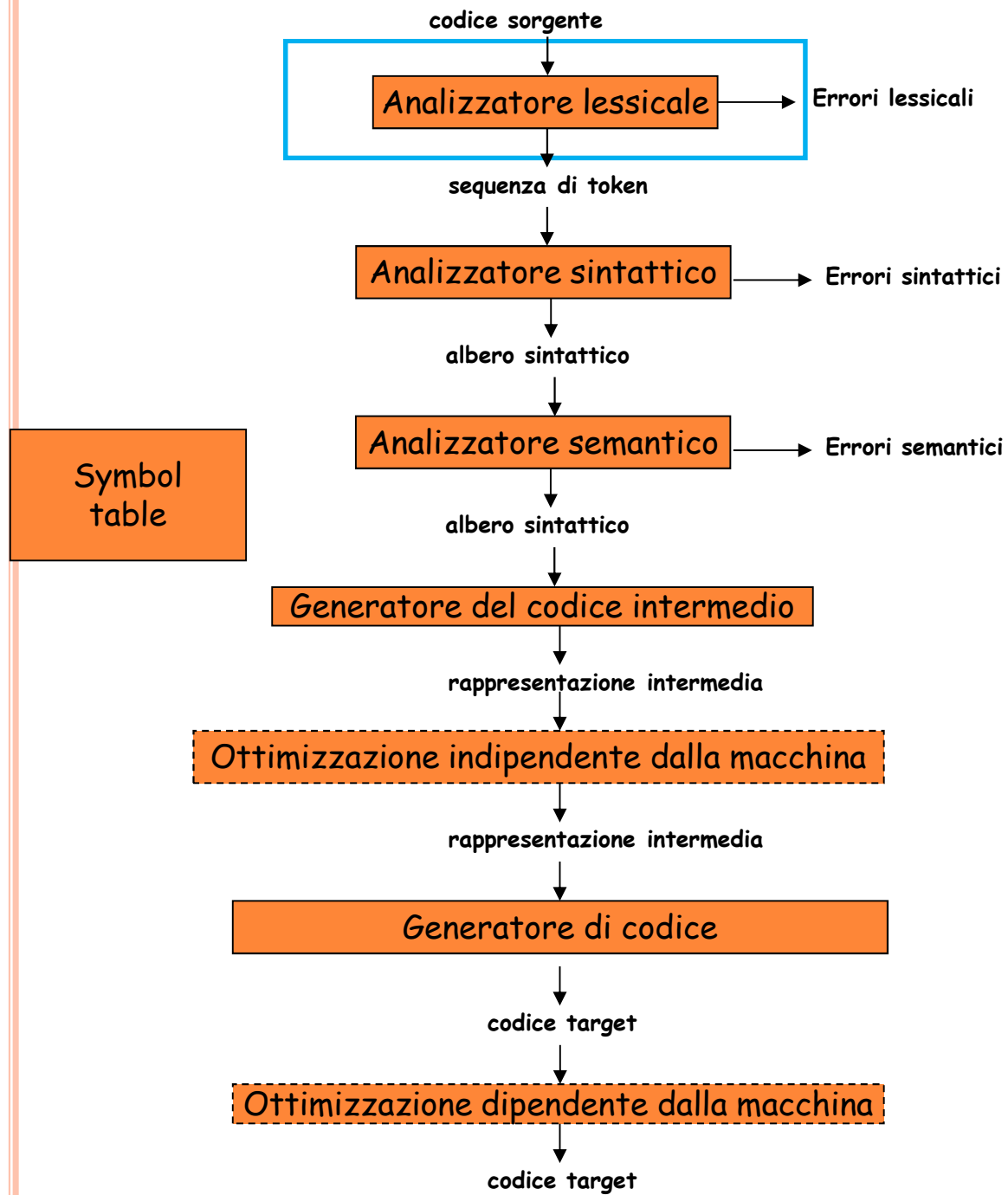


1

# *ANALISI LESSICALE*

Lunedì 7 Ottobre



# ANALISI LESSICALE

- E' una fase durante la quale l'analizzatore lessicale (scanner) scandisce la sequenza di caratteri che costituisce il codice sorgente (source code), li raggruppa in lessemi e produce una sequenza di **token** corrispondenti a tali lessemi.
- Viene spesso chiamata TOKENIZZAZIONE (Tokenizing).
- Per esempio, consideriamo la seguente linea di codice:

```
For index:=1 to N do a[index] := 4 + 2;
```

Questo codice contiene 30 caratteri non-blank ma invia 16 token al parser:

Keywords: For, to, do

Identificatori: index, a, N

Costanti: 1, 4, 2

Operatori: := , +

Punteggiatura: ;

Parentesi: [, ]

# OPERAZIONI PRELIMINARI

A meno che ciò non venga gestito da un precompilatore, lo scanner deve tener conto di:

**Rimuovere i commenti:** i commenti sono individuabili con simboli speciali. Lo scanner deve individuare tali simboli.

**Case Conversion:** Molti linguaggi di programmazione (ad es. Pascal) ignorano la capitalizzazione. Lo scanner deve convertire tutte le letter in uppercase, per esempio.

**Rimuovere gli spazi:** gli spazi bianchi (blank, tab, invio, ...) servono per separare certi tipi di token. Uno scanner deve solo riconoscere i token.

**Tenere traccia del numero di linea:** nel caso eventuali messaggi d'errore.

**Preparazione di un output listing:** Alcuni scanner creano una versione annotata del codice sorgente, contenente per es. il numero di linea, messaggi di errore o di warning, ...

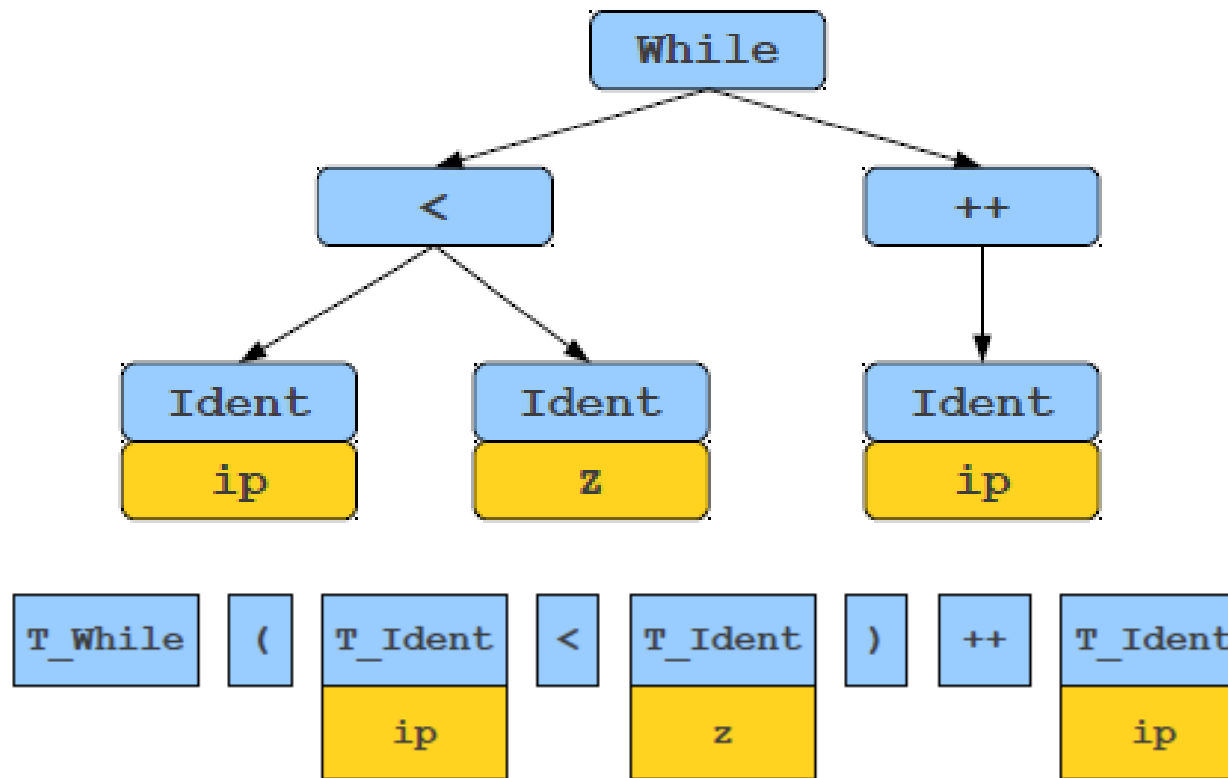
# SCANNING E ANALISI LESSICALE

Talvolta gli analizzatori lessicali applicano i seguenti due processi uno dietro l'altro:

**Scanning:** non richiede la tokenizzazione ma solo la rimozione dei commenti e la compattificazione degli spazi bianchi consecutivi in uno solo;

**Analisi lessicale:** lo scanner produce la sequenza di token.

## PRECEDE L'ANALISI SINTATTICA



Analisi  
sintattica

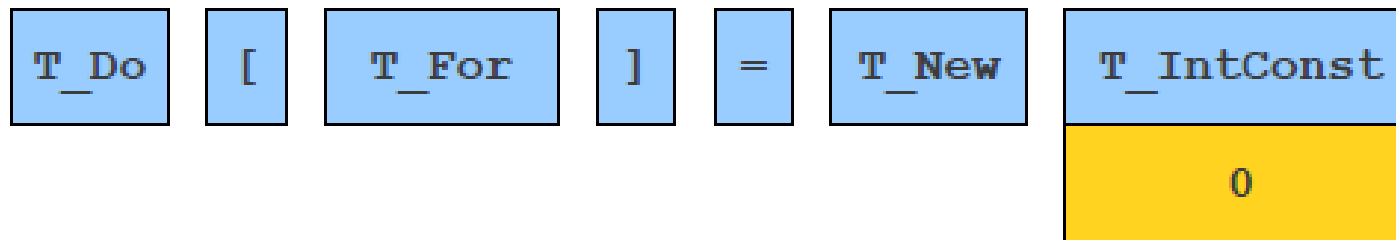
Analisi  
lessicale

```
while (ip < z)
    ++ip;
```

# NON RILEVA ERRORI SINTATTICI!



L'Analisi sintattica  
non andrà a buon fine!



Analisi  
lessicale

d	o	[	f	o	r	]		=		n	e	w		0	;
---	---	---	---	---	---	---	--	---	--	---	---	---	--	---	---

do[for] = new 0;

# TOKEN

**Token:** una coppia costituita da un nome del token e da un attributo opzionale; il **nome del token** è un simbolo astratto che rappresenta un tipo di unità lessicale o unità logica di informazione nel programma sorgente. Per esempio:

- le parole chiave di un linguaggio (keywords) come per esempio: if, while, do, then, else, ...
- gli identificatori: stringhe definite dall'utente costituite da lettere e numeri che iniziano con una lettera.
- Numeri
- stringhe di caratteri: sequenze di caratteri comprese tra virgolette
- operatori: simboli come \*, +, ... o simboli multicarattere come >=, <=, <>, ...
- simboli speciali: parentesi, ., :, ...

(Spesso quando si parla di token si intende il nome del token)



## LESSEMA

Sequenza di caratteri nel programma sorgente identificati dallo scanner, ovvero una specifica istanza di un token.

# PATTERN

Descrizione compatta della forma che il lessema di un token può assumere.

Nel caso delle keyword, il pattern è proprio la sequenza dei caratteri della keyword;

Nel caso di identificatori o altri token, il pattern è una struttura in grado di matchare con molte stringhe.

## TOKEN, PATTERN, LESSEMI

TOKEN	PATTERN	LESSEMI
T_if	Caratteri i,f	if
T_else	Caratteri e,l,s,e	else
T_Op_confronto	< or > or <= or >= or == or !=	<=, !=
T_Identificatore	Lettera seguita da lettera o cifra	pi, rate, d3
T_Costante	Qualsiasi costante numerica	3.14, 67, 5.02e23
T_Stringa	Qualsiasi sequenza di caratteri diversi da “, circondati da “	“ciao ciao”

# TOKEN E LESSEMI A CONFRONTO

```
if dist>=rate*(end - start) then dist:=maxdist;
```

<u>Token</u>	<u>Lessemi</u>
id	dist, rate, end, start, maxdist
relop	>=
(	(
)	)
if	if
then	then
:=	:=
*	*
-	-
;	;

L'analisi lessicale produce la seguente sequenza di simboli terminali per il parser, ciascuno dotato di un eventuale attributo:

```
if id relop id*(id-id) then id:=id;
```

# ANALISI LESSICALE DI UN FILE SORGENTE

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;
---	---	---	---	---	--	---	---	---	---	--	---	--	---	---	----	----	---	---	---	---

Lessema

Lessema

T\_While

(

Token

T\_IntConst

137

Token

Attribu  
to

# E' INDISPENSABILE L'ANALISI LESSICALE?

Si potrebbe scaricare il ruolo di analizzatore lessicale sul parser?

I vantaggi dell'uso dello scanner sono:

- ciascuna delle due fasi viene semplificata (anche gli spazi bianchi diventerebbero simboli terminali della grammatica complicandola ulteriormente;
- miglioramento dell'efficienza (esistono tecniche specializzate per l'analisi lessicale);

# ATTRIBUTI

Particolari informazioni relative ad uno specifico lessema di un particolare token.

E' necessario quando più lessemi possono matchare con un pattern. E' quindi importante identificare in qualche modo il lessema.

Il nome del token influenza le decisioni di parsing, l'attributo la traduzione del token dopo il parsing.

Esempio:

$$E=M*C^2$$

<id, punt. nella symbol table alla entry relativa a E>

<op\_ass>

<id, punt. nella symbol table alla entry relativa a M>

<op\_molt>

<id, punt. nella symbol table alla entry relativa a C>

<op\_exp>

<num, valore intero 2>

# ESEMPIO

SYMBOL TABLE

1	final	.....
2	initial	.....
3	rate	.....
4	.....	.....
5	.....	.....

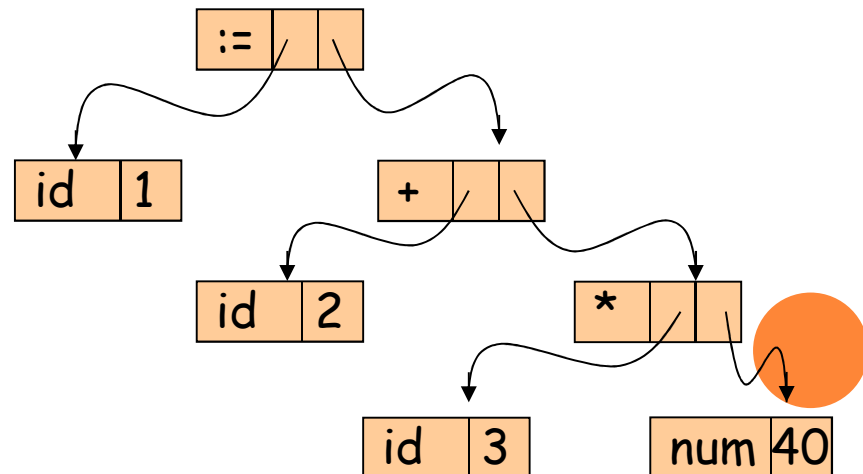
Token:  
<id, puntatore alla ST>  
<:=, >  
<+, >  
<\*, >  
<num, 40>  
...

final:= initial + rate \*40

Analisi lessicale

id:= id + id \*num

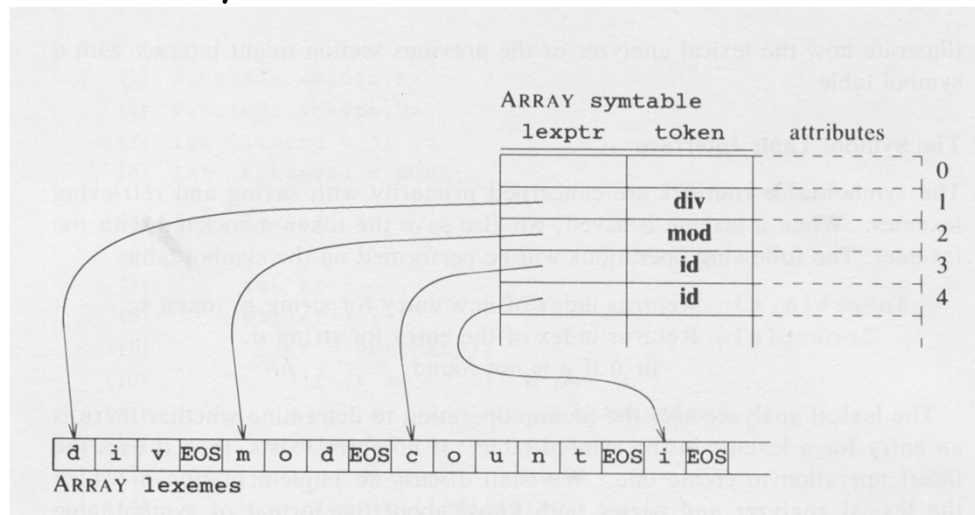
Analisi sintattica e semantica





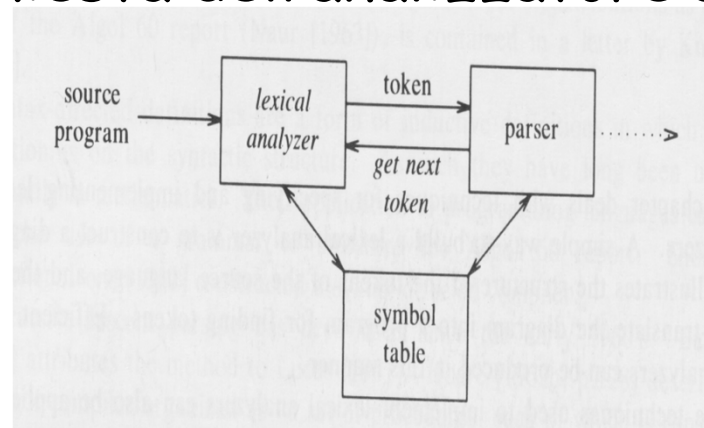
# IMPLEMENTAZIONI DELLA SYMBOL TABLE

- Spesso non è opportuno assegnare uno spazio fissato ad un lessema (troppo per alcuni lessemi, troppo poco per altri)
- Un array di lessemi separato mantiene le stringhe che formano gli identificatori
- Lo scanner comunica con la ST con operazioni del tipo  
**insert(s,t)**: restituisce l'indice di una nuova entry per la stringa s, token t  
**lookup(s)**: restituisce l'indice della entry per s, 0 se non la trova. Può riconoscere le keywords



## RUOLO DELLO SCANNER

- ◉ I token prodotti dallo scanner saranno usati dal parser per l'analisi sintattica.
- ◉ Questa interazione, descritta in figura, è comunemente implementata rendendo lo scanner una procedura o routine che legge un carattere per volta finchè non identifica un token. Tale procedura agisce su richiesta dell'analizzatore sintattico.



## ERRORI LESSICALI

- ◉ Un analizzatore lessicale ha una visione molto localizzata del codice sorgente, quindi è in grado di riconoscere pochi errori.
- ◉ Per esempio dato il codice `fi a=f(x) then ...`, in genere uno scanner non può sapere se `fi` è un misspelling di `if`, oppure se è un identificatore non dichiarato.
- ◉ Se uno scanner non riconosce una sequenza di caratteri come token può evidenziare l'errore oppure potrebbe essere dotato di strategie di recupero dell'errore:
  - > Cancellare caratteri successivi dal resto dell'input finché non riconosce un token;
  - > Cancellare, inserire, sostituire un carattere nel resto dell'input;
  - > Effettuare correzioni locali sulla base di un calcolo del criterio minimum-distance (è un metodo costoso da implementare poiché consiste nel trovare il più piccolo numero di trasformazioni necessarie per ottenere un lessema valido.).

# COME IMPLEMENTARE UNO SCANNER

Esistono 3 approcci:

- ◉ Usare un generatore automatico di analizzatori lessicali, come Lex, Flex, .... Il generatore fornisce anche le routine per la lettura dell'input.
- ◉ Scrivere l'analizzatore lessicale come un programma in un linguaggio di programmazione convenzionale. L'input si gestisce sfruttando le caratteristiche del linguaggio.
- ◉ Scrivere l'analizzatore lessicale come un programma in linguaggio assembly. In tal caso l'input deve essere gestito a basso livello.

## COME DESCRIVERE I TOKEN?

- ◉ Come definire le regole lessicali per un linguaggio di programmazione?
  - > Attraverso le espressioni regolari!
- ◉ Le espressioni regolari sono un'importante strumento per specificare i pattern. Rappresentano un modo compatto di denotare quali caratteri possono costituire un lessema che appartiene ad una certa classe di token.
- ◉ Un analizzatore lessicale è uno strumento in grado di riconoscere i pattern, ovvero un automa a stati finiti

# I LINGUAGGI FORMALI: RICHIAMIAMO ALCUNE DEFINIZIONI

**Alfabeto:** Insieme finito di simboli

**Stringa o parola:** sequenza finita di simboli dell'alfabeto

**Lunghezza di una parola:** numero dei caratteri che la compongono

**Parola vuota:** parola di lunghezza 0 (denotata con  $\epsilon$ )

**Linguaggio:** insieme finito o infinito di parole su un dato alfabeto

Operazioni sui linguaggi:

$L \cup M$	unione	$\{s \mid s \in L \text{ o } s \in M\}$
$LM$	concatenazione	$\{st \mid s \in L \text{ e } t \in M\}$
$L^*$	chiusura di Kleene	zero o più concatenazioni di L
$L^+$	chiusura positiva	una o più concatenazioni di L

# NOTAZIONI COMPATTE PER DESCRIVERE LINGUAGGI

Una **espressione regolare** è una delle seguenti:

- l'espressione  $\epsilon$  denota il linguaggio  $L=\{\epsilon\}$
- l'espressione  $\phi$  denota il linguaggio vuoto
- l'espressione  $a$  denota il linguaggio  $L=\{a\}$
- se  $r$  ed  $s$  sono espressioni regolari che denotano i linguaggi  $L(r)$  e  $L(s)$ ,
  - $r|s$  denota  $L(r) \cup L(s)$
  - $rs$  denota  $L(r)L(s)$
  - $r^*$  denota  $L(r)^*$
  - $(r)$  denota  $L(r)$

Regole di precedenza: operatore  $*$ , poi concatenazione e infine  $|$   
Tali operazioni sono associative a sinistra.

I linguaggi definiti mediante espressioni regolari sono detti **linguaggi regolari**.

Esempio:  
Identificatori=lettera(lettera|cifra)\*

## ESEMPI

- $bbb$  è un'espressione regolare che denota ...
  - >  $\{bbb\}$ .
- $ab|bbb$  è un'espressione regolare che denota ...
  - >  $\{ab, bbb\}$ .
- $ba^*$  è un'espressione regolare che denota ...
  - >  $\{b, ba, baa, baaa, \dots\}$ .
- $b(a|b)^*b|b$  è un'espressione regolare che denota ...
  - >  $\{b$  e le stringhe che cominciano e finiscono per  $b\}$ .
- $(a|b)(a|b)$  è un'espressione regolare che denota ...
  - >  $\{aa, ab, ba, bb\}$ .



## ESERCIZI

- RE per stringhe di lunghezza pari su alfabeto  $\{a,b\}$
- RE per stringhe con esattamente una "b"
- RE per gli INTEGER (possibilmente preceduti da '+' or '-', che non cominciano per 0; l'alfabeto è  $\{0,...,9,+,-\}$ )
- RE per gli identificatori del linguaggio Pascal

## NOTAZIONE ESTESA

- ◉ **Una o più ripetizioni:** data un'espressione regolare  $r$ , denotiamo con  $r^+$  una o più concatenazioni di  $r$ ;
- ◉ **Zero o una istanza:** denotiamo con  $r?$  l'espressione  $r|\epsilon$ ;
- ◉ **Carattere jolly:** denotiamo con "." un'espressione che individua un qualsiasi carattere dell'alfabeto. Per es.  $.*b.*$
- ◉ **Range di caratteri:**  $[a-z]$  denota l'espressione  $a|b|\dots|z$   
 $[a-zA-Z]$  denota un range multiplo
- ◉ **Carattere non presente in un dato insieme:**  
 $[\hat{a}bc]$  denota un carattere diverso da  $a$ ,  $b$  e  $c$ ;  
 $[\hat{a}]$  denota un carattere diverso da  $a$

# DEFINIZIONI REGOLARI

Sia  $A$  l'alfabeto dei simboli.

Una definizione regolare è una sequenza di definizioni della forma:

$$d_1 \longrightarrow r_1$$
$$d_2 \longrightarrow r_2$$
$$d_3 \longrightarrow r_3$$

....

$$d_n \longrightarrow r_n$$

dove ogni  $d_i$  ha un nome distinto e ogni  $r_i$  è un'espressione regolare sull'alfabeto  $A \cup \{d_1, d_2, d_3, \dots, d_{i-1}\}$

# CON QUESTE NOTAZIONI DEFINIAMO...

I numeri naturali, interi, reali in notazione esponenziale

`nat = [0-9]+`

`signedNat = (+|-)? nat`

`number = signedNat("." nat)? (E signedNat)?`

**Keywords**

`reserved = if | while | do | ...`

**Identificatore**

`letter_=[a-zA-Z_]`

`digit=[0-9]`

`id = letter_(letter_|digit)*`

**Commento in Pascal**

`{[^}]*}`

# AMBIGUITÀ

- Un lessico può essere definito mediante espressioni regolari multiple
  - Ciò può condurre ad ambiguità
- Per l'input "begin", la RE  $[a-z]^+$  individua
  - Ogni prefisso: "b", "be", "beg", etc.
  - Quali token scegliere?
- Per l'input "begin", entrambe le due RE vanno bene
  - begin
  - $[a-z]^+$

Quali espressioni regolari applicare?

## ELIMINIAMO L'AMBIGUITÀ

### ○ Longest Match

- Viene considerata come lessema la più lunga stringa che ha un match con un'espressione regolare.

Esempio

[a-z]+ prosegue la ricerca del match con caratteri minuscoli, non si ferma al primo match.

### ○ Regole di priorità

- Se due espressioni regolari hanno entrambe un match, allora la prima espressione regolare è quella che determina il match e quindi il tipo di token.

Esempio

- "if" è tokenizzato come IF, non come ID.

## ESISTONO LINGUAGGI NON REGOLARI

- Il linguaggio  $S=\{ab, aabb, aaabbb, \dots\}$  non può essere definito mediante un'espressione regolare.
- Le espressioni regolari possono essere usate per denotare solo un numero fissato di ripetizioni o un numero non specificato di ripetizioni di un dato costrutto. Non possono essere usate invece per denotare costrutti bilanciati o annidati.
- Uno strumento utile per mostrare che un dato linguaggio non è regolare è il Pumping Lemma.

## COME RICONOSCERE I TOKEN?

- Nel 1956 Kleene ha dimostrato l'equivalenza tra le espressioni regolari e gli automi a stati finiti deterministici.
- Gli automi a stati finiti sono un strumento efficace per riconoscere i token.
- Un analizzatore lessicale è uno strumento in grado di riconoscere i pattern (espressioni regolari), ovvero un **automa a stati finiti**



## ESEMPI FAMILIARI

Supponiamo di considerare un linguaggio di programmazione la cui sintassi è schematizzata con la seguente grammatica. I simboli in grassetto sono i terminali.

Ad es:

stmt  $\rightarrow$  **if** expr **then** stmt  
          | **if** expr **then** stmt **else** stmt  
          |  $\epsilon$

expr  $\rightarrow$  term **relop** term  
          | term

term  $\rightarrow$  **id**  
          | **number**

## DEFINIZIONE DEI TOKEN

I simboli terminali della grammatica diventano i token

Ad es:

**nat** =  $[0-9]^+$

**signedNat** =  $(+|-)?$  nat

**number** = **signedNat**  $( "."$  nat)? (E **signedNat**)?

**letter\_** =  $[a-zA-Z\_]$

**id** = **letter\_** (**letter\_** | nat)\*

**if** = if

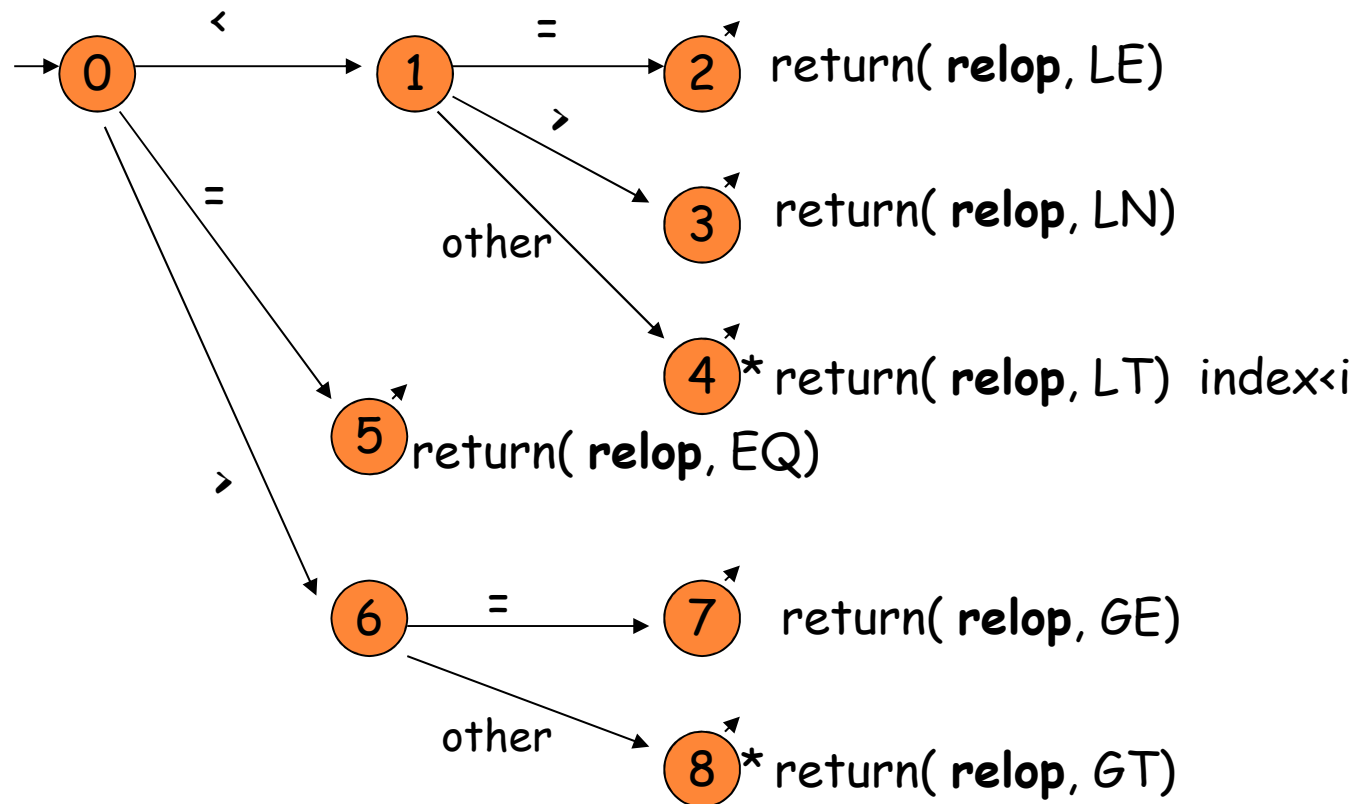
**then** = then

**else** = else

**relop** =  $<|>|<=|>=|=|<>$

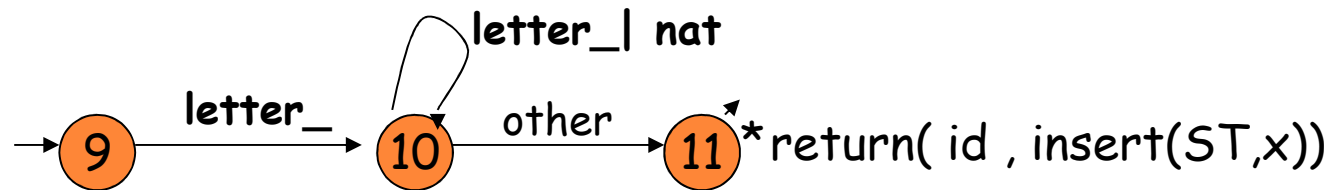
**ws** = (blank | tab | newline)+

# COME RICONOSCERE IL TOKEN RELOP?



Il simbolo \* indica che il puntatore di lettura deve ritrarsi di una posizione.  
Ad ogni stato finale è associata un'azione.

## RICONOSCIMENTO DI KEYWORD E IDENTIFICATORI

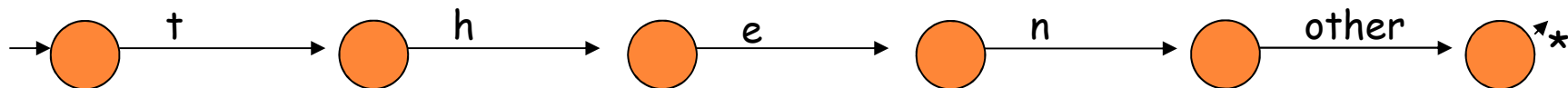


La funzione insert inserisce il token nella ST se non è già presente e restituisce il puntatore alla locazione.

Per le keywords:

O si inseriscono nella ST inizialmente e quindi li riconosciamo durante la chiamata della funzione insert

Altrimenti:



Bisogna fare in modo di dare priorità al riconoscimento delle keyword.

Esercizio: creare l'automa per **number**

# COME GESTIRE L'INPUT?

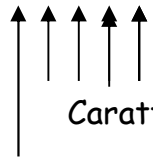
- In teoria lo scanner analizza la stringa sorgente un carattere per volta. In pratica deve essere in grado di accedere a sottostringhe della sorgente ovvero di tornare indietro di un blocco di caratteri, prima di poter annunciare un match.
  - Un identificatore viene riconosciuto solo dopo che si incontra un carattere diverso da numeri e lettere.
  - Gli operatori -, =, < sono prefissi di -, ==, <=.
- Visto l'elevato numero di caratteri e il tempo necessario ad elaborarli, non sarebbe conveniente invocare comandi di lettura per ogni carattere di input e tornare eventualmente indietro. Normalmente può essere più conveniente leggere N caratteri di input con un unico comando di lettura del sistema. Quindi, la stringa sorgente è letta attraverso un'area buffer cosicché lo scanner possa anche tornare indietro più facilmente.
- Normalmente si usa una tecnica chiamata "double buffering", ovvero due buffer ciascuno di dimensione N che vengono riempiti alternativamente. In genere N è scelto pari alla dimensione del blocco del disco (per es. 4K)

## PER ESEMPIO...

```
If dist >= rate * (end - start) then dist := max;  
For i := 1 to r do  
    rate := dist * i;
```



```
If dist >= rate * (end - start) then dist := max; For
```



Carattere corrente

Inizio del lessema



```
r i := 1 to r do rate := dist * i; (end - start) then dist := max; For
```

## QUESTIONI SUGLI AUTOMI

- Come costruirli?
- Come implementarli?

# AUTOMI A STATI FINITI

Un Automa Finito Deterministico (DFA) è una quintupla  
 $(Q, A, \delta, q_0, F)$

- $Q$  è l'insieme di stati
- $A$  è un insieme di simboli (alfabeto di input)
- $\delta$  è la funzione di transizione o di stato dell'automa ed associa alle coppie stato-simbolo uno stato  $\delta: Q \times A \rightarrow Q$
- $q_0$  è uno stato particolare detto lo stato iniziale dell'automa
- $F$  è un insieme di stati detti stati finali (o di accettazione) dell'automa.

Funzione di stato su una stringa  $x$

- $\delta'(q, \varepsilon) = q$  per ogni  $q$  in  $Q$  ;  $\delta'(q, xa) = \delta(\delta'(q, x), a)$
- $\delta'(q, x) =$  stato in cui si trova l'automa dopo aver letto tutti caratteri di  $x$ .

Linguaggio accettato dall' automa:  $L = \{x \in S^* : \delta'(q_0, x) \in F\}$



## IMPLEMENTAZIONE DI UN DFA

- Mediante il diagramma di transizione
  - Può essere facilmente implementato con un programma (si usa una variabile che tiene conto dello stato in cui ci si trova)
- Mediante la matrice di transizione (le righe sono gli stati, le colonne i simboli; ogni cella contiene lo stato di arrivo corrispondente)
  - Può essere implementato in modo semplice con un programma
  - La taglia del codice è ridotta
  - È facile da cambiare
  - Lo svantaggio è che le tabelle possono diventare molto grandi

## SIMULARE UN DFA CON DIAGRAMMI DI TRANSIZIONE

....

```
switch (state) {  
    case 0: c=nextchar();  
        if (c=='<') state =1;  
        else if (c=='>') state =2;  
        else ...
```

```
    case 1: ....
```

....

```
    case 5: return ('yes');  
}
```

## SIMULARE UN DFA CON MATRICE DI TRANSIZIONE

```
state=s0
c=nextchar();
while (c!=eof) {
    state=tab(state,c);
    c=nextchar();
}
if (state in F) return 'yes';
else return 'no';
```

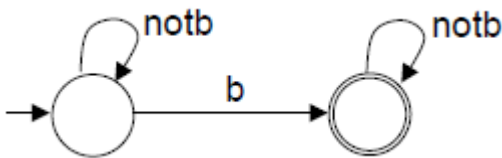
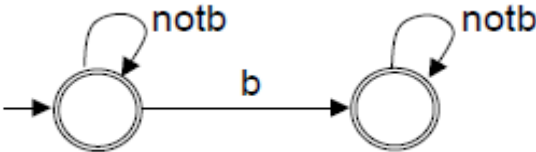
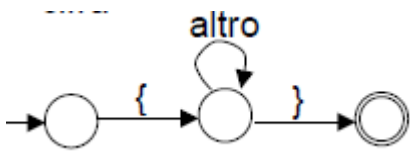
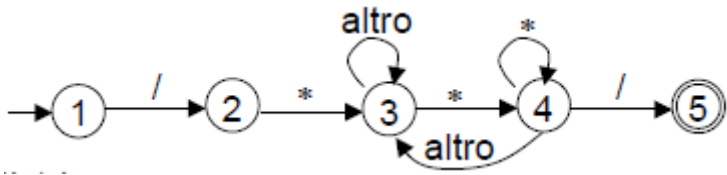
## IL PROBLEMA È UN PO' PIÙ COMPLESSO...

- I DFA sono un modo per rappresentare algoritmi che riconoscono stringhe in accordo con certi pattern.
- I diagrammi o le tabelle non descrivono tutti gli aspetti del comportamento di un algoritmo DFA. Per esempio
  - non descrivono cosa succede in presenza di errore.
  - non descrivono cosa succede in caso di arrivo su uno stato di accettazione.
  - dovrebbero tener conto anche delle operazioni di Lookahead e Backtracking, ovvero applicare le regole per eliminare le ambiguità.
- Bisogna arricchirli con le azioni corrispondenti e/o con le transizioni che definiscono errori.

## COME COSTRUIRE L'AUTOMA?

- Definire le espressioni regolari che definiscono i token.
- Trovare il DFA corrispondente (teorema di Kleene)

## A VOLTE PUÒ ESSERE IMMEDIATO

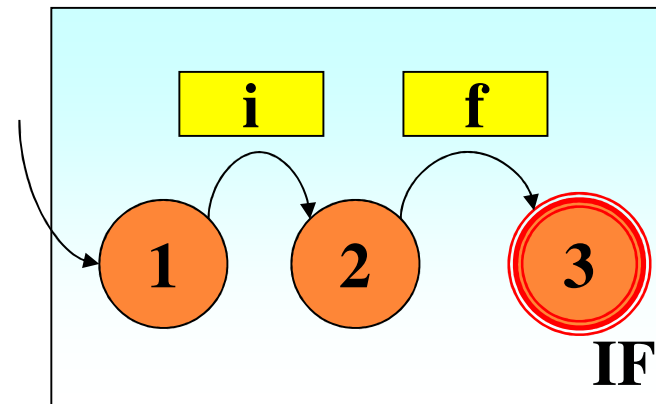
- $L = \{w \mid w \text{ contiene esattamente una "b"}\}$  
- $L = \{w \mid w \text{ contiene al più una "b"}\}$  
- $L = \{w \mid w \text{ è un commento in Pascal}\}$  
- $L = \{w \mid w \text{ è un commento in C}\}$  

## COSA SUCCEDDE PER I NOSTRI TOKEN?

Espressione regolare

**i f**

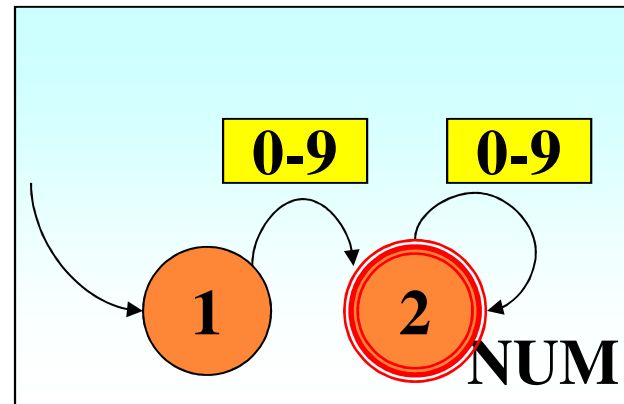
DFA



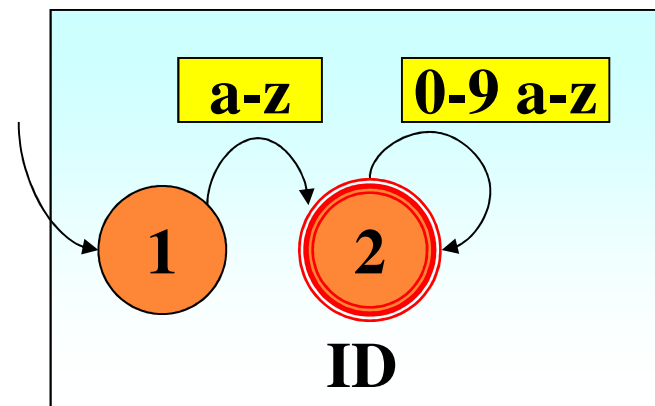
Scrivere il codice corrispondente al diagramma di transizione è molto semplice.

SI ESEGUONO “IN PARALLELO” O SI  
COMPONGONO?

**[0-9][0-9]\***

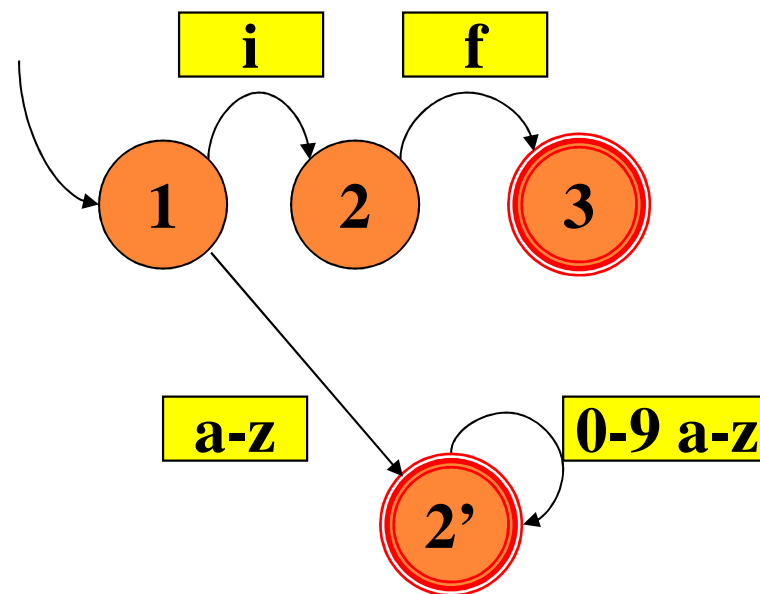
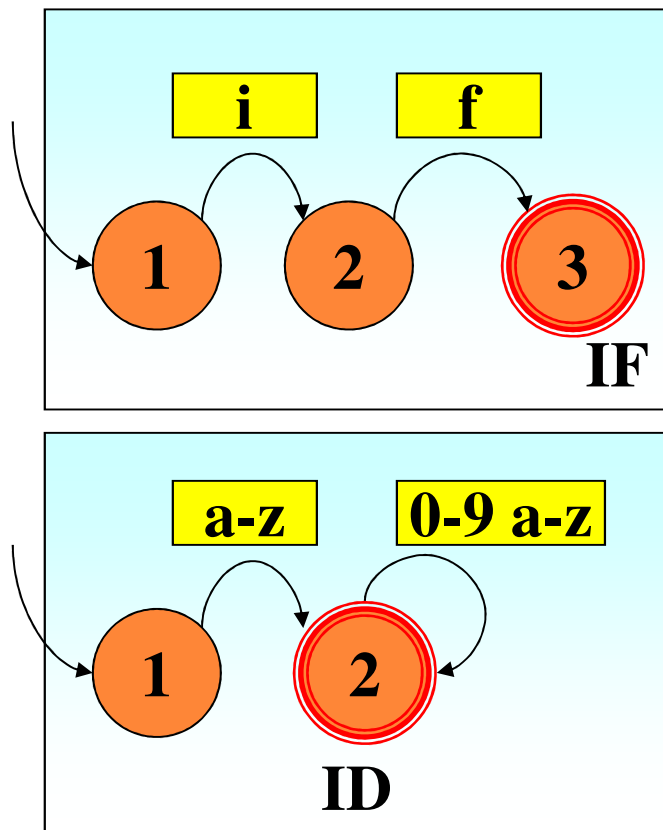


**[a-z][0-9a-z]\***

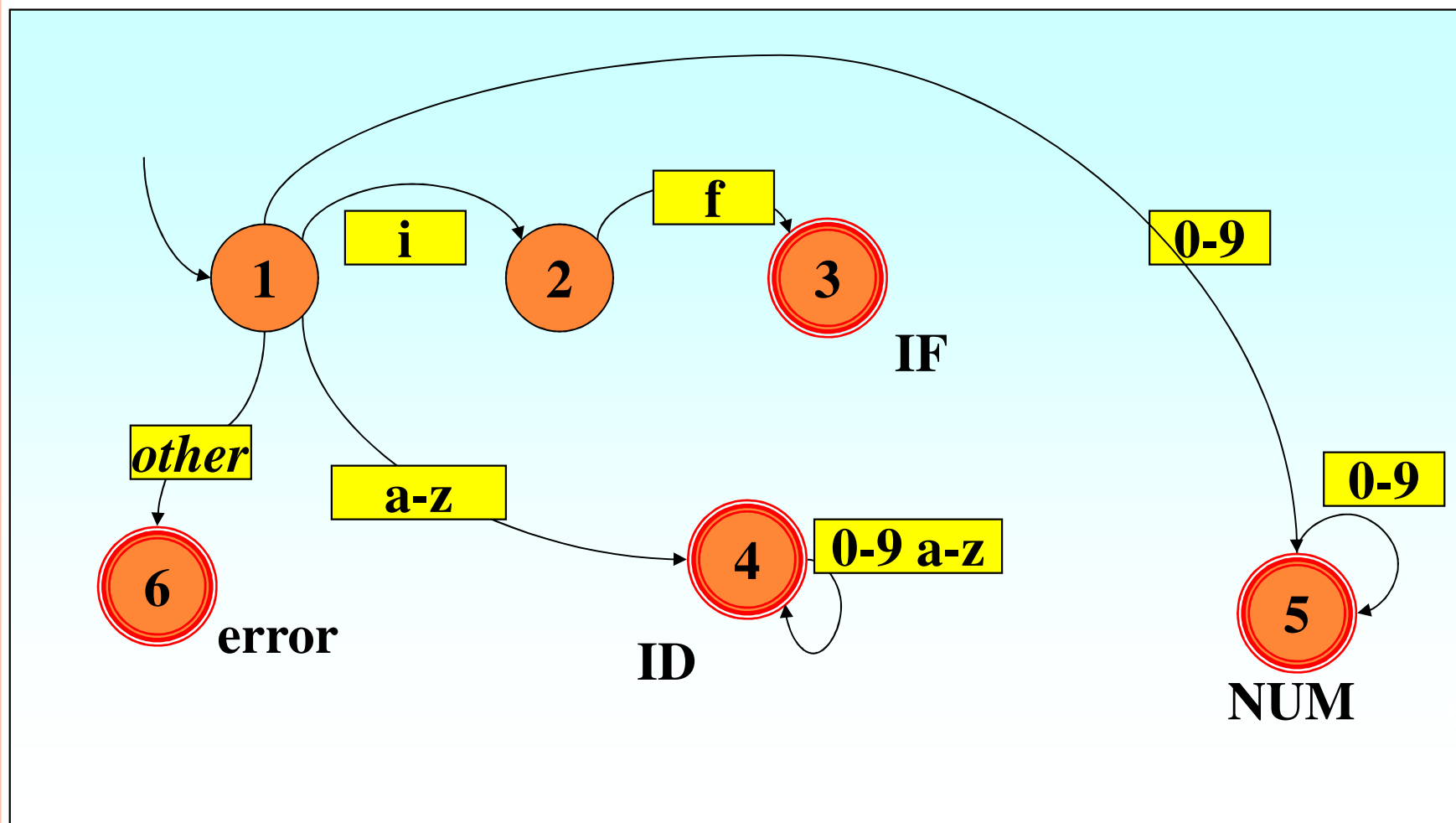




## Comporre DFA non è così immediato



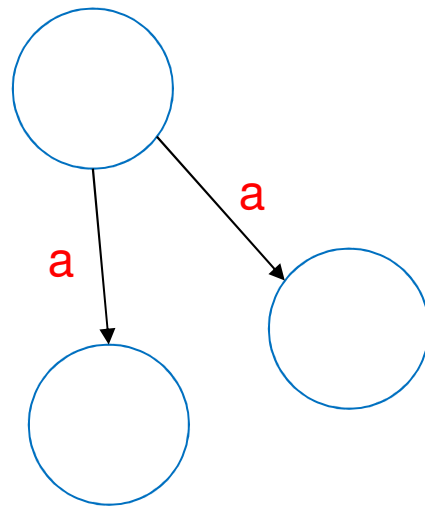
Si ottengono automi finiti non deterministici



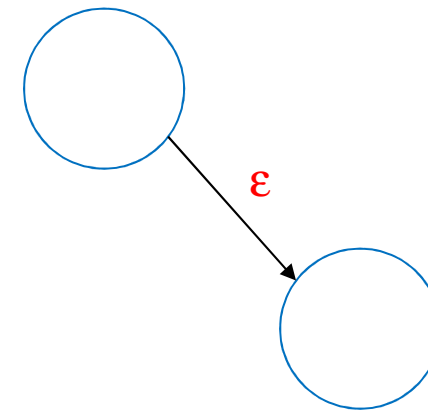
# Automi a stati finiti non-deterministici (NFA)

...sono quasi come i DFA, tranne che

Hanno più di una transizione per input



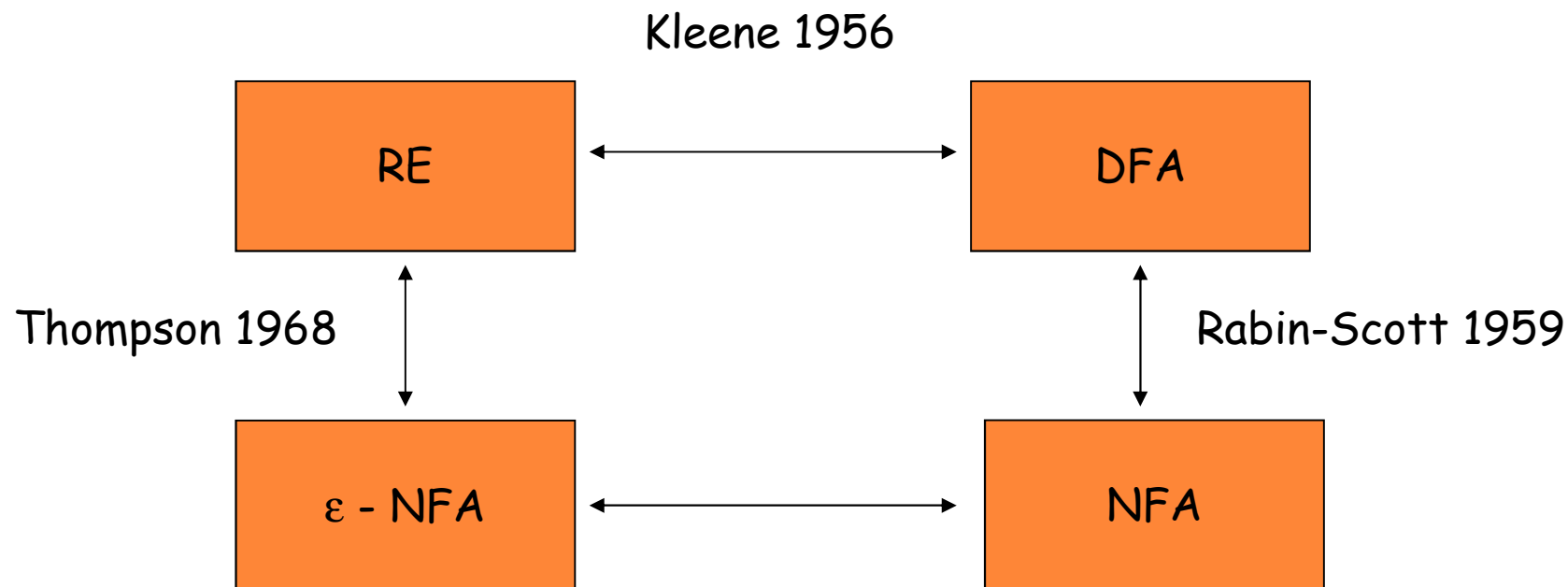
$\epsilon$  transizioni



# Automati non deterministici vs deterministici

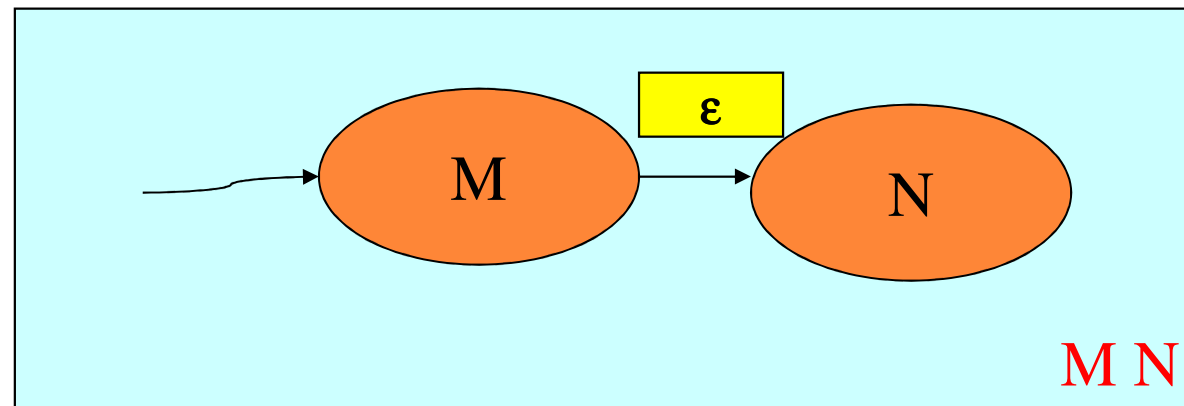
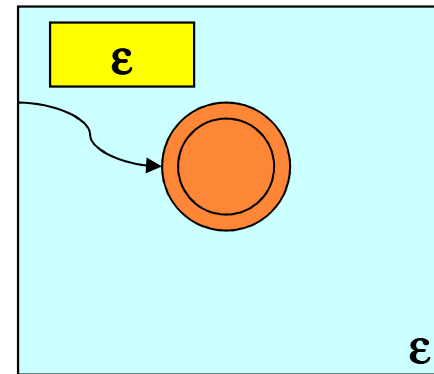
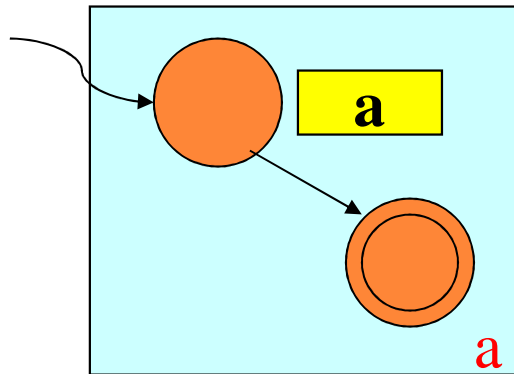
- Automi non-deterministici
  - Più semplici da creare.
  - Difficili da eseguire in modo efficiente.
- Automi deterministici
  - Non hanno  $\epsilon$  transizioni.
  - Da uno stesso stato non escono due archi con la stessa etichetta.
  - Sono più facili da implementare.
- E' possibile creare l'automa non deterministico e poi passare al deterministico.

# ESPRESSIONI REGOLARI ED AUTOMI

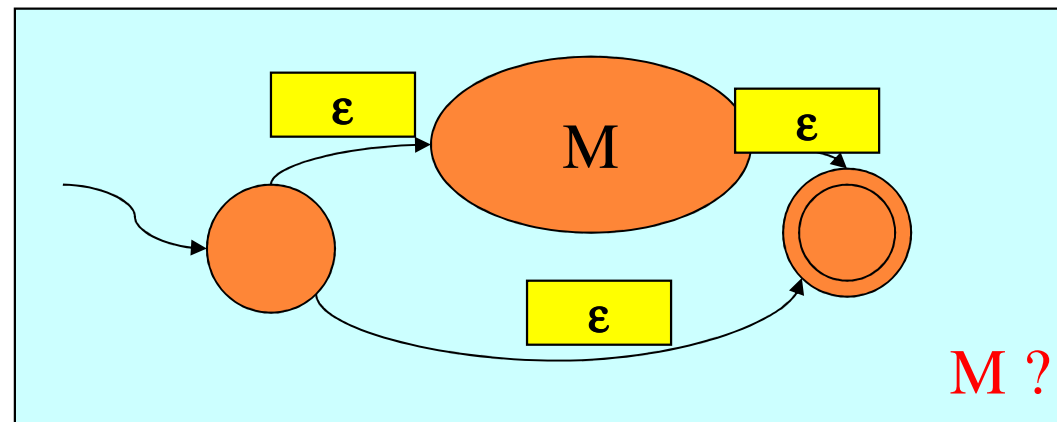
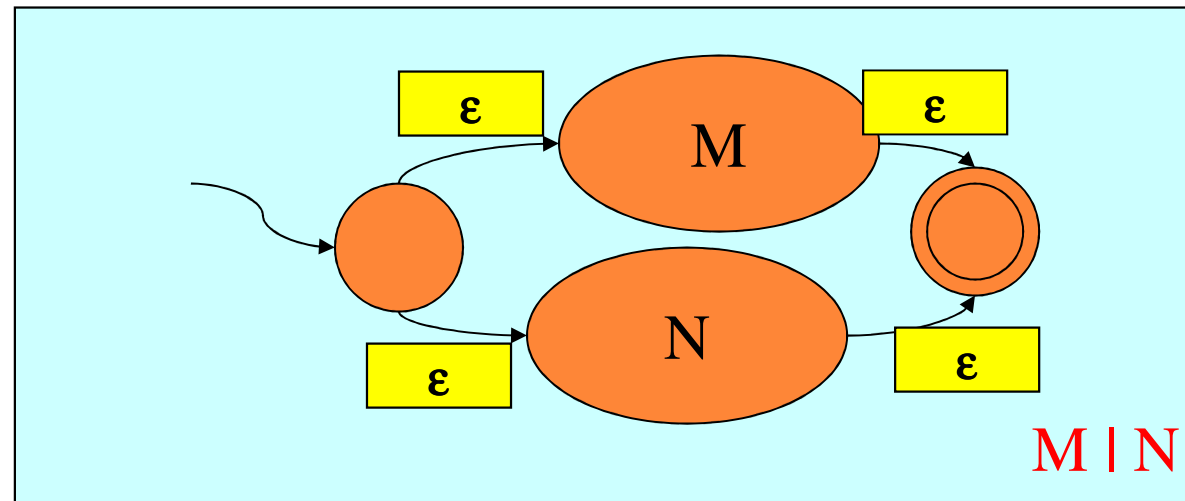


Ancora prima dello sviluppo di UNIX, Ken Thompson aveva studiato l'impiego delle espressioni regolari in comandi come grep - Global (search for) regular expression and print

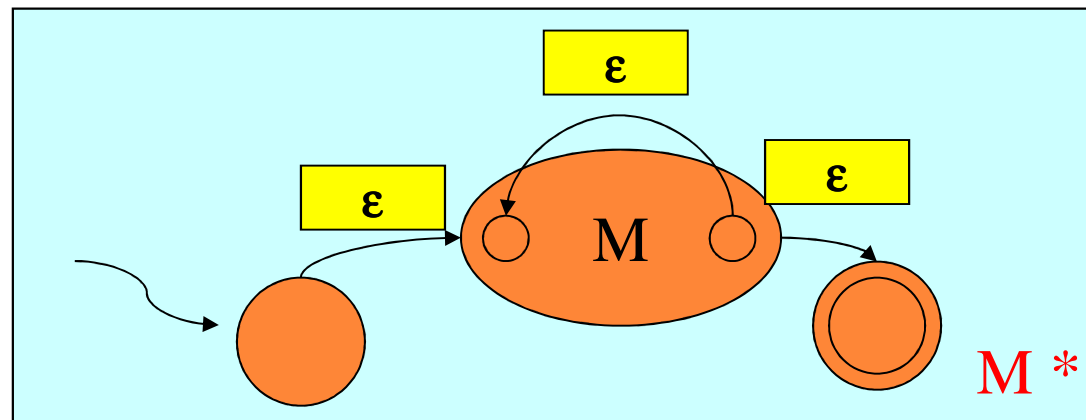
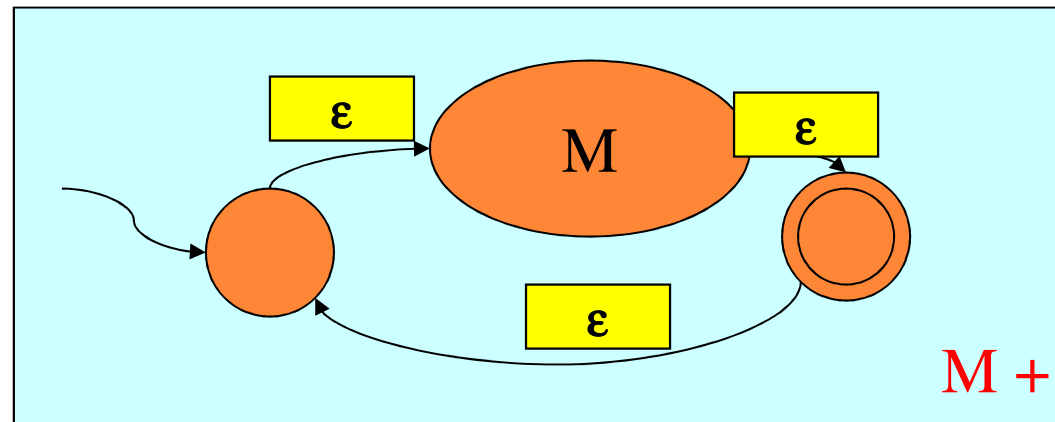
## DA ESPRESSIONE REGOLARE A NFA



## UNIONE E OPERAZIONE ?



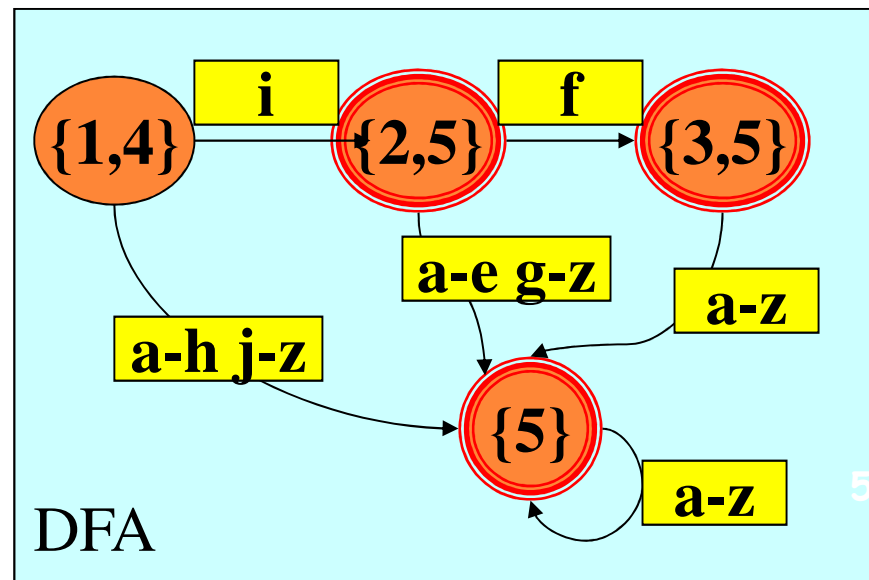
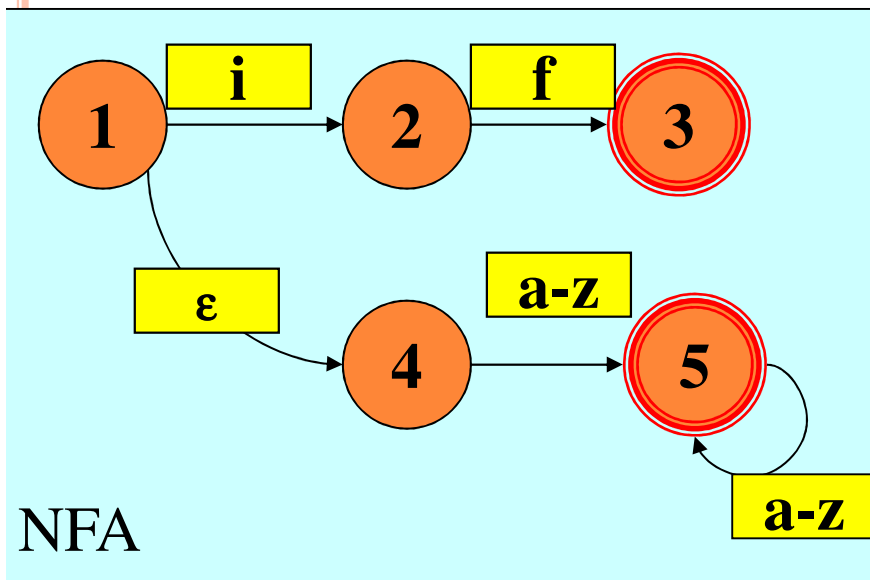
## OPERAZIONE \*





## DA NFA A DFA

- Ogni stato in DFA corrisponde ad un insieme di stati in NFA.
- Può produrre un DFA con  $2^N$  stati
- Esistono algoritmi efficienti per trasformare un NFA in DFA.
  - I generatori di analizzatori lessicali li usano!



# L'ALGORITMO "SUBSET CONSTRUCTION"

$$\begin{array}{l} \text{NFA} \quad \text{DFA} \\ M \rightarrow M' \end{array} \left\{ \begin{array}{l} M = (\Sigma, S, T, s_0, F) \\ M' = (\Sigma, S', T', s'_0, F'), \text{ in cui } \left\{ \begin{array}{l} S' \subseteq 2^S \\ T': S' \times \Sigma \rightarrow S' \\ s'_0 = s_0 \\ F' \subseteq S' \end{array} \right. \end{array} \right.$$

- Computazione di  $S'$ ,  $T'$ ,  $F'$ :

$S' := \{s'_0\}$ ;  $T' := \emptyset$ ;

**repeat**

Scegli un nodo  $A \in S'$  non marcato;

**for each**  $c \in \Sigma$  che marca una transizione di  $M$  uscente da un nodo in  $A$  **do**

**begin**

$A'_c := \{z \mid s \in A, s \xrightarrow{c} z \in T\}$ ;

$\underline{A'_c} := \varepsilon\text{-chiusura}(A'_c)$ ;

**if**  $\underline{A'_c} \notin S'$  **then**  $S' := S' \cup \{\underline{A'_c}\}$ ;

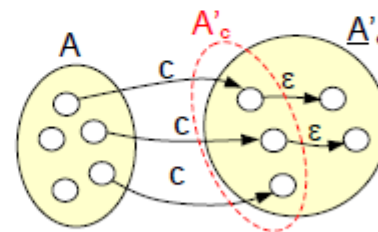
$T' := T' \cup \{A \xrightarrow{c} \underline{A'_c}\}$

**end;**

Marca  $A$

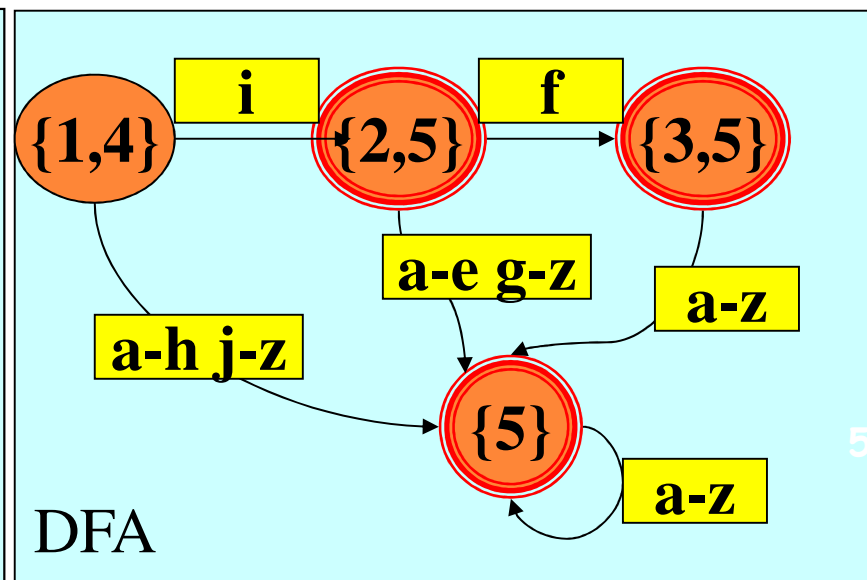
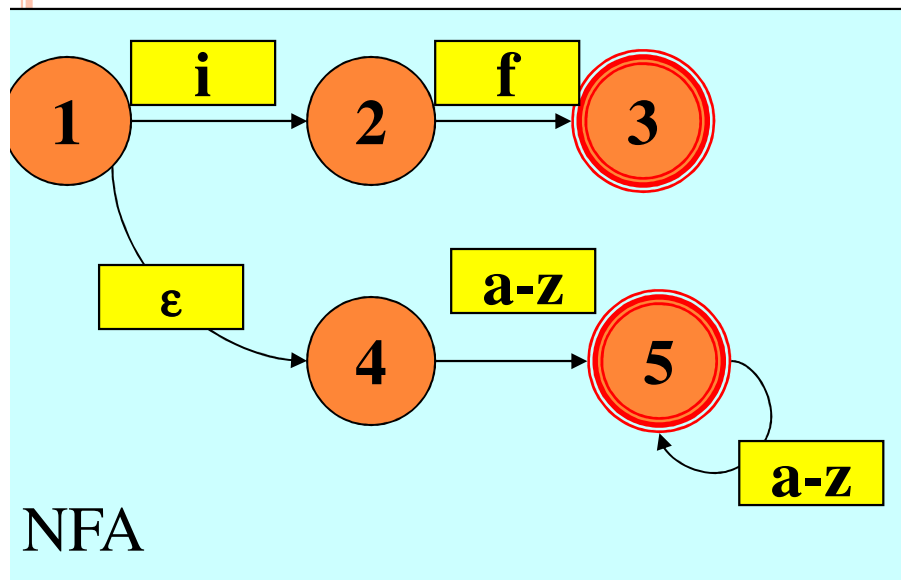
**until** tutti gli elementi in  $S'$  sono marcati;

$F' := \{A \mid A \in S', A \cap F \neq \emptyset\}$ .



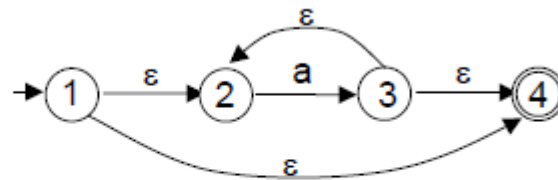
## NEL NOSTRO ESEMPIO

- Calcola la  $\epsilon$ -chiusura dello stato 1 in NFA:  $\{1, 4\}$ 
  - Gli stati Raggiungibili da 1 attraverso  $\epsilon$
- Crea le transizioni da ogni stato di  $\{1,4\}$ 
  - su "i" si ha:  $\{2,5\}$
  - calcolare la  $\epsilon$ -chiusura di  $\{2,5\}$  non ci dà un nuovo stato in DFA
  - $\{2,5\}$  è uno stato finale in DFA poichè 5 è finale in NFA
- Ripeti finchè non hai preso in considerazione o marcato tutti gli stati

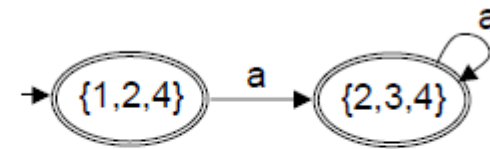


# ESEMPI: DA NFA A DFA

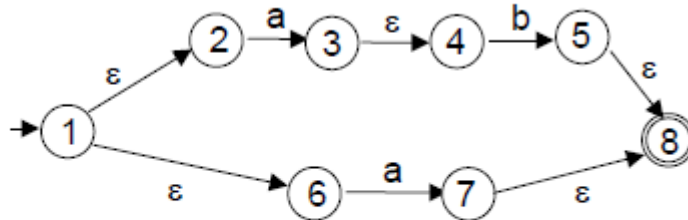
$a^*$



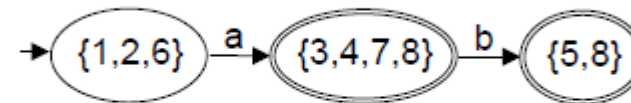
$$s'_0 = \underline{s_0} = \underline{1} = \{1, 2, 4\}$$



$ab \mid a$

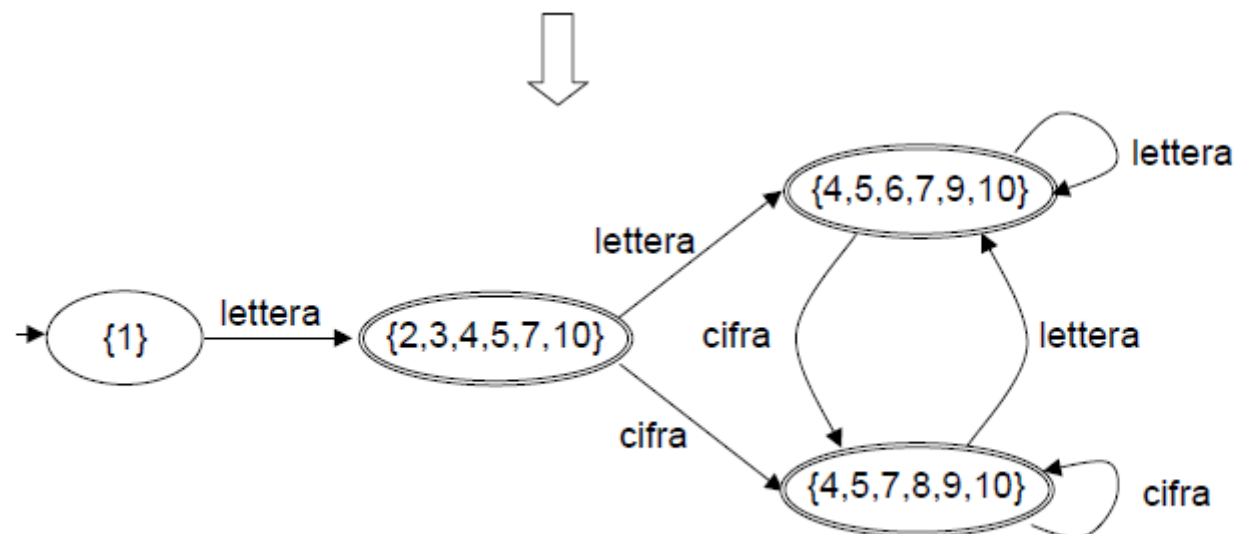
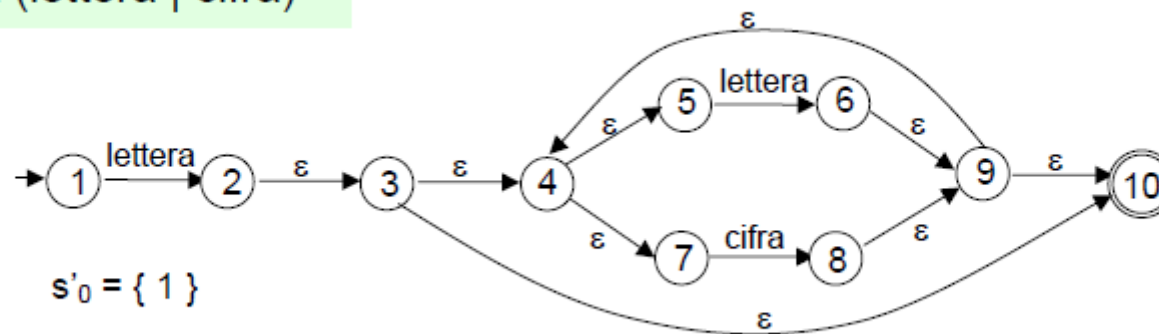


$$s'_0 = \underline{1} = \{1, 2, 6\}$$



## ESEMPI: DA NFA A DFA

lettera (lettera | cifra)\*



## SIMULARE UN NFA

Stati correnti

```
S = ε-closure(s0)
c = nextchar();
while (c != eof) {
    S = ε-closure(move(S, c));
    c = nextchar();
}
if (S ∩ F != ∅) return 'yes';
else return 'no';
```

Abbiamo bisogno di 2 pile:

**Oldstates**, memorizza l'insieme corrente di stati (parte dx, linea 4).

**Newstates**, memorizza gli stati successivi (parte sx, linea 4).

- Per ogni iterazione Newstates diventa Oldstates.
- Un array booleano indica quali stati stanno in Newstates.
- Una tabella di transizione in cui nell'elemento (t,x) ci possono essere liste di stati.
- N stati, M transizioni  
 $O(k(N+M))$ ,  $k = |x|$

## USARE NFA O DFA?

Data una RE  $r$  e una stringa  $s$ , esistono due strategie per testare se  $s$  sta in  $L(r)$ :

- Costruire l'NFA da  $r$  in  $O(|r|)$  tempo. Infatti il numero di stati è proporzionale ad  $|r|$  e ci sono al più 2 transizioni per ogni stato. La tabella di transizione viene memorizzata in  $O(|r|)$  spazio.

Il riconoscimento di una stringa  $s$  mediante un NFA con  $|r|$  stati può essere simulato efficientemente mediante due stack in  $O(|r| \times |s|)$ .

- Costruire il DFA da  $r$  applicando la subset construction ed eventualmente la minimizzazione. La subset construction costa un tempo  $O(|r|^2 M)$  dove  $M$  è il numero di stati del DFA ottenuto. Nel caso medio in cui  $M$  è circa  $|r|$ , allora  $O(|r|^3)$ .

## RIASSUMENDO I COSTI...

Automa	Costo iniziale	Costo per stringa
NFA	$O( r )$	$O( r  \times  s )$
DFA, caso medio	$O( r ^3)$	$O( s )$
DFA, caso peggiore	$O( r ^2 2^{ r })$	$O( s )$



## COSA SI PREFERISCE?

- I generatori di analizzatori lessicali e altri sistemi di string processing partono da espressioni regolari.
- Se lo string-processor deve essere usato più volte (è il caso dell'analizzatore lessicale) allora il costo della costruzione del DFA è sopportabile. Quindi si preferisce costruire il DFA o passando dal NFA o direttamente e poi applicando algoritmi di minimizzazione.
- Nel caso di applicazioni come grep in cui l'utente specifica un'espressione regolare, allora conviene simulare direttamente un NFA.
- Esistono anche strategie miste, in cui si comincia con la simulazione del NFA, memorizzando però gli insiemi degli stati e le transizioni via via calcolati. Ad ogni passo si vede se una data transizione è stata già calcolata.

## OTTIMIZZAZIONE DI UN DFA

- Si può costruire un DFA direttamente a partire da un'espressione regolare senza passare per il NFA. In alcuni casi si ottengono DFA con minor numero di stati
- Si può minimizzare il DFA ottenendo il DFA minimale. Il tempo è  $O(n \log n)$
- Rappresentazione compatta delle tabelle di transizione.



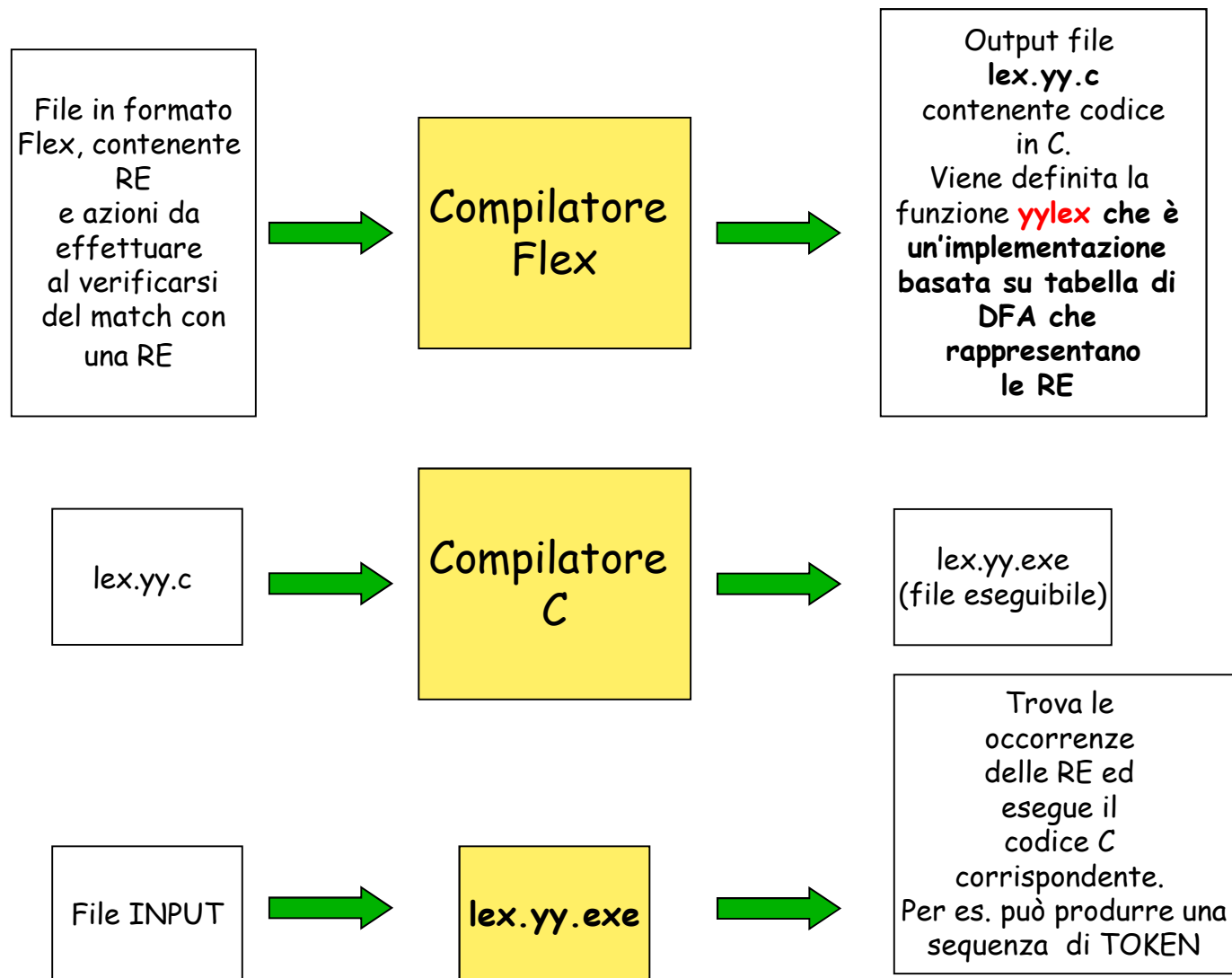
# *GENERATORI AUTOMATICI DI SCANNER: FLEX*

Lunedì 12

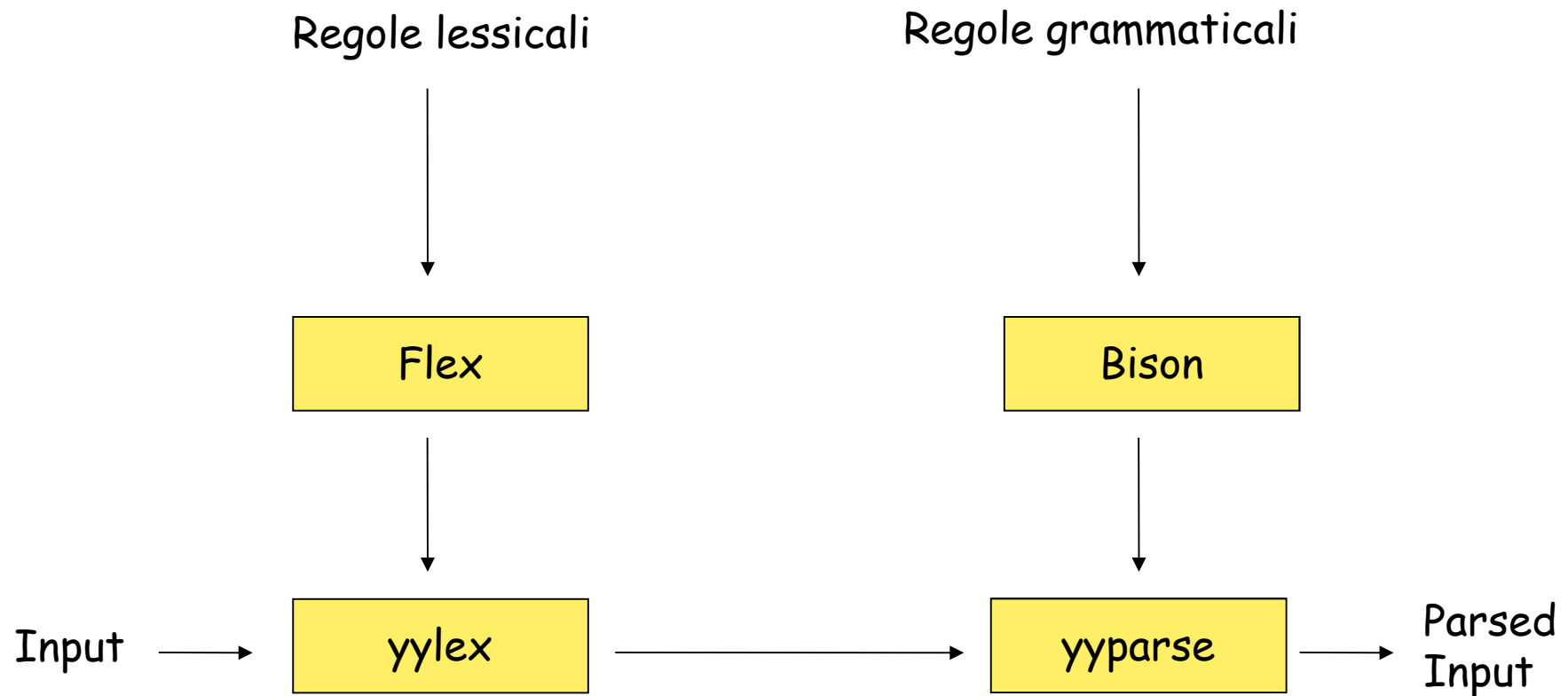
# GENERATORI DI SCANNER

- Un generatore automatico di scanner **prende in input** un file che specifica il lessico di un certo linguaggio:
  - solitamente nella forma di espressioni regolari
  - ...e includendo altre funzioni ausiliarie, definizioni di token, ...
- ...e **produce in output** un codice (scritto in un certo linguaggio) che implementa il ruolo dello scanner
- Esistono molti generatori automatici:
  - Lex, Flex, ScanGen, ...
    - generano un codice in C
  - JLex, Sable, Cup
    - generano un codice in Java
  - **Lex** fu il primo generatore di scanner. Esso fu inventato da Mike Lesk e Eric Schmidt (AT&T Bell Lab) nel 1975.
  - Esistono tanti software alternativi al Lex. Uno dei più conosciuti ed usati è **Flex** - Fast Lexical analyser generator (introdotta da Vern Paxson intorno al 1987 per risolvere problemi di efficienza).
  - Flex è un free software. Pur non essendo un software GNU, il GNU Project ne distribuisce un manuale.
    - <http://flex.sourceforge.net/> (linux)
    - <http://gnuwin32.sourceforge.net/packages/flex.htm> (windows)

# USO DI FLEX



# USO DI FLEX E BISON



# SCRIVERE UN PROGRAMMA IN FLEX

Un file in formato Flex (o Lex) consiste di 3 sezioni, separate da %%:

## DEFINIZIONI

che contiene:

- definizioni di variabili o definizioni regolari
- segmento di codice *C*, indentato oppure delimitato da %{ e %}, che deve comparire nel file di output

%%

## REGOLE

che contiene una sequenza di regole contenenti:

- i pattern espressi mediante RE
- codice *C* da eseguire in corrispondenza del match con un certo pattern

%%

## FUNZIONI AUSILIARIE (opzionale)

che contiene codice *C* che deve essere copiato nel file di output

Anche nella sezione REGOLE è possibile inserire codice tra %{ e %}. Ciò deve avvenire prima della prima regola.

Può servire per esempio per dichiarare variabili locali usate nella routine di scanning.