Appunti di
Linguaggi Formali

A cura di Roberta Prendin
robertaprendin@gmail.com

Appunti delle lezioni di Calcolabilità e Linguaggi Formali *tenute dal professor* Antonino Salibra*, anno accademico 2013/2014. Trattandosi di appunti saranno presumibilmente presenti giri di parole, errori e imprecisioni di dubbio gusto. Penso riuscirete a sopravvivere ugualmente.*

Introduzione.

Qualunque programma è scritto secondo regole e formalismi, sia sintattici (i criteri che regolano il formalismo) che semantici (il significato dato al programma): perfino compilare un codice significa vedere se rispetta le regole formali del linguaggio in cui è scritto. Studieremo ora alcune classi di linguaggi in cui si tende a far rientrare la maggior parte dei linguaggi di programmazione.

Definizione di *linguaggio L*. Un linguaggio, dal punto di vista sintattico, è un **insieme di sequenze di simboli**, senza alcun riferimento a ciò che tali sequenze possono significare. Formalmente, dato A, cioè un insieme finito di simboli, A^* è l'insieme formato da tutte le stringhe ottenibili concatenando tra loro i simboli di A. Un **linguaggio L è un sottoinsieme di A^***, cioè è un sottoinsieme formato da alcune delle stringhe generabili a partire dall'alfabeto A. Notare che ogni stringa ha sempre una **lunghezza:** λ è la stringa vuota, cioè di lunghezza 0, mentre le stringhe di lunghezza *k* sono tutte appartenenti all'insieme A^k. Se Se $k = 0$, allora $A^0 = \{\lambda\}$.

Relazione tra linguaggi e numeri naturali. Dato che A è un insieme *finito* di simboli, allora A^* sarà *numerabile*: esiste cioè una **corrispondenza biunivoca tra A^* e N** tale per cui ogni stringa è codificata da uno e un solo numero naturale e viceversa. Per esempio, se poniamo che A sia formato da n simboli ordinati…

$$\Sigma = \{a_1, \ \dots, \ a_n\} \quad con\ a_1 < \cdots < a_n$$

…esisterà una funzione $g: simbolo \rightarrow N$ che associa ad ogni simbolo di A un numero naturale: ogni stringa costruita a partire da A avrà un corrispondente valore naturale associato, ottenuto seguendo l'ordine di priorità dei caratteri:

$$\lambda \ \ a_1 \ \ a_2 \ \ \dots \ \ a_n \ \dots \ a_1a_1 \ \ a_1a_2 \ \ \dots \ \ a_2a_1 \ \dots \ \ a_na_n \dots$$

In questo caso, $a_ia_j < a_ka_r$ se e solo se una delle seguenti possibilità è vera:

$$i < k \quad \text{oppure} \quad i = k \, e \, j < r$$

Stando questa relazione tra numeri naturali e stringhe, si potrà parlare indifferentemente di insiemi di numeri come di linguaggi, e le proprietà degli uni saranno anche proprietà degli altri. Questa conseguenza è particolarmente interessante se si considerano le proprietà di ***decidibilità e semidecidibilità*** dei linguaggi:

- Un linguaggio è *semidecidibile* se il corrispondente insieme di naturali è semidecidibile.
- Un linguaggio è *decidibile* se il corrispondente insieme di naturali è decidibile.

Altrimenti, possiamo anche pensare la decidibilità in termini di programmi:

- Un linguaggio è *semidecidibile* se esiste un algoritmo che, data una stringa, dice "sì" se la stringa appartiene al linguaggio, ma non sa rispondere altrimenti.
- Un linguaggio è *decidibile* se esiste un algoritmo che, data una stringa, restituisce "sì" o "no" a seconda che la stringa appartenga o non appartenga al linguaggio.

In questo corso saremo interessati ad insiemi *almeno* semidecidibili (o altrimenti detti *ricorsivamente enumerabili*), dato che solo i linguaggi semidecidibili e decidibili sono rappresentabili nel calcolatore.

Operazioni sulle stringhe.

Le operazioni che possiamo compiere sulle stringhe sono:

- **Lunghezza di una stringa**: $\mathrm{A}^* \to N$. Data una stringa, ne restituisce la lunghezza. Per esempio la lunghezza di $casa$ è 4, mentre la lunghezza di λ è 0.
- **Concatenazione (su stringhe)**: date due stringhe, permette di concatenarle. $\cdot : A^* \times A^* \to \mathrm{A}^*$.

Per esempio, date le stringhe $x = casa$ e $y = albero$, $x \cdot y = xy = casaalbero$ (il simbolo della concatenazione può essere omesso). Gode delle seguenti proprietà: date le stringhe x, y, z:

- **Associativa**: $x(yz) = (xy)z$
- **Identità**: $\lambda x = x\lambda$, dove quindi λ è l'elemento d'identità.
- **Somma delle lunghezze**: $|xy| = |x| + |y|$.

Notare che la concatenazione **non è commutativa**: $xy \neq yx$.

- **Selezione di una sottostringa**. Permette di selezionare solamente parte di una stringa. Per esempio $(canestro)_{3,5} \to nes$. L'indice parte da 1.
- **Punto fisso sulle stringhe**. Sia dato il simbolo a dell'alfabeto A: tale simbolo è *punto fisso* di una certa stringa se esiste una funzione f tale che $f(a) = a$. In altre parole un punto fisso è un simbolo che la funzione f trasforma in se stesso.

Esempio #1. Ogni simbolo di una stringa palindroma è un punto fisso della funzione che "ribalta" la stringa: il carattere m di ama rimane uguale a se stesso anche se applichiamo la funzione "ribalta" alla stringa ama, e lo stesso vale per i due caratteri a.

Esempio #2. La **funzione successore** non ha punti fissi perché $S(x)$ non è mai x ma $x + 1$.

Esempio #3. La **funzione predecessore**, se definita sui naturali ha punto fisso in 0 dato che il predecessore di 0, nei naturali, è sempre 0. Lavorando invece sugli interi il predecessore non ha punti fissi.

Proprietà #1 (funzionali). Consideriamo una funzione *f* definita in modo ricorsivamente primitivo:

$$f(x) = \begin{cases} 1, & x = 0 \\ 2 * f(x-1), & x \neq 0 \end{cases}$$

Quando abbiamo funzioni ricorsive le loro soluzioni sono *punti fissi di funzioni che trasformano funzioni in funzioni,* cioè dei cosiddetti ***funzionali***. Per esempio definiamo il funzionale ***t*** che, data la funzione *f* e il suo argomento, trasforma f in una nuova funzione. Poniamo che la parte destra di t sia equivalente alla definizione ricorsiva di *f*:

$$t(f)(x) = \begin{cases} 1, & x = 0 \\ 2 * f(x-1), & x \neq 0 \end{cases}$$

Poniamo poi che la funzione *f* sia la funzione *Successore*, S:

$$t(S)(x) = \begin{cases} 1, & x = 0 \\ 2 * S(x-1), & x \neq 0 \end{cases}$$

Ora, l successore di $x - 1$, con $x \neq 0$, è comunque equivalente a x. Di conseguenza abbiamo che:

$$t(S)(x) = \begin{cases} 1, & x = 0 \\ 2x, & x \neq 0 \end{cases}$$

Ora, $t(S)$ è uguale o non è uguale a S? No, perché il successore di 0 è 1, ma il successore di un valore diverso da 0 non sarà uguale al suo doppio. Prendiamo invece la funzione $f(x) = 2^x$: $t(f) = f$? Verifichiamolo. Prima scriviamo $f(x)$ in modo ricorsivo:

$$f(x) = \begin{cases} 1 & x = 0 \\ 2^{f(x-1)} = 2 * 2^{x-1} = 2^x & x \neq 0 \end{cases}$$

Ora, $t(f)(x)$:

$$t(f)(x) = \begin{cases} 1 & x = 0 \\ 2^{f(x-1)} = 2 * 2^{x-1} = 2^x & x \neq 0 \end{cases}$$

…per cui $t(f) = f$. Il **procedimento** quindi è:

- Definisco la funzione in modo ricorsivo.
- Ricopio la parte destra della funzione ricorsiva come "argomento" del funzionale t (detta *funzionale* perché trasforma funzioni in funzioni).
- Verifico che questa parte che ho aggiunto equivalga alla funzione *f* di partenza; in quel caso, t(f)=f.

Definizioni e operazioni sui linguaggi.

Il linguaggio più povero di stringhe è il **linguaggio vuoto**:

$L = \{\}$ //linguaggio più povero di stringhe

Il linguaggio immediatamente maggiore è il linguaggio contenente solo la stringa vuota:;

$L = \{\lambda\}$

Sui linguaggi valgono le ***operazioni insiemistiche***, dato che ogni linguaggio è un *insieme* di stringhe:

- **Unione** di due linguaggi. Per esempio posso prendere i linguaggi sintatticamente corretti di C e di Java, unirli e ottenere i programmi sintatticamente corretti in C o in Java. Permette di prendere le stringhe di due linguaggi e di unirle in un unico linguaggio.
- **Intersezione** di due linguaggi. L'intersezione di due linguaggi è formata dalle stringhe comuni ai due linguaggi.
- **Complementazione**. Dato il linguaggio L, complementandolo posso ottenere tutte le stringhe che non appartengono ad L. Per esempio, se prendo l'insieme delle stringhe sintatticamente corrette in C, il suo complementare è l'insieme delle stringhe non sintatticamente corrette in C.

Esistono anche le **operazioni *proprie*** dei linguaggi.

- **Concatenazione (di due linguaggi).** Dati due linguaggi, L_1 e L_2, posso concatenarli:

$$L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$$

- **Chiusura o *concatenazione interna***. Dato un linguaggio L contenente un certo numero di stringhe, L^* è l'insieme delle **stringhe ottenibili concatenando a piacere stringhe di L**. L^* contiene *sempre* anche la stringa di lunghezza 0, λ.

Esempio #1. Sia $L = \{Cane, Casa\}$, allora L^* = { λ , Cane, Casa, CaneCasa, CasaCane CaneCaneCaneCane, CasaCasaCasaCasaCasa, … }.

Esempio #2. Notare che $L^* = L^1, L^2, L^3$ – dove L^n rappresenta l'insieme formato dalle stringhe di linguaggio L concatenate tra di loro esattamente *n* volte. Per esempio, dato $L = \{Cane, Casa\}$, L^3 è l'insieme delle stringhe ottenute concatenando stringhe di L esattamente 3 volte (vi appartengono "$CaneCaneCane$", "$CaneCaneCasa$" ma non "$CaneCasa$" o "$CaneCasaCaneCasa$"). Nel caso il linguaggio L comprenda anche la stringa vuota, cioè $L = \{Cane, Casa, \lambda\}$, allora a L^3 apparterranno $CaneCasa$, $CaneCane$, $CasaCasa$, $CasaCane$… perché tutte stringhe concatenate alla stringa λ; vi apparterrà anche la stringa vuota λ perché ottenuta concatenando la stringa vuota tre volte: $\lambda\lambda\lambda$.

Rappresentazione dei linguaggi.

Dopo aver definito cosa sia un linguaggio, dobbiamo capire come *rappresentare* un linguaggio in maniera formale. Per i linguaggi con un **numero finito di stringhe** la questione si risolve semplicemente elencando le stringhe in questione; se invece il linguaggio contiene un numero infinito di stringhe, le strategie sono tre:

- Rappresentare il linguaggio come l'insieme delle stringhe generate a partire da una certa grammatica (**metodo generativo**: si parte da un ***token* iniziale** e si prosegue applicando una serie di regole che generano tutte le espressioni sintatticamente corrette).
- Si costruisce un algoritmo o un automa che semidecide la stringa in input, stabilendo se appartenga al linguaggio (**metodo riconoscitivo**: si riconosce la stringa).
- Uso di equazione ricorsiva (**metodo algebrico**).

Nelle prossime pagine ci concentreremo sui metodi *generativo e riconoscitivo*.

Il metodo generativo

Il metodo generativo permette di rappresentare un linguaggio generandolo a partire da un *token* iniziale, seguendo una serie di *regole di sintassi* e di *relazioni tra le regole di sintassi* che prescindono completamente dal significato di ciò che si genera (**grammatica generativa**). Lo stesso linguaggio italiano può essere rappresentato in questo modo: la *frase* può essere suddivisa in una serie di parti ben definite dalla grammatica e in relazione tra loro (soggetto, predicato nominale, predicato verbale, complemento, nome comune, nome proprio, etc). I *token* (detti anche *categorie sintattiche*) della lingua italiana sono i seguenti:

Frase::= Soggetto Predicato

Soggetto::= Frase nominale (cioè senza verbo!)

Frase nominale::= nome proprio | articolo nome comune

Predicato::= verbo | verbo frase nominale

Nome proprio::= Alberto | Giovanni

Verbo::= guarda | abbaia

Nome comune::= cane | tavola

Notare che:

- | indica "oppure";

- I simboli sottolineati sono detti *non terminali.*
- I simboli non sottolineati sono detti *terminali,* dato che non possono essere ridotti ad altro.

Alcuni esempi di stringhe ottenute attraverso questa grammatica sono:

Alberto guarda Giovanni

La tavola abbia la cane

Entrambe sono frasi che rispettano la grammatica, sebbene solo la prima sia semanticamente corretta. L'aspetto "generativo" della grammatica entra in gioco quando c'accorgiamo che ogni frase – e, per estensione, l'intero linguaggio – può essere generata a partire dai *token* attraverso una serie di ***riscritture*:** Alberto guarda Giovanni diventa:

Frase → Soggetto Predicato → Frase nominale Predicato → Nome proprio Predicato → Alberto Predicato → Alberto Verbo Frase nominale → Alberto guarda Frase nominale → alberto guarda Nome proprio → Alberto guarda Giovanni.

Notare che vale il *non determinismo*: a partire da Frase possiamo scegliere un cammino qualsiasi e qualunque risultato sarà corretto sintatticamente anche se magari non conforme alla frase che volevamo ottenere. $Alberto\ guarda\ Giovanni$ è infine una stringa di **lunghezza 3**, perché 3 sono le parti terminali di cui è composta.

Definizione di grammatica.

Una grammatica G è definita da ***quattro* insiemi finiti e decidibili** (ossia un elemento *o* appartiene *o* non appartiene a tale insieme, senza dubbi)**:** $G = (NT, T, S, P)$

- NT insieme di simboli **non terminali**; sono indicati con la **lettera maiuscola**.
- T insieme di simboli **terminali**; sono invece indicati con la **lettera minuscola**.
- S simbolo **iniziale**, l'equivalente di Frase nell'esempio precedente.
- P insieme delle **produzioni** (cioè delle regole della grammatica, come $a ::= b$).

Il processo di generazione consiste nell'applicazione, a partire dal simbolo iniziale, delle varie produzioni di cui la grammatica è formata: le produzioni funzionano da "regole di riscrittura" proprio come nell'esempio precedente. Ogni grammatica genera il rispettivo linguaggio: data la grammatica G, il linguaggio $L(G)$ generato dalla grammatica G è l'insieme delle stringhe di terminali che possono essere generate a partire dal *token* iniziale, S.

Grammatiche libere dal contesto.

Le grammatiche più semplici che possiamo incontrare sono quelle *dipendenti* o *libere* dal contesto. È facile capire in che cosa si differenziano: la produzione $a ::= b$ può essere applicata in qualunque punto di una stringa che presenti il simbolo a, in quanto tale simbolo non è accompagnato da simboli né a sinistra né a destra. Perciò:

$a ::= b$

. $xay \rightarrow xby$

Se invece prendiamo la produzione $aA ::= abacABc$, questa può essere applicata alla stringa $aaaAbaaAAcAB$ solo laddove il simbolo terminale a è seguito dal simbolo non terminale A:

$aA ::= abacABc$

$aaaAbaaAAcAB \rightarrow aaabacABcbaaAAcAB$

Un'altra grammatica interessante è quella *regolare* o *lineare destra*, che presenta cioè produzioni in cui il simbolo non terminale è sempre a destra di un simbolo terminale:

$A ::= aA$ //non terminale A è a destra del terminale a

Se il simbolo non terminale si trova a sinistra di un terminale, la grammatica è detta *lineare sinistra*:

$B ::= Bb$ //non terminale B è a sinistra del terminale b.

Dal linguaggio la relativa grammatica. Come già detto, ogni grammatica G genera il corrispondente linguaggio $L(G)$, formato dalle stringhe di terminali ottenibili applicando le varie produzioni di G al *token* iniziale S. In realtà si può fare anche l'opposto: a partire dal linguaggio possiamo ricavare la grammatica corrispondente e, con un passo successivo, ottenere un automa che riconosca stringhe di quel linguaggio. Una regola generale è:

$$Se\ non\ ho\ bisogno\ di\ memoria, la\ grammatica\ \text{è}\ libera\ dal\ contesto$$

Esempio #1. Poniamo d'avere il seguente linguaggio:

$$L = \{a^n b^n | \ n \geq 0\}$$

Si tratta di un linguaggio che contiene le stringhe λ ($n = 0$), ab ($n = 1$), $aabb$ ($n = 2$), … La sua grammatica sarà libera dal contesto e avrà questa forma:

$S ::= aSb \mid \lambda$

Infatti posso ottenere $S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb \rightarrow \cdots \rightarrow a^n Sb^n \rightarrow a^n \lambda b^n \rightarrow a^n b^n$, ch'è cioè la stringa descritta in termini generali dal linguaggio L. Per quanto riguarda l'automa capace di riconosce questo linguaggio, non si tratta di un automa a stati finiti in quanto è necessario avere una memoria in cui conservare il numero di a lette dall'automa – e l'automa a stati finiti **non ha memoria!** È invece adatto l'**automa a pila**: vi inserisco cioè n-volte le a, quindi tolgo n-volte le b; se la pila è vuota allora la stringa è stata riconosciuta.

Esempio #2. Poniamo d'avere il seguente linguaggio:

$$J = \{a^n b^n c^n \mid n \geq 0\}$$

J, cioè, è formato dalle stringhe λ ($n = 0$), abc ($n = 1$), $aabbcc$ ($n = 2$), … La sua grammatica è più difficile da individuare perché avrò bisogno di una *memoria* in cui porre il valore di *n*, non potendo servirmi quindi né di un automa a stati finiti né di un automa a pila (che riconosce solo la stringa $a^n b^n$, ma non $a^n b^n c^n$). La grammatica dovrà quindi esser *contestuale*.

Esempio #3. Poniamo d'avere il seguente linguaggio:

$$L = \{a^n b^k \mid n, k \geq 1\}$$

Non abbiamo bisogno di memoria: non esiste una legge che leghi *n* e *k*. Definiamo quindi la grammatica di L:

$S ::= aS|aB$ //genero almeno una a a cui segue una a o almeno una a a cui segue una b.

$B ::= b \mid bB$ //genero una b a cui non segue niente, oppure genero una b a cui segue una b.

Dato che n e k sono valori slegati, a^3b^2 diventa $S \to aS \to aaS \to aaaB \to aaabB \to aaabb$

In generale, invece, il risultato è:

$$S \to aS \to aaS \to aaaS \to \cdots \to a^{n-1}S \to a^nB \to \cdots \to a^nb^{k-1}B \to a^nb^{k-1}b \to a^nb^k.$$

Ricapitolando: quando il linguaggio può essere riconosciuto *senza* l'uso di memoria, allora la sua grammatica è formata da produzioni del tipo:

$$A = aA \mid aB \mid a$$

…per, rispettivamente, continuare ad usare questo token, passare al token precedente, terminare.

Dalla grammatica il relativo automa. Una volta ottenuta una grammatica libera dal contesto, è facile trasformarla in un **automa non deterministico** seguendo questi passi**:**

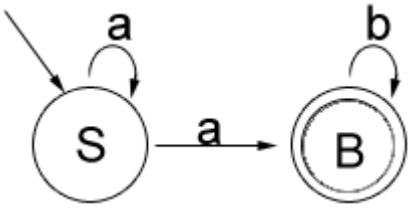
- Ogni ***token*** **non terminale diventa uno stato** dell'automa;
- La definizione del *token* stabilisce le **transizioni dell'automa**.

Per esempio, la grammatica…

$S ::= aS|aB$

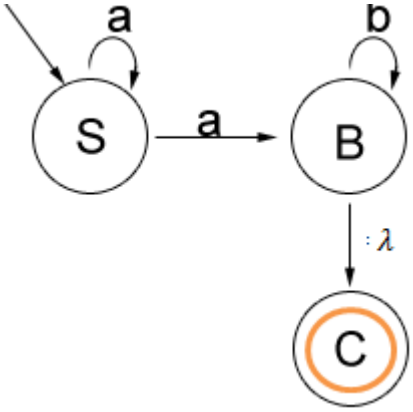
$B ::= b \mid bB$

…avrà gli stati S e B, con S stato iniziale, e le transizioni saranno "da S, leggo a e resto in S", "da S, leggo a e vado in B", "da B, termino", "da B, leggo b e resto in B".

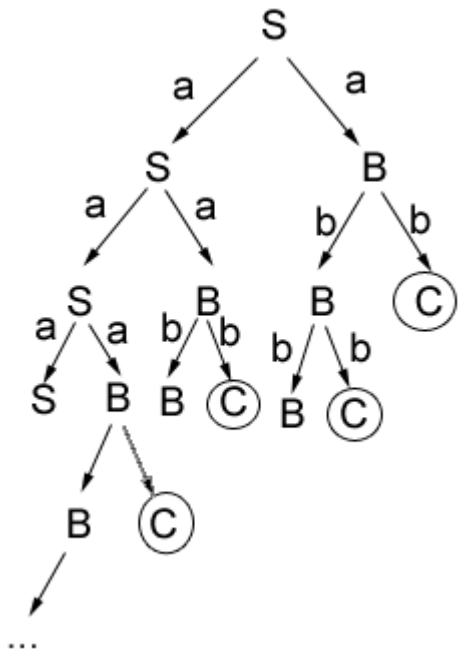


Posso anche aggiungere uno stato ulteriore, per chiarire che accade se l'automa legge la stringa vuota:

$S ::= aS \mid aB$
$B ::= bC \mid bB$
$C ::= \lambda$

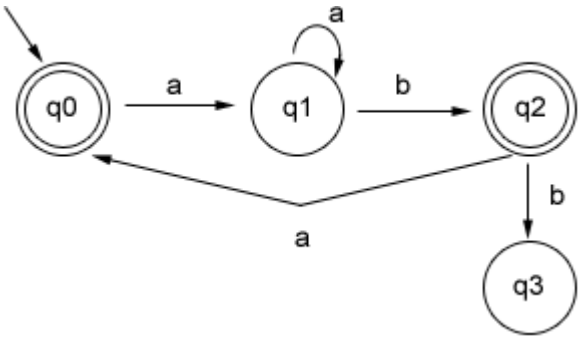


Notare che, non avendo memoria, l'automa può anche sbagliare strada come evidenziato in questo **albero di derivazione**:



Ciò significa che se la macchina non deterministica **può sbagliare strada**. Naturalmente esiste *almeno* una strada corretta: $a \rightarrow\ a \rightarrow\ a \rightarrow b \rightarrow b \rightarrow C$, ch'è la stringa che volevamo ottenere.

Dall'automa la relativa grammatica. Dato un automa, deterministico o non, possiamo anche ricavarne la grammatica. Dato ad esempio questo automa, si procede come segue:



- Stato da cui partono transizioni = *token* non terminale.
- Transizione verso un nuovo stato = simbolo terminale + token non terminale.
- Stato da cui non partono transizioni = non fare nulla.
- Stato iniziale = *token* iniziale.
- Stato finale = aggiungo λ.

Quindi il nostro automa diventa:

$Q_0 ::= aQ_1 | \lambda$ // a partire da Q_0, posso trovare a e passare in Q_1. È stato finale.

$Q_1 ::= aQ_1 \mid bQ_2$ //a partire da Q_1, posso restare in Q_1 o passare in Q_2.

$Q_2 ::= aQ_0 | bQ_3 | \lambda$ // da Q_2, posso restare in Q_0 leggendo a o passare in Q_3 leggendo b.

Notare che Q_3 non ha frecce uscenti ma solo entranti, quindi non è necessario scriverne la grammatica. Non essendo stato finale, in questo caso **la macchina si blocca** e la stringa abb non viene cioè riconosciuta:

$Q_0 \rightarrow a\ Q_1 \rightarrow abQ_2 \rightarrow ab$ Stringa correttamente riconosciuta!

$Q_0 \rightarrow a\ Q_1 \rightarrow abQ_2 \rightarrow abbQ_3$ Stringa non riconosciuta, la macchina si blocca!

Domanda: che linguaggio è riconosciuto da questa grammatica? Sicuramente λ, almeno una a, tante a, almeno una b:

$\lambda, a, aa^n b, aa^n ba, \ldots$ = $aa^*b + (aa^*ba)^*$ con $n \geq 0$

…dove il simbolo "+" indica "oppure".

Osservazione #1. Un ***linguaggio libero*** **è decidibile** perché sempre riconosciuto da una grammatica libera. Notare che un linguaggio libero si distingue da un linguaggio regolare perché il secondo ha produzioni che "avanzano" di un carattere alla volta.

Grammatiche regolari.

La **classe dei linguaggi *regolari*** può essere definita in più modi: grammatiche regolari, espressioni regolari e automi a stati finiti, deterministici e non.

- ***Grammatiche regolari***. Sono grammatiche rappresentabili sempre attraverso la solita notazione: una grammatica regolare G è fatta di un insieme di simboli non terminali (NT), un insieme di simboli terminali (T), un simbolo iniziale S e una o più P: $G = (NT, T, S, P)$

Le sue produzioni hanno sempre questa struttura:

$A ::= B$ corrisponde ad un ***cambio di stato*** senza che avvenga lettura della testina.

$A ::= bC$ corrisponde a simbolo terminale + cambiamento di stato

$A ::= b$ corrisponde semplicemente ad un simbolo

$A ::= \lambda$ corrisponde alla stringa vuota

- ***Automi a stati finiti***. L'automa a stati finiti corrispondente ad una grammatica regolare è formato da un insieme finito di stati Q, un insieme finito di stati finali $F \subseteq Q$, un alfabeto A e uno o più programmi P capaci di "legare" gli stati tra loro. I programmi hanno la struttura a tripla:

$$Stato\ Carattere\ NuovoStato$$

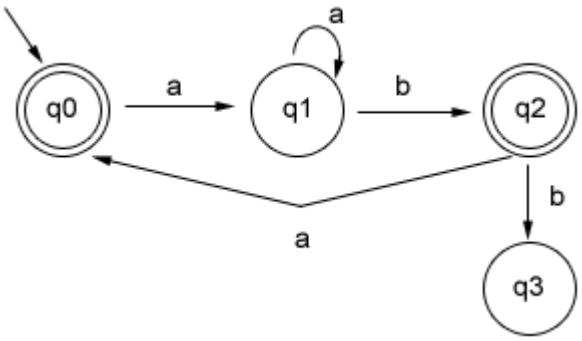
Per esempio, un programma è $q\ a\ q'$ (se nello stato q leggo a passo allo stato q'). P , quindi, è una funzione parziale che, dato stato e simbolo, restituisce gli stati verso cui avviene la transizione. Dato che gli stati risultati applicando P possono essere più di uno, allora P ha per codominio l'insieme delle parti degli stati:

$$P: QxA^* \rightarrow \mathcal{P}(Q)$$

A partire dall'alfabeto $A = \{a_1, \ldots, a_n\}$ formato da n simboli possiamo ricavare le cosiddette **espressioni regolari**, che indichiamo con il simbolo non terminale E: $E ::= \underline{\lambda} \mid a_1 \mid a_2 \mid \ldots \mid a_n$

Il linguaggio generato dalla grammatica è l'insieme L formato dalle stringhe di terminali x appartenenti all'insieme delle stringhe, A^* e tali che esiste almeno un percorso nell'automa che fa andare dallo stato iniziale allo stato finale, leggendo esattamente la stringa x che stiamo cercando.

Esempio #1. Riprendiamo l'automa visto precedentemente:



…allora abbiamo i seguenti insiemi:

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

$F = \{q_0, q_2\}$

$A = \{a, b\}$

I programmi dell'automa precedente sono:

$q_1\ b\ q_2 \quad q_1\ a\ q_1 \quad q_0\ a\ q_1 \quad q_2\ b\ q_3 \quad q_2\ a\ q_0 \quad q_0\ a\ q_0$

In generale, quindi abbiamo che il linguaggio generato dalla grammatica è l'insieme $L = \{x \in A^*$:

$P(q_0, a) = (q_0, q_1)$ //la funzione P, dato uno stato e un simbolo, restituisce 0, 1, 2… stati

$P(q_0, b) = \emptyset$

$P(q_1, a) = \{q_1\}$

$P(q_1, b) = \{q_2\}$

$P(q_2, b) = \{q_3\}$

$P(q_2, a) = \{q_0\}$

$\}$

Per leggere la stringa x dovremo allora passare attraverso una serie di stati e, per ciascuno di essi, si dovrà fare l'unione del simbolo associato a tale stato:

$$\underline{P}^*(q, \mathrm{x}) = \cup \underline{P}(t, a),$$

…dove $P(t, a)$ è un insieme formato da è lo stato ottenuto dopo aver letto la stringa x mentre si era in uno stato q.

- ***Espressioni regolari***. Sia dato l'alfabeto $A = \{a_1, \dots, a_n\}$ formato da n simboli: possiamo ricavare le seguenti *espressioni regolari,* utili a rappresentare i linguaggi in una maniera alternativa:

$$E ::= \underline{\lambda} \mid \underline{a_1} \mid \underline{a_2} \mid \dots \mid \underline{a_n} \mid \underline{\emptyset}$$

Notare che, per evitare confusione, le espressioni regolari sono sottolineate. In generale, se r è un'espressione regolare, $L(r)$ sarà il linguaggio denotato da r:

$L(\underline{\emptyset}) = \emptyset$ - $\underline{\emptyset}$ rappresenta il linguaggio vuoto.

$L(\underline{\lambda}) = \{\lambda\}$ –$\underline{\lambda}$ rappresenta il linguaggio che contiene sono la stringa vuota.

$L(a_i) = \{a_i\} -$ $\underline{a_i}$ rappresenta il linguaggio che contiene sono la stringa a_i.

Siano poi date due espressioni regolari, che indichiamo con E_1 e E_2, denotanti rispettivamente il linguaggio L_1 e L_2. Abbiamo allora a disposizione le seguenti operazioni: $E + E \mid EE \mid E^*$

- $L(E_1 + E_2) = L(E_1) \cup L(E_2) = L_1 \cup L_2$ – cioè il linguaggio ottenuto dalla somma di due espressioni regolari equivale all'unione dei linguaggi che esse rappresentano. Se quindi $L(a^*b) = \{a^n b : n \geq 0\}$, e $L(ccd^*) = \{ccd^n : n \geq 0\}$, allora $L(a^*b + ccd^*) = \{a^n b : n \geq 0\} \cup \{ccd^k : k \geq 0\}$.
- $L(E_1E_2) = L(E_1) \cdot L(E_2) = L_1L_2$ – cioè il linguaggio ottenuto dalla concatenazione di due espressioni regolari equivale alla concatenazione dei linguaggi che rappresentano.
- $L(E^*) = L(E)^*$ – cioè il linguaggio ottenuto da E^* è uguale al linguaggio L^*, cioè al linguaggio denotato da E a cui poi viene applicato l'operatore *. Ricordiamo che * prende tutte le stringhe di un linguaggio e le combina tra loro un numero finito di volte: se ad esempio si ottengono così tutte le possibili stringhe di L ottenute concatenando un numero finito di volte tutte le stringhe di L. Per esempio,

$$L(\underline{a^*b}) = L(\underline{a^*}) \cdot \mathrm{L}(\underline{\mathrm{b}}) = L(\underline{a^*}) \cdot \{b\} = \{a\}^* \cdot \{b\}.$$

Le grammatiche regolari risultano utili perché permettono di studiare i ***token* della programmazione**, che vengono interpretati come *espressioni regolari*: per esempio il token *while* viene interpretato come la concatenazione di w, h, i, l, e:

$$while = \{w\} \cdot \{\mathrm{h}\} \cdot \{i\} \cdot \{l\} \cdot [e] = \{while\}$$

Per esempio, dato $L_1 = \{x : x = canea, \alpha \in A^*\}$ - quindi L_1 conterrà canestro, cane… - e dato $L_2 = \{x : x = gattoa, \alpha \in A^*\}$, allora L_1 e L_2 concatenati producono:

- "canegatto", dove cioè "cane" appartiene a L_1 (concatenazione di "cane" e la stringa vuota), gatto appartiene a L_2 (concatenazione di "gatto" e la stringa vuota).
- "canegattone", dove abbiamo cioè "cane" che appartiene a L_1 (concatenazione di "cane" e la stringa vuota"), "gattone" appartiene a L_2 (concatenazione di "gatto" e "ne"). C'è poi "canegattogatto", "canecanegatto", …

…da cui $L_1L_2 = \{xy : x \in L_1, y \in L_2\}$. Tornando a *while*, possiamo anche ottenere il costrutto *while do then* usando semplicemente l'unione delle varie concatenazioni:

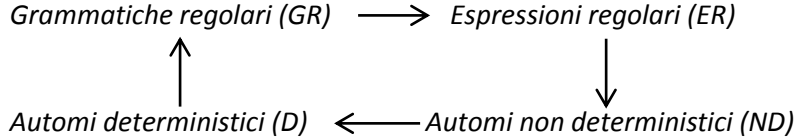
$$\{while\} + \{do\} + \{then\}$$

…e costruire il linguaggio formato dall'insieme finito dei *token*.

Esempio #2. L^* è uguale alla concatenazione (tra loro) di tutte le stringhe di L:

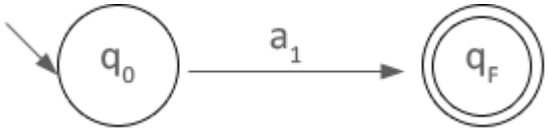
$$L^* = \{\{\lambda\} \cup L_1 \cup L_2 \cup \ldots L_n\}$$

Equivalenza tra *grammatiche, automi ed espressioni regolari*. Vogliamo ora dimostrare che grammatiche regolari, automi a stati finiti e espressioni regolari sono equivalenti tra loro:



Automa D, Automa ND

Singleton. Sia data l'espressione regolare a_1 che definisce il linguaggio regolare $\{a_1\}$: esiste allora un corrispondente *automa a stati finiti* (non deterministico) che riconosce questo linguaggio. Dato che si tratta di un singleton, per riconoscerlo basta considerare uno stato iniziale q_0, lo stato finale q_F e il passaggio tramite a_1:



Linguaggio vuoto. L'automa che riconosce il linguaggio vuoto è l'automa "non esistente", dato che non deve riconoscere nulla.

Linguaggio dotato di stringa vuota. Supponiamo invece d'avere il seguente linguaggio:

$L = \{\lambda\}$

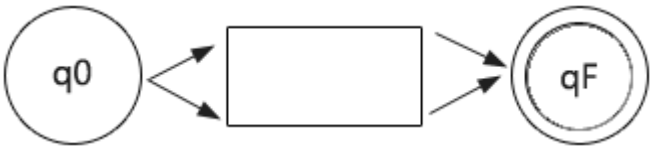
Questo linguaggio si riconosce tramite un automa *il cui stato iniziale è necessariamente anche lo stato finale*:

Linguaggio più complesso. Supponiamo d'avere due espressioni regolari:

E_1 definisce il lingauggio $L(E_1)$;

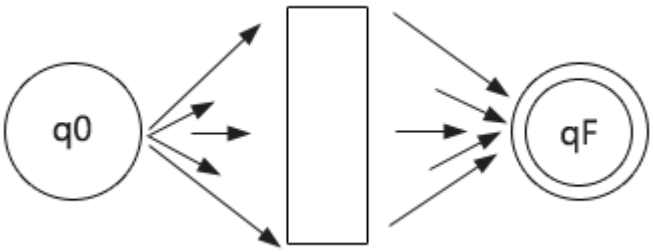
E_2 definisce il linguaggio $L(E_2)$;

Sappiamo poi che $E_1 + E_2$ è un'altra espressione regolare che definisce l'unione dei due linguaggi. Come ottenere il relativo automa? Il meccanismo si basa sull'assunzione che qualsiasi linguaggio regolare ha un automa con questa struttura (e per estensione qualsiasi automa può essere trasformato in un automa di questo tipo):



…ossia ogni linguaggio può sempre essere riconosciuto da un automa in cui sono isolati uno stati iniziale, uno stato finale e una struttura interna. Il motivo è che ogni automa *deterministico* M ha un corrispettivo automa *non deterministico* M' tale che il linguaggio riconosciuto da M è uguale al linguaggio riconosciuto da M': $L(M) = L(M')$

Un automa è semplice se è dotato di un solo stato iniziale $\mathrm{Q_i}$ e di un solo stato finale $\mathrm{Q_f}$, non coincidenti tra loro e tale per cui non esistono transazioni che vanno in $\mathrm{Q_i}$ e che partono da $\mathrm{Q_f}$.



Come progettare un automa di questo tipo? Sia dato M automa a stati finiti non deterministico così definito:

$$M = \{Q,\ A,\ q_0,\ F\ \subseteq Q,\ t\},\ dove\ t\colon QxA\ \rightarrow \mathcal{P}(Q)$$

Per compiere la trasformazione, prendiamo lo stato iniziale q_0 e un carattere a_i dell'alfabeto. Applichiamo t alla coppia q_0, a_i: potrò ora raggiungere al più tutti gli stati (almeno zero) a partire da q_0 e leggendo il carattere a_i.

$$t(q_0, a_i) = \text{stati raggiungibili da } q_0 \text{ leggendo } a_i$$

Abbiamo quindi più casi:

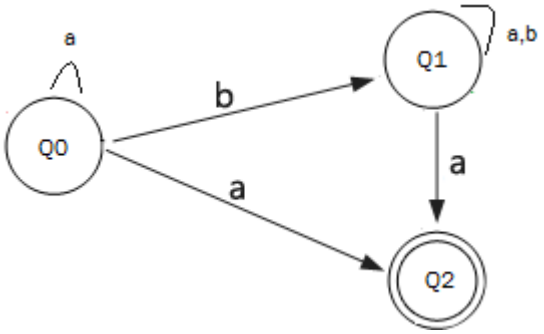
- Se $t(\mathrm{q}_0, a_i)\ \cap F = \emptyset$, cioè se non ho stati finali, l'automa è il seguente:

  $$t'(q_1,\ a_i) = t(q_0,\ a_i)$$

- $t(q_0,\ a_i)\ \cap F \neq \emptyset$, cioè invece se ho stati finali, l'automa è:

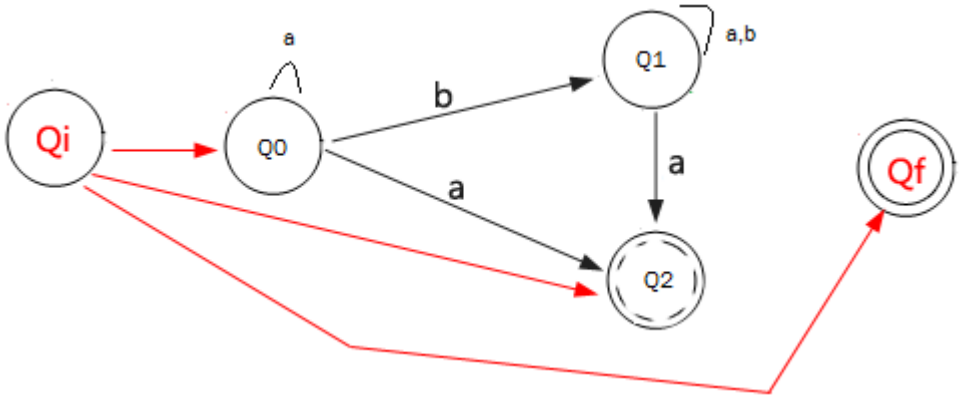
  $$t'(q_I,\ a_i) = t(q_0, a_i) \cup \{q_f\}$$

Per esempio, poniamo d'avere questa situazione:

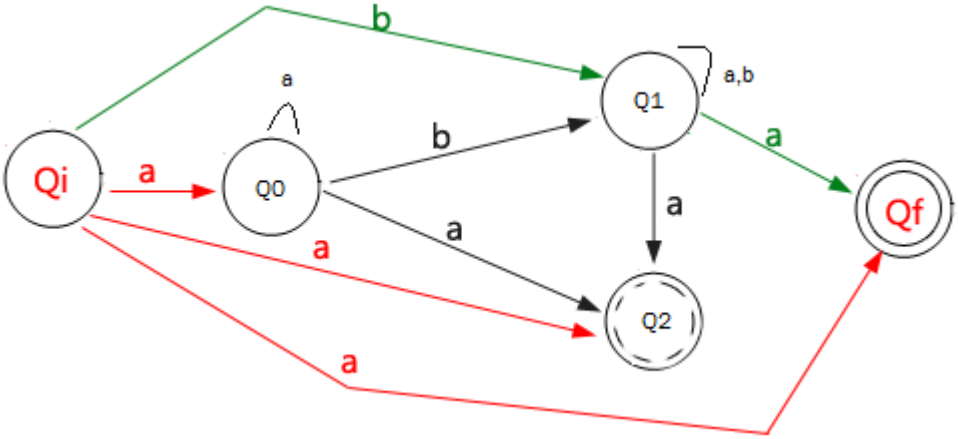


Concentriamoci sul carattere a:

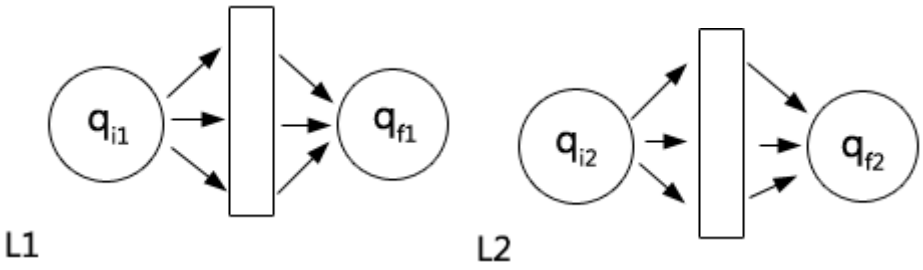
- Aggiungiamo uno stato q_i, che funziona da stato iniziale e uno stato q_f, che funziona da stato finale;
- Modifichiamo: dato che inizialmente la transazione era $t(q_0, a) = \{q_0, q_2\}$, cioè dallo stato iniziale q_0 si passa in q_0 o in q_2, allora dobbiamo avere delle transazioni che, a partire dal nuovo stato iniziale q_i, portino in q_0, q_2 e q_f leggendo a.



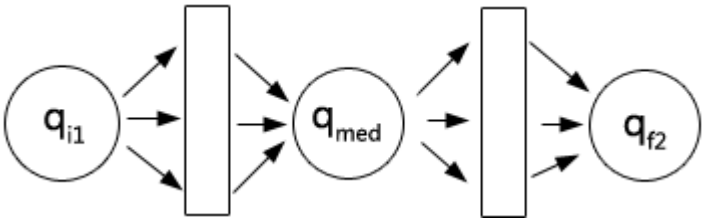
Concentriamoci infine sul carattere b: inizialmente, a partire da q_0, l'automa passava in q_1. Quindi abbiamo bisogno di una nuova transazione tra q_i e q_1. Infine, dato che da q_1 si passava nello stato finale q_2, ora da q_1 bisogna passare in q_f. Il risultato finale è:



In questo modo l'automa di partenza è equivalente all'automa così trasformato: tutto quello che poteva essere effettuato sul precedente automa può essere fatto anche su questo. Perciò, per concatenare due automi, basta riconoscere entrare nell'automa del primo linguaggio, riconoscere una stringa del primo linguaggio e arrivare in Q_f, quindi riconoscere una stringa del secondo linguaggio considerando Q_f lo stato da cui inizia la seconda fase di riconoscimento: se quindi partiamo dai seguenti automi…

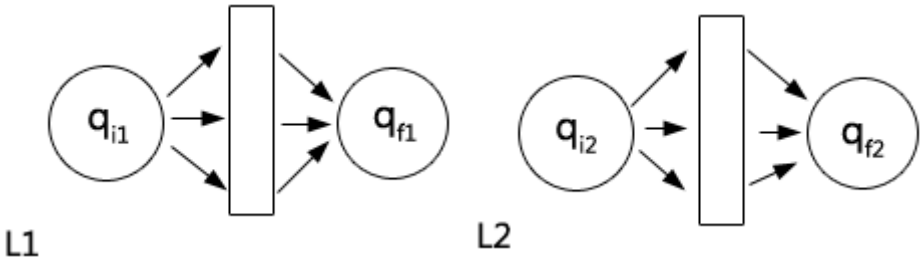


…ottengo:

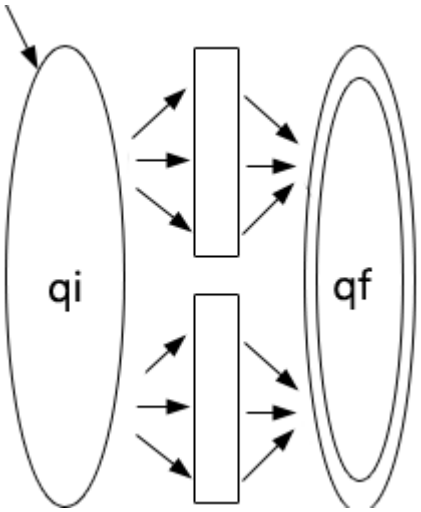


…che riconosce i due linguaggi concatenati. Notare che nel caso si voglia gestire la *stringa vuota*, è necessario operare alcune modifiche: è però dimostrabile che le transazioni di tipo lambda sono sempre eliminabili.

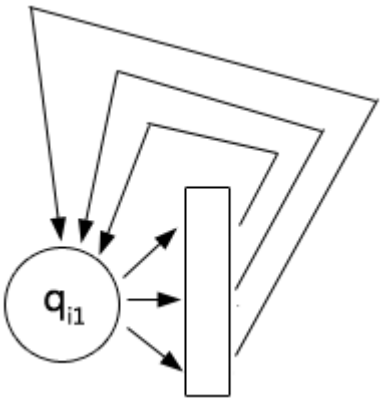
L'unione invece significa “andare in parallelo”: se ho due automi e voglio riconosce il linguaggio “unione”…



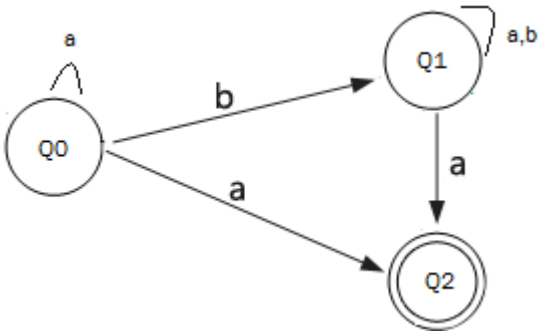
…fondo semplicemente gli stati iniziali e finali in modo che si possa seguire qualsiasi strada possibile:



Per $*$, infine, tolgo lo stato finale e faccio tornare allo stato iniziale:



Trasformare automa *non deterministico* in *deterministico*. Consideriamo il seguente automa:



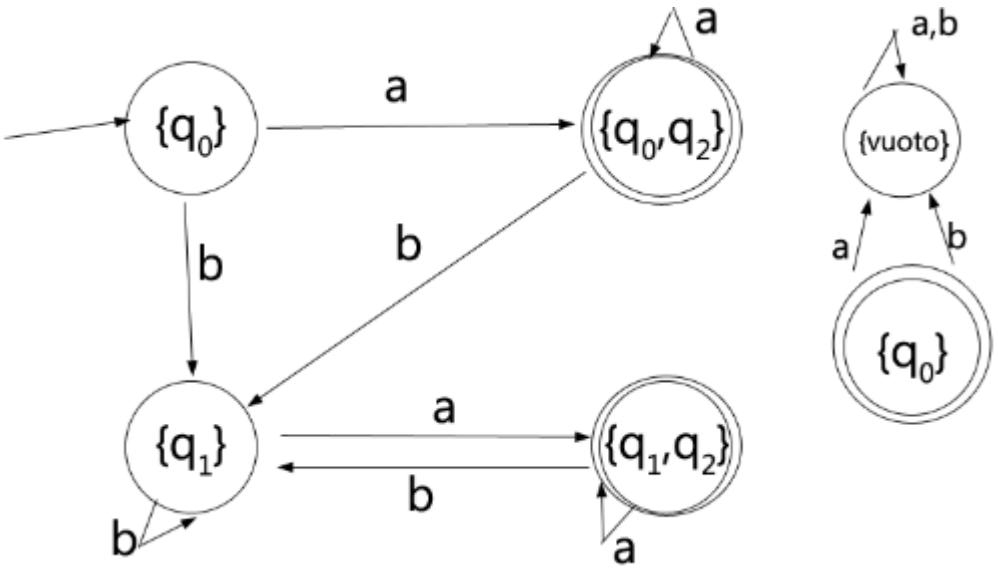
Non è deterministico perché, dato un certo stato, leggendo lo stesso carattere non finisco in uno e uno solo stato: per esempio, leggendo a, posso finire in q_0 o in q_2. La trasformazione in automa *deterministico* avviene prendendo l'unione degli stati: nel caso abbia una situazione "deterministica" lascio l'automa invariato, altrimenti cerco di unire gli stati in modo da riportarmi ad una situazione deterministica. Riprendendo il nostro esempio, avremo che:

- Da q_0 ho una situazione non deterministica perché, leggendo a, posso finire in q_0 e in q_2. Ottengo quindi due stati: $\{q_0\}$ perché da q_0, prendendo a, resto in q_0, e $\{q_0, q_2\}$, perché da q_0, leggendo a, vado in q_2. Da q_0, leggendo b, passo in $\{q_1\}$.
- Da q_1, leggendo a, posso restare in q_1 oppure passare in q_2, quindi avrò gli stati $\{q_1\}$ e $\{q_1, q_2\}$ Se leggo b, invece, resto in $\{q_1\}$.
- Da q_2, sia che legga a o che legga b, non vado da nessuna parte: ho bisogno di uno stato che etichetto con l'insieme vuoto: $\{\emptyset\}$.

Una volta analizzati gli stati dell'automa iniziale, si passa a considerare gli stati dell'automa "in costruzione":

- Da $\{q_0, q_2\}$, leggendo a, o non vado da nessuna parte (caso di q_2) o resto in q_0 (caso di q_0). Unisco i risultati e ottengo una freccia che ritorna in $\{q_0, q_2\}$. Se invece leggo b, vado in q_1 perché da q_0 andavo in q_1.
- Da $\{q_1, q_2\}$, leggendo a, o resto in q_1 o vado in q_2, quindi in sostanza resto in questo stato. Se invece leggo b, vado in $\{q_1\}$.

Notare che, da $\{\emptyset\}$, sia che io legga a o che legga b, comunque resto in $\{\emptyset\}$. Gli stati finali, nell'automa iniziale, erano solamente q_2: tutti i nuovi stati che presentano q_2 diventano automaticamente stati finali. Il risultato finale è:

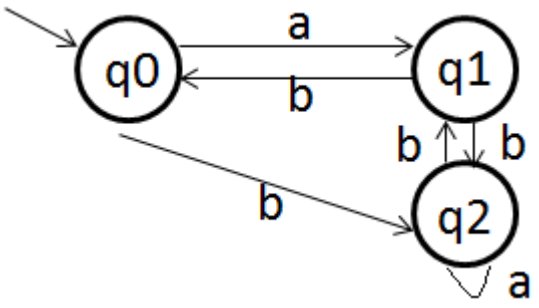


Tale automa riconosce tanto quanto l'automa precedente. Ora, proviamo a ricavarne il linguaggio: diamo ad ogni nodo un'etichetta e scriviamo il carattere "di passaggio" e lo stato d'arrivo:

$$A_1 ::= aA_3 | bA_2 | a$$
$$A_2 ::= aA_4 | bA_2 | a$$
$$A_3 ::= aA_3 | bA_3 | a | \lambda$$
$$A_4 ::= aA_4 | bA_2 | a | \lambda$$
$$A_5 ::= aA_6 | bA_6 | \lambda$$
$$A_6 ::= aA_6 | bA_6$$

…notare che se lo stato è finale inseriamo λ.

Relazione d'equivalenza e linguaggi formali. Una relazione d'equivalenza è una relazione che **divide un insieme in classi disgiunte** tra loro. Poniamo d'avere un automa non deterministico formato da un tot di stati e in grado di riconoscere stringhe di alfabeto A. È allora possibile **dividere l'insieme delle stringhe** ottenibili a partire da A **in tante classi quanti sono gli stati dell'automa**; la stringa x e la stringa y apparterranno alla stessa classe se e solo se, partendo dallo stato iniziale, leggendo entrambe so arriverà allo stesso stato finale. Per esempio, dato questo automa…



…abbiamo tre classi ottenibili partendo da q_0 e terminando in q_0, in q_1 e in q_2. Saranno così composti:

$$q_0 = \{ab, abab, bbb, abababababab, abaaaaaaaaabb \ \dots\}$$
$$q_1 = \{a, aba, abb, bb, \dots\}$$
$$q_2 = \{b, baaaaaa, ab, \dots\}$$

Il linguaggio riconosciuto da quest'automa si ottiene **unendo le classi d'equivalenza**. Una relazione d'equivalenza può inoltre essere di tipo **invariante destra**: date cioè due stringhe x e y appartenenti alla stessa classe d'equivalenza, allora vale l'invariante destra se, concatenando z ad entrambe, xz e yz sono ancora appartenenti alla stessa classe d'equivalenza: $x\ E\ y \quad \rightarrow \quad xz\ E\ yz$

Per esempio, $\boldsymbol{ab}\ aaabb\ E\ \boldsymbol{abab}\ aaabb$ rispettano la proprietà d'invarianza a destra. Notare che l'invarianza a destra gode della proprietà riflessiva ($x\ E\ x$) e transitiva (se $x\ E\ y$ e $y\ E\ z$ allora $x\ E\ y$).

Ora, poniamo d'avere un **linguaggio formato da un numero finito di classi d'equivalenza** ottenute attraverso la proprietà d'invariante destra: ne deriva che tale linguaggio **è riconoscibile da un automa a stati finiti deterministico**. Infatti:

- Se abbiamo un automa a stati finiti deterministico, allora la relazione d'equivalenza invariante destra è definita proprio come prima: posso cioè individuare le classi d'equivalenza osservando quali stringhe del linguaggio terminino nello stesso stato, e unire le classi ottenute ottenendo il linguaggio riconosciuto dall'automa.
- Se invece abbiamo un linguaggio suddiviso in un numero finito di classi d'equivalenza che rispettano la proprietà invariante destra, allora posso ricostruire l'automa corrispondente. Per capire come, poniamo d'avere un linguaggio formato da stringhe alternate di a e b:

$$L = \{a, b, ab, ba, aba, bab, \dots\}$$

Partizioniamo questo insieme di stringhe in sottoinsiemi disgiunti tra loro:

$L_0 = \{\lambda\}$

$L_1 = \{a, a\boldsymbol{ba}, ababa, \dots\} = \{a(ba)^n : n \geq 0\}$

$L_2 = \{b, b\boldsymbol{ab}, babab, \dots\} = \{b(ab)^n : n \geq 0\}$

$L_3 = \{ba, ba\boldsymbol{ba}, bababa, \dots\} = \{(ba)^n : n \geq 1\}$

$L_4 = \{ab, ab\boldsymbol{ab}, ababa, \dots\} = \{(ab)^n : n \geq 1\}$

$L_5 = \{bb, aa, bb\boldsymbol{a}, aa\boldsymbol{b}, \dots\} = \{ybbx : x, y \in A^*\} \cup \{yaax : x, y \in A^*\}$

Ora, queste classi rispettano la relazione d'invarianza a destra perché, se ad esempio prendiamo due stringhe appartenenti a $L_1 - aba$ e $a -$ e definiamo $z = a(ba)^r$, otteniamo che:

$$xz\ E\ yz \ \rightarrow\ aa(ba)^r\ E\ abaa(ba)^r$$

…da cui l'equivalenza. Possiamo ora ottenere il relativo automa, secondo queste regole:

- Per **ogni classe del linguaggio, uno stato**: $L_0, L_1, L_2, L_3, L_4, L_5, L_6$.
- Da L_0, leggendo a passo in L_1; Se leggo invece b, passo in L_2.
- Da L_1, leggendo a passo in L_5; se leggo invece b passo in L_4.
- Da L_2, leggendo a passo in L_3; se leggo invece b passo in L_5.
- Da L_3, leggendo a passo in L_5; se leggo invece b passo in L_2.
- Da L_4, leggendo a passo in L_1; se leggo invece b passo in L_5.
- Da L_5, leggendo a o b resto in L_5.

Dato poi che il linguaggio deve riconoscere stringhe che alternano a e b, gli stati finali saranno L_2, L_3 e L_4, che terminano in b. Lo stato iniziale, invece, è L_0, dov'è presente la stringa vuota.

Notare che quanto appena detto differenza i linguaggi regolari da ogni altro linguaggio: solo i linguaggi regolari presentano un numero *finito* di classi d'equivalenza ottenute grazie alla proprietà invariante destra, mentre ogni altro linguaggio definire un numero *non finito* di classi.

Automa minimo. Dato un linguaggio è anche possibile **costruire il minimo automa** in grado di conoscerlo (cioè formato dal numero *minimo* di stati). Supponiamo d'avere il linguaggio L in cui è definita la seguente relazione d'equivalenza invariante destra:

$$x \,\mathrm{E}'\, y \quad sse \quad \forall z \begin{cases} xz \in L & \rightarrow \; yz \in L \\ yz \in L & \rightarrow \; xz \in L \end{cases} \quad \text{oppure}$$

Se questa relazione definisce un numero *finito* di classi d'equivalenza in un linguaggio, allora possiamo costruire un automa *minimo* che riconosce tale linguaggio. Gli stati di quest'automa sono le classi d'equivalenza di questa relazione. Inoltre vale il teorema secondo cui **l'automa *minimo* che riconosce un linguaggio è unico**, a meno di cambiamento di nomi degli stati.

Torniamo all'esempio precedente in cui analizzavamo un linguaggio di stringhe che alternano a e b. L'automa è quello *minimo*? Possiamo cioè ottenerne uno con meno di sei stati? Q_0 non può essere finale (pena il riconoscere la stringa vuota, che però non appartiene al linguaggio): devo per forza leggere a o b; dato che devo alternare a e b ho bisogno di sapere se ho letto a o b, in modo da alternare: da cui gli stati Q_a e Q_b. Se, da Q_b, leggo una a posso andare in Q_a, mentre se da Q_a leggo una b posso passare in Q_b. Se infine leggo una a quando sono in Q_a o una b mentre sono in Q_b, finisco in uno stato di non riconoscimento, "Q_{no}". Resto in Q_{no} che legga sia a o b, mentre posso terminare ogni volta che leggo una a o una b. Il risultato finale è una automa con 4 stati.

Esercizio #1. Dimostrare che il linguaggio $L = \{a^n b^n : n > 0\}$ non è regolare.

Soluzione. Per dimostrare che L non è regolare basta verificare che la relazione d'equivalenza invariante destra definita in L produce un numero *infinito* di classi d'equivalenza (come sappiamo solo i linguaggi regolari hanno un numero *finito* di classi!). Poniamo quindi per assurdo che questa relazione d'equivalenza sia di indice finito, cioè che tutte le stringhe appartenenti al linguaggio L si possano suddividere in un numero finito di classi. Esiste quindi un $k \neq n$ tale che

$$a^k b^k \,\mathrm{E}\, a^n b^n$$

…cioè $a^k b^k$ e $a^n b^n$ appartengono alla stessa classe. Stringhe come a^n non appartengono invece al linguaggio L, cioè apparterranno ad una **classe di "non riconoscimento"**. Ora, è di sicuro vero che le stringhe a^n e a^k, con $k \neq n$ non appartengono al linguaggio, ossia appartengono alla medesima classe di non riconoscimento:

$$a^k \mathrm{E}\, a^n$$

Ma allora proprio perché vale la relazione d'invarianza a destra, possiamo concatenare ad entrambe la stringa $z = b^k$:

$$a^k b^k \text{ E } a^n b^k$$

In questo modo otteniamo un assurdo, perché da un lato la stringa $a^k b^k$ appartiene ad una classe d'equivalenza del linguaggio, mentre $a^n b^k$ no e non è nemmeno riconosciuta dal linguaggio! Quindi il linguaggio di partenza non è regolare.

Nota: una utile strategia per ragionare su un numero finito di classi d'equivalenza è quella di usare i "contenitori": possiamo cioè immaginare le **stringhe del linguaggio L suddivise in n (numero finito) contenitori**. Una stringa dovrà stare necessariamente in una e una sola degli n contenitori: nel caso però io abbia più stringhe che contenitori (= *infinite stringhe*) però, è certo che dovrò avere in un contenitore *almeno* due stringhe – come a dire che la stringa ab si troverà nel contenitore 1, a^2b^2 sarà nel contenitore 2, … fino a $a^n b^n$, che starà nell'ennesimo contenitore; la stringa $a^{n+1}b^{n+1}$, però, dovrà stare in uno dei contenitori già utilizzati, che conterrà così (almeno) due stringhe. Nel nostro esempio abbiamo *infinite* lettere diverse, dato che non viene fornito un limite superiore all'esponente *n*: perciò avrò sicuramente almeno due stringhe diverse che finiscono nella stessa classe d'equivalenza.

Esercizio #2. Sia dato un linguaggio L regolare. Come ottenere $A^* \backslash L$, cioè l'insieme delle stringhe non appartenenti a L? Basta considerare l'automa che riconosce L e invertire gli stati, ossia trasformare gli stati finali in non finali, gli stati non finali in finali. La classe dei linguaggi regolari è quindi ***chiusa per complementazione***.

Proprietà #1. ***Ogni linguaggio finito*** (=numero finito di stringhe) ***è regolare***. Il motivo è che classe dei linguaggi regolari contiene tutti i linguaggi *finali*. Consideriamo infatti un linguaggio formato solo da una stringa, $L = \{casa\}$. Come sappiamo il suo automa è formato da sei stati: stato *iniziale*, stato c, stato ca, stato cas, stato $casa$ (finale), stato $pozzo$ (dove si va in qualsiasi altro caso, cioè quando la stringa non viene riconosciuta). Altrimenti possiamo prendere l'espressione regolare $\{c\}\{a\}\{s\}\{a\}$, cioè ottenuta concatenando i quattro simboli.

Proprietà #2. Ogni linguaggio regolare è chiuso per unione, concatenazione, * e complementazione e, di conseguenza, anche per l'intersezione. Infatti:

$$L1 \cap L2 = \neg\neg L_1 \cup \neg\neg L_2$$

Proprietà #3. ***Un linguaggio regolare è decidibile***. Motivo: presa una stringa x posso sempre decidere se appartiene o meno al linguaggio L, perché posso sempre farla "riconoscere" o "non riconoscere" dall'automa di L.

Proprietà #4. Un linguaggio regolare riconosciuto da un automa con n stati è *infinito* sse esiste una stringa $x \in L$ **lunga più di *n***. La dimostrazione è:

→ è banalmente vero.

← dato che riconosco il linguaggio con *n* stati, allora la stringa x, s'è appartenente a L e quindi riconosciuta dall'automa, dovrà passare per almeno due volte per lo stesso stato, cioè l'automa presenterà un ciclo. Dato ch'è possibile percorrere lo stesso ciclo infinite volte, allora il linguaggio è infinito.