

Linguaggi formali

Stringhe Linguaggi
Grammatiche

Simboli, Alfabeto e stringhe

- Un **simbolo** è una entità atomica
- Un **alfabeto** è un insieme finito di simboli

$$V = \{s_1, s_2, s_3, \dots, s_n\}$$

- Una **stringa** è una sequenza di simboli dell'alfabeto

$$S = a_1 a_2 a_3 \dots a_k \quad a_i \in V$$

- La **lunghezza** (k) della stringa è il numero di simboli che la compone
- La **stringa vuota** ε non contiene simboli (ha lunghezza 0)

Operazioni su stringhe

- Concatenazione

$$\begin{aligned} X &= x_1 x_2 \dots x_m & Y &= y_1 y_2 \dots y_n \\ XY &= x_1 x_2 \dots x_m y_1 y_2 \dots y_n \\ |XY| &= |X| + |Y| = m + n \end{aligned}$$

La concatenazione è associativa ma non commutativa

- Potenza n-esima S^n

- E' la stringa ottenuta concatenando n stringhe uguali a S
- $S^1 = S$ e $S^0 = \epsilon$
- $|S^n| = n |S|$

$$S = +\text{id} \quad S^2 = +\text{id} + \text{id} \quad S^3 = +\text{id} + \text{id} + \text{id} \dots$$

Linguaggi

- Un **linguaggio** è un insieme di stringhe definite su un alfabeto V
 - per enumerazione (... se è finito)
 - con regole insiemistiche
 - con una grammatica
- La **cardinalità** di un linguaggio è il numero di stringhe che lo compongono

$V = \{0, 1\}$ alfabeto (*dizionario*)

$$L_1 = \{\epsilon, 0, 00, 000, 0000, 00000\} \quad |L_1| = 6$$

$$L_2 = \{1, 01, 001, 0001, 00001, \dots\} = \{0^n 1 | n \geq 1\} \quad |L_2| = \infty$$

Operazioni sui linguaggi

Sono insiemi.. quindi si possono definire gli operatori

- **Unione**

$$L_z = L_x \cup L_y = \{z | z \in L_x \vee z \in L_y\}$$

- **Intersezione**

$$L_z = L_x \cap L_y = \{z | z \in L_x \wedge z \in L_y\}$$

- **Differenza**

$$L_z = L_x - L_y = \{z | z \in L_x \wedge z \notin L_y\}$$

Esempio

$$L_1 = \{0, 00, 000\} \quad L_2 = \{1, 11, 111\}$$

$$L_1 \cup L_2 = \{0, 00, 000, 1, 11, 111\}$$

$$L_1 \cap L_2 = \emptyset$$

Concatenazione e chiusura

- Concatenazione

$$L_{xy} = L_x \cdot L_y = L_x L_y = \{xy | x \in L_x \wedge y \in L_y\}$$

- Potenza n-esima L^n

- E' il linguaggio risultante dalla concatenazione di n linguaggi uguali a L
 - $L^1 = L$ e $L^0 = \{\epsilon\}$

- Chiusura L^*

E' il linguaggio dato dall'unione di tutte le potenze n-esime di L

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Chiusura e universo linguistico

- Chiusura positiva L^+

E' il linguaggio dato dall'unione di tutte le potenze n-esime di L con $n > 0$

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

- Universo linguistico

Dato un alfabeto V l'universo linguistico su V, indicato con V^* , è l'insieme di tutte le possibili stringhe che si possono costruire con i simboli di V inclusa la stringa vuota

- Linguaggio

E' un sottoinsieme dell'universo linguistico

$$L(V) \subseteq V^*$$

Esempi

$$V = \{0,1\}$$

$$L_1 = \{0,01\} \quad L_2 = \{1,11\}$$

- $L_1 L_2 = \{01,011,0111\}$
- $L_1^* = \{\varepsilon, 0, 00, 000, \dots, 01, 0101, 010101, \dots, 001, 0001, \dots\}$
- $L_2^* = \{\varepsilon, 1, 11, 111, 1111, \dots\}$
- $L_2^+ = \{1, 11, 111, 1111, \dots\}$

V^* contiene tutte le stringhe di 0 e 1 (numeri binari)

Grammatiche

- Una grammatica è basata su un **insieme di regole** con le quali si possono **generare** le stringhe del linguaggio
- Una grammatica G è una quadrupla

$$G = \{T, N, P, S\}$$

- T è l'alfabeto dei simboli terminali (compongono le stringhe)
- N è un insieme (finito) di simboli non terminali (le categorie sintattiche) - $N \cap T = \emptyset$
- P è l'insieme delle regole della grammatica (produzioni)
- $S \in N$ è il simbolo iniziale (assioma)

Produzioni

- Una produzione è una relazione fra una coppia di stringhe

$$\alpha \rightarrow \beta$$

dove $\alpha \in (T \cup N)^+$ contiene almeno un simbolo di N e $\beta \in (V \cup N)^*$

- La produzione indica che α (la parte sinistra) può essere sostituita con β (la parte destra)

$$\begin{aligned} T &= \{a,b,c\} \\ N &= \{A,B,C\} \end{aligned}$$

$$\begin{aligned} A &\rightarrow abc \\ A &\rightarrow aBbc \\ Bb &\rightarrow bB \\ Bc &\rightarrow Cbcc \\ bC &\rightarrow Cb \\ aC &\rightarrow aaB \\ aC &\rightarrow aa \end{aligned}$$

produzioni

$$L(G) = \{a^n b^n c^n \mid n \geq 1\}$$

$$\begin{aligned} A &\Rightarrow aBbc \Rightarrow abBc \Rightarrow abCbcc \\ &\Rightarrow aCbbcc \Rightarrow aabbcc \end{aligned}$$

Derivazione

- Il meccanismo generativo della grammatica si basa sulla **derivazione**: si sostituisce una sottostringa che corrisponde col lato sinistro di una regola con la stringa sul suo lato destro

$$\begin{array}{ccccccc} abBc & \Rightarrow & abCbcc & \Rightarrow & aCbbcc & \Rightarrow & aabbcc \\ & & Bc \xrightarrow{\quad} Cbcc & & bC \xrightarrow{\quad} Cb & & aC \xrightarrow{\quad} aa \end{array}$$

- Date due stringhe $\gamma = \varphi\alpha\lambda$ e $\nu = \varphi\beta\lambda$, si dice che γ **produce direttamente** ν ($\gamma \Rightarrow \nu$) se $\alpha \xrightarrow{\quad} \beta$ è una produzione di G
- Si dice inoltre che γ **produce indirettamente** ν ($\gamma \Rightarrow^* \nu$) se esiste una sequenza di derivazioni dirette tali che

$$\gamma = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \alpha_{n-1} \Rightarrow \alpha_n = \nu$$

Linguaggio generato

- Data un grammatica G il linguaggio generato è l'insieme delle stringhe di simboli terminali derivabili a partire dall'assioma

$$L_G = \{ x \in T^* \mid S \Rightarrow^* x \}$$

Esempio: grammatica che genera le espressioni aritmetiche

$$T = \{\text{num}, +, *, (), \}$$

$$N = \{E\}$$

$$S = E$$

$$\begin{array}{l} E \xrightarrow{} (E) \\ E \xrightarrow{} \text{num} \\ E \xrightarrow{} E * E \\ E \xrightarrow{} E + E \end{array}$$

assioma

$$E \xrightarrow{} E * E \xrightarrow{} (E) * E \xrightarrow{} (E + E) * E$$

$$\xrightarrow{} (\text{num} + E) * E \xrightarrow{} (\text{num} + \text{num}) * E$$

$$\xrightarrow{} (\text{num} + \text{num}) * \text{num}$$

frase del linguaggio

Classificazione delle grammatiche

- La classificazione è dovuta a Chomsky
 - **Grammatiche a struttura di frase**
Le regole hanno l'unica restrizione che la parte sinistra contenga almeno un simbolo non terminale
 - **Grammatiche contestuali**
Le regole sono della forma $\phi X \gamma \rightarrow \phi \alpha \gamma$ con $\phi, \gamma \in (T \cup N)^*$, $X \in N$ e $\alpha \in (T \cup N)^+$ ovvero la sostituzione di X con α avviene solo nel contesto di ϕ e γ , ossia se X compare fra le stringhe ϕ e γ
 - **Grammatiche non contestuali**
Le regole sono della forma $X \rightarrow \alpha$ con $X \in N$ e $\alpha \in (T \cup N)^*$
 - **Grammatiche regolari (destre)**
Le regole sono della forma $X \rightarrow sY$ o $X \rightarrow s$ con $X, Y \in N$ e $s \in T^*$

Linguaggi regolari

- Sono possibili descrizioni con formalismi diversi e fra loro equivalenti
 - Automi a Stati Finiti (FSA)
 - Grammatiche regolari (RG)
 - Espressioni Regolari (RE)
- Ciascun formalismo è adatto ad un particolare scopo
 - Un automa a stati finiti definisce un riconoscitore per stabilire se una stringa appartiene ad un dato linguaggio regolare
 - Le grammatiche regolari definiscono un modello generativo delle stringhe
 - Le espressioni regolari descrivono la struttura delle stringhe del linguaggio

Espressioni regolari - Operatori Atomici

- Si definiscono degli operatori alla base di un'algebra
 - Operatori Atomici
 - Un simbolo dell'alfabeto
$$s \in V$$
 - La stringa vuota
$$\epsilon \in V^*$$
 - L'insieme vuoto
$$\emptyset$$
 - Una variabile che rappresenta una configurazione definita a sua volta da un'espressione regolare

Espressioni regolari - Valore di una RE

- Il valore di un'espressione regolare E descrive un linguaggio indicato con $L(E)$
 - Se $E = s$, $s \in V$ allora $L(E) = \{s\}$
 - Il valore di una RE che corrisponde ad un simbolo è un linguaggio che contiene una sola stringa di lunghezza 1 costituita dal simbolo stesso
 - Se $E = \epsilon$ allora $L(E) = \{\epsilon\}$
 - Il valore di una RE che corrisponde alla stringa vuota è un linguaggio che contiene solo la stringa vuota
 - Se $E = \emptyset$ allora $L(E) = \emptyset$
 - Il valore di una RE che corrisponde all'insieme vuoto è un linguaggio che non contiene elementi
 - Una variabile assume il valore corrispondente a quello dell'espressione regolare a cui fa riferimento

Espressioni regolari - Operatori: unione

- Si definiscono tre operatori che permettono di combinare RE per ottenere una nuova RE

- Unione di due RE $U = R | S$
 - $L(R | S) = L(R) \cup L(S)$

$$L(0) = \{0\} \quad L(1) = \{1\} \quad L(0|1) = \{0, 1\}$$

- L'operatore corrisponde all'operazione insiemistica di unione ed è quindi
 - Commutativo $R | S = S | R$
 - Associativo $R | S | U = (R | S) | U = R | (S | U)$
 - La cardinalità del linguaggio risultante è tale che

$$L(R|S) \leq |L(R)| + |L(S)|$$

Espressioni regolari - Concatenazione

- Concatenazione di due RE $C = RS$

- $L(RS) = L(R)L(S)$ - Il valore della RE è il linguaggio definito concatenando tutte le stringhe in $L(R)$ con quelle in $L(S)$

$$R = a \quad S = b \quad RS = ab \quad L(RS) = \{ab\}$$

- L'operatore non è commutativo $RS \neq SR$ in generale
 - L'operatore è associativo $RSU = (RS)U = R(SU)$
 - Per quanto riguarda le cardinalità del linguaggio definito dalla concatenazione

$$L(RS) \leq |L(R)| \cdot |L(S)|$$

- La stessa stringa può essere ottenuta in più modi da stringhe in $L(R)$ e $L(S)$

Espressioni regolari - un esempio

$$R = a \mid (ab) \quad S = c \mid (bc)$$

$$\begin{aligned} RS &= (a \mid (ab))(c \mid (bc)) = ac \mid (ab)c \mid a(bc) \mid (ab)(bc) = \\ &= ac \mid abc \mid abbc \end{aligned}$$

$$L(RS) = \{ac, abc, abbc\}$$

Vale la proprietà distributiva della concatenazione rispetto all'unione

$$(R(S \mid T)) = RS \mid RT \quad ((S \mid T)R) = SR \mid TR$$

Espressioni regolari - chiusura di Kleene

- La **chiusura di Kleene** è un operatore unario postfisso

$$(R)^*$$

- Ha la priorità massima fra tutti gli operatori (usare le parentesi!)
- Indica zero o più concatenazioni dell'espressione R
- $L(R^*)$ contiene
 - La stringa vuota ϵ (corrisponde a zero concatenazioni di R – R^0)
 - Tutte le stringhe di $L(R)$, $L(RR)$, $L(RRR)$,.... ovvero

$$L(R^*) = \bigcup_{i=0}^{\infty} L(R^i)$$

In forma di espressione regolare (impropria)

$$L(R^*) = \epsilon \mid R \mid RR \mid RRR \mid \dots \mid R^n \mid \dots$$

Espressioni regolari - esempi e precedenza

$$R = (a \mid b) \quad L(R) = \{a, b\}$$

$$R^* = (a \mid b)^* \quad L(R^*) = \{\varepsilon, a, b, aa, ab, bb, ba, aaa, aba, \dots\}$$

- La precedenza fra gli operatori è in ordine di priorità
 - Chiusura
 - Concatenazione
 - Unione
- Per costruire RE corrette si deve ricorrere alle parentesi ()

$$R = a \mid bc^*d = a \mid b(c^*)d = (a) \mid (b(c^*)d) = ((a) \mid (b(c^*)d))$$

$$L(R) = \{a, bd, bcd, bcd, \dots, bc^n d, \dots\}$$

Espressioni regolari - esempi e variabili

- Nomi di variabili in un linguaggio di programmazione
 - Stringhe che iniziano con una lettera e sono poi composte da caratteri alfanumerici

alpha = A | B | C | ... | Z | a | b | |z|

numeric = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |9

variableid = alpha (alpha | numeric)*

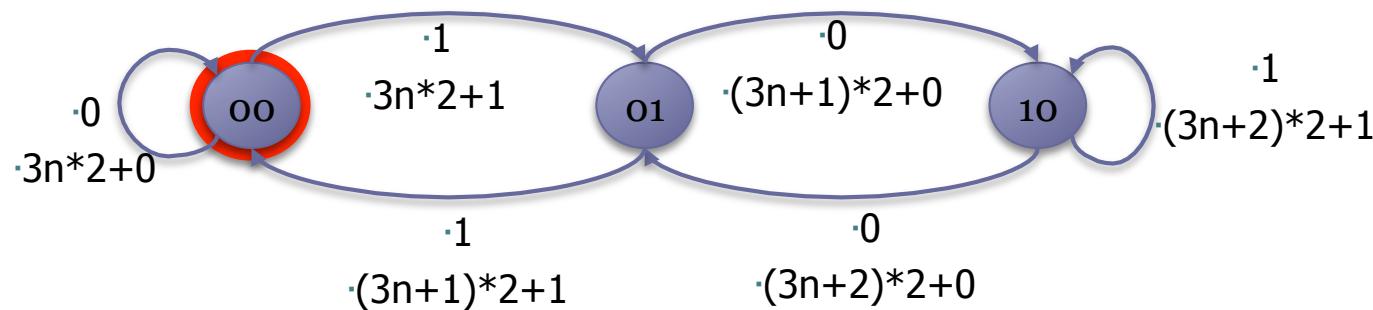
L(variableid) = {A,B,...,a,...,z,AA,....,V1,...,i1,...,myvar,...}

Espressioni regolari - esempi

- Tutte le stringhe di 0,1 che
 - terminano con 0 - $R = (0 \mid 1)^*0$
 - con almeno un 1 – $R = (0\mid 1)^*1(0,1)^*$
 - con al più un 1 – $R = 0^*10^*$
 - hanno nella terza posizione a partire da destra un 1
 - $R = (0\mid 1)^*1(0\mid 1)(0\mid 1)$
 - con parità pari (numero pari di 1) – $R = (0 \mid 10^*1)^*$
 - con tutte le sequenze di 1 di lunghezza pari – $R = (0 \mid 11)^*$
 - intepretate come numeri binari rappresentano multipli di 3 (11)
 - $R = (0\mid 11 \mid 1(01^*0)^*1)^*$

Espressioni regolari - multipli di 3 in binario

- L'espressione si può ricavare dal calcolo dei resti di una divisione per 3
 - I possibili resti sono 00, 01, 10
 - Si può costruire un automa a stati finiti che calcola i resti supponendo di scorrere il numero da sinistra a destra aggiungendo un bit per volta in coda



- I percorsi da 00 a 00 sono $0^* \mid 11^* \mid 101^*01$ e loro concatenazioni....

Espressioni regolari - equivalenza

- Due espressioni regolari si dicono **equivalenti** se definiscono lo stesso linguaggio

$$R \equiv S \iff L(R) = L(S)$$

- utilizzando le equivalenze algebriche fra espressioni si possono semplificare le espressioni regolari
 - Elemento neutro
 - unione $(\emptyset \mid R) = (R \mid \emptyset) = R$
 - concatenazione $\varepsilon R = R\varepsilon = R$
 - Elemento nullo
 - concatenazione $\emptyset R = R\emptyset = \emptyset$
 - Commutatività (unione) e Associatività (unione e concatenazione)

Espressioni regolari - equivalenze algebriche

- Distributività della concatenazione sull'unione
 - sinistra - $R(S|T) = RS | RT$ destra – $(S|T)R = SR | TR$
- Idempotenza dell'unione
 - $(R | R) = R$
- Equivalenze per la chiusura di Kleene
 - $\emptyset^* = \epsilon$
 - $RR^* = R^*R = R^+$ (una o più concatenazioni di stringhe in $L(R)$)
 - $RR^*|\epsilon = R^*$
- Esempio

$$(0|1)^*(10|11)(0|1) = (0|1)^*1(0|1)(0|1) = (0|1)^*1(00|01|10|11) = (0|1)^*(100|101|110|111)$$

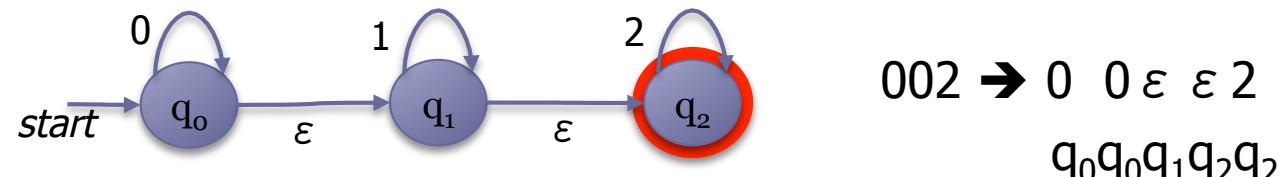
Espressioni regolari e FSA

- E' possibile trasformare una RE R in un automa a stati finiti non deterministico che riconosce il linguaggio definito da R
- E' possibile trasformare un automa a stati finiti (non deterministico) in una RE che definisce il linguaggio riconosciuto da tale automa

RE e FSA (NFSAs) sono modelli equivalenti per la definizione di linguaggi regolari

FSA con ϵ -transizioni

- Si estende la nozione di automa ammettendo transizioni di stato etichettate con il simbolo ϵ (ϵ -transizioni)
 - Questo comporta che l'automa possa avere transizioni di stato anche senza leggere dati in ingresso
 - L'automa accetta una stringa in ingresso se esiste almeno un cammino w dallo stato iniziale ad uno stato finale accettatore
 - Il cammino può comprendere archi di ϵ -transizioni oltre a quelli corrispondenti ai simboli della sequenza in ingresso
 - L'automa è quindi detto non deterministico perché possono esistere più cammini (sequenze di stati) per un dato ingresso



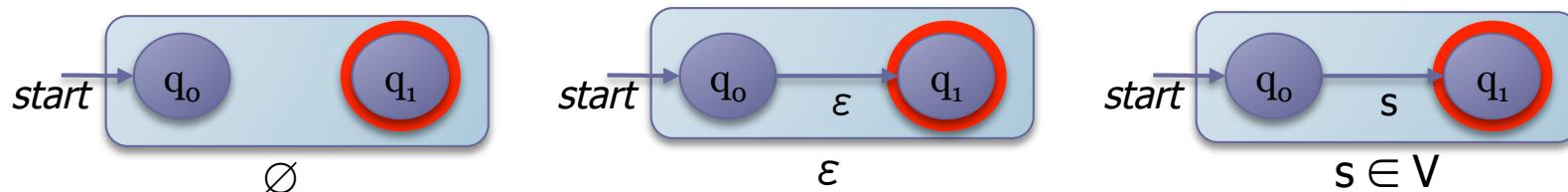
$$R = 0^* 1^* 2^*$$

FSA con ϵ -transizioni - definizione

- Un **automa a stati finiti con ϵ -transizioni** è definito da una quintupla (Q, V, δ, q_0, F) dove
 - $Q = \{q_0, \dots, q_n\}$ è l'insieme finito di stati
 - $V = \{s_1, s_2, \dots, s_k\}$ è l'alfabeto di ingresso
 - $\delta: Q \times (V \cup \{\epsilon\}) \rightarrow 2^Q$ è la funzione di transizione di stato (la transizione è verso un insieme di stati futuri data la presenza della ϵ -transizione)
 - $q_0 \in Q$ è lo stato iniziale
 - $F \subseteq Q$ è l'insieme degli stati finali

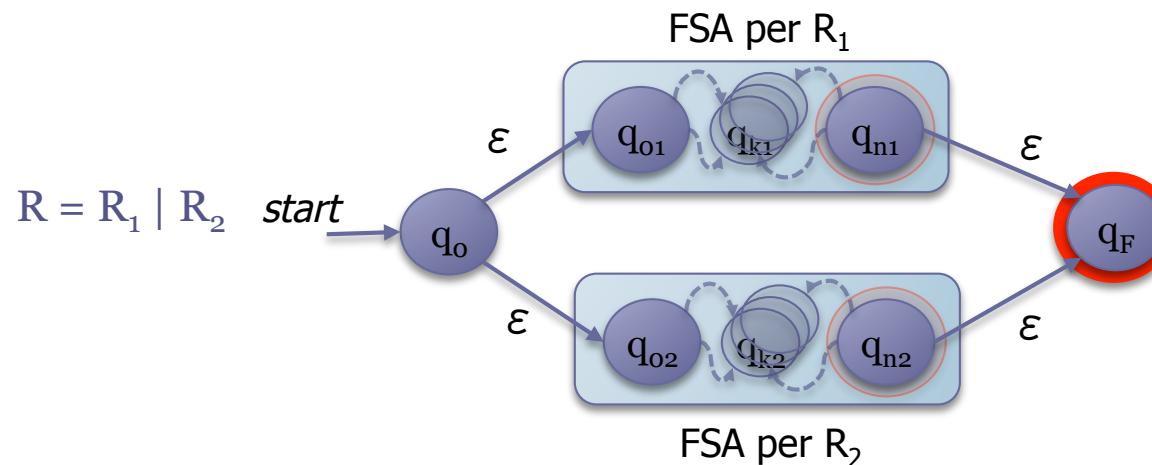
Da RE a FSA con ϵ -transizioni

- Data una RE R esiste un automa a stati finiti con ϵ -transizioni A che accetta solo le stringhe in $L(R)$
 - A ha un solo stato di accettazione
 - A non ha transizioni verso lo stato iniziale
 - A non ha transizioni uscenti dallo stato di accettazione
- La tesi si dimostra induttivamente per induzione sul numero n di operatori nell'espressione regolare R
 - $n=0$
R ha solo un operatore atomico \emptyset, ϵ o $s \in V$



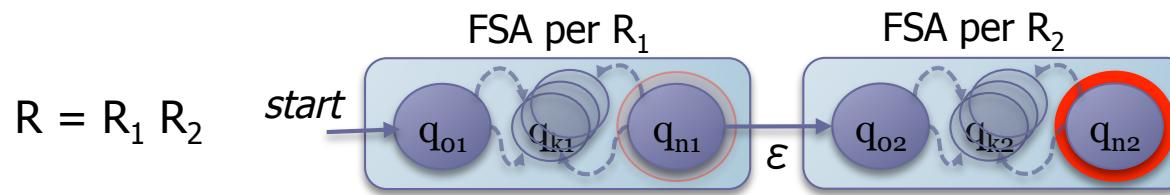
Da RE a FSA con ϵ -transizioni - $n > 0$

- Si suppone per induzione di avere la procedura costruttiva per ottenere l'automa per RE con $n-1$ operatori
 - Se si aggiunge un operatore per passare a n operatori si hanno i seguenti casi
 1. $R = R_1 \mid R_2$
 2. $R = R_1 R_2$
 3. $R = R_1^*$
 dove R_1 e R_2 hanno al più $n-1$ operatori

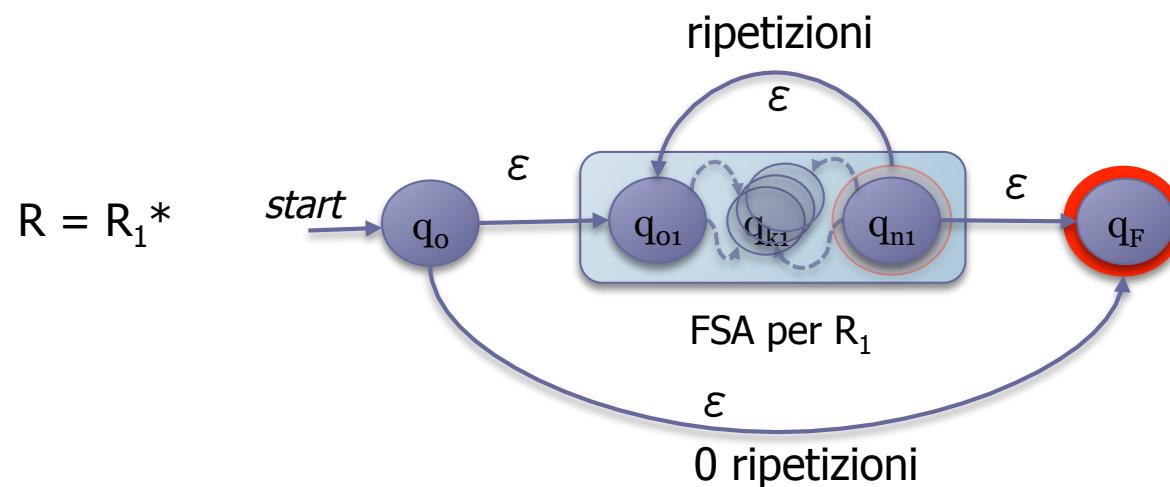


si accettano le stringhe di $L(R_1)$ passando per il cammino superiore e quelle di $L(R_2)$ passando per il cammino inferiore

Da RE a FSA con ϵ -transizioni - $n > 0$



si concatenano gli automi che riconoscono $L(R_1)$ e $L(R_2)$



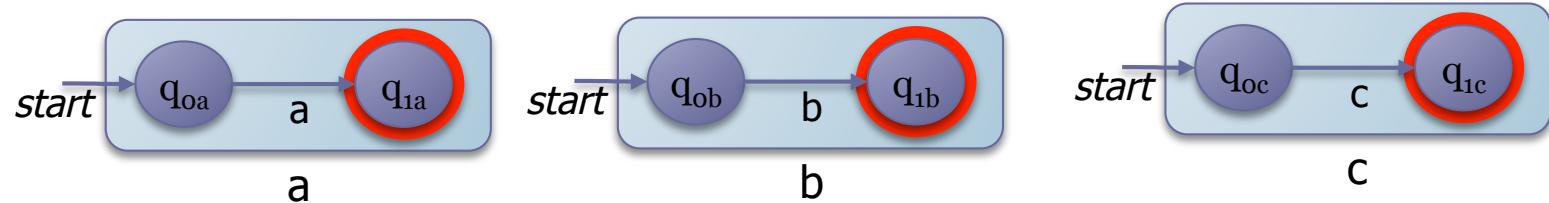
l'arco ϵ fra q_0 e q_F permette di riconoscere le concatenazioni delle stringhe in $L(R_1)$

l'arco ϵ fra q_0 e q_F permette di riconoscere la stringa nulla

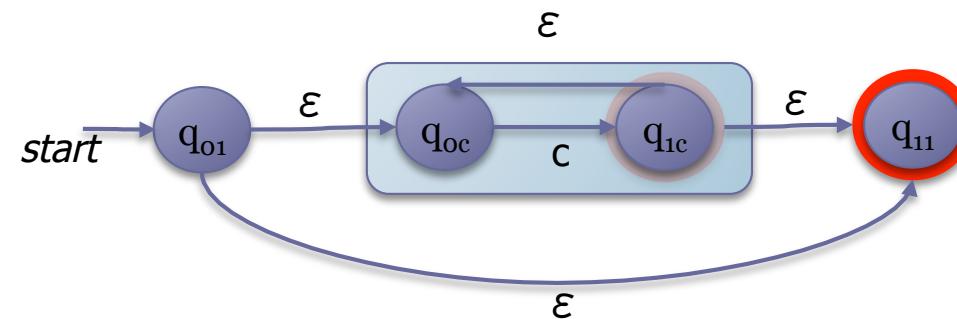
Da RE a FSA - esempio 1

$$R = a \mid bc^*$$

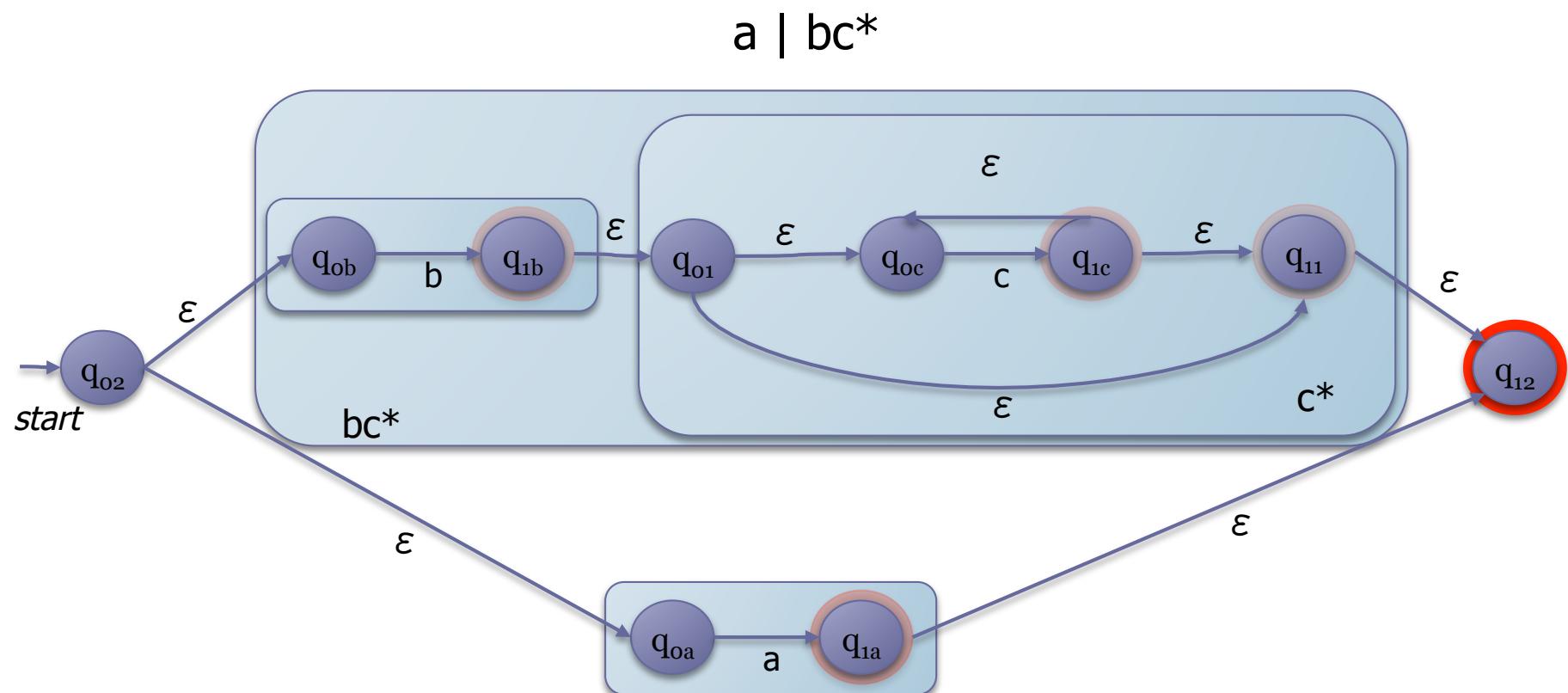
Atomi



c^*



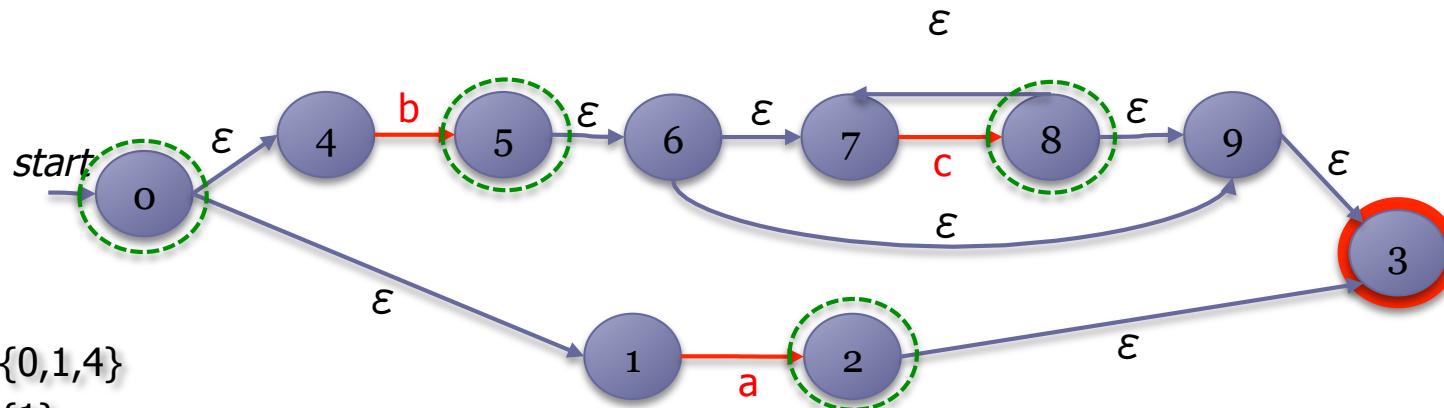
Da RE a FSA - esempio 2



Eliminazione delle ϵ -transizioni

- E' sempre possibile trasformare un automa con ϵ -transizioni in un automa deterministico (DFSA)
 - se ci si trova in uno stato q è come se fossimo anche in tutti gli stati raggiungibili da q con ϵ -transizioni
 - Per ogni stato si devono determinare tutti gli stati raggiungibili con sole ϵ -transizioni (raggiungibilità su un grafo)
 - Si eliminano tutte le transizioni non etichettate con ϵ
 - Si esegue una visita in profondità sul grafo
 - Ad ogni stato originario si associano tutti gli stati raggiungibili (si costruiscono gli stati candidati del DFA)

Da ϵ -FSA a DFSA - stati importanti



$$R(0) = \{0,1,4\}$$

$$R(1) = \{1\}$$

$$R(2) = \{2,3\}$$

$$R(3) = \{3\}$$

$$R(4) = \{4\}$$

$$R(5) = \{3,5,6,7,9\}$$

$$R(6) = \{3,7,9\}$$

$$R(7) = \{7\}$$

$$R(8) = \{3,7,9\}$$

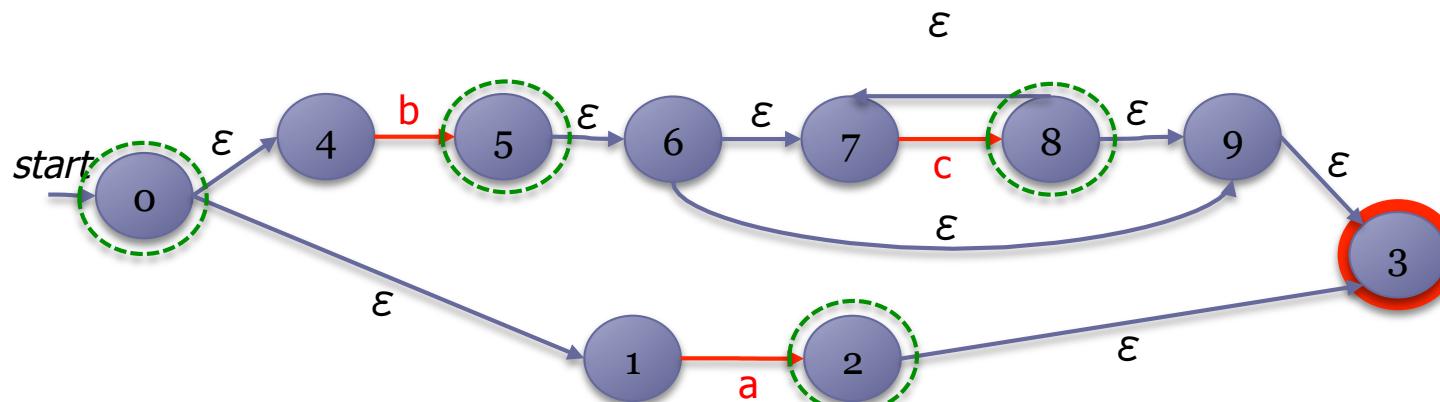
$$R(9) = \{3,9\}$$

Si definiscono gli **stati importanti**

- stati con un simbolo in ingresso
- stato iniziale

$$\cdot SI = \{0,2,5,8\}$$

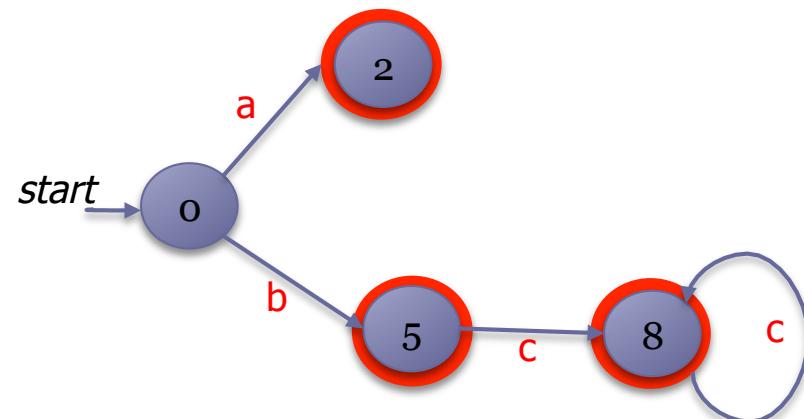
Da ϵ -FSA a DFSA - transizioni



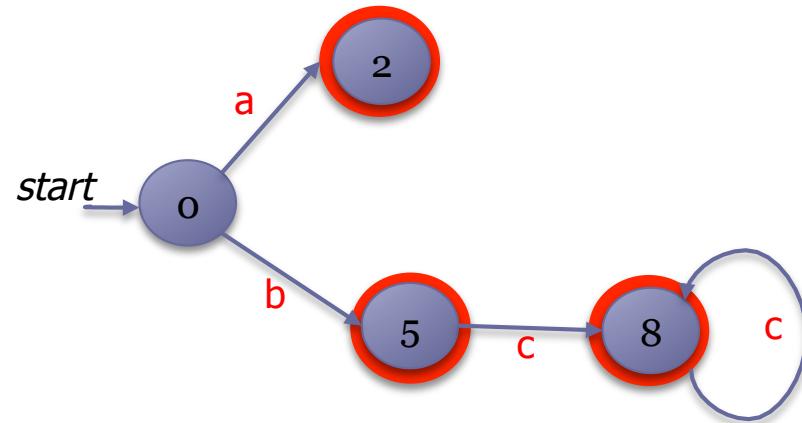
Esiste una transizione fra lo stato importante i e lo stato importante j con simbolo s, se

- esiste uno stato k in $R(i)$
- esiste una transizione da k a j con etichetta s

Uno stato importante i è di accettazione se almeno uno stato di accettazione è in $R(i)$

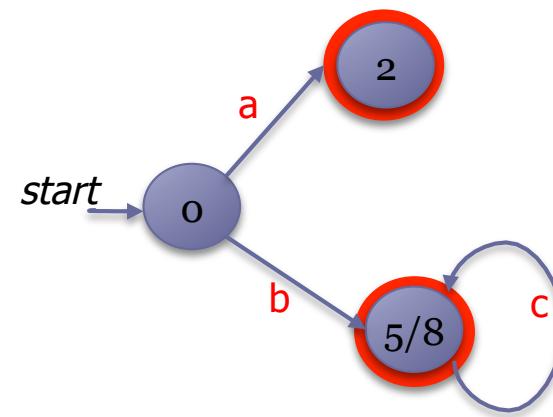


Da ϵ -FSA a DFSA - NDFSA e minimizzazione



L'automa risultante può essere non deterministico

- può avere più transizioni in uscita dallo stesso stato etichettate con lo stesso simbolo
- esiste comunque un algoritmo che permette di ottenere dal NDFSA un DFSA (si aggiungono stati corrispondenti agli insiemi di stati raggiungibili con le transizioni multiple)
- NDFSA, DFSA e ϵ -FSA sono modelli equivalenti

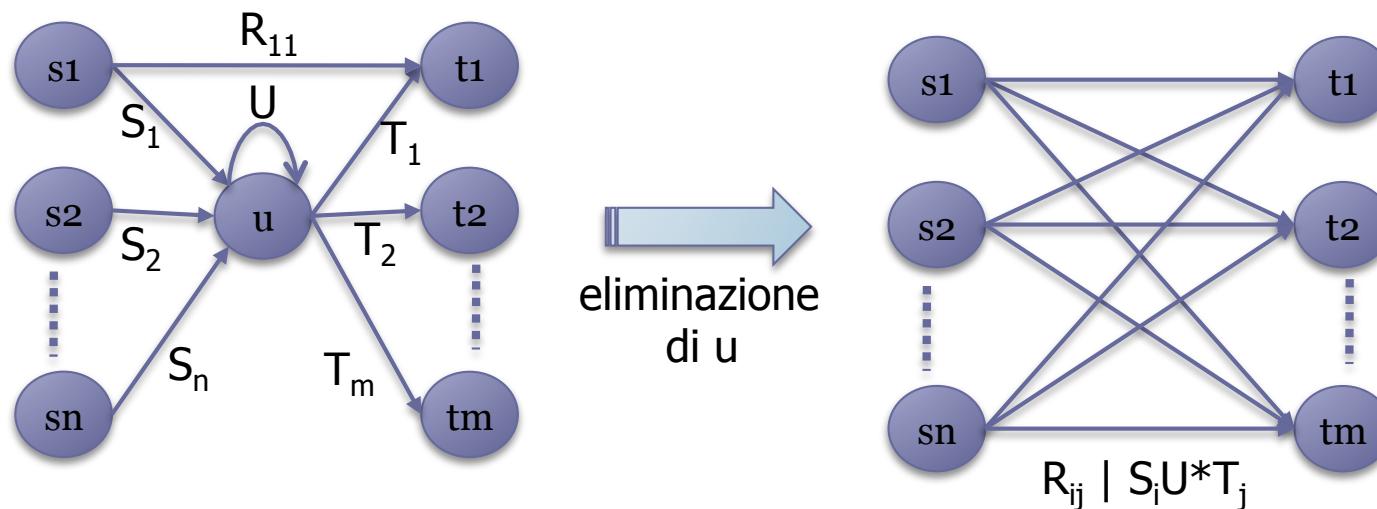


L'automa può essere minimizzato trovando le classi di stati equivalenti

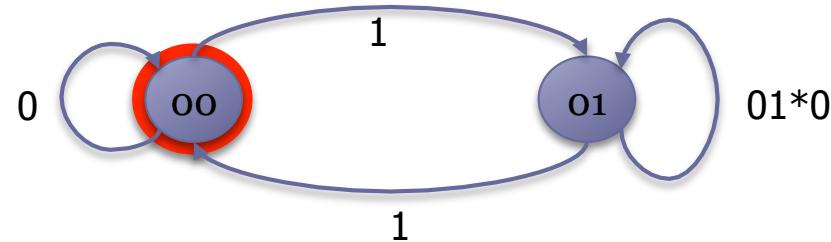
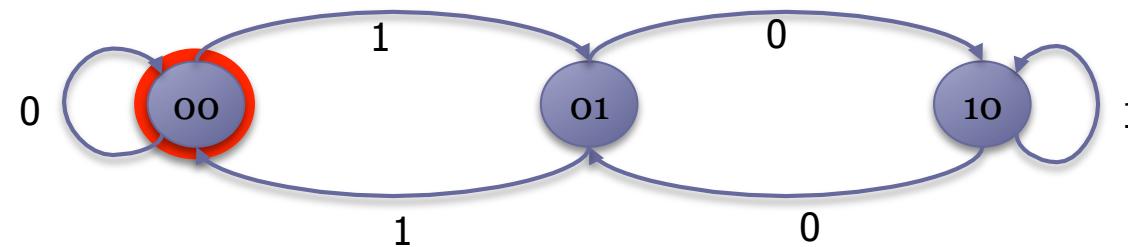
- equivalenza a 0 passi (stessa uscita) $\{0\} \{2,5,8\}$
- equivalenza a 1 passo (ingresso a b c) $\{0\} \{2\} \{5,8\}$ (si differenziano per c)
- da 2 passi in poi 5 e 8 sono indistinguibili

Da FSA a RE

- Per ogni FSA A esiste una espressione regolare $R(A)$ che definisce lo stesso insieme di stringhe riconosciute da A
 - Si può ricavare per eliminazione successiva degli stati
 - Gli archi sono etichettati con espressioni regolari che descrivono i cammini generati dagli stati eliminati fino ad un certo passo

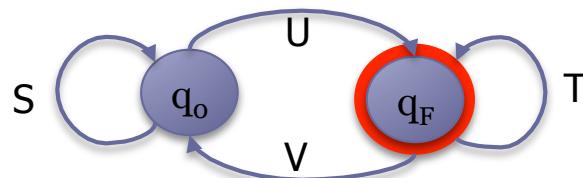


Da FSA a RE - esempio



Da FSA a RE - riduzione completa

- Il procedimento di riduzione deve essere effettuato per ogni stato accettatore
 - L'espressione regolare finale è l'unione delle espressioni regolari ottenute per ogni stato accettatore
 - Se si considera uno stato accettatore l'espressione regolare corrispondente è l'etichetta del cammino fra lo stato iniziale q_0 e lo stato accettatore q_F
 - Si eliminano tutti gli stati fino a che non rimangono solo q_0 e q_F



- Si determina l'espressione regolare che descrive i cammini che iniziano in q_0 e terminano in q_F

$$R = S^* U (T | V S^* U)^*$$

Applicazioni e standard per RE

- Sono disponibili molti programmi/librerie che utilizzano RE o hanno il supporto per la gestione di RE
 - comandi di ricerca in editor di testi
 - programmi di ricerca di pattern in file (grep, awk)
 - funzioni di libreria che implementano il matching con espressioni regolari (regex della stdlib C, supporto in PHP, perl, ecc.)
 - programmi per la generazione di analizzatori lessicali (lex)
- Lo standard IEEE POSIX 1003.2 definisce uno standard per scrivere RE
 - prevede Extended RE e Basic RE (obsolete)

POSIX 1003.2 - operatori

- L'operatore di unione è indicato dal carattere |
- L'operatore di concatenazione si ottiene specificando in sequenza i simboli o le classi di simboli da concatenare
- Sono definiti gli operatori unari
 - * - 0 o più occorrenze dell'operando a sinistra
 - + - 1 o più occorrenze dell'operando a sinistra
 - ? - 0 o 1 occorenza dell'operando a sinistra
 - {n} - esattamente n occorrenze dell'operando a sinistra
 - {n,m} - fra n e m occorrenze dell'operando a sinistra
 - {n,} - più di n occorrenze dell'operando a sinistra

POSIX 1003.2 - atomi 1

- Atomi a cui si possono applicare gli operatori binari/unari
 - Un carattere
 - I caratteri speciali sono espressi usando la sequenza di escape \ es. \\ \| \. \^ \\$
 - Una RE fra ()
 - La stringa vuota ()
 - Espressione con parentesi quadre [] – classe di caratteri
 - [abcd] uno qualsiasi fra i caratteri elencati
 - [0-9] le cifre fra 0 e 9
 - [a-zA-Z] i caratteri minuscoli e maiuscoli
se si vuole specificare il carattere – nella lista va messo per primo
 - Un carattere qualsiasi .
 - L'inizio della linea ^
 - La fine della linea \$

POSIX 1003.2 - atomi 2

- Esclusione di una classe di caratteri
 - [^abc] tutti i caratteri esclusi a b c (il carattere ^ è subito dopo [)
- Classi predefinite di caratteri
 - [:digit:] solo i numeri fra 0 e 9
 - [:alnum:] qualsiasi carattere alfanumerico fra 0 e 9, a e z o A e Z
 - [:alpha:] qualsiasi carattere alfabetico
 - [:blanc:] spazio e TAB
 - [:punct:] qualsiasi carattere di punteggiatura
 - [:upper:] qualsiasi carattere alfabetico maiuscolo
 - [:lower:] qualsiasi carattere alfabetico minuscolo
 - ecc...

POSIX 1003.2 - esempi

- Una RE fa il match con la prima sottostringa di lunghezza massima che la verifica
- Esempi
 - stringhe che contengono le vocali in ordine
 $.*a.*e.*i.*o.*u.*$
 - numeri con cifre decimali
 $[0-9]^+ \cdot [0-9]^* | \cdot [0-9]^+$
 - numeri con 2 cifre decimali
 $[0-9]^+ \cdot [0-9]\{2\}$

Analisi lessicale e lex

- Il comando **lex** permette di generare uno scanner ovvero un modulo/programma che riconosce elementi lessicali in un testo
 - Si specifica come funziona lo scanner in un file sorgente di lex che contiene regole e codice (C)
 - lex genera un sorgente (C) lex.yy.c che contiene il codice della funzione yylex()
 - Si compila il sorgente linkando al librerie di lex (-lfl)
 - L'eseguibile analizza un file in ingresso ricercando le espressioni regolari specificate nelle regole. Quando una RE viene verificata da una sottostringa si esegue la parte di programma (C) corrispondente

Uso dello scanner

- Lo scanner generato permette di suddividere il file letto in token (sottostringhe indivisibili) come ad esempio
 - **identificatori**
 - **costanti**
 - **operatori**
 - **parole chiave**
- Ciascun token è specificato da una RE
- Flex ha come linguaggio target il C ma esistono versioni anche per altri linguaggi (es. Jflex per Java)

Flex - Fast Lexical Analyzer Generator

- Permette di generare uno scanner
 - Per default il testo che non verifica nessuna regola viene riprodotto in uscita, altrimenti viene eseguito il codice associato alla RE
 - Il file di regole ha la seguente struttura

definizioni

%%

regole

%%

codice C

definizioni di nome
condizioni di partenza

pattern (RE) AZIONE

codice (opzionale) copiato
direttamente in lex.yy.c
- codice di supporto (es. funzioni)

Flex - definizioni di nome

- E' una direttiva del tipo

NOME DEFINIZIONE

- NOME è un identificatore che inizia con una lettera
- La definizione è riferita come {NOME}
- DEFINIZIONE è una RE

Esempio

ID [A-Za-z][A-Za-z0-9]*

definisce {ID}

- Nella sezione definizioni le linee indentate o comprese fra %{ e %} (all'inizio della linea) sono copiate nel file lex.yy.c

Flex - Le regole

- Nella sezione regole testo indentato o fra %{ e }% all'inizio della sezione può essere usato per definire variabili locali alla procedura di scan
- Una regola ha la forma

PATTERN (RE) AZIONE

- La parte PATTERN è una RE con le seguenti aggiunte
 - Si possono specificare stringhe fra “ ” in cui i caratteri speciali (es. []) perdono il loro valore come operatori
 - Si possono specificare i caratteri col loro codice esadecimale (es. \x32)
 - r/s fa il match di r solo se è seguita da s
 - <s1,s2,s3>r verifica il matchdi r solo se lo scanner è in una delle condizioni s1,s2,s3.. (si può usare <*> per indicare ogni condizione)
 - <<EOF>> viene verificata dalla fine del file

Flex - applicazione delle regole

- Se il testo soddisfa più regole viene attivata quella soddisfatta dalla sottostringa più lunga
- Se a parità di lunghezza ci sono più regole che sono soddisfatte viene attivata la prima in ordine di dichiarazione
 - trovato il match il token trovato è reso disponibile nella variabile yytext e la sua lunghezza in yyleng
 - viene poi eseguita l'azione associata
 - se non c'è match l'input è copiato sull'output come default

Flex - esempio: contatore di linee/parole/caratteri

variabili globali	int nLines = 0, nChars=0, nWords = 0;
	%%
PATTERN (RE)	\n [^[:space:]]+ .
	++nLines; ++nChars; ++nWords; nChars += yylen; ++nChars;
	%%
Attivazione dello scanner	main() { yylex(); printf("%d lines, %d words, %d characters\n", nLines, nWords, nChars); }

Se si invertono le regole per i singoli caratteri . e per le parole [^[:space:]]+ lo scanner non funziona correttamente (fa sempre il match la prima regola)

Lo scanner legge l'input dallo stream yyin (per default stdin)

Flex - esempio: mini-linguaggio di programmazione

```
%{  
#include <math.h>  
%}  
DIGIT [0-9]  
ID   [a-zA-Z][a-zA-Z0-9]*  
%%  
  
{DIGIT}+ {printf("int %s (%d)\n", yytext, atoi(yytext));}  
{DIGIT}+.{DIGIT}+ {printf("float %s (%f)\n", yytext, atof(yytext));}  
if|for|while|do {printf("keyword %s \n", yytext);}  
int|float|double|struct {printf("data type %s \n", yytext);}  
{ID} {printf("identifier %s \n", yytext);}  
+"|"-|"*"/ {printf("arithmetic operator %s \n", yytext);}  
//[^\\n]* /* elimina commenti su una linea */  
/*(.|\n)* */ /* elimina commenti su (multi)linea */  
>{"|"} {printf("block delimiter \n");}  
[ \\t\\n]+ /* elimina gli spazi etc*/  
. {printf("invaid char %s \n", yytext);}  
%%
```

definizione dei NOMI

yytext è la stringa che fa match

Flex - variabili e azioni

- Due variabili permettono di far riferimento alla sottostringa che verifica una delle RE
 - `yytext`
 - per default è un `char *` ovvero è un riferimento al buffer in cui è memorizzato il testo originale
 - con la direttiva `%array` nella parte di inizializzazioni si indica che la variabile deve essere un `char []` ovvero una copia del buffer (se si vuole modificare)
 - `yytext`
 - è il numero di caratteri della sottostringa che fa il match con la RE
- L'azione permette di specificare il codice da eseguire quando una RE è verificata da una sottostringa
 - l'azione è codice C (fra `{}` se è su più righe)
 - se il codice contiene l'istruzione `return`, si esce dalla funzione `yylex()`. Se la si richiama la scansione riprende dal punto in cui si era arrestata.

Flex - direttive speciali nelle azioni

- Nelle azioni possono essere specificate delle direttive speciali
 - **ECHO**
 - copia yytext sull'output
 - **BEGIN(condizione)**
 - mette lo scanner nello stato “condizione”. Se definiti, gli stati permettono la selezione di un sottoinsieme di regole dipendenti dallo stato.
 - **REJECT**
 - Attiva la seconda regola migliore (può essere verificata dalla stessa sottostringa o solo da un suo prefisso)

```
\n          ++nLines; ++nChars;  
pippo      ++nPippo; REJECT;  
[^[:space:]]+ ++nWords; nChars += yyleng;  
.           ++nChars;
```

Flex - funzioni di libreria

- Nelle azioni si possono usare anche funzioni di libreria dello scanner
 - `ymove()`
 - Viene ricercato il match successivo e il suo valore è aggiunto a yytext
 - `yyless(n)`
 - reinserisce n caratteri nel buffer di ingresso
 - `unput(c)`
 - reinserisce il carattere c nel buffer di ingresso
 - `input()`
 - legge il prossimo carattere facendo avanzare di 1 il cursore di lettura dell'input
 - `yyterminate()`
 - equivale a return
 - `yyrestart()`
 - reinizializza lo scanner per leggere un nuovo file (non reinizializza la condizione) – yyin è il file da cui si legge (stdin per default)

Flex - condizioni

- Le **condizioni** permettono una attivazione selettiva delle regole

`<SC>RE {azione;}`

- la regola è attivata solo se lo scanner è nella condizione SC
- le condizioni sono definite nella sezione di inizializzazione
 - %s SC1 SC2 – condizioni inclusive (le RE senza condizione rimangono attive)
 - %x XC1 XC2 – condizioni esclusive (solo le RE con la condizione attuale sono attive – si attiva uno scanner “locale alla condizione”)
- lo scanner è messo nella condizione SC eseguendo la direttiva
 - BEGIN(SC)
- la condizione iniziale è ripristinata con
 - BEGIN(o) o BEGIN(INITIAL)
- YYSTART memorizza lo stato attuale (è un int)
- le RE di uno stesso stato si possono dichiarare in un blocco `<SC>{...}`

Flex - condizioni: esempio

RE nello stato
COMMENT

```
%x COMMENT
int nCLinee=0;
%%
/* BEGIN(COMMENT); nCLinee++;
<COMMENT>[^*\n]*      /* salta i caratteri che non sono * e \n */
<COMMENT>*+[^\n]*    /* salta * non seguito da * o */
<COMMENT>\n          nCLinee++;
<COMMENT>*+/*        BEGIN(INITIAL);
[^/]*/" [^*/]*
%%
%
```

- Conta le linee di commento C /* */

Flex - esempio: parsing delle costanti stringa in C

```
%x string
%%

inizio della
stringa con "char str_buf[1024], *str_buf_ptr;
\" str_buf_ptr = str_buf; BEGIN(string);

<string> {
\" { BEGIN(INITIAL); *str_buf_ptr = '\0';
    printf("%s\n",str_buf); }
\n printf("Stringa non terminata\n"); yyterminate();
\\[0-7]{1,3} {int r; sscanf(yytext+1,"%o",&r);
              if(r>0xff) {printf("codice ASCII non valido\n"); yyterminate();}
              *(str_buf_ptr++) = r; }
\\[0-9]+ printf("codice ottale non valido\n"); yyterminate();
\\n      *(str_buf_ptr++) = '\n';
...
\\(.|\n)   *(str_buf_ptr++) = yytext[1];
[^\\n']+ {int i; for(i=0;i<yylen;i++) *(str_buf_ptr++) = yytext[i];}
}

parsing della
stringa fino a "
%%
```

Flex - buffer di input multipli

- L'uso di buffer di input multipli fornisce il supporto per la scansione simultanea di più file (es. include)
 - si interrompe momentaneamente lo scan di un file per passare all'analisi di un altro incluso
 - si possono definire più buffer di ingresso
 - `YY_BUFFER_STATE yy_create_buffer(FILE *file, int size)`
 - si può selezionare il buffer da analizzare
 - `void yy_switch_to_buffer(YY_BUFFER_STATE new_buffer)`
 - l'analisi continua col nuovo buffer senza variazioni di stato
 - si possono deallocare i buffer creati
 - `void yy_delete_buffer(YY_BUFFER_STATE buffer)`
 - `YY_CURRENT_BUFFER` si riferisce al buffer in uso
 - la regola `<<EOF>>` permette di gestire la fine dell'analisi su un file

Flex - esempio di include

inizio include

```
"#include" BEGIN(incl);
<incl>{
  [[:space:]]* /* salta spazi */
  \"[[:alnum:].[.]+" { if(stack_ptr>=MAX_DEPTH) { /*errore troppi livelli include */ }
    include_stack[stack_ptr++]=YY_CURRENT_BUFFER;
    strncpy(filename,yytext+1,yylen-1);
    filename[yylen-2]='\0';
    if(!(yyin=fopen(filename,"r"))){ /* errore apertura del file */ }
    yy_switch_to_buffer(yy_create_buffer(yyin,YY_BUF_SIZE));
    BEGIN(INITIAL); }
  [^[:space:]]+ /* errore di include */
}
```

estrazione del nome del file e apertura

```
<<EOF>> { if(--stack_ptr<0)
  yyterminate();
else {
  fclose(YY_CURRENT_BUFFER->yy_input_file);
  yy_delete_buffer(YY_CURRENT_BUFFER);
  yy_switch_to_buffer(include_stack[stack_ptr]) }
}
```

fine del file incluso