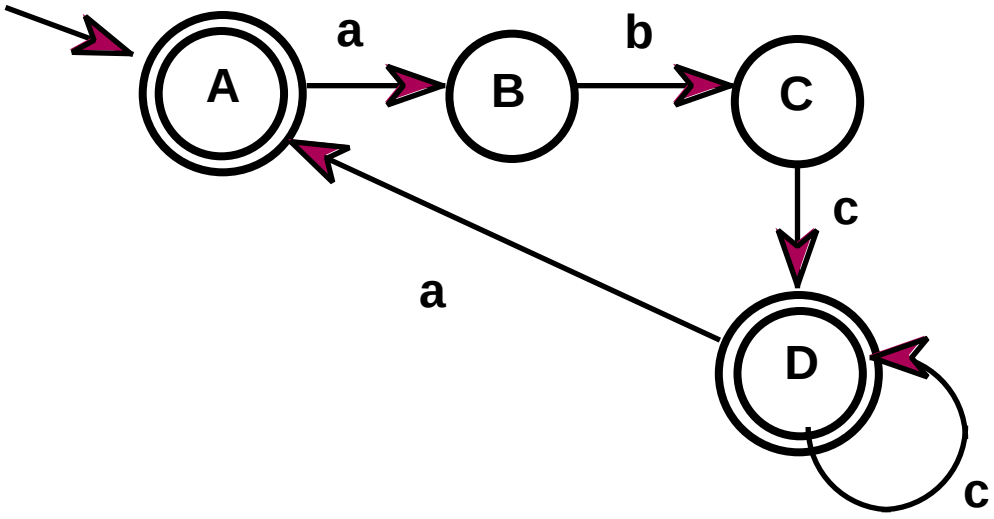
DFAs, REs & Scanning

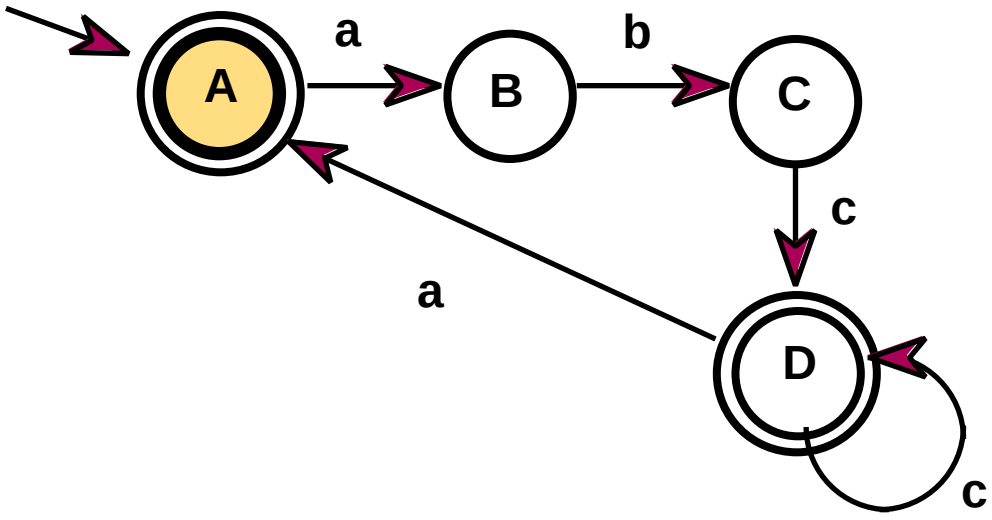


Finite set of states: S
Finite alphabet: A
Transition function:
T: SxA -> S
Start State: s
Set of final states f1..fn

A,a: B
B,b: C
C,c: D
D,a: A
D,c: D with
s = A and
F = {A, D}

(ab(cc*)a)*

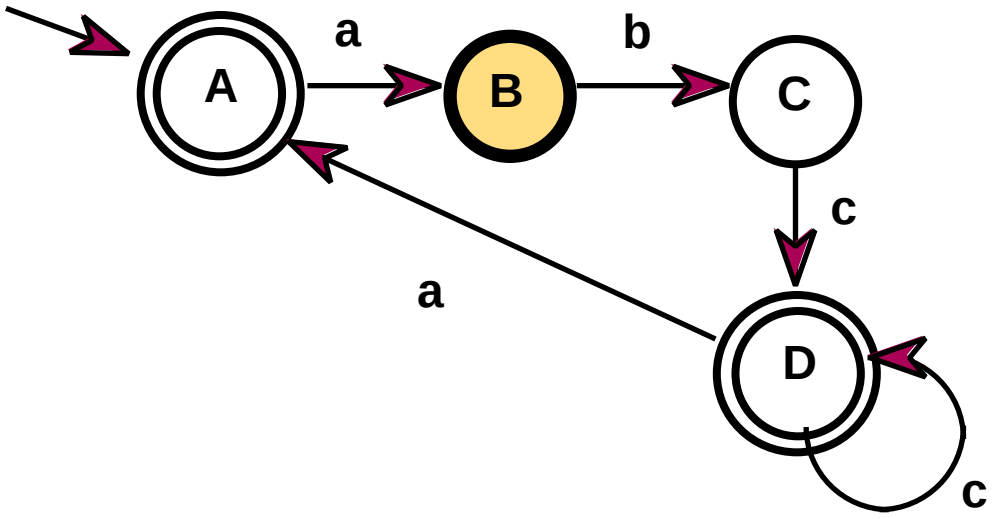
DFAs, REs & Scanning



abcccabc

(ab(cc*)a)*

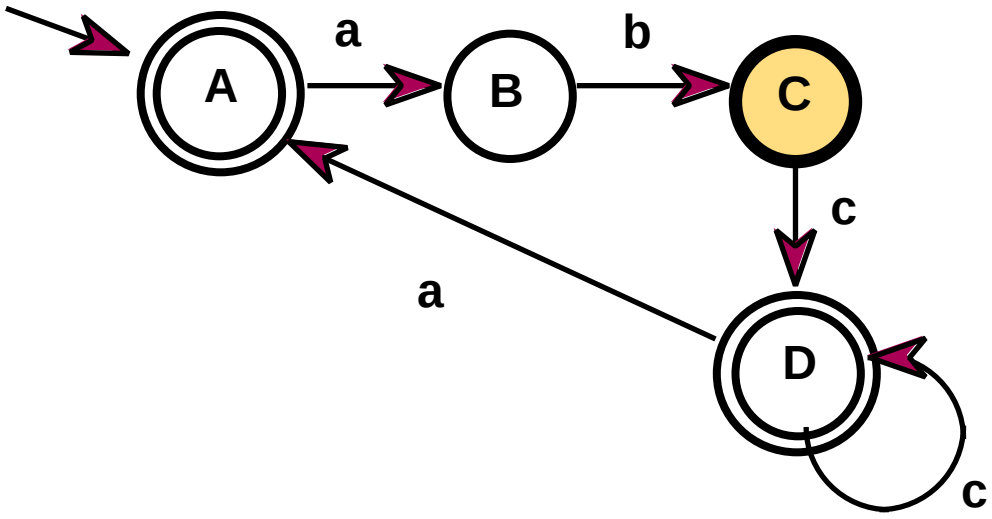
DFAs, REs & Scanning



abcccabc

(ab(cc*)a)*

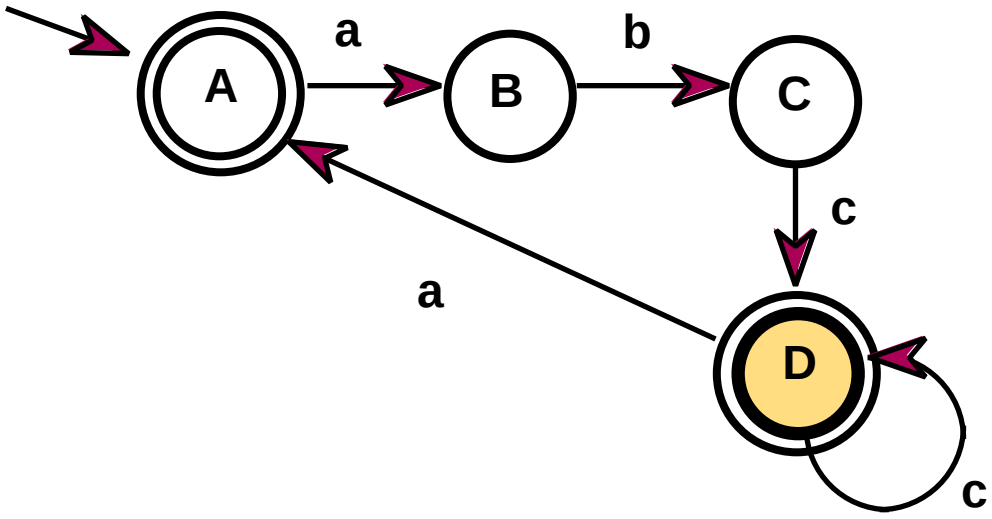
DFAs, REs & Scanning



abcccabc

(ab(cc*)a)*

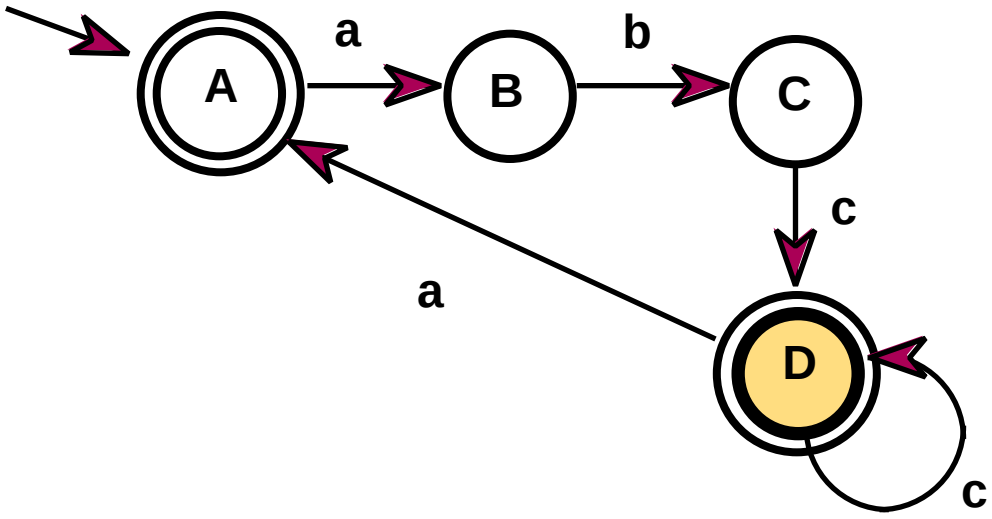
DFAs, REs & Scanning



abccccabc

(ab(cc*)a)*

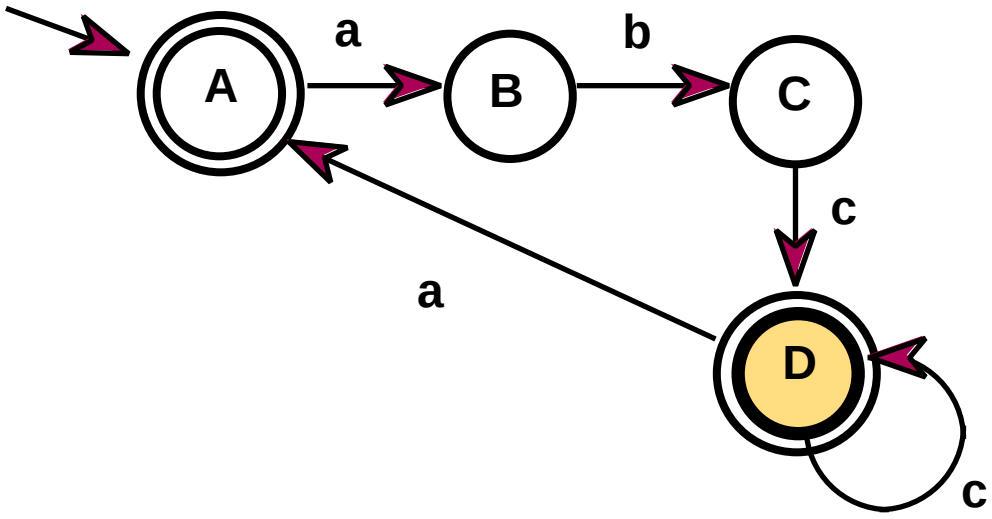
DFAs, REs & Scanning



abcccabc

(ab(cc*)a)*

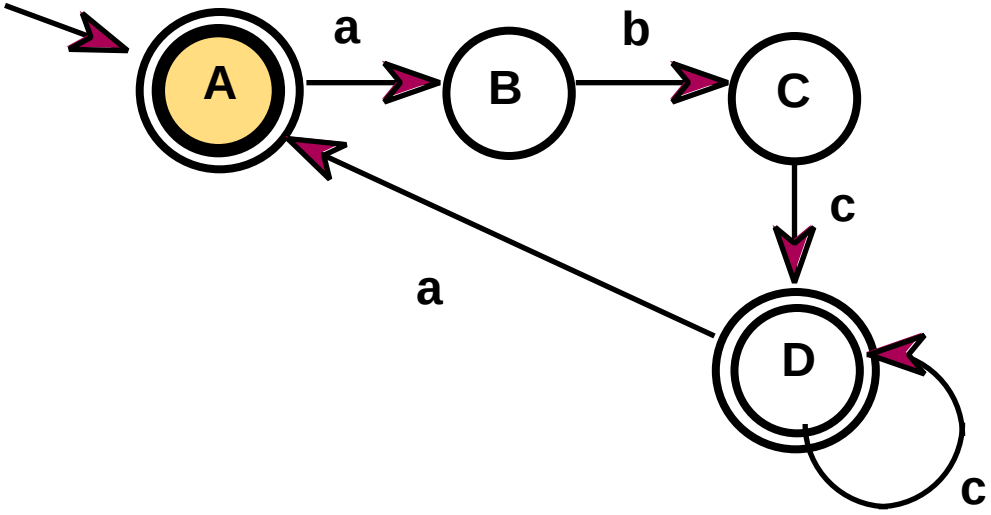
DFAs, REs & Scanning



abcccabc

(ab(cc*)a)*

DFAs, REs & Scanning



abcccabc

ERROR, No Transition

(ab(cc*)a)*

DFAs, REs & Scanning

A DFA divides the set of all finite strings over its alphabet into two sets: accepted strings vs rejected

To ACCEPT the input string

Consume entire input string - and -

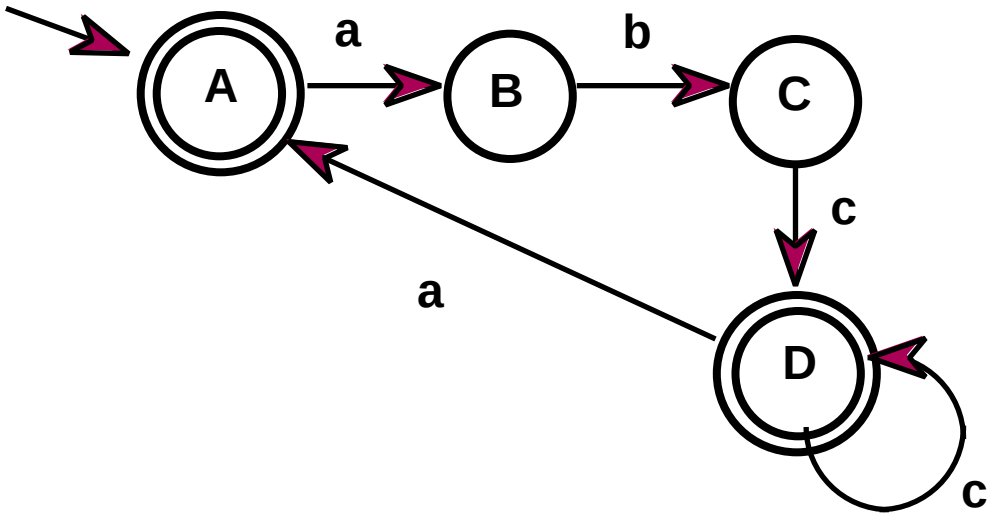
Finish in a FINAL state.

REJECT if either

No transition - or -

Finish in a non-Final state

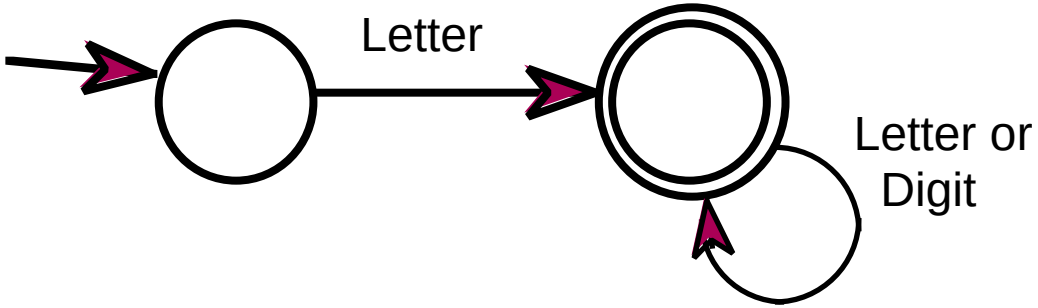
Simulation

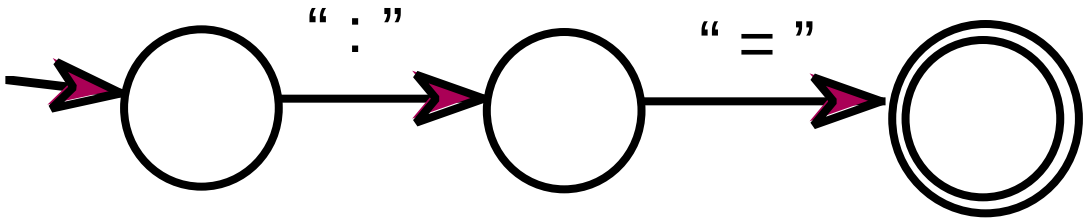



```
fun getToken: token =
    clearbuffer
    return A
```

```
fun A: token =
  if endstr {return “A”}
  getchar(x); bufferchar(x)
  if x = a {return B} else error
fun B: token =
  if endstr { error}
  getchar(x); bufferchar(x)
  if x = b {return C} else error
fun C: token
  if endstr { error}
  getchar(x); bufferchar(x)
  if x = c {return D} else error
fun D: token
  if endstr { return “D”}
  getchar(x): bufferchar(x)
  if x = c {return D} else
  if x = a {return A} else error
```

DFA examples - Pascal tokens





DFAs, REs & Scanning

Finite alphabet A plus meta symbols

λ the empty string

Finite strings of symbols from A (catenations)

() grouping

| alternatives

* zero or more

+ one or more

“x” literal x

A = {a,b,c}

$(abc^{+}a)^{*}$

e.g.

“” empty string

abcccaabcc

abca

Regular Expression examples

D = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
L = A | a | B | ... | z

ID1 = L (L | D)*

Letter followed by letters and digits

ID2 = L (L |D | ”_” (L | D))*

Letter followed by letters and digits and underscores
but no adjacent underscores or terminated by underscore.

PR = D^+ “.” D^+ (λ | “E” (λ | “+” | “-”) D^+)

Pascal real numbers

DFAs, REs & Scanning

Regular Expression	Regular Set
Φ	{ }
λ	{ “” }
ab...c	{ “ab...c” }
A | B	$R_A + R_B$
A B	{ab | a in R_A & b in R_B }
A*	{ “” } + {x..z | x..z in R_A }

Where a,b,...,c are in alphabet and A,B are regular expressions

Where + is set union and

R_A is the Regular Set for A

A set is regular if some regular expression denotes it

Notes:

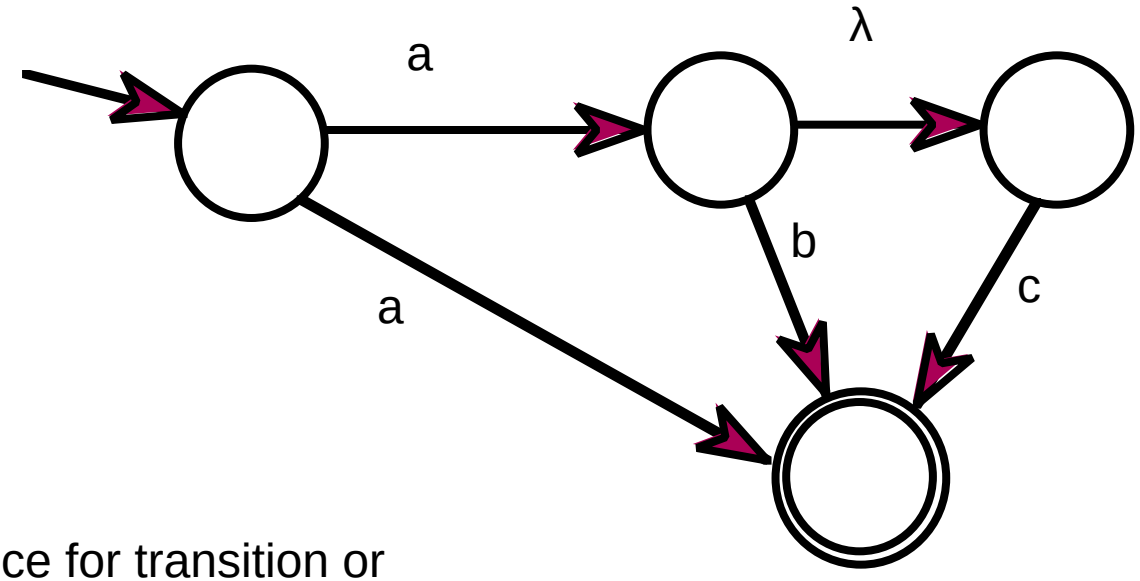
Not all sets are regular

E.g.

{ a...ab...b | same number of a's as b's }

Modern Computer languages have the property that their set of tokens is a regular set.

A finite automoton can be non-deterministic

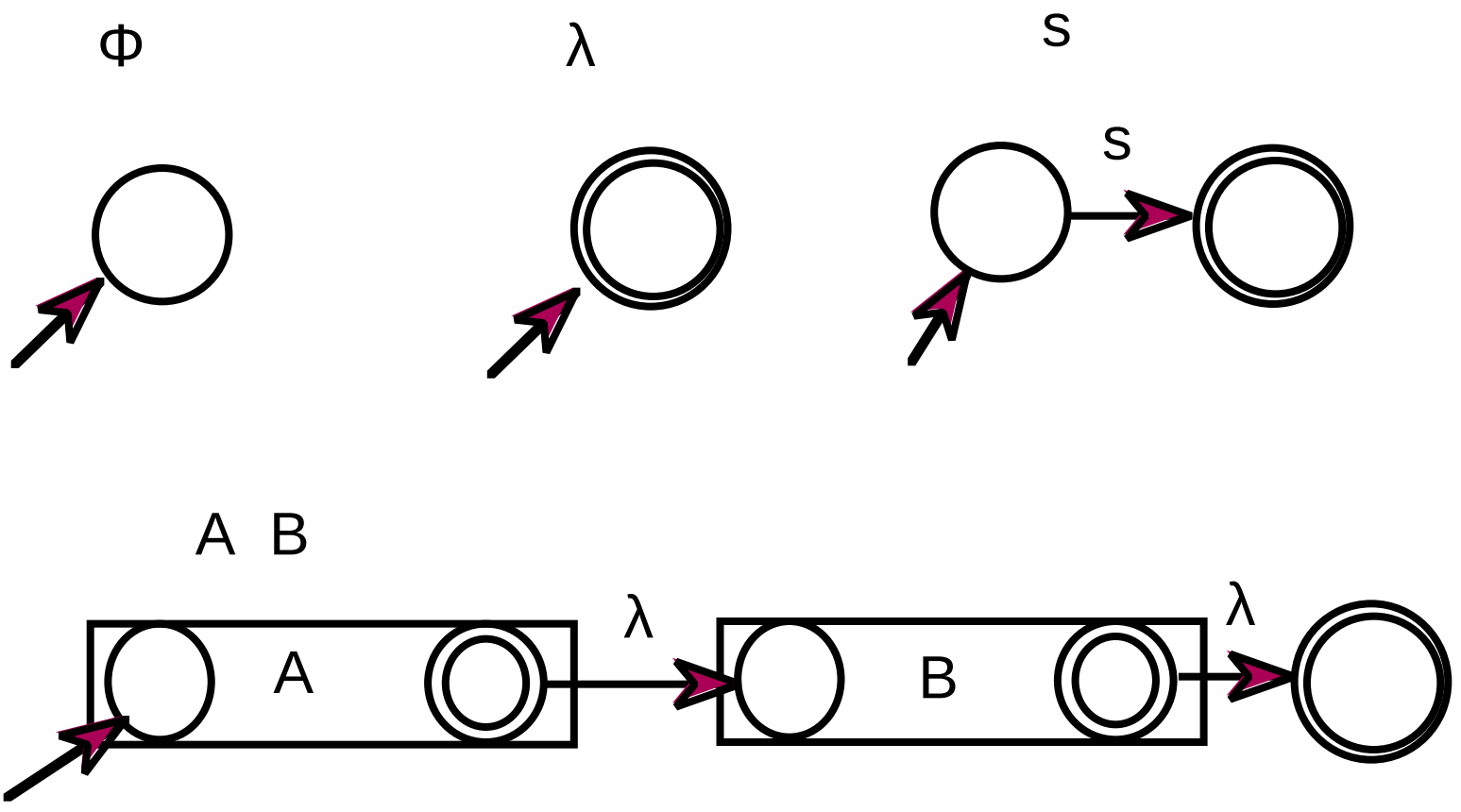


Choice for transition or
transition on no character read

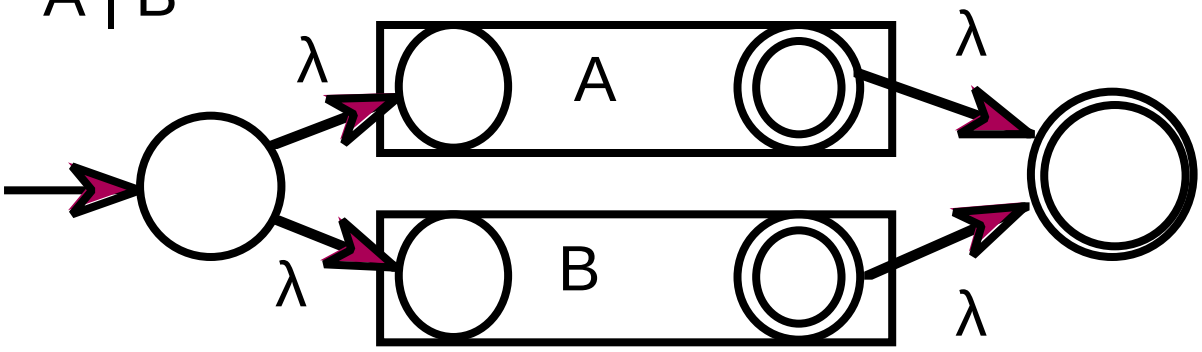
a | ab | ac

When faced with a choice the NFA always guesses right.

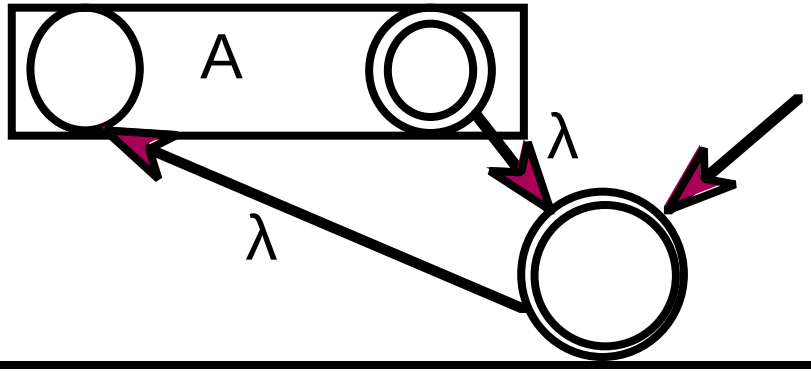
RE to NFA pt 1



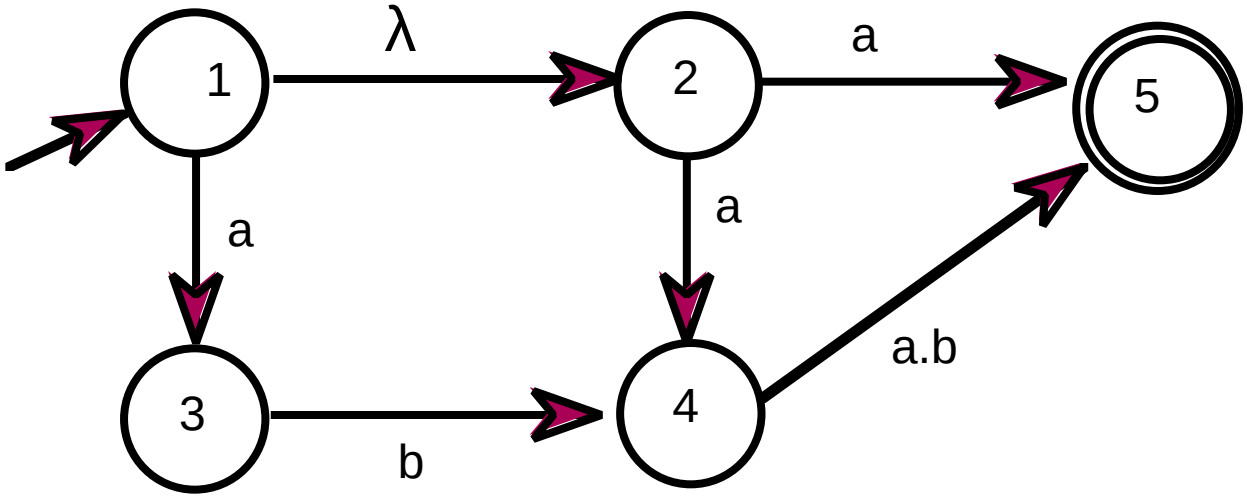
A | B



A *



NFA to DFA pt 1

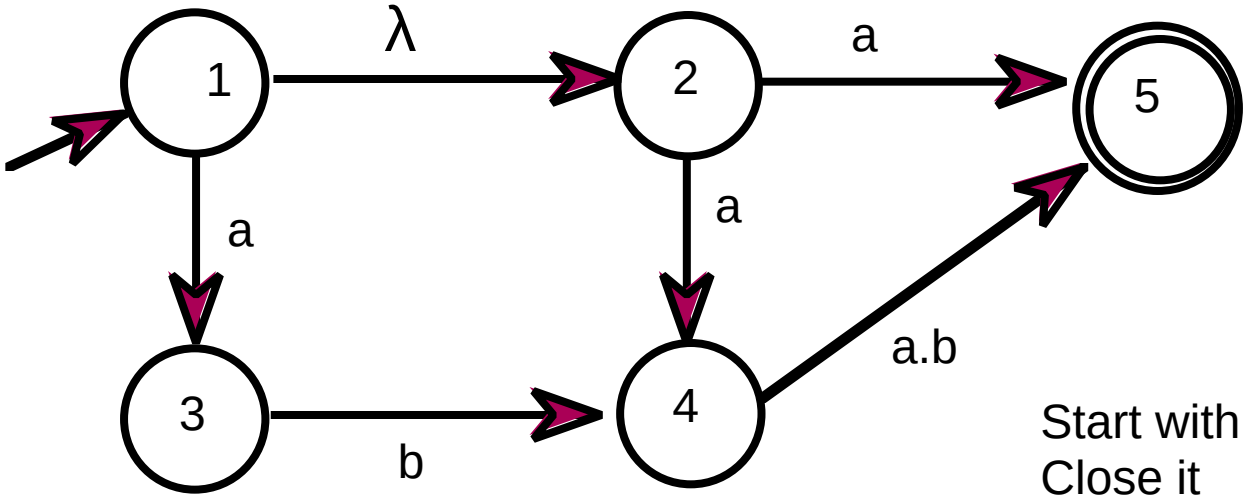


Proc Close = add to a state those states to which you can go on lambda transitions

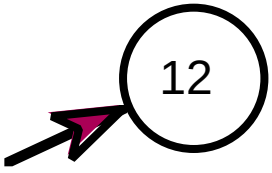
Proc MakeDeterministic = add states if you can go on lambda or same symbol transitions

States of the DFA are sets of NFA states

NFA to DFA pt 2

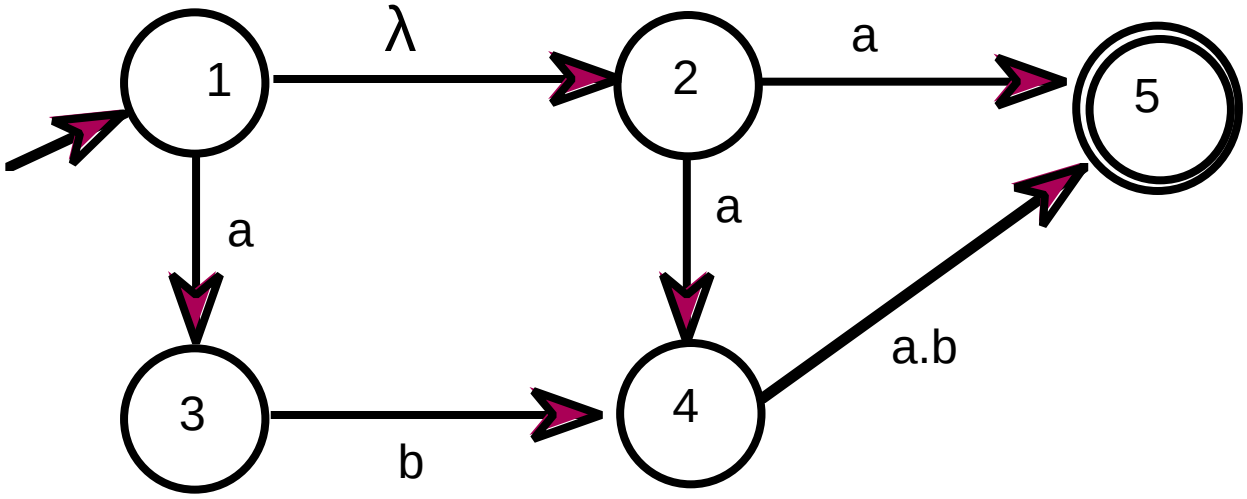


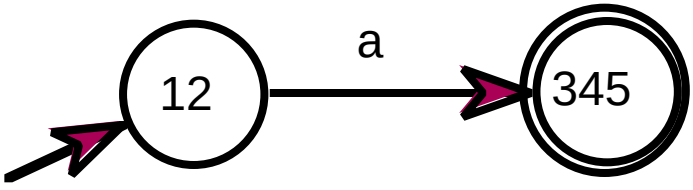
Start with initial state
Close it



Now, from state 12 we can reach states 3 or 4 or 5 on an “a” transition so:

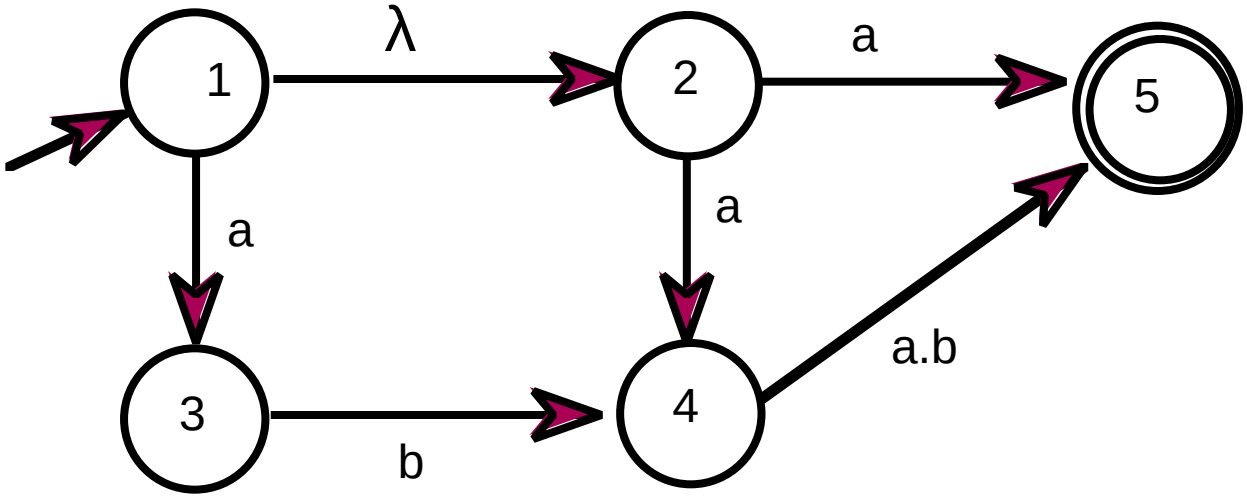
NFA to DFA pt 3

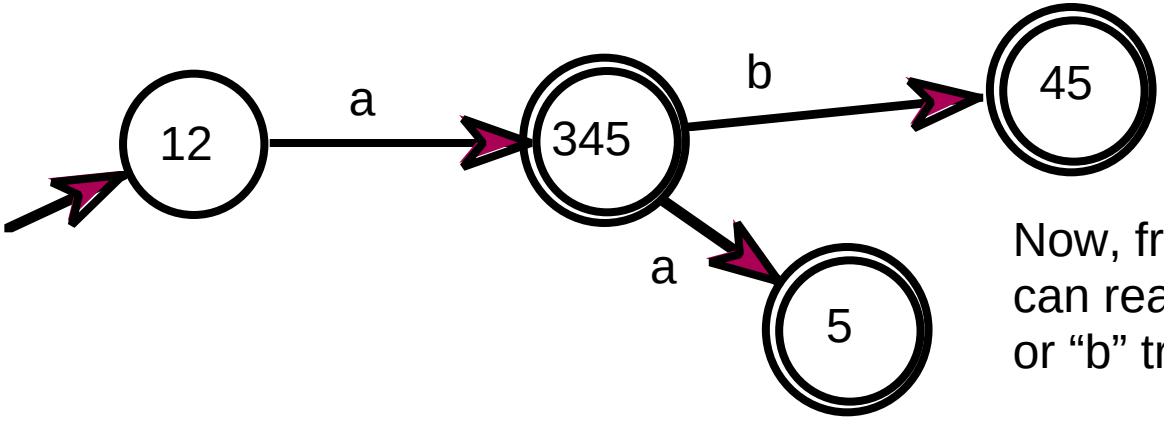




Now, from state 345 we can reach 5 on an “a” transition or states 4 or 5 on a “b” transition so:

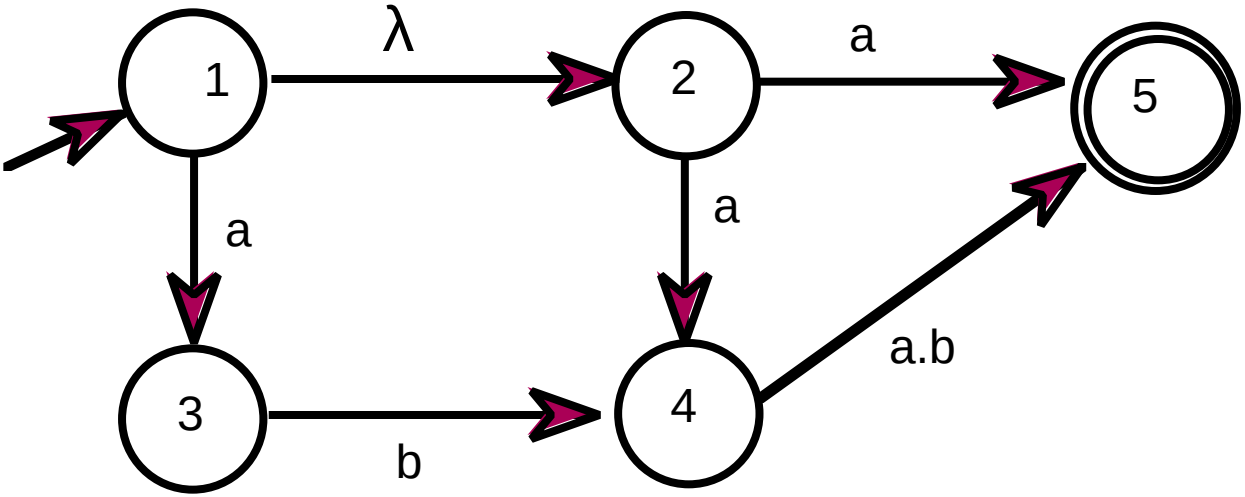
NFA to DFA pt 4

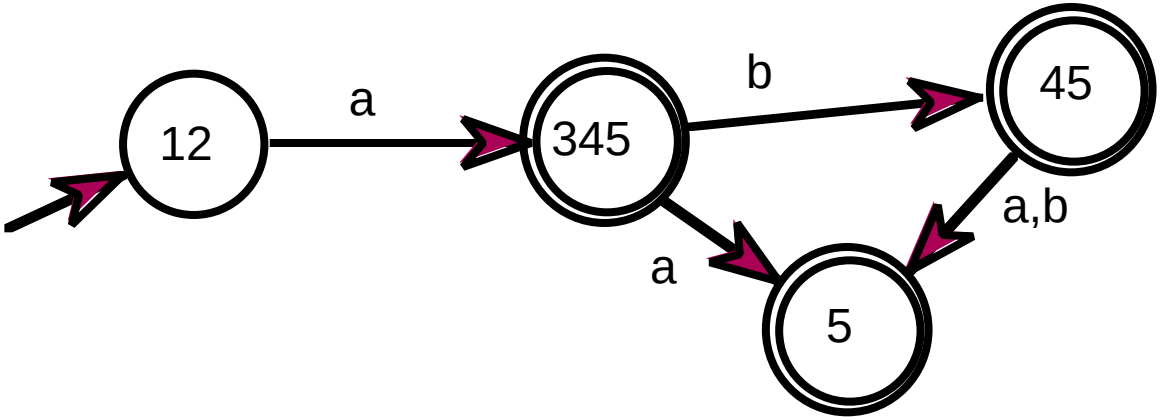




Now, from state 45 we can reach 5 on an “a” or “b” transition so:

NFA to DFA pt 5





We now have an equivalent DFA

DFAs, REs & Scanning

To automate the construction of a scanner

Define the tokens with a regular expression

Create the equivalent NFA

Construct the equivalent DFA

Program the DFA

Caveat: Then NFA -> DFA construction can require a LARGE number of states

The program LEX does this