LEX : Generatore di Scanner

- Lex è un software, che gira sotto Unix, originariamente scritto da Mike Lesk & Eric Schmidt nel 1975
- FLEX (versione più recente) è un free software scritto da Vern Paxson nel 1987 (va con Bison)

flex.sourceforge.net

Lex/Flex è un generatore di analizzatori lessicali

- input: un file di tipo .l che contiene un insieme di definizioni regolari ed una serie di azioni corrispondenti
- output: un programma C che realizza l'automa riconoscitore e che associa ad ogni istanza di una definizione la relativa azione

Sorgente Lex per il ling. $\mathcal{L}$ lex.l → LEX compiler → Analizzatore Lessicale per $\mathcal{L}$ (programma C di nome lex.yy.c

>lex lex.l

lex.yy.c → [Compilatore C] → a.out (eseguibile)

> cc lex.yy.c -ll

-ll: per linkare le librerie di subroutine di Lex

programma sorgente scritto in $\mathcal{L}$ (prog) → [Analizzatore Lessicale per $\mathcal{L}$] → lista di token

> a.out prog

Come è fatto un file di input lex.l ?

Diviso in 3 parti

(1) Dichiarazioni (definizioni regolari)

Ad Es:

```
cifra   [0-9]
cifre   [0-9]+
ide     [a-zA-Z]([a-zA-Z]|[0-9])*
%%
```

(2) Regole $\underbrace{\{\text{espr. regolare}\}}_{\text{pattern}}$ $\underbrace{\{\text{azione}\}}_{\text{frammento di codice C}}$

Ad es:

```
{cifre}  {printf("<NUM,%s>", yytext);}
{ide}    {printf("<IDE,%s>", yytext);}
%%
```

yytext: variabile che contiene il testo matched!

(3) Funzioni Ausiliarie

Se si usano funzioni complesse nella parte "azione", queste potrebbero essere definite qui

Come funziona il programma lex.yy.c ?

Il programma C ottenuto, e compilato per renderlo eseguibile in a.out, implementa essenzialmente il DFA riconoscitore dell'insieme delle espressioni regolari contenute nelle "Regole".

- scandisce il testo sorgente alla ricerca di una stringa che corrisponda al (cioè sia un lessema per) una delle espr. regolari (pattern di categoria sintattica)
- quando riconosce un lessema, esegue l'azione specificata, e passa in output il risultato dell'azione al posto del lessema
- quando l'input non corrisponde a nessun pattern, lo lascia inalterato e "segnala" la cosa al "gestore degli errori".

Esempio banale

(No dichiarazioni)

%% Regole

	pattern	azione corrispondente
P_1	{a}	{ printf("Z"); }
P_2	{abb}	{ printf("K"); }
P_3	{a*b+}	{ printf("Y"); }

- il pattern P_1 è un prefisso di P_2 $\Rightarrow$ cerca il pattern più lungo
- il pattern P_2 è un caso speciale di P_3 $\Rightarrow$ riconosci il primo in elenco

Se l'input è b a b b a , lex.yy.c riconosce i seguenti lessemi

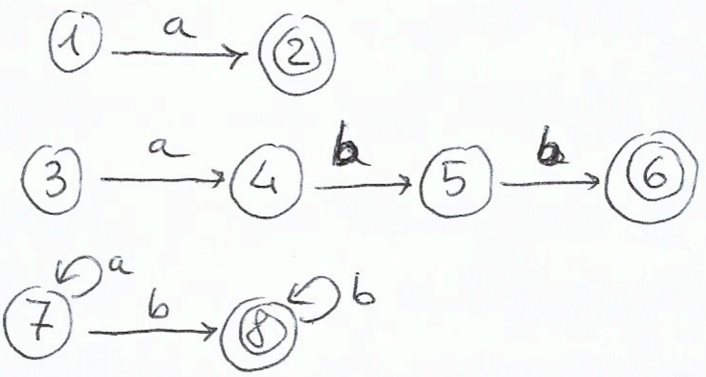
$$\underbrace{b}_{P_3}\ \underbrace{abb}_{P_2}\ \underbrace{a}_{P_1}$$

che "traduce" in Y K Z

Oss: non spezzo abb in $\underbrace{a}_{P_1}\ \underbrace{bb}_{P_3}$

Ma come fa a riconoscere questi lessemi (pattern)?

NFA per ciascuno dei 3 pattern



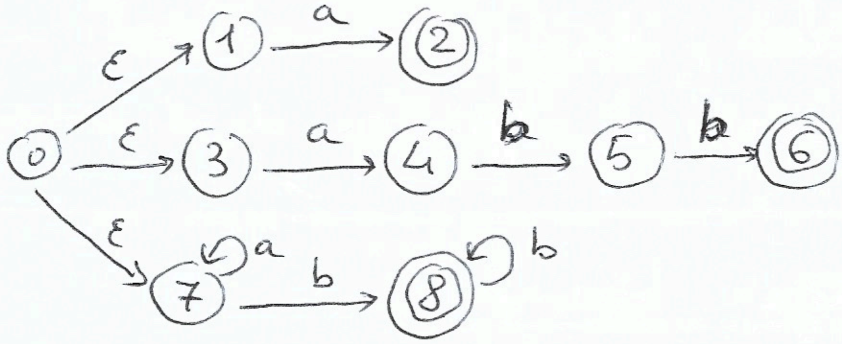
P_1

P_2

P_3

automa NFA per P_3 non canonico

NFA per i 3 pattern insieme



E ora fa una "simulazione" DFA dell'NFA risultante

- stato iniziale = ε-closure(0) = {0,1,3,7}

Stato	Simbolo input a	b	Pattern riconosciuto	Azione corrispondente
0137	247	8	nessuno	—
<u>2</u>47	7	58	$P_1 = a$	printf("Z")
<u>8</u>	–	8	$P_3 = a^* b^+$	printf("Y")
7	7	8	nessuno	—
5<u>8</u>	–	68	$P_3 = a^* b^+$	printf("Y")
<u>6</u> <u>8</u>	–	8	$P_2 = abb$	printf("K")

a.out procede in questo modo:

- va avanti a leggere l'input finché non si blocca
- se lo stato in cui si trova è di riconoscimento ⟹ ok. Altrimenti, torna indietro fino a trovare uno stato di riconoscimento
  (questo garantisce che se più pattern possono essere riconosciuti sullo stesso prefisso, il più lungo viene scelto)
- se arrivati al "match", vediamo che potenzialmente più di un pattern andrebbe bene, allora prendiamo quello elencato prima
  (nell'esempio lo stato 68 è ok sia per P_2 che per P_3, ma prendiamo P_2 perché primo in elenco)

Vediamo il funzionamento sull'input

b a b b a

Inizio in $0137 \xrightarrow{b} 8 \overset{a}{\not\to}$

Mi sono bloccato in 8 ⟹ b è un lessema del pattern $P_3 = a^* b^+$

Ora l'input residuo è abba

Ricominciamo da 0137

$$0137 \xrightarrow{a} 247 \xrightarrow{b} 58 \xrightarrow{b} 68 \overset{a}{\not\rightarrow}$$

Mi sono bloccato in 68 ⇒ abb è un lessema del pattern $p_2 = abb$

(• Nota che sono passato attraverso stati di riconoscimento, ma ho proseguito fino al blocco ⇒ ho preso il lessema più lungo!

• Nota che 68 è OK sia per p_2 che per p_3, ma ho scelto il primo in elenco!)

Ricominciamo da 0137 col residuo a

$$0137 \xrightarrow{a} 247 \quad \text{FINE}$$ ⇒ a è un lessema del pattern $p_1 = a$

Nei file .l di un ling. di progr., le parole riservate vanno messe prima degli identificatori

%

WHILE	Az_1	- se trovo "while" allora riconosco la parola riservata perché è listata prima
IF	Az_2	
⋮		
IDE	Az_{ide}	- se trovo "whiles", ⇒ IDE perché prendo il match più lungo!

Un esempio più interessante

```
cifra           [0-9]
cifre           [0-9]+
ide             [a-zA-Z][a-zA-Z0-9]*
separatore      [ \t\n]*
razionale       {cifre}"."{cifre}
bin             [+*]
```

blank, tab, newline

variabile con testo matched

```
%%

{cifre}                               {printf("<NUM,%s>", yytext);}
{razionale}                           {printf("<FRACT,%s>", yytext);}
":="|";"|while|"("|")"|"{"|"}"        {printf("<%s>",yytext);}
{ide}                                 {printf("<IDE,%s>", yytext);}
{separatore}                          {printf("␣");}
{bin}|"-"                             {printf("<OP2,%s>", yytext);}
```

Figura 3.10 Un semplice analizzatore lessicale in Lex.

Applicando lo scanner prodotto da Lex su un progr.

```
pigreco := 3.14;
temp1 := pigreco * (temp1 + 2);
```

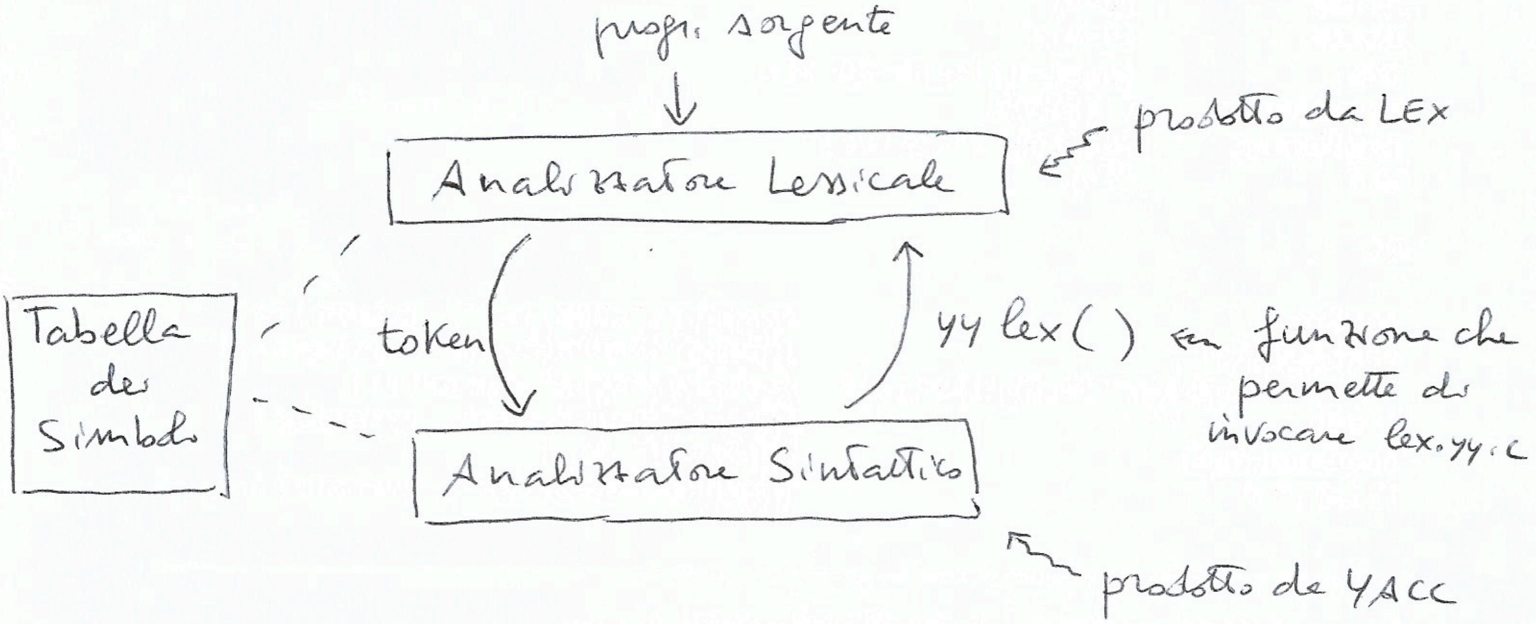
si genera la sequenza di token

<IDE, pigreco> <:=> <FRACT, 3.14> <;>

<IDE, temp1> <:=> <IDE, pigreco> <OP2,*>

<(> <IDE, Temp1> <OP2, +> <NUM, 2> <)> <;>

Utilizzo con YACC



- il programma generato da LEX non è usato da solo
- è una subroutine per YACC (generatore di analizzatori sintattici)
- alcune variabili comuni ai due programmi (come ad esempio yylval) permettono di scambiare informazioni
- lex.yy.c è usato da YACC "on demand", richiedendo il token successivo (alla bisogna)
- yylex() restituisce il nome del token, mentre il valore del token è condiviso nella variabile yylval

Esempio d'uso con YACC

dichiarazioni ausiliarie

```
%{
#define NUM 1
#define FRACT 2
#define IDE 3
#define OP2 4

#define PIU 41
#define PER 42
#define MEN 43
%}
```

dichiarazioni vere

```
cifra           [0-9]
cifre           [0-9]+
ide             [a-zA-Z][a-zA-Z0-9]*
separatore      [ \t\n]*
razionale       {cifre}"."{cifre}
bin             [+*]
```

REGOLE

variabile condivisa tra Lex e YACC
lex mette in yylval il puntatore nella tabella al lessema appena individuato

```
%%
{separatore}        { }
{cifre}             {yylval = inConstTable(); return(NUM);}
{razionale}         {yylval = inConstTable(); return(FRACT);}
{ide}               {yylval = inSymbolTable(); return(IDE);}
+                   {yylval = PIU; return(OP2);}
*                   {yylval = PER; return(OP2);}
-                   {yylval = MEN; return(OP2);}
```

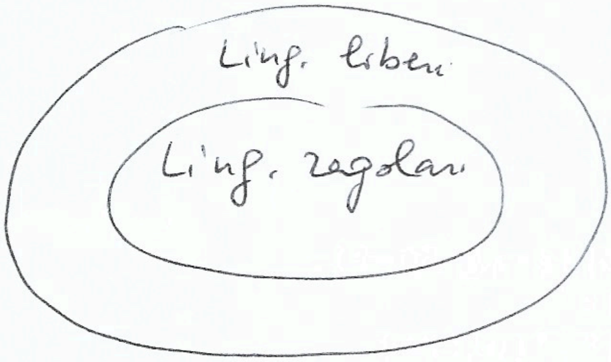
DEF. AUSILIARIE

```
%%
int inConstTable(){/*Converte la stringa puntata da yytext in
                     uno (NUM) o due (FRACT) interi, li inserisce
                     nella tabella delle costanti e restituisce
                     (come int) un puntatore all'elemento
                     inserito*/
}
int inSymbTable(){/*Inserisce nella tabella dei simboli la
                    stringa puntata da yytext e restituisce (come
                    int) un puntatore all'elemento inserito*/
}
```

Figura 3.12 Un programma Lex che interagisce con Yacc.

PROPRIETÀ ALGORITMICHE DEI LING. REGOLARI

Finora non abbiamo dimostrato che esistono ling. non regolari.
Sappiamo che una gr. regolare è un caso speciale di gr. libera. Quindi sicuramente



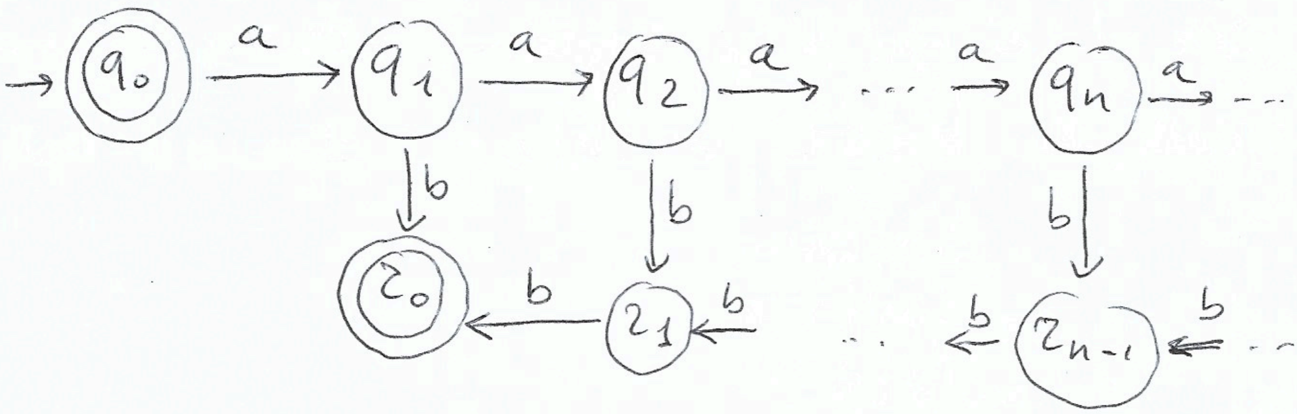
Ora vedremo una proprietà algoritmica dei ling. regolari che sarà utile per dimostrare che certi linguaggi, non possedendo quella proprietà, non sono regolari!

<u>Ad es</u>: $L = \{a^n b^n \mid n \geq 0\}$ è <u>sicuramente libero</u> perché generato da una gr. libera $S \to \varepsilon \mid aSb$ e dimostreremo che <u>non è regolare</u>!

<u>Intuitivamente</u>, perché $\{a^n b^n \mid n \geq 0\}$ non può essere regolare?

Necessità di contare illimitatamente e per farlo mi servirebbero <u>infiniti</u> stati.

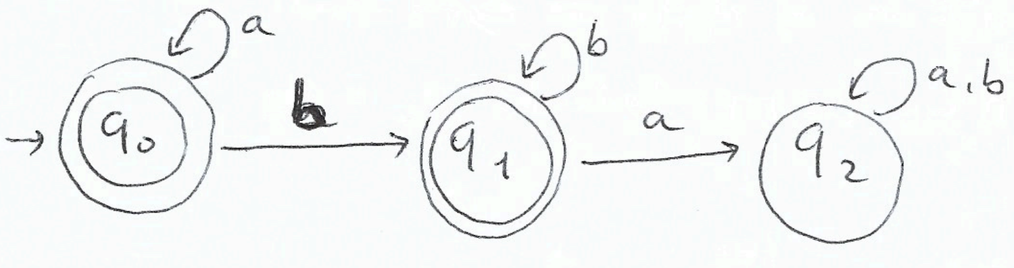
$\{a^n b^n \mid n \geq 0\}$



Se devo farlo per ogni $n \geq 0$, mi servono infiniti stati! Un NFA/DFA ha solo un numero finito di stati, per cui può solo "contare" tanto quanto gli stati che ha.

<u>Attenzione</u>

$\{a^n b^m \mid n, m \geq 0\}$ è regolare e corrisponde all'espr. regolare $a^* b^*$



Vediamo ora la proprietà algoritmica che hanno tutti i ling. regolari.

Pumping Lemma

Se L è un ling. regolare, allora $\exists N > 0$ tale che $\forall z \in L$ con $|z| \geq N$, $\exists u, v, w$ tali che

- $z = uvw$
- $|uv| \leq N$
- $|v| \geq 1$
- $\forall k \geq 0 \quad uv^k w \in L$

Inoltre N è minore o uguale del numero di stati del DFA minimo che accetta L.

Dimostrazione

Sia $N = |Q_M|$, dove M è il DFA minimo che accetta L.

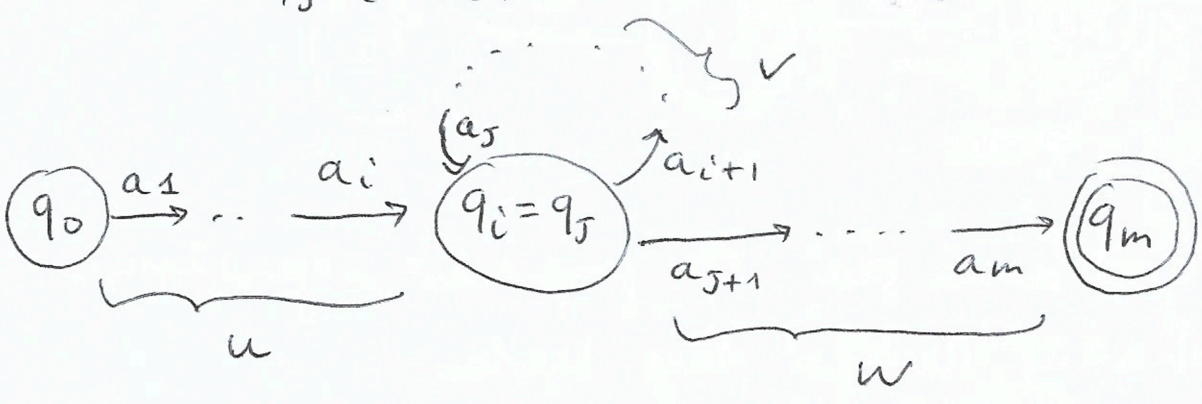
Sia $z = a_1 a_2 \dots a_m \in L$ con $m \geq N$

Quindi

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \dots \xrightarrow{a_m} q_m \in F$$

Ora $0 - m$ è dato da $m+1$ stati con $m+1 > N$

$\Rightarrow \exists i, J \ (i \neq J)$ tali che $q_i = q_J$



- Poiché $i \neq J$, $\Rightarrow v = a_{i+1} \dots a_J$ è tale che $|v| \geq 1$
- La condizione $|uv| \leq N$ mi dice che (se m è molto grande potrebbero esserci più cicli) prendi il primo ciclo!

$$
\left.\begin{array}{l}
uv^0w = uw \in L \\
uv^1w = uvw = z \in L \\
uv^2w = uvvw \in L \\
\vdots \\
uv^kw \in L
\end{array}\right\} \quad \forall k \geq 0 \;\; uv^kw \in L
$$

perché il ciclo v può essere percorso un numero arbitrario di volte (anche zero)

$\boxed{///}$

Oss: Se L è finito, allora non esiste nessuna $z \in L$ con $|z| \geq N$ e quindi "l'implicazione" è vera perché è falsa la premessa

Oss: Se $\exists z \in L$ con $|z| \geq N$, allora M riconosce un ling. infinito

---

Come usare il pumping lemma per dimostrare che un linguaggio non è regolare?

Se L è regolare $\Rightarrow$ P (dice il pumping lemma)

Se $\sim P \Rightarrow$ L non è regolare

$\uparrow$

questa è l'implicazione che useremo!

Negazione del Pumping Lemma

Se [$\forall N > 0 \quad \exists z \in L$ con $|z| \geq N$

$\forall uvw.$ se $\quad - \; z = uvw$

$\qquad - \; |uv| \leq N$

$\qquad - \; |v| \geq 1$

allora $\exists k \geq 0. \; uv^k w \notin L$]

allora $\quad L$ non è regolare

Dimostriamo che $L = \{ a^n b^n \mid n \geq 1 \}$ non è regolare

- Fissiamo un N generico $\qquad (\forall N > 0)$
- Scegliamo $z = a^N b^N \qquad (\exists z \in L$ con $|z| \geq N)$
- Guardiamo tutte le possibili scomposizioni di z in tre sottostringhe ($\forall uvw$) tali che
  - $z = uvw$
  - $|uv| \leq N$
  - $|v| \geq 1$

  La condizione $|uv| \leq N$ impone che u e v siano fatte di sole "a". Quindi $v = a^J$ con $J \geq 1$.
- $\exists k = 2$ tale che $uv^2w = uvvw = a^{N+J} b^N \notin L$

$\Rightarrow L$ non è regolare

$L_1 = \{ a^n \mid n$ è un quadrato perfetto$\}$

$= \{ a^{n^2} \mid n \geq 0 \}$ è regolare? <u>No</u>

- Fissiamo $N > 0$ generico $(\forall N > 0)$
- Scegliamo $z = a^{N^2}$ $(\exists z \in L$ con $|z| \geq N)$
- Per ogni possibile scomposizione di $z = uvw$ tale che
$(\forall uvw)$ (i) $|uv| \leq N$
(ii) $|v| \geq 1$

deve essere che $1 \leq |v| \leq N$

- Prendiamo $K = 2$ cioè consideriamo uv^2w
Vale che

$$|z| = N^2 < |uv^2w| = |z| + |v| \leq N^2 + N < N^2 + 2N + 1 = (N+1)^2$$

cioè uv^2w ha una lunghezza strettamente compresa tra N^2 e $(N+1)^2$

$\Rightarrow uv^2w \notin L_1$

$\Rightarrow L_1$ non è regolare

$L_2 = \{ a^n \mid n$ è una potenza di 2 $\}$

$= \{ a^{2^n} \mid n \geq 0 \}$ è regolare? NO

- Fissiamo $N > 0$ $\quad (\forall N > 0)$
- Scegliamo $z = a^{2^N}$ $\quad (\exists z \in L$ con $|z| \geq N)$
- Per ogni u, v, w tali che $z = uvw$ e

$$(i)\ |uv| \leq N$$
$$(ii)\ |v| \geq 1$$

deve essere $1 \leq |v| \leq N$

- Prendiamo $k = 2$, cioè consideriamo uv^2w

$$|z| = 2^N < |uv^2w| = |z| + |v| \leq 2^N + N < 2^N + 2^N = 2^{N+1}$$

cioè uv^2w ha una lunghezza strettamente compresa tra 2^N e 2^{N+1}

$\Rightarrow uv^2w \notin L_2$

$\Rightarrow L_2$ non è regolare

$L_3 = \{ ww \mid w \in \{a,b\}^* \}$ è regolare? <u>NO</u>

- Fissiamo $N > 0$ $(\forall N > 0)$
- Scegliamo $z = a^N b^N a^N b^N$ $(\exists z \in L \text{ con } |z| \geq N)$
- Per ogni u, v, w tali che $z = uvw$ e $|uv| \leq N$ e $|v| \geq 1$
  deve essere che v è fatta solo di "a" con $1 \leq |v| \leq N$
  Sia $v = a^J$ con $J \geq 1$
- Prendiamo $K = 2$.

$$uv^2w = a^{N+J} b^N a^N b^N \notin L_3$$

$\Rightarrow L_3$ non è regolare

$L_4 = \{ ww^R \mid w \in \{a,b\}^* \}$

- Fissiamo $N > 0$ $(\forall N > 0)$
- Scegliamo $z = a^N b^N b^N a^N$ $(\exists z \in L \text{ con } |z| \geq N)$
- Per ogni u, v, w tali che $z = uvw$, $|uv| \leq N$ e $|v| \geq 1$
  deve essere che v è fatta solo di "a" $\Rightarrow v = a^J$ con $J \geq 1$
- Prendiamo $K = 2$

$$uv^2w = a^{N+J} b^N b^N a^N \notin L_4$$

$\Rightarrow L_4$ non è regolare

$L_5 = \{a^n \mid n$ è un numero primo$\}$

- Fissiamo $N > 0$ $\quad (\forall N > 0)$
- Scegliamo $z = a^p$ con p primo e $p \geq N+2$
- Per ogni u, v, w tali che $z = uvw$ e

$$(i)\ |uv| \leq N$$
$$(ii)\ |v| \geq 1$$

deve essere $1 \leq |v| \leq N$. Sia $m = |v|$.

Allora $|uv^0w| = |uw| = p - m$ (che deve essere un primo se $uv^0w \in L_5$)

- Ora consideriamo $k = p - m$

$$|uv^{p-m}w| = |uw| + (p-m) \cdot |v| =$$
$$= (p-m) + (p-m) \cdot m$$
$$= (p-m) \cdot (1+m)$$ che non è un numero primo!!

$$\Rightarrow uv^{p-m}w \notin L_5$$

$\Rightarrow L_5$ non è regolare

$(p-m) \cdot (1+m)$ non è primo se $(p-m) \neq 1$ e $(1+m) \neq 1$. Ma
- $m \geq 1 \Rightarrow (1+m) > 1$
- $p \geq N+2$ e $m \leq N$ da cui $(p-m) \geq 2$

Altre Proprietà dei Ling. Regolari

La classe dei ling. regolari è chiusa per

(1) unione

(2) concatenazione

(3) stella di Kleene

(4) complementazione

(5) intersezione

Dim:

(1), (2) e (3) sono ovvie. Ad esempio, se L_1 e L_2 sono regolari, allora esistono s_1 e s_2 espr. regolari tali che $L_1 = \mathcal{L}[s_1]$ e $L_2 = \mathcal{L}[s_2]$. Ma allora

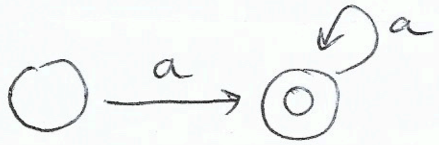
$L_1 \cup L_2 = \mathcal{L}[s_1 | s_2]$, cioè $L_1 \cup L_2$ è regolare.

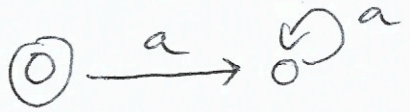
(4) è vera perché, dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$ tale che $L = L[M]$, possiamo costruire il DFA $\bar{M} = (\Sigma, Q, \delta, q_0, Q \setminus F)$ tale che $\bar{L} = L[\bar{M}]$. Infatti $w \in L[M]$ sse $w \notin L[\bar{M}]$ e quindi

$$L[\bar{M}] = \Sigma^* \setminus L[M]$$

(5) discende dalla legge di De Morgan

$$L(M_1) \cap L(M_2) = \overline{\overline{L(M_1)} \cup \overline{L(M_2)}}$$

M  $L[M] = a^+$

$\bar{M}$  $L[\bar{M}] = \{\varepsilon\}$

La chiusura per intersezione può essere utile per dimostrare che un ling. non ~~è~~ è regolare.
Infatti, se $L \cap L_{reg} = L_{non-reg} \Rightarrow L$ non è reg.

$L = \{w \in \{a,b\}^* \mid$ in w occorrono tante "a" quante "b" $\}$

L è regolare? Se lo fosse, allora $L \cap a^* b^*$ dovrebbe essere regolare. Ma

$L \cap a^* b^* = \{a^n b^n \mid n \geq 0\}$ che abbiamo dimostrato non essere regolare

$\Rightarrow L$ non è regolare

Modo alternativo per dimostrare la chiusura per $\cap$

Dati $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ e $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$,

$M = (Q_1 \times Q_2, \Sigma, \delta, (q_{01}, q_{02}), F_1 \times F_2)$ con

$$\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$$

è tale che $L[M] = L[M_1] \cap L[M_2]$

(M_1, M_2 e M sono DFA)

Per i linguaggi regolari, si possono decidere (cioè verificare algoritmicamente) le seguenti proprietà: (M, M_1, M_2 sono DFA)

- $w \in L[M]$? Basta leggere w e vedere se si è finiti su uno stato finale

- $L[M] = \emptyset$? "Esiste un cammino aciclico dallo stato iniziale ad uno finale?"

- $L[M] = A^*$? $(\iff L[\bar{M}] = \emptyset)$

- $L[M_1] \subseteq L[M_2]$? $(\iff L[\bar{M}_2] \wedge L[M_1] = \emptyset)$

- $L[M_1] = L[M_2]$? $(\iff L[M_1] \subseteq L[M_2] \wedge L[M_2] \subseteq L[M_1])$
  o anche: costruisco il DFA minimo per M_1 e M_2 e verifico se sono isomorfi

- $L[M]$ è infinito? "$\exists$ una parola $z \in L[M]$ tale che $n \leq z \leq 2n$ dove $n = |Q|$?"