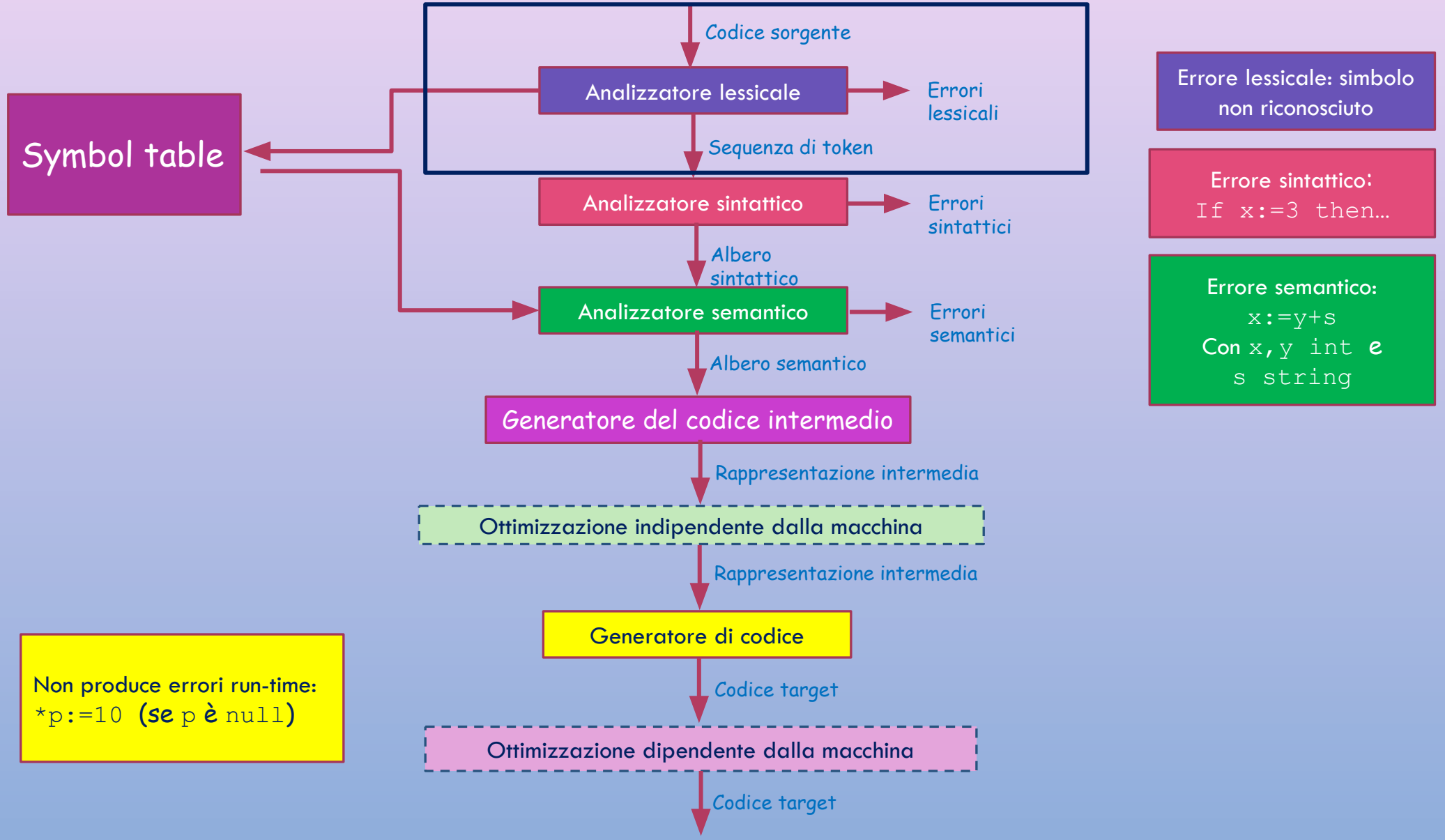
ANALISI LESSICALE


Codice sorgente
Analizzatore lessicale
Errori lessicali
Sequenza di token
Symbol table
Analizzatore sintattico
Errori sintattici
Albero sintattico
Analizzatore semantico
Errori semantici
Albero semantico
Generatore del codice intermedio
Rappresentazione intermedia
Ottimizzazione indipendente dalla macchina
Rappresentazione intermedia
Generatore di codice
Codice target
Ottimizzazione dipendente dalla macchina
Codice target
Errore lessicale: simbolo non riconosciuto
Errore sintattico: If x:=3 then…
Errore semantico: x:=y+s Con x,y int e s string
Non produce errori run-time: *p:=10 (se p è null)

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. Questo processo viene spesso chiamato tokenizzazione (tokenizing).

Per esempio, consideriamo la seguente linea di codice:

```
for index:=1 to n do a[index] := index * 2;
```

Questo codice contiene 33 caratteri non-blank ma invia 13 token al parser:

Keywords: `for, to, do` operatori: `:= , *`

Identificatori: `index, a, n` punteggiatura: `;`

Costanti: `1, 2` parentesi: `[, ]`

OPERAZIONI PRELIMINARI

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

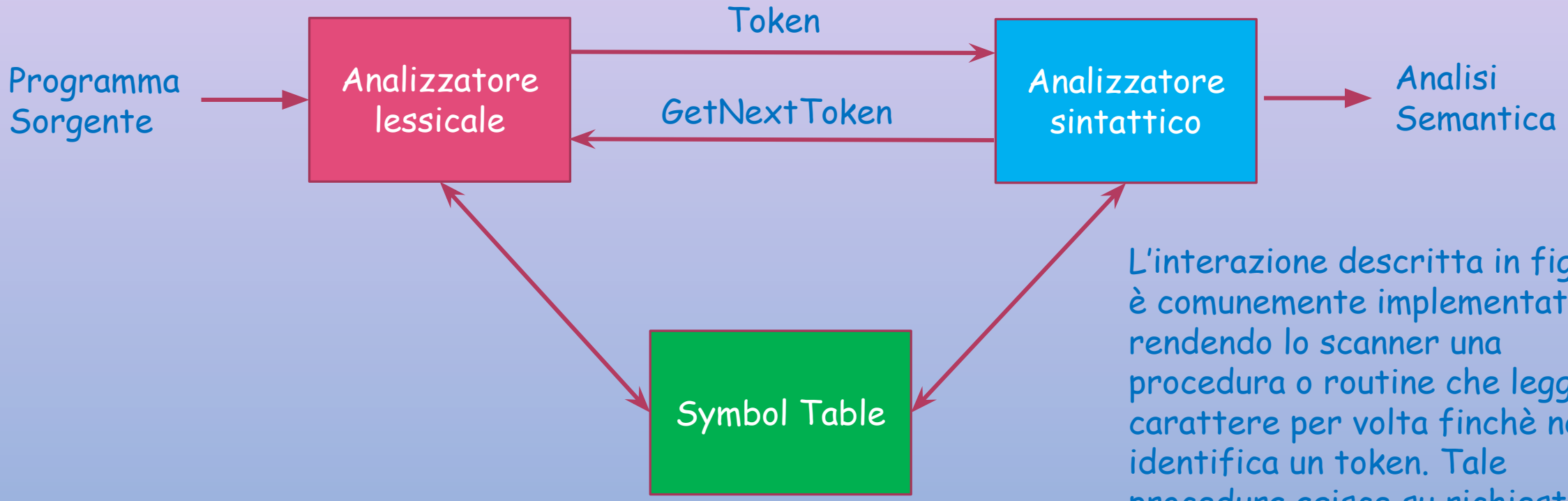
- **Rimuovere i commenti:** i commenti sono individuabili con simboli speciali (per esempio */). Lo scanner deve individuare tali simboli.
- **Case conversion:** molti linguaggi di programmazione (ad es. Pascal) ignorano la lettere maiuscole. Lo scanner deve convertire tutte le letter in lowercase, per 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:

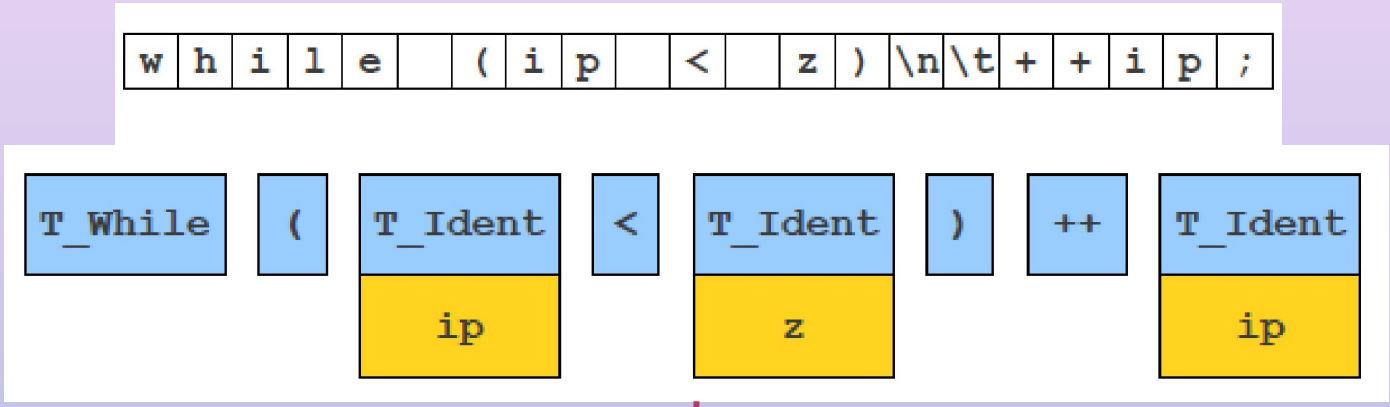
- **Scanning**: non richiede la tokenizzazione ma solo la rimozione dei commenti e la compattificazione degli spazi bianchi consecutivi in uno solo;
- **Analisi lessicale**: lo scanner produce la sequenza di token.

INTERAZIONI FRA SCANNER E PARSER

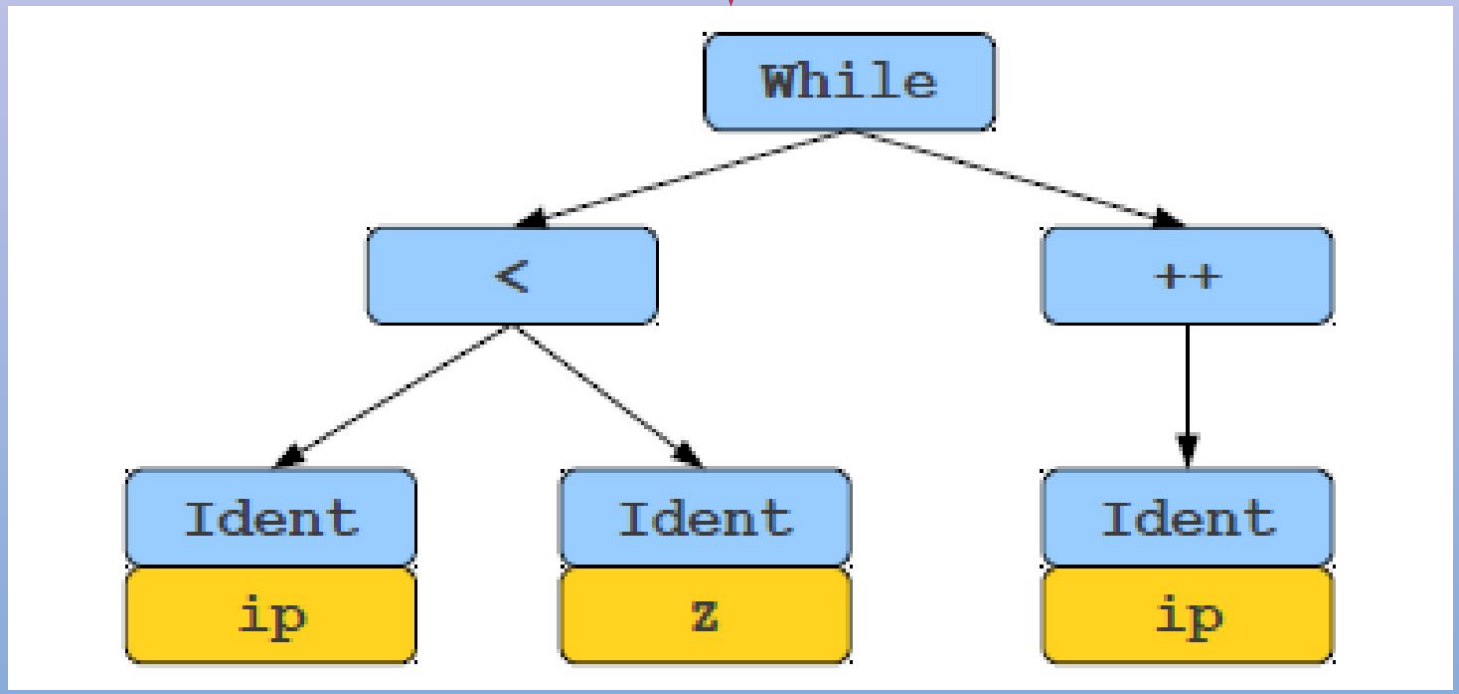


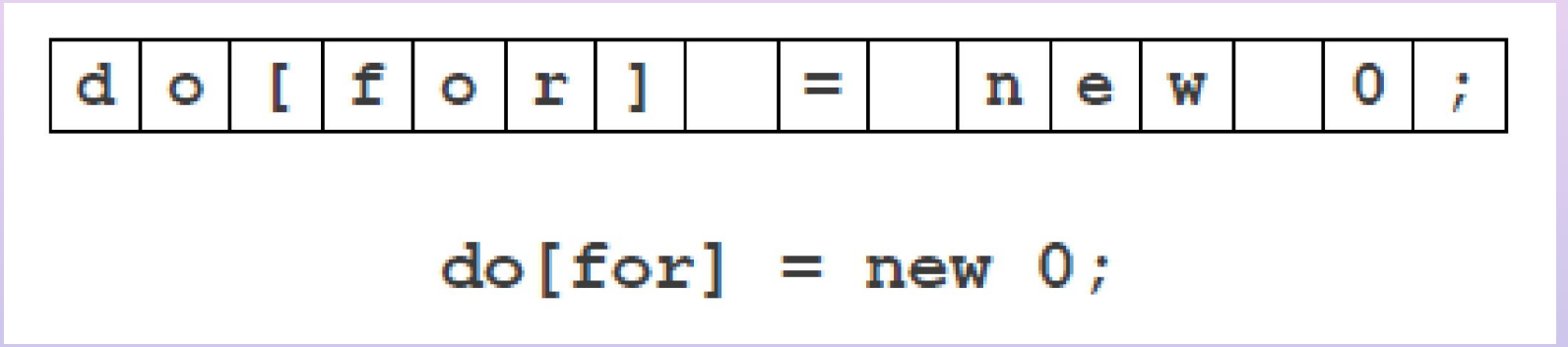
L'interazione descritta in figura, è comunemente implementata rendendo lo scanner una procedura o routine che legge un carattere per volta finchè non identifica un token. Tale procedura agisce su richiesta dell'analizzatore sintattico mediante la funzione GetNextToken

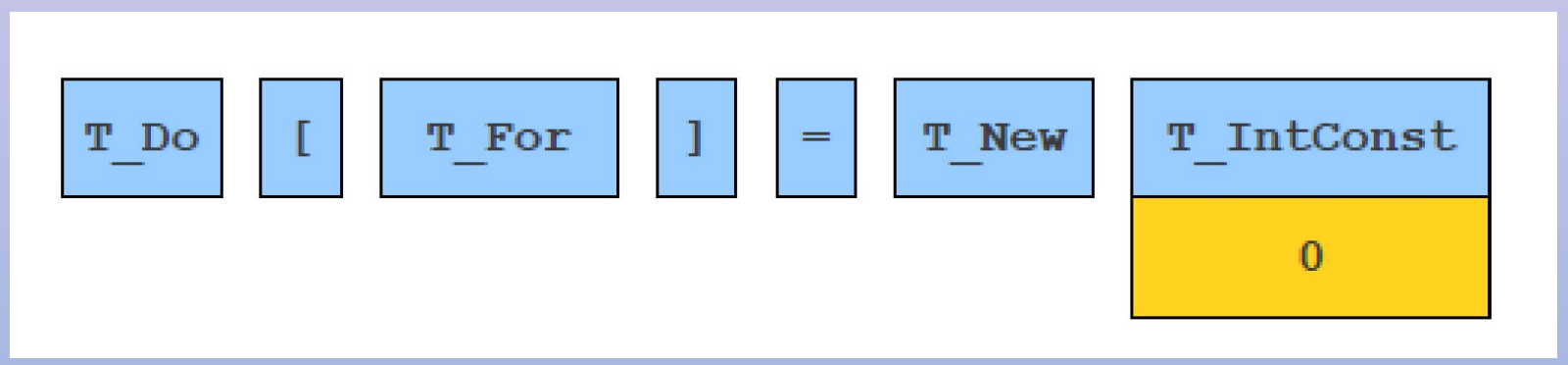
Analisi lessicale



Analisi sintattica







L'analisi lessicale non è in grado di rilevare gli errori sintattici. L'analisi lessicale sarà eseguita anche se l'analisi sintattica non andrà a buon fine

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. Le principali categorie sono:

1. le parole chiave di un linguaggio (keywords) come per esempio: if, while, do, then, else, …
2. gli identificatori: stringhe definite dall'utente costituite da lettere e numeri che iniziano con una lettera.
3. Costanti Numeriche 35,2; 48; 3,141597; 0
4. Costanti di tipo stringa di caratteri: sequenze di caratteri comprese tra virgolette
5. operatori: simboli come *, +, … o simboli multicarattere come >=, <=, <>, …
6. simboli speciali: parentesi, la virgola, il punto e virgola, …

LESSEMI E PATTERN

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 la descrizione una struttura in grado di definire un insieme di stringhe.

ESEMPIO

TOKEN	PATTERN	LESSEMA
T_if	Sequenza di caratteri i, f	If
T_else	Sequenza di caratteri e,l,s,e	Else
T_op_confronto	< or > or == or <> or <= or >=	<=, <,== etc
T_identificatore	Lettera seguita da lettere o cifre	x, y, rate, d
T_costante	Qualsiasi costante numerica	3.14159, 67, 0, 3.2
T_stringa	Qualsiasi sequenza di caratteri diversi da " e circondati da "	"Linguaggi formali"
T_openbrace	{	{
…	…	…

TOKEN E LESSEMI

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

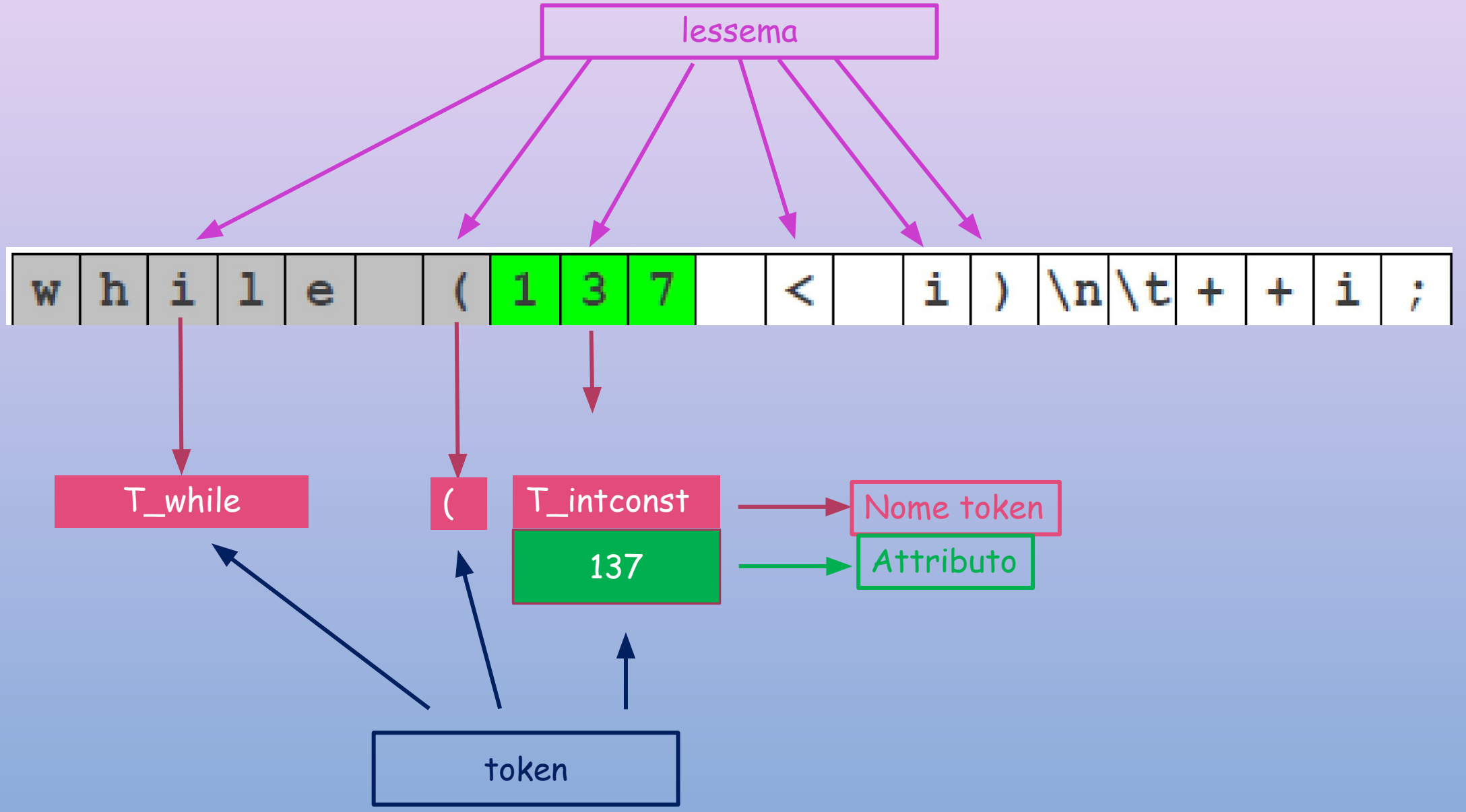
In questo esempio il token id indica gli identificatori. A questi corrisponde un insieme infinito di possibili lessemi, per cui occorre una regola per definirli

Le parole chiave invece sono rappresentate dalla sequenza delle loro lettere

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

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

TOKEN	LESSEMA
Id	dist, rate, end, start, maxdist
Relop	>=
(	(
)	)
If	If
Then	Then
:=	:=
*	*
-	-
;	;


lessema
w h i l e (1 3 7 < i) \n \t + + i ;
T_while
(
T_intconst
137
Nome token
Attributo
token

PERCHÉ 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 (per esempio, anche gli spazi bianchi diventerebbero simboli terminali della grammatica complicandola ulteriormente);
- miglioramento dell'efficienza (esistono tecniche specializzate per l'analisi lessicale);
- Permette la portabilità del compilatore perché tutte le funzioni di ingresso dipendenti dalla macchina sono localizzate in un unico componente

ATTRIBUTI

Gli **attributi** sono particolari informazioni relative ad uno specifico lessema di un particolare token.

Gli attributi sono necessari quando più lessemi possono corrispondere a un pattern. E' quindi importante identificare in qualche modo ogni singolo lessema.

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

ESEMPIO TOKENIZZAZIONE

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>

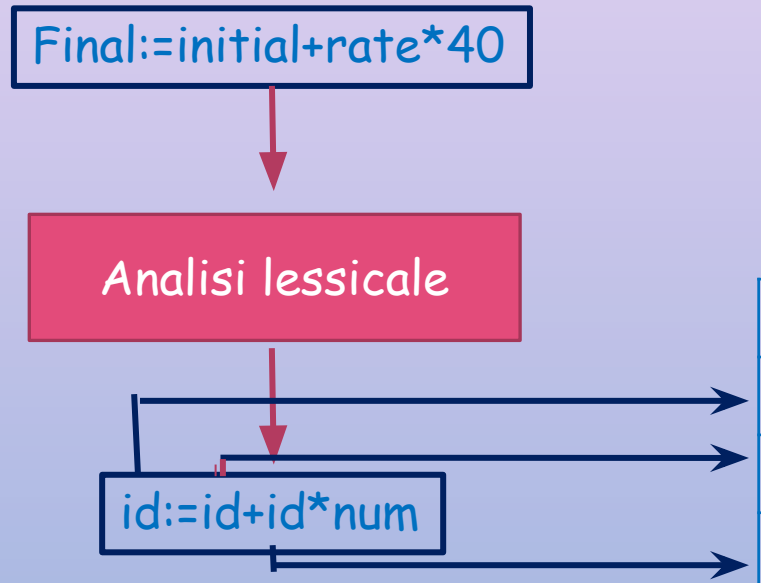
<id, punt. nella symbol table alla entry relativa a C>

<op_exp>

<num, valore intero 2>

<id> <op_ass> <id> <op_molt> <id> <op_exp> <num>

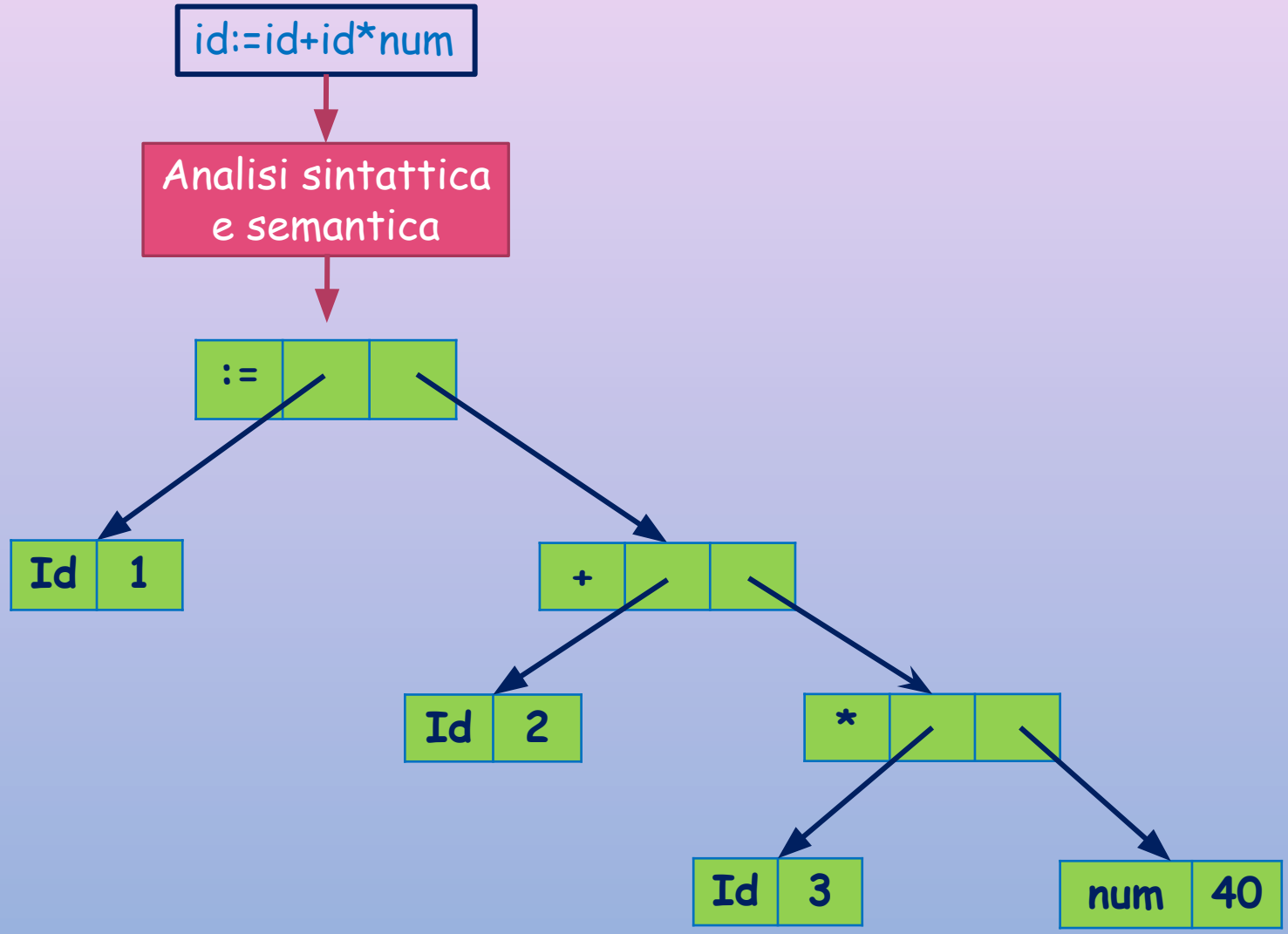
ESEMPIO



Symbol table

indirizzo	Simbolo	proprietà
1	Final	Float, …
2	Initial	Float,…
3	Rate	Float,…

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


id:=id+id*num
Analisi sintattica
e semantica
:=
Id
1
+
Id
2
*
Id
3
num
40

IMPLEMENTAZIONI DELLA SYMBOL TABLE

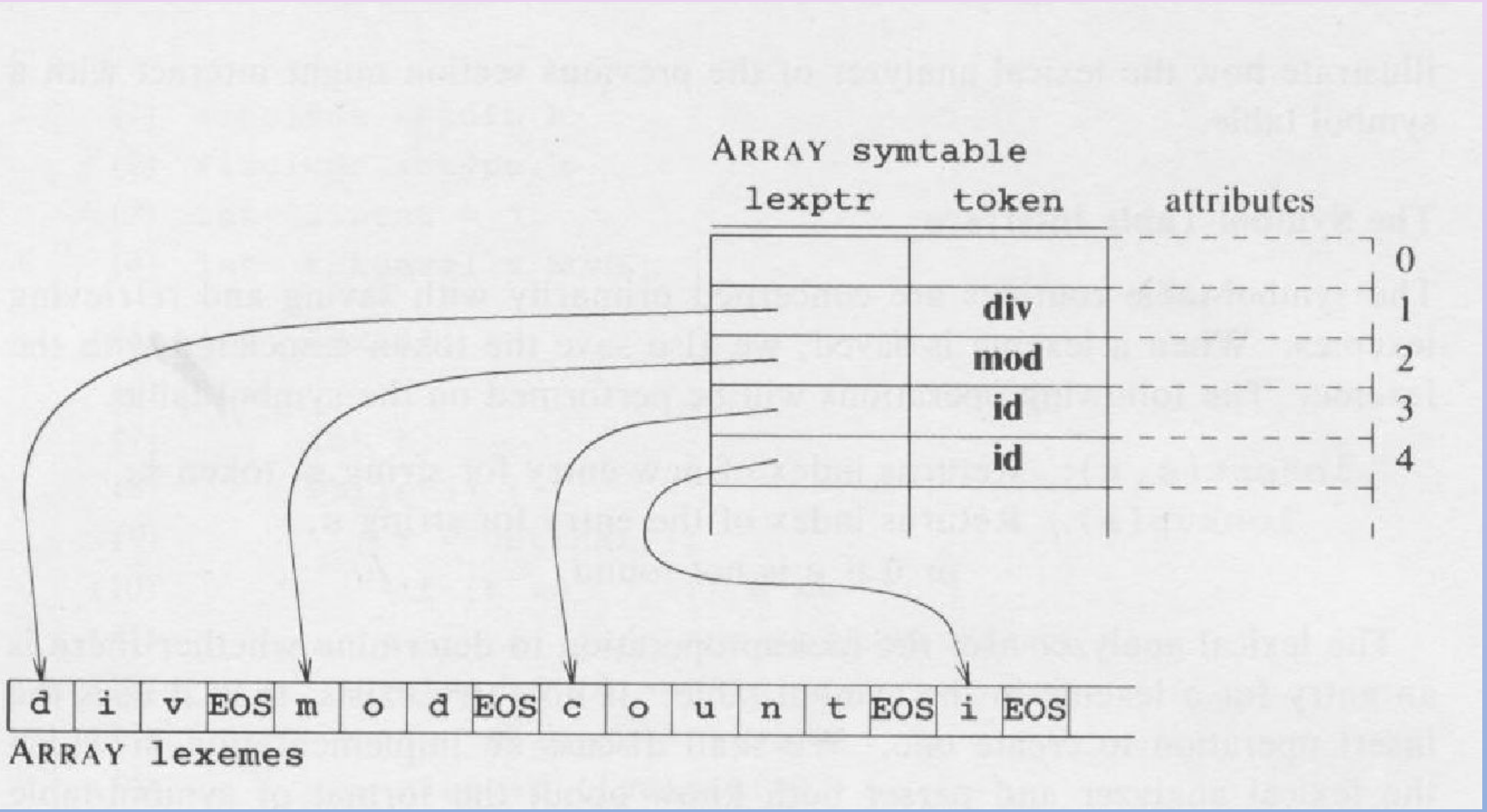
Spesso non è opportuno assegnare uno spazio fissato ad un lessema (troppo per alcuni lessemi, troppo poco per altri). Le informazioni salvate dipendono dal genere di oggetto, per tale motivo si ha la necessità che le entry della tabella dei simboli abbiano un formato flessibile.

Lo scanner comunica con la Symbol Table 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

L'implementazione della symbol table verrà effettuata mediante una struttura dati in cui tali operazioni siano svolte in maniera efficiente. In genere vengono implementate mediante tabelle hash.


ARRAY symtable
lexptr
token
attributes
div
mod
id
id
0
1
2
3
4
d i v EOS m o d EOS c o u n t EOS i EOS
ARRAY lexemes

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<=x) then …,` in genere uno scanner non può sapere se "`fi`" è un mispelling 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:

1. Scrivere l'analizzatore lessicale come un programma in linguaggio assembly. In tal caso l'input deve essere gestito a basso livello.
2. Scrivere l'analizzatore lessicale come un programma in un linguaggio di programmazione convenzionale. L'input si gestisce sfruttando le caratteristiche del linguaggio.
3. usare un generatore automatico di analizzatori lessicali, come lex, flex, …. il generatore fornisce anche le routine per la lettura dell'input.

ANALISI LESSICALE: GESTIONE DELL'INPUT

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. Per esempio:

- 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 elaborali, 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)

DOPPIO BUFFERING

La tecnica usa due puntatori nel buffer: il puntatore forward e il puntatore lexeme_beginning.

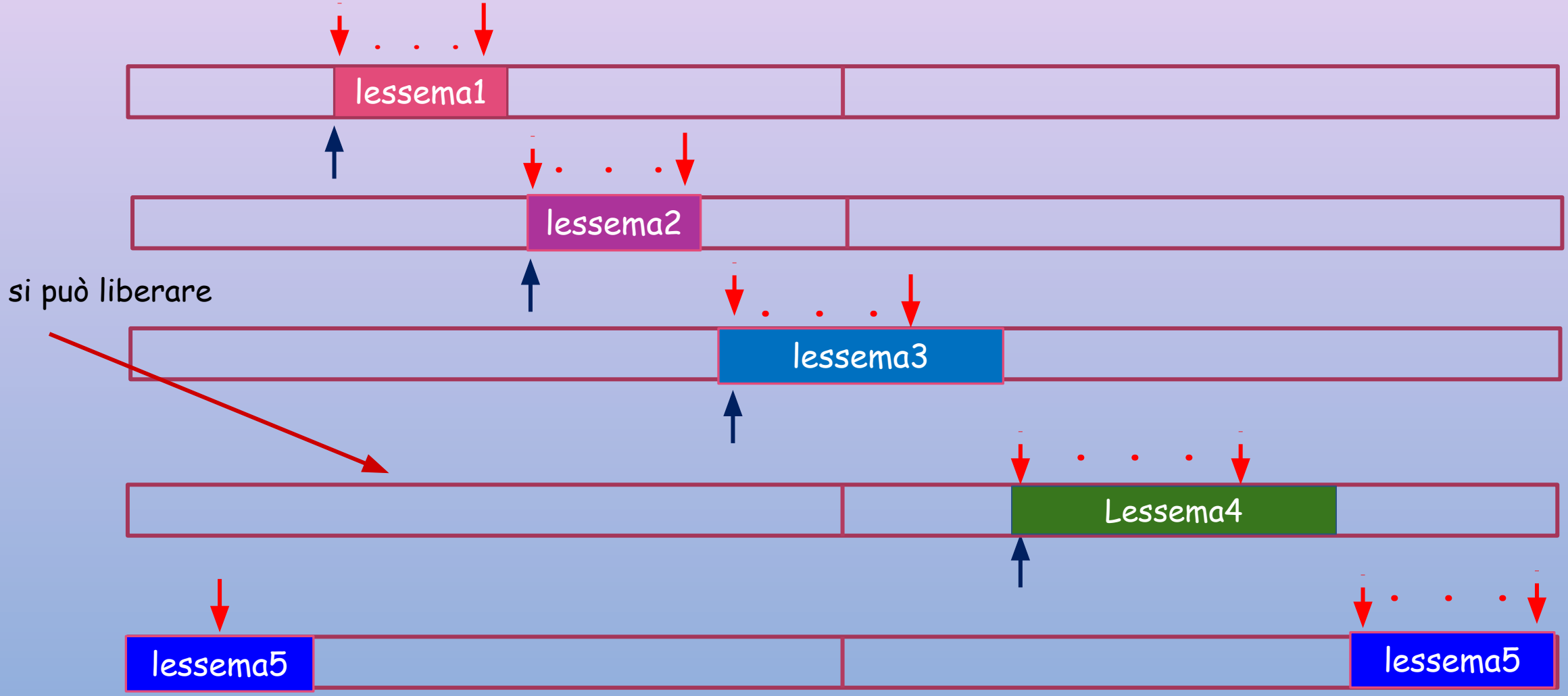
All'inizio e dopo il riconoscimento di un lessema, entrambi puntano allo stesso carattere (cioè il primo carattere dell'input oppure il carattere successivo a un lessema trovato). Il puntatore lexeme_beginning rimane fermo e il puntatore forward va avanti finchè viene trovato un match con un pattern. La stringa di caratteri fra i due puntatori è il corrente lessema.

Quindi entrambi i puntatori sono settati al carattere destra del lessema appena trovato, e si procede iterativamente fino a che l'input non viene completamente esaminato.

Quando il puntatore forward arriva alla fine della prima metà, la seconda metà è caricata con N nuovi caratteri input e il puntatore forward è semplicemente incrementato.

Quando il puntatore forward arriva alla fine della seconda metà, la prima metà è caricata con N caratteri e il puntatore forward è settato all'inizio del buffer

DOPPIO BUFFERING



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;
```

Viene letto il primo blocco e viene sostituito

```
for i:=1 to r do rate:=dist + i;
```

DESCRIZIONE DEI TOKEN: ESPRESSIONI REGOLARI E AUTOMI A STATI FINITI

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**

LINGUAGGI FORMALI: TERMINOLOGIA E NOTAZIONE

- **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 ε)
- **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

NOTAZIONE PER ESPRESSIONI REGOLARI

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

- l'espressione ε denota il linguaggio L={ε}
- l'espressione ∅ 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) ∪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. I linguaggi definiti mediante espressioni regolari sono detti linguaggi regolari. Esempio: identificatori=lettera(lettera|cifra)*

ESEMPI

- bbb è un'espressione regolare che denota l'insieme {bbb}.
- ab|bbb è un'espressione regolare che denota {ab, bbb}.
- ba* è un'espressione regolare che denota {b, ba, baa, baaa, …}.
- 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

1. RE per stringhe di lunghezza pari su alfabeto {a,b}

2. RE per stringhe con esattamente una "b" sull'alfabeto {a,b}

3. RE per gli integer (possibilmente preceduti da '+' or '-', che non cominciano per 0; l'alfabeto è {0,...,9,+,-})

4. RE per gli identificatori del linguaggio C

LINGUAGGI REGOLARI

Un **linguaggio** si dice **regolare** se esiste un'espressione regolare che lo definisce

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

Esempio le espressioni (a|b)* e (a*b*)* sono equivalenti perché entrambe definiscono il linguaggio di tutte le parole sull'alfabeto {a,b}

AUTOMI A STATI FINITI

Un **automa finito deterministico** (**dfa**) è una quintupla **(Q, A, δ, q_0, F)** dove

- **Q** è **l'insieme di stati**
- ***A*** è un insieme di simboli (**alfabeto di input**)
- **δ:** $Q \times A \rightarrow Q$ è la **funzione di transizione** o di stato dell'automa ed associa alle coppie stato-simbolo uno stato.
- **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: **$\delta'(q,x)$** = stato in cui si trova l'automa dopo aver letto tutti caratteri di x. E' definito ricorsivamente come:

$\delta'(q,\varepsilon) = q$ per ogni q in Q ;

$\delta'(q,xa) = \delta(\delta'(q,x),a)$

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

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 in cui:

- la taglia del codice è ridotta
- è facile da modificare
- lo svantaggio sta nel fatto che le tabelle possono diventare molto grandi

SIMULARE UN DFA CON DIAGRAMMI DI TRANSIZIONE

```
switch (state) {
case 0: c=nextchar();
if (c==‘a’) state =1;
else if (c==‘b’) state =2;
else …
case 1: ….
….
case 5: return (‘yes’); }
```

Un caso per ogni stato dell’automa

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’;
```

NEL CASO DELL'ANALISI LESSICALE IL PROBLEMA È UN PO' PIÙ COMPLESSO…

I DFA sono un modo per rappresentare algoritmi che riconoscono stringhe secondo 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.

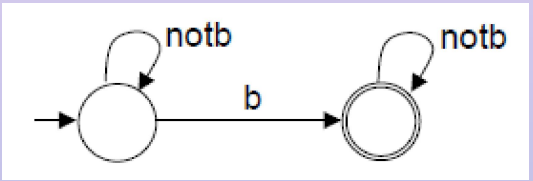
Bisogna arricchirli con le **azioni** corrispondenti e/o con le transizioni che definiscono errori.

COME COSTRUIRE L'AUTOMA?

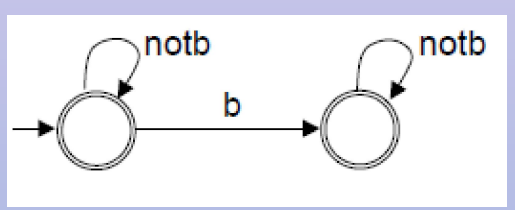
- Determinare le espressioni regolari che definiscono i token.
- Trovare il corrispondente NFA (Teorema di Kleene)
- Se serve, determinizzare l'automa per ottenere il DFA corrispondente (subset construction)

ESEMPI DI DFA ASSOCIATI A TOKEN

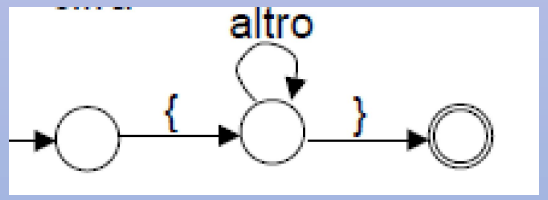
L ={w |w contiene esattamente una "b"}



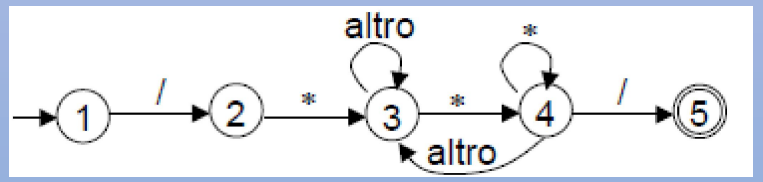
L ={w |w contiene al più una "b"}



L={w | w è un commento in Pascal}



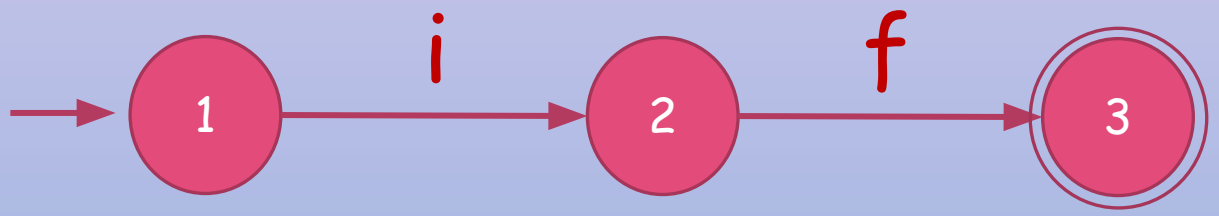
L={w | w è un commento in C}



IN PARTICOLARE PER I TOKEN?

Facciamo un semplice esempio: Automa che riconosce il token «if»

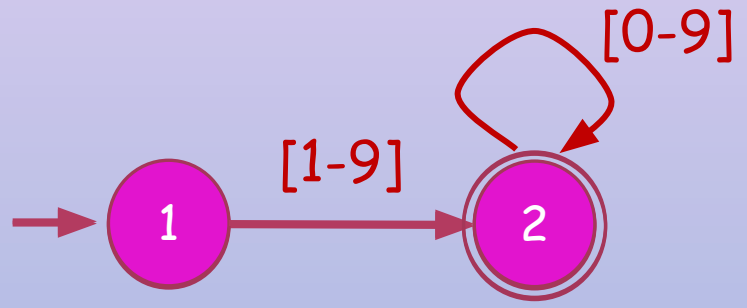
if



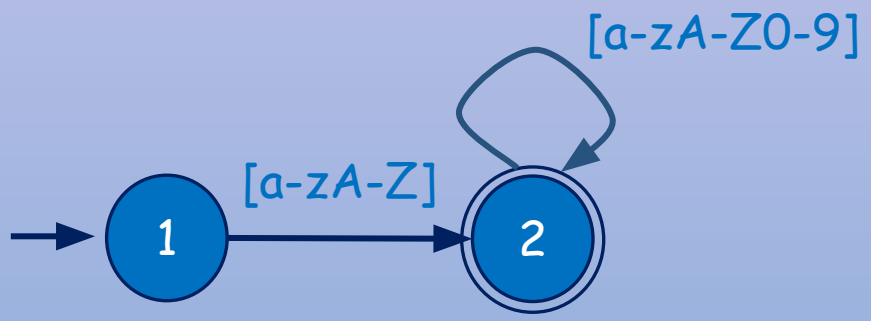
In questo caso la soluzione è molto semplice

Token numerici e identificatori

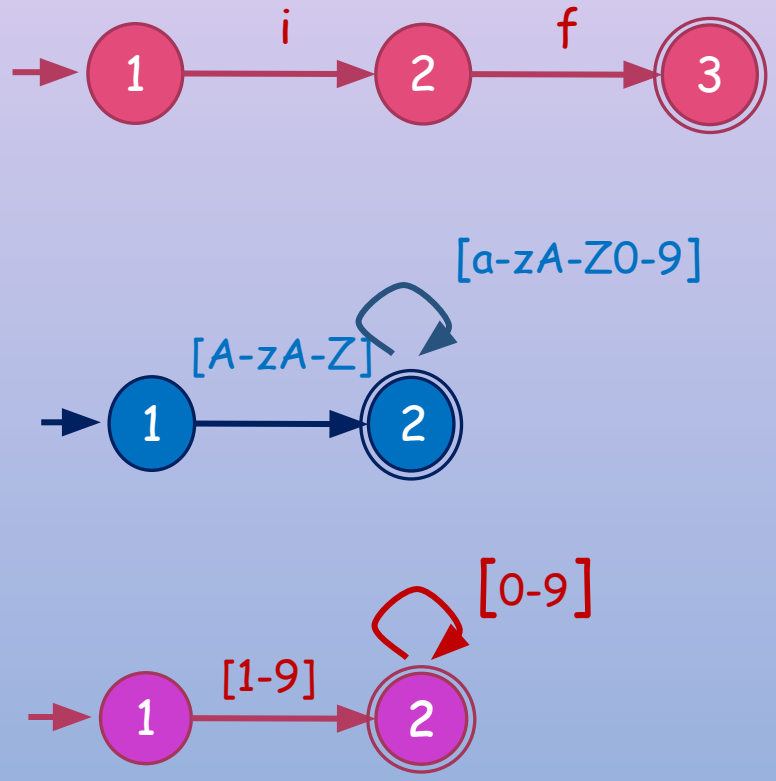
Interi positivi [1-9][0-9]*

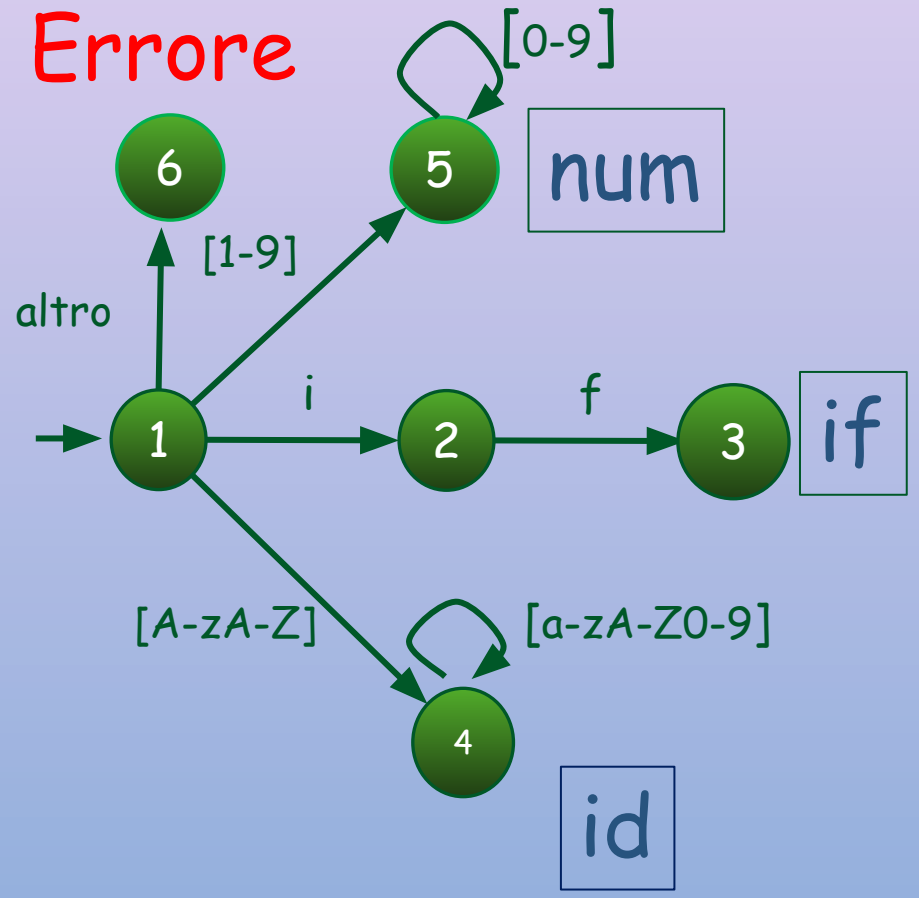


identificatori [a-z, A-Z][a-z, A-Z,0-9]*



COME COMPORRE?

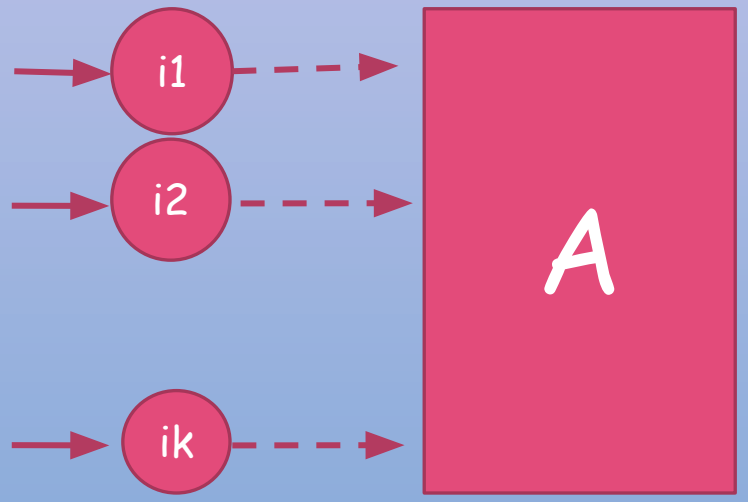




In genere si ottiene un automa non deterministico. Dallo stato 1 con la lettera i si può arrivare allo stato 2 o allo stato 4

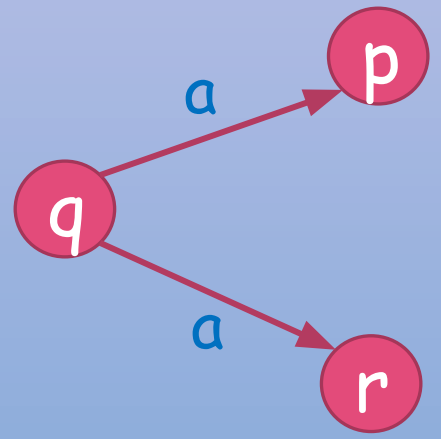
AUTOMI NON DETERMINISTICI

Possono avere più stati iniziali



Ammettono ε-transizioni

Ci sono più transizioni che partono dallo stesso stato ed etichettati con la stessa lettera



RICONOSCIMENTO TRAMITE NFA

Una stringa w è **accettata** da un **automa non deterministico** se <u>almeno uno</u> dei cammini da uno stato iniziale a uno stato finale è etichettato con w.

```
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»
```

S_0 insieme degli stati iniziali

ε-closure(S) rappresenta tutti gli stati raggiungibili da S tramite ε-mosse

move(S,c) è la funzione di transizione

nextChar() restituisce il carattere successivo del testo

Essendo in genere l' ε-closure (S) un insieme di stati, esso viene gestito mediante due pile **oldstates** e **newstates**

RICONOSCIMENTO TRAMITE NFA

Il riconoscimento di una stringa w tramite automa non deterministico può essere effettuato con l'ausilio di due pile:

Oldstates: dopo la lettura di un prefisso w_1 di w, Oldstates rappresenta l'insieme degli stati di arrivo di tutti i cammini dallo stato iniziale dopo la lettura di w_1

Newstates: se il prefisso successivo di w è w_1a, allora vengono inseriti in Newstates via via gli stati di arrivo degli stati di oldstates dopo la lettura di a.

RICONOSCIMENTO TRAMITE NFA

Alla lettura di un nuovo simbolo a, si entra in un ciclo che termina quando Oldstates diventa vuota, che ad ogni iterazione effettua u=Pop(Oldstates), ed effettua Push(Newstates, δ (u,a)). Infine Oldstates viene impostata a Newstates prima della lettura del nuovo carattere.

In supporto, a fini dell'efficienza, si possono utilizzare anche le seguenti strutture dati:

- un array booleano che indica quali stati stanno già in Newstates, per accelerare la verifica per evitare un eventuale riinserimento nella pila
- Una tabella di transizione in cui nell'elemento (u,a) ci possono essere liste di stati.

Se l'automa ha N stati, M transizioni, **il riconoscimento di un stringa w** avviene in tempo **O(|w|(N+M))**.

NFA O DFA?

In genere gli automi non deterministici sono più facili da costruire dalle espressioni regolari, ma in generale il riconoscimento può essere pesante proprio a causa del non determinismo. Inoltre gli automi deterministici sono più facili da implementare. Quindi una strategia può essere la seguente:

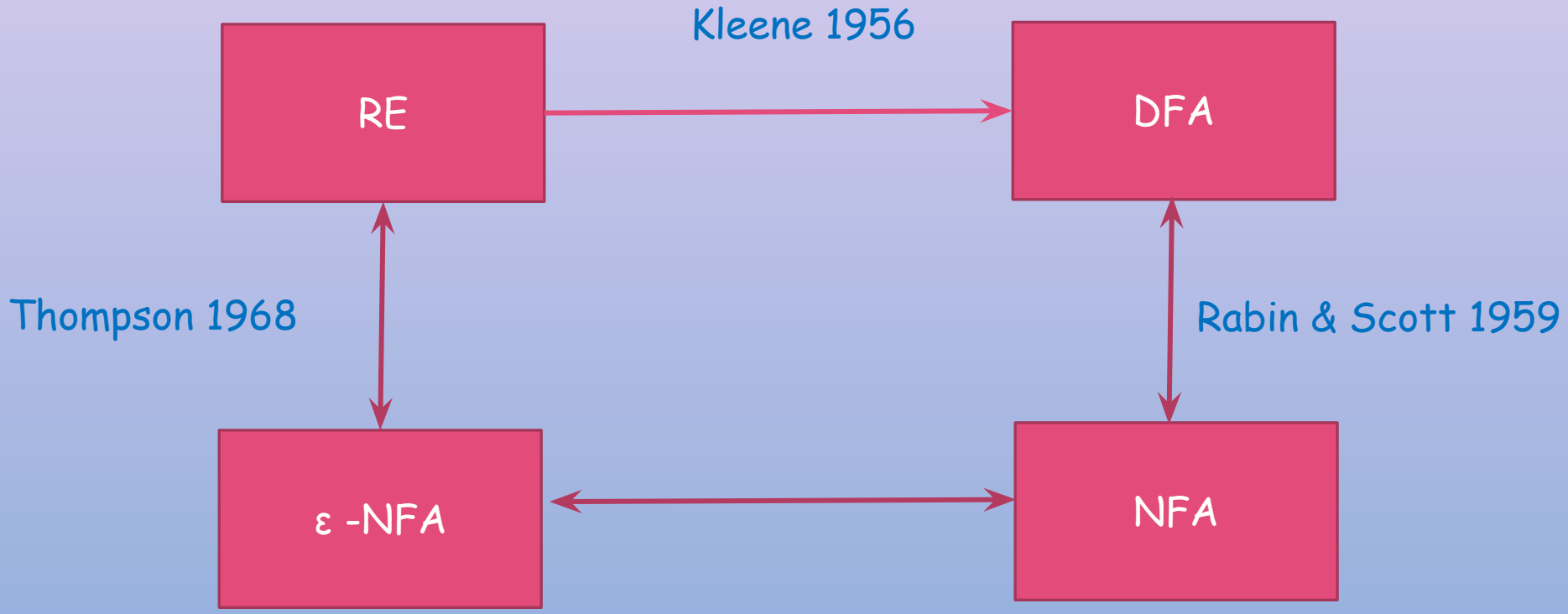
1. Si costruisce un automa non deterministico corrispondente all'espressione regolare
2. Si determinizza l'automa, ossia si trova un automa deterministico equivalente al non deterministico dato.

<u>Osservazioni:</u>

1. Il DFA ottenuto può essere esponenzialmente più grande come numero di stati rispetto all'NFA

2. Non si ottiene necessariamente l'automa minimale, per cui potrebbe essere necessaria una minimizzazione

EQUIVALENZE FRA CLASSI DI LINGUAGGI



DA RE A E- NFA

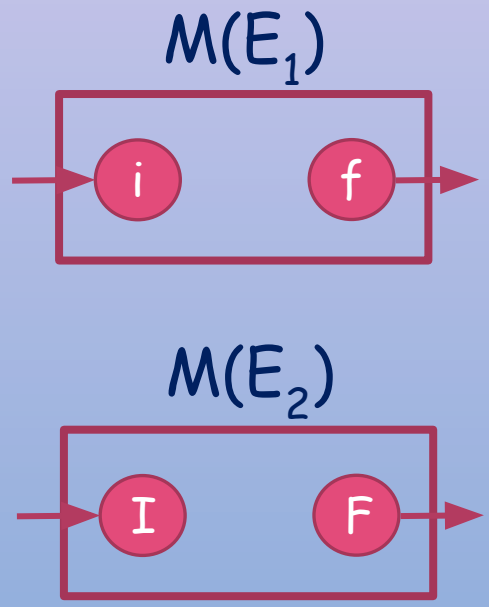
La costruzione è dovuta a Thomson. L'idea è di costruire un automa corrispondente ad ogni regola ricorsiva per la costruzione di espressioni regolari, tenendo conto che una transizione può avvenire spontaneamente, ossia senza leggere simboli in input.

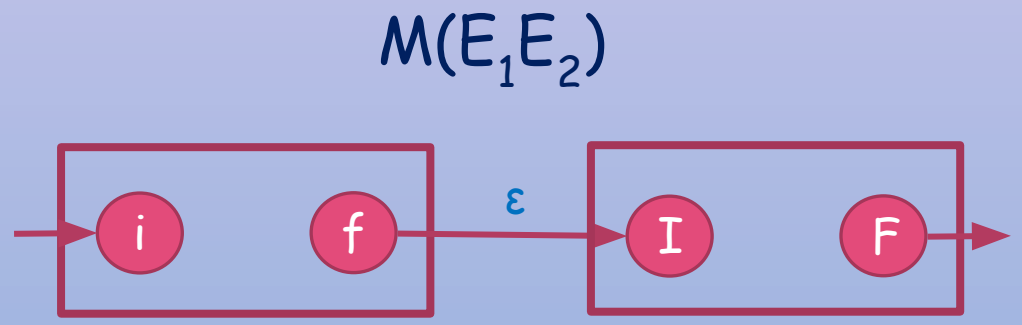
Casi base: - {ε} è un'ER

- per ogni a ∈ Σ, {a} è un'ER.

DA RE A E-NFA

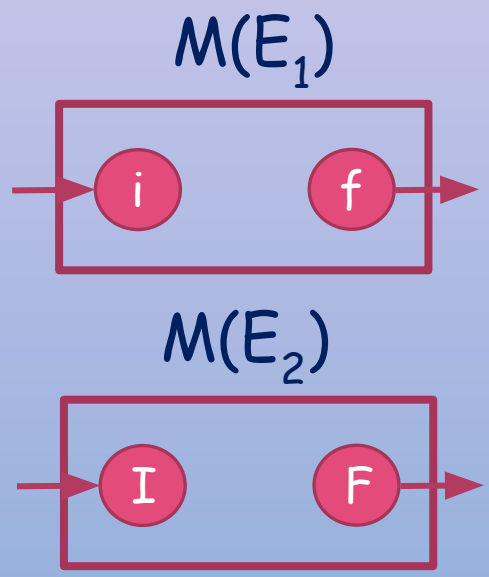
Se E_1 ed E_2 sono espressioni regolari, anche E_1E_2 è un'espressione regolare.
Sia $M(E_1)$ l'automa che riconosce E_1 e $M(E_2)$ l'automa che riconosce E_2:

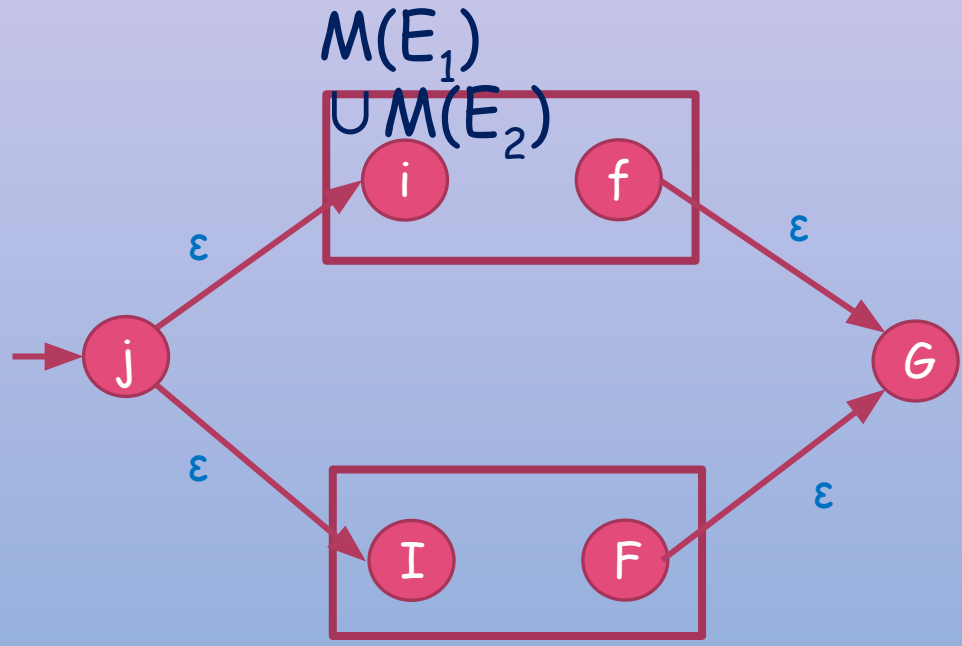




DA RE A E-NFA

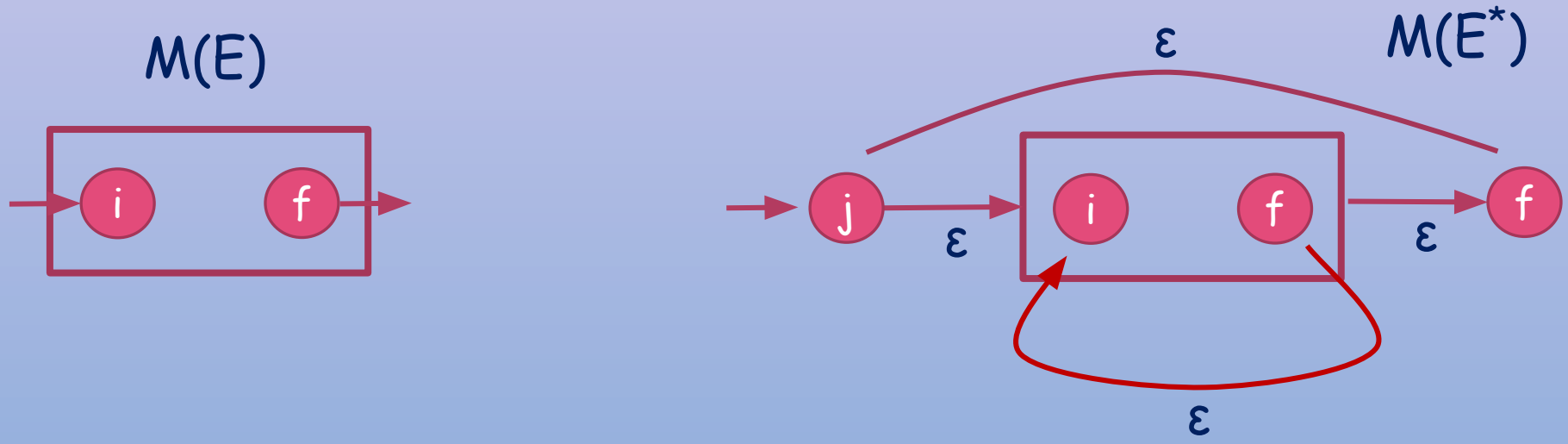
Se E_1 ed E_2 sono espressioni regolari, anche $E_1 \cup E_2$ è un'espressione regolare.
Sia $M(E_1)$ l'automa che riconosce E_1 e $M(E_2)$ l'automa che riconosce E_2





DA RE A E-NFA

Se E è un'espressione regolare, anche E^* è un'espressione regolare.
Sia M(E) l'automa che riconosce E:



DA NFA A DFA

Abbiamo visto che il riconoscimento tramite NFA ha costo $O(|w|(N+M))$.

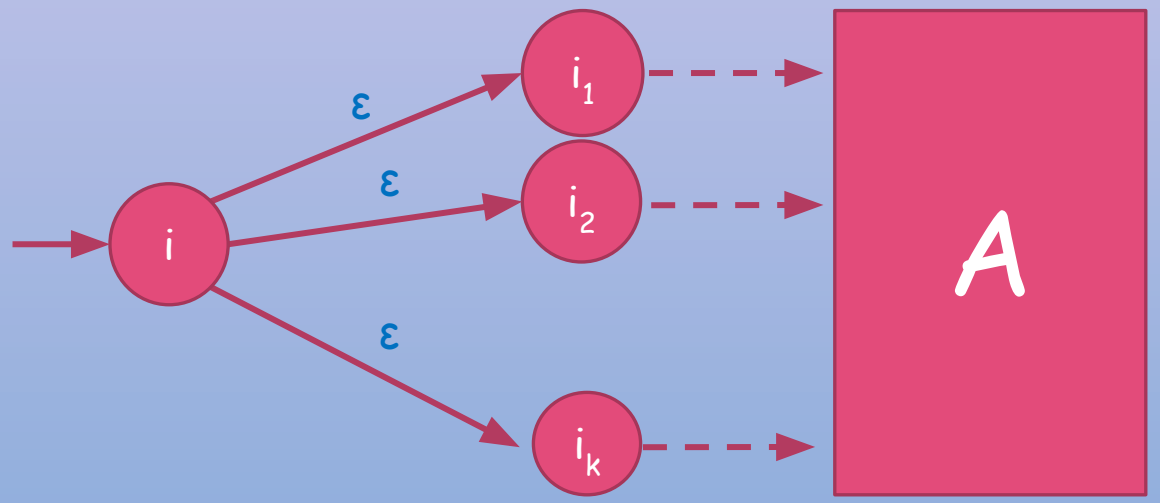
Un'alternativa è costruire un automa deterministico equivalente a quello ottenuto mediante le espressioni regolari.

Tale operazione in generale può essere molto costosa in termini di spazio e di tempo, ma potrebbe essere conveniente avere un automa deterministico per molti lookup, come nel caso degli analizzatori lessicali

DA NFA A DFA

UNIFICAZIONE DEGLI STATI INIZIALI

Per avere un NFA equivalente con un solo stato iniziale, basta creare uno stato nuovo, renderlo iniziale e associarlo ai vecchi stati iniziali mediante ε-transizioni

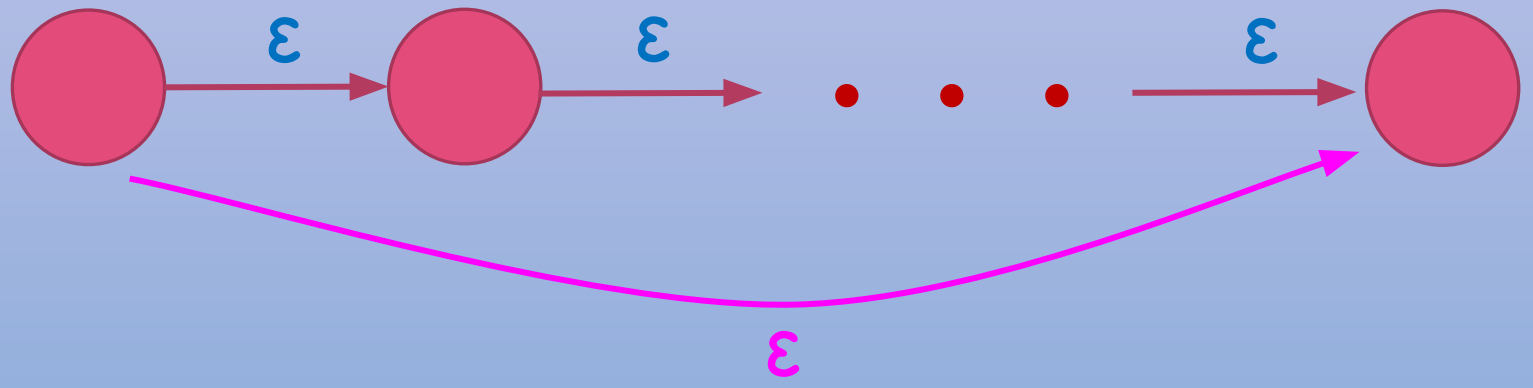


DA NFA A DFA

ELIMINAZIONE DELLE ε-TRANSIZIONI

Per passare da NFA al DFA bisogna prima di tutto applicare delle trasformazioni per cui l'NFA è trasformato in uno equivalente che non contiene E-TRANSIZIONI.

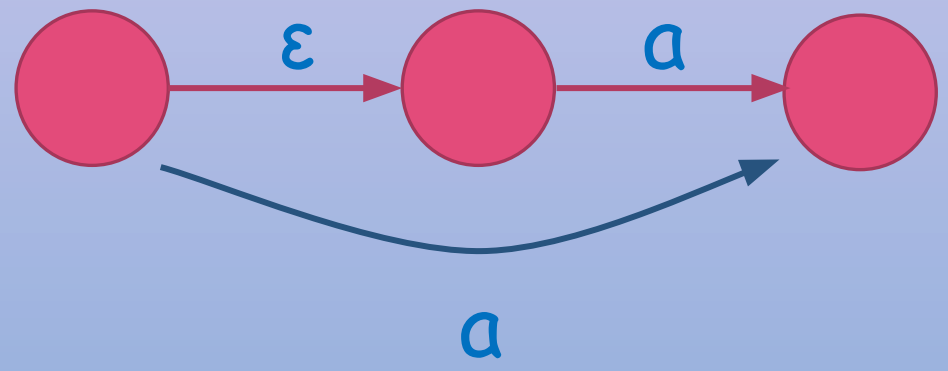
PRIMO PASSO: Chiusura transitiva degli ε-cammini



DA NFA A DFA

ELIMINAZIONE DELLE ε-TRANSIZIONI

SECONDO PASSO: Retropropagazione delle mosse di lettura sugli ε-archi

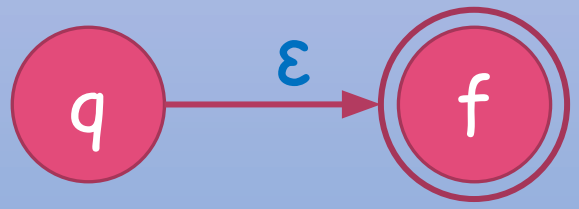


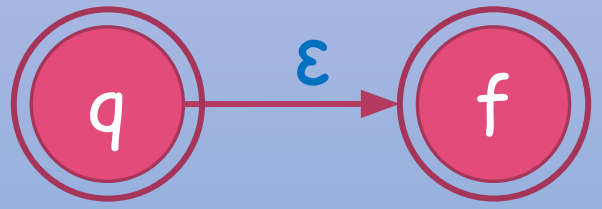
DA NFA A DFA

ELIMINAZIONE DELLE ε-TRANSIZIONI

TERZO PASSO: Costruzione dell'insieme dei nuovi stati finali

$F := F \cup \{q \text{ in } Q \mid \text{c'è un arco da } q \text{ a } f \text{ etichettato con } \varepsilon \text{ e } f \in F\}$

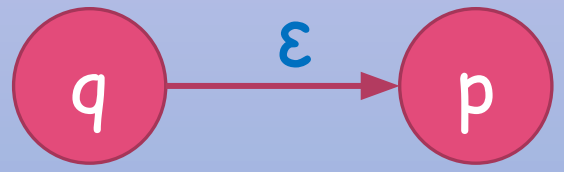




DA NFA A DFA

ELIMINAZIONE DELLE ε-TRANSIZIONI

QUARTO PASSO: Pulitura - cancellazione di tutti gli epsilon archi e gli stati inaccessibili dallo stato iniziale



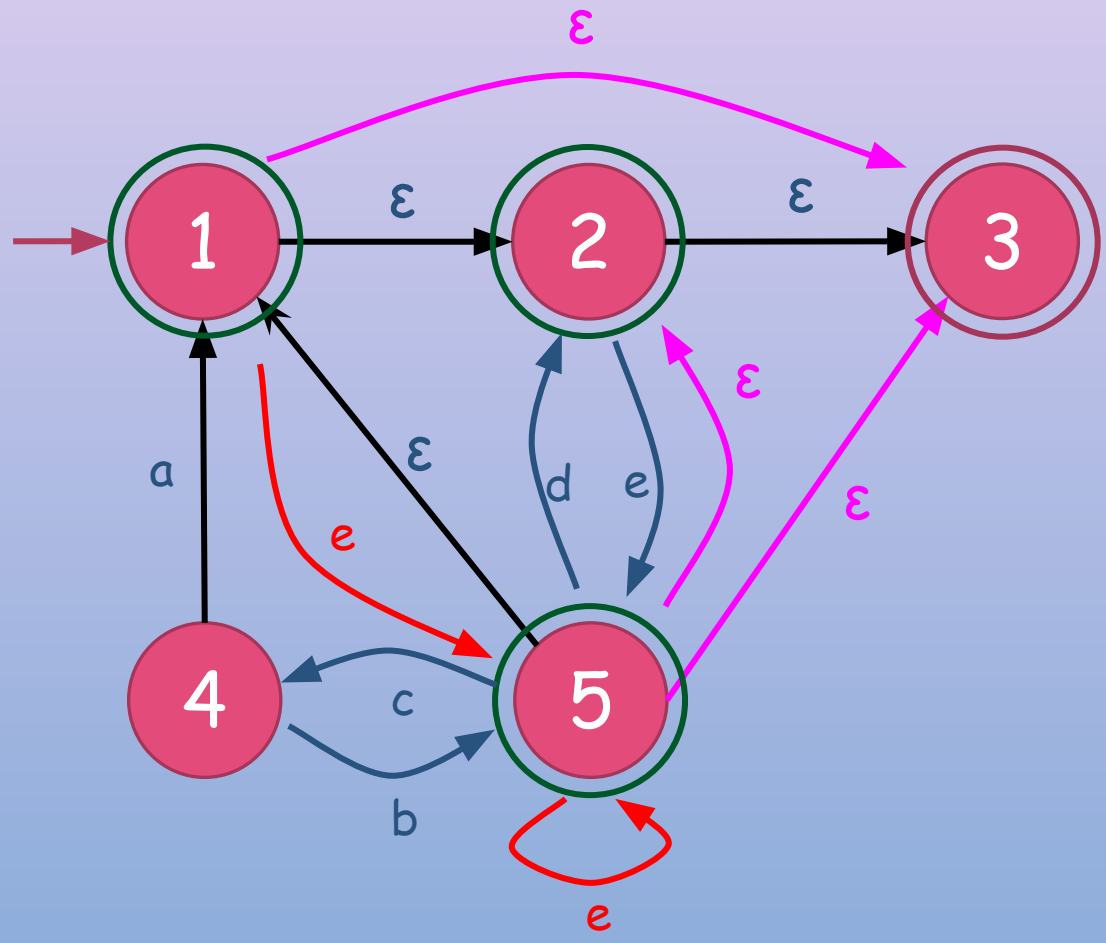
ESEMPIO

PRIMA FASE: chiusura transitiva degli epsilon cammini

SECONDA FASE: Retropropagazione delle mosse di lettura sugli ε-archi

TERZA FASE: Costruzione dell'insieme dei nuovi stati finali

QUARTA FASE:pulitura degli epsilon archi e degli stati non più accessibili



DA NFA A DFA
SUBSET CONSTRUCTION

Una volta eliminate le epsilon-transizioni l'unica causa di non determinismo sono le transizioni da uno stato che con una certa etichetta conducono a più stati diversi. Questo non determinismo si elimina mediante un algoritmo detto «**subset construction**»

In linea generale, l'idea è che gli stati del DFA sono tutti i sottoinsiemi dell'insieme degli stati dell'NFA. Esiste una transizione etichettata con a da un sottoinsieme S a un sottoinsieme S' e l'insieme delle transizioni da stati in S alla lettura di a coincide con l'insieme S'

PARTI FINITE RAGGIUNGIBILI

La subset construction genera spesso molti stati inaccessibili dallo stato iniziale o non co-accessibili, ossia da cui non si giunge allo stato finale. Questi stati sono superflui e vanno eliminati.

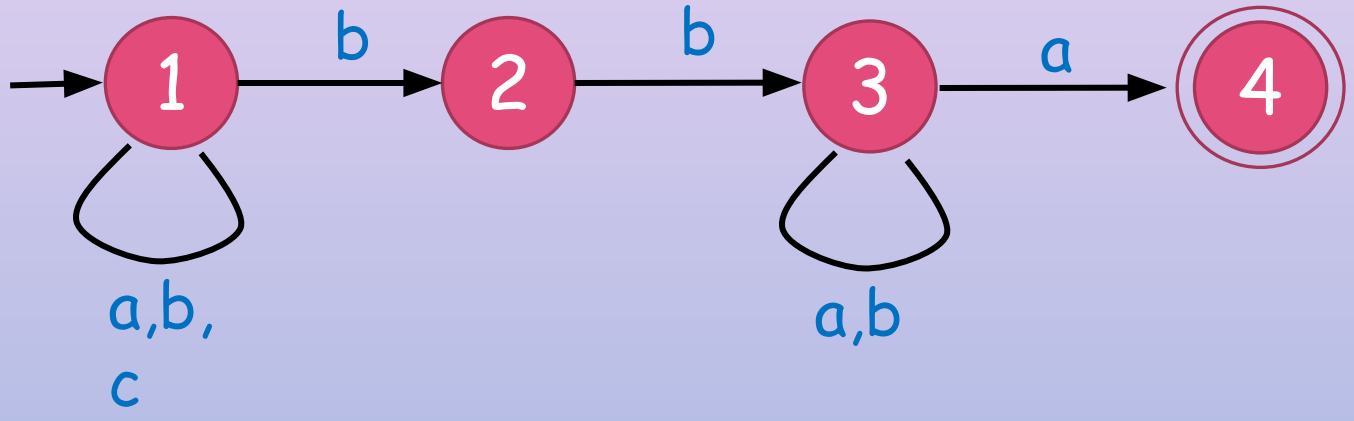
L'algoritmo generalmente utilizzato evita la costruzione di questi stati non accessibili e non co-accessibili, e costruisce le parti finite raggiungibili.

L'algoritmo consiste nell'unificare in un unico stato, a partire dallo stato iniziale, tutti gli stati di arrivo da uno stesso stato con la stessa transizione. Questa costruzione si propaga su tutto il grafo degli stati.

Inoltre gli stati finali del nuovo automa sono in corrispondenza di quegli stati che «contengono» almeno uno dei vecchi stati finali.

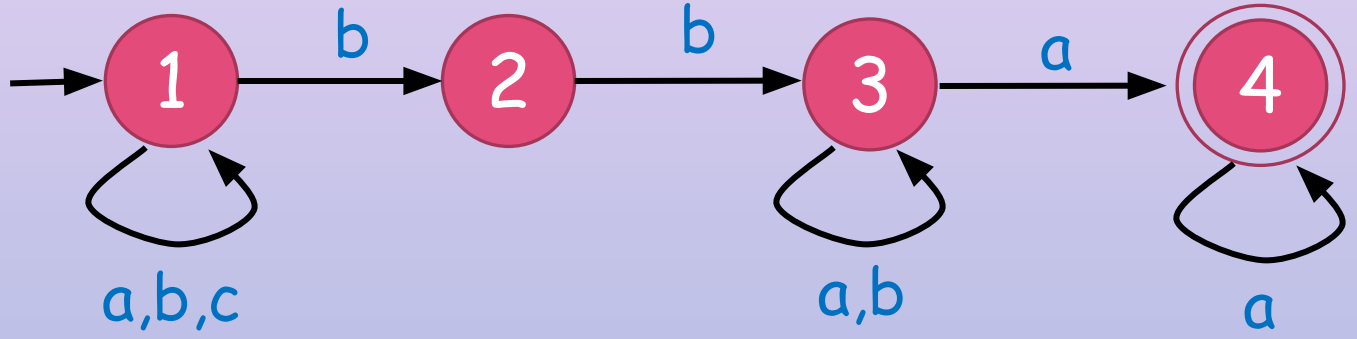
DA NFA A DFA

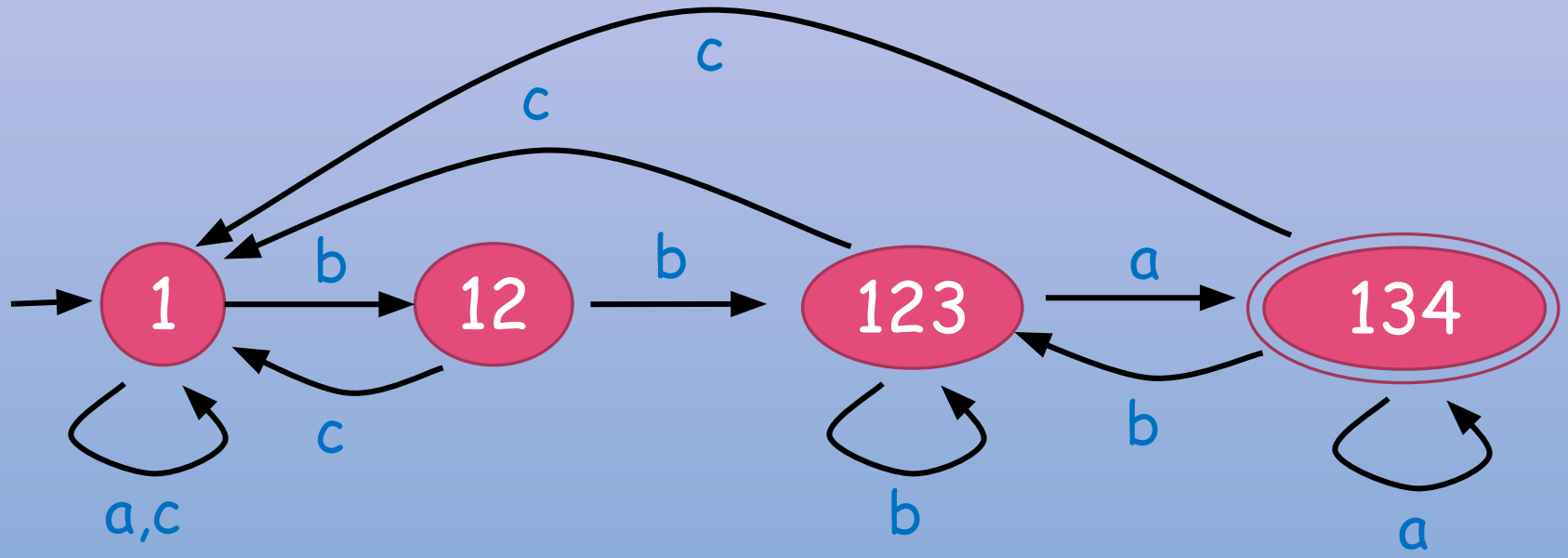
SUBSET CONSTRUCTION



DA NFA A DFA

SUBSET CONSTRUCTION





NFA VERSUS DFA

Osservazione: Il DFA ottenuto da un NFA mediante la subset construction ha il difetto di poter essere in generale molto più grande come numero di stati.

In teoria il numero di stati del DFA potrebbe essere esponenzialmente più grande di quello del NFA, questo perché tutti i possibili sottoinsiemi sono 2^n se n è il numero di stati dell'NFA.

Si dimostra che esistono automi non deterministici per cui si ha effettivamente l'esplosione esponenziale.

Esempio

$L=\{a,b\}^*a\{a,b\}^n$.

Ogni DFA per L ha 2^n stati, mentre l'NFA ne ha n

USARE NFA O DFA?

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

1. costruire l'NFA da r in $O(|r|)$ tempo. Infatti il numero di stati è proporzionale ad $|r|$ e ci sono al più k transizioni per ogni stato, dove k è la cardinalità dell'alfabeto (quindi $k|r|$) quindi sia il numero di stati che il numero delle transizioni è $O(|r|)$. La tabella di transizione viene memorizzata in $O(|r|)$ spazio. Il riconoscimento di una stringa w mediante un NFA con $|r|$ stati può essere simulato efficientemente mediante due stack in $O(|r| \times |w|)$.
2. Costruire il DFA dall'NFA applicando la subset construction ed eventualmente la minimizzazione ($O(n \log n)$). 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 $O(|r|^3)$. Il riconoscimento di una stringa w mediante il DFA può essere simulato in $O(|w|)$. Lo spazio può diventare $O(2^{|r|})$.

SINTESI DEI COSTI

r espressione regolare
w parola da riconoscere

Automa	Spazio	Tempo di costruzione	Riconoscimento di una stringa
NFA	$O(\|r\|)$	$O(\|r\|)$	$O(\|r\| \times \|w\|)$
DFA caso medio	$O(\|r\|)$	$O(\|r^3\|)$	$O(w)$
DFA caso pessimo	$O(2^{\|r\|})$	$O(2^{\|r\|})$	$O(w)$

COME SCEGLIERE?

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 che deve essere cercata in un testo, 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 (Algoritmo di Berry e Sethi). In alcuni casi si ottengono DFA con minor numero di stati.
- Oppure si può minimizzare il NFA ottenuto per giungere al DFA minimale. Il tempo per la minimizzazione è $O(n \log n)$
- rappresentazione compatta delle tabelle di transizione.

RICONOSCIMENTO DI TOKEN

L'analizzatore lessicale deve individuare i lessemi verificando quali hanno un match con i pattern specificati dai token. Questa operazione si chiama Pattern Matching.

Nel riconoscimento dei token ci si potrebbe però trovare in una condizione di ambiguità per cui una parola riconosciuta potrebbe corrispondere a due diverse «categorie».

Per esempio la parola chiave «if» si potrebbe anche pensare come un identificatore, poiché ne soddisfa le regole. Questa sequenza verrebbe riconosciuta da due diversi degli automi corrispondenti ai token. Come si risolve il problema?

LONGEST BEST MATCH

Supponiamo di avere definito dei pattern p_i con le relative azioni $\{a_i\}$

p_1 $\{a_1\}$

p_2 $\{a_2\}$

...

p_n $\{a_n\}$

Vogliamo una procedura che ad ogni passo individua il prefisso più lungo che ha un match con uno dei pattern p_i, ossia il longest best match. «Best» perché nell'eventualità di un doppio riconoscimento devo dare priorità a una scelta piuttosto che ad un'altra, che in genere corrisponde all'ordine in cui figurano i pattern nella lista

Come si realizza il riconoscimento del longest best match e la regola di priorità?

Cominciamo con gli NFA:

LONGEST BEST MATCH

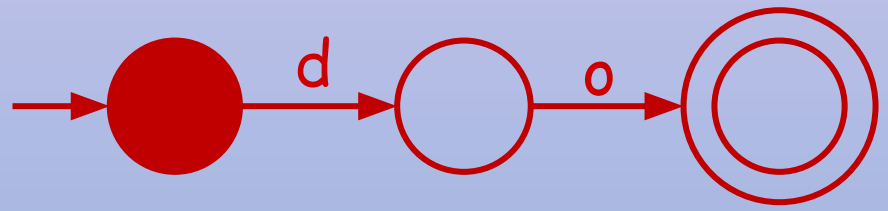
Supponiamo di avere i seguenti token

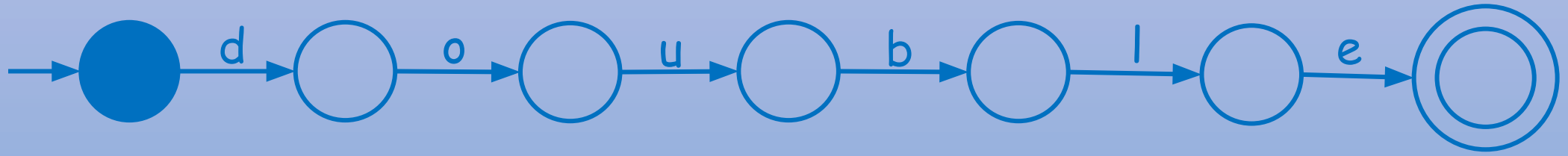
T_do do

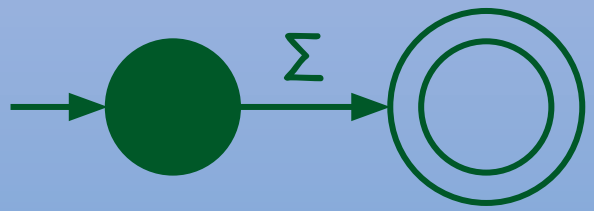
T_double double

T_identif [a-zA-Z]

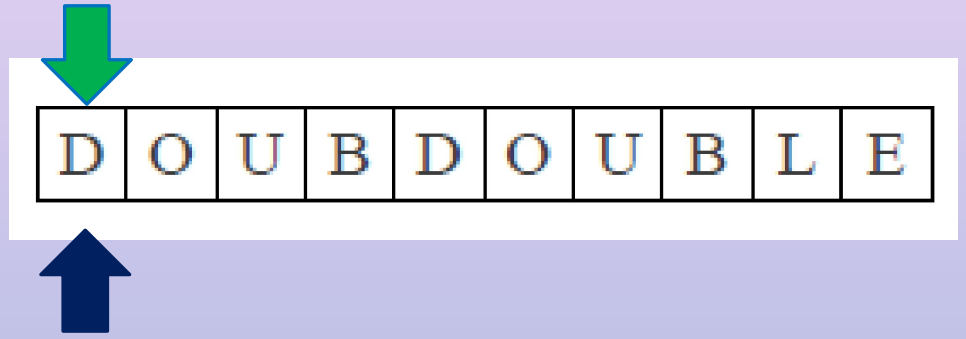
D	O	U	B	D	O	U	B	L	E

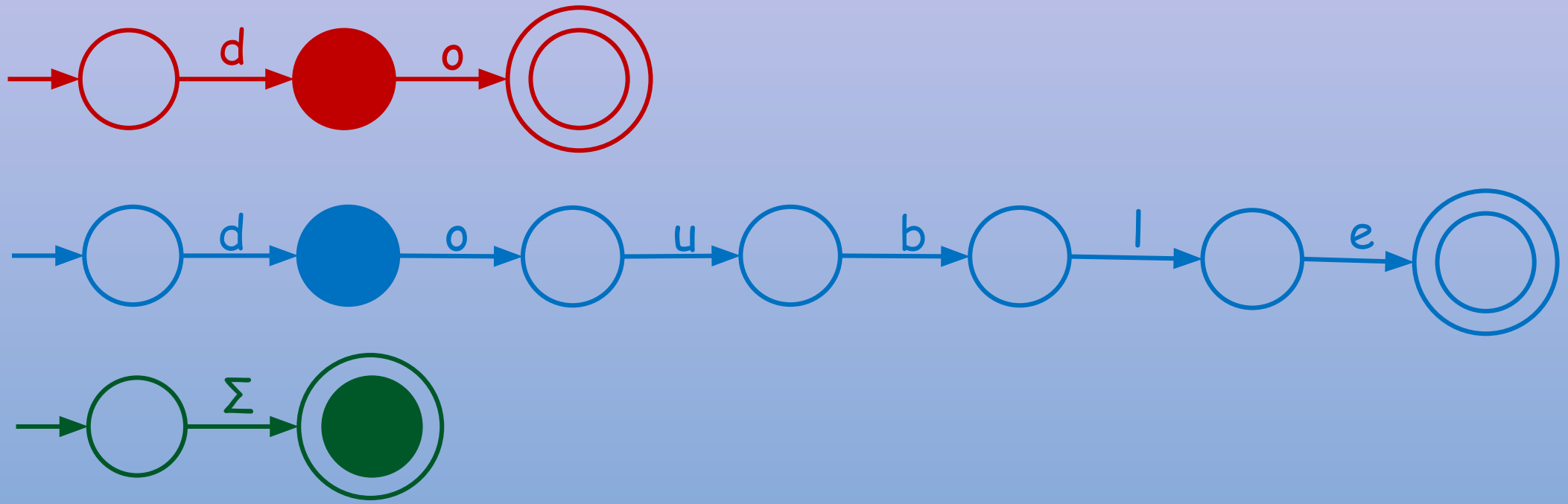




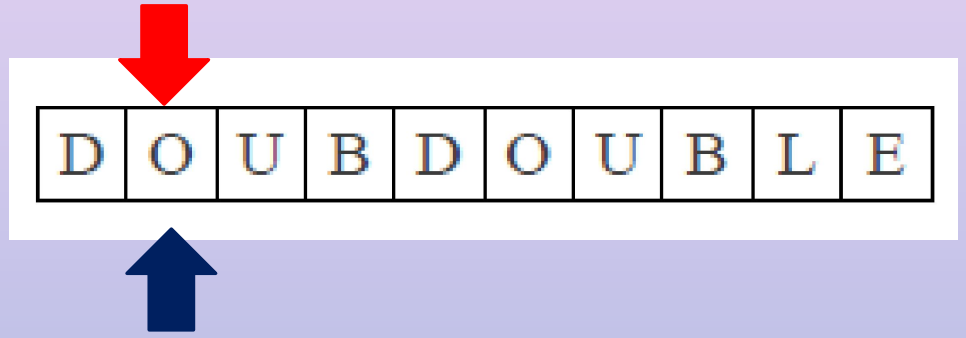


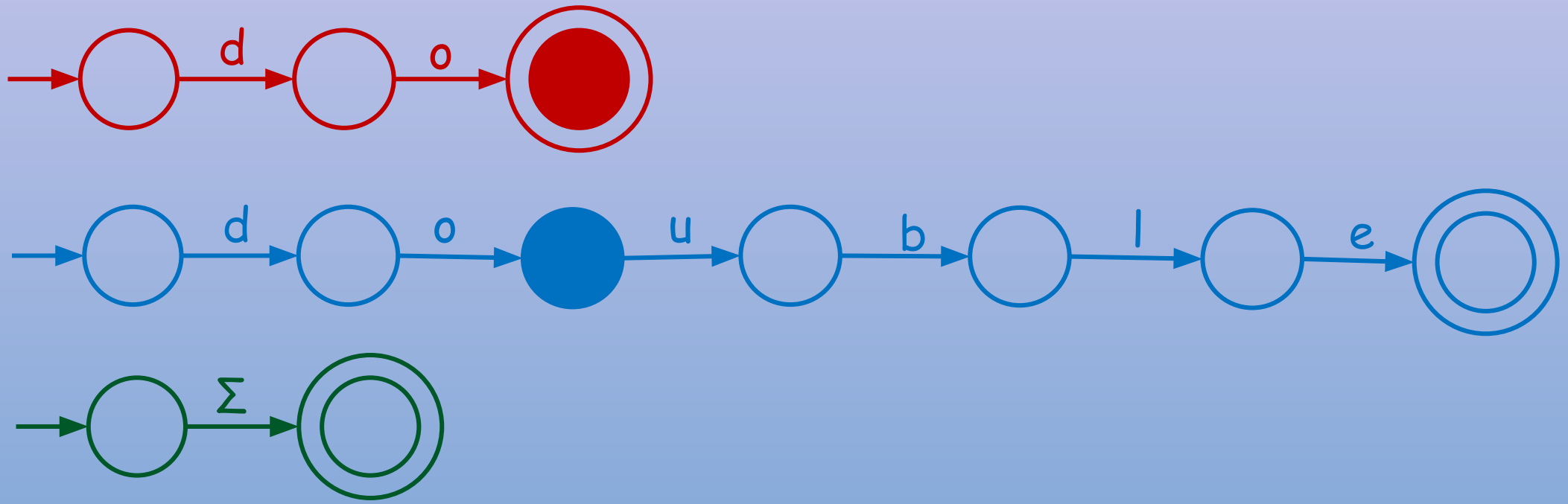
PIÙ IN DETTAGLIO…



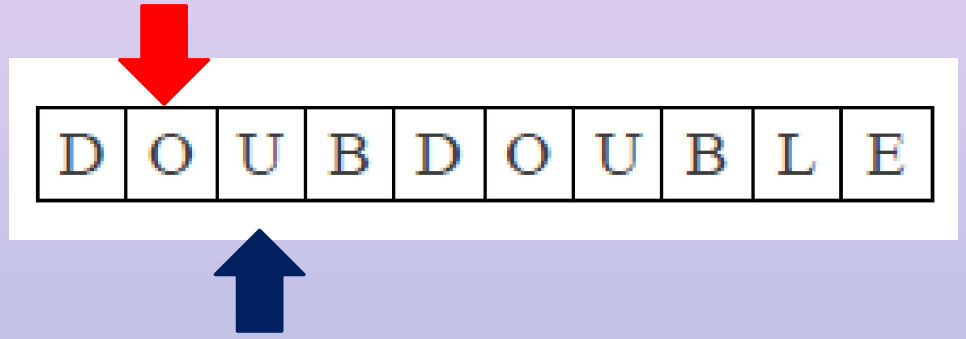


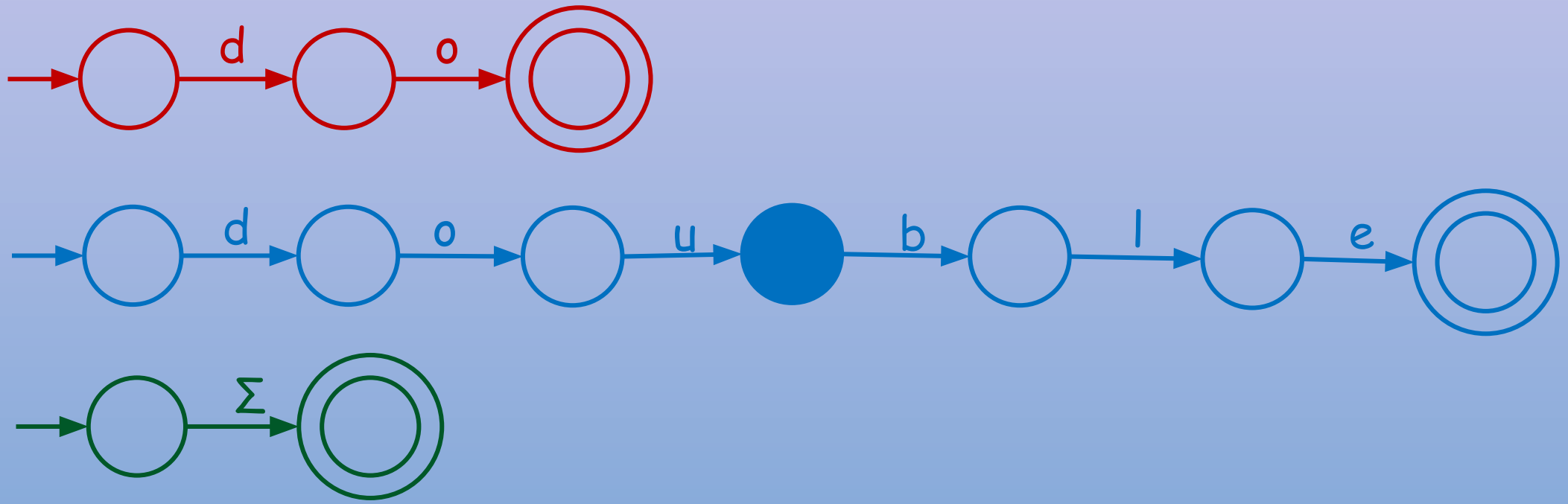
PIÙ IN DETTAGLIO…



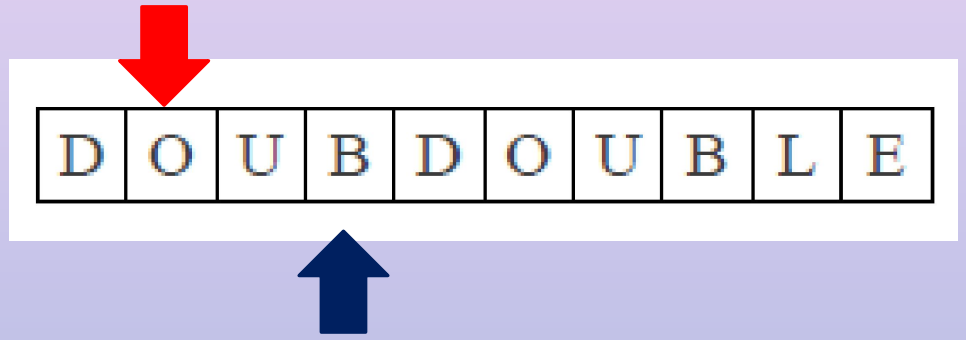


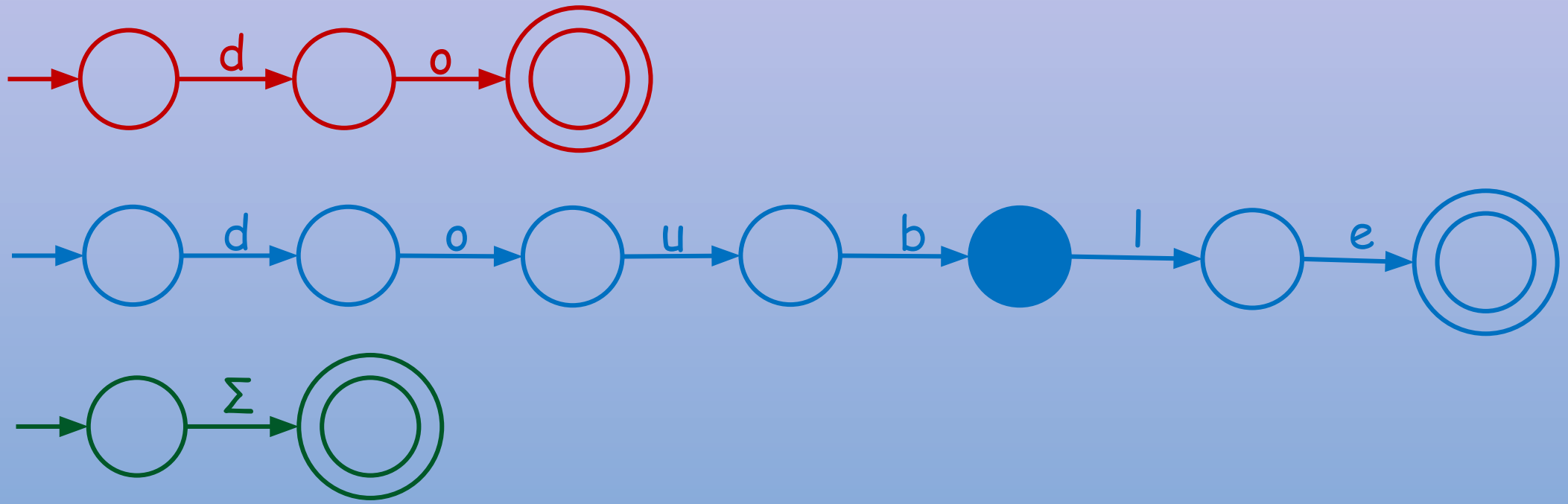
PIÙ IN DETTAGLIO…



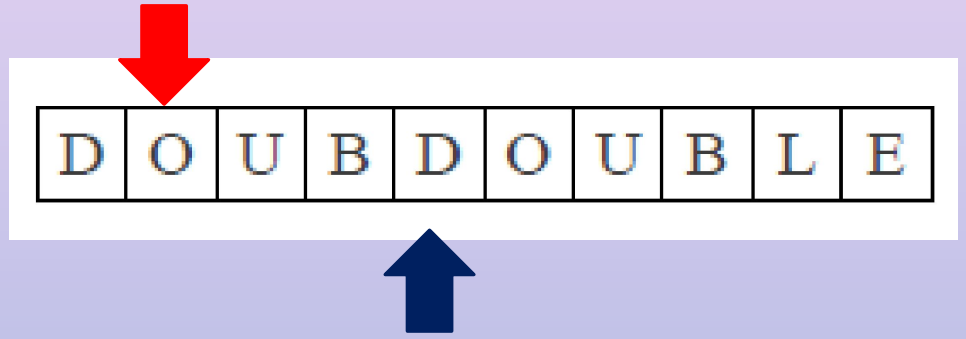


PIÙ IN DETTAGLIO…

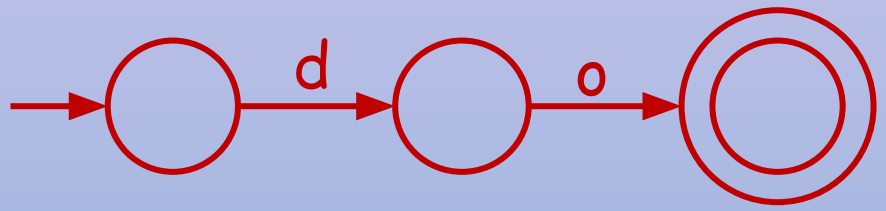


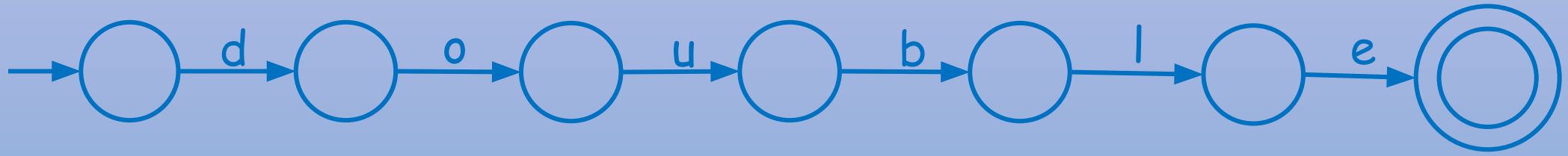


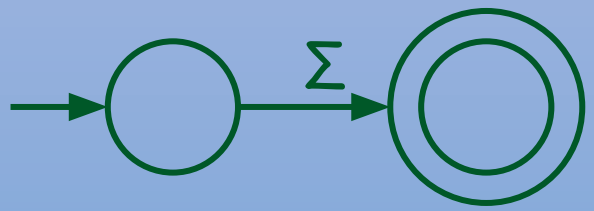
PIÙ IN DETTAGLIO…



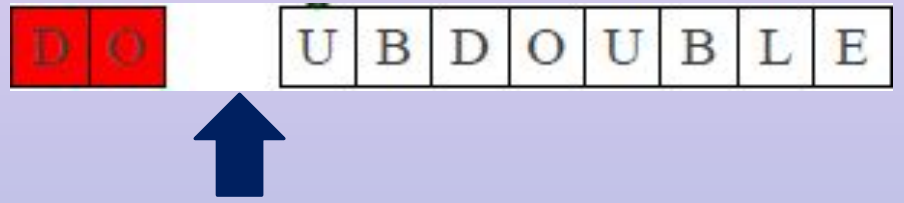
Non corrisponde a nessun match.
Quindi si considera l'ultimo lessema riconosciuto DO



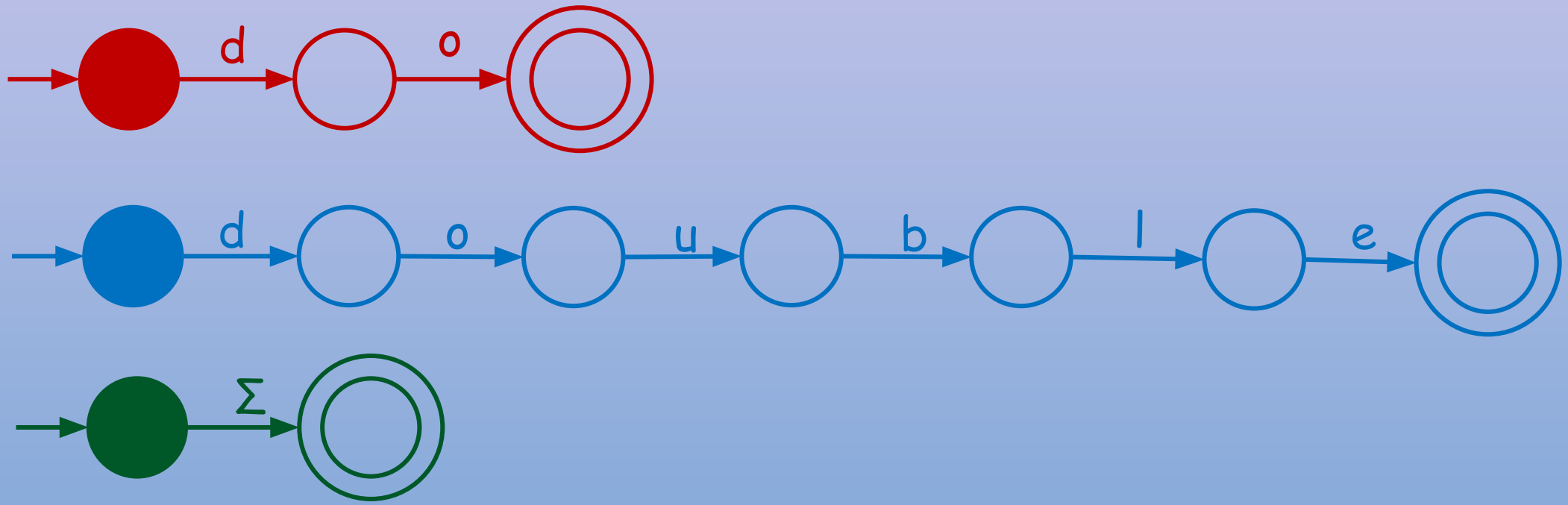




PIÙ IN DETTAGLIO…



Ripartiamo da zero



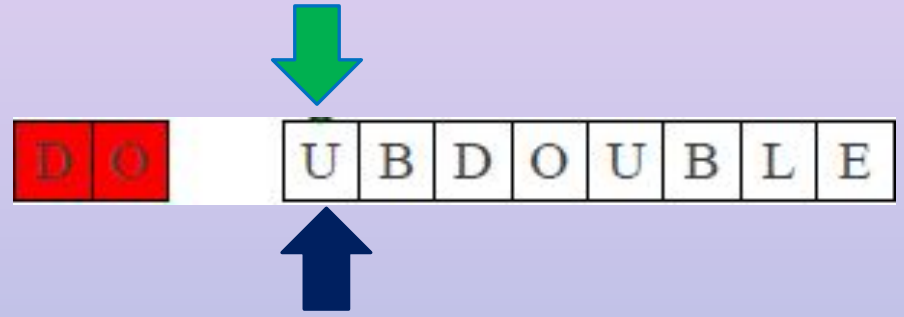
PIÙ IN DETTAGLIO…

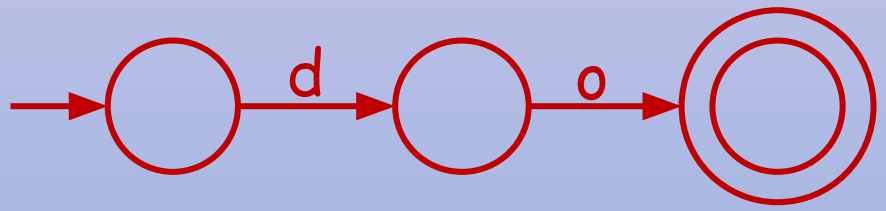
Supponiamo di avere i seguenti token

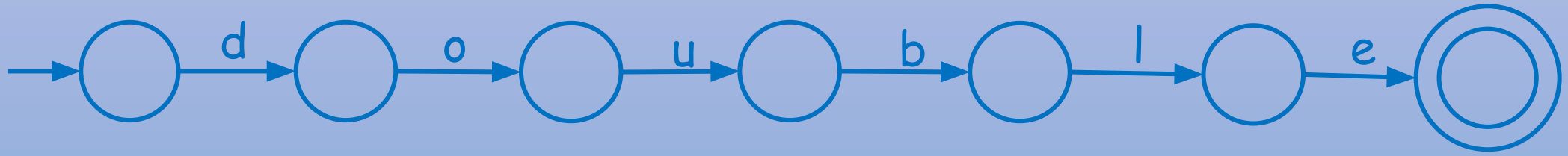
T_do do

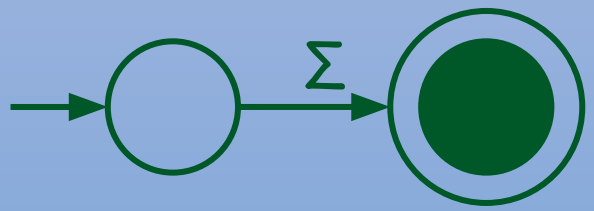
T_double double

T_identif [a-za-z]

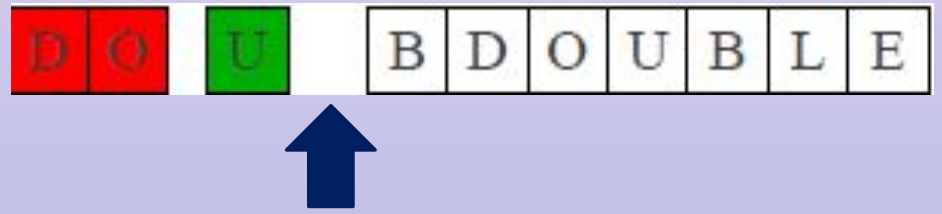


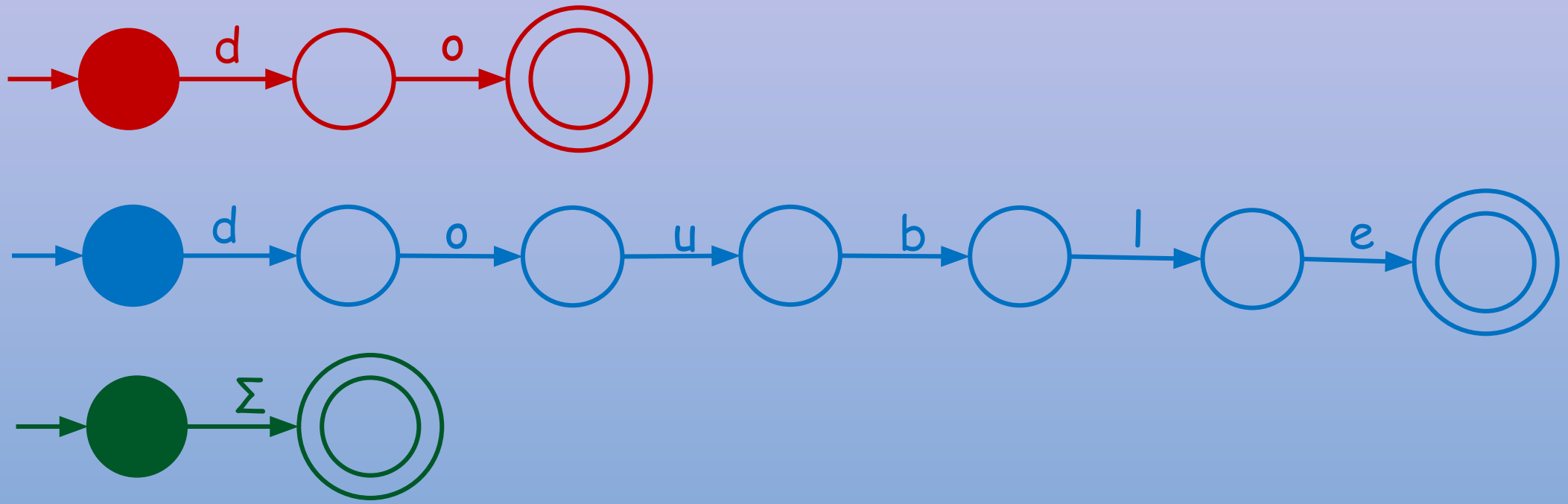




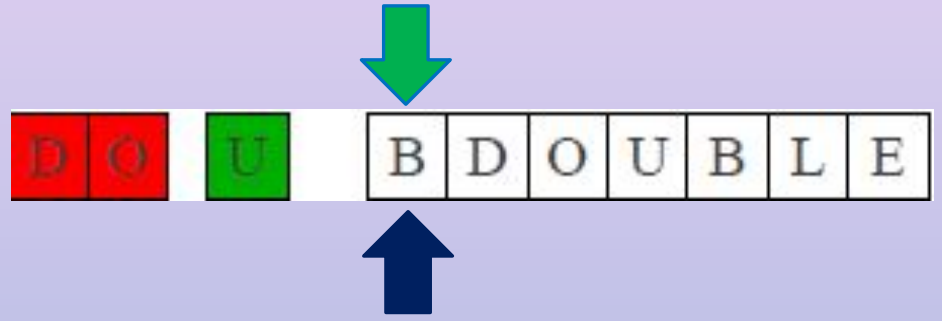


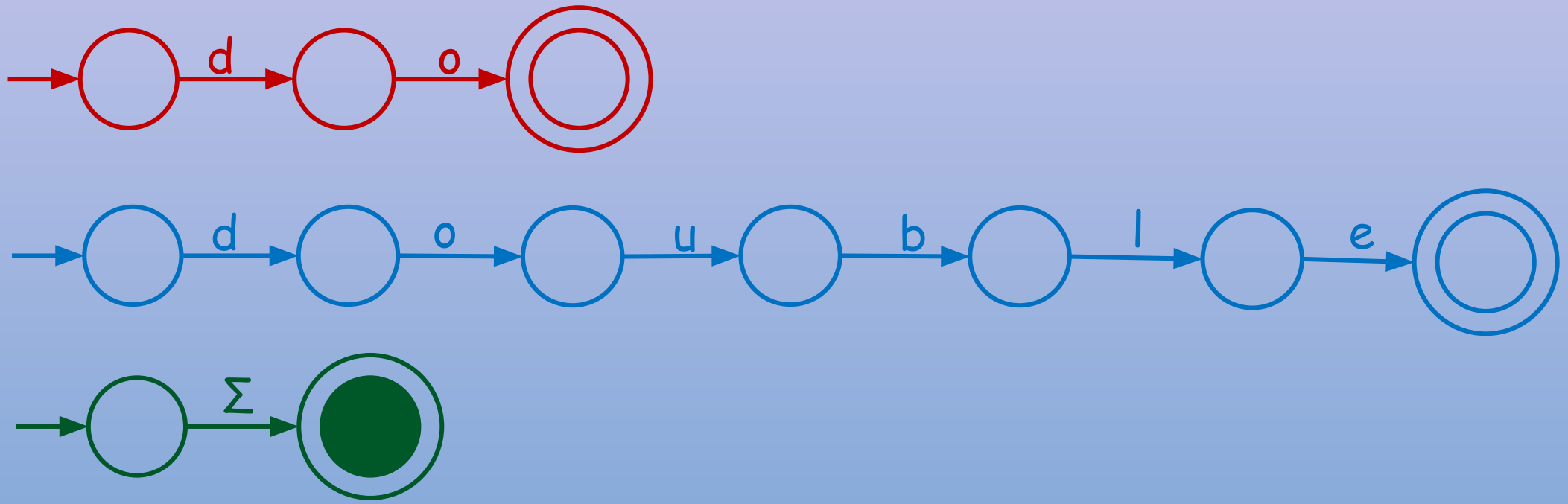
PIÙ IN DETTAGLIO…



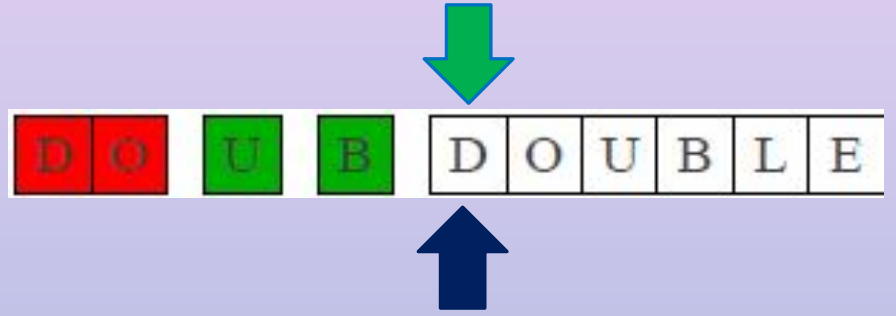


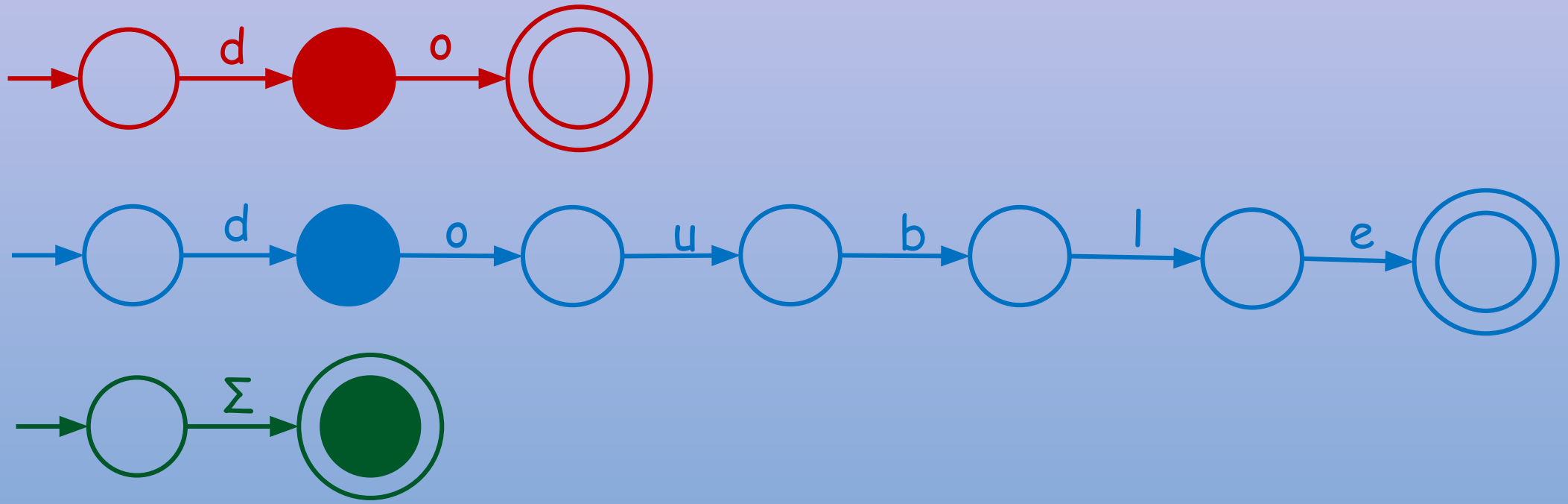
PIÙ IN DETTAGLIO…



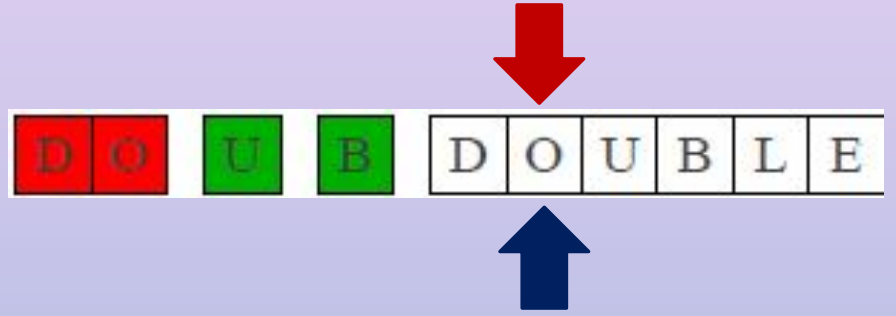


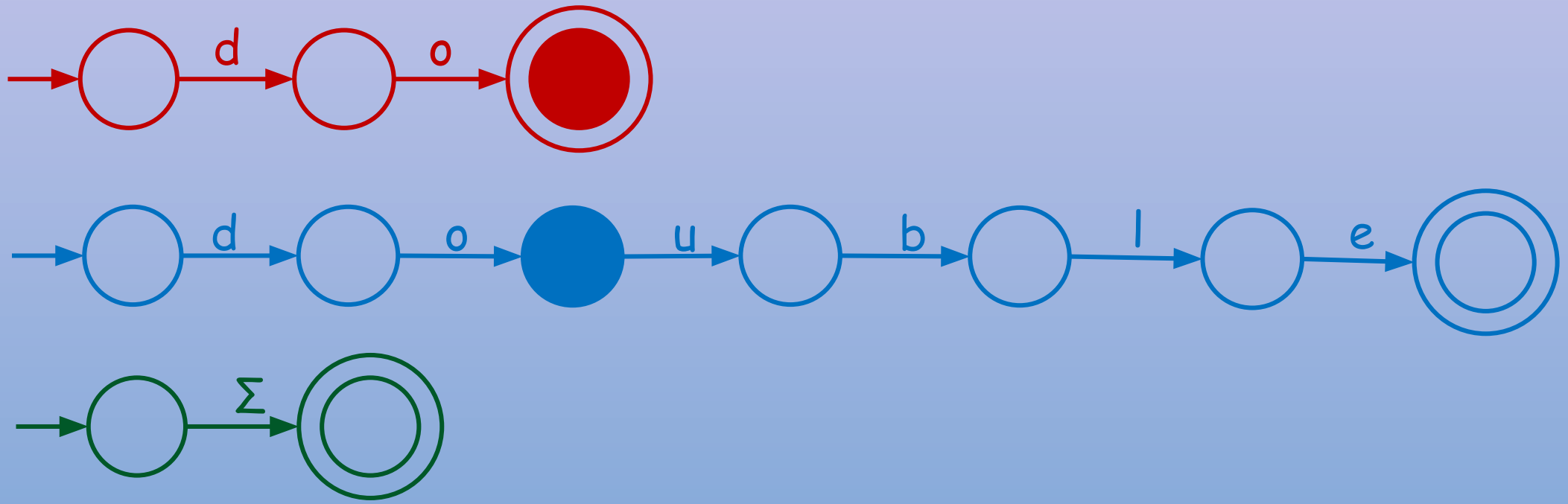
PIÙ IN DETTAGLIO…



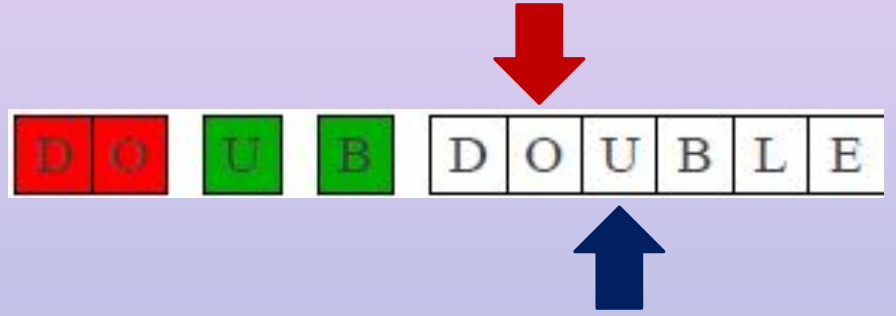


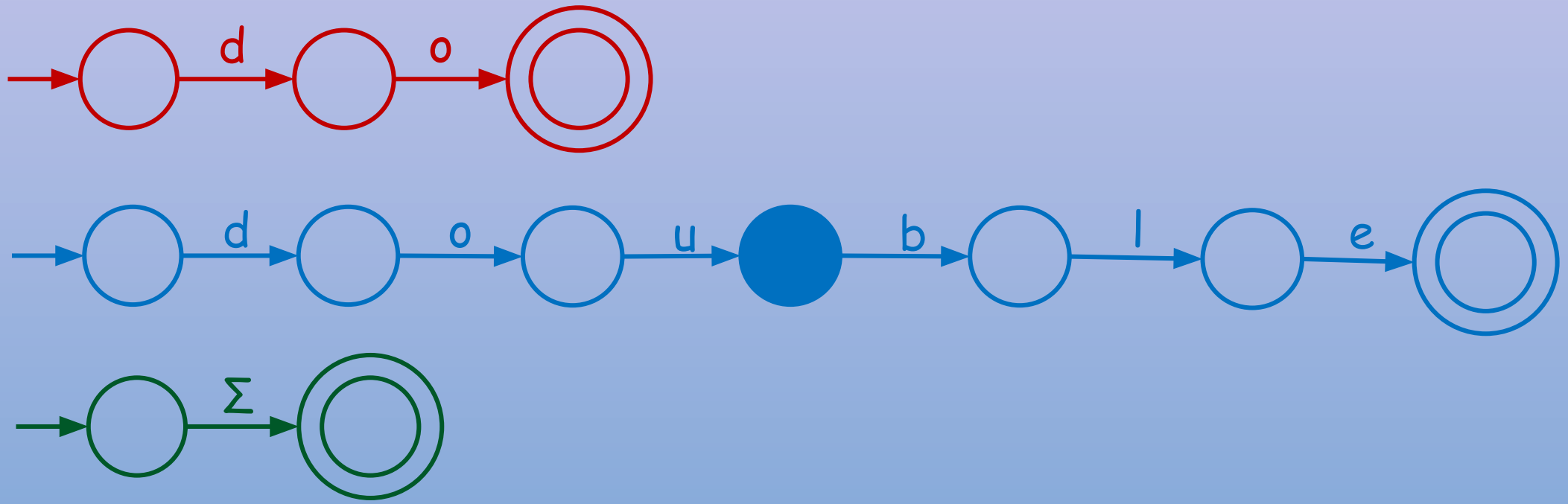
PIÙ IN DETTAGLIO…



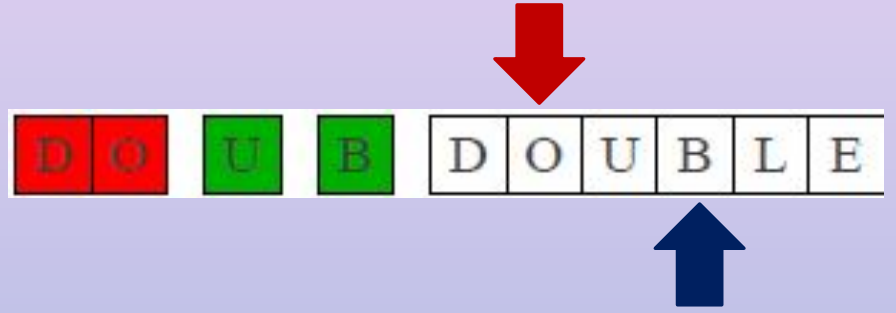


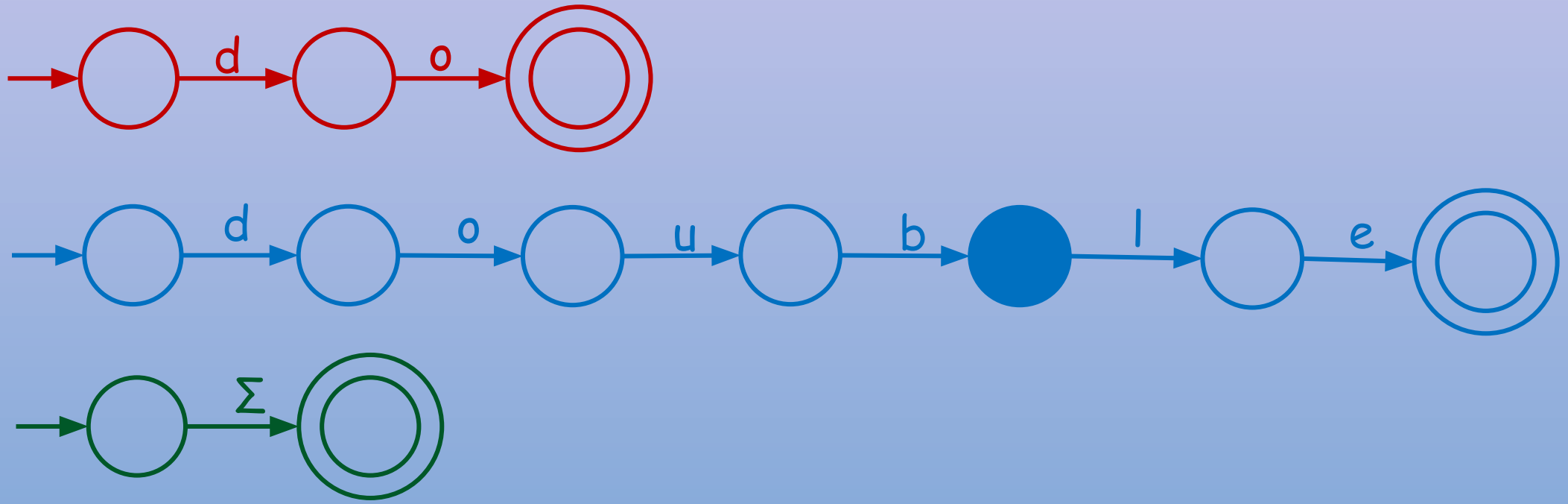
PIÙ IN DETTAGLIO…



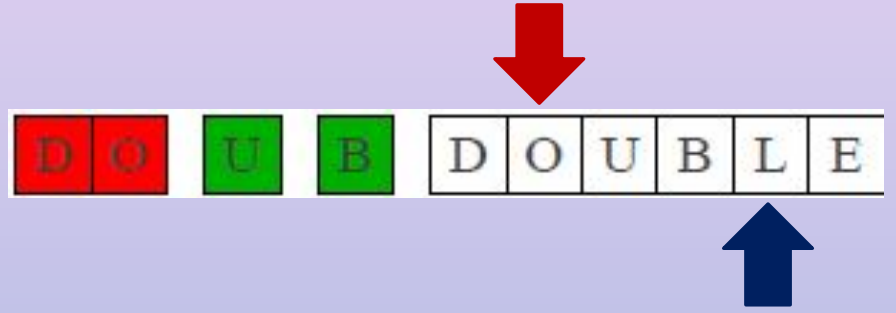


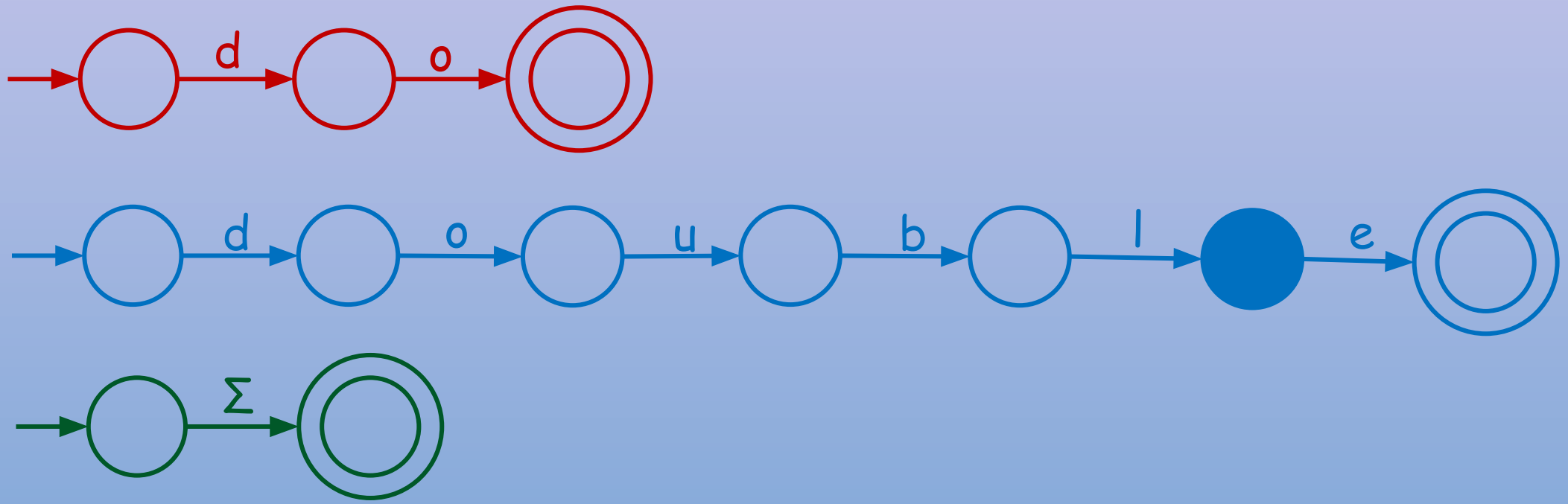
PIÙ IN DETTAGLIO…



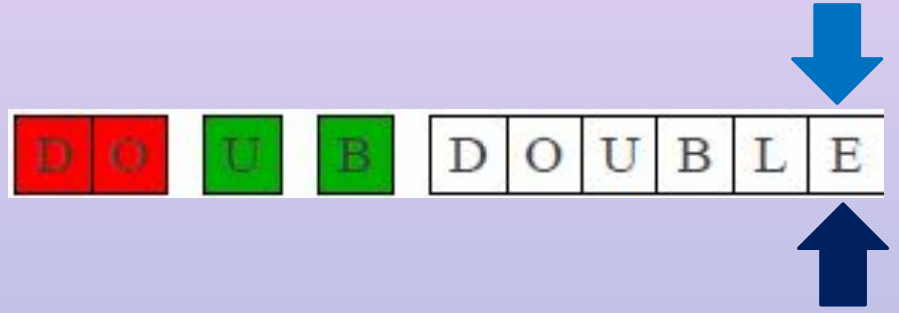


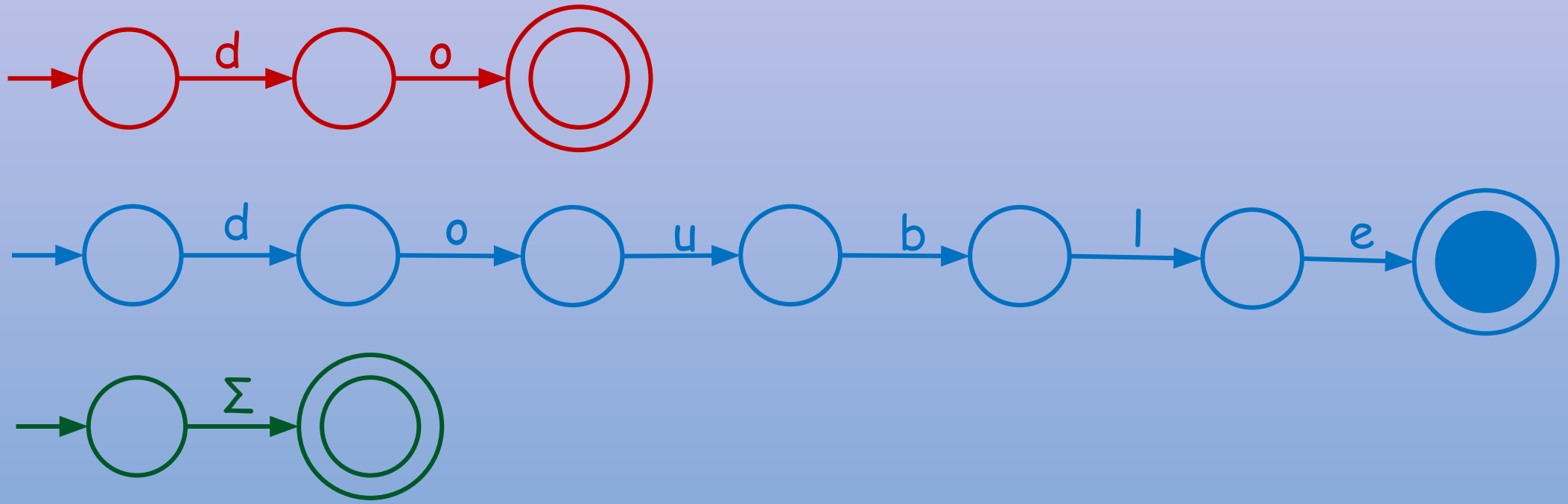
PIÙ IN DETTAGLIO…





PIÙ IN DETTAGLIO…





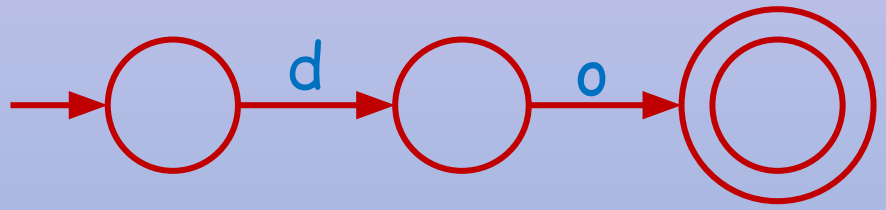
PIÙ IN DETTAGLIO…

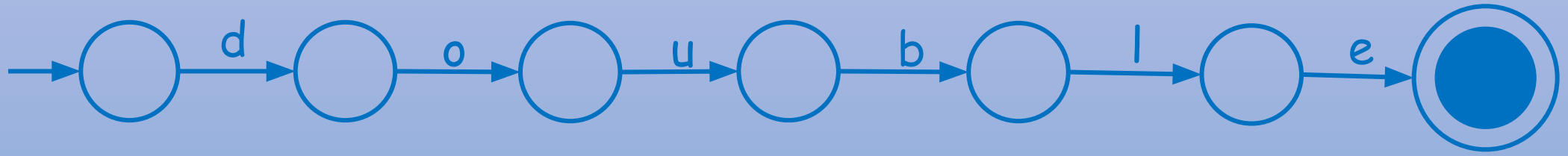
Supponiamo di avere i seguenti token

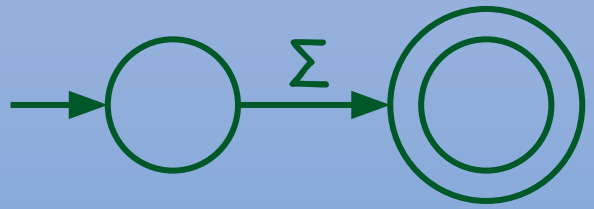
T_do do

T_double double

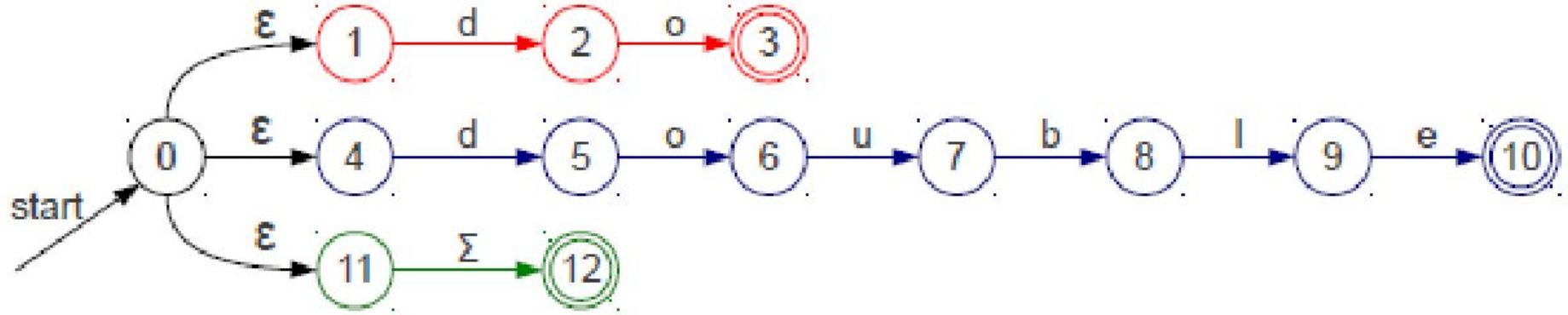
T_identif [a-za-z]

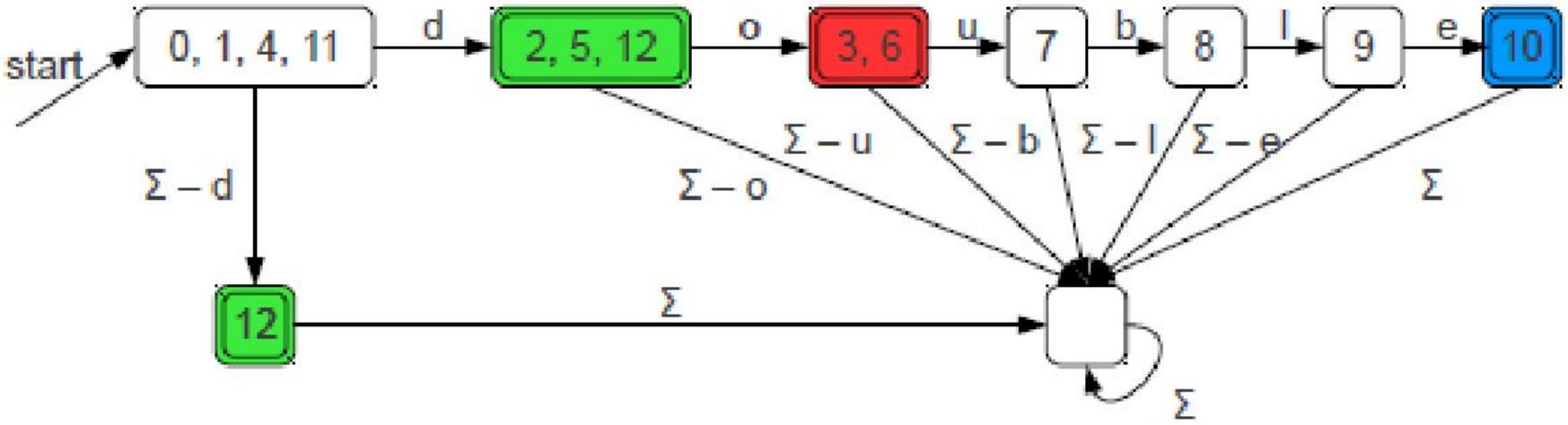






IL PASSAGGIO AL DFA



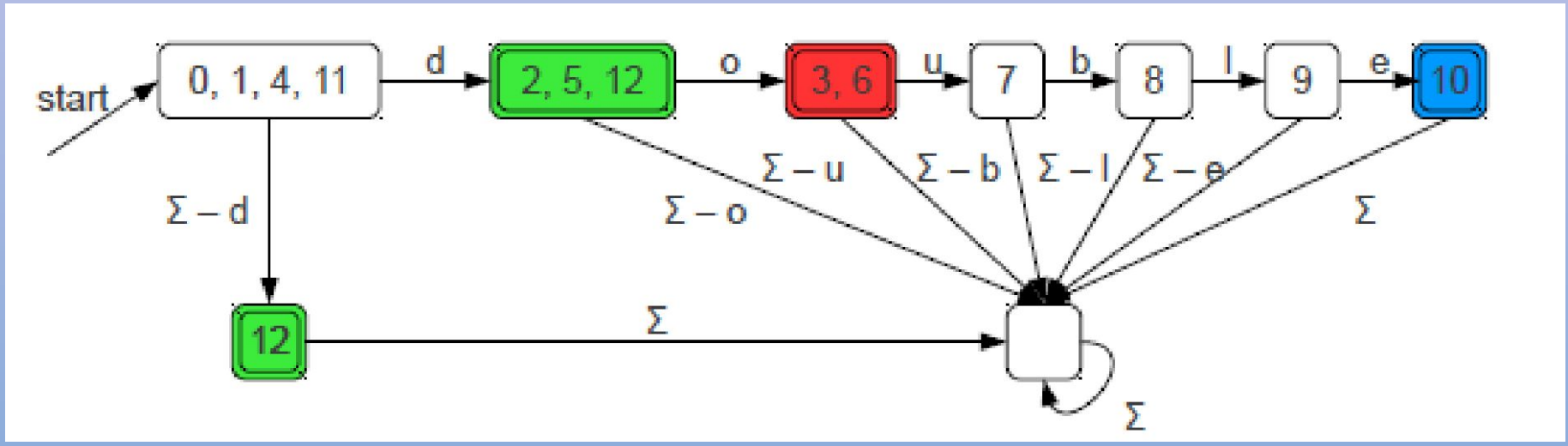


E' possibile unificare i tre automi per costruirne uno (non deterministico)che riconosca l'unione.

Si applica la subset construction e in più nel DFA risultante ogni stato che contiene stati accettanti (che diventa accettante), conterrà anche l'informazione sul NFA di provenienza (colore)

COME PROCEDERE COL DFA

- Se in uno stato ci fossero più stati accettanti verrebbe definito un ordine di priorità tra questi.
- L'automa legge il testo un carattere per volta e mantiene memoria gli stati di accettazione intermedi (per esempio è possibile memorizzarli in una pila). Appena si arriva in uno stato pozzo o non ci sono le transizioni uscenti, si cerca a ritroso l'ultimo stato di accettazione incontrato e si dà in output il token corrispondente.



ESERCIZIO

a

abb

a^*b^+

Costruire i relativi Dfa, comporli e ottenere l'automa deterministico

Vedere cosa succede sugli input aaba e abba