

*FLEX*

Giovedì 10 Ottobre

# FORMATO DI UN FILE IN FLEX

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

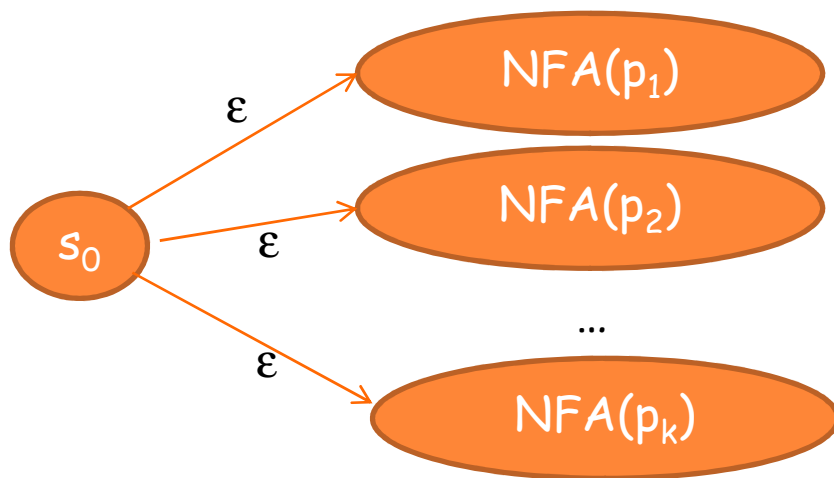
Anche nella sezione REGOLE è possibile inserire codice tra %{ e %}. Ciò deve avvenire prima della prima regola.

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

## COME FUNZIONA FLEX?

Si descrivono tutte le espressioni regolari che definiscono i token

Converte le espressioni regolari in automi non deterministici  
Combina tutti i NFAs in un unico NFA



# COME FUNZIONA FLEX?



Converte il NFA in un automa deterministico (DFA).

Ottimizza l'automa deterministico

Produce codice per simulare l'azione del DFA



COME COSTRUIRE IL DFA?  
E' SUFFICIENTE UNA SUBSET CONSTRUCTION?  
COME SI REALIZZA IL RICONOSCIMENTO  
DEL LONGEST BEST MATCH O LA REGOLA DI  
PRIORITÀ?

Partiamo dagli NFA...

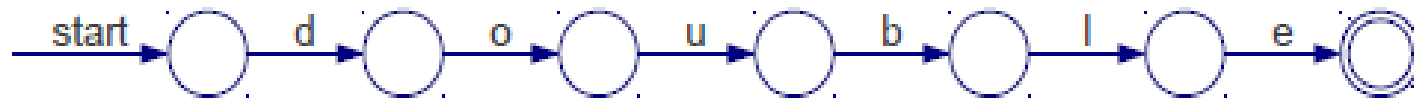
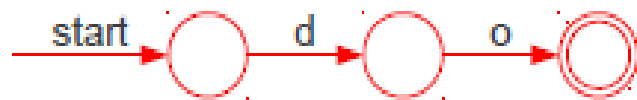


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]



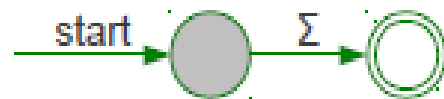
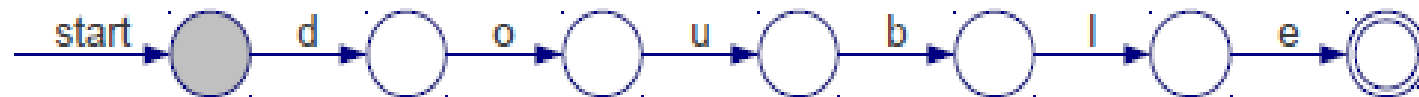
D	O	U	B	D	O	U	B	L	E
---	---	---	---	---	---	---	---	---	---

PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]



D	O	U	B	D	O	U	B	L	E
---	---	---	---	---	---	---	---	---	---

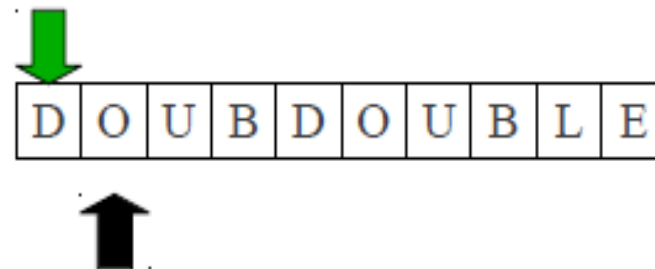
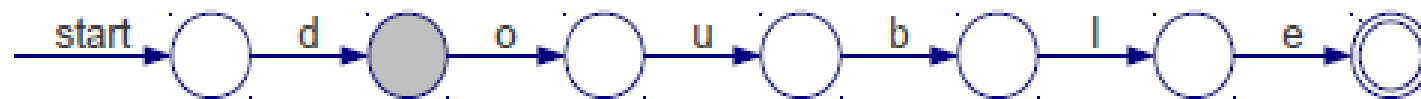


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]



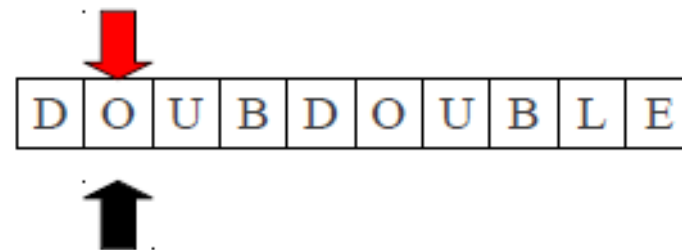
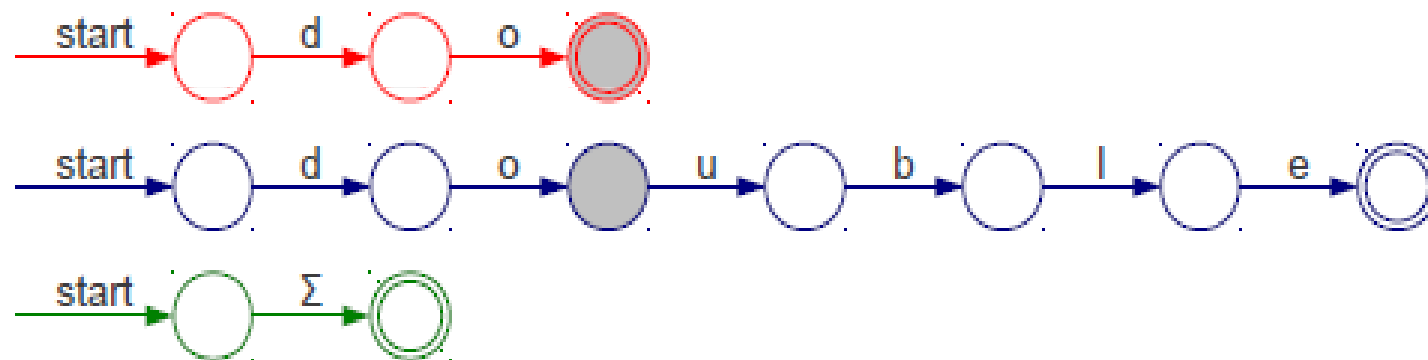


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]

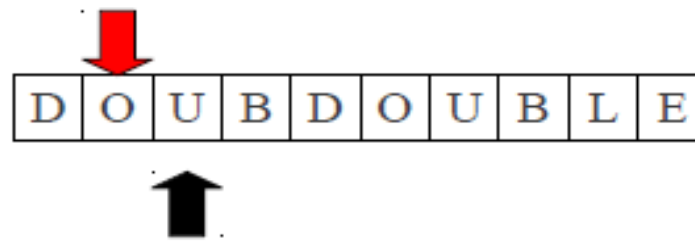
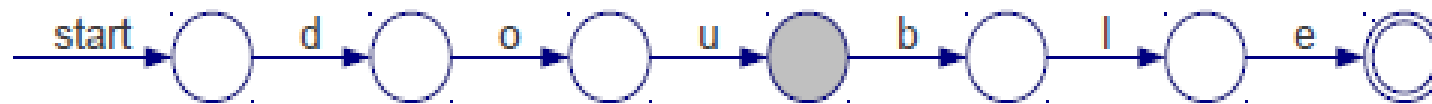


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]

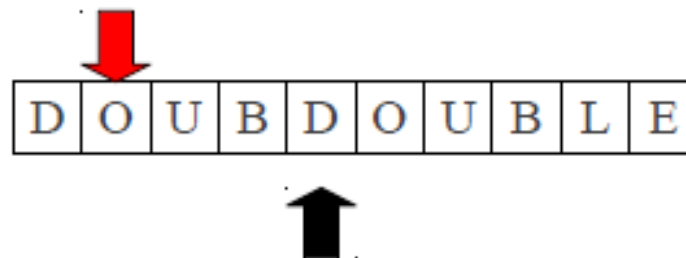
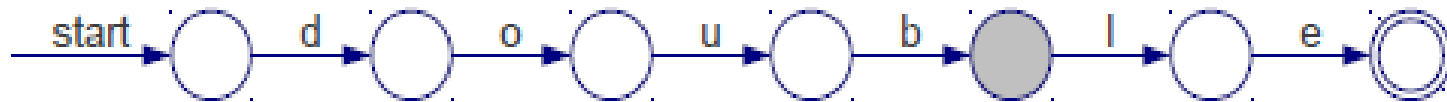


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]

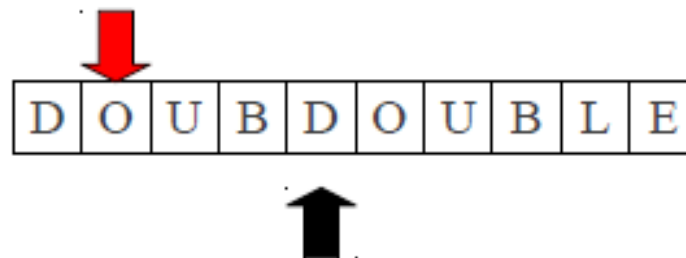
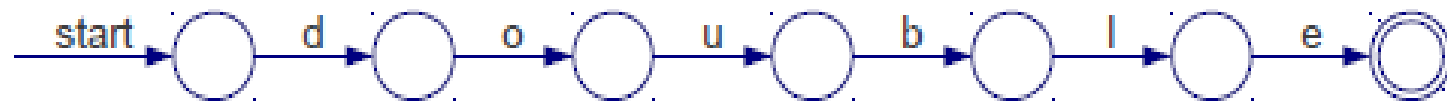


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]

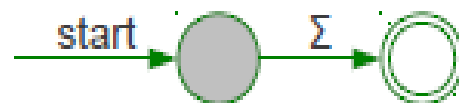
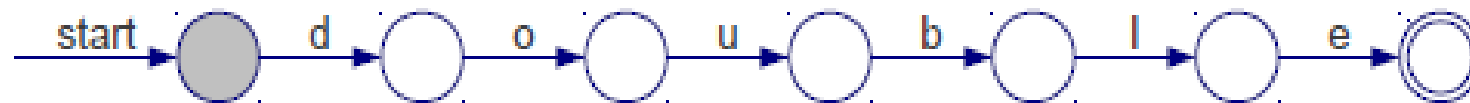


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]



D O

U B D O U B L E

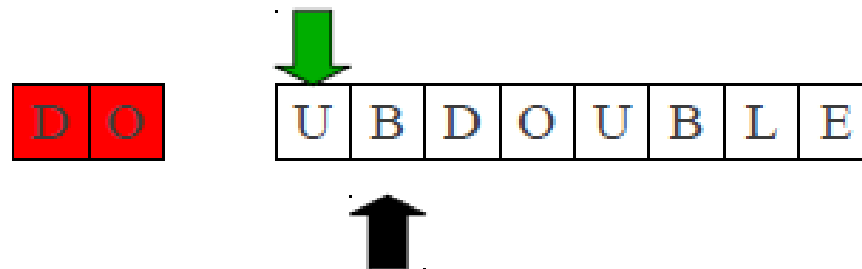
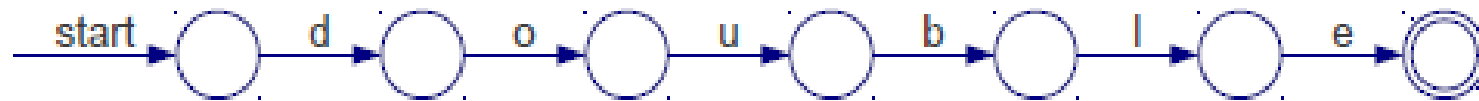


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]

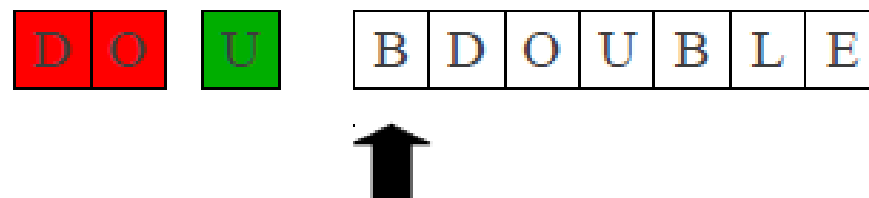
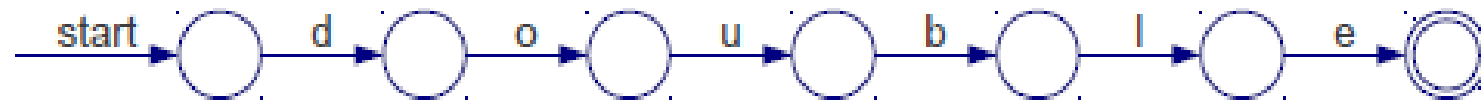


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]

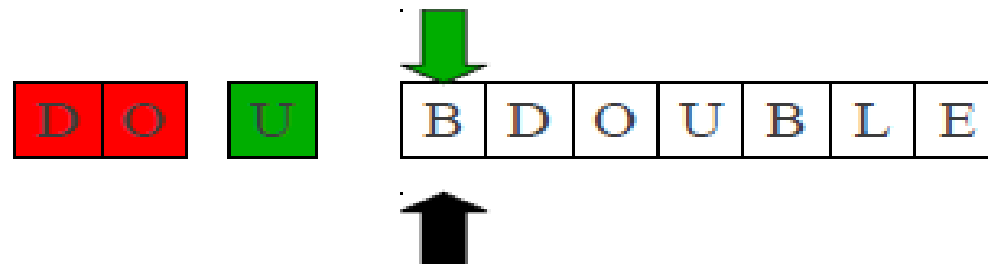
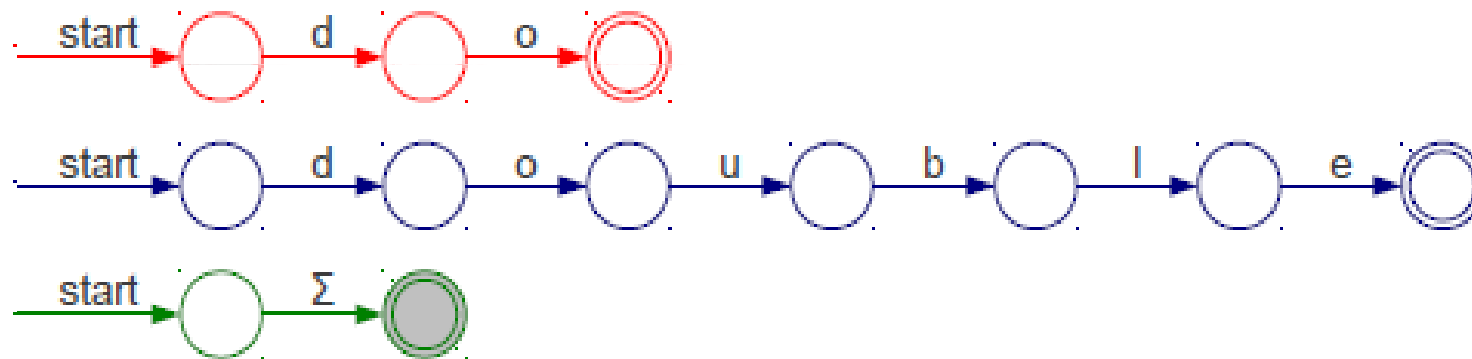


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]



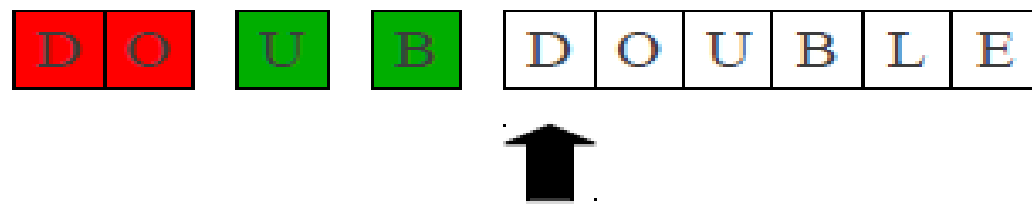
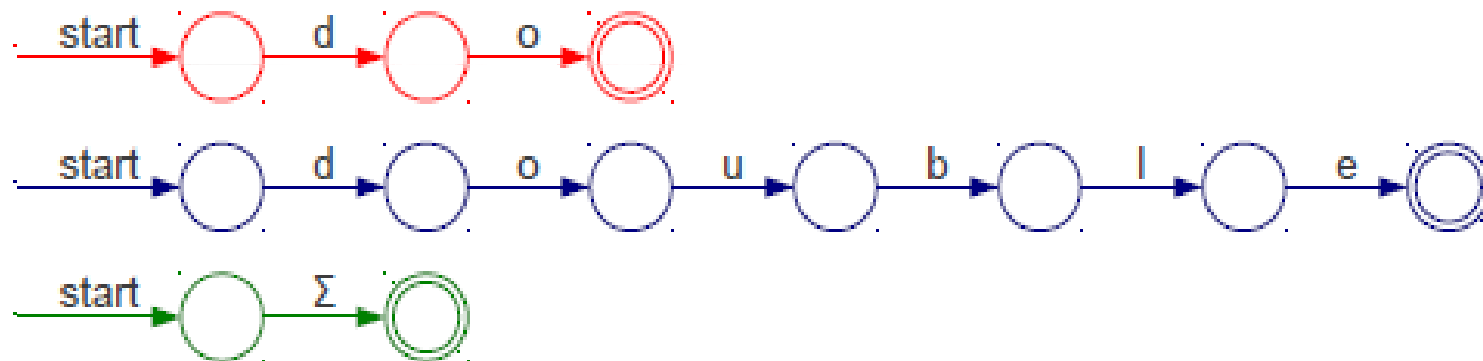


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]

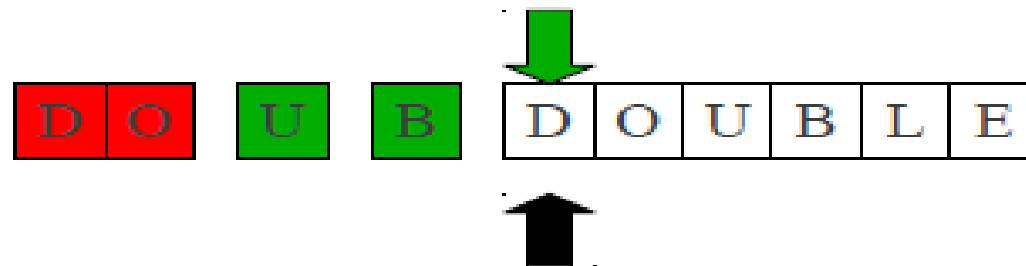
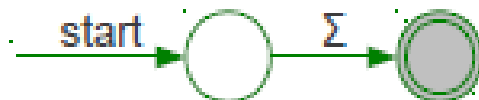
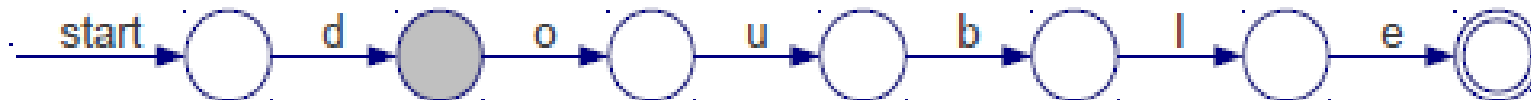
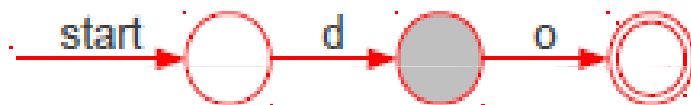


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]

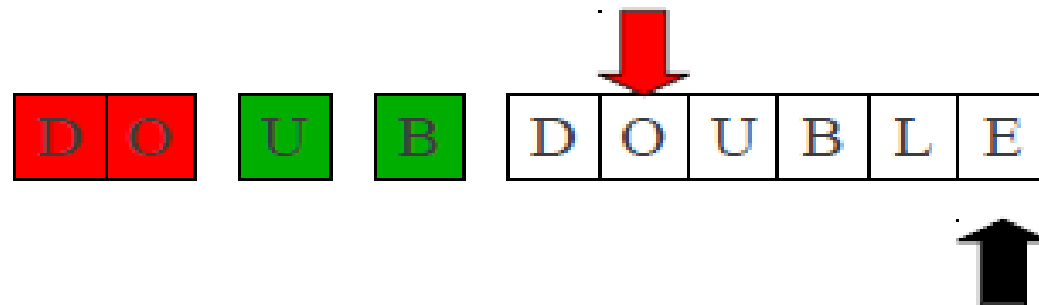
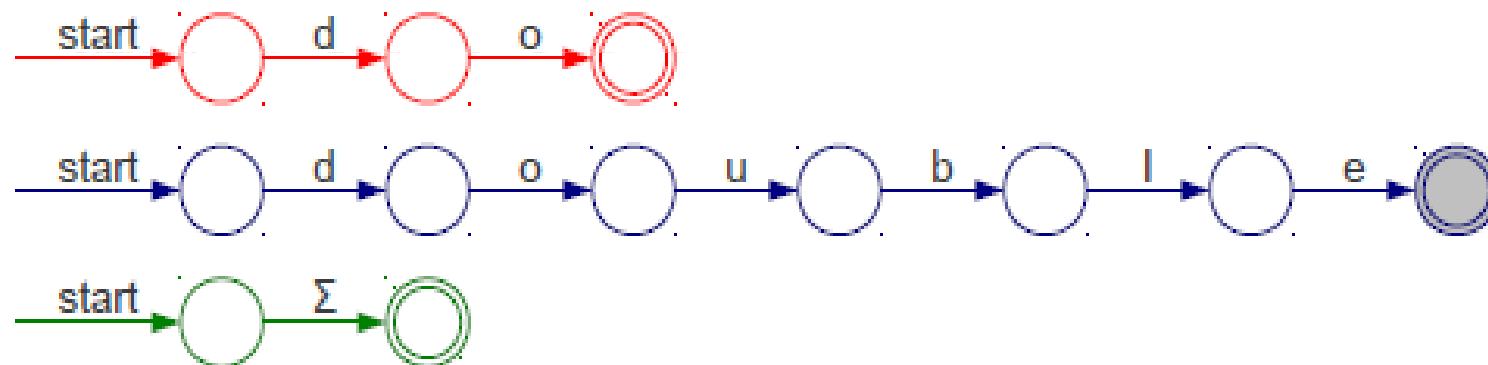


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

T\_nonso [a-zA-Z]

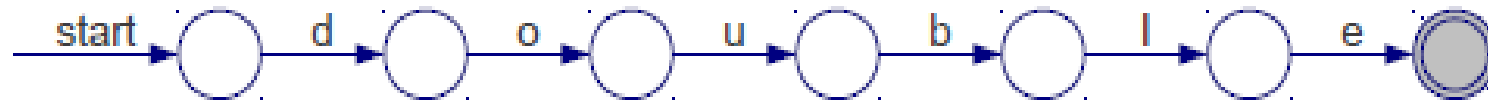


PIÙ IN DETTAGLIO...

T\_do do

T\_double double

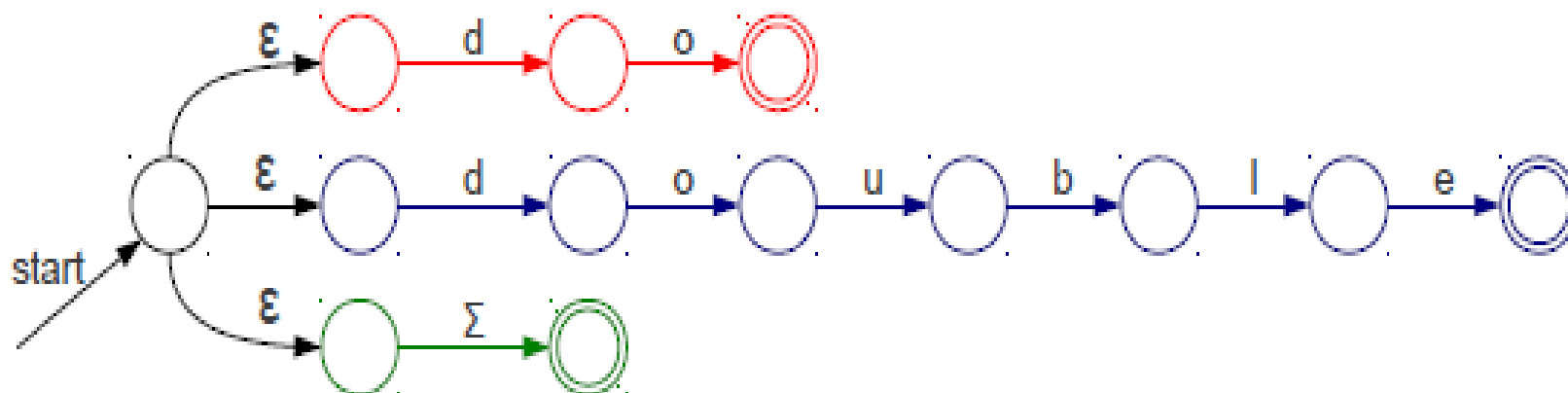
T\_nonso [a-zA-Z]



DOUB DOUBBLE

*COSA SUCCEDE NEL DFA?*

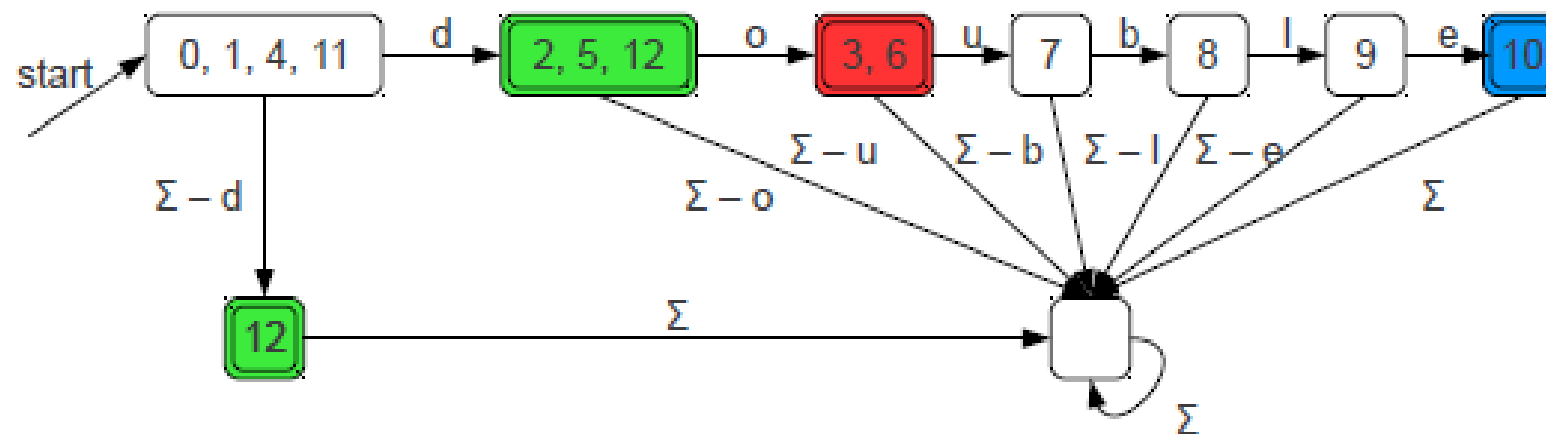
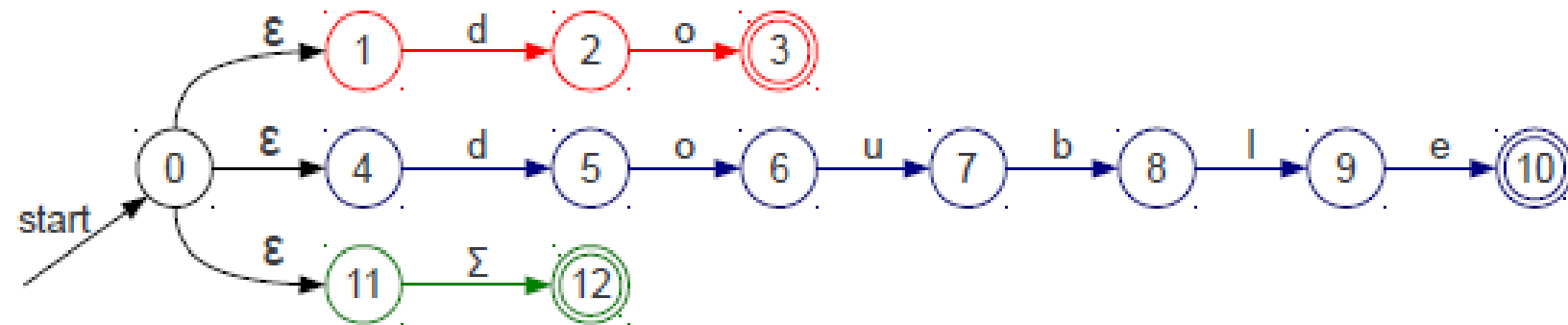
## VARIANTE DELLA SUBSET CONSTRUCTION



Si applica la subset construction

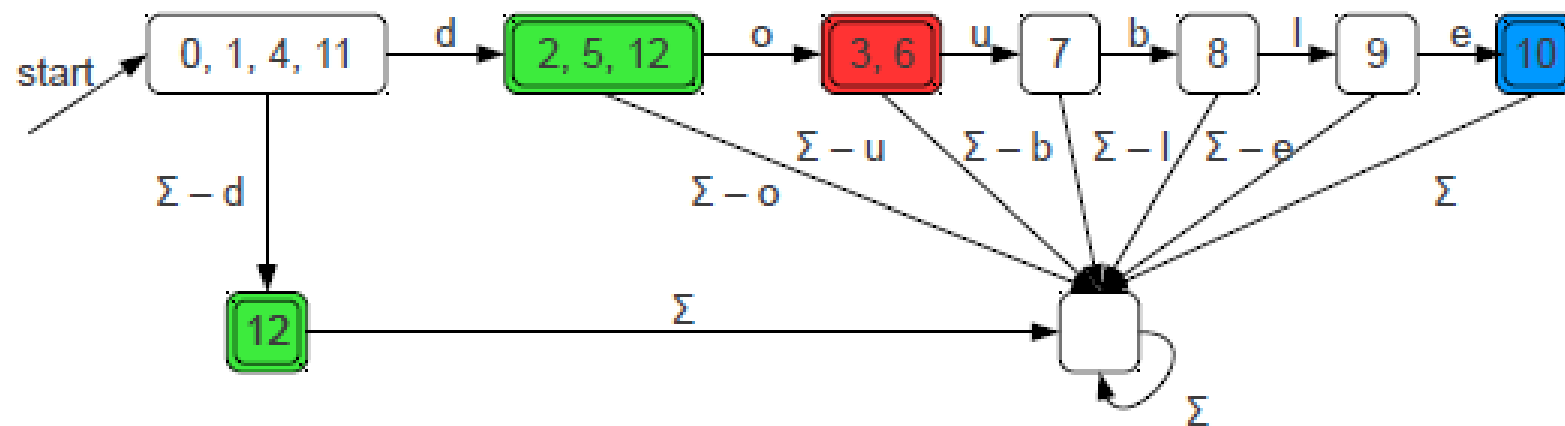
Nel DFA ogni stato che contiene stati accettanti, conterrà anche l'informazione sul NFA di provenienza.

# DAL NFA AL DFA



# DAL NFA AL DFA

- Se in uno stato ci fossero più stati accettanti verrebbe definito un ordine tra questi.
- Si legge il testo un carattere per volta e mantiene memoria gli stati intermedi. Appena si arriva in uno stato pozzo o non ci sono le transizioni uscenti, si cerca a ritroso lo stato di accettazione.
- *Suggerimento per l'implementazione: E' possibile usare la pila in cui si inseriscono via via gli stati di accettazione incontrati durante il processo di matching*





PER ESERCIZIO...

a  
abb  
 $a^*b^+$



Costruire e  
combinare i  
NFAs



Costruire il  
DFA

Cosa succede se il testo è:

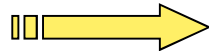
aaba  
abba



# I PRIMI ESEMPI...

Il programma più semplice:

**primo.fl**



%%

Esiste la regola di default:  
il testo che non “matcha” con nulla  
viene ricopiato sull’output.

Genera uno scanner che semplicemente copia il suo input (un carattere per volta) nel suo output.

N.B. Sia nella sezione delle definizioni che in quella delle regole:

- ogni testo indentato o racchiuso tra %{ e %} viene copiato parola per parola sull’output.
- %{ e %} devono apparire a inizio di linea (non indentati).
- nella sezione delle definizioni, un commento non-indentato che comincia con /\* e finisce con \*/ diventa un commento nel file di output

Come si usa:

1. flex primo.fl
2. gcc lex.yy.c -o primo
3. primo < prova.txt >output.txt

## ESEMPIO 1

- Mostriamo come deve essere scritto il file primo.fl

```
/* E' un opzione che serve nel caso si voglia utilizzare la  
funzione main generata di default */  
%option main
```

```
%%
```

## ALCUNE OPZIONI

%option seguito da

main: flex fornisce un routine main() di default

noyywrap: lo scanner non chiamerà la routine yywrap che serve per la gestione di più file di input



## COME DEFINIRE I PATTERN NELLA SEZIONE DELLE REGOLE?

`x` match con il carattere 'x'

`.` Qualsiasi carattere escluso il newline

`[xyz]` una "classe di caratteri"; in questo caso, il pattern matcha o con 'x', 'y', o 'z'

`[abj-oZ]` una "classe di caratteri" contenente un range; matcha con 'a', 'b', una qualsiasi lettera da 'j' a 'o', oppure 'Z'

`[^A-Z]` una "classe negata di caratteri", i.e., qualsiasi carattere escluso quelli della classe. In questo caso, un qualsiasi carattere escluso le lettere maiuscole.

`[^A-Z\n]` qualsiasi carattere tranne le maiuscole e il newline

Note:

Con le parentesi quadre solo `\` - e `^` sono caratteri speciali; per includere il carattere – questo deve apparire come primo o ultimo carattere.

## COME DEFINIRE I PATTERN NELLA SEZIONE DELLE REGOLE?

$r^*$	zero o più espressioni $r$
$r^+$	una o più espressioni $r$
$r?$	zero o una $r$
$r\{2,5\}$	da due a cinque $r$
$r\{2,\}$	due o più $r$
$r\{4\}$	esattamente 4 $r$
$\{name\}$	espansione della definizione "name"
$"[xyz]\"foo"$	la stringa: $[xyz]\"foo$

Note:

il carattere  $\backslash$  ha il ruolo di carattere di escape.

## COME DEFINIRE I PATTERN NELLA SEZIONE DELLE REGOLE?

`\x` se `x` è `'a'`, `'b'`, `'f'`, `'n'`, `'r'`, `'t'`, o `'v'`, allora `\x` ha la stessa interpretazione che per ANSI-C. Altrimenti, sta per il carattere `x`. (usato per l'escape di operatori come `'*'`)

`\0` carattere NULL (ASCII code 0)

`\123` carattere con valore ottale 123

`\x2a` carattere con valore esadecimale 2a

`(r)` Espressione `r`; le parentesi servono per imporre delle precedenze

`rs` espressione `r` seguita da `s`

`r|s` o `r` oppure `s`

`r/s` una `r` ma solo se seguita da `s`. Il testo matchato da `s` è considerato incluso per effetto del "longest match", ma è restituito all'input prima che venga eseguita l'azione. Questo tipo di pattern è chiamato "trailing context".

Nota: valgono le regole di precedenza per le espressioni regolari; ad es. `ciao|bu*` è equivalente a `(ciao)|(b(u)*)`

## COME DEFINIRE I PATTERN NELLA SEZIONE DELLE REGOLE?

<code>^r</code>	<code>r</code> , ma solo all'inizio di una linea
<code>r\$</code>	<code>r</code> , ma solo alla fine di una linea. Equivalente a " <code>r/\n</code> ".
<code>&lt;s&gt;r</code>	<code>r</code> , ma solo nella start condition <code>s</code>
<code>&lt;s1,s2,s3&gt;r</code>	stessa cosa, ma in una qualsiasi start condition <code>s1</code> , <code>s2</code> , o <code>s3</code>
<code>&lt;*&gt;r</code>	<code>r</code> in una qualsiasi start condition, anche in una esclusiva.
<code>&lt;&lt;EOF&gt;&gt;</code>	end-of-file
<code>&lt;s1,s2&gt;&lt;&lt;EOF&gt;&gt;</code>	end-of-file in start condition <code>s1</code> o <code>s2</code>

`[:digit:]` indica tutti i caratteri per cui "isdigit"  
restituisce true

ES: `[:alpha:][:digit:]` è equivalente a `[a-zA-Z0-9]`



# COME AVVIENE IL MATCH?

- Quando viene determinato un match, il testo corrispondente al match viene reso disponibile attraverso la variabile globale **yytext** e la sua lunghezza viene memorizzata in **yylen**.
- Vengono quindi eseguite le azioni corrispondenti al pattern per il quale avviene il match.
- Prosegue la scansione dell'input alla ricerca di altri match.
- Se non viene trovato alcun match, viene eseguita la *default rule*: il carattere dell'input viene considerato un match e copiato nell'output

Nota: la variabile `yytext` può essere definita come `char pointer` o `array`; E' possibile controllare tale definizione attraverso la direttiva `%pointer` o `%array` nella sezione delle definizioni.  
Se si usa l'opzione `-l` (compatibilità con Lex), `yytext` è un array.  
`% pointer` consente scansioni più veloci

# COME DEFINIRE LE AZIONI?

## pattern      azione

- Se l'azione è vuota (contiene solo ; ), al match del pattern non succede nulla.
- Se l'azione contiene un {, allora verranno eseguite le azioni fino al raggiungimento di }.
- Un'azione che consiste solo di | indica che è definita nella stesso modo della regola che segue.

Es1:  
[ ] |  
\t |  
\n ;

Es2:  
[a-z]+      {printf("%s",yytext);}  
È equivalente a  
[a-z]+      {ECHO;}

Direttive speciali che possono essere incluse in un'azione:

ECHO: copia yytext nell'output dello scanner

BEGIN: seguito dal nome di una condizione START posiziona lo scanner in tale condizione

REJECT: dirige lo scanner a procedere sul secondo longest best match (di solito un prefisso dell'input)