Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 6

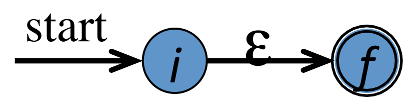
- From RE to DFA, directly
- Minimization of DFA's
- Exercises on lexical analysis

From Regular Expression to DFA Directly

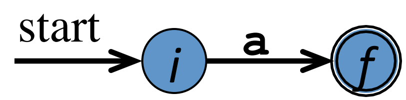
- The "*important states*" of an NFA are those with a non-ε outgoing transition,
  - if $move(\{s\}, a) \neq \emptyset$ for some a then s is an important state
- The subset construction algorithm uses only the important states when it determines $\varepsilon\text{-}closure(move(T, a))$

What are the “important states” in the NFA built from Regular Expression?

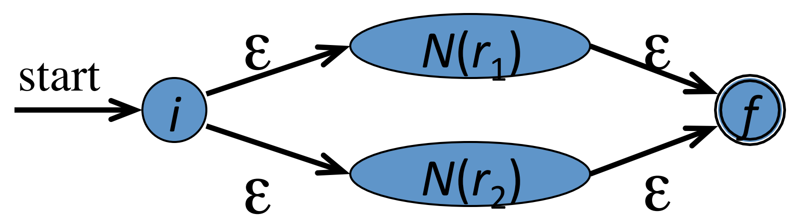
ε



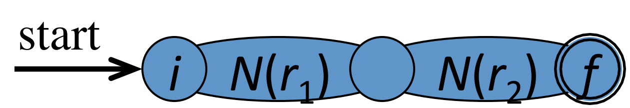
a



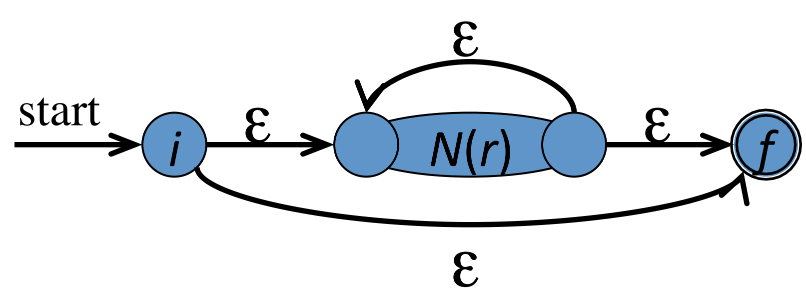
$r_1 | r_2$



$r_1 r_2$



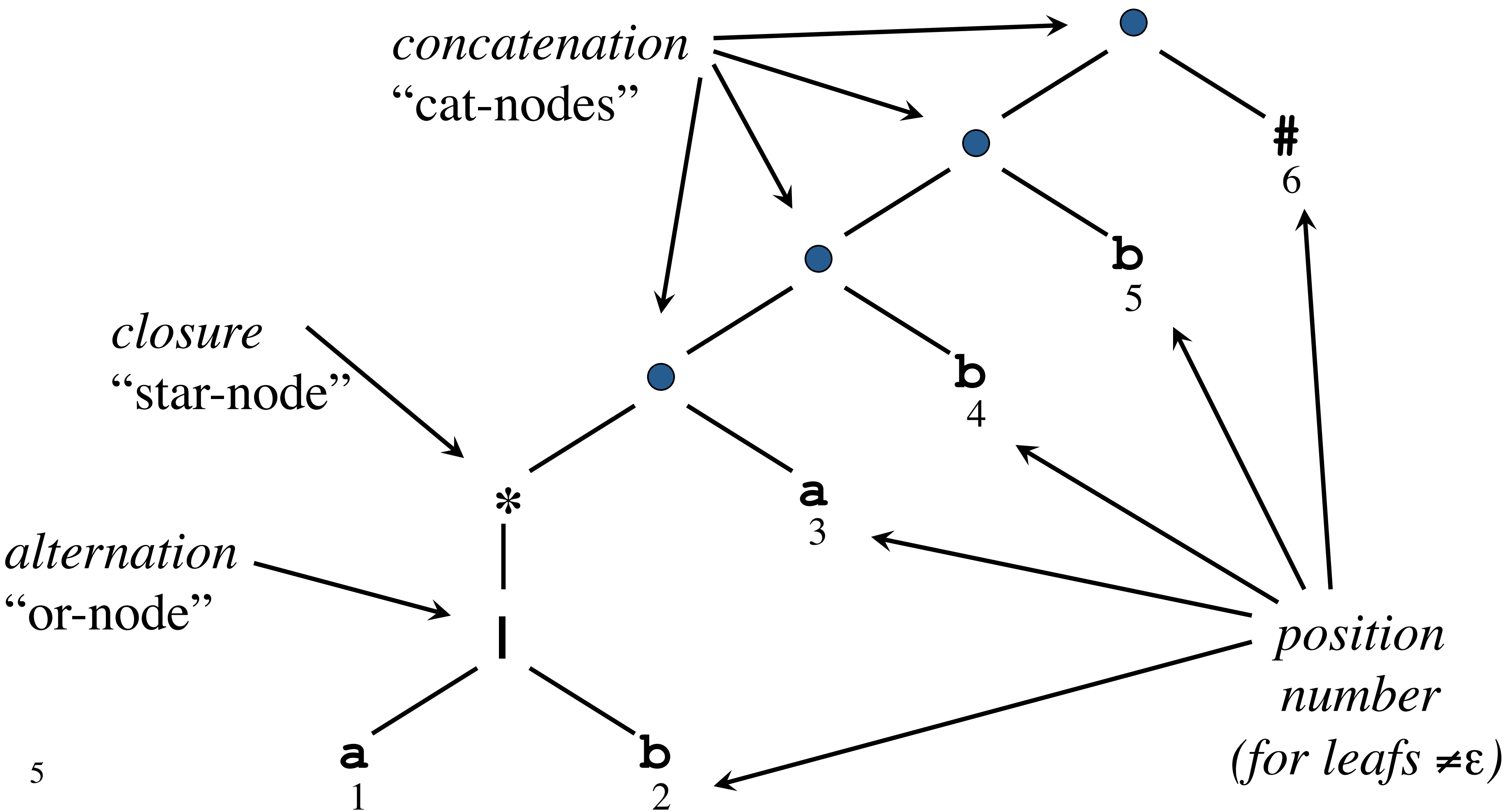
$r*$



From Regular Expression to DFA Directly (Algorithm)

- The only accepting state (via the Thompson algorithm) is not important
- Augment the regular expression *r* with a special end symbol # to make accepting states important: the new expression is *r*#
- Construct a syntax tree for *r*#
- Attach a unique integer to each node not labeled by ε

From Regular Expression to DFA Directly: Syntax Tree of `(a|b)*abb#`



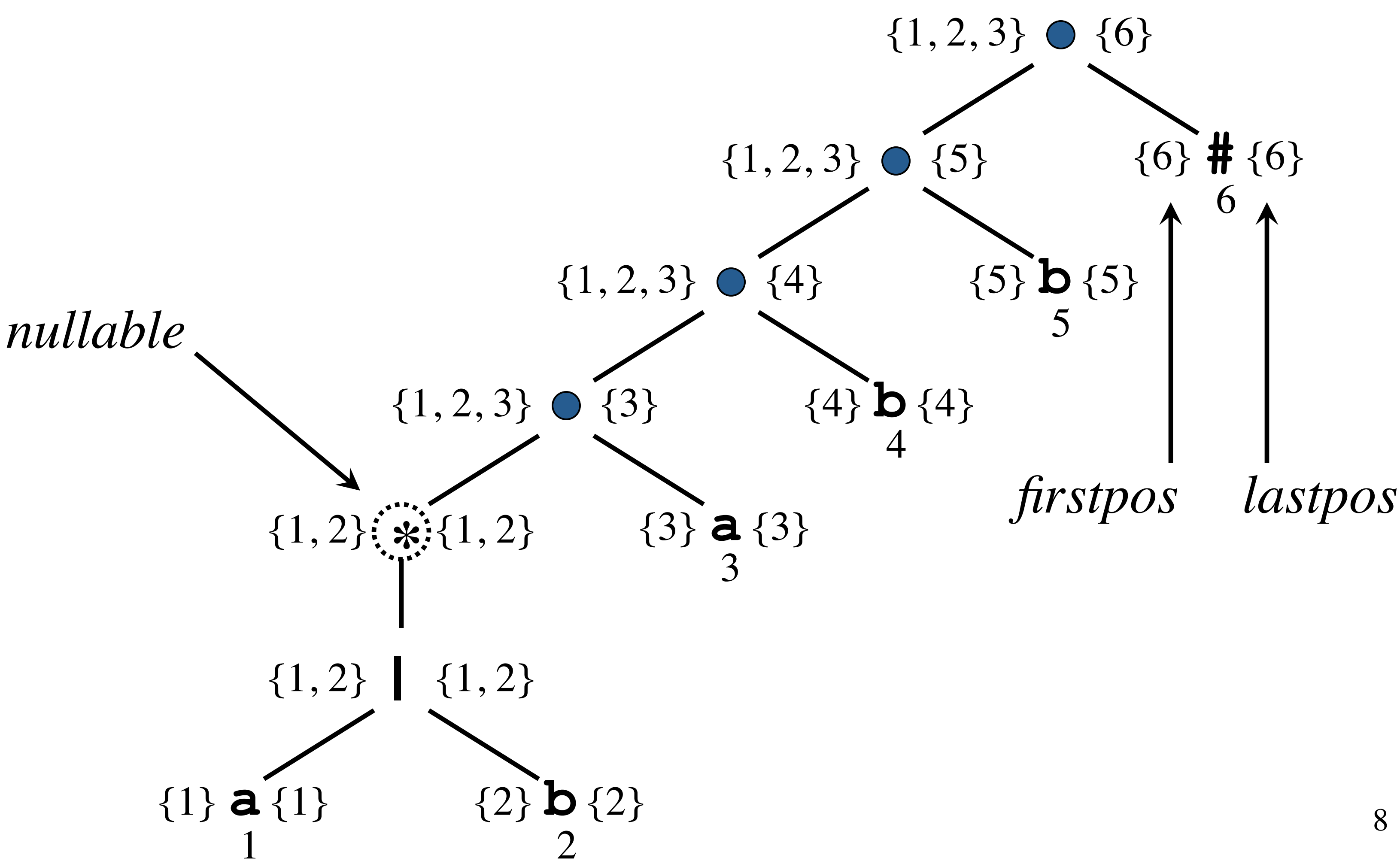
From Regular Expression to DFA Directly: Annotating the Tree

- Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*
- For a node *n*, let *L(n)* be the language generated by the subtree with root *n*
- *nullable*(*n*): *L(n)* contains the empty string ε
- *firstpos*(*n*): set of *positions* under *n* that can match the first symbol of a string in *L(n)*
- *lastpos*(*n*): the set of *positions* under *n* that can match the last symbol of a string in *L(n)*
- *followpos*(*i*): the set of positions that can follow position *i* in any generated string

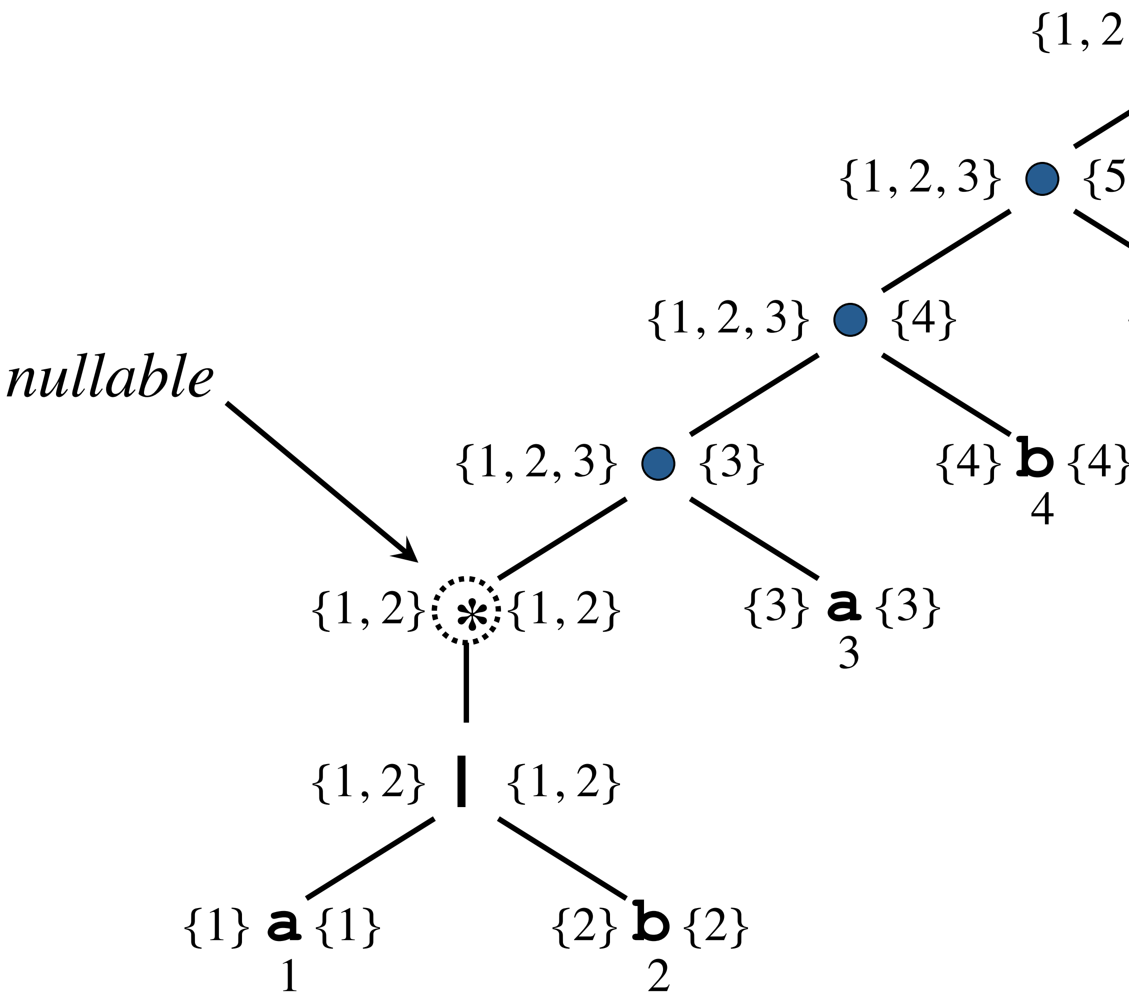
From Regular Expression to DFA Directly: Annotating the Tree

Node n	*nullable*(n)	*firstpos*(n)	*lastpos*(n)
Leaf ε	true	∅	∅
Leaf i	false	$\{i\}$	$\{i\}$
| / \\ c_1 c_2	*nullable*(c_1) or *nullable*(c_2)	*firstpos*(c_1) ∪ *firstpos*(c_2)	*lastpos*(c_1) ∪ *lastpos*(c_2)
• / \\ c_1 c_2	*nullable*(c_1) and *nullable*(c_2)	**if** *nullable*(c_1) **then** *firstpos*(c_1) ∪ *firstpos*(c_2) **else** *firstpos*(c_1)	**if** *nullable*(c_2) **then** *lastpos*(c_1) ∪ *lastpos*(c_2) **else** *lastpos*(c_2)
* | c_1	true	*firstpos*(c_1)	*lastpos*(c_1)

From Regular Expression to DFA Annotating the Syntax Tree of (a|b)*abb#



From Regular Expression to DFA *followpos* on the Syntax Tree of (a|b)*abb#



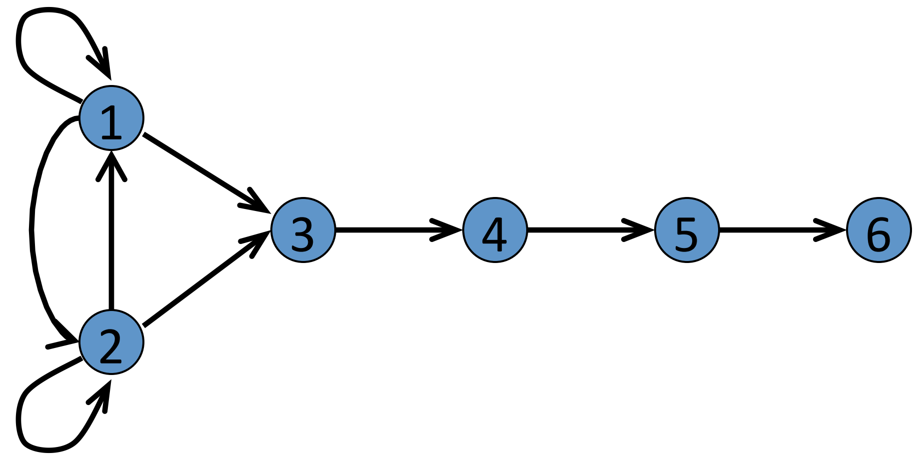
Node n	*followpos*(n)
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

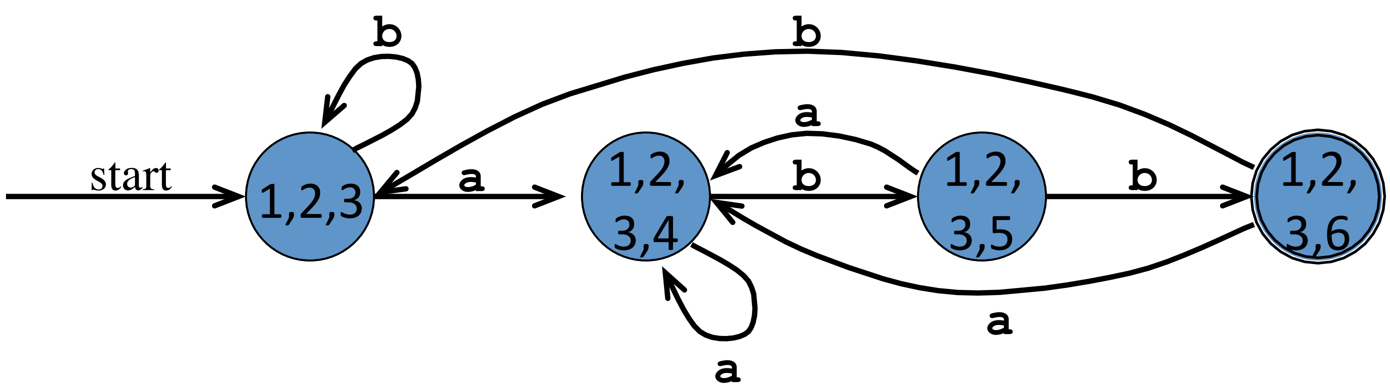
From Regular Expression to DFA Directly: *followpos*

```
for each node n in the tree do
    if n is a cat-node with left child c1 and right child c2 then
        for each i in lastpos(c1) do
            followpos(i) := followpos(i) ∪ firstpos(c2)
        end do
    else if n is a star-node
        for each i in lastpos(n) do
            followpos(i) := followpos(i) ∪ firstpos(n)
        end do
    end if
end do
```

From Regular Expression to DFA Directly: Example

Node	*followpos*
a 1	{1, 2, 3}
b 2	{1, 2, 3}
a 3	{4}
b 4	{5}
b 5	{6}
# 6	-

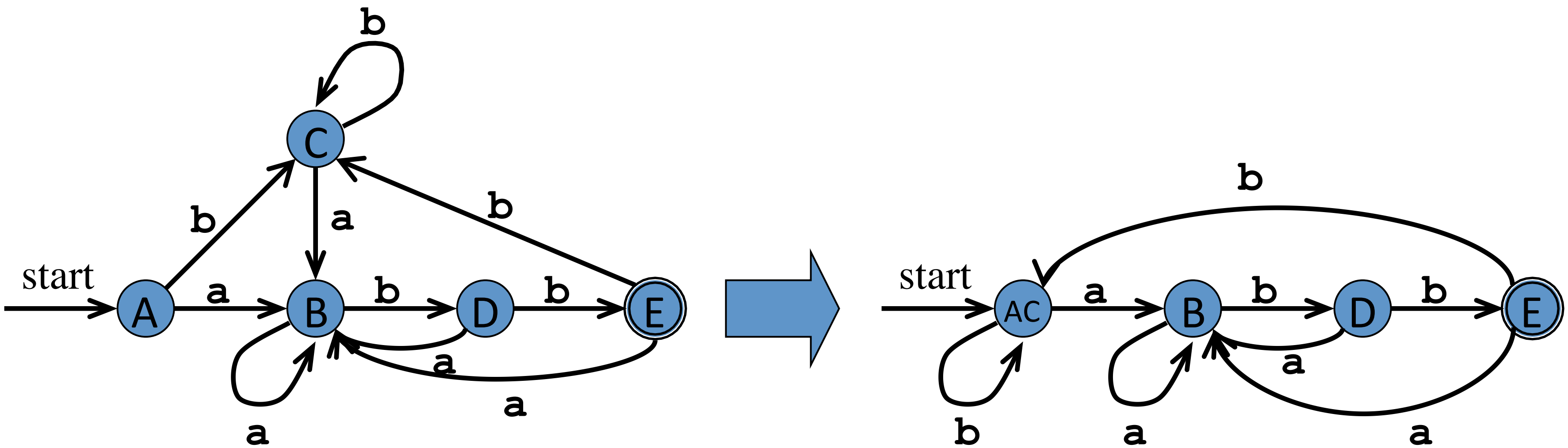




From Regular Expression to DFA Directly: The Algorithm

s_0 := *firstpos*(*root*) where *root* is the root of the syntax tree for ***(r)#***
Dstates := {s_0} and is unmarked
while there is an unmarked state *T* in *Dstates* **do**
    mark *T*
    for each input symbol $a \in \Sigma$ **do**
        let *U* be the union of *followpos*(*p*) for all positions *p* in *T*
            such that the symbol at position *p* is *a*
        if *U* is not empty and not in *Dstates* **then**
            add *U* as an unmarked state to *Dstates*
        end if
        Dtran[*T*, *a*] := *U*
    end do
end do

Minimizing the Number of States of a DFA

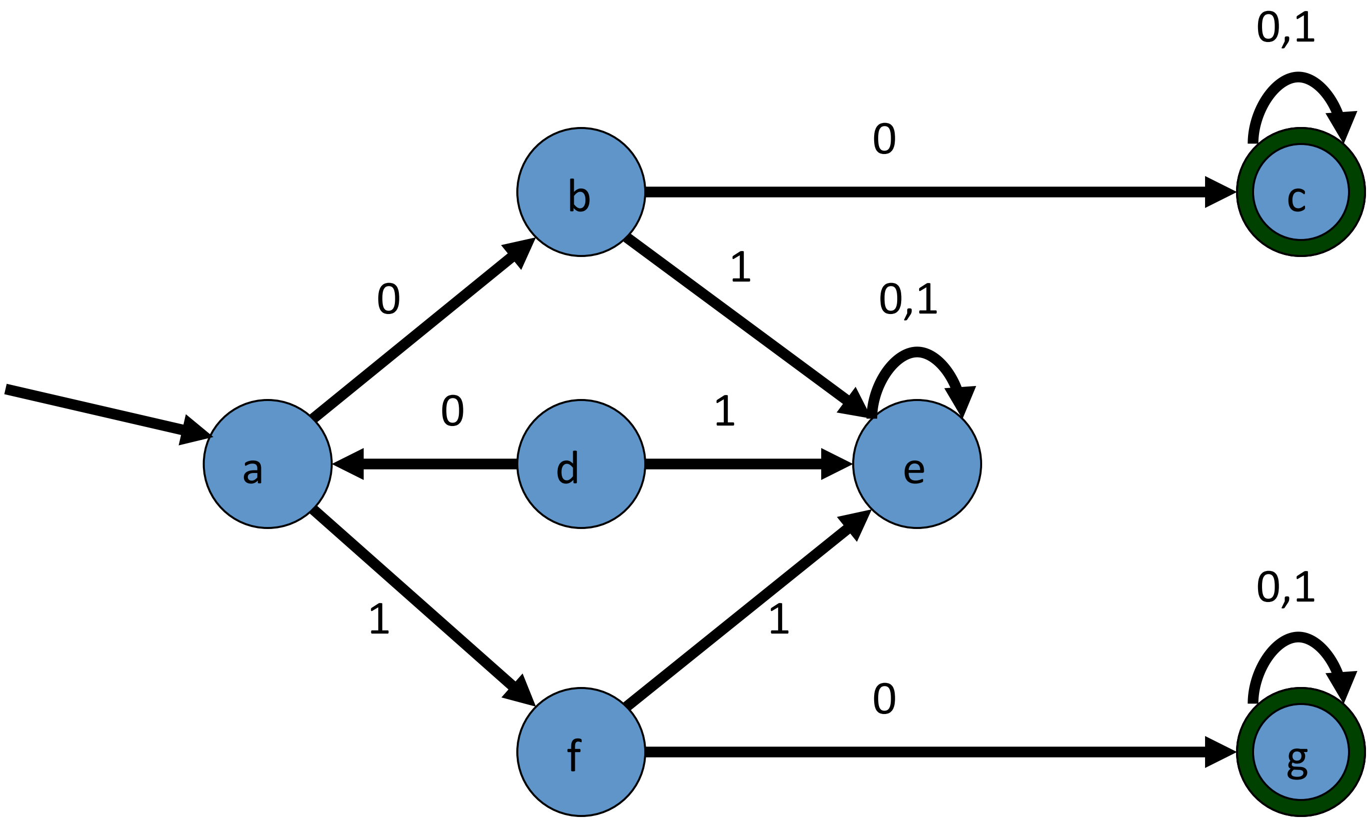


- Given a DFA, let us show how to get a DFA which accepts the same regular language with a minimal number of states

Equivalent States: Example

Consider the accept states **c** and **g**. They are both ***sinks***.

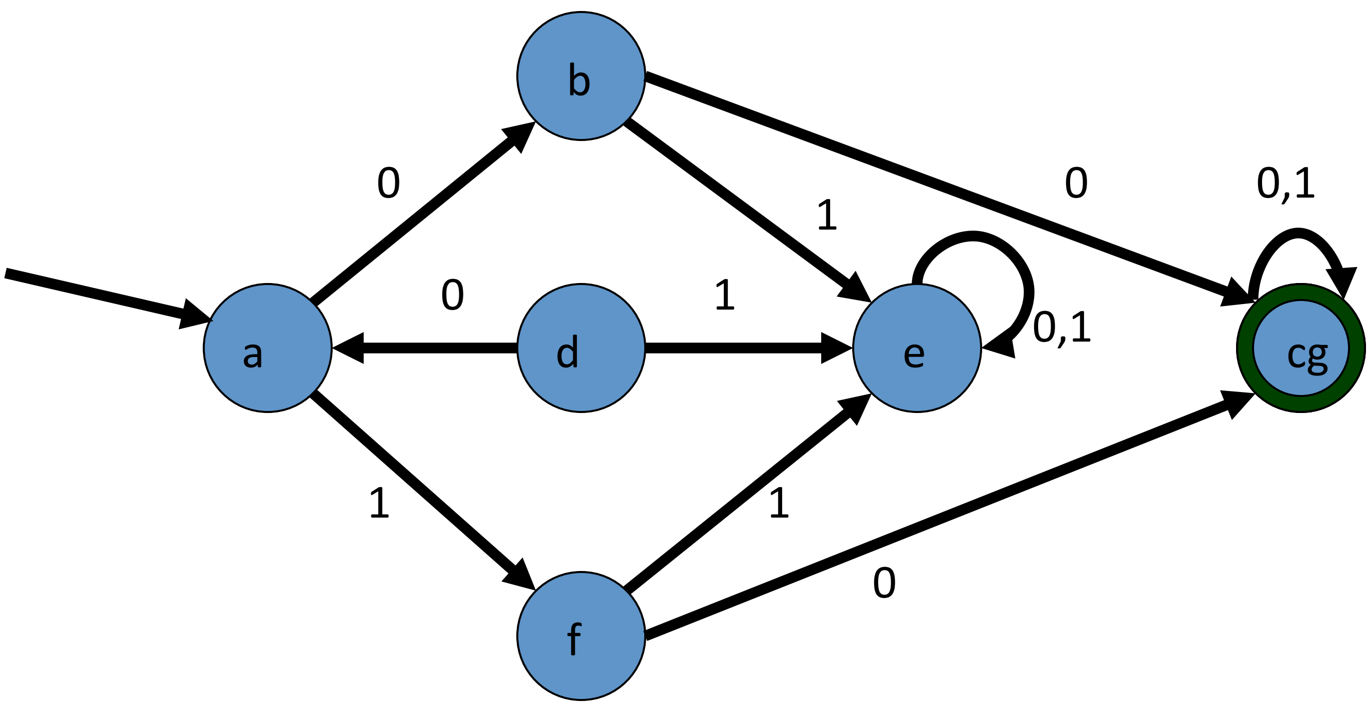
Q: Do we need both states?



Equivalent States: Example

A: No, they can be merged!

Q: Can any other states be merged because any subsequent string suffixes produce identical results?

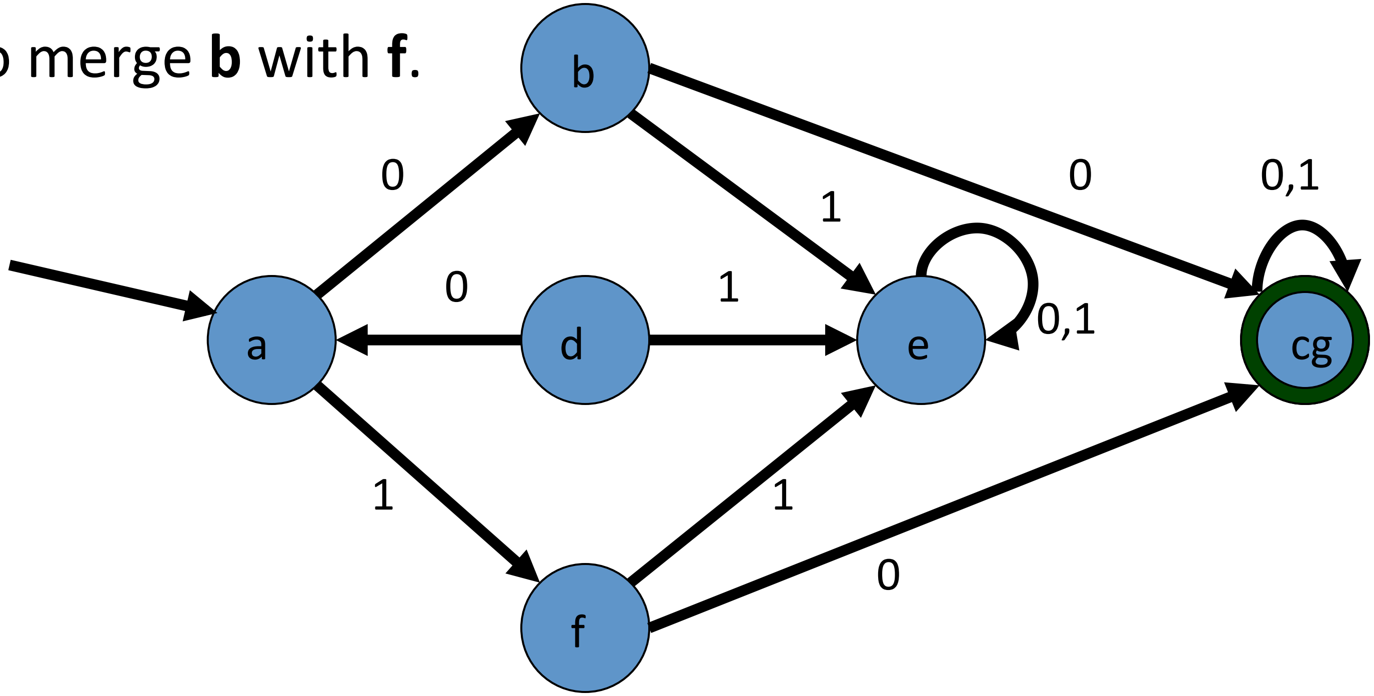


Equivalent States: Example

A: Yes, **b** and **f**. Notice that if you're in **b** or **f** then:

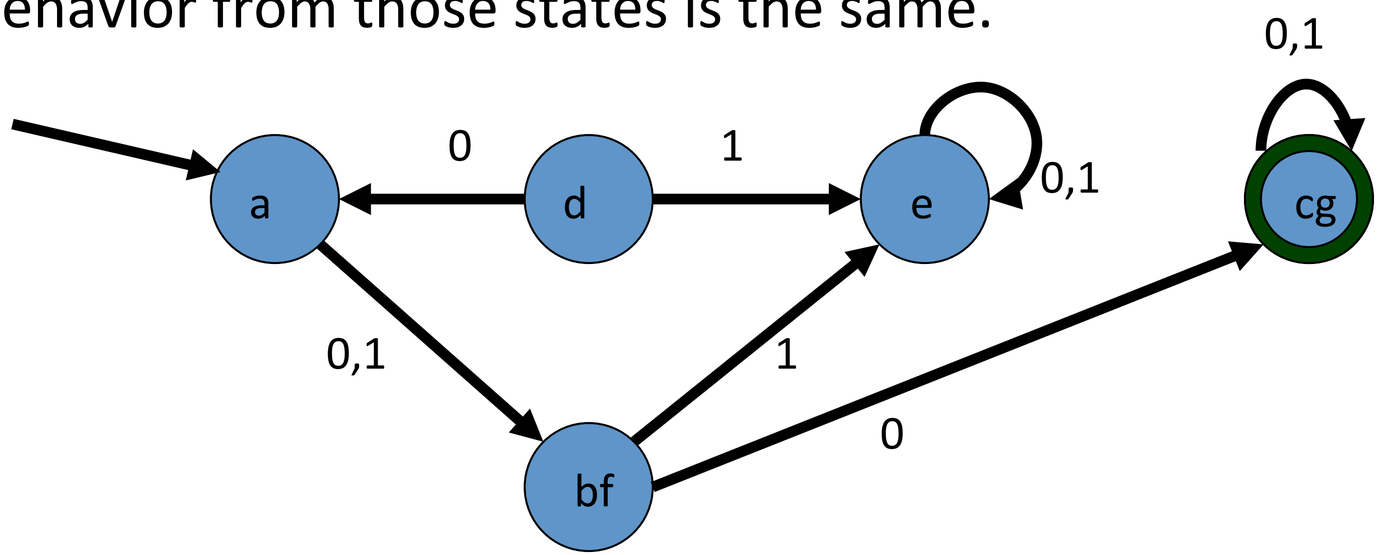
1. if string ends, reject in both cases
2. if next character is 0, forever accept in both cases
3. if next character is 1, forever reject in both cases

So merge **b** with **f**.



Equivalent States: Definition

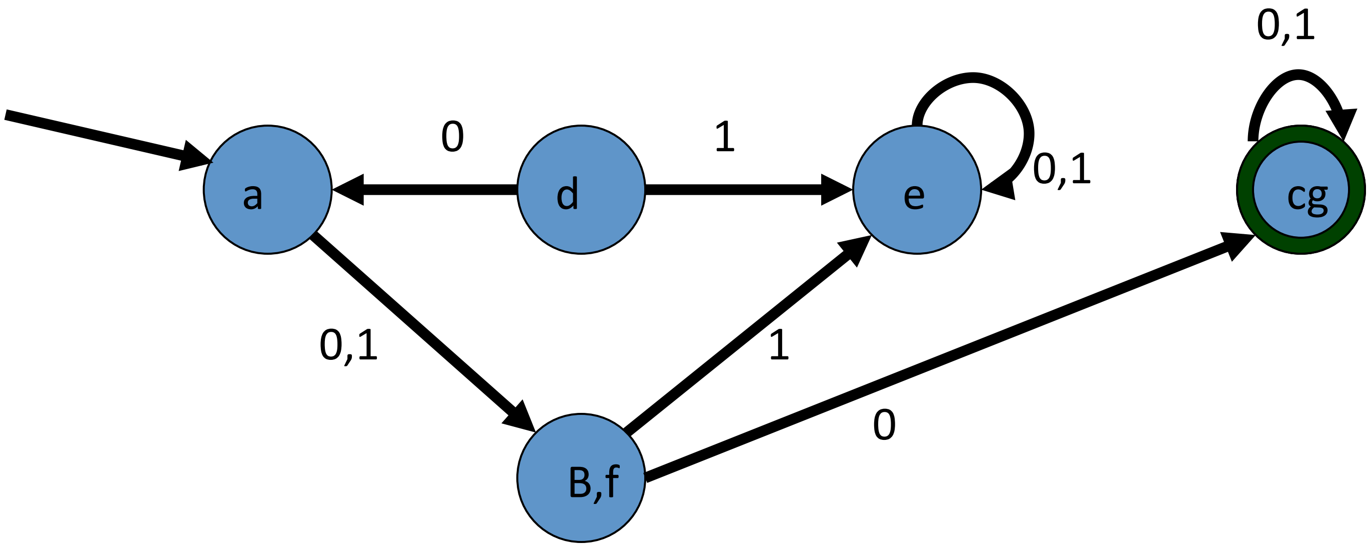
Intuitively two states are equivalent if all subsequent behavior from those states is the same.



DEF: Two states q and q' in a DFA $M = (Q, \Sigma, \delta, q_0, F)$ are ***equivalent*** (or ***indistinguishable***) if for all strings $u \in \Sigma^*$, the states on which u ends on when read from q and q' are both *accept*, or both *non-accept*.

Finishing the Example

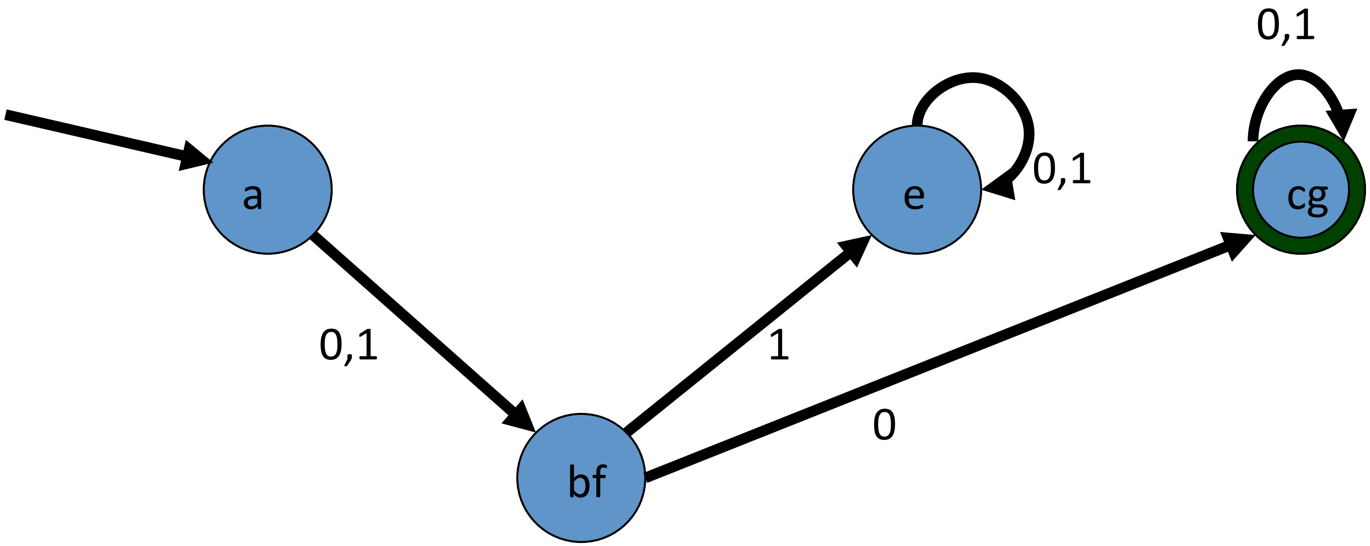
Q: Any other ways to simplify the automaton?



Useless States

A: Get rid of **d**.

Getting rid of unreachable ***useless states*** doesn't affect the accepted language.



Minimization Algorithm: Goals

DEF: An automaton is ***irreducible*** if
- it contains no useless states, and
- no two distinct states are equivalent.

The goal of the **Minimization Algorithm** is to create an irreducible automata from an arbitrary one, accepting the same language.

The minimization algorithm incrementally builds a **partition** of the states of the given DFA:
- It starts with a partition separating just accepting/non accepting states
- Next it splits an equivalence class if it contains two non equivalent states

Minimization Algorithm. (Partition Refinement) Code

```
DFA minimize(DFA (Q, S, d, q0, F ) )
 remove any state q unreachable from q0
 Partition P = {F, Q - F }
 boolean Consistent = false
 while ( Consistent == false ) Consistent = true
  for(every Set S ∈ P, char a ∈ S, Set T ∈ P )
            // collect states of T that reach S using a
      Set  temp = {q ∈ T | d(q,a) ∈ S }
      if (temp != Ø  && temp != T )
            Consistent = false
            P = (P -T )∪{temp,T-temp}
 return defineMinimizor( (Q, S, d, q0, F ), P )
```

Minimization Algorithm. (Partition Refinement) Code

DFA defineMinimizor (DFA $(Q, \Sigma, \delta, q_0, F)$, Partition P)

Set $Q' = P$

State q'_0 = the set in P which contains q_0

$F' = \{ S \in P \mid S \subseteq F \}$

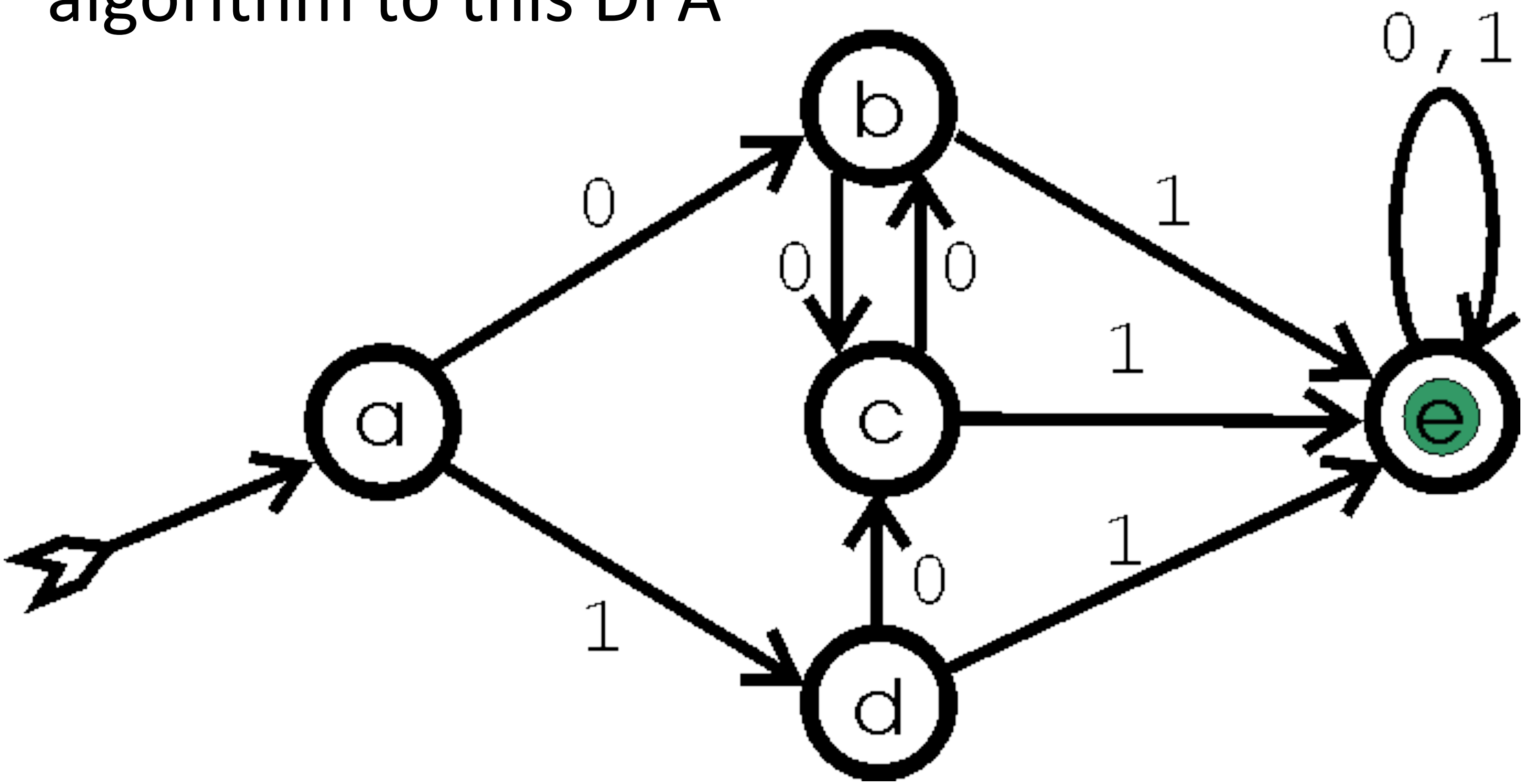
for (each $S \in P$, $a \in \Sigma$)

define $\delta'(S,a)$ = the set $T \in P$ which contains the states $\delta'(S,a)$

return $(Q', \Sigma, \delta', q'_0, F')$

Minimization Algorithm: Example

Show the result of applying the minimization algorithm to this DFA



Proof of Minimal Automaton

Previous algorithm guaranteed to produce an ***irreducible*** DFA. Why should that FA be the smallest possible FA for its accepted language?

Analogous question in calculus: Why should a local minimum be a global minimum? *Usually* not the case!

Proof of Minimal Automaton

THM (Myhill-Nerode): *The minimization algorithm produces the smallest possible automaton for its accepted language.*

Proof. Show that any irreducible automaton is the smallest for its accepted language L:

We say that two strings $u,v \in \Sigma^*$ are ***indistinguishable*** if for all strings x, $ux \in L \Leftrightarrow vx \in L$.

Notice that if u and v *are* **distinguishable**, their paths from the start state must have different endpoints.

Proof of Minimal Automaton

Consequently, the number of states in any DFA for *L* must be as great as the number of mutually distinguishable strings for *L*.

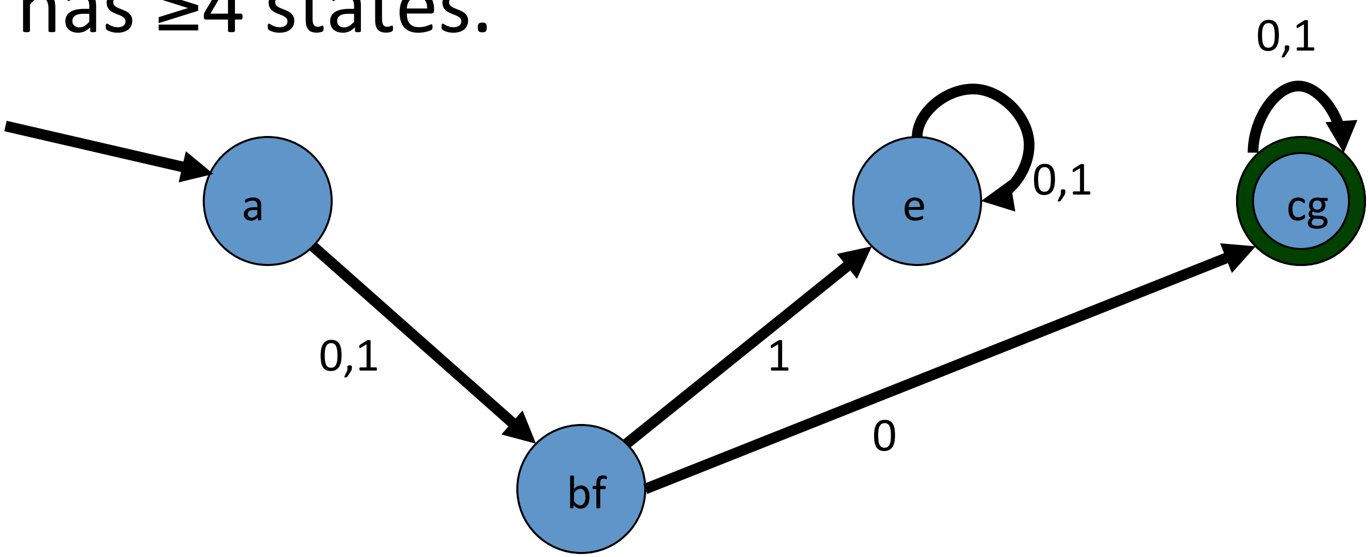
But an irreducible DFA has the property that every state gives rise to another mutually distinguishable string!

Therefore, any other DFA must have at least as many states as the irreducible DFA

Let's see how the proof works on a previous example:

Proof of Minimal Automaton: Example

The "spanning tree of strings" {ε,0,01,00} is a mutually distinguishable set (otherwise redundancy would occur and hence DFA would be reducible). Any other DFA for *L* has ≥4 states.



Exercises on Lexical Analysis

3.1.1 Divide the following C++ program into appropriate lexemes:

```
float limitedSquare(x){float x;
  /* returns x-squared, but never more than 100 */
  return (x <= -10.0 || x >= 10.0) ? 100 : x*x;
}
```

Which lexemes should get associated lexical values? What should those values be?

From RE to Automata and backwards

- We have seen:
  - RE → NFA
  - NFA → DFA [and obviously DFA → NFA]
  - RE → DFA, directly
  - DFA → minimal DFA
- What about NFA, DFA → RE? More difficult. Three approaches (not presented):
    - Dynamic Programming [Scott Section 2.4 on CD][Hopcroft, Motwani, Ullman, Section 3.2.1]
    - Incremental state elimination [HMU, Section 3.2.2]
    - RE as fixpoint solution of system of language equations

  [uses right-linear grammars for Regular Languages]

Exercises on Regular Expressions

3.3.2 Describe the languages denoted by the following regular expressions:

- b) $((\varepsilon|a)b^{*})^{*}$
- c) $(a|b)^{*}a(a|b)(a|b)$

3.3.5 Write regular definitions for the following languages:

- b) All strings of lowercase letters in which the letters are in ascending lexicographic order.
- c) Comments, consisting of a string surrounded by /* and */, without an intervening */, unless it is inside double-quotes (”)
- i) All strings of a's and b's that do not contain the subsequence **abb**.

Exercises with Lex or Flex

- 3.5.2 Write a Lex program that copies a file, replacing each non-empty sequence of white spaces by a single blank.
- 3.5.3 Write a Lex program that copies a C program, replacing each instance of the keyword `float` by `double`.

Exercises on Finite Automata

- 3.6.2 Design finite automata for the following languages (providing both the transition graph and the transition table):

  a) All strings of lowercase letters that contain the five vowels in order.

  d) All strings of digits with no repeated digits. Hint: Try this problem first with a few digits, such as {O, 1, 2}.

  f) All strings of a's and b's with an even number of a's and an odd number of b's.

Exercises: from RE to DFA

3.7.3 Convert the following regular expressions to deterministic finite automata, using the [McNaughton-Yamada-]Thompson algorithm (3.23) and the *subset construction* algorithm (3.20):

a) $(a|b)^*$

b) $(a^*|b^*)^*$

c) $((\varepsilon|a)|b^*)^*$

d) $(a|b)^*abb(a|b)^*$

Exercises: Minimizing DFA

- 3.9.3 Show that the RE
  - a) $(a|b)^*$
  - b) $(a^*|b^*)^*$
  - c) $((\varepsilon|a)|b^*)^*$

are equivalent by showing that their minimum state DFA's are isomorphic.