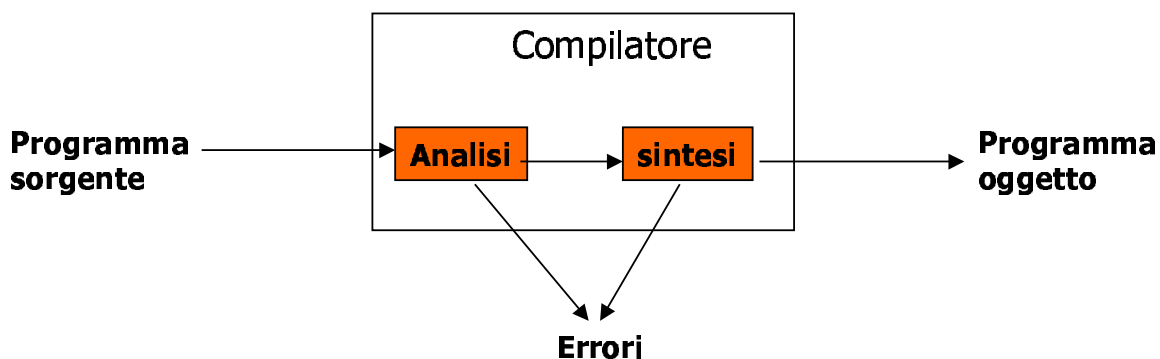


Linguaggi Formali e Compilatori

I compilatori

- Un **compilatore** è un **traduttore** in grado di trasformare un programma scritto in un linguaggio (**sorgente**) in un programma scritto in un altro linguaggio (**oggetto**)

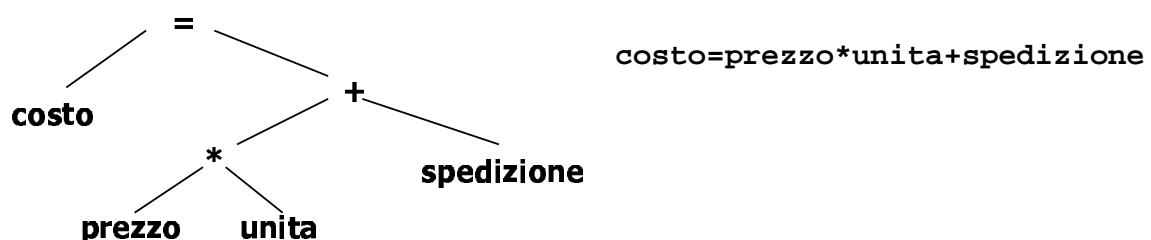


I compilatori II

- Il processo di compilazione consiste di due parti: **analisi** e **sintesi**
- analisi
 - divide il programma sorgente nei suoi componenti elementari
 - basandosi sulle relazioni fra i componenti crea una **rappresentazione intermedia** del programma
- sintesi
 - costruisce il programma oggetto dalla rappresentazione intermedia

Analisi

- I componenti elementari di un programma sono: identificatori, operatori, variabili,...
- La rappresentazione intermedia è codificata da una struttura dati a forma di albero (**albero sintattico**)



Sintesi



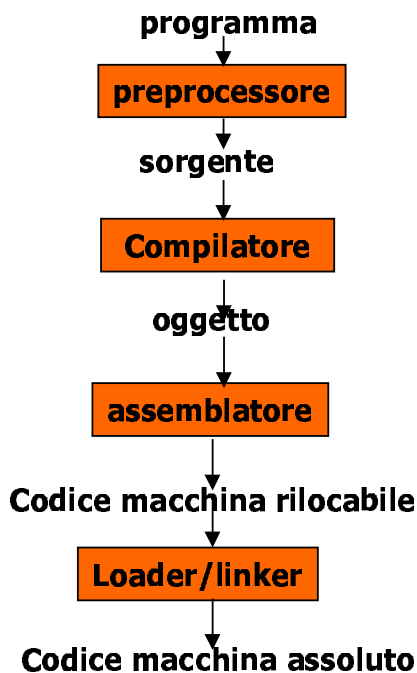
- L'albero sintattico descrive il programma sorgente e le relazioni fra i suoi componenti elementari
- La fase di sintesi usa questa informazione per produrre un codice oggetto **semanticamente equivalente** al codice sorgente
- Il codice oggetto è un programma che può essere eseguito sulla stessa macchina o su una diversa

Campi di applicazione



- Molti strumenti implementano tecniche derivanti da quelle impiegate nei compilatori
 - Interpreti
 - Editori guidati da sintassi
 - RAD
 - Browser
 - Word processor
 - Formattatori di testo
 - Interpreti di interrogazioni

L'uso di un tipico compilatore



- La tipica compilazione di un programma prevede l'uso di altri strumenti oltre al compilatore

- un preprocessore
- un assembler
- un loader/linker

La struttura di un compilatore

- Un compilatore opera in diverse fasi

- **analisi lessicale:**

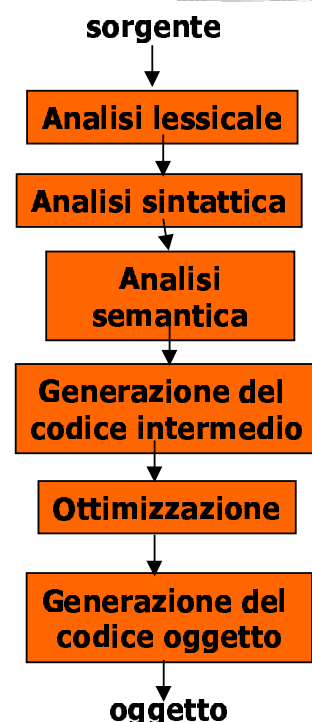
individua le componenti elementari

- **analisi sintattica:**

generazione dell'albero sintattico

- **analisi semantica:**

si fanno verifiche sul programma in ingresso per accertarsi che sia corretto (ad es., una variabile è dichiarata prima di essere usata)



La struttura di un compilatore II

■ generazione codice intermedio:

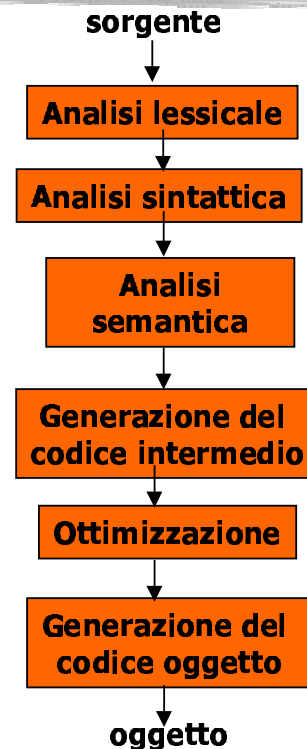
si genera un codice intermedio indipendente dalla macchina per la quale si compila

■ ottimizzazione:

il codice viene ottimizzato (ad es., si rimuove il codice non raggiungibile)

■ generazione del codice oggetto:

Il codice intermedio ottimizzato viene tradotto nel codice oggetto



Analisi lessicale e sintattica

■ Analisi lessicale

costo=prezzo*10+spedizione



id1=id2*10+id3

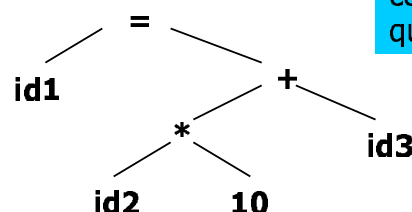
Tabella dei simboli

costo
prezzo	...
spedizione
....

E' inizializzata dall'analizzatore lessicale e completata da quello sintattico

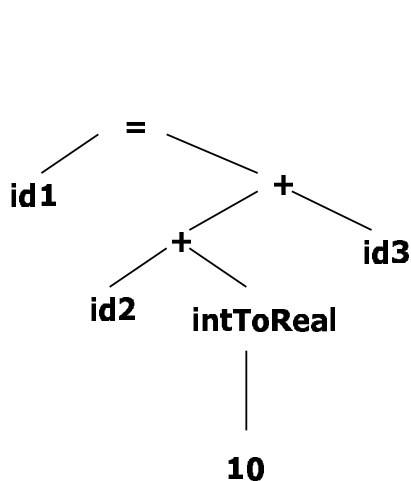
■ Analisi sintattica

id1=id2*10+id3



Analisi semantica

■ Analisi semantica

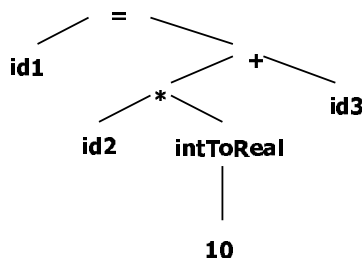


Controlli

- id1, id2, id3 sono stati definiti ?
- id2, id3 sono stati inizializzati?
- id1, id2, id3 sono di tipo Real?
-

Generazione del codice intermedio e ottimizzazione

■ Generazione codice intermedio



```
tmp1=intToReal(10)
tmp2=id2*tmp1
tmp3=id3+tmp2
id1=tmp3
```

■ Ottimizzazione

```
tmp1=id2*10.0
id1=tmp1+id3
```

Generazione del codice oggetto

■ Generazione codice oggetto

```
tmp1=id2*10.0  
id1=tmp1+id3
```

Codice generico



```
movf id2, R2  
mulf #10.0, R2  
movf id3, R1  
addf R2, R1  
movf R1, id1
```

**Assembler per una
architettura specifica**

Cosa studieremo

- Ci concentreremo sull'analisi lessicale e sintattica (le prime due fasi)
- analisi lessicale:
 - linguaggi regolari (i linguaggi che riconosce)
 - come si costruisce un analizzatore lessicale
- analisi sintattica:
 - linguaggi liberi da contesto (i linguaggi che riconosce)
 - esistono vari tipi di analizzatori sintattici, ne vedremo alcuni

Analisi lessicale

Linguaggi



- Un **alfabeto** $\Sigma = \{a_1, a_2, \dots, a_n\}$ è un insieme di simboli
- Una **stringa** è una sequenza finita di simboli
- Con l'insieme Σ^* si rappresenta l'insieme di tutte le stringhe sull'alfabeto Σ
- Un **linguaggio** $L(\Sigma)$ sull'alfabeto Σ è un sottoinsieme di Σ^* , cioè $L(\Sigma) \subseteq \Sigma^*$

Linguaggi II

- Ad esempio, l'insieme dei numeri binari reali

$$\Sigma = \{0, 1, "."\}$$

$$\Sigma^* = \{0, 1, 0.0, 0.1, \dots, 0.0.0, 0.0.1, \dots\}$$

contiene stringhe che **non rappresentano numeri reali**

$$L(\Sigma) = \{0, 1, 0.0, 0.1, \dots, \dots\}$$

contiene solo i numeri reali corretti

Linguaggi III

- Un linguaggio $L(\Sigma)$ può essere definito

- elencando le stringhe (se è finito)

- definendo delle regole che

- verificano se una stringa appartiene a L

- costruiscono tutte le stringhe che appartengono a L

- Le **espressioni regolari** definiscono una classe di **linguaggi regolari**

Espressioni regolari

- Le **espressioni regolari** definiscono un'algebra analoga a quella delle espressioni aritmetiche

- Simboli atomici

- Un carattere c ($c \in \Sigma$)
- Il simbolo ϵ (la stringa vuota)
- Il simbolo \emptyset (insieme vuoto)
- Una variabile

- 3 operatori

- La concatenazione (binario)
- L'unione $|$ (binario)
- La chiusura di Kleene $*$ (unario)

Un'espressione regolare
sull'alfabeto $\Sigma = \{0,1\}$

$0(0|1)^*$

Le parentesi
definiscono la
precedenza fra gli
operatori

Il valore delle espressioni regolari

- Il **valore di un'espressione** regolare E è dato dal linguaggio che essa rappresenta **$L(E)$**

- Valore dei simboli atomici

- $L(c) = \{c\}$ (c è un simbolo dell'alfabeto Σ)
- $L(\epsilon) = \{\epsilon\}$ (ϵ è un simbolo speciale che rappresenta la stringa vuota, $\epsilon \notin \Sigma$)
- $L(\emptyset) = \emptyset$ (l'insieme vuoto)
- Per una variabile v a cui è stata associata un'espressione E ,
 $L(v) = L(E)$

L'unione

- Il valore dell'operatore di **unione** è definito da

Se R e S sono due espressioni regolari, allora

$$L(R \mid S) = L(R) \cup L(S)$$

ad esempio, l'espressione che rappresenta il linguaggio che contiene 1, 0 e la stringa vuota è

$$L(1 \mid 0 \mid \epsilon)$$

- L'operatore \mid è **associativo** (l'unione fra insiemi è associativa)

$$R \mid S \mid T = (R \mid S) \mid T = R \mid (S \mid T)$$

La concatenazione

- La **concatenazione** non è rappresentata da alcun simbolo, ma dalla giustapposizione di due espressioni RS
- Il valore della concatenazione RS è costruito concatenando tutte le stringhe in R con quelle in S

$$L(RS) = \{rs \mid r \in R, s \in S\}$$

Concatenazione fra stringhe

- La concatenazione è associativa

$$IRST = (RS)T = R(ST)$$

La concatenazione II

■ Esempio 1

$$R=\{+,-\}, S=\{0,1\}$$

$$L(RS)=\{+0,-0,+1,-1\}$$

■ Esempio 2

$$R=\{\text{anda,da}\}, M=\{\text{re,to}\}$$

$$L(RS)=\{\text{andare,dare, andato,dato}\}$$

La cardinalità

- Con $|L(E)|$ si denota la **cardinalità** di L , ovvero il numero delle stringhe contenute in E

- La cardinalità soddisfa le seguenti proprietà

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

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

- E' possibile sostituire \leq con $=$?

La chiusura

- L'operatore di **chiusura** di Kleene $*$ indica 0 o più ripetizioni
- $L(R^*)$ contiene
 - La stringa vuota ε
 - tutte le stringhe di $L(R)$, $L(RR)$, $L(RRR)$, $L(RRRR)$,...
- Esempio
 - $R=\{0,1\}$
 - $L(R^*)$ è l'insieme di tutti gli interi binari più la stringa vuota
 - $L(R^*)=\{\varepsilon, 0, 1, 00, 01, 10, 11, \dots\}$

La chiusura

- Nel seguito R^n rappresenta R ripetuto n volte
 - $R^0=\{\varepsilon\}$, $R^1=R$, $R^2=RR$, $R^3=RRR$
- Formalmente:
 - $R^0=\{\varepsilon\}$, $R^n=R R^{n-1}$
 - $L(R^*) = \bigcup_{n=0.. \infty} L(R^n)$
- Attenzione: $L(R^*)$ non contiene stringhe infinite !!

Esempi di espressioni regolari

- Gli identificatori in PASCAL

Cifra (Carattere | Cifra)*

■ dove si è definito

Carattere = A | B ... | Z | a | b | ... | z

Cifra = 0 | 1 | ... | 9

- La **precedenza fra gli operatori** è la seguente

■ chiusura

■ concatenazione

■ unione

Esempi di espressioni regolari II

- Tutte le stringhe di 0 e 1 che terminano con 0

(0 | 1)*0

- Tutte le stringhe di 0 e 1 con almeno un 1

(0 | 1)*1 (0 | 1)*

- Tutte le stringhe di 0 e 1 con al più un 1

0*10*

- Tutte le stringhe di 0 e 1 tali che la terza posizione a partire da destra è 1

(0 | 1)*1(0 | 1)(0 | 1)

Esempi di espressioni regolari

III

- Tutte le stringhe di 0 e 1 che contengono un numero pari di 1

$$(0 | 10^*1)^*$$

- Tutte le stringhe di 0 e 1 tali che le sequenze di 1 hanno lunghezza pari

$$(0 | 11)^*$$

- Tutte le stringhe di 0 e 1 che rappresentano multipli di 3

$$(0 | 1(01^*0)^*1)^*$$

Proprietà algebriche delle espressioni regolari

- Due espressioni regolari E , S si dicono equivalenti $E \equiv S$ se

$$L(E) = L(S)$$

- Le proprietà algebriche delle espressioni regolari ricordano quelle delle espressioni aritmetiche (concatenazione = prodotto, unione=somma), ma la **concatenazione non è commutativa**

- Elemento neutro

$$E\epsilon \equiv \epsilon E \equiv E \quad (\epsilon \text{ è l'elemento neutro della concatenazione})$$

$$E|\emptyset \equiv \emptyset|E \equiv E \quad (\emptyset \text{ è l'elemento neutro dell'unione})$$

- Elemento nullo

$$E\emptyset \equiv \emptyset E \equiv \emptyset \quad (\emptyset \text{ è l'elemento nullo della concatenazione})$$

Proprietà algebriche delle espressioni regolari II

■ Commutatività

$$\text{■ } R|S \equiv S|R$$

■ (la concatenazione non è commutativa)

■ Associatività

$$\text{■ } R | S | T \equiv (R | S) | T \equiv R | (S | T)$$

$$\text{■ } RST \equiv (RS)T \equiv R(ST)$$

■ Distributività della concatenazione sull'unione

$$\text{■ } (S|T)R \equiv (SR|TR)$$

$$\text{■ } R(S|T) \equiv (RS|RT)$$

Proprietà algebriche delle espressioni regolari III

■ Idempotenza dell'unione

$$\text{■ } R|R \equiv R$$

■ Proprietà della chiusura

$$\text{■ } \emptyset^* \equiv \varepsilon$$

$$\text{■ } R^*R \equiv RR^*$$

$$\text{■ } R^*R | \varepsilon \equiv R^*$$

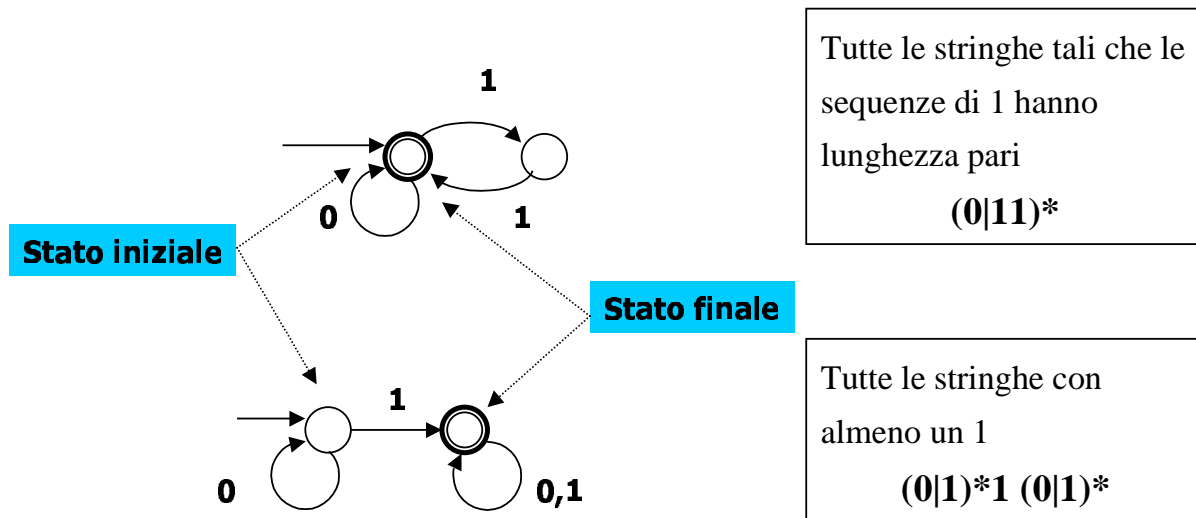
■ Esempio

$$\text{■ } (0|1)^*(10|11)(0|1)$$

$$\text{■ } \equiv (0|1)^*1(0|1)(0|1)$$

$$\text{■ } \equiv (0|1)^*1(00|01|10|11)$$

Espressioni regolari e automi a stati finiti



Espressioni regolari e automi a stati finiti II

■ Si può dimostrare che

■ **Data un'espressione regolare E esiste un automa a stati finiti A che accetta il linguaggio $L(E)$**

■ **Dato un automa a stati finiti A che accetta il linguaggio L, esiste un'espressione regolare E tale che $L=L(E)$**

■ Quindi

■ **Automi a stati finiti e espressioni regolari hanno lo stesso potere espressivo**

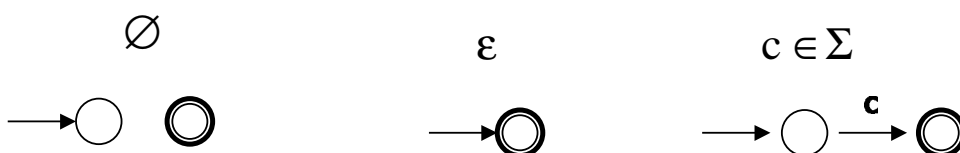
■ **Gli automi a stati finiti possono essere usati per riconoscere se una stringa appartiene o meno ad un linguaggio regolare**

Da espressioni regolari e automi a stati finiti

- La costruzione dell'automa richiede tre passi
 - Dall'espressione regolare si costruisce un **automa non deterministico** con ε -transizioni
 - Si trasforma l'automa non deterministico in un **automa deterministico**
 - Eventualmente, si trasforma ancora l'automa in modo da **minimizzare il numero degli stati**
- Di seguito vedremo come funzionano tali passi

Costruzione di automi non deterministici

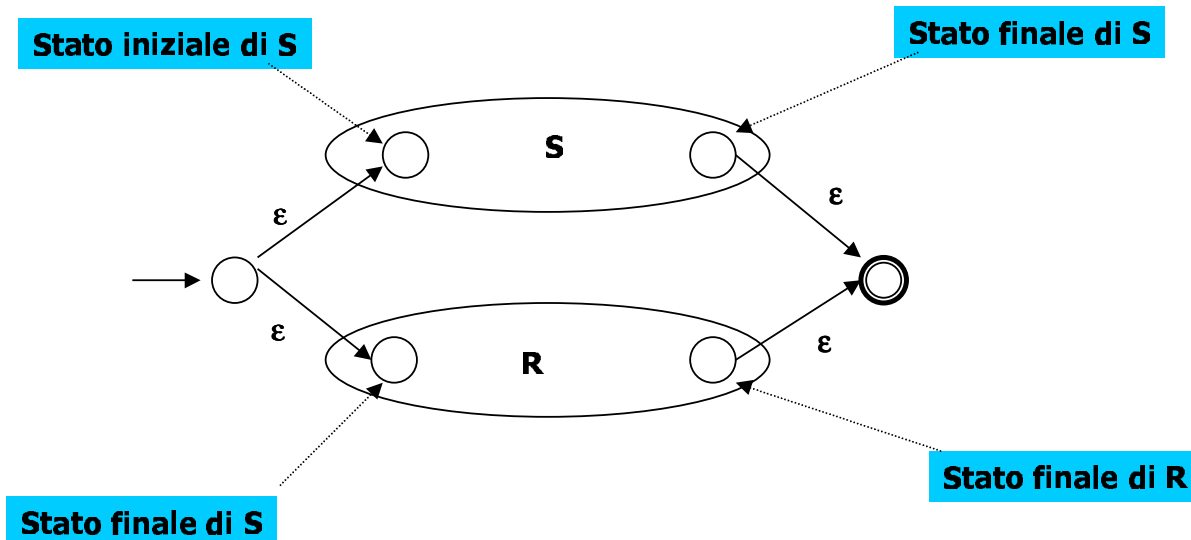
- La strategia di costruzione dell'automa non deterministico può essere **definita per induzione**
- Se l'espressione E è un operando atomico $\emptyset, \varepsilon, c \in \Sigma$



- Nel seguito si suppone di conoscere come costruire l'automa per le espressioni R, S e si mostra come costruire quelle per $R|S$, RS e R^*

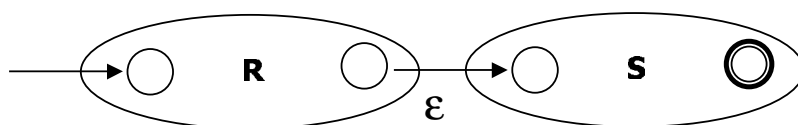
Unione

L'espressione $R|S$



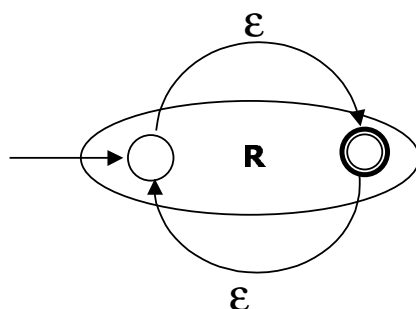
Concatenazione e chiusura

L'espressione RS



- Lo stato iniziale di RS coincide con quello di R
- Lo stato finale di RS coincide con quello di S

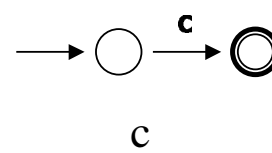
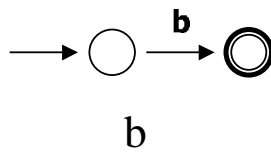
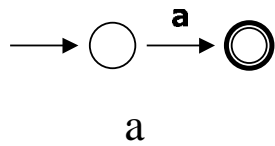
L'espressione R^*



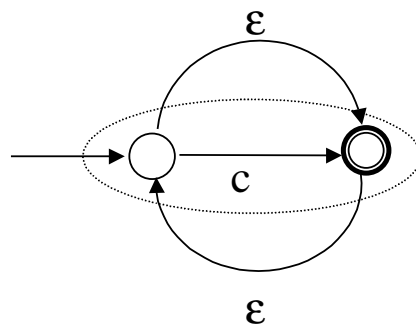
Gli stati iniziali e finali di R^* coincidono con quelli di R

Un esempio: $a|bc^*$

Le espressioni atomiche

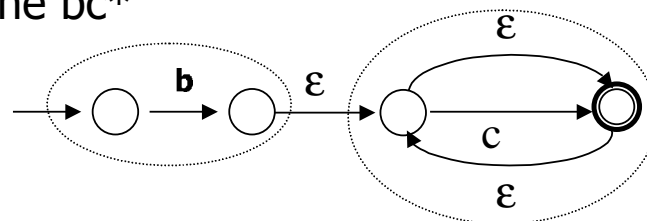


L'espressione c^*

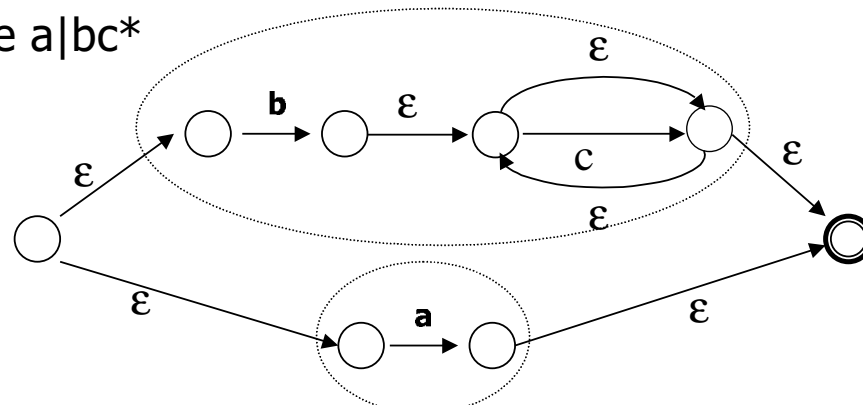


Un esempio: $a|bc^*$

L'espressione bc^*



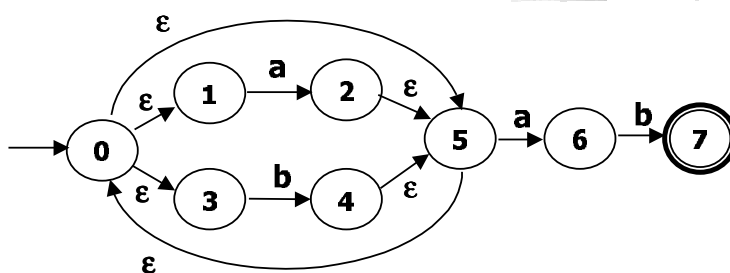
L'espressione $a|bc^*$



Da automa non deterministico ad automa deterministico

- E' possibile trasformare un automa non deterministico (NFA) in uno deterministico (DFA)
- Gli stati del DFA corrispondono a **insiemi di stati** del NFA
- Si definisce l' ϵ -chiusura di un insieme di stati $\{s_1, s_2, \dots, s_n\}$ del NFA
 - $\epsilon\text{-chiusura}(s) = \{\text{gli stati raggiungibili da } s \text{ con } \epsilon\text{-transizioni}\}$
 - $\epsilon\text{-chiusura}(\{s_1, s_2, \dots, s_n\}) = \bigcup_n \epsilon\text{-chiusura}(s_n)$
- Si definisce la funzione transizione
 - $\text{tr}(\{s_1, s_2, \dots, s_n\}, a) = \{\text{gli stati raggiungibili con una transizione } a \text{ da uno degli stati } s_1, s_2, \dots, s_n\}$

L' ϵ -chiusura

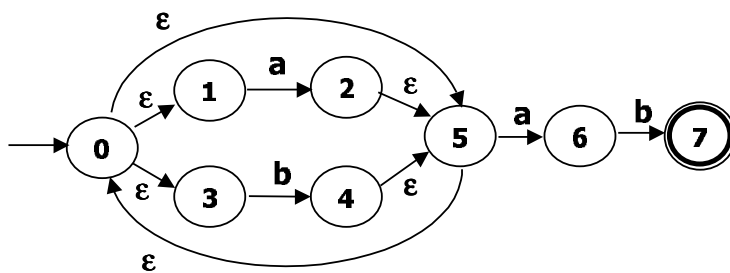


$(a|b)^*abb$

■ Esempi

- $\epsilon\text{-chiusura}(0) = \{0, 1, 3, 5\}$, $\epsilon\text{-chiusura}(1) = \{1\}$, $\epsilon\text{-chiusura}(2) = \{2, 5, 0, 1, 3\}$
- $\epsilon\text{-chiusura}(3) = \{3\}$, $\epsilon\text{-chiusura}(4) = \{4, 5, 0, 1, 3\}$, $\epsilon\text{-chiusura}(5) = \{5, 0, 1, 3\}$
- $\epsilon\text{-chiusura}(6) = \{6\}$, $\epsilon\text{-chiusura}(7) = \{7\}$
- $\epsilon\text{-chiusura}(\{5, 4\}) = \epsilon\text{-chiusura}(4) \cup \epsilon\text{-chiusura}(5) = \{4, 5, 0, 1, 3\} \cup \{5, 0, 1, 3\} = \{4, 5, 0, 1, 3\}$

La funzione tr



$(a|b)^*ab$

Esempi

$\text{tr}(0,a) = \emptyset, \text{tr}(1,a) = \{2\}, \text{tr}(2,a) = \emptyset, \text{tr}(3,a) = \emptyset, \dots$

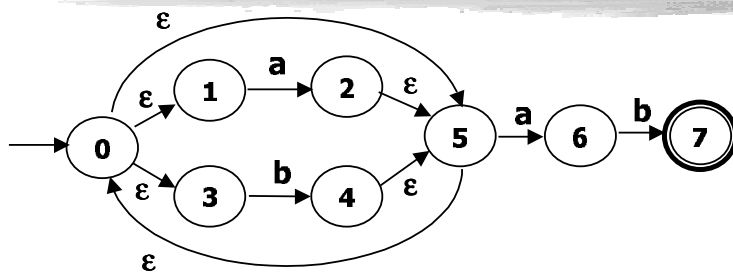
$\text{tr}(0,b) = \emptyset, \text{tr}(1,b) = \emptyset, \text{tr}(2,b) = \emptyset, \text{tr}(3,b) = \{4\}, \dots$

|

Da automa non deterministico ad automa deterministico II

```
| Inizializza il DFA con lo stato  $\epsilon$ -chiusura(0)
| For each S=(stato non ancora marcato del DFA) do
|   | marca S
|   | For each a=simbolo do
|     | R=  $\epsilon$ -chiusura(tr(S,a))
|     | if (R non esiste) then aggiungi lo stato R al DFA
|     | aggiungi una transizione a dallo stato S allo stato R
|   | end
| end
```

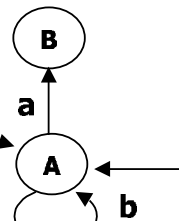
Un esempio



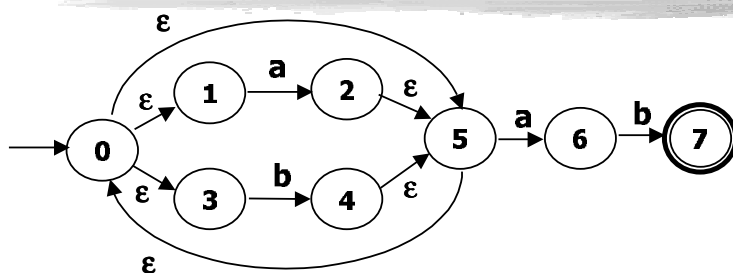
Si considerano solo gli stati importanti (si escludono gli stati che hanno solo ϵ -transizioni uscenti)

- $A = \epsilon\text{-chiusura}(0) = \{0, 1, 3, 5\} \equiv \{1, 3, 5\}$
- $B = \epsilon\text{-chiusura}(\text{tr}(A, a)) = \epsilon\text{-chiusura}(\{2, 6\}) = \{2, 5, 0, 1, 3, 6\} \equiv \{5, 1, 3, 6\}$
- $\epsilon\text{-chiusura}(\text{tr}(A, b)) = \epsilon\text{-chiusura}(\{4\}) = \{4, 5, 0, 1, 3\} \equiv \{5, 1, 3\} = A$

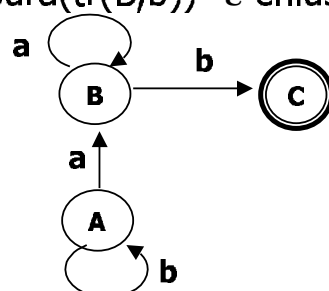
Lo stato iniziale è $\epsilon\text{-chiusura}(0)$



Un esempio II

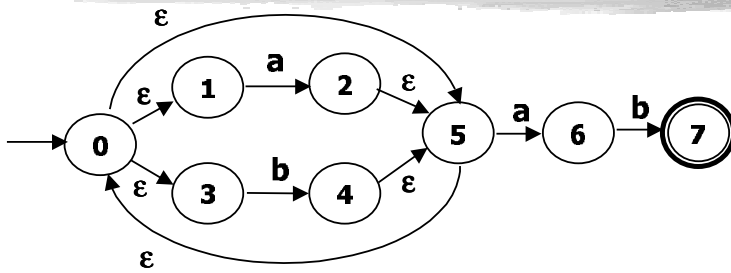


- $A = \{1, 3, 5\}, B = \{5, 1, 3, 6\}$
- $\epsilon\text{-chiusura}(\text{tr}(B, a)) = \epsilon\text{-chiusura}(\{6, 2\}) = \{6, 2, 5, 0, 1, 3\} \equiv B$
- $C = \epsilon\text{-chiusura}(\text{tr}(B, b)) = \epsilon\text{-chiusura}(\{4, 7\}) = \{4, 7, 5, 0, 1, 3\} \equiv \{7, 5, 1, 3\}$

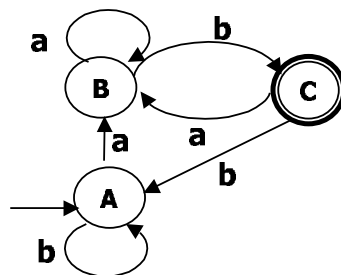


Ogni stato contenente uno stato finale del NFA è finale nel DFA

Un esempio III

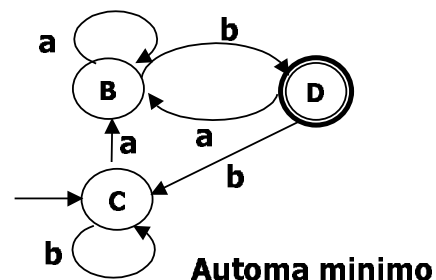
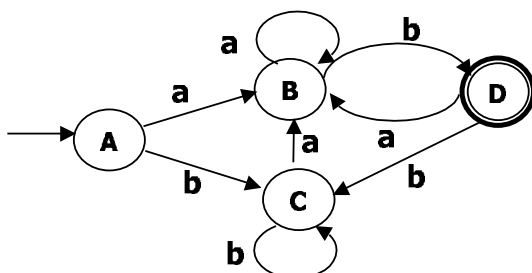


- $A = \{1, 3, 5\}$, $B = \{5, 1, 3, 6\}$, $C = \{7, 5, 1, 3\}$
- $\epsilon\text{-chiusura}(\text{tr}(C, a)) = \epsilon\text{-chiusura}(\{6, 2\}) = \{6, 2, 5, 0, 1, 3\} \equiv \{6, 5, 1, 3\} = B$
- $\epsilon\text{-chiusura}(\text{tr}(C, b)) = \epsilon\text{-chiusura}(\{4\}) = \{4, 5, 0, 1, 3\} \equiv \{5, 1, 3\} = A$



Trovare l'automa minimo

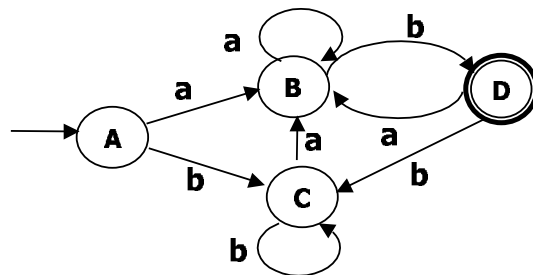
- L'automa prodotto potrebbe non essere quello minimo (che ha il numero minimo di stati)
- Si può dimostrare che per ogni linguaggio **l'automa minimo è unico**
- alcuni stati possono essere equivalenti, nell'esempio $A \equiv B$



Trovare gli stati equivalenti

- L'algoritmo inizia con una partizione $P = \{F, A\}$ degli stati che contiene l'insieme degli stati finali (F) e quello degli altri stati (A)
- **Repeat**
 - scegli un insieme S in P e un simbolo a tale che $\text{tr}(S, a) \notin P$
 - suddividi S in insiemi S_1, S_2, \dots, S_n tali che quando P viene aggiornato $\text{tr}(S_i, a) \in P$
 - **until** $\text{tr}(S, a) \in P$ per ogni a e ogni S

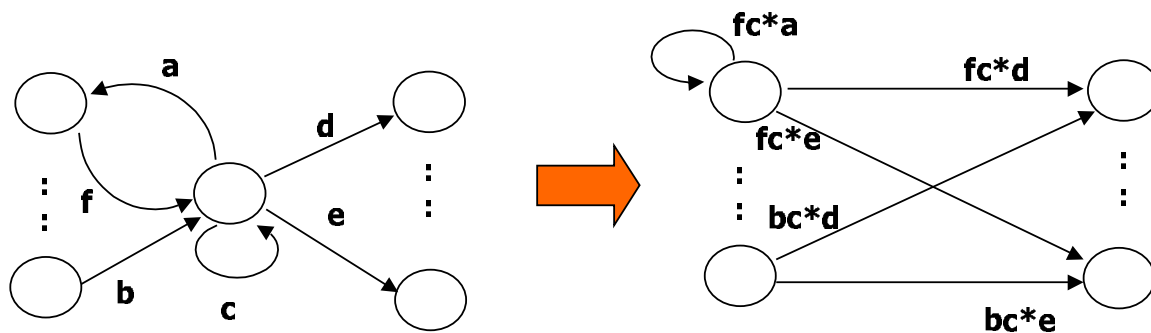
Un esempio



- All'inizio $P = \{\{A, B, C\}, \{D\}\}$
- Si considera $\{A, B, C\}$ e il simbolo b
- Si osserva che $\text{tr}(B, b) = \{D\}$, mentre $\text{tr}(\{A, C\}, b) = \{C\}$
- Quindi $P = \{\{A, C\}, \{B\}, \{D\}\}$
- A questo punto l'algoritmo si può fermare!!

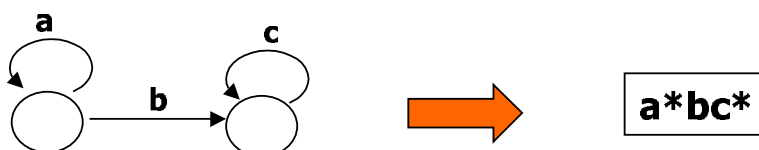
Da automa a espressione regolare

- Dato un automa è possibile generare l'espressione regolare che esso accetta
- Occorre rimuovere, ad uno ad uno, tutti gli stati dell'automa

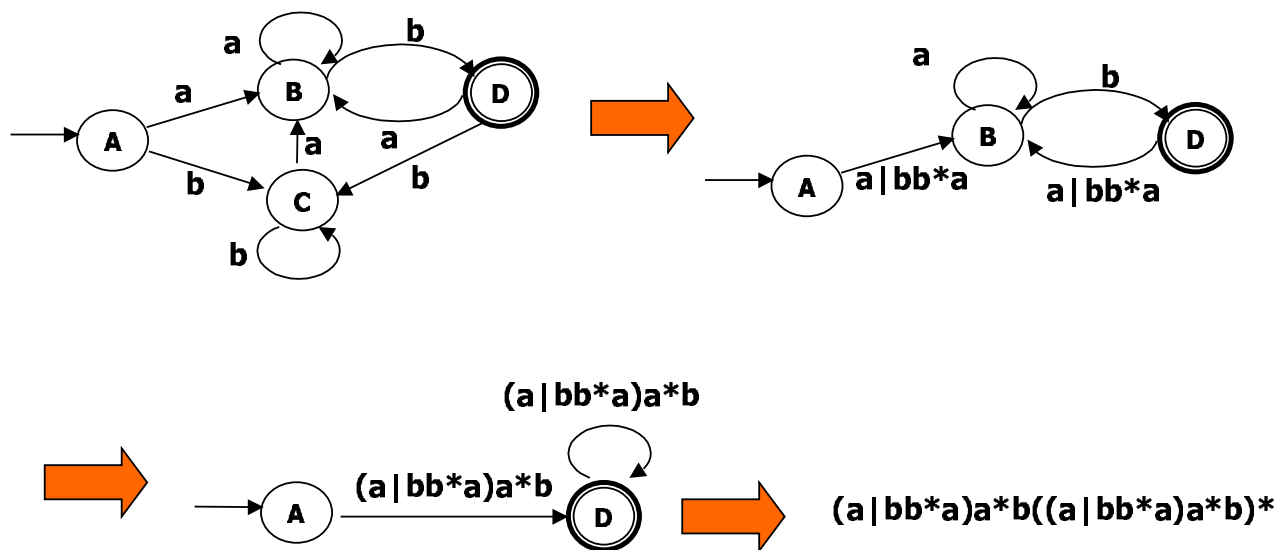


Da automa a espressione regolare II

- Si considerano **tutti** gli stati finali, **uno alla volta**
- Si **rimuovono gli stati** dell'automa fino quando non rimangono solo quello iniziale e quello finale scelto



Un esempio



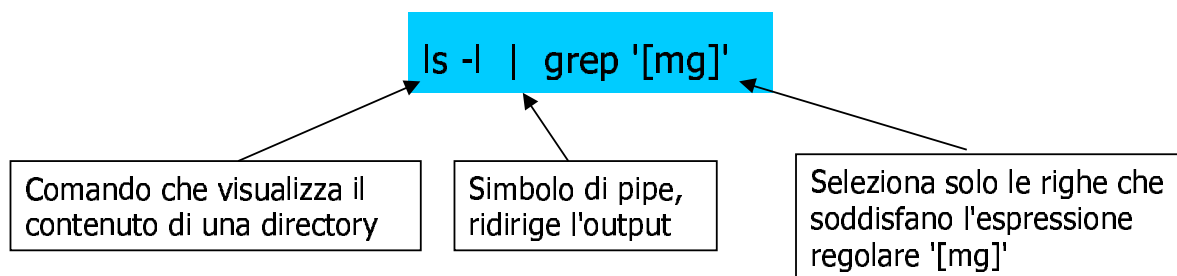
Dove si usano le espressioni regolari ?

- Le espressioni regolari sono usate per
 - generazione di **analizzatori lessicali**
 - comandi di ricerca **negli editor**
 - **funzioni di libreria** che implementano il matching con espressioni regolari
 - **comandi** per la ricerca di pattern
- Molti di questi programmi sono stati sviluppati originariamente in Unix

Esempio: il comando grep

- Il comando Unix **grep** permette di realizzare un filtro con cui selezionare alcune righe di un testo

Questo comando Unix mostra il contenuto di una directory selezionando solo alcune righe



Posix 1003.2

- Posix 1003.2 definisce uno standard con cui si specificano le espressioni regolari
- Operatori
 - `E?` (L'espressione E è opzionale)
 - `E*` (Chiusura di Kleene)
 - `E+` (L'espressione E è ripetuta almeno una volta).
 - `E{n}` (L'espressione E è ripetuta esattamente n volte)
 - `^E` (indica che E appare all'inizio di una riga)
 - `E$` (indica che E appare alla fine di una riga)

Posix 1003.2 II

■ Atomi

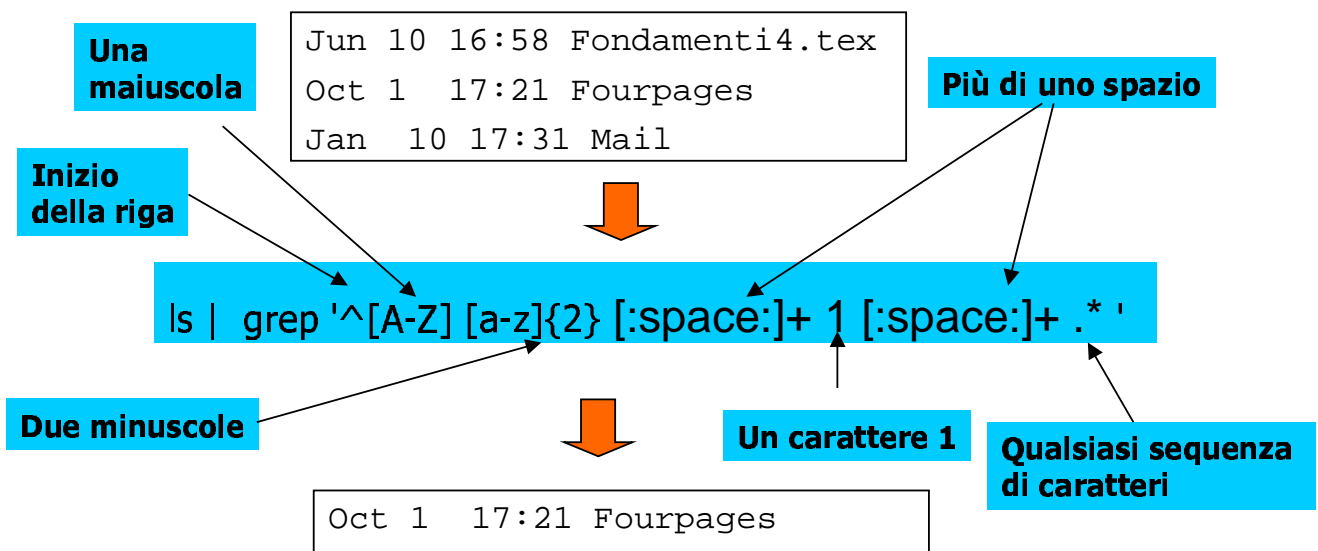
- . (il punto indica un qualsiasi carattere)
- [abcd] (qualsiasi carattere fra a, b, c, d)
- [0-9] (le cifre, il simbolo "-" specifica che si devono prendere tutti in caratteri compresi fra 0 e 9)
- [a-zA-Z] (tutte le lettere, maiuscole e minuscole)

■ Classi predefinite

- [:alpha:] (qualsiasi lettera)
- [:digit:] (qualsiasi cifra)
- [:space:] (qualsiasi carattere di spaziatura)

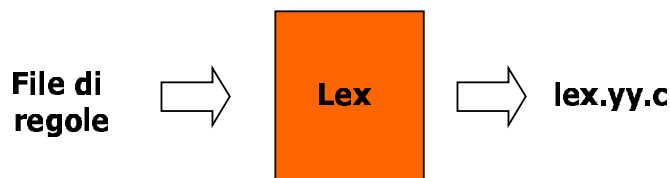
Un esempio

- Questo comando seleziona tutti i file modificati il primo del mese



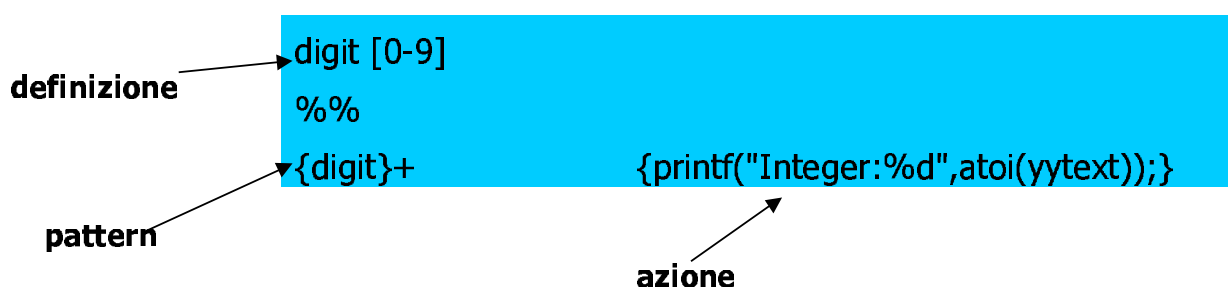
Generazione di analizzatori lessicali (Lex)

- In pratica, esistono strumenti che facilitano la realizzazione di analizzatori lessicali
- Lex è un generatore di analizzatori lessicali
- Lex prende in ingresso un file che contiene un insieme di regole e genera un file C che implementa un analizzatore lessicale



File di regole Lex

- Un file di regole Lex è costituito da una parte di **definizioni** e una di **regole**
- Le regole sono costituite da un **pattern** e un'**azione**
 - il pattern è un'espressione regolare
 - l'azione è codice C
 - l'analizzatore esegue il comando quando riconosce il pattern



Un esempio: riconoscere identificatori e parole chiave

Definizioni	digit [0-9]	
	id [a-z][a-z0-9]*	
inizio regole	%%	
interi	{digit}+	{printf("Integer:%d",atoi(yytext));}
reali	{digit}+"."{digit}+	{printf("Float:%g",atof(yytext));}
parole chiave	if then begin end	{printf("Keyword:%s",yytext);}
identificatori	{id}	{printf("Identifier:%s",yytext);}
operatori	"+" "-" "/" "*"	{printf("Operator:%s",yytext);}
rimuove gli spazi	[\t\n]	
caratteri sconosciuti	.	{printf("Unknown char.:%s",yytext);}