve trovare in ultima posizione. Una grammatica $G = \langle V, T, P, S\rangle$ che genera il linguaggio L ha $V = \{\mathtt{Z}, \mathtt{U}\}$, $T = \{\mathtt{0}, \mathtt{1}\}$, e le produzioni $\mathtt{S} \to \mathtt{ZU}$, $\mathtt{S} \to \mathtt{U}$, $\mathtt{U} \to \mathtt{1}$, $\mathtt{Z} \to \mathtt{Z0}$, and $\mathtt{Z} \to \mathtt{0}$.

Possiamo notare che la stringa `001` viene generata dalla grammatica G tramite la seguente sequenza di applicazioni di produzione:

$$\mathtt{S} \underset{\mathtt{S}\to\mathtt{ZU}}{\Longrightarrow} \mathtt{ZU} \underset{\mathtt{Z}\to\mathtt{Z0}}{\Longrightarrow} \mathtt{Z0U} \underset{\mathtt{U}\to\mathtt{1}}{\Longrightarrow} \mathtt{Z01} \underset{\mathtt{Z}\to\mathtt{0}}{\Longrightarrow} \mathtt{001}$$

Questa derivazione non è l'unica possibile per generare la stringa `001` e la grammatica G che abbiamo descritto non è l'unica in grado di generare il linguaggio G. Possiamo adesso dare la definizione formale di applicazione di una produzione.

Definizione 2.13 Applicazione di una produzione Siano $\alpha, \beta, \gamma, \delta$ stringhe su alfabeto $V \cup T$. Sia $G = \langle V, T, P, S\rangle$ una grammatica e sia $\alpha \to \beta$ una sua produzione (quindi $\alpha \neq \epsilon$). L'**applicazione** della produzione $\alpha \to \beta$ alla stringa $\gamma\alpha\delta$ è denotata con $\gamma\alpha\delta \underset{\alpha\to\beta}{\Longrightarrow} \gamma\beta\delta$ e rappresenta il fatto che α viene sostituita con β. Se la produzione $\alpha \to \beta$ è facilmente identificabile, oppure se non ci interessa specificare la produzione usata, possiamo denotare l'applicazione indicando solo la grammatica G con $\gamma\alpha\delta \underset{G}{\Longrightarrow} \gamma\beta\delta$. Se anche la grammatica coinvolta non è ambigua, possiamo ometterla usando la scrittura $\gamma\alpha\delta \Longrightarrow \gamma\beta\delta$.

applicazione

Per denotare in modo compatto l'effetto di una sequenza di applicazioni di produzioni, e definire il linguaggio generato da una grammatica, introduciamo il concetto di produzione estesa.

Definizione 2.14 (Applicazione estesa di una produzione) Sia $G = \langle V, T, P, S\rangle$ una grammatica e siano α, β due forme sentenziali di G. Allora possiamo scrivere $\alpha \overset{*}{\underset{G}{\Longrightarrow}} \beta$ se esiste una sequenza $\langle \gamma_1, \ldots, \gamma_n\rangle$ di n forme sentenziali tali che $\alpha = \gamma_1$, $\beta = \gamma_n$, e $\gamma_i \underset{G}{\Longrightarrow} \gamma_{i+1}$ per ogni $1 \leq k < n$. In questo caso diciamo che β è il risultato dell'**applicazione estesa** di produzioni di G alla stringa α.

applicazione estesa

Definizione 2.15 (linguaggio generato) Sia $G = \langle V, T, P, S\rangle$ una grammatica. Il **linguaggio generato** da G, denotato con $L(G)$ è l'insieme $L(G) = \{w \in T^* : S \overset{*}{\underset{G}{\Longrightarrow}} w\}$.

linguaggio generato

Inoltre la definizione di grammatica è estremamente poco vincolata, rendendo difficile capire quali sia il linguaggio che una grammatica genera. Per questo motivo sono interessanti studiare grammatiche ristrette dove valgono proprietà più stringenti di interesse sia per studiare le proprietà dei linguaggi generati che per risolvere più efficientemente il problema di appartenenza. Una prima classificazione delle grammatiche corrisponde alla gerarchia di Chomsky che classifica le grammatiche in tipo 0, 1, 2, 3.

Definizione 2.16 Ogni grammatica $G = \langle V, T, P, S \rangle$ definita secondo la Definizione 2.10 è di tipo 0.

Definizione 2.17 Ogni grammatica $G = \langle V, T, P, S \rangle$ di tipo 0 per cui tutte le sue produzioni hanno lunghezza almeno pari a alla testa è di tipo 1. Queste grammatiche sono anche dette **grammatiche dipendenti dal contesto**.

grammatiche dipendenti dal contesto

Chiaramente le grammatiche di tipo 1 sono incluse in quelle di tipo 1 e si dimostra facilmente che l'inclusione è stretta. Noi trascureremo queste grammatiche perchè quasi tutti i linguaggi artificiali, quali i linguaggi di programmazione, sono generati da una grammatica più ristretta. Le grammatiche di tipo 1 sono invece utilizzate nell'analisi dei linguaggi naturali.

Definizione 2.18 Ogni grammatica $G = \langle V, T, P, S \rangle$ di tipo 0 per cui tutte le sue produzioni hanno testa formata da esattamente una variabile e per coda una forma sentenziale non vuota è di tipo 2. Queste grammatiche sono anche dette **grammatiche libere dal contesto**.

grammatiche libere dal contesto

Quasi tutti i linguaggi artificiali sono generati da grammatiche di tipo 2: per questo motivo saranno oggetto di uno studio estensivo in questo testo. Spesso però siamo interessati ad ulteriori restrizioni.

Definizione 2.19 Ogni grammatica $G = \langle V, T, P, S \rangle$ di tipo 0 per cui tutte le sue produzioni sono forma $A \to aB$ o $A \to a$, dove A e B sono variabili e a è un terminale, è di tipo 3. Queste grammatiche sono anche dette **grammatiche regolari**.

grammatiche regolari

In realtà questa gerarchia ammette la possibilità di classi di grammatiche intermedie fra una classe e un'altra. Vedremo nella Sezione 5.3 diverse classi di grammatiche che sono strettamente incluse in quelle di tipo 2 e includono strettamente quelle di tipo 3.

Con un leggero abuso di linguaggio, talvolta diremo che un linguaggio è di tipo 0, 1, 2, 3 se è generato da una grammatica di tipo 0, 1, 2, 3.

2.3 Automi

Nella sezione precedente abbiamo introdotto il concetto di linguaggio generato da una grammatica che collega grammatiche e linguaggi. In questa sezione collegheremo classi di linguaggi con il modello di calcolo, o classe di automi, più semplice in grado di risolvere il corrispondente problema di appartenza. Non introduciamo una definizione formale di modello di calcolo che unifichi il concetto, ma introdurremo gli specifici modelli per ogni classe di grammatiche che andremo a studiare. Sintetizziamo nella Sezione 2.3 le relazioni che svilupperemo più avanti.

Tabella 2.1: Grammatiche, linguaggi e modelli di calcolo

Linguaggi di tipo	Grammatiche	Modello di calcolo
0	senza restrizioni	Macchina di Turing
1	dipendenti dal contesto	Macchina di Turing con spazio lineare
2	libere dal contesto	Automi a pila
3	regolari	Automi a stati finiti

Linguaggi Regolari 3

3.1 Introduzione

In questo caso vogliamo identificare una classe di linguaggi che sia semplice da definire, che ammetta un riconoscitore efficiente, ma non sia così ristretta da ammettere solo linguaggi banali. Per ottenere questo obiettivo, introduciamo le espressioni regolari che portano ad una definizione di tipo insiemistico dei linguaggi regolari.

espressione regolare

Definizione 3.1 (espressione regolare) Sia Σ un alfabeto. Un'**espressione regolare** è definita secondo le seguenti regole.

1. ε e $\emptyset$ sono espressioni regolari;
2. Ogni simbolo $\sigma \in \Sigma$ è un'espressione regolare;
3. Se E_1 e E_2 sono due espressioni regolari, allora $E_1 \cup E_2$ è un'espressione regolare corrispondente all'unione, o alternanza, fra E_1 e E_2;
4. Se E_1 e E_2 sono due espressioni regolari, allora E_1E_2 è un'espressione regolare corrispondente alla concatenazione di E_1 e E_2;
5. Se E è un'espressione regolare, allora E_1^* è un'espressione regolare corrispondente alla chiusura di Kleene di E, ovvero alla ripetizione di E un numero imprecisato (incluso 0) di volte;
6. Se E è un'espressione regolare, allora (E) è un'espressione regolare.

Le operazioni hanno un ordine di precedenza di valutazione (a parità di precedenza, si valuta da sinistra a destra), in ordine descrescente di precedenza:

1. parentesi
2. chiusura di Kleene *
3. concatenazione
4. unione

Notiamo che le espressioni regolari sono una sintassi che, al momento, può apparire priva di significato concreto. Abbiamo bisogno di introdurre il linguaggio associato ad un'espressione regolare, seguendo pedissequamente i casi della Definizioni 3.1.

Definizione 3.2 (Linguaggio associato ad un'espressione regolare) Sia E un'espressione regolare. Il linguaggio $L(E)$ associato a E è definito secondo le seguenti regole.

1. $L(\epsilon) = \{\epsilon\}$, $L(\emptyset) = \emptyset$;
2. Se E consiste unicamente di un simbolo $\sigma \in \Sigma$, allora $L(E) = \{\sigma\}$;
3. Se $E = E_1 \cup E_2$, allora $L(E) = \{w \in \Sigma^* \,:\, w \in L(E_1) \vee w \in L(E_2)\}$;
4. Se $E = E_1E_2$, allora $L(E) = \{w_1w_2 \,:\, w \in L(E_1)L(E_2)\}$;
5. Se $E = E_1^*$, allora $L(E) = \{\epsilon\} \cup L(E_1) \bigcup_{k=1}^{*} \{w_1 \cdots w_k \,:\, w_1, \ldots, w_k \in L(E)\}$.
6. Se $E = (E_1)$, allora $L(((E)) = L(E)$.

Un linguaggio L viene detto regolare se e solo se esiste un'espressione regolare E che ha L come linguaggio associato. Notiamo che un'espressione regolare non è una grammatica, ma una sintassi minimale per specificare

un linguaggio regolare. Se consideriamo due espressioni regolari identiche quando hanno lo stesso linguaggio associato, possiamo dare alcune proprietà delle operazioni.

Proposizione 3.1
- L'unione è commutativa ($E_1 \cup E_2 = E_2 \cup E_1$), associativa $E_1 \cup (E_2 \cup E_3) = (E_1 \cup E_2) \cup E_3$ e idempotente ($E \cup E = E$).
- La concatenazione è associativa ma non è commutativa.
- $\emptyset$ è identità per l'unione ($\emptyset + E = E + \emptyset = E$),
- $\{\varepsilon\}$ è identità per la concatenazione ($\varepsilon E = E\varepsilon = L$).
- $\emptyset$ è l'annichilitore per la concatenazione ($\emptyset L = L\emptyset = \emptyset$).
- $(E^*)^* = E^*$,
- $\emptyset^* = \varepsilon$;
- $\varepsilon^* = \varepsilon$;

L'enfasi sulla minimalità della sintassi è tipica dell'analisi della capacità espressiva. In altre parole, per capire quali siano i linguaggi regolari è preferibile avere espressioni costruite a partire da un numero estremamente limitato di operazioni. Se invece vogliamo pensare ad espressioni regolari che possano essere utilizzate in pratica, ci aspettiamo di avere altre operazioni o della sintassi addizionale che permetta di esprimere con semplicità casi particolarmente rilevanti. Mostriamo di seguito alcune possibili estensioni della Definizioni 3.1 che **non ne aumentano il potere espressivo**. In altre parole, l'insieme dei linguaggi associati ad un'espressione regolare non cambia se ammettiamo queste estensioni, dove E, E_1, E_2 sono tutte espressioni regolari:

1. E^+, corrispondente alla ripetizione di E un numero imprecisato, ma strettamente positivo, di volte. È equivalente all'espressione EE^*;
2. $E\{n, m\}$, corrispondente alla ripetizione di E un numero di volte compreso fra n ed m, estremi inclusi. L'espressione $E\{0, 0\}$ è equivalente a ϵ, $E\{0, m\}$ (con $m > 0$) è equivalente a $E \cup E\{1, m\}$, $E\{n, m\}$ (con $n > 0$) è equivalente a $EE\{n - 1, m - 1\}$;
3. $[abc]$, dove a, b, c sono tutti simboli di Σ, corrisponde alla scelta fra i simboli indicati. L'espressione regolare equivalente è $a \cup b \cup c$;
4. $[bc]$, dove a, b, c sono tutti simboli di Σ, corrisponde alla scelta fra i simboli *non* indicati. L'espressione regolare equivalente è $\bigcup_{a_1 \in \Sigma \setminus \{a,b,c\}} a_1$.

L'idea di estendere un modello con nuove operazioni che non ne aumentano il potere espressivo semplifica notevolmente lo studio di tale potere espressivo. Intuitivamente, consideriamo due modelli M_1 e M_2, entrambi dotati di un insieme ridotto di operazioni. Sia M_1 che M_2 vengono estesi (ottenendo rispettivamente i modelli estesi M_1^* e M_2^*) senza aumentare il rispettivo potere espressivo. Quando vogliamo dimostrare che i due modelli M_1 e M_2 hanno lo stesso potere espressivo, possiamo simulare M_1 con M_2^* (mostrando quindi che tutto quello che può essere fatto con M_1 può essere fatto con M_2^*) e simulare M_2 con M_1^*. In questo caso, il fatto di potere usare due modelli estesi rende più semplice la dimostrazione di correttezza della simulazione. Al tempo stesso, studiare un modello ristretto, come M_1, rende più semplice dimostrare che qualcosa *non* può essere realizzato da M_1. Sfrutteremo questa intuizione nel resto del libro.

Esempio 3.1 Si consideri il linguaggio su Σ_B delle parole non nulle formate da `0` e `1` alternati — equivalentemente, nessuna parola contiene la sottostringa `00` e `11` — Trovare un'espressione regolare che ha associato tale linguaggio.
Dividiamo le parole del linguaggio a seconda che abbiano lunghezza pari o dispari e che inizino per `0` e `1`, facendo l'unione di questi quattro linguaggi. Otteniamo così l'espressione $(\texttt{01})^*|(\texttt{10})^*|\texttt{1}(\texttt{01})^*|\texttt{0}(\texttt{10})^*$.

$$01 \to \{01\}$$

$$(01)^* \to \{\epsilon, 01, 0101, 010101, \cdots\}$$

$$(01)^* + (10)^* \to \{\epsilon, 01, 10, 0101, 1010, \cdots\}$$

ma posso volere diverse quantità di 0 e 1, sempre mantenendo l'alternanza, metto o uno 0 o un 1 davanti a quanto ottenuto appena sopra:

$$(01)^* + (10)^* + 0(10)^* + 1(01)^* \to \{\epsilon, 01, 10, 010, 101, \cdots\}$$

non è comunque l'unica soluzione, si può avere:

$$(\epsilon + 1)(01)^*(\epsilon + 0) \to \{\epsilon, 01, 10, 010, 101, \cdots\}$$

oppure ancora:

$$(\epsilon + 0)(10)^*(\epsilon + 1)$$

Esercizio 3.1. Ho $E = (0 + 1)^*0^*(01)^*$:

- 001 fa parte del linguaggio?
- 1001 fa parte del linguaggio?
- 0101 fa parte del linguaggio?
- 0 fa parte del linguaggio?
- 10 fa parte del linguaggio?

Si ricorda che: $(0 + 1)^* \neq 0^* + 1^*$

Esempio 3.2 ho $ER = ((01)^* \cdot 10 \cdot (0 + 1)^*)^*$

- 0101 fa parte del linguaggio? No
- 01000 fa parte del linguaggio? No
- 01011 fa parte del linguaggio? No
- 10111 fa parte del linguaggio? Si, $\epsilon \cdot 10 \cdot 111$
- 101010 fa parte del linguaggio? Si, prendo $10 \cdot 1010$
- 101101 fa parte del linguaggio? Si, $\epsilon \cdot 10 \cdot 1$ due volte
- 0101100011 fa parte del linguaggio? Si, $0101 \cdot 10 \cdot 0011$ (0011 lo posso prendere da $(0 + 1)^*$)

Esempio 3.3 ho $ER = ((01)^* \cdot 10 \cdot (0 + 1))^*$

- 0101 fa parte del linguaggio? No
- 01000 fa parte del linguaggio? No
- 01011 fa parte del linguaggio? No
- 10111 fa parte del linguaggio? No
- 101010 fa parte del linguaggio? No
- 101101 fa parte del linguaggio? Si, $\epsilon \cdot 10 \cdot 1$ due volte
- 0101100011 fa parte del linguaggio? No

Esempio 3.4 Da $L \subseteq \{0, 1\}|$ stringhe contenenti almeno una volta 01 quindi: $(0 + 1)^*01(0 + 1)^*$

Esempio 3.5 ho $ER = (00^*1^*)^*$, quindi: $L = \{\epsilon, 0, 01, 000, 001, 010, 011\} = \{\epsilon\} \cup \{w \in \{0,1\}^* |$ w che inizia con 0$\}$

Esempio 3.6 ho $ER = a(a+b)^*b$, quindi: $L = \{w \in \{a,b\}^* |$ w inizia con a e termina con b$\}$

Esempio 3.7 ho $ER = (0^*1^*)^*000(0+1)^*$, quindi, sapendo che $\{0,1\}^*$ mi permette tutte le combinazioni che voglio come $(0+1)^*$: $L = \{w \in \{0,1\}^* |$ w come voglio con tre 0 consecutivi$\}$

Esempio 3.8 ho $ER = a(a+b)^*c(a+b)^*c(a+b)^*b$, quindi: $L = \{w \in \{a,b,c\}^* |$ w inizia con a, termina con b e contiene almeno due c, eventualmente non adiacenti$\}$

Esempio 3.9 Da $L \subseteq \{0,1\}$ ogni 1 è seguito da 0, a meno che non sia l'ultimo carattere, ovvero 11 non compare
quindi: $(10+0)^*(\epsilon+1)^*$

Esempio 3.10 cerco ER per $L \subseteq \{0,1\}^* |$ stringhe contenenti un numero pari di 1: $(0^*10^*1)^*0^*$ oppure: $(0+10^*1)^*$

Esempio 3.11 cerco ER per $L \subseteq \{0,1\}^* |$ stringhe contenenti un numero dispari di 1: $(0^*10^*)^*0^*10^*$ oppure: $(0+10^*1)^*10^*$

Esempio 3.12 cerco ER per $L \subseteq \{0,1\}^* |$ stringhe contenenti un numero divisibile per 3 di 0: $(1^*01^*01^*0)^*1^*$

Esempio 3.13 cerco ER per $L \subseteq \{0,1\}^* |$ stringhe contenenti al più una coppia di 1 consecutivi: $(10+0)^*(11+1+\epsilon)(01+0)^*$

Esempio 3.14 cerco ER per $L \subseteq \{a,b,c\}^* |$ stringhe contenenti almeno una a e almeno una b : $c^*\,(a(a+c)^*b + b(b+c)^*a)\,(a+b+c)^*$

3.2 Grammatiche regolari

Una grammatica è regolare se ha una delle seguenti due forme.

Definizione 3.3 Sia $G = \langle V, T, P, S\rangle$ una grammatica. Allora G è detta regolare destra se tutte le su produzioni sono nella forma $A \to a$ o nella forma $A \to Ba$, dove $A, \in VB$ e $a \in T$. Inoltre, G è detta regolare sinistra se tutte le su produzioni sono nella forma $A \to a$ o nella forma $A \to aB$, dove $A, \in VB$ e $a \in T$.

3.3 Diagrammi di transizione

Consideriamo il diagramma di Figura 3.1, dove i nodi sono degli stati e le frecce rappresentano transizioni fra stati al verificarsi di una determinata condizione. Il diagramma rappresenta un tipico ciclo di edit, compilazione, test, messa in produzione del codice. Questa idea può essere formalizzata in un modello di calcolo che è in grado di riconoscere i linguaggi regolari, come vedremo nella prossima sezione.

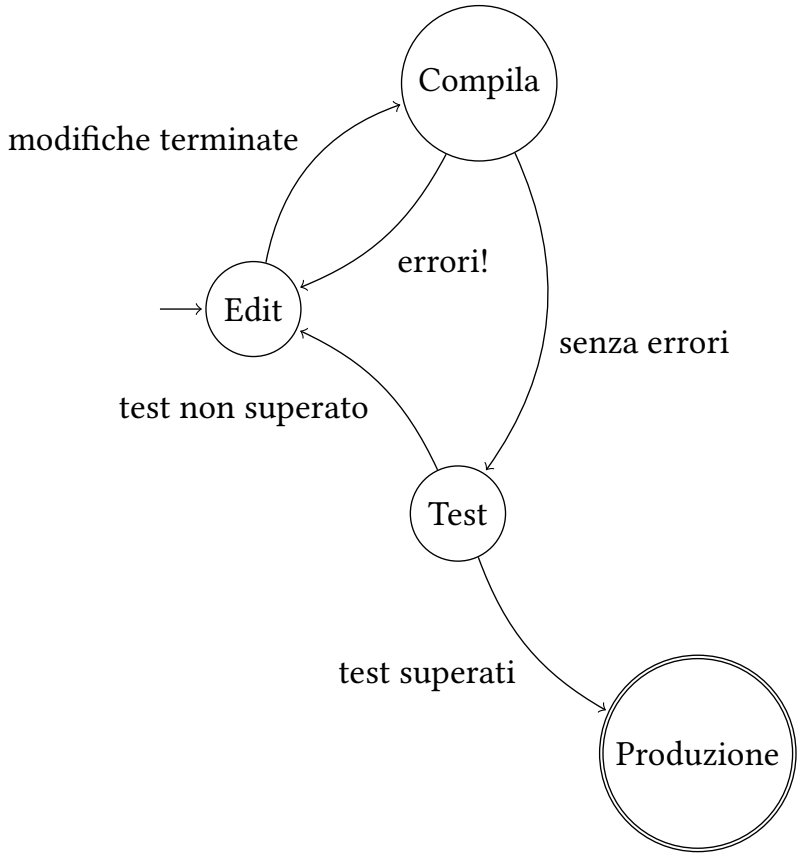


Figura 3.1: Esempio di diagramma di transizione

3.4 Automi deterministico a stati finiti

L'obiettivo di questa sezione è introdurre un modello di calcolo, ispirato ai diagrammi di transizione, che sia il più semplice possibile e in grado di riconoscere tutti i linguaggi regolari. Questo modello viene chiamato automa a stati finiti deterministico (**DFA**).

DFA

Definizione 3.4 (automa deterministico a stati finiti) Un **automa deterministico a stati finiti** è una quintupla $\mathscr{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ consiste nelle seguenti parti:

automa deterministico a stati finiti

- Σ: alfabeto di input;
- Q: insieme finito di *stati*;
- $\delta : Q \times \Sigma \mapsto Q$: *funzione di transizione*. Prende come argomento uno stato e un simbolo di input e restituisce uno stato;
- $q_0 \in Q$: stato iniziale;
- $F \subseteq Q$: insieme di *stati finali*, o accettanti.

L'idea intuitiva è che iniziamo nello stato q_0. Ad ogni passo viene letto (o *consumato*) un carattere c in input e determinato il prossimo stato tramite la funzione di transizione. Più precisamente, l'automa si muove nello stato $\delta(q, c)$, dove q è lo stato attuale. Dopo avere consumato tutta la stringa in input, questa stringa viene accettata (o riconosciuta) se e solo se l'automa si trova in uno stato finale. La 3.4 specifica un automa *deterministico* a stati finiti in quanto la funzione di transizione restituisce esattamente un singolo stato e riceve esattamente un carattere. Di conseguenza l'automa si trova sempre in un solo stato.

La descrizione della funzione di transizione δ può avvenire in due modi: con una tabella dove vengono indicati tutti i valori di $\delta(q, c)$, oppure con un diagramma che ricorda quelli di transizione. Nell'ultimo caso denotiamo lo stato iniziale con una freccia entrante che non ha un nodo di inizio, gli stati finali con un doppio cerchio, e l'etichetta di un arco dallo stato q_1 allo stato q_2 è l'insieme di caratteri c tali che $\delta(q_1, c) = q_2$.

Esempio 3.15 Consideriamo il linguaggio L formato dalle strighe binarie che hanno la sottostringa `01`. Equivalentemente $L = \{x\texttt{01}y : x, y \in \Sigma_B^*\}$. Leggendo la stringa in input, l'automa è in uno dei seguenti casi:

1. se ha "già visto" 01, accetterà qualsiasi input, indipendentemente dai caratteri che leggerà in futuro;
2. pur non avendo ancora visto 01, l'input più recente è stato 0. Se leggerà `1`, andrà nello stato accettante, altrimenti rimane nello stesso stato;
3. non ha ancora visto `0`, e rimane in questo stato finchè non legge `0`.

la terza condizione rappresenta lo stato iniziale. All'inizio bisogna vedere uno 0 e poi un 1. Ma se nello stato q_0 si vede per primo un 1 allora non abbiamo fatto alcun passo verso 01, e dunque dobbiamo permanere nello stato $q_0, \delta(q_0, 1) = q_0$. D'altra parte se nello stato iniziale vedo 0 siamo nella seconda condizione, uso quindi q_2 per questa condizione, si avrà quindi $\delta(q_0, 0) = q_2$. Vedo ora le transizoni di q_2, se vedo 0 ho che 0 è l'ultimo simbolo incontrato quindi uso nuovamente q_2, $\delta(q_2, 0) = q_2$, in attesa di un 1. Se arriva 1 passo allo stato accertante q_1 corrispondente alla prima condizione, $\delta(q_2, 1) = q_1$. Ora abbiamo incontrato 01 quindi può succedere qualsiasi cosa e dopo qualsiasi cosa accada potremo nuovamente aspettarci qualsiasi cosa, ovvero $\delta(q_1, 0) = \delta(q_1, 1) = q_1$. Si deduce quindi che: $Q = \{q_0, q_1, q_2\}$ e $F = \{q_1\}$ quindi: $A = \{\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}\}$ con in totale le seguenti transizioni: $\delta(q_0, 1) = q_0$ $\delta(q_0, 0) = q_2$ $\delta(q_2, 0) = q_2$ $\delta(q_2, 1) = q_1$ $\delta(q_1, 0) = q_1$ $\delta(q_1, 1) = q_1$

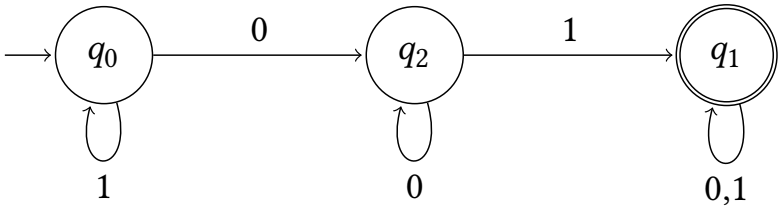
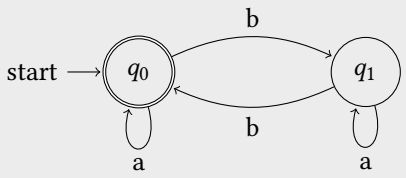


Figura 3.2: DFA che accetta il linguaggio $\Sigma_B^*\texttt{01}\Sigma_B^*$

δ	0	1
$\rightarrow q_0$	q_1	q_0
$*\, q_1$	q_1	q_1
q_2	q_2	q_1

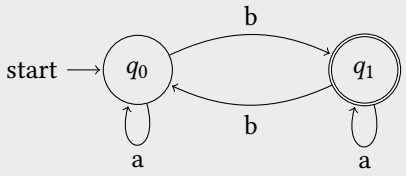
Tabella 3.1: L'automa di Figura 3.2, in forma tabellare.

Esempio 3.16 Trovo automa per: $L = \{w \in \{a,b\}^* |$ w che contiene un numero pari di b$\}$



ovvero se da q_0 vado a q_1 sono obbligato ab generare due b, dato che il nodo accettnate è q_0. In entrambi i nodi posso generare quante a voglio.

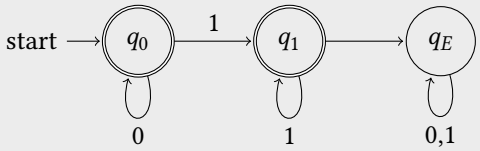
Esempio 3.17 Trovo automa per: $L = \{w \in \{a,b\}^* |$ w che contiene un numero dispari di b$\}$



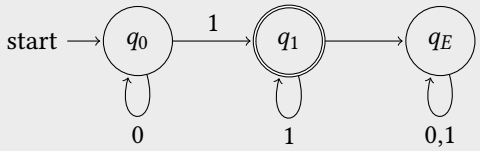
ovvero se da q_0 vado a q_1 sono obbligato ab generare una sola b, dato che il nodo accettnate è q_1. In entrambi i nodi posso generare quante a voglio e posso tornare da q_1 a q_0 per generare altre b.

Esempio 3.18 Trovo automa per: $L = \{w \in \{0,1\}^* | w = 0^n 1^m\}$ vediamo i vari casi: **Si ha che q_E è lo stato pozzo dove vanno le stringhe venute male**

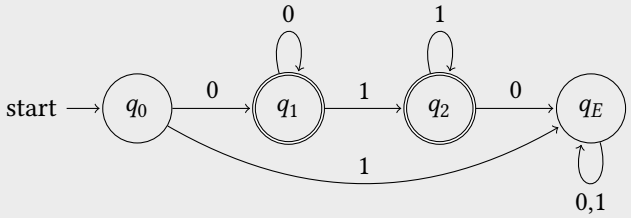
- $n, m \geq 0$:



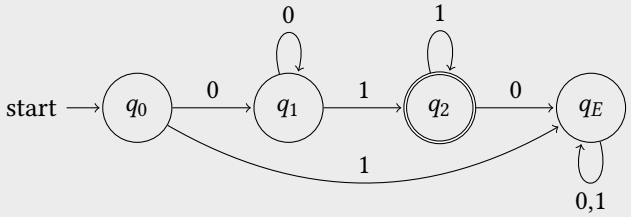
  ovvero posso non generare nulla e uscire subito con q_0, generare solo un 1 e passare a q_1 e uscire oppure generare 0 e 1 a piacere con l'ultimo stato o generare 0 a piacere dal primo e 1 a piacere dal secondo.
- $n \geq 0$ $m > 0$:



  ovvero come l'esempio sopra solo che non posso uscire in q_0 in quanto almeno un 1 deve essere per forza generato
- $n > 0$ $m \geq 0$:

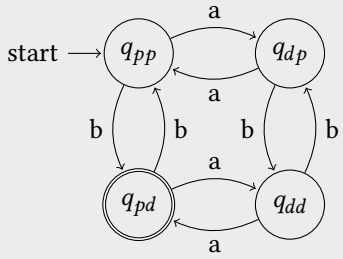


  CHIARIRE
- $n, m > 0$:

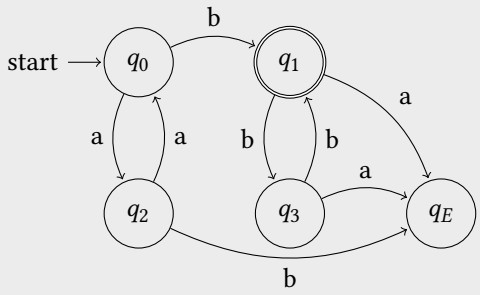


  CHIARIRE

Esempio 3.19 Trovo automa per: $L = \{w \in \{a,b\}^* |$ w che contiene un numero pari di a e dispari di b$\}$



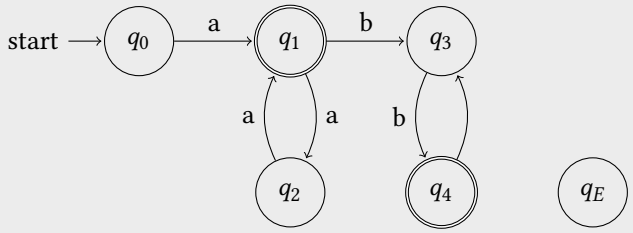
Esempio 3.20 Trovo automa per: $L = \{w \in \{a,b\}^* |$ w che contiene un numero pari di a seguito da uno dispari di b$\}$
$L = \{a^{2n}b^{2k+1} | j,k \geq 0\}$



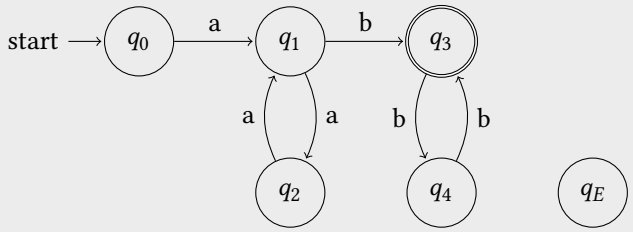
ovvero in tabella:

δ	a	b
$\rightarrow q_0$	q_1	q_2
q_1	q_0	q_E
$* q_2$	q_E	q_3
q_3	q_E	q_2
q_E	q_E	q_E

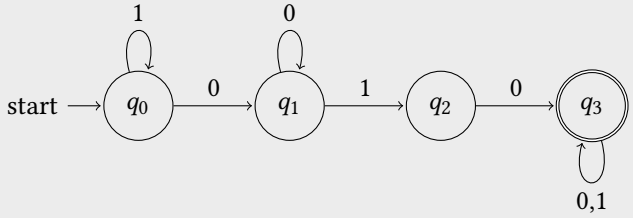
Esempio 3.21 Trovo automa per: $L = \{a^{2k+1}b^{2h} | h,k \geq 0\}$



Esempio 3.22 Trovo automa per: $L = \{a^{2n+1}b^{2k+1} | n,k \geq 0\}$



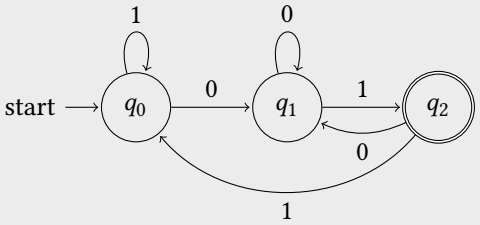
Esempio 3.23 Trovo automa per: $L = \{x010y | x,y \in \{o,1\}^*\}$



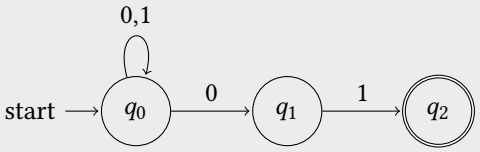
3.4.1 Automi non deterministici

Un automa a stati finiti non deterministici (*NFA*) può trovarsi in diversi stati contemporaneamente. Come i DFA accettano linguaggi regolari e spesso sono più semplici da trattare rispetto ai DFA.
Un NFA è definito come un DFA ma si ha un diverso tipo di transizione δ, che ha sempre come argomenti uno stato e un simbolo di input ma restituisce zero o più stati.

Esempio 3.24 Sia $L = \{x01 | \, x \in \{o, 1\}$ ovvero il linguaggio formato da tutte le stringhe binarie che terminano in 01.
Avremo il seguente automa determinsitico:



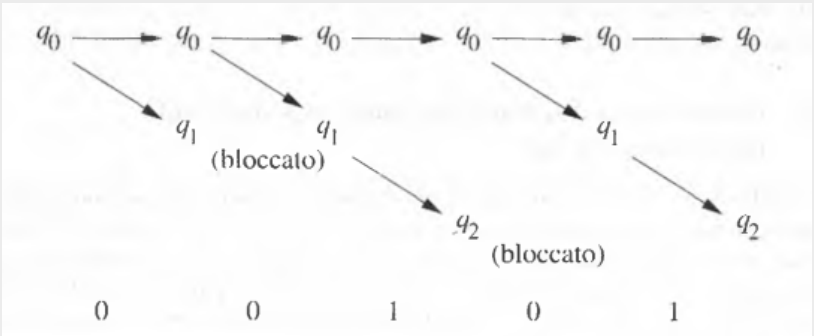
che diventa il seguente NFA:



quindi con: $\delta(q_0, 0) = \{q_0, q_1\}$ $\delta(q_0, 1) = \{q_0\}$ $\delta(q_1, 0) = \emptyset$ $\delta(q_1, 1) = \{q_2\}$ $\delta(q_2, 0) = \emptyset$ $\delta(q_2, 1) = \emptyset$ in forma tabulare:

δ	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_1\}$
q_1	$\emptyset$	$\{q_2\}$
$* q_0$	$\emptyset$	$\emptyset$

vediamone la simulazione per la stringa 00101:



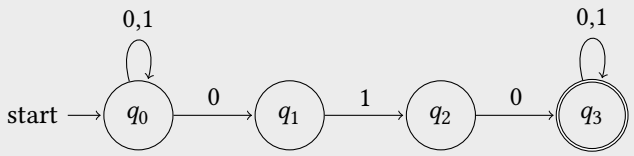
ovvero si parte dallo stato inziale, quando viene letto 0 si passa a q_0 e q_1, poi viene letto il secondo 0 quindi q_0 va nuovamente verso q_0 e q_1 mentre il primo q_1 muore non avendo transizioni su 0. Arriva poi l'1 quindi q_0 va solon verso q_0 e q_1 verso q_2 e sarebbe accettante ma l'input non è finito. Ora arriva 0 e q_2 si blocca mentre q_0 va sia in q_0 che in q_1. Arriva infine un 1 che manda q_0 in q_0 e q_1 in q_2 che è accettante e non avendo altri input si è dimostrata l'appartenenza della stringa al linguaggio

definisco quindi un NFA come una quintupla: $A = (Q, \Sigma, \delta, q_0, F)$ con, a differenza dei DFA: $\delta : Q \times F \to 2^Q$ Possiamo ora definire δ, delta cappuccio che prende in ingresso uno stato e l'intera stringa w. Definisco ricorsivamente:

- **caso base:** se $|w| = 0$ ovvero se $W = \epsilon$ si ha: $\hat{\delta}(q, \epsilon) = \{q\}$
- **caso passo:** se $|w| > 0$, allora $W = xa$, $a \in \Sigma$ e $x \in \Sigma^*$. Posto $\hat{\delta}(q, x) = \{p_1, \cdots, p_k\}$ si ha: $\hat{\delta}(q, w) = \cup\delta(p_i, a)$

Per il linguaggio L accettato dall'automa si ha: $L(A) = \{w \in \Sigma^* | \hat{\delta}(q_0, q) \cap F \neq \emptyset\}$

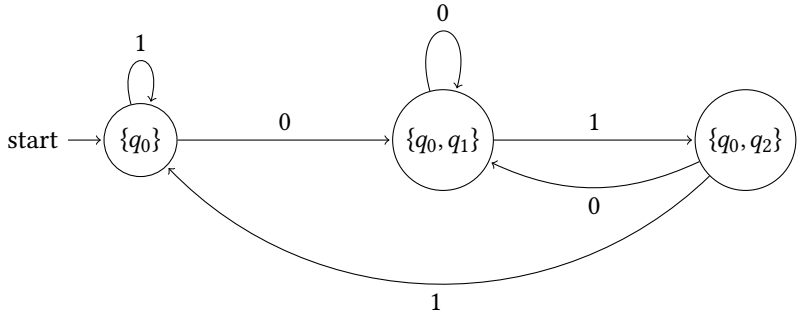
Esempio 3.25 Automa per $L = \{x010y | x, y \in \{0,1\}^*\}$ ovvero tutte le stringhe con dentro la sequenza 010:



Troviamo ora un algoritmo che trasformi un NFA in un DFA.
Dal penultimo esempio esempio ricavo:

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\to \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

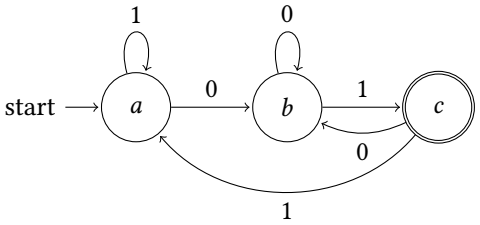
ovvero:



che è il DFA che si era anche prima ottenuto. Si hanno però dei sottoinsiemi mai raggiungibili. Si ha quindi:

	0	1
$\to \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

e definendo $\{q_0\} = a$, $\{q_0, q_1\} = b$ e $\{q_0, q_2\} = c$ si avrà:



Definiamo questo algoritmo che avrà:

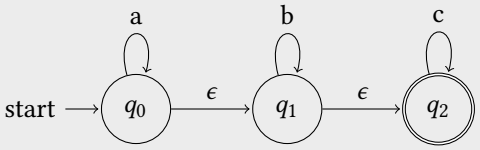
- come input un NFA $N = (Q_n, \Sigma, \delta_N, q_0, F_N)$
- come output un DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ tale che $L(D) = L(N)$

con:

- $Q_D = 2^{Q_N}$ (quindi se $Q_N = n$ si ha $|Q_D| = 2^n$)
- $F_D = \{S \subseteq Q_n | \, S \cap F_N \neq \emptyset\}$
- $\forall S \subseteq Q_N$ e $\forall a \in \Sigma$: $\delta_D(S, a) = \cup \delta_n(p, a)$ per esempio: $\delta_D(\{q_0, q_1, q_2\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_1, 0) \cup \delta_N(q_2, 0)$

Si definisce l'$\epsilon - NFA$, l'automa astati finiti non deterministici con ϵ transizioni. Con la transizione ϵ posso saltare i nodi, ovvero avanza senza aggiungere caratteri

Esempio 3.26 Si ha $ER = a^*b^*c^*$, che genera: $L = \{a^n b^m c^k | \, n, m, k \geq 0\}$ si ha:



ovvero con ϵ posso per esempio generare quante a voglio da q_0 e passare a q_2, uscendo senza generare altro

Si definisce la funzione $ECLOSE : Q \to 2^Q$, con $ECLOSE(q)$ insieme degli stati raggiungibili da q tramite $\epsilon - mosse$. Nell'esempio precedente si avrebbe: $ECLOSE(q_0) = \{q_0, q_1, q_2\}$ $ECLOSE(q_1) = \{q_1, q_2\}$ $ECLOSE(q_2) = \{q_2\}$ si ha inoltre che:

- $ECLOSE 2^Q \to 2^Q \;\; P \subseteq Q$
- $ECLOSE(P) = \cup ECLOSE(p)$
- $ECLOSE(\emptyset) = \emptyset$

mettendo in tabella l'esempio precedente si ha:

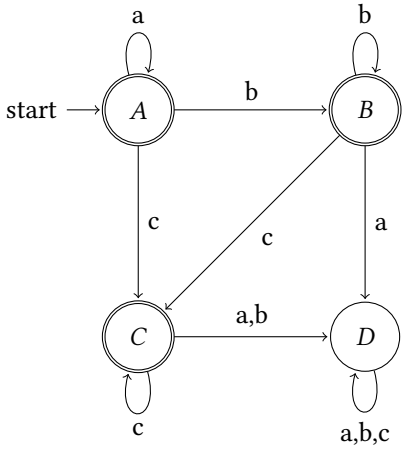
	a	b	c
$* \to \{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_1, q_2\}$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$	$\{q_2\}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

riscrivendo:

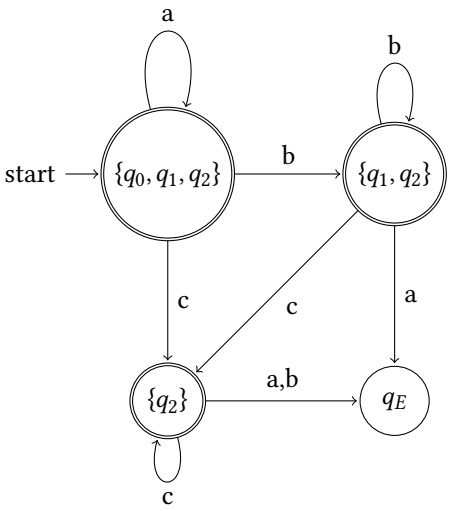
- $a = \{q_0, q_1, q_2\}$
- $b = \{q_1, q_2\}$
- $c = \{q_2\}$
- $d = \emptyset$

ovvero: $\delta_D(\{q_0,q_1,q_2\},a) = ECLOSE(\delta_N(q_0,a) \cup \delta_N(q_1,a) \cup \delta_N(q_2,a)) = ECLOSE(\{q_0\} \cup \emptyset \cup \emptyset) = ECLOSE(\{q_0\}) = ECLOSE(q_0) = \{q_0,q_1,q_2\}$ e $\delta_D(\{q_0,q_1,q_2\},B) = ECLOSE(\delta_N(q_0,b) \cup \delta_N(q_1,b) \cup \delta_N(q_2,b)) = ECLOSE(\emptyset \cup \{q_1\} \cup \emptyset) = ECLOSE(\{q_1\}) = ECLOSE(q_1) = \{q_1,q_2\}$ e $\delta_D(\{q_0,q_1,q_2\},c) = ECLOSE(\delta_N(q_0,c) \cup \delta_N(q_1,c) \cup \delta_N(q_2,c)) = ECLOSE(\emptyset \cup \emptyset \cup \{q_2\}) = ECLOSE(\{q_1\}) = ECLOSE(q_2) = \{q_2\}$

si ottiene quindi il seguente NFA:

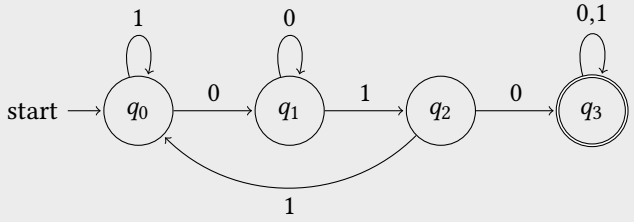


che diventa il seguente DFA:

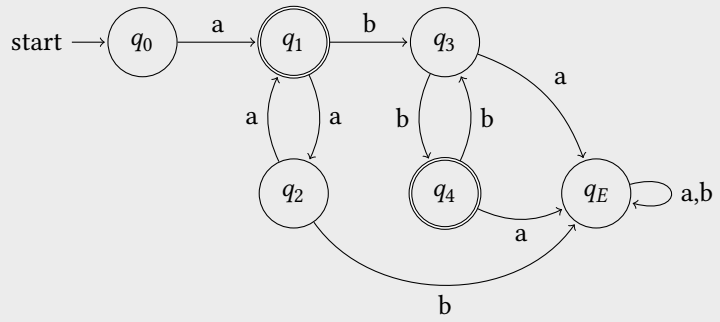


esercizi

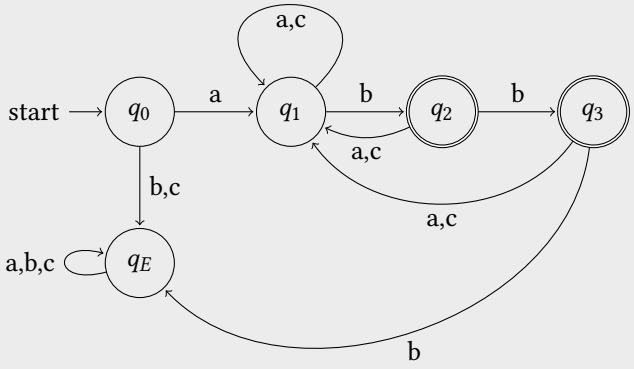
Esempio 3.27 automa DFA per $w = x010y,\ x,y \in \{0,1\}^*$:
la stringa più corta è 010



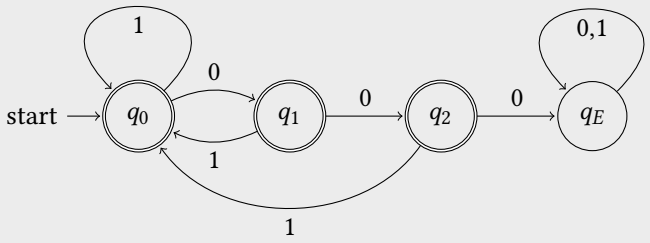
Esempio 3.28 automa DFA per $a^{2k+1}b^{2h}$, $h, k \geq 0$:
concatenazione di a dispari e b pari:



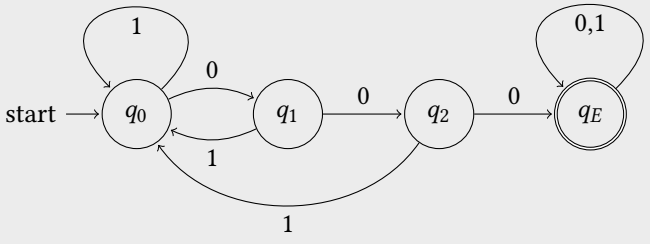
Esempio 3.29 cerco DFA per stringhe inizianti con a e finenti con b,
con occorrenze di b singole o a coppie, nessuna regola per c.
per esempio *abbcb* è nel linguaggio



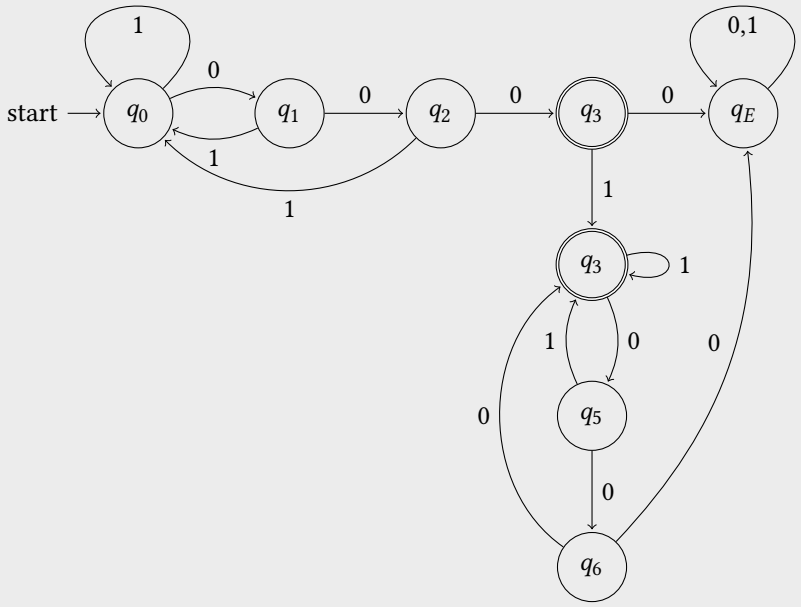
Esempio 3.30 cerco DFA per stringhe di bit non contegano 000



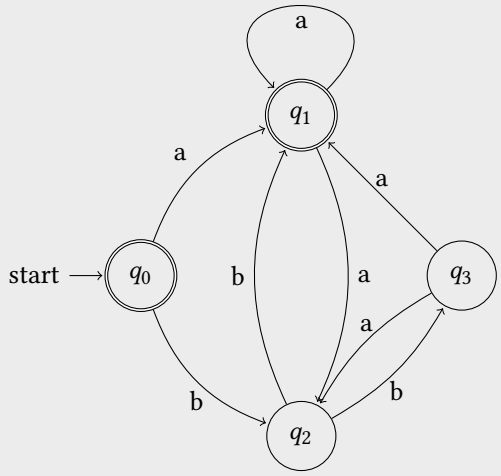
Esempio 3.31 cerco DFA per stringhe di bit non contegano 000 almeno
una volta



Esempio 3.32 cerco DFA per stringhe di bit che contengono 000 solo una volta



Esempio 3.33 Trasformare il seguente NFA in un DFA:



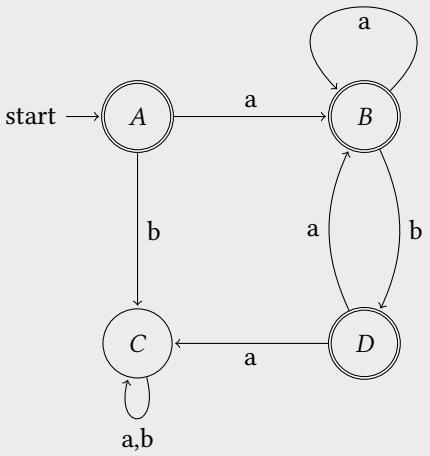
abbiamo quindi: $\delta_D(\{q_0\}, a) = \delta_N(q_0, a) = \{q_1, q_2\}$ $\delta_D(\{q_0\}, b) = \delta_N(q_1, b) = \emptyset$ $\delta_D(\{q_1, q_2\}, a) = \delta_N(q_1, a) \cup \delta_N(q_2, a) = \{q_1, q_2\} cup \emptyset = \{q_1, q_2\}$ $\delta_D(\{q_1, q_2\}, b) = \delta_N(q_1, b) \cup \delta_N(q_2, b) = \emptyset \cup \{q_1, q_3\} = \{q_1, q_3\}$ $\cdots$ ottengo quindi:

DFA	a	b
$* \rightarrow \{q_0\}$	$\{q_1, q_2\}$	$\emptyset$
$*\{q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_1, q_3\}$
$\emptyset$	$\emptyset$	$\emptyset$
$*\{q_1, q_3\}$	$\{q_1, q_2\}$	$\emptyset$

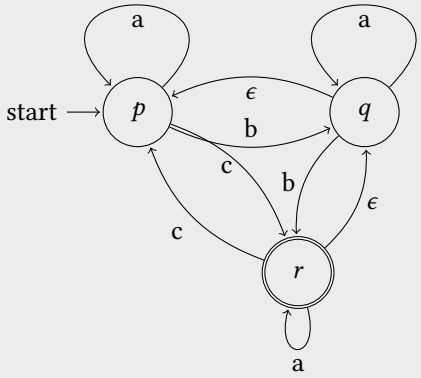
Posso ora rinominare:

- $A = \{q_0\}$
- $B = \{q_1, q_2\}$
- $C = \emptyset$
- $D = \{q_1, q_3\}$

ottengo quindi il seguente DFA:



Esempio 3.34 Trasformare il seguente ϵ-NFA in un DFA:



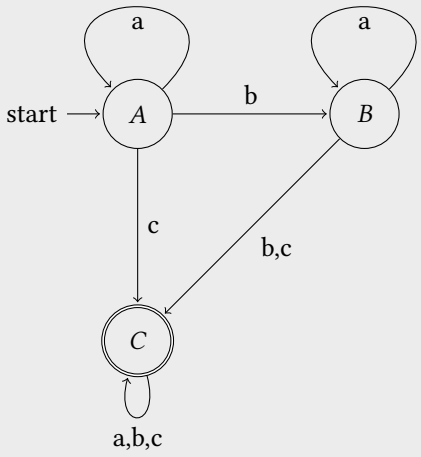
vediamo le ECLOSE: $ECLOSE(p) = \{p\}$ $ECLOSE(q) = \{p, q\}$ $ECLOSE(r) = \{p, q, r\}$ si ottiene quindi:

	a	b	c
$to\{p\}$	$\{p\}$	$\{p, q\}$	$\{p, q, r\}$
$\{p, q\}$	$\{p, q\}$	$\{p, q, r\}$	$\{p, q, r\}$
$*\{p, q, r\}$	$\{p, q, r\}$	$\{p, q, r\}$	$\{p, q, r\}$

infatti, per esempio: $\delta_D(\{p\}, a) = ECLOSE(\delta_(p, a)) = ECLOSE(\{p\}) = ECLOSE(p) = \{p\}$ $\delta_D(\{p, q\}, a) = ECLOSE(\delta_N(p, a) \cup \delta_N(q, a)) = ECLOSE(\{p\} \cup \{q\}) = ECLOSE(P) \cup ECLOSE(q) = \{p\} \cup \{p, q\} = \{p, q\} \cdots$ si hanno quindi le seguenti rinominazioni:

- $A = \{p\}$
- $b = \{p, q\}$
- $C = \{p, q, r\}$

ovvero:



torniamo a dare bene qualche definizione per δ in un DFA:

$$\delta : Q \times \Sigma^* \to Q$$

con: $q \in Q$, $w \in \Sigma^*$ e $\delta(p, w) = q$

- **caso base:** $w = \epsilon \to |w| = 0 \to \delta(q, \epsilon) = q$
  caso passo: $|w| \neq 0 \to w = ax$ con $a \in \Sigma, x \in \Sigma^*$: $\delta(q, w) = \delta(q, ax) = \delta(\delta(q, a), x)$

- **caso base:** $w = \epsilon \rightarrow |w| = 0 \rightarrow \delta(q, w) = \delta(q, \epsilon) = q$
  caso passo: $|w| \neq 0 \rightarrow w = xa$ con $a \in \Sigma, x \in \Sigma^*$: $\delta(q, w) = \delta(q, xa) = \delta(\delta(q, x), a)$

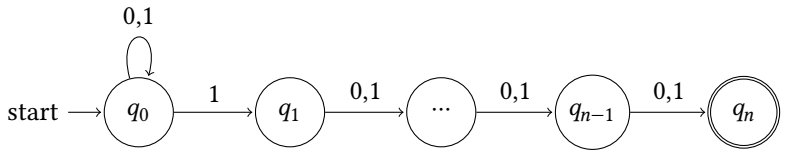
in un NFA si ha:

- **caso base:** $\delta(q, \epsilon = \{q\},\ \forall q \in Q$
- **caso passo:** posto $w = ax$ e $\delta(q, a) = \{p_1, \cdots, p_n$ allora $\delta(q, w) = \cup\delta(p_i, x)0\{r_1, \cdots, r_n\}$.
  $\delta(q, a) = p$
  $\delta(q, w) = \delta(q, xa) = \delta(p, x) = r$

oppure:

- **caso base:** $\delta(q, \epsilon) = \{q\},\ \forall q \in Q$
- **caso passo:** posto $w = xa$ si ha $\delta(q, q) = \delta(q, xa) = \cup\delta(p, a)$ con $\delta(q, x) = \{p_1, \cdots, p_n\}$

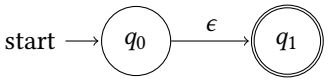
Se ho un NFA $N = (Q_N, \Sigma, \delta_N, q_{0_N}, F_N)$ con $\delta_N : Q \times \Sigma^* \rightarrow 2^{Q_N}$ che accetta un lingiaggio L posso ottenere un DFA $D = (Q_D, \Sigma, \delta_D, q_{0_D}, F_D)$ equivalente con $Q_D = 2^{Q_N}$ e $q_{0_D} = \{q_{0_N}\}$ che accetta L.
$\forall S \subseteq Q_N$ e $\forall a \in \Sigma$ si ha: $F_D = \{S \subseteq Q_N \,|\, S \cap F_n \neq \emptyset\}$ con $\delta_D(S, a) = \cup\delta_N(p, a)$
Si ha che: $|Q_D| = |2^{Q_N}| = 2^{|Q_N|}$ partiamo con un esempio:



che definisce un $L \subseteq \{0, 1\}^*$ tale che $w \in L$ sse n-simo elemento della fine è 1. Si ha: $|Q_N| = n + 1 \rightarrow |Q_D| = 2^n$ Si ha che dato un ϵ-NFA $E = (Q_E, \Sigma, \delta_E, q_{0_E}, F_E)$ che riconosce L in un NFA $N = (Q_N, \Sigma, \delta_N, q_{0_N}, F_N)$ equivalenti con $Q_N = 2^{Q_E}$ stati in Q_E ϵ-close: *ECLOSE(S)=S* e $q_N = ECLOSE(q_E)$: $F_N\{S \in Q_F,\ S \subseteq Q_E | S \cap F_E \neq \emptyset\}$ quindi: $\forall a \in \Sigma$ e $\forall S = \{p_1, \cdots, p_k\}, \forall S \in Q_F$ e $Q_E\, \delta_F(S, a)$ si ottiene $\cup\delta_E(p_i, a) = \{r_1, \cdots, r_n\}\, \delta_N(S, a) = ECLOSE(\{r_1, \cdots, r_n\})$

3.5 Da espressioni regolari a automi E-NFA

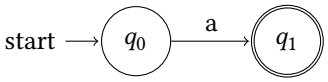
- **caso base**:
  1. se $R = \epsilon$ ovvero $L(R) = \{\epsilon\}$ allora:

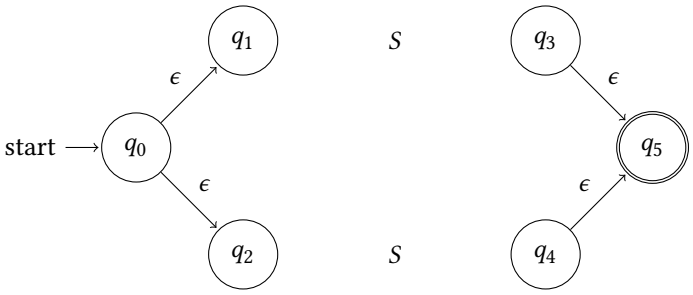
  2. se $R = \emptyset$ ovvero $L(R) = \emptyset$ allora:

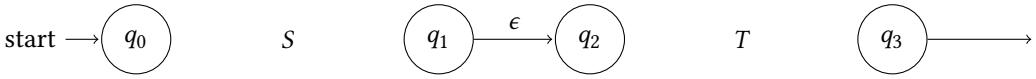
  3. se $R = a$ ovvero $L(R) = \{a\}$ allora:

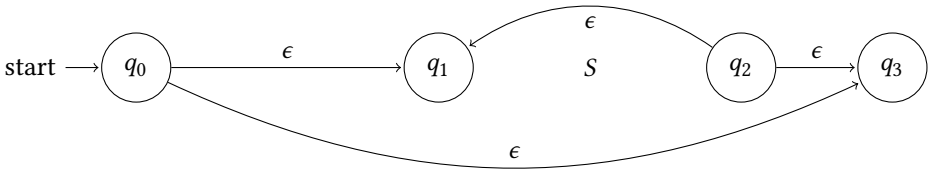
- **caso passo:**
  1. se $R = S + T$ quindi $L(R) = L(S) + L(T)$ allora:



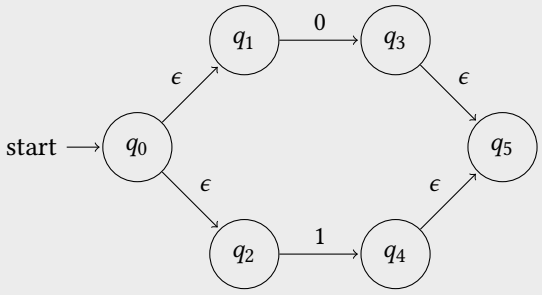
2. se $R = ST$ quindi $L(R) = L(S)L(T)$:



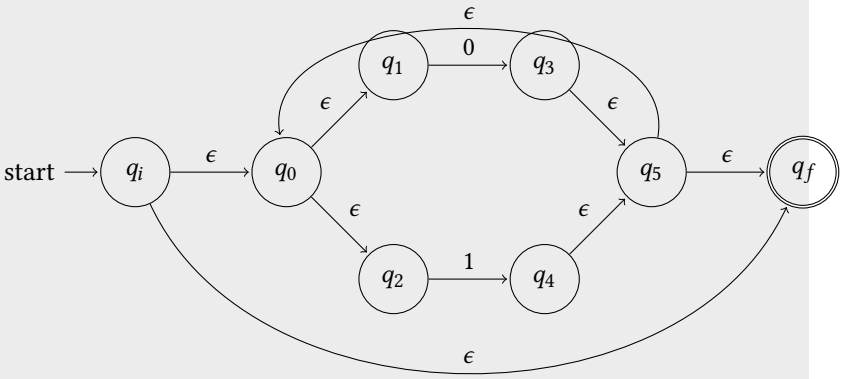
3. se $R = S^*$ quindi $L(R) = (L(S))^*$:



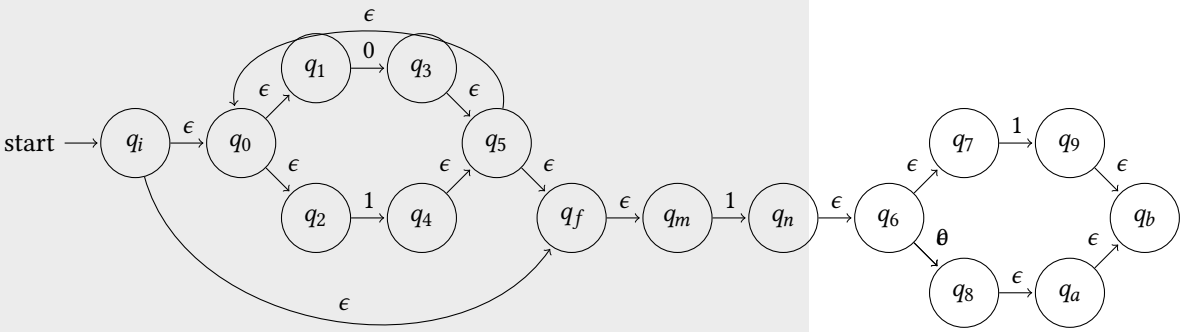
Esempio 3.35 creo E-NFA per $ER = (0 + 1)^*1(0 + 1)$ si ha che per 0+1:



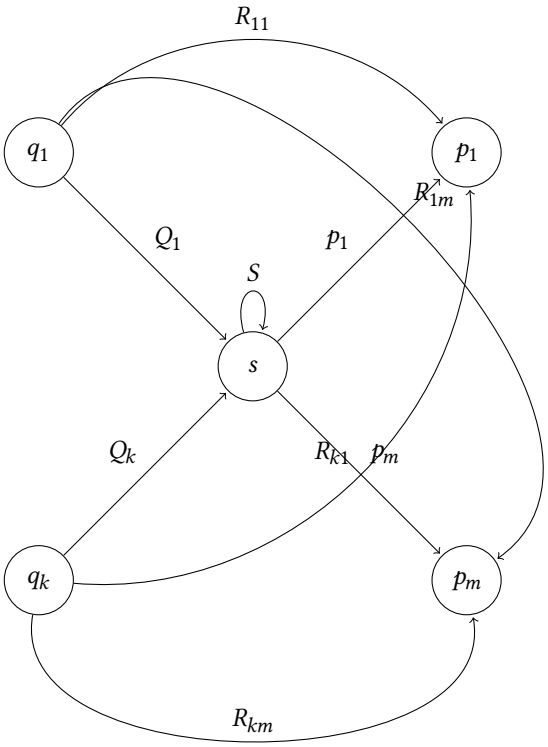
da qui ottengo $(0 + 1)^*$



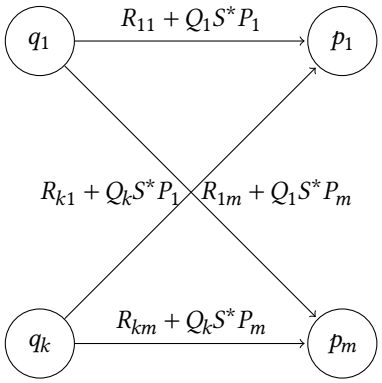
unisco e aggiungo uno nel mezzo:



Vediamo ora l'algoritmo che trasforma DFA in una ER. Questo algoritmo permette di verificare quale linguaggio è accettato o meno dall'automa: dato DFA $A = (Q, \Sigma, \delta, q_0, F)$ con Q e F stati non finali e Q, F, q_0 che sono i primi stati da eliminare. L'algoritmo procede per eliminazioni successive. Si costruisce quindi l'automa B che ha solo q_0 e $F = \{q_1, \cdots, q_k\}$. Scrivo quindi K espressioni regolari considerando solo q_0 e il k-esimo stato finale eliminando pian piano gli altri stati. Alla fine ottengo le varie espressioni regolari da unire: $E = E1 + E2 + \cdots + Ek$ vediamo ora come eliminare uno stato. Partiamo dal seguente automa:

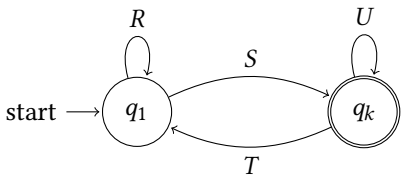


ed elimino s:

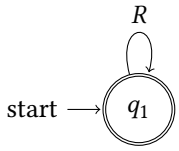


quindi quando si ha uno stato iniziale q_0 e uno finale q_1:

- se $q_0 \neq q_1$:

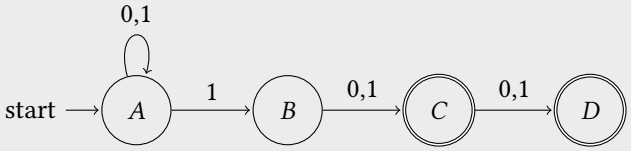


  rappresenta: $E_i = (R + SU^*T)^*SU^*$
- se $q_0 = q$:

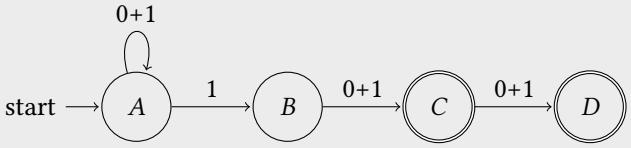


rappresenta: $E_i = R^*$

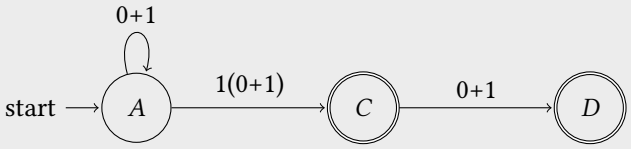
Esempio 3.36 passo dall'automa a l'espressione regolare:



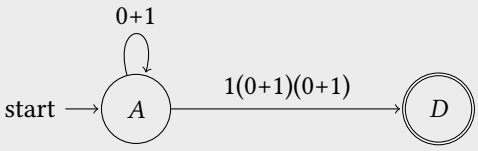
questo automa accetta le stringhe binarie con un uno in penultima o terzultima posizione.
Rietichetto con le ER:



elimino B che non è iniziale o finale . Il predecessore di B è A e il successore è C. Si ha quindi: $A \to B : 1\ B \to C : 0+1$ e non ho loop in B ($\emptyset$) e nessun arco tra A e C ($R_{a,c} = \emptyset$). Quindi: $R'_{A,C} = \emptyset + 1\emptyset^*(0+1) = 1(0+1)$ ovvero:

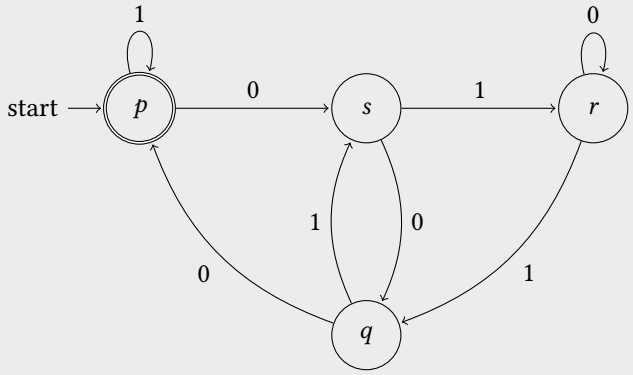


Ho ora solo stati iniziali e finali, quindi $E = E_1 + E_2$. Trovo $E1$, elimino quindi C. Si ha: $A \to C : 1(0+1)\ C \to D : 0+1$ e non ho loop in B ($\emptyset$) e nessun arco tra A e D($R_{A,D} = \emptyset$), ovvero: $R_{A,D} = \emptyset + 1(0+1)\emptyset^*(0+1) = 1(0+1)(0+1)$ che è:

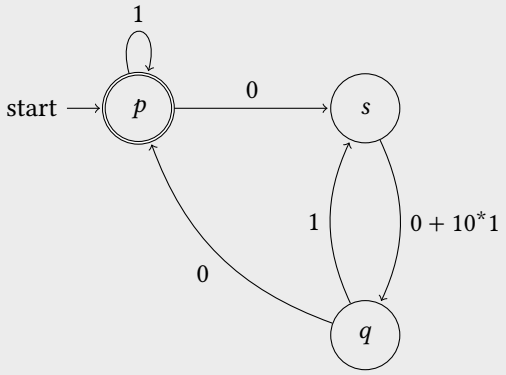


quindi: $E_1 = ((0+1) + 1(0+1)(0+1)\emptyset^*\emptyset)^*1(0+1)(0+1)\emptyset^* = (0+1)^*1(0+1)(0+1)$ ottengo $E2$ eliminando D che non ha successori, quindi: $E_2 = (0+1)^*1(0+1)$ e quindi infine: $E = E_1 + E_2 = (0+1)^*1(0+1)(0+1) + (0+1)^*1(0+1) = (0+1)^*1(0+1)(\epsilon + 0 + 1)$

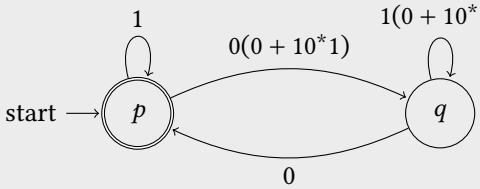
Esempio 3.37



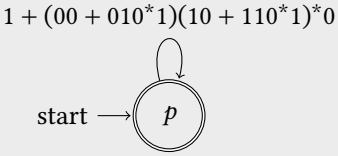
rimuovo r: $s \to r : 1$ $r \to q : 1$ $loop : 0$ $R_{s,q} = 0$ Si ottiene quindi: $R'_{s,q} = 0 + 10^*1$ ovvero:



elimino ora s, che ha due predecessori, p e q, quindi: $p \to s : 0$ $q \to s : 1$ $s \to q : 0 + 10^*1$ non ha loop e non ha archi diretti tra p e q ($R_{p,q} = \emptyset$). Si ottengono quindi: $R_{p,q} = \emptyset + 0\emptyset^*(0 + 10^*1) = 0(0 + 10^*1)$ $R_{q,q} = \emptyset + 10^*(0 + 10^*1) = 1(0 + 10^*1)$



elimino ora q che ha p sia come predecessore che come successore: $p \to q : 0(0 + 10^*1)$ $q \to p : 0$ $loop : 1(0 + 10^*1)$ $R_{p,p} : 1$ ottenendo: $R'_{p,p} = 1 + 0(0 + 10^*1)(1(0 + 10^*1))^*0 = 1 + (00 + 010^*1)(10 + 110^*1)^*0$
ovvero:

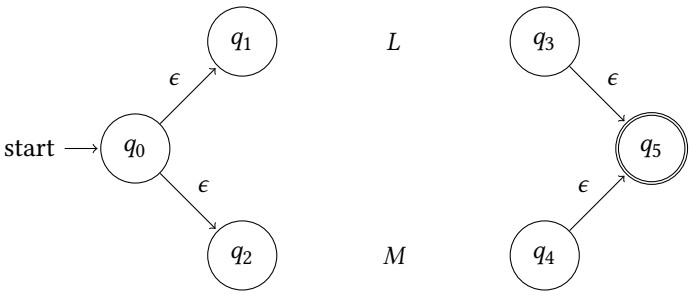


Essendo p sia iniziale che finale si ha: $E = (1 + (00 + 010^*1)(10 + 110^*1)^*0)^*$

3.6 Chiusura di un Linguaggio regolare

Sia *REG* la classe dei linguaggi regolari (ovvero ogni linguaggio regolare su un alfabeto finito non vuoto). Si ha la seguente proprietà: $L, M \in REG \rightarrow L \cup M \in REG$ si può dimostrare in due maniere:

- **con le espressioni regolari:** $L, M \in REG \rightarrow \exists R, S$ ER tali che $L(R) = L$ e $L(S) = M\, L \cup M = L(R + S)$ e quindi appartenente a *REG*
- **con gli automi:** se $L, M \in REG \rightarrow \exists \epsilon{-}NFA$ che con L e M va dallo stato iniziale a quello finale:



accetta $L \cup M$ che quindi è *REG*

Inoltre siano due i due alfabeti $\Sigma \subseteq \Gamma$, si ha che: se L è regolare su $\Sigma \rightarrow$ L è regolare su Γ inoltre: se L è REG su Σ_1, M è REG su Σ_2, allora $L \cup M$ è REG su $\Sigma_1 \cup \Sigma_2$

Teorema 3.1 Si ha che:
Se L e M sono linguaggi regolari allora LM, ovvero la concatenazione è regolare.
Se L è un linguaggio regolare allora L^* è regolare

Teorema 3.2 se $L \in REG$ su Σ allora $\overline{L} = \Sigma^* - L \in REG$

Dimostrazione. se $L \in REG$ allora $\exists$ DFA $A = (Q, \Sigma, \delta, q_0, F)$ tale che $L(A) = L$. Costruisco $B = (Q, \Sigma, \delta, q_0, Q - F)$, $L(B) = \overline{L}$.

□

si osserva che: $L \in REG$ *su* Σ *e* $\Sigma \subseteq \Gamma$ inoltre: $\overline{L} = \Sigma^* - L$ *oppure* $\overline{L} = \Gamma^* - L$
Se ho un espressione regolare per L e voglio ottenere un'espressione regolare per $\overline{L}$ *devo ottenere un* $\epsilon - NFA$ *che devo convertire in DFA. A quel punto complemento con F e ottengo un'espressione regolare per* $\overline{L}$.
Passiamo ora all'intersezione:

Teorema 3.3 se L e M sono regolari allora la loro intersezione è regolare

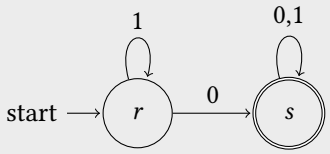
Dimostrazione. $L, M \in REG \rightarrow \exists A_L, A_M(DFA)$ tali che $L(A_L) = L$ *e* $L(A_M) = M$ e quindi:

TIKZ

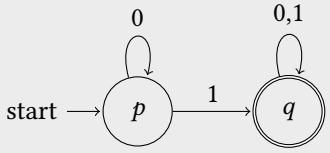
□

Passiamo al problema della **chiusura dei linguaggi regolari rispetto all'intersezione**, ovvero che due automi che usano entrambi contemporaneamente una certa stringa in input, il ché non è possibile. Si risolve usando l'**automa prodotto**: $A = (Q_L \times Q_M, \Sigma, \delta, (q_{OL}, q_{OM}).F_L \times F_M)$ con: $\delta(p, q), a) = (\delta_L(p, a), \delta_M(q, a))$ $A_L = (Q_L, \Sigma, \delta_L, q_{OL}, F_L)$ $A_M = (Q_M, \Sigma, \delta_M, q_{OM}, F_M)$

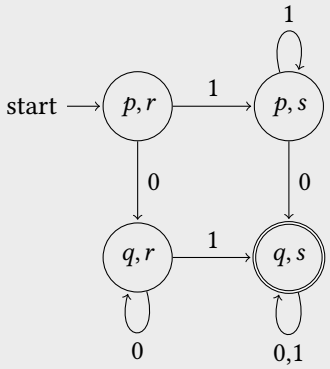
Esempio 3.38 Siano:
A_L, stringhe binarie con almeno uno zero:



A_M, stringhe binarie con almeno un uno:

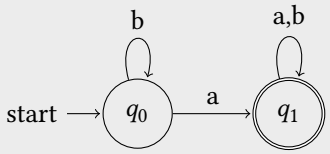


ottengo l'automa prodotto:

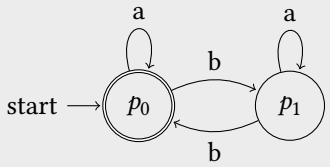


$\delta((p,r),0) = (\delta_L(p,o), \delta_M 8r, o)) = (q,r)$

Esempio 3.39 Siano:
A, stringhe contenenti a e b contenenti almeno una b:



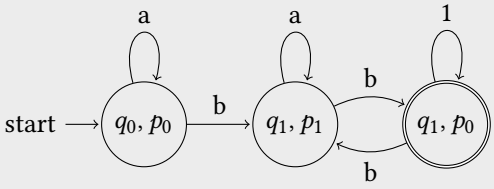
A_M, stringhe contenenti a e b contenenti $2n$, $n \geq 0$ b:



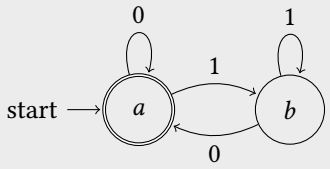
si ha che $L(A \wedge B)$ è il linguaggio formato da stringhe contenenti a e b contenenti $2n$, $n \geq 1$ b: faccio la tabella:

	a	b
$\rightarrow (q_0, p_0)$	(q_0, p_0)	(q_1, p_1)
(q_1, p_1)	(q_1, p_1)	(q_1, p_0)
$*(q_1, p_0)$	(q_1, p_0)	(q_1, p_1)

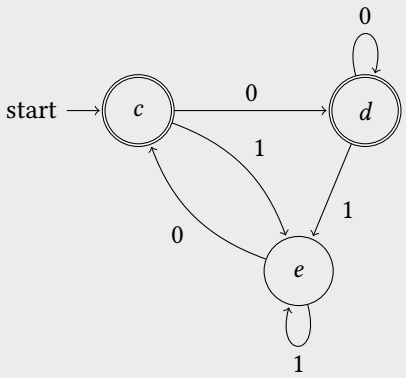
ottengo l'automa prodotto:



Esempio 3.40 si hanno due linguaggi $L(A)$ e $L(B)$ tali per cui la loro intersezione non è vuota.
Sia A:



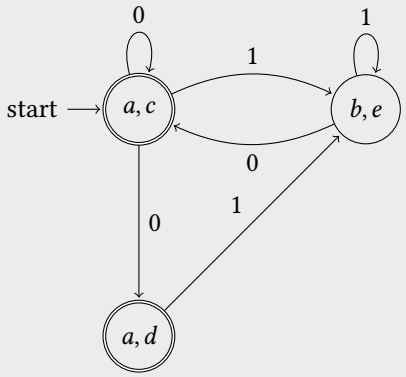
e B:



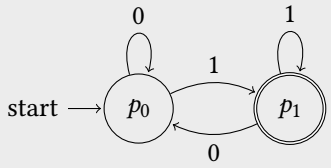
faccio la tabella:

	0	1
$\overset{*}{\rightarrow}(a,c)$	(a,d)	(b,e)
$*(a,d)$	(a,d)	(b,e)
(b,e)	(a,c)	(b,e)

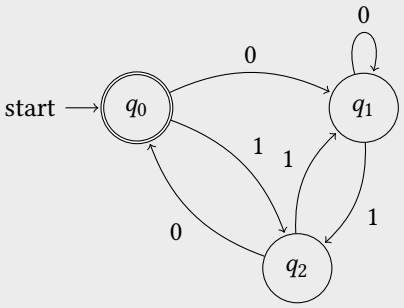
e ottengo:



Esempio 3.41 Sia A:



e B:



faccio la tabella:

	0	1
$\rightarrow (p_0, q_0)$	(p_0, q_1)	(p_1, q_2)
(p_0, q_1)	(p_0, q_1)	(p_1, q_2)
(p_1, q_2)	(p_0, q_0)	(p_1, q_1)
(p_1, q_1)	(p_0, q_1)	(p_1, q_2)

non si raggiungono stati finali quindi $L(A \cap B) = \emptyset$

3.7 Minimizzazione

definiamo la **relazione di equivalenza tra stati**:

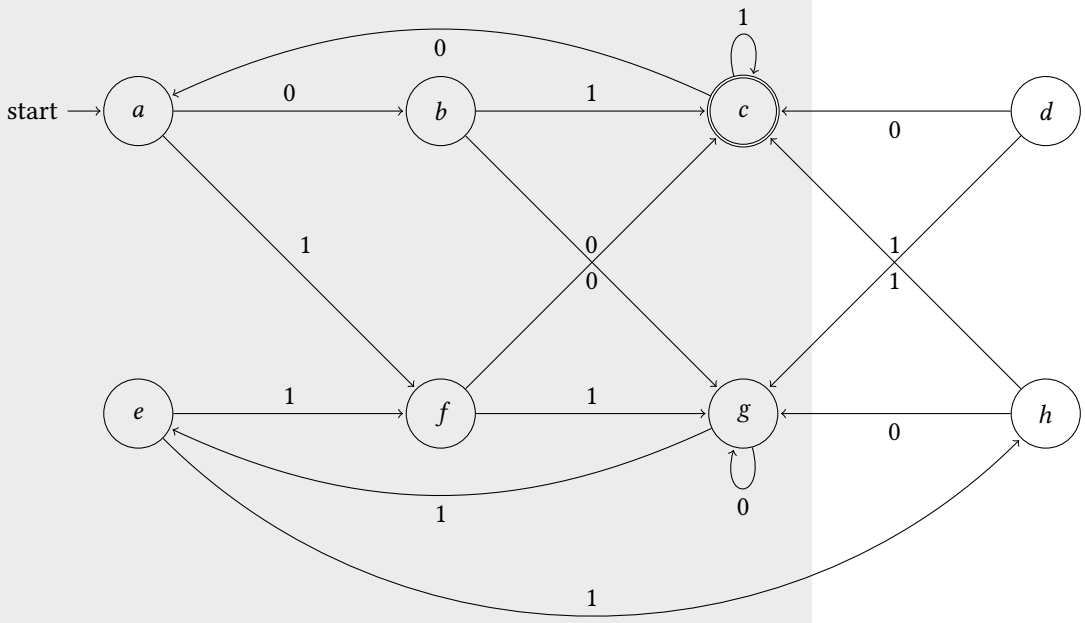
Definizione 3.5 Sia A un DFA, $A = ()Q, \Sigma, \delta, q_0, F)$. Siano: $p, q \in Q \rightarrow p \approx q$ vale se $\forall w \in \Sigma^* \quad \hat{\delta}(p, w) \in F \longleftrightarrow \hat{\delta}(q, w) \in F$ inoltre p e q sono distinguibili se $p \not\approx q$ ovvero: $\exists w \in \Sigma^*$ tale che: $\hat{\delta}(p, w) \in F \; e \; \hat{\delta}(q, w) \notin F \; o \; \hat{\delta}(p, w) \notin F \; e \; \hat{\delta}(q, w) \in F$

quindi si ha una **relazione di equivalenza** e quindi si hanno le tre proprietà:

- **riflessività:** $\forall p \in Q, p \approx p$
- **simmetricità:** $\forall p, q \in Q, p \approx q \rightarrow q \approx p$
- **transitività:** $\forall p, q, r \in Q, p \approx q \vee q \approx r \rightarrow p \approx r$

e quindi **in ogni classe di equivalenza ci sono stati tra loro equivalenti** due stati, uno finale e uno non finale, sono sicuramente distinguibili, basti pensare alla stringa vuota.

Esempio 3.42 Analizziamo il seguente automa:



cerco di capire se a e g sono equivalenti:

- ϵ non li distingue perché entrambi sono non accettanti
- 0 non li distingue perché li porta rispettivamente in b e g, che non sono accettanti
- 1 non li distingue perché li porta rispettivamente in f e e, che non sono accettanti
- 01 li distingue perché $\hat{\delta}(a, 01) = c$ e $\hat{\delta}(g, 01) = e$ e il primo è accettante mentre il secondo no, quindi i due stati non sono equivalenti

si verifica che invece a e e sono equivalenti

diamo una definizione ricorsiva dell'equivalenza tra stati:

- **caso base:** se $p \in F$ e $q \notin F$ o viceversa si ha che gli stati sono distinguibili
- **caso passo:** se per $a \in \Sigma$ gli stati $r = \delta(p, a)$ e $s = \delta(q, a)$ sono distinguibili allora anche p e q lo sono

quindi se un certo arco w porta due stati a due diversi di cui uno è accettante essi sono distinti.
Esiste anche un algoritmo detto **riempi-tabella**:
partiamo dalla seguente tabella

b		/	/	/	/	/	/
c			/	/	/	/	/
d				/	/	/	/
e					/	/	/
f						/	/
g							/
h							
	a	b	c	d	e	f	g

c è accettante quindi segniamo tutte le caselle relative a c per il caso base:

b		/	/	/	/	/	/
c	x	x	/	/	/	/	/
d			x	/	/	/	/
e			x		/	/	/
f			x			/	/
g			x				/
h			x				
	a	b	c	d	e	f	g

poiché, per esempio c, h è distinguibile e gli stati e e f con 0 vanno in h e c si ha che anche e e f sono distinguibili, segno quindi nelle caselle appropriate. Proseguo controllando anche le altre coppie e riempio le caselle. Ottengo:

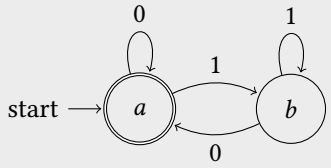
b	x	/	/	/	/	/	/
c	x	x	/	/	/	/	/
d	x	x	x	/	/	/	/
e		x	x	x	/	/	/
f	x		x	x	x	/	/
g	x	x	x	x	x	x	/
h	x		x	x	x	x	x
	a	b	c	d	e	f	g

quindi si ha solo che: $a \approx e$ $b \approx h$ $d \approx f$ si ha che:

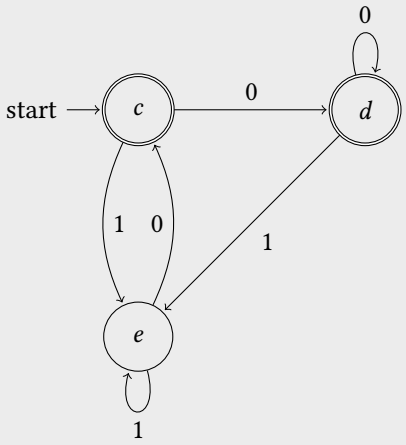
Teorema 3.4 due stati non distinti dall'algoritmo riempi-tabella sono equivalenti. Se ne calcola la complessità: $|Q| = n \to \frac{n(n-1)}{2}$ *caselle* $= O(n^2)$ se ho una crocetta a iterazione ho il caso peggiore $n^2 n^2 = O(n^4)$

quindi per vedere se due linguaggi regolari sono equivalenti si ha che:
$L, M \in REG$ $\exists A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ $\exists A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$ costruisco $A = (Q_L \cup Q_M, \Sigma, \delta, q_L, F_L \cup F_M)$
δ è δ_L per Q_L e δ è δ_M per Q_M per vedere se $q_L \approx q_M$ uso il riempi tabella e se sono equivalenti si ha che $L = M$

Esempio 3.43 Sia A:



e B:



entrambi i linguaggi accettano $(\epsilon + (0+1)^*0)$ applico la tabella:

b	x	/	/	/
c		x	/	/
d		x		/
e	x		x	x
	a	b	c	d

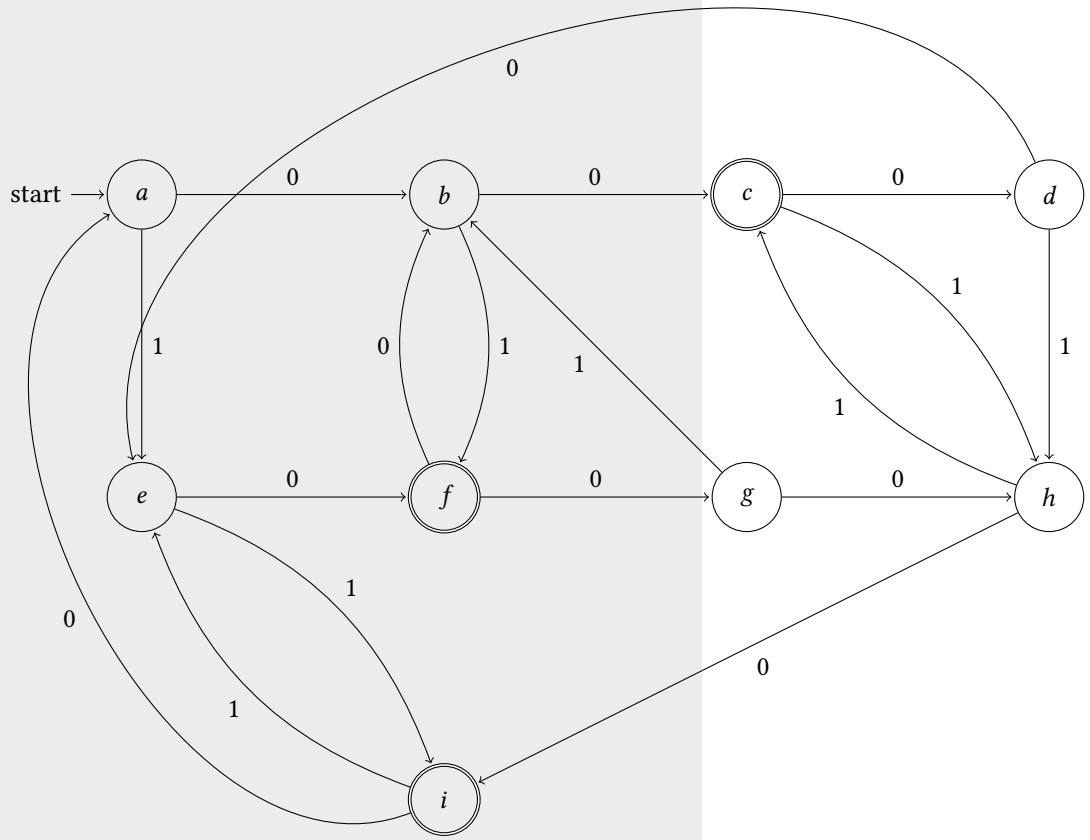
e quindi dato che $a \approx c$ si ha che i due linguaggi sono lo stesso linguaggio.

passiamo ora alla **minimizzazione**, che prende in input un DFA A e restituisce un $DFA\ A_{min}$ tale che $L(A) = L(A_{min})$ e che A_{min} ha il numero più piccolo possibile di stati per distinguere $L(A)$.
Procedo così:

1. si rimuovono gli stati non raggiungibili
2. applico la tabella per scoprire le tabelle di equivalenza, gli stati del nuovo automa sono le classi di equivalenza

si ha che lo stato iniziale è la classe di equivalenza dello stato finale e gli stati finali sono le classi di equivalenza degli stati finali.
La minimizzazione si dimostra per assurdo:
Sia M il DFA ottenuto dalla tabella. Suppongo esista un DFA N tale che $L(A) = L(N)\, |Q_N| < |Q_M|$. I due stati iniziali sono indistinguibili. Sia $p \in M\ q \in N$ tali $p \approx q\ \forall a \in \Sigma$ quindi $\delta p, a) = \delta(q, a)$, Ogni stato $p \in Q$,è quindi indistinguibile da almeno uno stato di N. Avendo però N meno stati si avranno almeno due stati di M indistinguibili dallo stesso di N ma non sono indistinguibili per la tabella. SI ha un assurdo.

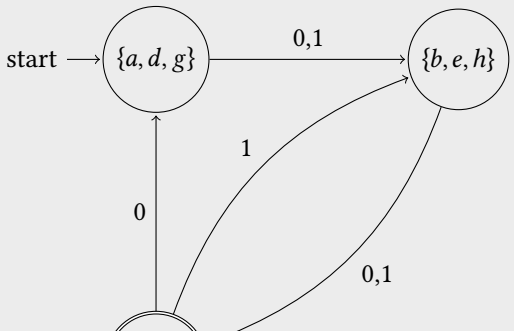
Esempio 3.44 Si ha il seguente automa:



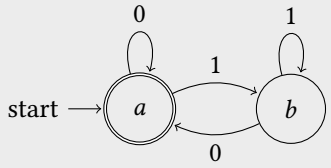
faccio la tabella

b		/	/	/	/	/	/	/
c	x	x	/	/	/	/	/	/
d		z	x	/	/	/	/	/
e	x		x	x	/	/	/	/
f	x	x		x	x	/	/	/
g		x	x		x	x	/	/
h	x		x	x		x	x	/
i	x	x		x	x		x	x
	a	b	c	d	e	f	g	h

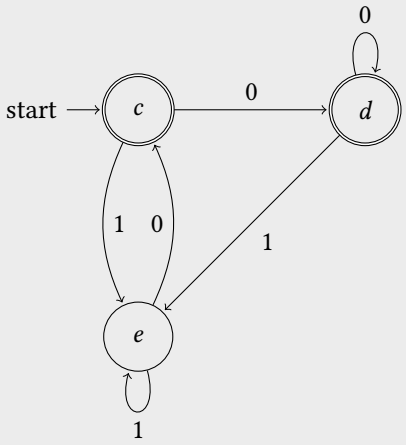
al caso base segno tutte le caselle degli stati accettanti tranne quelle che incrociano con altri stati accettanti e poi completo il resto della tabella.
Ottengo: $a \approx d \approx g\ b \approx e \approx h\ c \approx f \approx i$
e quindi ottengo, l'automa minimizzato:



Esempio 3.45 Sia A:



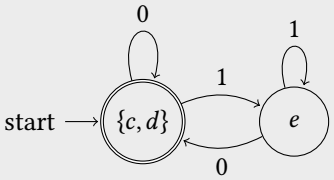
e B:



riduco B:

d		/
e	x	x
	c	d

quindi $c \approx d$. Ottengo quindi:



quindi A è B minimizzato, per questo riconoscevano lo stesso linguaggio

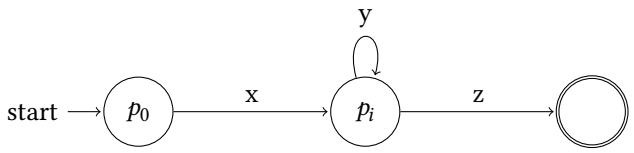
La tabella non funziona ovviamente por gli NFA

3.8 Pumping Lemma per i linguaggi regolari

Questo lemma serve a dimostrare che un linguaggio L non è regolare. Si procede per assurdo.
Partiamo da un esempio: $L_{01} = \{0^n 1^n | \, n \geq 1\} = \{01, 0011, \cdots\}$ supponiamo che questo linguaggio sia rappresentabile da un DFA A con k stati ($L(A) = L_{01}$).
0^k implica $k + 1$ prefissi: $\epsilon, 0, 00, 000, \cdots, 0^k$ e quindi $\exists i, j$ tali che 0^i e 0^j finiscono nello stesso stato. Allora l'automa è ingannabile e accetterebbe $0^i 0^j$ con $i \neq j$.
Formalmente si ha:

Teorema 3.5 Sia $L \in REG$ allora $\exists n$ che dipende da L tale che $\forall w \in L$ con $|w| \geq n$, w può essere scomposta in tre stringhe $w = xyz$ in modo che: $y \neq \epsilon \; |xy| \leq n \; \forall k \geq 0 \;\; zy^k z \in L$ in altre parole posso sempre trovare una stringa non vuota y non troppo distante dall'inizio di w, da *replicare* da ripetere o cancellare ($k = 0$) senza uscire dal linguaggio L

Dimostrazione. essendo $L \in REG$ esiste un DFA A tale che $L = L(A)$. Suppongo $|Q| = n$ e considero $w = a_1 a_2 \cdots a_n$ con $m \geq n$. Sia: $p_i = \hat{\delta}(q_0, a_1, \cdots a_i) \; \forall i = 0, 1, \cdots, n$ quindi: $p_0 = q_0, p_1, \cdots, p_n \rightarrow n + 1$ *stati* allora $\exists i, j$, con $0 \leq i < j \leq n$ tali che $p_i = p_j$. Scompongo ora w in $w = xyz$ con: $x = a_1 a_2 \cdots a_n \; y = a_{i+1} a_{i+2} \cdots a_j \; z = a_{j+1} a_{j+2} \cdots a_m$



□

Esempio 3.46 mostriamo che L01 non è regolare. Suppongo per assurdo che sia regolare, allora vale il pumping lemma. Sia $n \in \mathbb{N}$ la costante del pumping lemma. Pongo $w = 0^n 1^n = xyz$ tale che $|xy| = n$, ovvero $|xy| = 0^n$, quindi xy è formato da soli 0. Poniamo $x = 0^{n-1} \;\; y = 0 \neq \epsilon \; z = 1^n$. Però per il pumping lemma $\forall k \geq 0 \;\; xy^k z \in L_{01}$. $xy^k z = 0^{n-1} 0^k 1^n = 0^{n+k-1} 1^n \in L_{01}$ ma se $k \neq 1$ ciò non è vero e quindi il linguaggio non è regolare

Esempio 3.47 $L = \{w \in \{0,1\}^* | \, w = w^R\}$, linguaggio delle stringhe palindrome, è regolare?
suppongo per assurdo di sì, con n costante del pumping lemma.
$w = 0^n 10^n$ quindi $x = 0^{n-1} \;\; y = 0 \;\; z = 10^n$. Per $k = 0$ si ha $xy^k z = 0^{n-1} 10^n \notin L$, quindi non è regolare

Esempio 3.48 $L = \{1^p | \; p \; primo\} = \{11, 111, 11111\}$ è regolare?
$w = 1^p = 1^{n-1} 11^{p-n}$ con: $x = 1^{n-1} \;\; y = 1 \;\; z = 1^{p-n}$ per $k = 0$ si ha 1^{p-1} e $p - 1$ non è , quindi non è regolare

Esempio 3.49 $L = \{0^n 1^m | \, n \leq m\}$ è regolare?
$w = 0^{n-l} 0^l 1^m$ con: $x = 0^{n-l} \;\; y = 0^l \;\; z = 1^m$ con $|y| = l$ e $0 < l \leq n$
quindi $\forall k \geq 0$ si ha $xy^k z \in L = 0^{n-l} 0^{kl} 1^m$ con: $x = 0^{n-l} \;\; y = 0^{kl} \;\; z = 1^m$
scelgo k tale che $b - l + kl = n + (k-1)l > m \rightarrow k > 1 + \frac{m-n}{l}$ quindi non è regolare

Esempio 3.50 $L = \{0^n 1^m | \, n \geq m\}$ è regolare?
$w = 0^{n-l} 0^l 1^m$ con: $x = 0^{n-l} \;\; y = 0^l \;\; z = 1^m$ con $|zy| \leq n \; y \neq \epsilon \; |y| = l$ e $0 < l \leq n$
scelgo l tale che $n - l < m$ quindi non è regolare

Linguaggi liberi dal contesto 4

4.1 Grammatiche Context-Free

Esempio 4.1 Sia $G = (V, T, O, E)$, con $V = \{E, I\}$ e $T = \{a, b, 0, 1, (,), +, *\}$ quindi ho le seguenti regole, è di tipo 3:

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. $I \to I1$

voglio ottenere $a * (a + b00)$ sostituisco sempre a destra (right most derivation)

$$E \to E * E \to E * (E) \to E * (E + E) \to E * (E + I) \to E + (E + I0)$$

$$\to R + (I + b00) \to E * (a + b00) \to I * (a + b00) \to a * (a + b00)$$

usiamo ora *l'inferenza ricorsiva*:

passo	stringa ricorsiva	var	prod	passo stringa impiegata
1	a	I	5	\
2	b	I	6	\
3	b0	I	9	2
4	b00	I	9	3
5	a	E	1	1
6	b00	E	1	4
7	a+b00	E	2	5,6
8	(a+b00)	E	4	7
9	a*(a+b00)	E	3	5, 8

definisco formalmente la derivazione $\to$:

Definizione 4.1 Prendo una grammatica $G = (V, T, P, S)$, grammatica CFG. Se $\alpha A \beta$ è una stringa tale che $\alpha, \beta \in (V \cup T)^*$, appartiene sia a variabili che terminali. Sia $A \in V$ e sia $a \to \gamma$ una produzione di G. Allora scriviamo:

$$\alpha A \beta \to \alpha \gamma \beta$$

con $\gamma \in (V \cup T)^*$.
Le sostituzioni si fanno indipendentemente da α e β. Questa è quindi la definizione di derivazione.

Definizione 4.2 Definisco il simbolo $\rightarrow_*$, ovvero il simbolo di *derivazioni in 0 o più passi.* Può essere definito in modo ricorsivo. Per induzione sul numero di passi.

- la base dice che $\forall \alpha \in (V \cup T)^*$, $\alpha \rightarrow * \alpha$
- il passo è: se $\alpha \rightarrow_{G_*} \beta$ e $\beta \rightarrow_{G_*} \gamma$ allora $\alpha \rightarrow_* \gamma$

Si può anche dire che $\alpha \rightarrow_{G_*} \beta$ sse esiste una sequenza di stringhe $\gamma_1, ..., \gamma_n$ con $n \geq 1$ tale che $\alpha = \gamma_1$, $\beta = \gamma_n$ e $\forall i$, $1 < i < n-1$ si ha che $\gamma_1 \rightarrow \gamma_{i+1}$ la derivazione in 0 o più passi è la chiusura transitiva della derivazione

Definizione 4.3 avendo ora definito questi simboli possiamo definire una forma sentenziale. Infatti è una stringa α tale che:

$$\forall \alpha \in (V \cup T)^* \text{ tale che } S \rightarrow_{G_*} \alpha$$

Definizione 4.4 data $G = (V, T, P, S)$ si ha che $L(G) = \{w \in T^* | S \rightarrow_{G_*} w\}$ ovvero composto da stringhe terminali che sono derivabili o 0 o più passi.

Esempio 4.2 formare una grammatica CFG per il linguaggio:

$$L = \{0^n 1^n | n \geq 1\} = \{01, 0011, 000111, ...\}$$

con x^n intendo una concatenazione di n volte x (che nel nostro caso sono 0 e 1).
posso scrivere:

$$0^n 1^n = 00^{n-1} 1^{n-1} 1$$

il nostro caso base sarà la stringa 01, Poi si ha: $G = (V, T, P, S)$, $T = \{0, 1\}$, $V = \{S\}$, il caso base $S \rightarrow 01$ e $S \rightarrow 0S1$ il caso passo è quindi: se $w = 0^{n-1} 1^{n-1} \in L$ allora $0w1 \in L$.
Ora voglio dimostare che $000111 \in L$, ovvero $S \rightarrow * 000111$:

$$S \rightarrow 0S1 \rightarrow 00S11 \rightarrow 000S111$$

Teorema 4.1 data la grammatica $G = \{V, T, P, S)$ CFG e $\alpha \in (V \cup T)^*$. Si ha che vale $S \rightarrow_* \alpha$ sse $S \rightarrow_{lm_*} \alpha$ sse $S \rightarrow_{rm_*} \alpha$. Con $\rightarrow_{lm_*}$ simbolo di *left most derivation* e $\rightarrow_{rm_*}$ simbolo di *right most derivation*

Esempio 4.3 formare una grammatica CFG per il linguaggio:

$$L = \{0^n 1^n | n \geq 0\} = \{\varepsilon, 01, 0011, 000111, ...\}$$

stavolta abbiamo anche la stringa vuota. Il caso base stavolta è $S \rightarrow \varepsilon | 0S1$

Esempio 4.4 Fornisco una CFG per $L = \{a^n | n \geq 1\} = \{a, aa, aaa, ...\}$. La base è a
il passo è che se $a^{n-1} \in L$ allora $a^{n-1} a \in L$ (o che $aa^{n-1} \in L$).
Si ha la grammatica $G = \{V, T, P, S)$, $V = \{S\}$, $T = \{a\}$ e si hanno $S \rightarrow a | Sa$ (o $S \rightarrow a | aS$). Dimostro che $a^3 \in L$.

$$S \rightarrow Sa \rightarrow Saa \rightarrow aaa$$

oppure

$$S \rightarrow aS \rightarrow aaS \rightarrow aaa$$

Esempio 4.5 trovo una CFG per $L = \{(ab)^n | n \geq 1\} = \{ab, abab, ababab, ...\}$
La base è ab
il passo è che se $(ab)^{n-1} \in L$ allora $(ab)^{n-1}ab \in L$.
Si ha la grammatica $G = \{V, T, P, S\}$, $V = \{S\}$, $T = \{a, b\}$ (anche se in realtà $T = \{ab\}$) e si hanno $S \to ab|\, Aab$. Poi dimostro come l'esempio sopra

Esempio 4.6 trovo una CFG per $L = \{a^n cb^n | n \geq 1\} = acb, aacbb, aaacbbb, ...\}$
Il caso base è acb il passo è che se $a^{n-1}cb^{n-1} \in L$ allora $a^{n-1}cb^{n-1}acb \in L$
Si ha la grammatica $G = \{V, T, P, S\}$, $V = \{S\}$, $T = \{a, b, c\}$ e si hanno $S \to aSb|acb$.
dimostro che $aaaacbbbbb \in L$:

$$S \to aSb \to aaSbb \to aaaaSbbb \to aaaacbbbb$$

provo a usare anche una grammatica regolare, con le regole $S \to aS|c$, $c \to cB$ e $B \to bB|b$;

$$S \to aS \to aaS \to aaC \to aacB \to aacb...$$

non si può dimostrare in quanto non si può imporre una regola adatta

Esempio 4.7 $L = \{a^n cb^{n-1} | n \geq 2\}$, con $a^n cb^{n-1} = a^{n-1}acb^{n-1}$. $S \to aSb|aacb$. Quindi:

$$S \to aSb \to aaaccbb \in L$$

Esempio 4.8 cerco CFG per $L = \{a^n c^k b^n |\, n,\, k > 0\}$. a e b devono essere uguali, uso quindi una grammatica context free, mentre c genera un linguaggio regolare.
Si ha la grammatica $G = \{V, T, P, S\}$, $V = \{S, C\}$, $T = \{a, b, c\}$ e si hanno $S \to aSb|aCb$ e $C \to cC|c$. dimostro che $aaaccbbb \in L, n = 3,\, k = 2$:

$$S \to aSb \to aaSbb \to aaaCbbb \to aaacCbbb \to aaaccbbb$$

Esempio 4.9 scrivere CFG per $L = \{a^n b^n c^k b^k |\, n,\, k \geq 0\}$

$$= \{w \in \{a, b, c, d\}^* |\, a^n b^n c^k b^k |\, n,\, k \geq 0\}$$

quindi L concatena due linguaggi $L1$ e $L2$, $X = \{a^n b^n\}$ e $Y = \{c^k d^k\}$:

$$X \to aXb|\varepsilon$$

$$Y \to cYd|\varepsilon$$

$$S \to XY$$

voglio derivare $abcd$:

$$S \to XY \to XcYd \to aXbcYd \to aXbc\varepsilon d \to a\varepsilon bc\varepsilon d \to abcd$$

voglio derivare cd

$$S \to XY \to Y \to cYd \to cd$$

Quindi se ho $w \in L1, L2$, ovvero appartenente ad una concatenazione di linguaggi prima uso le regole di un linguaggio, poi dell'altro e infine ottengo il risultato finale.

Esempio 4.10 scrivere CFG per $L = \{a^n b^k c^k d^n | \, n > 0, \, k \geq 0\}$.

$$S \to aSd | \, aXd$$

$$X \to bXc | \varepsilon$$

derivo *aabcdd*:

$$S \to aSd \to aaXdd \to aabXcdd \to aabcdd$$

Esempio 4.11 scrivere CFG per $L = \{a^n c b^n c^m a d^m | \, n > 0, \, m \geq 1\}$.

$$S \to XY$$

$$X \to aXb | c$$

$$Y \to cUd | cad$$

$$S \to XY \to cY \to ccad$$

Esempio 4.12 scrivere CFG per $L = \{a^{n+m} x c^n y d^m | \, n, \, m \geq 0\}$. $a^{n+m} = a^n a^m$ o $a^m a^n$. Si hanno 2 casi:

1. $L = \{a^n a^m x c^n y d^m | \, n, \, m \geq 0\}$
2. $L = \{a^m a^n x c^n y d^m | \, n, \, m \geq 0\}$

ma solo $L = \{a^m a^n x c^n y d^m | \, n, \, m \geq 0\}$ può generare una CFG (dove non si possono fare incroci, solo concatenazioni e inclusioni/innesti).

$$S \to aSd | Y$$

$$Y \to Xy$$

$$X \to aXc | x$$

si può fare in 2:

$$S \to aSd | Xy$$

$$X \to aXc | x$$

derivo con $m = n = 1$, *aaxcyd*:

$$S \to aSd \to aXyd \to aaXcyd \to aaxcyd$$

Esempio 4.13 scrivere CFG per $L = \{a^n b^m | \, n \geq m \geq 0\}$.

$$L = \{\varepsilon, a, ab, aa, aab, aabb, aaa, aaab, aaabb, aaabbb, ...\}$$

Se $n \geq m$ allora $\exists k \geq 0 \rightarrow n = m + k$. Quindi:

$$l = \{a^{m+k} b^m | m, k \geq 0\}$$

si può scrivere in 2 modi:

1. $l = \{a^m a^k b^m | m, k \geq 0\}$ quindi con innesto
2. $l = \{a^k a^m b^m | m, k \geq 0\}$quindi con concatenazione

entrambi possibili per una CFG:

1. $$S \rightarrow XY$$
   $$X \rightarrow aX|\varepsilon \text{ si può anche scrivere } X \rightarrow Xa|\varepsilon$$
   $$Y \rightarrow aYb|\varepsilon$$
   oppure
   $$S \rightarrow aS|X$$
   $$X \rightarrow aXb|\varepsilon$$
2. $$S \rightarrow aSb|\varepsilon$$
   $$X \rightarrow aX|\varepsilon$$

Esempio 4.14 scrivere CFG per $L = \{a^n b^{m+n} c^h | \, m > h \geq 0, \, n \geq 0\}$.
Se $n > h$ allora $\exists k \rightarrow n = h + k$, quindi:

$$L = \{a^n b^{m+h+k} c^h | \, m > h \geq 0, \, n \geq 0\}$$

. ovvero:

$$L = \{a^n b^n b^k b^h c^h | \, m \geq 0, k > 0, h \geq 0\}$$

si ha:

$$S \rightarrow XYZ$$
$$X \rightarrow aXb|\varepsilon$$
$$Y \rightarrow Yb|b$$
$$Z \rightarrow bZc|\varepsilon$$

si può anche fare:

$$S \rightarrow XY$$
$$X \rightarrow aXb|\varepsilon$$
$$Y \rightarrow bYc|Z$$
$$Z \rightarrow bZ|b$$

Esempio 4.15 scrivere CFG per $L = \{a^n b^m c^k | \, k > n + m, \, n, m \geq 0\}$.
per $n = m = 0$, $k = 1$ avrò la stringa c. se $k > n + m$ allora $\exists l > 0 \rightarrow k = n + m + l$ quindi:

$$L = \{a^n b^m c^{n+m+l} | \, l > 0, \, n, m \geq 0\}$$

$$= L = \{a^n b^m c^n c^m c^l | \, l > 0, \, n, m \geq 0\}$$

sistemando:

$$= L = \{a^n b^m c^l c^m cnl | \, l > 0, \, n, m \geq 0\}$$

quindi:

$$S \rightarrow aSc|X$$

$$X \rightarrow bXc|Y$$

$$Y \rightarrow cY|c$$

Esempio 4.16 scrivere CFG per $L = \{a^n x c^{n+m} y^h z^k d^{m+h} | \, n, m, k, h \geq 0\}$. ovvero:

$$L = \{a^n x c^n c^m y^h z^k d^h d^m | \, n, m, k, h \geq 0\}$$

quindi avrò:

$$S \to XY$$

$$X \to aXc|x$$

$$Y \to cYd|W$$

$$W \to yWd|X$$

$$Z \to zZ|\varepsilon$$

Esempio 4.17 vediamo un esempio di grammatica dipendente dal contesto:

$$L = \{a^n b^n c^n | \, n \geq 1\}$$

$G = \{V, T, P, S\} = \{(S, B, C, X)\} = \{(a, b, c), P, S\}$ ecco le regole di produzione (qui posso scambiare variabili a differenza delle context free):

1. $S \to aSBC$
2. $S \to aBC$
3. $CB \to XB$
4. $XB \to XC$
5. $XC \to BC$
6. $aB \to ab$
7. $bB \to bb$
8. $bC \to bc$
9. $cC \to cc$

vediamo un esempio di derivazione: per $n = 1$ ho abc ovvero:

$$S \to aBC \to abC \to abc$$

con $n = 2$ ho $aabbcc$: $S \to aSBC \to aaBCBC \to aaBXBC \to aaBXCC \to aaBBCC \to aabBCC \to aabbCC \to aabbcC \to aabbcc$

Esempio 4.18 vediamo un esempio di grammatica dipendente dal contesto:

$$L = \{a^n b^m c^n d^m | \, n, m \geq 1\}$$

Si ha:

$$G = (\{S, X, C, D, Z\}, \{a, b, c, d\}, P, S)$$

con le seguenti regole di produzione:

- $S \to aSc | \, aXc$
- $X \to bXD | \, bD$
- $DC \to CD$
- $DC \to DZ$
- $DZ \to CZ$
- $XZ \to CD$
- $bC \to bc$
- $cC \to cc$
- $cD \to cd$
- $dD \to dd$

provo a derivare *aabbbccddd* quindi con $n = 2,\ m = 3$:

$$S \to aSC \to aaXCC \to aabXDCC \to aabbXDDCC \to$$

$$aabbbDDDCC \to aabbbCCDDD \to aabbbccddd$$

Esempio 4.19 Sia $L = \{w \in \{a, b\}^* |$ w contiene lo stesso numero di a e b$\}$:

$$S \to aSbS | \, bSaS | \, \varepsilon$$

dimostro per induzione che è corretto:

- **caso base:** $|w| = 0 \to w = \varepsilon$
- **caso passo:** si supponga che G produca tutte le stringhe (di lunghezza < di n) di $\{a, b\}^*$ con lo stesso numero di a e b e dimostro che produce anche quelle di lunghezza n, sia:

  $w \in \{a, b\}^* \mid |w| = n$ con a e b in egual numero, $m(a) = m(b)$ con m() che indica il numero di caratteri

  quindi si ha che:

  $$w = aw_1bw_2 \text{ o } w = bw_1aw_2$$

  sia.

  $$k_1 = m(a) \in w_1 = m(b) \in w_1$$

  $$k_2 = m(a) \in w_2 = m(b) \in w_2$$

  allora:

  $$k_1 + k_2 + 1 = m(a) \in w = m(b) \in W$$

  sapendo che $|w_1| < n$ e $|w_2| < n$ allora w_1 e w_2 sono egnerati da G per ipotesi induttiva

4.2 Alberi Sintatici

Definizione 4.5 Data una grammatica CFG, $G = \{V, T, P, S\}$ un **albero sintattico** per G soddisfa le seguenti condizioni:

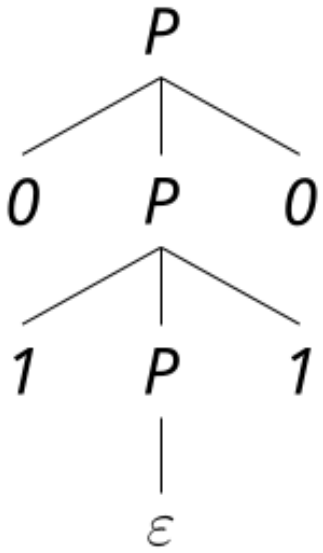
- ogni nodo interno è etichettato con una variabile in V
- ogni foglia è anch'essa etichettata con una variabile o col simbolo di terminale T o con la stringa vuota ε (in questo caso la foglia è l'unico figlio del padre)
- se un nodo interno è etichettato con A i suoi figli saranno etichettati con X1, ..., Xk e $A \to X1, ..., Xk$ sarà una produzione di G in P. Se un X_i è ε sarà l'unica figlio e $A \to \varepsilon$ sarà comunque una produzione di G

La concatenazione in ordine delle foglie viene detto **prodotto dell'albero**

Esempio 4.20 Usiamo l'esempio delle stringhe palindrome:

$$P \to 0P0|\,1P1|\varepsilon$$

sia il seguente albero sintatico:

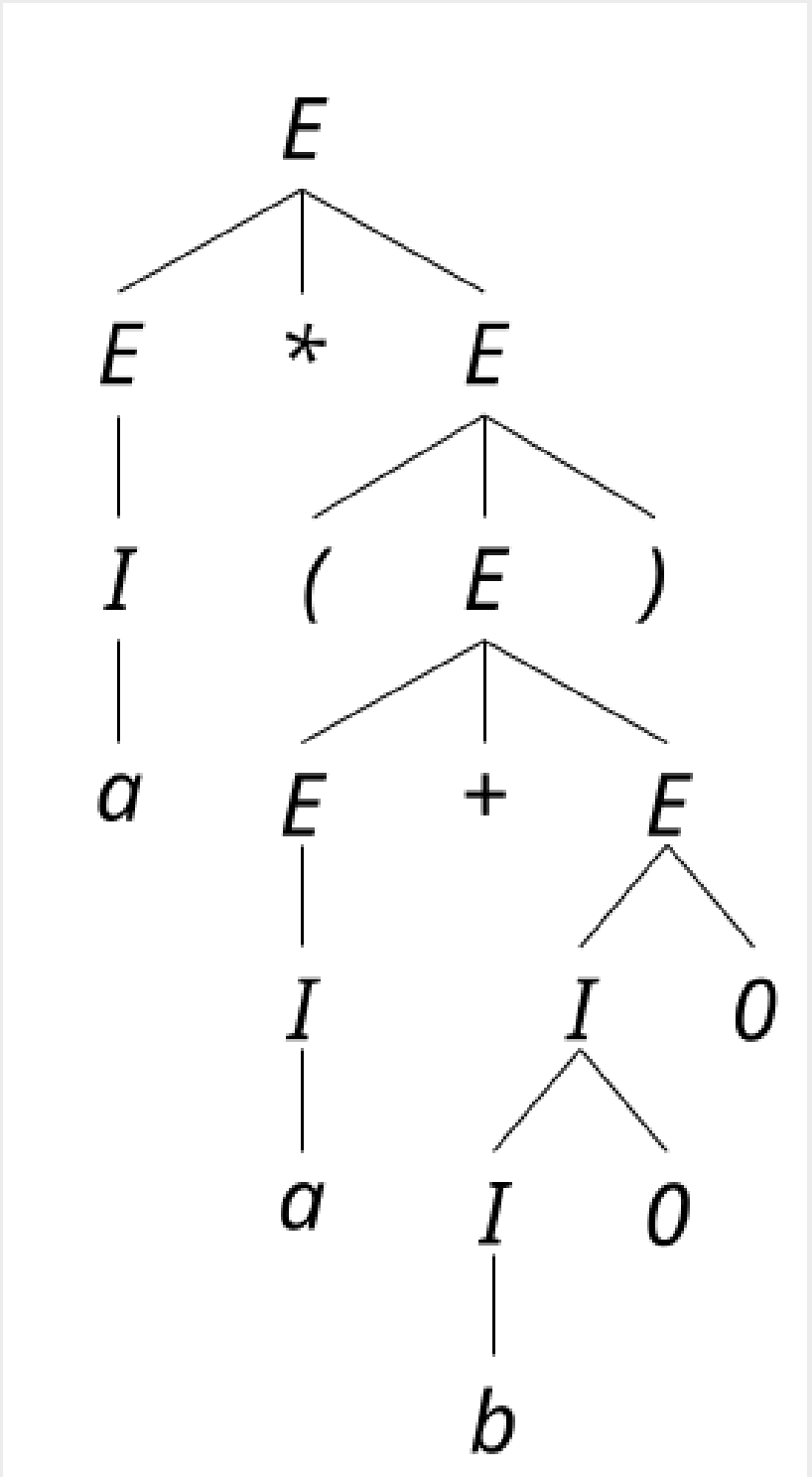


Esempio 4.21 Si ha:

$$E \to I|\, E + E|\, E * E|\, (E)$$

$$I \to a|\, b|\, Ia|\, Ib|\, I0|\, I1$$

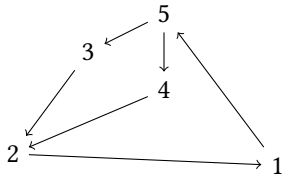
un albero sintattico per $a * (a + b00)$ può essere:



Data una CFG si ha che i seguenti cinque enunciati si equivalgono:

1. la procedura di inferenza ricorsiva stailisce che una stringa w di simboli terminali appartiene al linguaggio $L(A)$ con A variabile
2. $A \to^* w$
3. $A \to^*_{lm} w$
4. $A \to^*_{rm} w$
5. esiste un albero sintattico con radice A e prodotto w

queste 5 proposizioni si implicano l'uni l'altra:



vediamo qualche dimostrazione di implicazione tra queste proposizioni:

da 1 a 5. si procede per induzione:

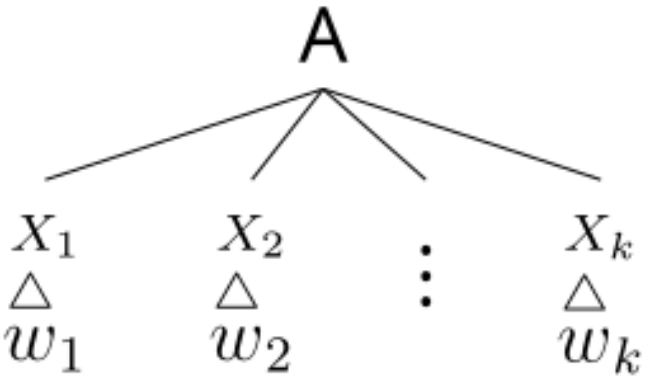
- **caso base:** ho un livello solo (una sola riga), $\exists A \to w$:

- **caso passo:** suppongo vero per un numero di righe $\leq n$, lo dimsotro per $n+1$ righe:

$$A \to X_1, X_2, ..., X_k$$

$$w = w_1, w_2, ..., w_k$$

ovvero, in meno di $n+1$ livelli:



□

da 5 a 3. procedo per induzione:

- **caso base (n=1):** $\exists A \to w$ quindi $A \to_{lm} w$, come prima si ha un solo livello:

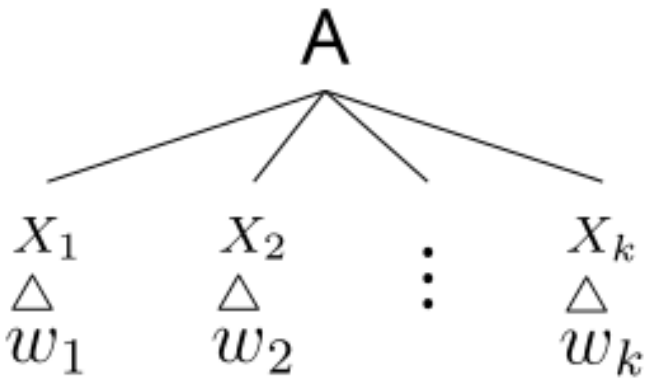
$$\begin{matrix} A \\ \triangle \\ w \end{matrix}$$

- **caso passo:** suppongo che la proprierà valga per ogni albero di profondità minore uguale a n, dimostro che valga per gli alberi profondi $n+1$:

$$A \to X_1, X_2, ..., X_k$$

$$w = w_1, w_2, ..., w_k$$

ovvero, in meno di $n+1$ livelli:



$$A \to_{lm} X_1, X_2, ..., X_k$$

$x_1 \to^*_{lm} w_1$ per ipotesi induttiva si ha un albero al più di n livelli

quindi:

$$A \to_{lm} X_1, ..., X_k \to^*_{lm} w_1, X_2, ..., X_k \to^*_{lm} ... \to^*_{lm} w_1, ..., w_k = w$$

Esempio 4.22

$$E \to I \to Ib \to ab$$

$$\alpha E\beta \to \alpha I\beta \to \alpha Ib\beta \to \alpha ab\beta, \quad \alpha, \beta \in (V \cup T)^*$$

□

Esempio 4.23 Mostro l'esistenza di una derivazione sinistra dell'albero sintattico di $a * (a + b00)$:

$$E \to^*_{lm} E * E \to^*_{lm} I * E \to^*_{lm} a * E \to^*_{lm} a * (E) \to^*_{lm} a * (E + E) \to^*_{lm}$$

$$a*(I+E) \to^*_{lm} a*(a+E) \to^*_{lm} a*(a+I) \to^*_{lm} a+(a+I0) \to^*_{lm} a*(a+I00) \to^*_{lm} a*(a+b00)$$

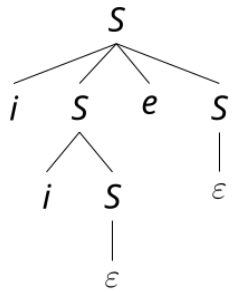
4.3 Grammatiche ambigue

Definizione 4.6 Una grammatica è definita ambigua se esiste una stringa w di terminali che ha più di un albero sintattico

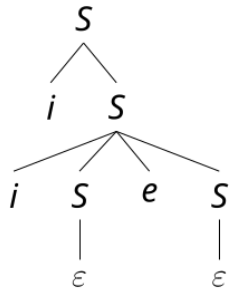
Esempio 4.24 vediamo un esempio:

1. $E \to E + E \to E + E * E$ ovvero:

$1.\ S \to iSeS \to iiSeS \to iie:$



$2.\ S \to iS \to iiSeS \to iieS \to iie:$



2. $E \to E * E \to E + E * E$ ovvero:

si arriva a due stringhe uguali ma con alberi diversi. Introduciamo delle categorie sintatiche, dei vincoli alla produzione delle regole:

1. $E \to T | E + T$
2. $T \to F | T + F$
3. $F \to I | (E)$
4. $I \to a | b | Ia |, Ib | I0 | I1$

Possono esserci più derivazioni di una stringa ma l'importante è che non ci siano alberi sintattici diversi. Capire se una CFG è ambigua è un problema indecidibile

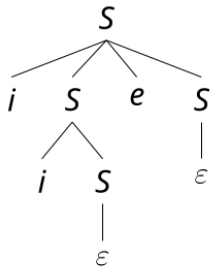
Esempio 4.25 vediamo un esempio:

$$S \to \varepsilon | SS | iS | iSeS$$

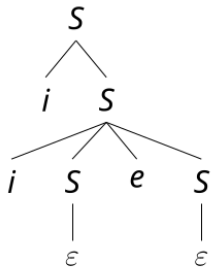
con S=statement, i=if e e=else. Considero due derivazioni:

1. $S \to iSeS \to iiSeS \to iie$:

1. $S \to iSeS \to iiSeS \to iie:$



2. $S \to iS \to iiSeS \to iieS \to iie:$



Si ha quindi una grammatica ambigua

Teorema 4.2 Per ogni CFG, con $G = (V, T, P, S)$, per ogni stringa w di terminali si ha che w ha due alberi sintattici distinti sse ha due derivazioni sinistre da S distinte.
Se la grammatica non è ambigua allora esiste un'unica derivazione sinistra da S

Linguaggi inerentemente ambigui

Definizione 4.7 Un linguaggio L è inerentemente ambiguo se tutte le grammatiche CFG per tale linguaggio sono a loro volta ambigue

Esempio 4.26 Sia $L = \{a^n b^n c^m d^m | \, n, m \geq 1\} \cup \{a^n bmnc^m d^n | \, n, m \geq 1\}$
si ha quindi un CFL formato dall'unione di due CFL. L è inerentemente ambiguo e generato dalla seguente grammatica:

- $S \to AB | \, C$
- $A \to aAb | \, ab$
- $B \to cBd | \, cd$
- $C \to aCd | \, aDd$
- $D \to bDc | \, bc$

si possono avere due derivazioni:

1. $S \to_{lm} AB \to_{lm} aAbB \to_{lm} aabbB \to_{lm} aabbcBd \to_{lm} aabbccdd$
2. $S \to_{lm} C \to_{lm} aCd \to_{lm} aaBdd \to_{lm} aabBcdd \to_{lm} aabbccdd$

a generare problemi sono le stringhe con n=m perché possono essere prodotte in due modi diversi da entrambi i sottolinguaggi. Dato che l'intersezione tra i due sottolinguaggi non è buota si ha che L è ambiguo

4.4 Grammatiche Regolari

Sono le grammatiche che generano i linguaggi regolari (quelli del terzo tipo) che sono casi particolari dei CFL.
Si ha la solita grammatica $G = (V, T, P, S)$ con però vincoli su P:

- ε si può ottenere solo con $S \to \varepsilon$
- le produzioni sono tutte lineari a destra ($A \to aA$ o $A \to a$) o a sinistra ($A \to Ba$ o $A \to a$)

Esempio 4.27 $I \to a | \, b | \, Ia | \, Ib | \, I0 | \, I1$ è una grammatica con le produzioni lineari a sinistra.
Potremmo pensarlo a destra $I \to a | \, b | \, aI | \, bI | \, 0I | \, 1I$.
Vediamo esempi di produzione con queste grammatiche:

- con $I \to a | \, b | \, Ia | \, Ib | \, I0 | \, I1$ possiamo derivare $ab01b0$:

$$I \to I0 \to Ib0 \to I1b0 \to I01b0 \to Ib01b0 \to ab01b0$$

- con $I \to a | \, b | \, aI | \, bI | \, 0I | \, 1I$ invece non riusciamo a generare nulla:

$$I \to 0I \to 0a$$

definisco quindi un'altra grammatica (con una nuova categoria sintattica):

$$I \to aJ | \, bJ$$

$$J \to a | \, b | \, aJ | \, bJ | \, 0J | \, 1J$$

che però non mi permette di terminare le stringhe con 0 e 1, la modifico ancora otterdendo:

$$I \to aJ | \, bJ$$

$$J \to a | \, b | \, aJ | \, bJ | \, 0J | \, 1J | \, 0 | \, 1$$

e questo è il modo corretto per passare da lineare sinistra a lineare destra

Esempio 4.28 Sia $G = (\{S\}, \{0, 1\}, P, S)$ con $S \to \varepsilon | \, 0 | \, 1 | \, 0S | \, 1S$. Si ha quindi:

$$L(G) = \{0, 1\}^*$$

si hanno comunque due proposizioni ridondanti, riducendo trovo:

$$S \to \varepsilon | \, 0S | \, 1S$$

con solo produzioni lineari a destra. Con produzioni lineari a sinistra ottengo:

$$S \to \varepsilon | \, S0 | \, S1$$

Esempio 4.29 Trovo una grammatica lineare destra e una sinistra per $L = \{a^n b^m | \, n, m \geq 0\}$:

- **lineare a destra:** si ha $G = (\{S, B\}, \{a, b\}, P, S)$ e quindi:

  $$S \to \varepsilon | \, aS | \, bB$$

  $$B \to bB | \, b$$

  ma non si possono generare stringhe di sole b, infatti:

  $$S \to aS \to abB \to abbB \to abbb$$

  ma aggiungere ε a B **non è lecito**. posso però produrre la stessa stringa da due derivazioni diverse:

  $$S \to \varepsilon | \, aS | \, bB | \, b$$

  $$B \to bB | \, b$$

  che risulta quindi la nostra lineare a destra
- **lineare a sinistra:** si ha $G = (\{S, A\}, \{a, b\}, P, S)$ e quindi:

  $$S \to \varepsilon | \, Sb | \, Ab | \, a$$

  $$A \to Aa | \, a$$

Esempio 4.30 Trovo una grammatica lineare destra e una sinistra per $L = \{ab^n cd^m e | \, n \geq 0 \, , m > 0\}$:

- **lineare a destra:** si ha si ha $G = (\{S, A, B, E\}, \{a, b, c, d, e\}, P, S)$ e quindi:

$$S \to aA$$
$$A \to bA | \, cB$$
$$B \to dB | \, dE$$
$$E \to e$$

- **lineare a sinistra:** si ha si ha $G = (\{S, X, Y, Z\}, \{a, b, c, d, e\}, P, S)$ e quindi:

$$S \to Xe$$
$$A \to Xd | \, Yd$$
$$B \to Zc$$
$$E \to a | \, Zb$$

quindi se per esempio ho la stringa ”ciao” si ha:

- **lineare a destra:**

$$S \to Ao$$
$$A \to Ba$$
$$B \to Ei$$
$$E \to c$$

- **lineare a sinistra:**

$$S \to cA$$
$$A \to iB$$
$$B \to aE$$
$$E \to o$$

Esempio 4.31 A partire da $G = (\{S, T\}, \{0, 1\}, P, S)$ con:

$$S \to \varepsilon | \, 0S | \, 1T$$
$$T \to 0T | \, 1S$$

trovo come è fatto $L(G)$:

$$L(G) = \{w \in \{0, 1\}^* | \, w \text{ ha un numero di 1 pari}\}$$

Esempio 4.32 fornire una grammatica regolare a destra e sinistra per:

$$L = \{w \in \{0, 1\}^* | \, w \text{ ha almeno uno 0 o almeno un 1}\}$$

Si ah che tutte le stringhe tranne quella vuota ciontengono uno 0 o un 1 quindi $G = (\{S\}, \{0, 1\}, P, S)$:

- **lineare a destra:**

$$S \to 0 | \, 1 | \, 0S | \, 1S$$

- **lineare a sinistra:**

$$S \to 0 | \, 1 | \, S0 | \, S1$$

Vediamo ora un esempio di *Context Free Language (CFL)*, costruito a partire da una *Context Free Grammar (CFG)*:

Esempio 4.33 Sia $\Sigma = \{0, 1\}$ e L_{pal} = "*stringhe palindrome binarie*". Quindi, per esempio, $0110 \in L$, $11011 \in L$ ma $10010 \notin L$. Si ha che ε, la stringa vuota, appartiene a L. Diamo una definizione ricorsiva:

- **base:** $\varepsilon,\ 0\ 1 \in L_{pal}$
- **passo:** se w è palindroma allora $0w0$ è palindromo e $1w1$ è palindromo

una variabile generica S può sottostare alle *regole di produzione* di una certa grammatica. In questo caso si ha uno dei seguenti:

$$S \to \varepsilon,\ S \to 0,\ S \to 1,\ S \to 0S0,\ S \to 1S1$$

riprendiamo l'esempio sopra:

Esempio 4.34

$$G_{pal} = (V = \{S\},\ T = \{0, 1\},\ P,\ S)$$

con:

$$P = \{S \to \varepsilon,\ S \to 0,\ S \to 1,\ S \to 0S0,\ S \to 1S1\}$$

Si può ora costruire un algoritmo per creare una stringa palindroma a partire dalla grammatica G:

$$\underbrace{S}_{\text{start}} \quad \underbrace{\to}_{\text{applico una regola}} \quad 1S1 \to 01S10 \to \quad \underbrace{01010}_{\text{sostituisco variabile}}$$

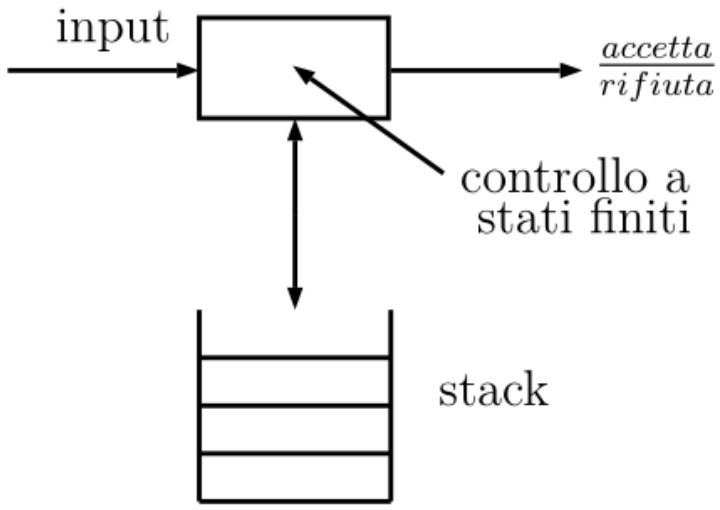
con S, $1S1$ *e* $01S10$ che sono *forme sentenziali*. Posso così ottenere tutte le possibili stringhe. Esiste anche una forma abbreviata:

$$S \to \varepsilon|o|1|0S0|1S1$$

Non si fanno sostituzioni in parallelo, prima una S e poi un'altra

4.5 Automi a Pila

Si introduce un nuovo tipo di automa, il PDA (push down automata) che può essere pensato come un $\varepsilon - NFA$ col supporto di una pila (stack):



Tikz

e viene definito un PDA P come:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$$

con;

- Q: insieme finito e non vuoto di stati
- Σ: alfabeto di simboli di input
- Γ: alfabeto di simboli di stack
- $q_0 \in Q$: stato iniziale
- $z_0 \in \Gamma \backslash \Sigma$: simbolo iniziale dello stack
- $F \in Q$: insieme degli stati accettanti o finali

si ha che:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\} \times \Gamma \to 2^{Q \times \Gamma^*}$$

quindi:

$$\delta(q_0, a, X) = \{(p_1, X_1), (p_2, X_2), ...\} \, insieme \;\; finito \;\; p_i \in Q \;\; X_i \in \Gamma^*$$

si hanno dei casi particolari:

- lo stato p potrebbe coincidere con Q e si avrebbe un cappio
- se $\Gamma = \varepsilon$ si ha il pop di X dallo stack
- se $\Gamma = X$ si lascia lo stack invariato
- se $\Gamma = Y \neq X$ si ha la sostituzione di X con Y in cima allo stack
- se Γ è una stringa di simboli si ha il la rimozione di X dallo stack e l'aggiunta a uno a uno dei simboli nello stack

Esempio 4.35 Trovo PDA per il linguaggio delle stringhe binarie palindrome di lunghezza pari: $L = \{ww^R | w \in \{0,1\}^*\}$. Con R che indica rovesciato
Si ha la CFG $G = (\{P\}, \{0,1\}, Prod, P)$ tale che:

$$P \to 0P0|1P1|\varepsilon$$

si hanno quindi tre stati:

- q_0 che è quello iniziale che legge w e spinge i dati sullo stack
- q_1 che letta w legge i simboli di wR e li confronta con quelli dello stack
- q_2 sarà la stringa accettata

descriviamo formalmente l'automa con la funzione di transizione δ. PDA $P = (\{q_0, q_1, q_2\}, \{0,1\}, \{o, 1, z_0\}, \delta, q_0, z_0, \{q_2\})$ ovvero:

$$\delta(q_0, 0, z_0) = \{(q_0, 1z_0)\}$$

$$\delta(q_0, 1, z_0) = \{(q_0, 0z_0)\}$$

$$\delta(q_0, 0, 0) = \{(q_0, 00)\}$$

$$\delta(q_0, 0, 1) = \{(q_0, 01)\}$$

$$\delta(q_0, 1, 0) = \{(q_0, 10)\}$$

$$\delta(q_0, 1, 1) = \{(q_0, 11)\}$$

$$\delta(q_0, \varepsilon, z_0) = \{(q_0, z_0)\}$$

$$\delta(q_0, \varepsilon, 0) = \{(q_0, 0)\}$$

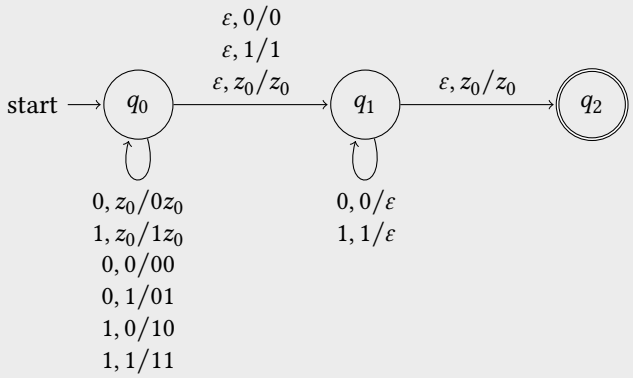
$$\delta(q_0, \varepsilon, 1) = \{(q_0, 1)\}$$

$$\delta(q_1, 0, 0) = \{(q_1, \varepsilon)\}$$

$$\delta(q_2, 1, 1) = \{(q_1, \varepsilon)\}$$

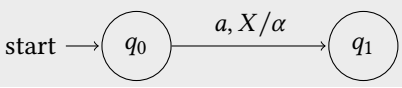
$$\delta(q_2, \varepsilon, z_0) = \{(q_2, 0z_0)\}$$

otteniamo il seguente PDA:



e si definisce questa notazione per gli archi:

$$(p, \alpha) \in \delta(q, a, X)$$



analizziamo meglio i PDA. Si ha che la **descrizione istantanea (ID)** di

un PDA è una tripla:

$$ID : (q, w, \gamma)$$

con $q \in Q$ stato attuale $w \in \Sigma^*$ input rimanente e $\gamma \in \Gamma^*$ contenuto attuale dello stack.
Definiamo ora il concetto di **mossa in un passo** dato $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ la mossa è una relazione $\vdash_p$:

$$(p, \alpha) \in \delta(q, a, X) \textit{ allora } \forall w \in \Sigma^* \textit{ e } \forall \beta \in \Gamma^* \rightarrow (q, aw, X\beta) \vdash (p, w, \alpha\beta$$

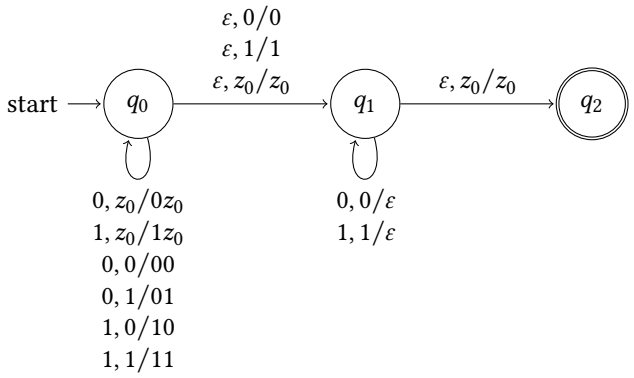
e

$$(p, \alpha) \in \delta(q, \varepsilon, X) \textit{ allora } \forall w \in \Sigma^* \textit{ e } \forall \beta \in \Gamma^* \rightarrow (q, w, X\beta) \vdash (p, w, \alpha\beta)$$

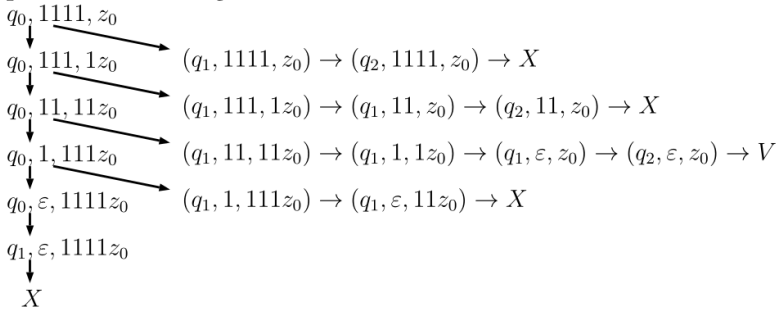
ora possiamo anche definire la relazione con 0 o più mosse in forma induttiva $\overset{*}{\vdash}_p$:

- **caso base:** $\forall ID\ I, I \overset{*}{\vdash} I$
- **caso passo:** $I \overset{*}{\vdash} J$ se $\exists ID\ K$ tale che $I \vdash K\ e\ K \overset{*}{\vdash} J$

vediamo un esempio con un PDA che accetta $ww^R | w \in \{0,1\}^*$:



e prendiamo la stringa 1111:



chiamiamo **computazione** una sequenza di mosse, non necessariamente di successo. Si hanno alcune proprietà:

- se una se una sequenza di ID è lecita per un PDA P allora è lecita anche la sequenza di Id ottenuta concatenando $w \in \Sigma^*$ in ogni ID
- se una se una sequenza di ID è lecita per un PDA P e resta una coda di input non consumata allora posso rimuovere tale coda in ogni ID e ottenere un'altra sequenza lecita
- se una se una sequenza di ID è lecita per un PDA P allora è lecita la sequenza ottenuta aggiungendo $\gamma \in \Gamma^*$ in coda alla terza sequenza di ogni ID

del resto però:

$$(q, Xw, \alpha\gamma) \overset{*}{\vdash}_p (p, Yw, \beta\gamma) \not\Rightarrow (q, X, \alpha) \overset{*}{\vdash} (p, y, \beta),\ \ x, w, y \in \Sigma^*\ \ \alpha, \beta, \gamma \in \Gamma^*$$

per queste proprietà valgono i seguenti teoremi:

Teorema 4.3 per la seconda: Se $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ è un PDA e $(q, Xw, \alpha) \overset{*}{\vdash}_p (p, Yw, \beta)$ allora vale anche:

$$(q, X, \alpha) \overset{*}{\vdash}_p (p, Y, \beta)$$

Teorema 4.4 per la prima e la terza: Se $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ è un PDA e $(q, X, \alpha) \vdash_p^* (p, Y, \beta)$ allora:

$$\forall \gamma \in \Gamma^* \ \textit{vale anche} \ (q, Xw, \alpha\gamma) \vdash_p^* (p, Yw, \beta\gamma)$$

Si definiscono due modalità di accettazione per i PDA:

1. **per stato finale:** sia $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ si ha che:

   $$L(P) = \{w \in \Sigma^* | (q_0, w, z_0) \vdash_p^* (q, \varepsilon, \alpha)\}$$

   con $q \in F$ e $\forall \alpha \in \Gamma^*$
2. **per stack vuoto:** sia $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ si ha che:

   $$N(P) = \{w \in \Sigma^* | (q_0, w, z_0) \vdash_p^* (q, \varepsilon, \varepsilon)\}$$

   con $q \in Q$ e in questo caso l'insieme degli stati finali F non ha alcuna influenza

In realtà si ha che la classe di linguaggi accettati dai PDA per stato finale è uguale a quella per stack vuoto, anche se passare da un tipo all'altro di PDA è complesso. SI ha il seguente teorema per la trasformazione:

Teorema 4.5 se $L = N(P_N)$ per un PDA $P_N = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ allora $\exists$ *PDA* P_F *tale che* $L = L(P_F)$

Dimostrazione. Sia $x_0 \in \Gamma$, che indica la fine dello stack di P_F. Si ha:

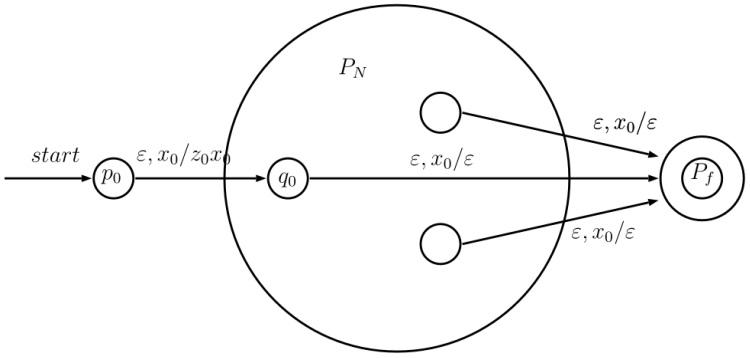
$$\delta(p_0, \varepsilon, x_0) = \{(q_0, z_0 x_0\}$$

e:

$\forall q \in Q, \ \forall a \in \Sigma \cup \{\varepsilon\}, , \forall y \in \Sigma : \ \delta_F(q, a, y)$ contiene tutte le coppie di $\delta_N(q, a, y)$

$$\forall q \in Q, \delta_F(q, \varepsilon, x_0) = \{(P_F, \varepsilon)\}$$

quindi graficamente:



Bisogna dimostrare che effettivamente $w \in L(P_F) \longleftrightarrow \in N(P_N)$.
se $w \in N(P_N) \exists$ una sequenza di ID $(q_0, w, z_0) \vdash_{P_N}^* (q, \varepsilon, \varepsilon)$ per un qualche $q \in Q$:

$$(q_0, w, z_0 x_0) \vdash_{P_N}^* (q, \varepsilon, x_0)$$

inoltre:

$$(q_0, w, z_0 x_0) \vdash_{P_F}^* (q, \varepsilon, x_0)$$

e quindi:

$$(p_0, w, x_0) \vdash_{P_F} (q_0, w, z_0 x_0) \vdash_{P_F}^* (q, \varepsilon, x_0) \vdash_{P_F} (P_F, \varepsilon, \varepsilon)$$

solo se togliendo il primo e l'ultimo passo di P_F ripercorro all'indietro quanto scritto sopra. □

Esempio 4.36 trasformazione da accettante per stack vuoto a accettante per stato finale. Siano:

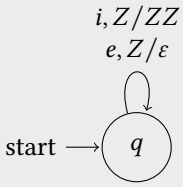
$$\Sigma = \{i, e\}$$

$$P_n = (\{q\}, \{i, e\}, \{Z\}, \delta_M, q, Z)$$

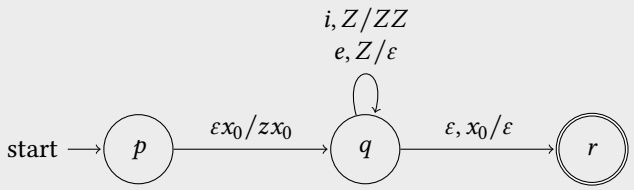
$$\delta_N(q, i, Z) = \{(q, ZZ)\}$$

$$\delta_N(q, e, Z) = \{(q, \varepsilon)\}$$

quindi:



quindi inseriamo una Z quando leggiamo i e ne rimuoviamo una se leggiamo e e si parte con una Z nello stack.
Costruisco ora il PDA P_F che accetta lo stesso linguaggio ma per stato finale, introduco lo stato iniziale p e quello accettante r, uso x_0 come segnale della fine dello stack:



e si ha formalmente:

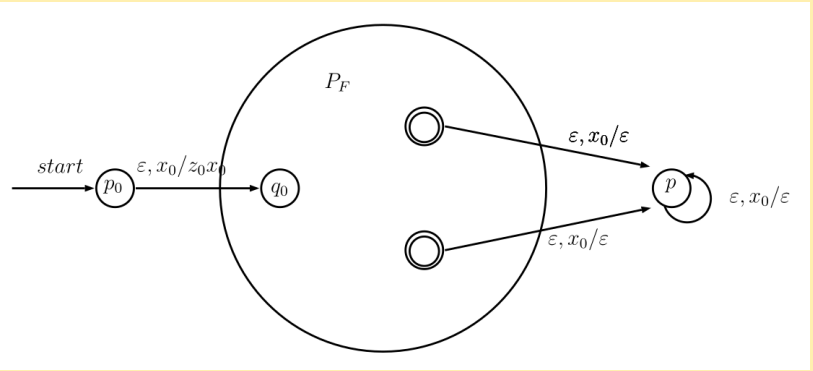
$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, x_0\}, \delta_F, p, x_0, \{r\})$$

con δ_F che rappresenta le seguenti quattro regole:

1. $\delta_F(p, \varepsilon, x_0) = \{(q, Zx_0)\}$ regola che fa partire P_F con x_o come segnalatore dello stack
2. $\delta_F(p, i, Z) = \{(q, ZZ)\}$ regola che inserisce Z quando si ha i simulando P_N
3. $\delta_F(p, e, Z) = \{(q, Z\varepsilon)\}$ regola che rimuove Z quando si ha e simulando P_N
4. $\delta_F(p, e, x_0) = \{(r, \varepsilon)\}$ regola che permette a P_F di accettare quando P_N esaurisce lo stack

Si può anche effettuare la trasformazione inversa:

Teorema 4.6 Sia $P_F = (Q, \Sigma, \Gamma, \delta_f, q_0, Z_0, F)$.
Si aggiunge una transizione ε a un nuovo stato p da ogni accettante di P_F. quando si ha p P_N svuota lo stack senza consumare input. Quindi se P : F entra in uno stato accettante dopo aver consumato l'input w, P_N svuota lo stack dopo aver consumato w. Per evitare che si svuoti lo stack per una stringa non accettata uso x_0 per indicare il fondo dello stack. Il nuovo P_N parte da p_0 che ha il solo scopo di inserire il simbolo iniziale di P_F e passare al suo stato iniziale. Si ottiene quindi:



e si ha formalmente:

$$P_F = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{x_0\}, \delta_N, p_0, x_0)$$

dove δ_N è così definita:

1. $\delta_N(p_0, \varepsilon, x_0) = \{(q_0, Z_0x_0)\}$ inserisce il simbolo iniziale di P_F nello stack e va allo stato iniziale di P_F
2. $\forall q \in Q$ ogni simbolo di input $a \in \Sigma$, compreso l'input vuoto, e $\forall y \in \Gamma$, $\delta_N(q, a, y)$ contiene tutte le coppie di $\delta_F(q, a, y)$. Quindi P_N simula P_F
3. per tutti gli stati accettanti $q \in F$ e i simboli di stack $y \in \Gamma$, compreso x_0, si ha che $\delta_N(q, \varepsilon, y)$ contiene (p, ε), quindi ogni volta che P_F accetta P_N inizia scaricare lo stack senza consumare ulteriori input
4. per tutti i simboli di stack $y \in \Gamma$, compreso x_0, si ha che $\delta_N(q, \varepsilon, y) = \{(p, \varepsilon)\}$, quindi giunti allo stato p, ovvero quando P_F ha accettato, P_N elimina ogni simbolo nel suo stack fino a svuotarlo

inoltre formalmente voglio dimostrare che:

$$w \in L(P_F) \rightarrow w \in N(P_N)$$

e quindi ho le seguenti mosse:

$$(q_0, w, z_0) \vdash^*_{P_F} (q, \varepsilon, \alpha)\ q \in F,\ \alpha \in \Gamma^*$$

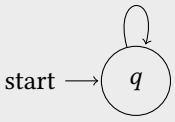
$$(p_o, w, x_0) \vdash (q_0, w, z_0x_0) \vdash^*_{P_N} (q, \varepsilon, \alpha, x_0) \vdash^*_{P_N} (p, \varepsilon, \varepsilon)$$

Esempio 4.37 si ha una CFG $G = (\{i, e\}, \{a, b, 0, 1, *, +, (,)\}, P, E)$ con:

$$P : I \to a|b|Ia|Ib|I0|I1$$

$$E \to E + E|E * E|(E)$$

si ha il PDA $P_G = (\{q\}, \Sigma, \Sigma \cup \{i, e\}, \delta, q, E)$:



si ha quindi:

$$\delta(q, \varepsilon, i) = \{(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)\}$$

$$\delta(q, \varepsilon, E) = \{(q, I), (q, E + E), (q, E * E), (q, I(E))\}$$

$$\delta(q, a, a) = \{(q, \varepsilon)\}$$

$$\delta(q, b, b) = \{(q, \varepsilon)\}$$

$$\cdots = \{(q, \varepsilon)\}$$

quindi si ha:

$$E \to E + E \to i + E \to a + (E) \to a + (i) \to a + (i0) \to a + (b0)$$

$$(q, a + (b0), E) \vdash (q, a + (b0), E + E) \vdash (q, a + (b0), i + E) \vdash (q, a + (b0), a + E)$$
$$\vdash (q, +(b0), +E) \vdash (q, (b0), E) \vdash (q, (b0), (E)) \vdash (q, b0), E))$$
$$\vdash (q, b0), i)) \vdash (q, b0), i0)) \vdash (q, b0), b0) \vdash (q, 0), 0))$$
$$\vdash (q,),)) \vdash (q, \varepsilon, \varepsilon)$$

questo esempio è generalizzabile ad ogni CFG

Teorema 4.7 sia $G = (V, T, P, S)$ una CFG:

$$\exists\ PDA\ Q = (\{q\}, T, V \cup T, \delta, q, S) \text{ tale che } N(Q) = L(G)$$

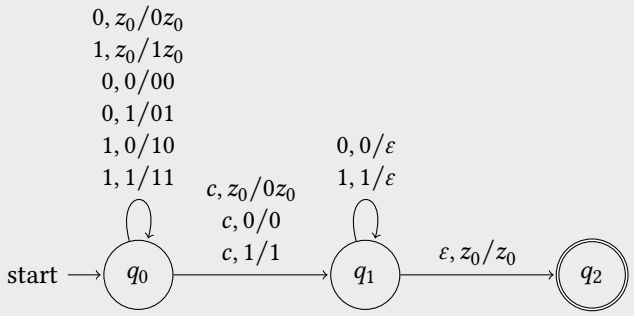
$$\forall A \in V\ \delta(q, \varepsilon, A) = \{(q, \beta)|\ A \to B \text{ e' una produzione di G}\}$$

$$\forall a \in T\ \delta(q, a, a) = \{(a, \varepsilon)\}$$

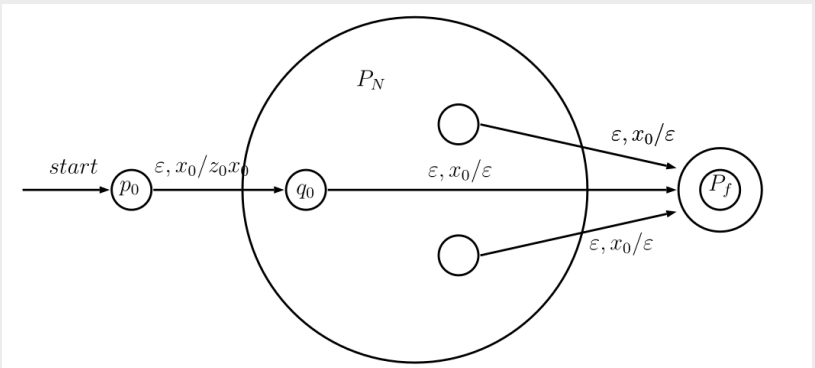
Questo dimostra che ogni CFL può essere accettato da un PDA accettante per stack vuoto. Per il teorema visto in precedenza, posso sempre costruire un altro PDA accettante per stati finale. I PDA accettano tutti e soli i linguaggi CF. Mostrare che accettano solo linguaggi di tipo 2 è complicato.
Un tipo di PDA interessante, soprattutto per i parse, è il PDA deterministico, il **DPDA**.
Un PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ è deterministico se:

1. $|\delta(q, a, x)| \leq 1\ \forall q \in Q, \forall a \in \Sigma \cup \{\varepsilon\}, \forall x \in \Gamma$
2. se $|\delta(q, a, x)| \neq 0$ per qualche $a \in \Sigma$ allora $|\delta(q, \varepsilon, x)| = 0$

Esempio 4.38 abbiamo il linguaggio $L_{wcw^R} = \{wcw^R | \, w \in \{0,1\}^*\}$
Gli automi a pila deterministici non riconosco tutti i CFL, ma solo una classe strettamente più piccola. Ad esempio non potrebbero riconoscere il linguaggio delle palindrome senza "il segnalibro" c. SI ha quindi:



quindi:



si ha infatti il seguente teorema:

Teorema 4.8 *$L \in REG \rightarrow \exists PDA \; P$ tale che $L = L(P)$*

Dimostrazione.

$$L \in REG \rightarrow \exists DFA \; A = (Q, \Sigma, \delta_A, q_0, F) \text{ tale che } L = L(A)$$

costruisco il DPDA $P = (Q, \Sigma, \{z_0\}, \delta_p, q_0, z_0, F)$ con:

$$\delta_p(q, a, z_0) = \{p, z_0\} \; \forall p, q \in Q \text{ tali che } \delta_A(q, a) = 0$$

vale:

$$(q_0, w, z_0) \overset{A}{\vdash_p} (p, \varepsilon, z_0) \longleftrightarrow \overset{\wedge}{\delta_A} (q_0, w) = p$$

□

si ha inoltre il seguente teorema:

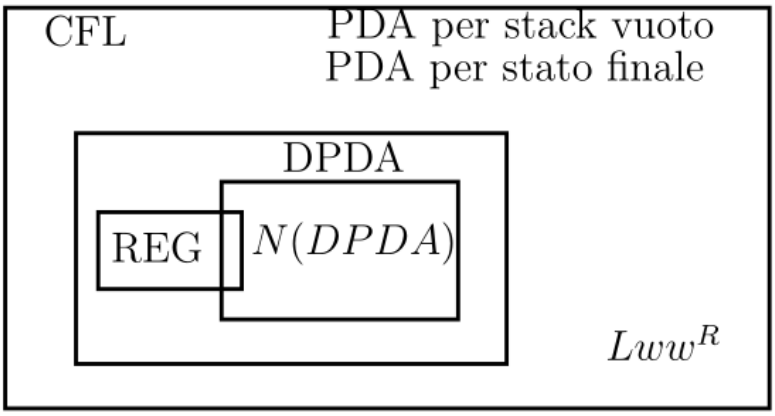
Teorema 4.9 *L è $N(P)$ per un DPDA P sse L è $L(P')$ per un DPDA P' e L ha le proprietà di prefisso* **prefix-free**

definiamo così la proprietà di prefisso:

$$\nexists x, y \in L \text{ tali che } x \neq y \; e \; x \text{ è prefisso di } y$$

per esempio $L = \{0\}^0 = \{\varepsilon, 0, 00, 000, ...\}$ non ha la proprietà di prefisso. Osserviamo che se la stringa vuota appartiene al linguaggio, tale stringa

è prefissa di tutte le altre e quindi il linguaggio non può avere la proprietà di prefisso. Affermiamo che L è regolare, quindi è accettato da un DPDA per stati finali ma non da uno per stack vuoto. Completiamo il diagramma precedente sulle classi di linguaggi:



SI ha che L_{wcw^R} gode della proprietà di prefisso:

$$y = wcw^R \in L \; Se \; x \neq y, \text{ prefisso di } y, x \notin L$$

tornando alle grammatiche si hanno ora due teoremi:

Teorema 4.10 se $L = N(P)$ per un DPDA P, allora L ha una CFG non ambigua

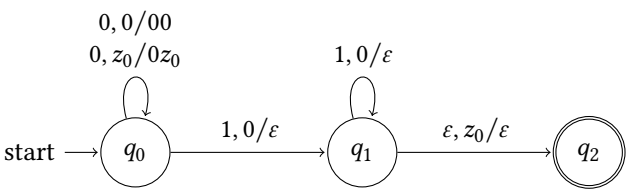
Teorema 4.11 se $L = L(P)$ per un DPDA P, allora L ha una CFG non ambigua

dimostriamo il secondo:

Dimostrazione. $L = L(P)$ per un DPDA P, costruiamo $L^{'} = L$, quindi $L^{'}$ ha la proprietà di prefisso. Esiste quindi un DPDA $P^{'}$ tale che $L^{'}N(P)$, esiste quindi per il teorema sopra una CFG $G^{'}$ tale che $L(G^{'}) = L^{'}$ che non è ambigua.
Costruiamo G per L con le stesse produzioni di $G^{'}$ più $\$ \rightarrow \varepsilon$, applicata solo all'ultimo passo. □

Vogliamo scoprire se è vero il viceversa: per ogni L che ha una CFG non ambigua è vero che L è accettato da un DPDA? No, mostriamo infatti un controesempio:
$S \rightarrow 0S0|1S1|\varepsilon$ produce L_{ww^R} che non è accettato da alcun PDA

Esercizio 4.1. costruire un PDA per $L = \{0^n n | n \geq 1\}$: CONTROLLARE LINGUAGGIO

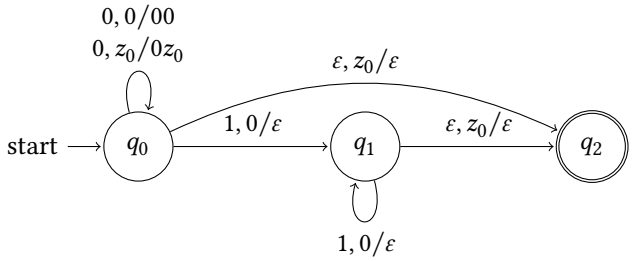


Osserviamo che questo è un DPDA e L è accettato sia per stato finale che per stack vuoto. Osserviamo anche che L ha la proprietà di prefisso:

$$y = 0^n 1^n \in L, x \neq y$$

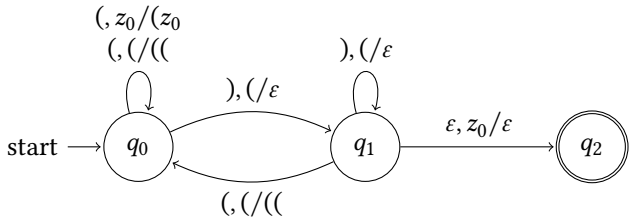
e x è prefissa di y e quindi x ha un numero di 0 diverso da quelli di 1 e quindi $x \notin L$

Esercizio 4.2. costruire un PDA per $L = \{0^n n | \, n \geq 0\}$:

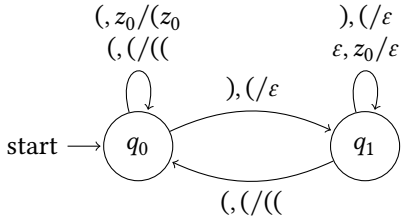


Ora la stringa vuota appartiene al linguaggio. Il linguaggio non ha la proprietà del prefisso. Si può dimostrare che non esiste un DPDA per L.

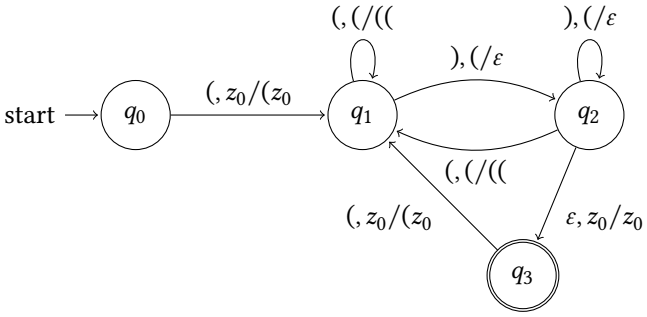
Esercizio 4.3. considero il linguaggio generato da $B \to (BB)|(B)|()$.
Il linguaggio ha la proprietà di prefisso perché (BB). Se fosse B -> BB allora non lo avrebbe, perché potremmo costruire le stringhe () e ()().
Abbiamo quindi il DPDA per stato finale:



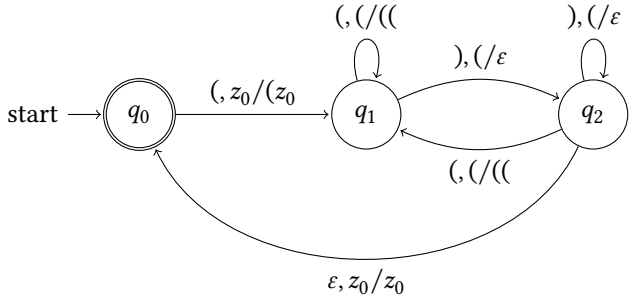
volendo accettare solo per stack vuoto:



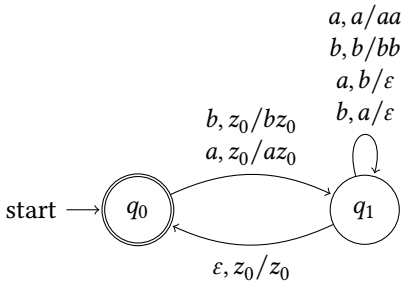
Esercizio 4.4. Si ha $B \to BB|(B)|()$. La stringa vuota non appartiene al linguaggio ma L non ha la proprietà del prefisso (come abbiamo mostrato nell'esercizio precedente). Possiamo quindi realizzare solo un DPDA per stato finale:



Esercizio 4.5. Si ha $B \to BB|(B)|\varepsilon$ ho un DPDA per stato finale perché L non ha la proprietà del prefisso:



Esercizio 4.6. sia $L = \{q \in \{a,b\}^* | \textit{numero uguale di a e b}\}$ DPDA per stato finale (L non ha la proprietà di prefisso):

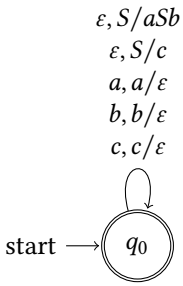


Esercizio 4.7. sia dato il CFL $L = \{a^n cb^n | n \geq 0\}$
è generato da :

$$G = (\{S\}, \{a, b, c\}, P, S)$$

$$S \to aSb|c$$

si ha il seguente automa non deterministico:



con:

$$\delta(q, \varepsilon, S) = \{(q, aSb), (q, c)\}$$

$$\delta(q, a, a) = \{(q, \varepsilon)\}$$

$$\delta(q, b, b) = \{(q, \varepsilon)\}$$

$$\delta(q, c, c) = \{(q, \varepsilon)\}$$

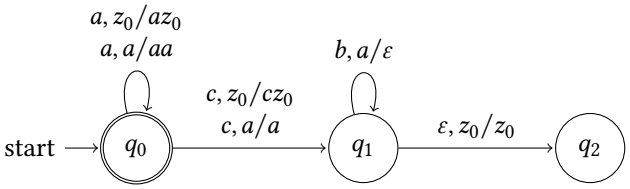
mostro la derivazione per n=3, aaacbbb e il comportamento dell'automa:

$$(q, aaacbbb, S) \vdash (q, aaacbbb, aSb) \vdash (q, aacbb, Sb) \vdash (q, aacbb, aSb)$$

$$\vdash (q, acbbb, Sbb) \vdash (q, acbbb, aSbbb) \vdash (q, cbbb, Sbbbb) \vdash (q, cbbb, cbbb)$$

$$\vdash (q, bbb, bbb) \vdash (q, bb, bb) \vdash (q, b, b) \vdash (q, \varepsilon, \varepsilon) \to \ accetta$$

si ha che vale la proprietà del prefisso e si ha il seguente DPDA:



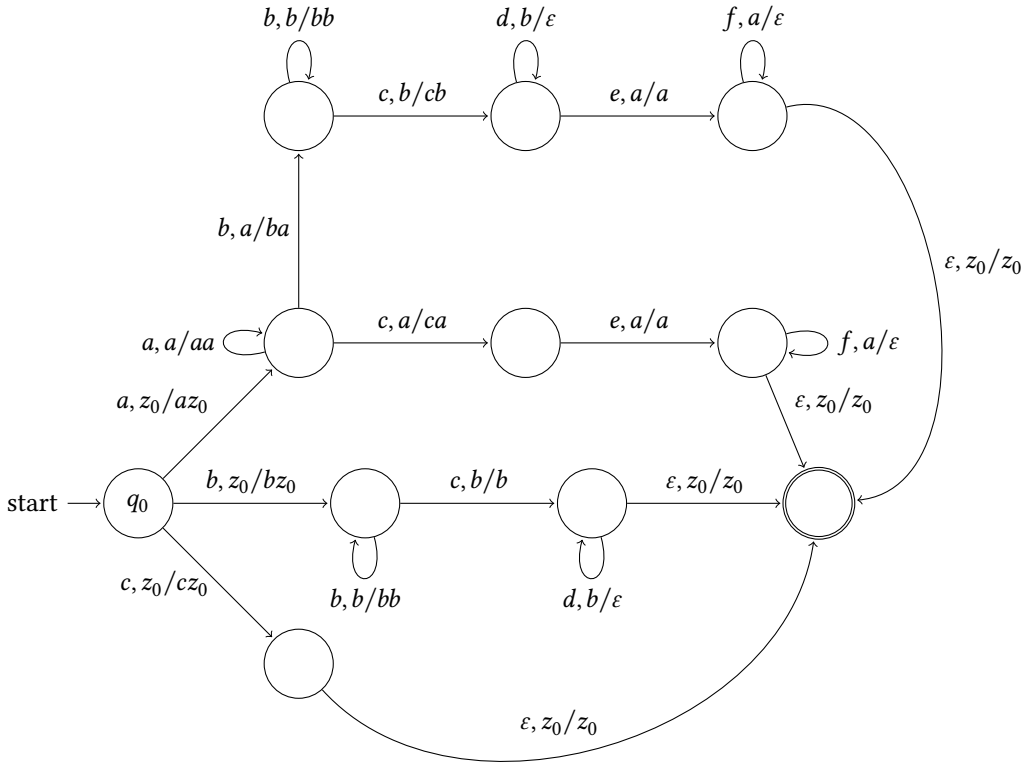
Esercizio 4.8. realizzare il pda per $L = \{a^n b^m c d^m e f^n | n, m \geq 0\}$; si ha quindi:

$$a^n c e f^n \ n > 0$$

$$b^m c d^m e \ m > 0$$

$$a^n b^m c d^m e f^n \ n, m > 0$$

$$ce \ n = 0 = m$$


b, b/bb
d, b/ε
f, a/ε
c, b/cb
e, a/a
b, a/ba
ε, z0/z0
a, a/aa
c, a/ca
e, a/a
f, a/ε
a, z0/az0
ε, z0/z0
start
q0
b, z0/bz0
c, b/b
ε, z0/z0
c, z0/cz0
b, b/bb
d, b/ε
ε, z0/z0

Computabilità

5 Macchine di Turing

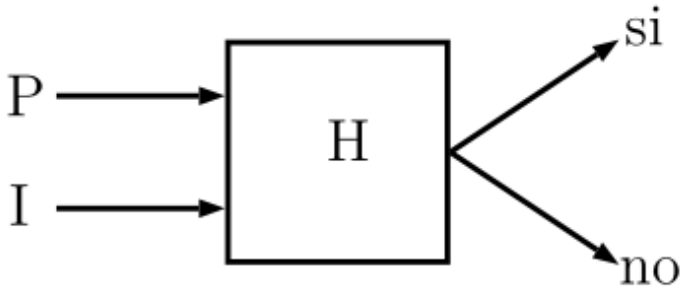
Nascono in risposta al problema di Hilbert, che si chiedeva se esiste un algoritmo per dimostrare teoremi. Non ci sono comunque funzioni non calcolabili con le macchine di Turing: $f : \mathbb{N} \to \{0,1\}$ $s^{|\mathbb{N}|} > \mathbb{N}$ Siamo interessati a studiare la calcolabilità (computabilità). Esiste un programma che calcola una certa funzione? Se non esiste, f è indecidibile. Se sì, in quanto tempo è calcolabile (complessità computazionale), quante mosse e quante celle del nastro sono necessarie?
vediamo due esempi:

Esempio 5.1 data una CFG $G = (V, T, P, S)$ stabilire se è ambigua. Questo problema è indecidibile, cioè non esiste nessun algoritmo che dia una risposta.

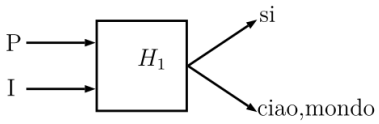
Esempio 5.2 Problema "ciao mondo": dato un sorgente di un programma in C/Java/ , i primi undici caratteri stampati dal programma sono "ciao, mondo"? Consideriamo il seguente algoritmo, che prende in input un numero intero N e stampa "ciao mondo" sse $\exists x, y, z \in \mathbb{N}$ tali che $x^n + y^n = z^n$.

quest'ultimo problema è indecidibile:

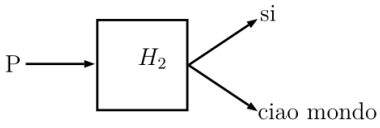
Dimostrazione. procediamo per assurdo.
Supponiamo che esista un algoritmo che risolva il problema. Assumiamo che P sia corretto (e quindi compilabile) e che stampi solo stringhe sulla console:



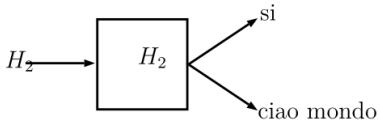
quindi:



suppongo di dare in input solo programmi, sse H_1 stampa con $P = I$:



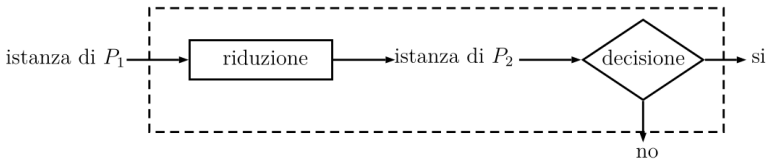
Allora posso pensare di dare ad H_2 il programma H_2 stesso:



quindi:
suppongo di dare in input solo programmi, sse H_1 stampa con $P = I$: Allora posso pensare di dare ad H_2 il programma H_2 stesso: Ma questo è assurdo: perché H_2 dice si quando in realtà stampa ciao mondo e viceversa, quindi non funziona. Ma allora in conclusione H non può esistere (perché i passaggi da H a H_1 e da H_1 a H_2 sono leciti) e quindi il problema non è decidibile. □

5.1 Riduzioni

Supponiamo P_2 indicibile. Considerato un nuovo problema P_2, vorrei stabilire se anch'esso è indecidibile. Vorremmo quindi fare una "riduzione" da



P_1 a P_2:

Supponiamo per assurdo che P_2 sia decidibile. Allora esiste l'algoritmo di decisione. Esistendo il processo di riduzione, allora l'intero rettangolo tratteggiato sarebbe un algoritmo di decisione per P_1, assurdo perché P_1 è indecidibile per ipotesi.

Esempio 5.3 P_2 (Problema della chiamata): dato un programma Q, esso chiama il metodo $m()$?
Dobbiamo quindi trovare un algoritmo di riduzione. Procediamo:

- rinominiamo tutte le istanze di m in qualcosa d'altro
- aggiungiamo un metodo m che non verrà quindi mai chiamato
- modifichiamo questo programma in modo che salvi in un array i primi 11 caratteri che stampa a video
- modifichiamo questo programma in modo che se sfora gli 11 caratteri e controlla tali caratteri. Se sono esattamente "ciao mondo", chiama m

quindi anche questo problema è indecidibile

È importante che la riduzione sia fatta nel verso corretto. Se facessimo la riduzione da P_2 a P_1, staremmo mostrando che se P_1 è decidibile allora P_2 è decidibile. Ma sappiamo che P_1 è indecidibile, quindi non staremmo

dimostrando niente. Nel caso contrario stiamo invece dicendo che se P_2 è decidibile allora P_1 è decidibile. Ma essendo P_1 indecidibile, allora l'antecedente è falsa e quindi P_2 non può essere decidibile, quindi è indecidibile. Diamo ora la definizione formale della Macchina di Turing:

Definizione 5.1 Si definisce MdT:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

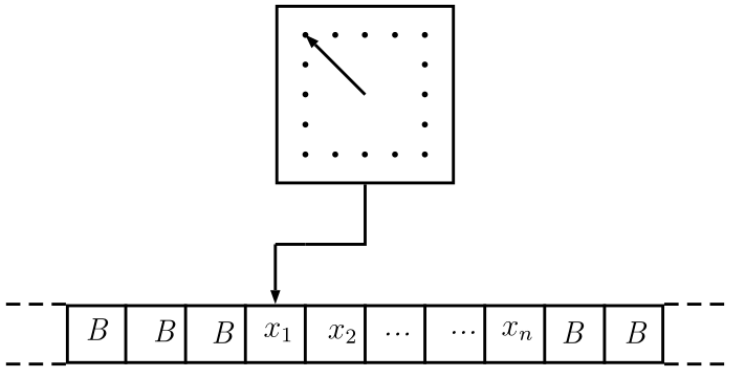
con:

- Q insieme finito non vuoto di stati
- Σ insieme finito non vuoto di simboli di input
- Γ insieme finito non vuoto di simboli sul nastro
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- $q_0 \in Q$ stato iniziale
- $B \in \Gamma \backslash \Sigma$ simboli di blank
- $F \subseteq Q$ insieme di stati finali

inoltre si ha che:

$$\Sigma \subseteq \Gamma \ \ e \ \ \Gamma \cap \Sigma \neq \emptyset$$

quindi si ha per esempio questa rappresentazione, per l'input $x_1, \dots, x_n \in \Sigma^*$:



Questa era una definizione deterministica, passiamo ora ad una definizione *istantanea*:

Definizione 5.2 suppongo di avere lo stato q, il nastro $x_1, \dots, x_n$ con la testina x_i. SI ha:

$$ID : \ x_1 x_2, \dots q x_i x_{i+1} \dots x_n$$

e suppongo $Q \cap \Gamma = \emptyset$ senza perdere generalità e uso la simbologia dei $\vdash$ introdotta coi PDA.
Se $\delta(q, x_i) = (p, y, L)$ allora $\vdash \ x_1 x_2 \dots x_{i-2} p x_{i-1} y x_{i+1} \dots x_n$.
Si hanno dei casi particolari:

- se $i = 1$ si ha $q x_1 x_2 \dots x_n \vdash pBy x_2 \dots x_n$
- se $i = n$ $y = B$ si ha $q x_1 x_2 \dots q x_n \vdash x_1 x_2 \dots x_{n-2} p x_{n-1}$

a destra abbiamo invece: Se $\delta(q, x_i) = (p, y, R)$ allora $\vdash$ $x_1 x_2 \dots x_{i-1} Y p x_{i+1} \dots x_n$.
Si hanno dei casi particolari:

- se $i = 1$ si ha $q_1 x_2 \dots x_{n-1} \vdash ypB$
- se $i = n$ $y = B$ si ha $q x_1 x_2 \dots q x_n \vdash p x_2 \dots x_n$

Esempio 5.4 sia $L = \{0^n 1^n | n \geq 1\}$ un CFL. vediamo una tabella per capire l'azione della macchina:

	0	1	x	y	B
q_0	(q_1, x, R)	–	–	(q_3, y, R)	–
q_1	$(q_1, 0, R)$	(q_2, y, L)	–	(q_1, y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, x, R)	(q_2, y, L)	–
q_3	–	–	–	(q_2, y, L)	(q_4, B, R)
q_4	–	–	–	–	–

ovvero, per esempio:

$$000111$$

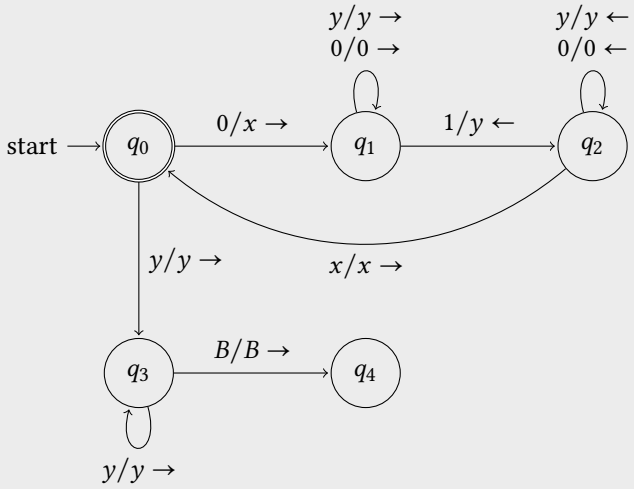
$$x00111$$

$$x00y11$$

$$xx0yy1$$

$$xxxyyy$$

ma la tabella non è comodissima, usiamo quindi i diagrammi di transizione:



e per l'input 0010 si hanno i seguenti passi:

$$q_0 010 \vdash x q_1 010 \vdash x0 q_1 10 \vdash x q_2 0 y0 \vdash q_2 x0 y0$$

$$\vdash x x q_2 y0 \vdash x x y q_1 0 \vdash x x y0 q_1 B$$

definiamo quini il linguaggio accettato da una macchina di Turing:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

$$L(M) = \{w \in \Sigma^* | q_0 w \overset{*}{\vdash} \alpha p \beta \ p \in F \ \alpha, \beta \in \Gamma^*\}$$

La classe dei linguaggi accettati dalle MdT sono i ricorsivamente enumerabili (RE). Le MdT possono anche calcolare le funzioni:

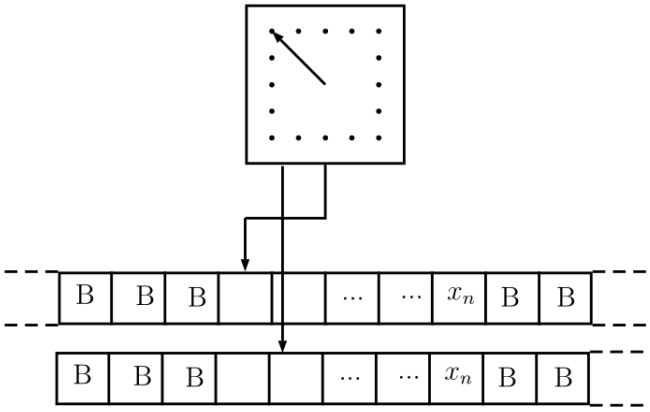
Esempio 5.5

$$f(n, m) = m - n = max(m - n, 0) = \begin{cases} m - n & m \geq n \\ 0 & m < n \end{cases}$$

quindi se l'input fosse $0^m 10^n$ si avrebbe in output 0^{m-n}

si ha che se la MdT accetta, allora si ferma e che se la MdT non accetta, allora non si può dire se si ferma oppure no. Se non si ferma: linguaggi ricorsivamente enumerabili. Se si ferma sempre, otteniamo una sottoclasse che sono i linguaggi ricorsivi. Inoltre il problema dell'arresto, **halting problem**, è indecidibile e quindi i linguaggi ricorsivamente enumerabili sono quindi semidecibili.
Si possono avere delle estensioni della macchina di Turing:

- macchina non deterministica; anziché una sequenza di ID avremmo un albero. Non si aggiunge comunque potenza di calcolo, perché posso simulare una nondet con una det che esplora questo albero
- MdT multinastro Hanno un numero finito di nastri. In una mossa guarda lo stato del controllo e il simbolo sotto ciascuna delle testine, si muove in un nuovo stato, scrive un nuovo simbolo per ogni nastro e si muove indipendentemente su ogni nastro:



si ha il seguente teorema:

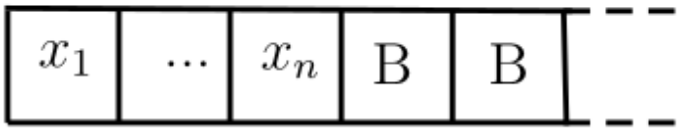
Teorema 5.1 ogni linguaggio accettato da una MdT multinastro è Ricorsivamente Enumerabile

Ora possiamo definire in maniera più precisa cosa si intendeva per simulazione di una macchina nondet: uso due nastri, il primo con l'input e il secondo gestito come una coda di ID da elaborare

Teorema 5.2 Se Mn è una NTM (Nondeterministic Turing Machine) allora esiste una DTM (Deterministic Turing Machine) tale che il linguaggio accettato dalla NTM è uguale a quello accettato dalla DTM

5.2 Restrizioni delle macchine di Turing

Consideriamo una MdT con nastro semiinfinito (ovvero è infinito solo da un lato):

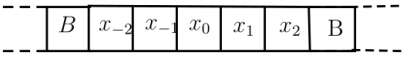


si ha il seguente teorema:

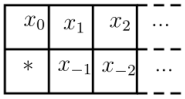
Teorema 5.3 Ogni linguaggio accettato da una DTM M2 è anche accettato da una DTM M1 tale che:

- La testina di M1 non va mai a sinistra della posizione iniziale
- M1 non scrive mai un Blank:

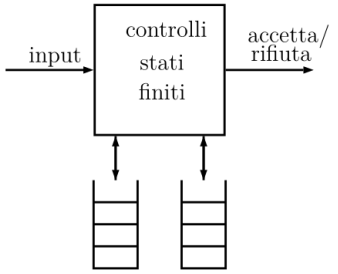
$$B^{'} \notin \Gamma \text{ e scrive } B^{'} \text{ al posto di } B$$



e con un nastro semiinfinito:



dove si ha l'alfabeto $\Gamma \times (\Gamma \cup \{*\})$
Macchine multi stack, che sono DPDA con un controllo a stati finiti e un numero finito di pile:

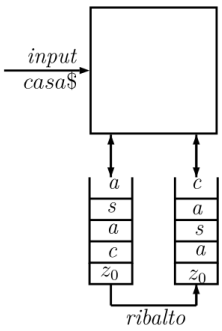


Consideriamo ora una MdT tradizionale con il nastro:

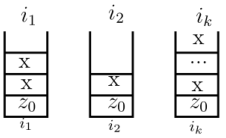
e si ha:

$$\delta(q, a, x_1, x_2, \dots, x_k) = (p, \gamma_1, \gamma_2, \dots, \gamma_k)$$

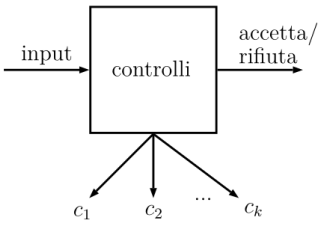
si ha un'ulteriore restrizione: Supponiamo che i simboli sulle pile possano essere solo x o z_0. Si riescono ancora a simulare le macchine di Turing



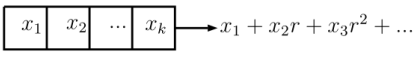
si ha un'ulteriore restrizione: Supponiamo che i simboli sulle pile possano essere solo x o z_0. Si riescono ancora a simulare le macchine di Turing



e il modello diventa:



dove possiamo modificare i contatori, che sono numeri naturali, a nostro piacimento "aggiungendo o togliendo X". Vediamo perché questo modello simula una MdT. Supponiamo che l'alfabeto del nastro contenga $R-1$ simboli. Allora:



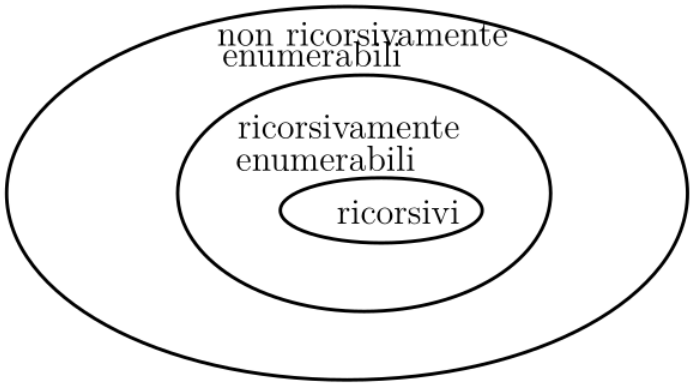
Così come codifico il nastro posso codificare una pila. Per togliere x_1 in cima alla pila devo dividere il numero per R e prendere il resto. Per aggiungere un nuovo simbolo, moltiplico per R e aggiungo il nuovo simbolo. Per modificare il simbolo, basta aggiungere la differenza tra il nuovo e la cima precedente. È immediato farlo con tre contatori (usandone uno di appoggio). Per fare tutto ciò bastano due contatori (e quindi due pile) dove nel primo è codificato come:

$$2^{c_1}3^{c_2}5^{c_3}$$

Per i linguaggi si ha quindi:

- linguaggi ricorsivi → decidibili e MdT si ferma sempre
- linguaggi ricorsivamente enumerabili → semidecidibili (ma comunque indecidibili) e MdT si ferma se accetta ma potrebbe non fermarsi
- linguaggi non ricorsivamente enumerabili → indecidibili e ∄ MdT

quindi, insiemisticamente:



Esempio 5.6 vediamo un linguaggio non ricorsivamente enumerabile. $w \in \{o, 1\}^*$ biiezione tra stringhe e numeri:

w_1	w_2	w_3	w_4	w_5	w_6	...
1	2	3	4	5	6	...
ε	0	1	00	01	10	..

Consideriamo però le due stringe 00101 e 0101: se le leggessi come numeri binari sono entrambi 5, quindi abbiamo perso la biiezione. Forziamo quindi l'interpretazione delle stringhe mettendoci davanti un 1. Codifichiamo anche una MdT in binario:

$$M = (Q, \{0, 1\}, \Gamma, \delta, q_1, B, \{q_2\})$$

$$Q = \{q_1, \dots, q_r\}$$

$$\Gamma = \{x_1, \dots, x_s\} = \{0, 1, B, \dots\}$$

$$direzioni: \quad L \to D_1 \; R \to D_2$$

$$\underbrace{\delta(q_i, x_j) = (q_k, x_1, D_m}_{0^i 1 0^j 1 0^k 1 0^l 1 0^m}$$

$$Cod_1 11 Cod_2 11 Cod_3 \dots = Cpd(\delta)$$

Esempio 5.7 sia $M = (\{q_1, q_2, q_3\}, \{o, 1\}.\{0, 1, B\}, \delta, q_1, B, q_2)$ con:

$$\overbrace{0}^{q_1}\,1\,\overbrace{00}^{x_2}\,1\,\overbrace{000}^{q_3}\,1\,\overbrace{0}^{x_1}\,1\,\overbrace{00}^{D_2}$$

$$\delta(q_1, 1) = (q_3, 0, R)$$

$$\delta(q_3, 0) = (q_1, 1, R)$$

$$\delta(q_3, 1) = (q_2, 0, R)$$

$$\delta(q_3, B) = (q_3, 1, L)$$

Quindi ora possiamo dire che M è Mi, ovvero l'i-esima macchina di Turing. si ha però un problema: io posso permutare le delta, ottenendo numeri diversi, pur descrivendo la stessa macchina. Quindi la stessa macchina compare più volte nella sequenza infinita. Inoltre, data una stringa, non è detto che rappresenti una MdT. Inoltre:

$$Cod((M, W)) = Cod(M)111Cod(W)$$

definisco quindi L_d

$$L_d = \{w_i \in \{0, 1\}^* | \, w_L \in L(M_i)\}$$

e considero la tabella infinita:

		j					
	1	2	3	4	5	6	
	1	0	1	1	0	…	…
	2	1	1	0	0	…	…
	3	0	0	1	1	…	…
(Mi) i	4	0	1	0	1	…	…
	5	⋮	⋮	⋮	⋮	…	…
	6	⋮	⋮	⋮	⋮	…	⋱

Osserviamo la diagonale della tabella. Fanno parte di Ld solo le stringhe che hanno 0 sulla diagonale. Prendo in sostanza il complemento della diagonale. Se esistesse una MdT accettante, allora esisterebbe una riga uguale a tale complemento. Però tale complemento non è uguale a nessuna delle righe, perché differisce per l'i-esimo valore dall'i-esima riga (essendo il negato di tale valore!).

Teorema 5.4 Se L è ricorsivo, allora il complementare di L è ricorsivo.

Dimostrazione. Se L è ricorsivo, allora esiste una MdT M tale che $L = L(M)$. Costruiamo M' tale che $L(M'$ = complementare di L. □

Teorema 5.5

$$L, \overline{L} \in RE \rightarrow L, \overline{L} \in RIC$$

se un linguaggio e il suo complementare sono ricorsivamente enumerabili allora quel linguaggio è ricorsivo

Dimostrazione.

$$L \in RE \rightarrow \exists M_L \rightarrow L = L(M_L)$$

$$\widetilde{L} \in RE \rightarrow \exists M_{\overline{L}} \rightarrow \overline{L} = L(M_L)$$

□

Poiché ogni stringa o appartiene a L o al complemento di L, costruiamo una nuova macchina M che simula M_L e $M_{\overline{L}}$.
Prima o poi una delle due macchina deve fermarsi. Allora possiamo accettare o rifiutare.
Si ha quindi la seguente tabella per i vari casi:

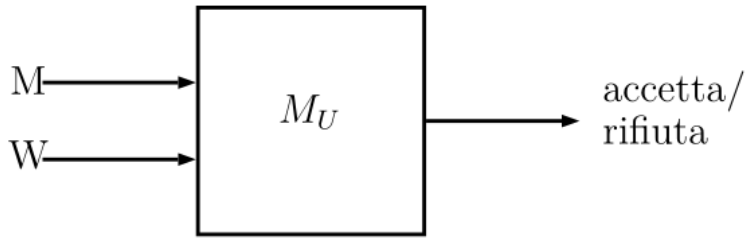
accettabile	Ric	RE	non RE
si	$L, \overline{L}$		
no	L	$\overline{L}$	
no	L		$\overline{L}$
no	$\overline{L}$	L	
no		$L, \overline{L}$	
si		L	$\overline{L}$
no	$\overline{L}$		L
si		$\overline{L}$	L
si			$L, \overline{L}$

5.3 Macchina di Turing Universale

Abbiamo già visto che si può codificare in binario una MdT, possiamo codificare anche la coppia MdT con il suo input:

$$Cod(M)111Cod(M)\ [M111W]$$

Possiamo quindi pensare a una MdT universale che sappia simulare qual-



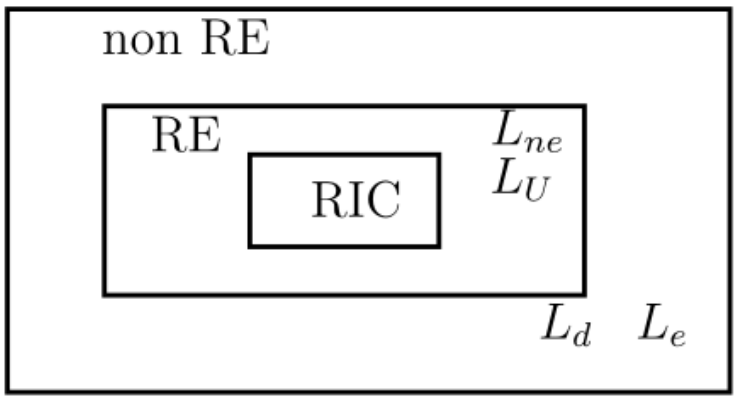
siasi altra MdT specificata come da codifica:

$$L_U\{(M, W)|W \in L(M)\}$$

La macchina universale corrisponde alla nostra idea di computer programmabile. Descriviamo questa macchina universale, che ha quattro nastri:

- **primo nastro:** $M111W$
- **secondo nastro:** nastro/codifica di M
- **terzo nastro:** stato/codifica di M
- **quarto nastro:** ausiliario

Il linguaggio universale è ricorsivamente enumerabile ma non ricorsivo (non potrebbe esserlo, visto che la macchina che simula potrebbe non fermarsi). Il diagramma delle classi dei linguaggi è quindi diventato il seguen-

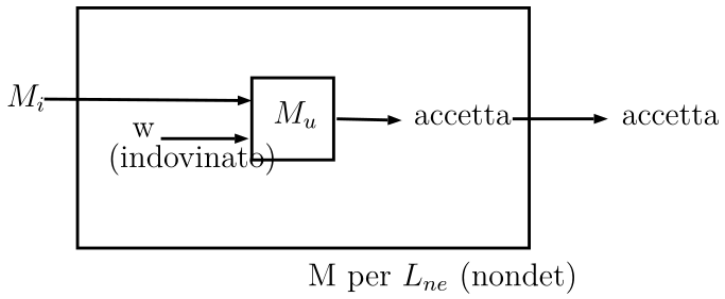


te:
Valgono ancora le riduzioni. Sapendo che P_1 è indecidibile (non RE o RE), posso fare una riduzione dalle istanze di P_1 alle istanze di P_2 mostrando che P_2 è indecidibile. Con la riduzione stiamo dicendo che P_2 è almeno difficile quanto P_1 (e non viceversa!). Notiamo che non è nemmeno necessario che tutte le istanze di P_2 siano "coperte" dal processo di riduzione.

Teorema 5.6

$$L_{ne} \in RE$$

Dimostrazione. Dobbiamo fare una riduzione da L_u a L_{ne} per mostrare che L_{ne} è ricorsivamente enumerabile. La riduzione è descritta da questo schema:



□

Teorema 5.7

$$L_{ne} \notin RIC$$

Dimostrazione. infatti essendo l_e il complementare di l_{ne}, per i teoremi visti in precedenza l_e non può essere ricorsivo né può essere ricorsivamente enumerabile. Segue che l_e è non RE □

Esempio 5.8 considero:

$$L_e = \{M | L(M) = \emptyset\}$$

$$L_{ne} = \{M | L(M) = \emptyset\}$$

Parser

Analisi lessicale 6

6.1 Parser Top-down

L'idea intuitiva dei **parser top-down** è estremamente semplice: costruire l'albero di derivazione a partire da un albero D formato dalla sola radice ed estendere l'albero con l'espansione di una produzione alla volta, finchè non si ottiene esattamente il testo T.

Questa tipologia di parser si contrappone a quella dei **parser bottom-up** che invece partono dall'analisi del testo e costruiscono l'albero di derivazione partendo dalle foglie, finchè non viene ricostruita la radice dell'albero. Questa tipologia era dominante in passato, quando i computer erano decisamente meno potenti e procedure efficienti, ma complesse, erano necessarie per effettuare il parsing in tempi ragionevoli. Negli ultimi anni si vede invece una tendenza verso i parser top-down, in quanto gli algoritmi utilizzati sono semplici, sia da descrivere che da implementare, sebbene non abbiano le stesse garanzie di efficienza di alcuni parser bottom-up.

6.2 Parser LL

6.3 Parser LRk

6.4 Parsing Expression Grammars

Bibliografia

[1] Keith D Cooper e Linda Torczon. *Engineering a compiler*. Morgan Kaufmann, 2022 (citato a pag. 2).

Lista dei simboli

Lista dei simboli.

ϵ	Parola vuota
Σ	Alfabeto
A, B	Insieme generico
L	Linguaggio
w, x, y, z	Parole o stringhe