



<http://algs4.cs.princeton.edu>

## 5.4 REGULAR EXPRESSIONS

---

- ▶ *regular expressions*
- ▶ *REs and NFAs*
- ▶ *NFA simulation*
- ▶ *NFA construction*
- ▶ *applications*



<http://algs4.cs.princeton.edu>

## 5.4 REGULAR EXPRESSIONS

---

- ▶ *regular expressions*
- ▶ *REs and NFAs*
- ▶ *NFA simulation*
- ▶ *NFA construction*
- ▶ *applications*

# Pattern matching

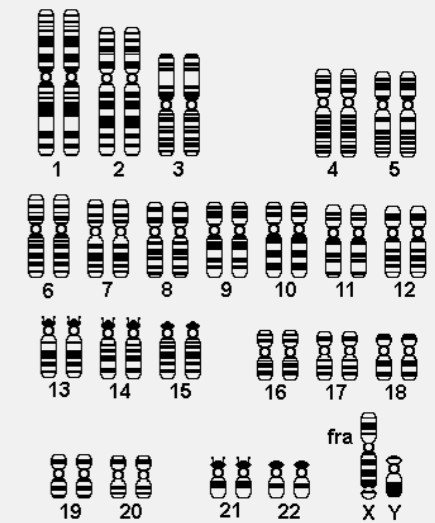
---

**Substring search.** Find a single string in text.

**Pattern matching.** Find one of a **specified set** of strings in text.

**Ex.** [genomics]

- Fragile X syndrome is a common cause of mental retardation.
- A human's genome is a string.
- It contains triplet repeats of CGG or AGG, bracketed by GCG at the beginning and CTG at the end.
- Number of repeats is variable and is correlated to syndrome.



**pattern** GCG(CGG|AGG)\*CTG

**text** GCGGCGTGTGTGCGAGAGAGTGGGTTTAAAGCTGGCGCGGAGGCGGCTGGCGCGGAGGCTG

# Syntax highlighting

---

```
/*  
 * Compilation: javac NFA.java  
 * Execution: java NFA regexp text  
 * Dependencies: Stack.java Bag.java Digraph.java DirectedDFS.java  
 *  
 * % java NFA "(A*B|AC)D" AAAABD  
 * true  
 *  
 * % java NFA "(A*B|AC)D" AAAAC  
 * false  
 *  
 *****/
```

```
public class NFA  
{  
    private Digraph G;           // digraph of epsilon transitions  
    private String regexp;      // regular expression  
    private int M;              // number of characters in regular expression  
  
    // Create the NFA for the given RE  
    public NFA(String regexp)  
    {  
        this.regexp = regexp;  
        M = regexp.length();  
        Stack<Integer> ops = new Stack<Integer>();  
        G = new Digraph(M+1);  
        ...  
    }  
}
```

GNU source-highlight 3.1.4

input	output
Ada	HTML
Asm	XHTML
Applescript	LATEX
Awk	MediaWiki
Bat	ODF
Bib	TEXINFO
Bison	ANSI
C/C++	DocBook
C#	
Cobol	
Caml	
Changelog	
Css	
D	
Erlang	
Flex	
Fortran	
GLSL	
Haskell	
Html	
Java	
Javalog	
Javascript	
Latex	
Lisp	
Lua	
:	

# Google code search

---

## Search public source code

Search via regular expression, e.g. `^java/.*\.java$`

### Search Options

### In Search Box

Package	<input type="text"/>	package:linux-2.6
Language	<input type="text" value="Any language"/>	lang:c++
File Path	<input type="text"/>	file:(code  [ <sup>^</sup> or]g)search
Class	<input type="text"/>	class:HashMap
Function	<input type="text"/>	function:toString
License	<input type="text" value="Any license"/>	license:mozilla
Case Sensitive	<input type="text" value="No"/>	case:yes

<http://code.google.com/p/chromium/source/search>

# Prosite (computational biochemistry)

---

[Home](#) | [ScanProsite](#) | [ProRule](#) | [Documents](#) | [Downloads](#) | [Links](#) | [Funding](#)



## Database of protein domains, families and functional sites

PROSITE consists of documentation entries describing protein domains, families and functional sites as well as associated patterns and profiles to identify them [[More...](#) / [References](#) / [Commercial users](#)].

PROSITE is complemented by [ProRule](#), a collection of rules based on profiles and patterns, which increases the discriminatory power of profiles and patterns by providing additional information about functionally and/or structurally critical amino acids [[More...](#)].

**Release 20.113 of 26-Mar-2015 contains 1718 documentation entries, 1308 patterns, 1112 profiles and 1112 ProRule.**

Search

e.g. PDOC00022, PS50089, SH3, zinc finger

Search

type an RE here

Browse

- by documentation entry
- by ProRule description
- by taxonomic scope
- by number of positive hits

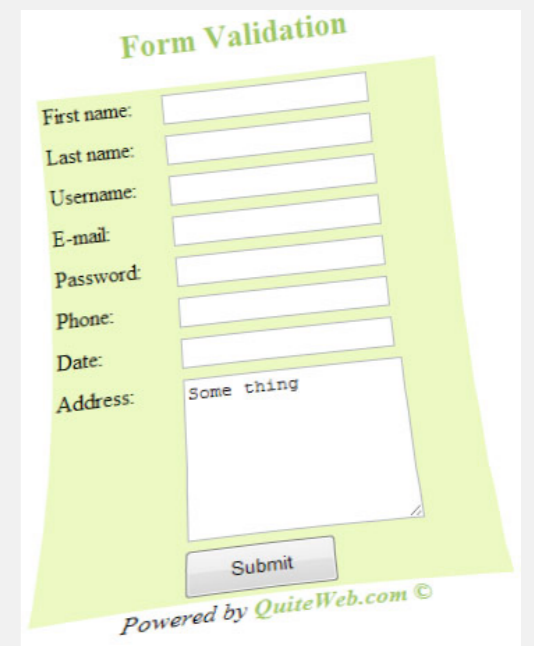
<http://prosite.expasy.org>

# Pattern matching: applications

---

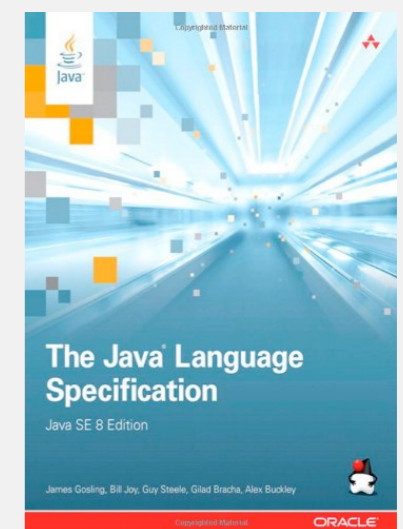
## Test if a string matches some pattern.

- Scan for virus signatures.
- Process natural language.
- Specify a programming language.
- Access information in digital libraries.
- Search genome using PROSITE patterns.
- Filter text (spam, NetNanny, Carnivore, malware).
- Validate data-entry fields (dates, email, URL, credit card).
- ...



## Parse text files.

- Compile a Java program.
- Crawl and index the Web.
- Read in data stored in ad hoc input file format.
- Create Java documentation from Javadoc comments.
- ...



# Regular expressions

A **regular expression** is a notation to specify a set of strings.

↑  
possibly infinite

operation	order	example RE	matches	does not match
<b>concatenation</b>	3	AABAAB	AABAAB	<i>every other string</i>
<b>or</b>	4	AA   BAAB	AA BAAB	<i>every other string</i>
<b>closure</b>	2	AB*A	AA ABBBBBBBBA	AB ABABA
<b>parentheses</b>	1	A(A B)AAB	AAAAB ABAAB	<i>every other string</i>
		(AB)*A	A ABABABABABA	AA ABBA



# Regular expression: quiz 1

---

Which one of the following strings is **not** matched by the regular expression  $(A B | C * D) * ?$

**A.** A B A B A B

**B.** C D C C D D D D

**C.** A B C C D A B

**D.** A B D A B C A B D

**E.** *I don't know.*

# Regular expression shortcuts

---

Additional operations further extend the utility of REs.

operation	example RE	matches	does not match
<b>wildcard</b>	<code>.U.U.U.</code>	CUMULUS JUGULUM	SUCCUBUS TUMULTUOUS
<b>character class</b>	<code>[A-Za-z][a-z]*</code>	word Capitalized	camelCase 4illegal
<b>one or more</b>	<code>A(BC)+DE</code>	ABCDE ABCBCDE	ADE BCDE
<b>exactly k</b>	<code>[0-9]{5}-[0-9]{4}</code>	08540-1321 19072-5541	111111111 166-54-111

**Note.** These operations are useful but not essential.

**Ex.** `[A-E]+` is shorthand for `(A|B|C|D|E)(A|B|C|D|E)*`

# Regular expression examples

---

RE notation is surprisingly expressive.

regular expression	matches	does not match
<code>. *SPB. *</code> <i>(substring search)</i>	RASPBERRY CRISPBREAD	SUBSPACE SUBSPECIES
<code>[0-9]{3}-[0-9]{2}-[0-9]{4}</code> <i>(U. S. Social Security numbers)</i>	166-11-4433 166-45-1111	11-55555555 8675309
<code>[a-z]+@([a-z]+\.)+(edu com)</code> <i>(simplified email addresses)</i>	wayne@princeton.edu rs@princeton.edu	spam@nowhere
<code>[\$_A-Za-z][\$_A-Za-z0-9]*</code> <i>(Java identifiers)</i>	ident3 PatternMatcher	3a ident#3

REs play a well-understood role in the theory of computation.

## Regular expression: quiz 2

---

Which of the following REs match **genes**:

- (1) alphabet is { A, G, T, C }
- (2) length is a multiple of 3
- (3) starts with ATG (a start codon)
- (4) ends with TAG or TAA or TTG (a stop codon)



- A.**  $ATG(A|C|T|G)(A|C|T|G)(A|C|T|G)^*(TAG|TAA|TTG)$
- B.**  $ATG([AGTC]\{3\})^*(TAG|TAA|TTG)$
- C.** Both A and B.
- D.** Neither A nor B.
- E.** *I don't know.*

# Illegally screening a job candidate

---

“ [First name]! and pre/2 [last name] w/7  
bush or gore or republican! or democrat! or charg!  
or accus! or criticiz! or blam! or defend! or iran contra  
or clinton or spotted owl or florida recount or sex!  
or controvers! or fraud! or investigat! or bankrupt!  
or layoff! or downsiz! or PNTR or NAFTA or outsourc!  
or indict! or enron or kerry or iraq or wmd! or arrest!  
or intox! or fired or racis! or intox! or slur!  
or controvers! or abortion! or gay! or homosexual!  
or gun! or firearm! ”

— *LexisNexis search pattern used by Monica Goodling  
to screen candidates for DOJ positions*



LexisNexis™

<http://www.justice.gov/oig/special/s0807/final.pdf>

# Regular expressions to the rescue





# Regular expression caveat

---

Writing a RE is like writing a program.

- Need to understand programming model.
- Can be easier to write than read.
- Can be difficult to debug.



*“ Some people, when confronted with a problem, think 'I know I'll use regular expressions.' Now they have two problems. ”*

*— Jamie Zawinski (flame war on alt.religion.emacs)*

**Bottom line.** REs are amazingly powerful and expressive, but using them in applications can be amazingly complex and error-prone.





<http://algs4.cs.princeton.edu>

## 5.4 REGULAR EXPRESSIONS

---

- ▶ *regular expressions*
- ▶ **REs and NFAs**
- ▶ *NFA simulation*
- ▶ *NFA construction*
- ▶ *applications*

# Duality between REs and DFAs

---

**RE.** Concise way to describe a set of strings.

**DFA.** Machine to recognize whether a given string is in a given set.

## Kleene's theorem.

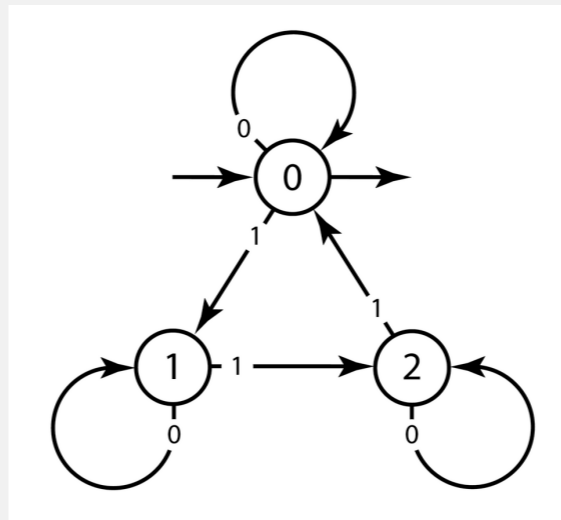
- For any DFA, there exists a RE that describes the same set of strings.
- For any RE, there exists a DFA that recognizes the same set of strings.

RE

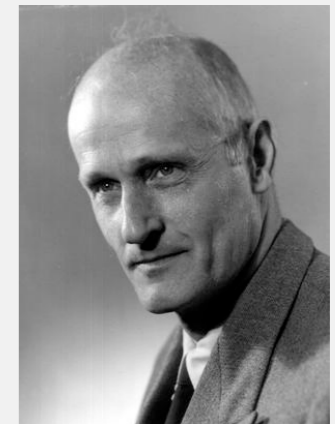
$0^* \mid (0^*10^*10^*10^*)^*$

number of 1's is a multiple of 3

DFA



number of 1's is a multiple of 3

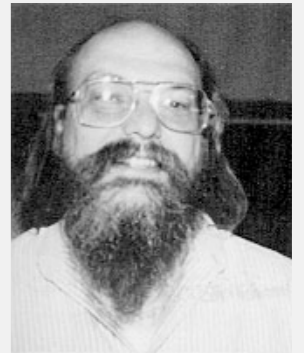


**Stephen Kleene**  
Princeton Ph.D. 1934

# Pattern matching implementation: basic plan (first attempt)

Overview is the same as for KMP.

- No backup in text input stream.
- Linear-time guarantee.

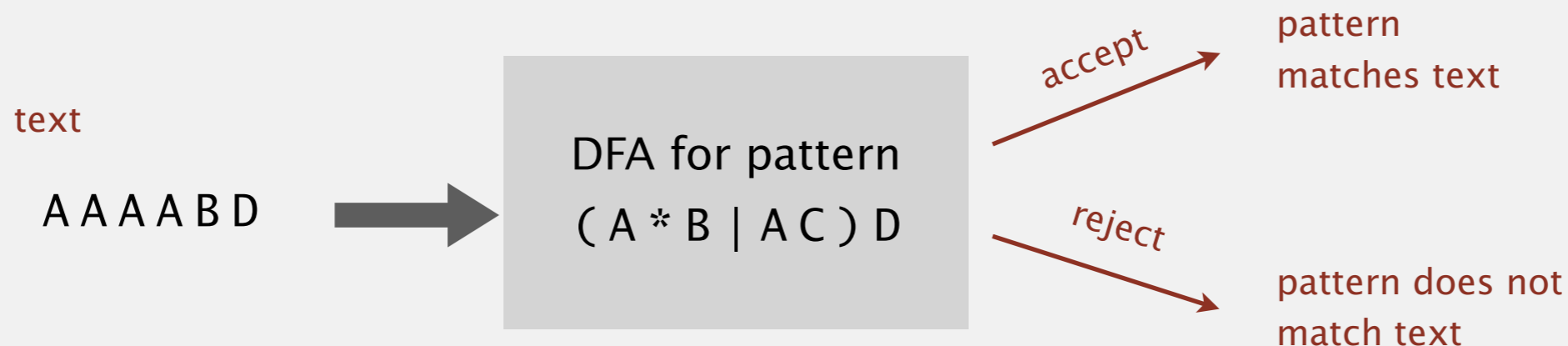


Ken Thompson  
Turing Award '83

Underlying abstraction. Deterministic finite state automata (DFA).

Basic plan. [apply Kleene's theorem]

- Build DFA from RE.
- Simulate DFA with text as input.

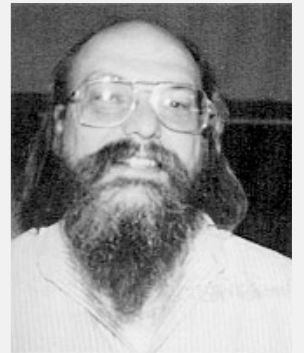


Bad news. Basic plan is infeasible (DFA may have exponential # of states).

# Pattern matching implementation: basic plan (revised)

Overview is similar to KMP.

- No backup in text input stream.
- **Quadratic-time guarantee** (linear-time typical).

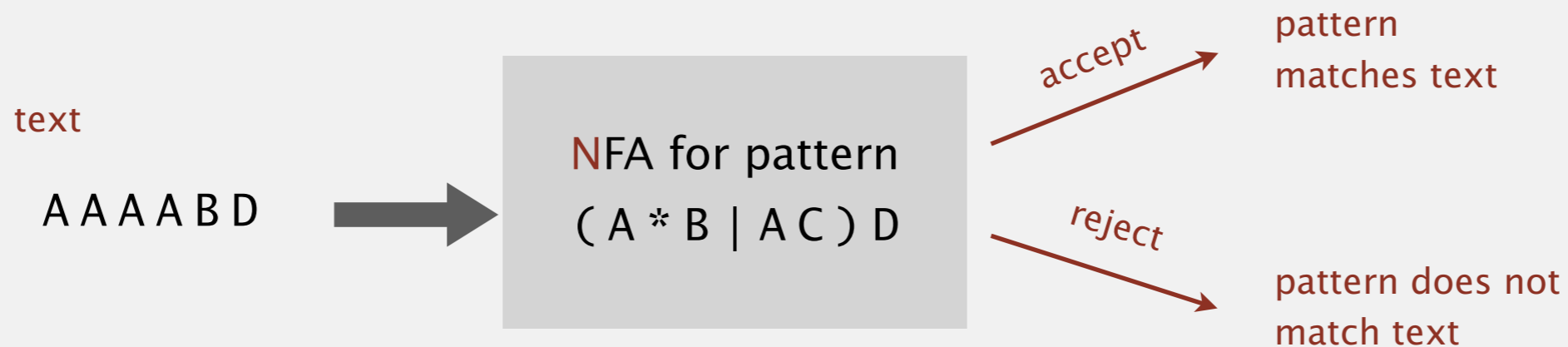


Ken Thompson  
Turing Award '83

Underlying abstraction. **N**ondeterministic finite state automata (**NFA**).

Basic plan. [apply Kleene's theorem]

- Build **NFA** from RE.
- Simulate **NFA** with text as input.



Q. What is an NFA?

# Nondeterministic finite-state automata

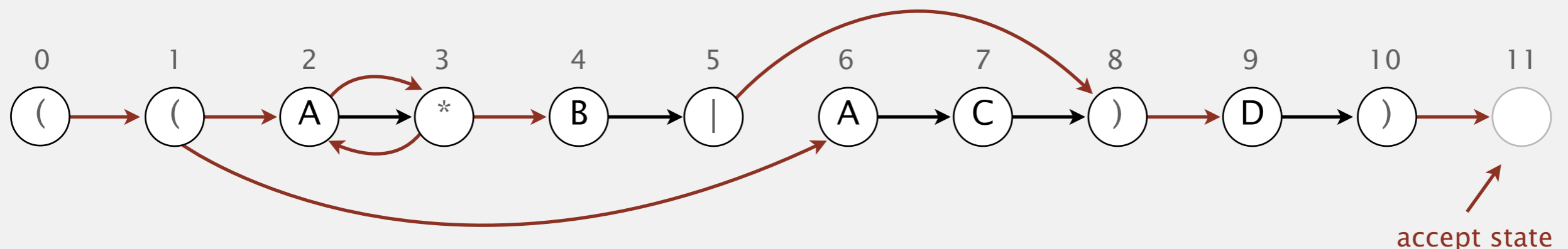
## Regular-expression-matching NFA.

- We assume RE enclosed in parentheses.
- One state per RE character (start = 0, accept =  $M$ ).
- Red  $\epsilon$ -transition (change state, but don't scan text).
- Black match transition (change state and scan to next text char).
- Accept if **any** sequence of transitions ends in accept state.

after scanning all text characters

## Nondeterminism.

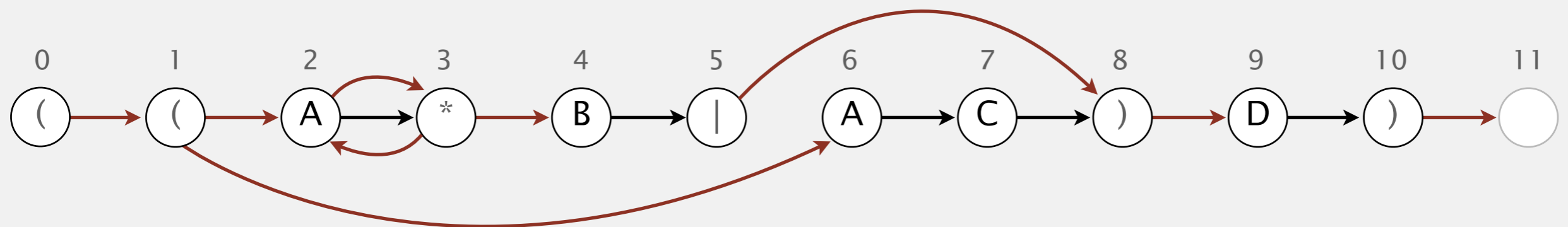
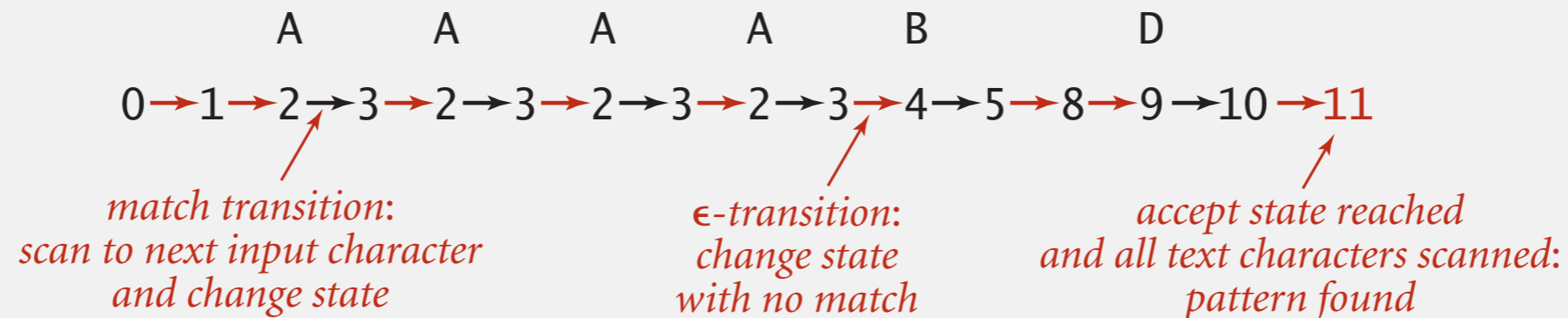
- One view: machine can guess the proper sequence of state transitions.
- Another view: sequence is a proof that the machine accepts the text.



# Nondeterministic finite-state automata

Q. Is A A A B D matched by NFA?

A. Yes, because **some** sequence of legal transitions ends in state 11.

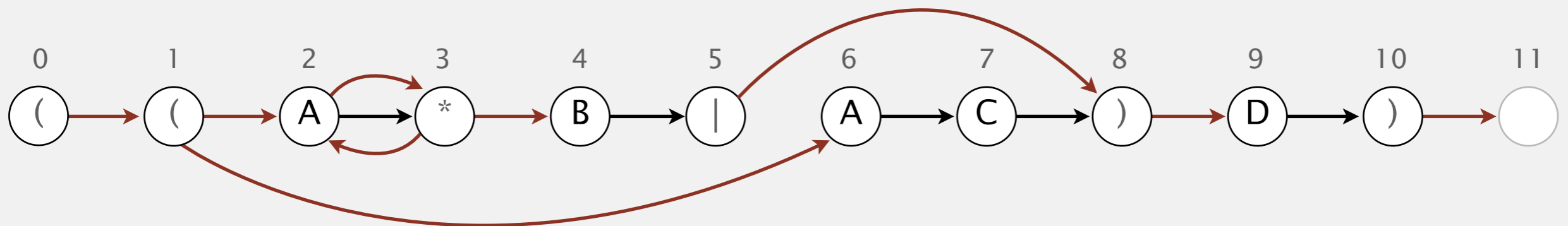
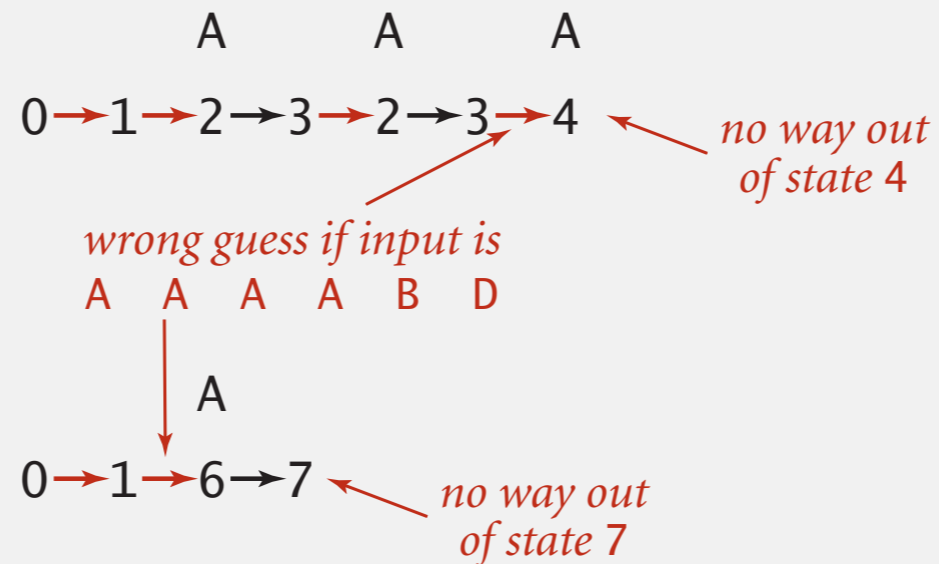


NFA corresponding to the pattern  $( ( A * B | A C ) D )$

# Nondeterministic finite-state automata

Q. Is A A A B D matched by NFA?

A. Yes, because **some** sequence of legal transitions ends in state 11.  
[ even though some sequences end in wrong state or get stuck ]



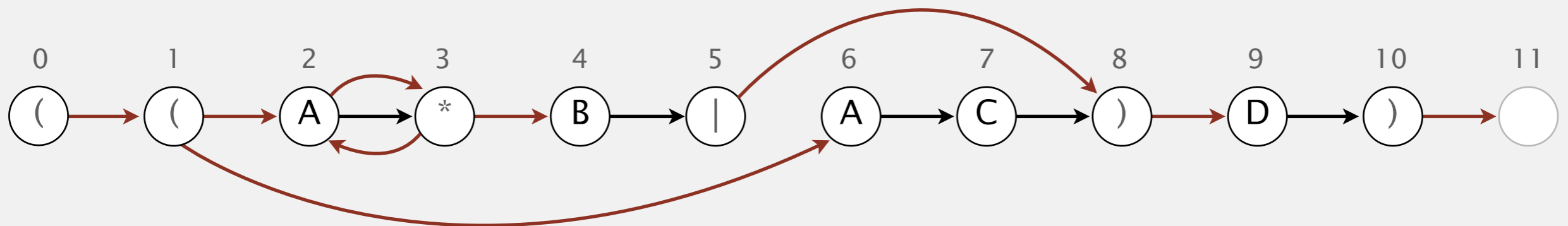
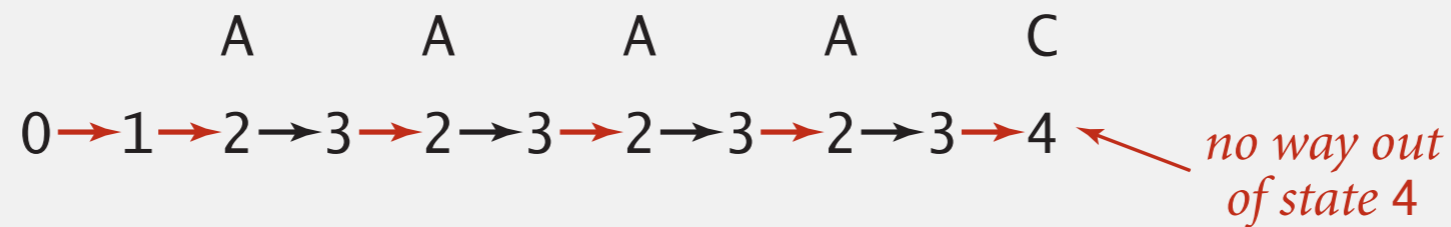
NFA corresponding to the pattern ( ( A \* B | A C ) D )

# Nondeterministic finite-state automata

Q. Is A A A C matched by NFA?

A. No, because **no** sequence of legal transitions ends in state 11.

[ but need to argue about all possible sequences ]



NFA corresponding to the pattern ( ( A \* B | A C ) D )



# Nondeterminism

---

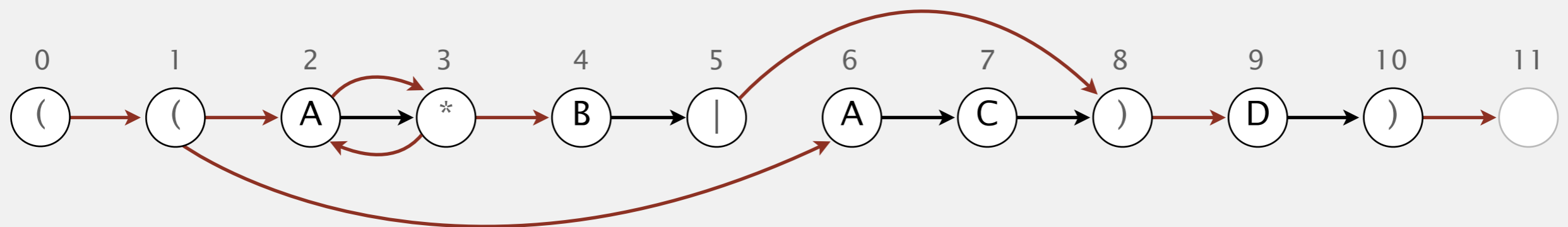
Q. How to determine whether a string is matched by an automaton?

DFA. Deterministic  $\Rightarrow$  easy (only one applicable transition at each step).

NFA. Nondeterministic  $\Rightarrow$  hard (can be several applicable transitions at each step; need to select the "right" ones!)

Q. How to simulate NFA?

A. Systematically consider **all** possible transition sequences. [stay tuned]



NFA corresponding to the pattern ( ( A \* B | A C ) D )



<http://algs4.cs.princeton.edu>

## 5.4 REGULAR EXPRESSIONS

---

- ▶ *regular expressions*
- ▶ *REs and NFAs*
- ▶ ***NFA simulation***
- ▶ *NFA construction*
- ▶ *applications*

# NFA representation

State names. Integers from 0 to  $M$ .

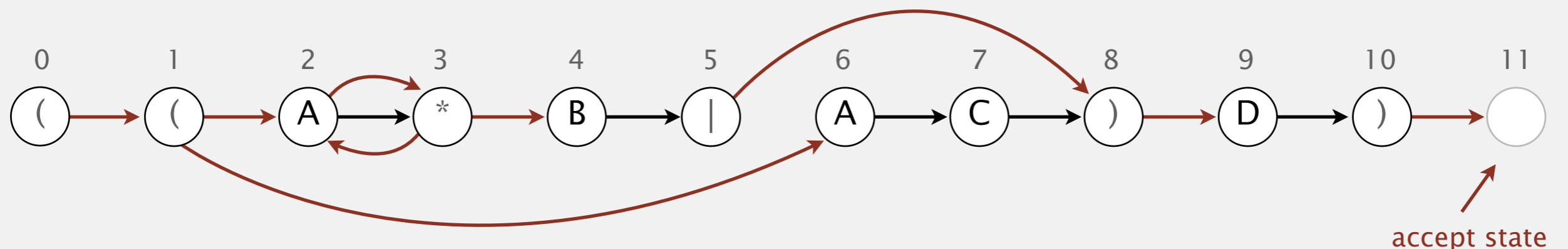
number of symbols in RE

Match-transitions. Keep regular expression in array `re[]`.

	0	1	2	3	4	5	6	7	8	9	10
<code>re[]</code>	(	(	A	*	B		A	C	)	D	)

$\epsilon$ -transitions. Store in a **digraph**  $G$ .

$0 \rightarrow 1$ ,  $1 \rightarrow 2$ ,  $1 \rightarrow 6$ ,  $2 \rightarrow 3$ ,  $3 \rightarrow 2$ ,  $3 \rightarrow 4$ ,  $5 \rightarrow 8$ ,  $8 \rightarrow 9$ ,  $10 \rightarrow 11$



NFA corresponding to the pattern `( ( A * B | A C ) D )`

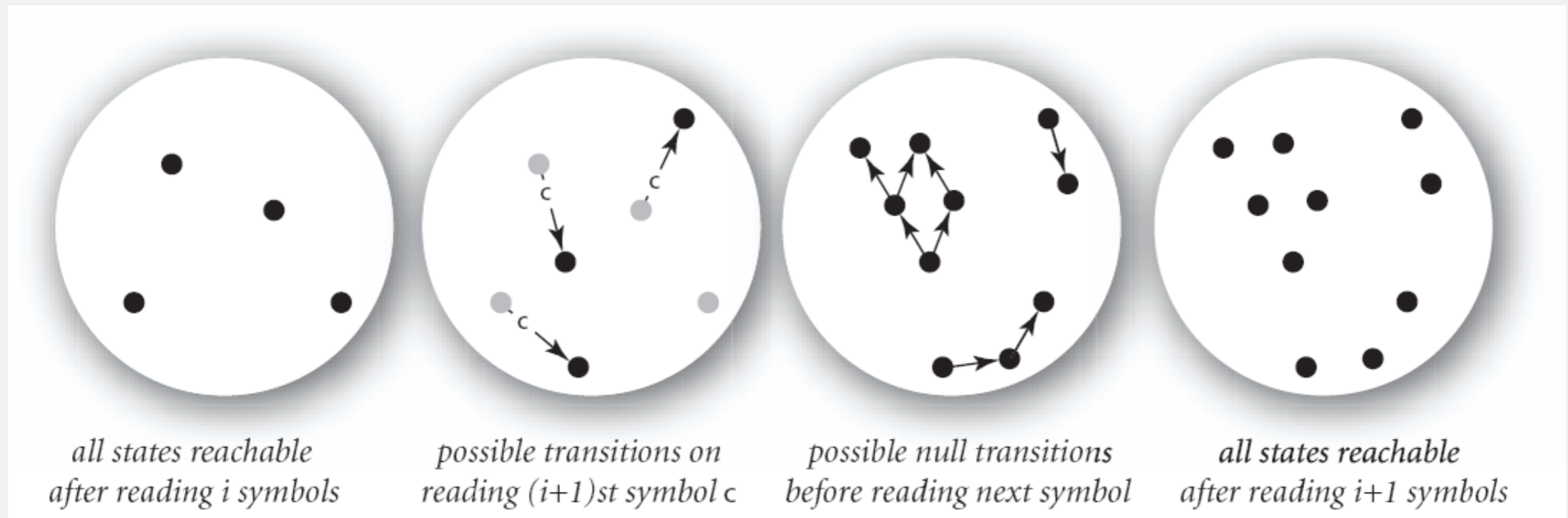
# NFA simulation

---

Q. How to efficiently simulate an NFA?

A. Maintain set of **all** possible states that NFA could be in after reading in the first  $i$  text characters.

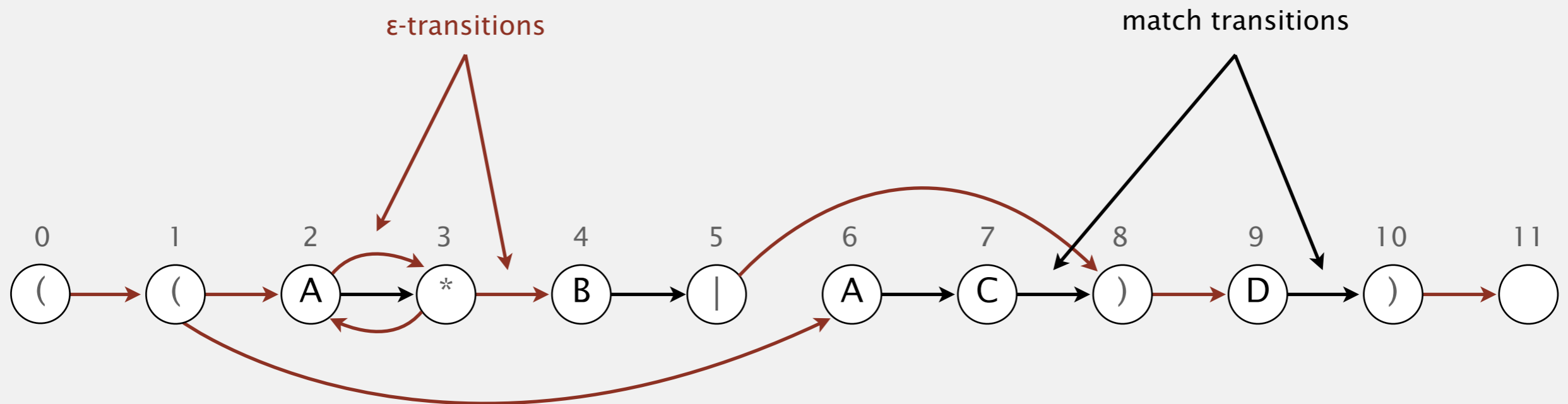
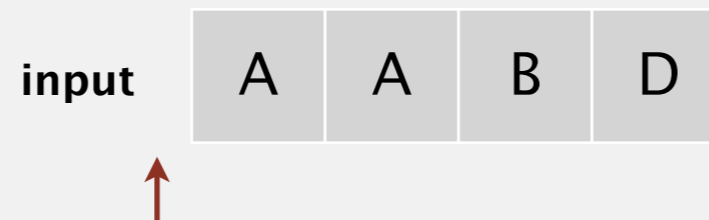
one step in simulating an NFA



Q. How to perform reachability?

# NFA simulation demo

**Goal.** Check whether input matches pattern.



NFA corresponding to the pattern  $( ( A * B | A C ) D )$

# Digraph reachability review

---

**Goal.** Find all vertices reachable from a given vertex or **set** of vertices.

 recall Section 4.2

```
public class DirectedDFS
```

```
    DirectedDFS(Digraph G, int s)
```

*find vertices reachable from s*

```
    DirectedDFS(Digraph G, Iterable<Integer> s)
```

*find vertices reachable from sources*

```
    boolean marked(int v)
```

*is v reachable from source(s)?*

**Solution.** Run DFS from each source, without unmarking vertices.

**Performance.** Runs in time proportional to  $E + V$ .

# NFA simulation: Java implementation

---

```
public class NFA
{
    private char[] re;        // match transitions
    private Digraph G;       // epsilon transition digraph
    private int M;          // number of states

    public NFA(String regexp)
    {
        M = regexp.length();
        re = regexp.toCharArray();
        G = buildEpsilonTransitionDigraph();
    }

    public boolean recognizes(String txt)
    { /* see next slide */ }

    public Digraph buildEpsilonTransitionDigraph()
    { /* stay tuned */ }
}
```

← stay tuned

# NFA simulation: Java implementation

```
public boolean recognizes(String txt)
{
```

```
    Bag<Integer> pc = new Bag<Integer>();
    DirectedDFS dfs = new DirectedDFS(G, 0);
    for (int v = 0; v < G.V(); v++)
        if (dfs.marked(v)) pc.add(v);
```

← states reachable from start by  $\epsilon$ -transitions

```
    for (int i = 0; i < txt.length(); i++)
    {
```

```
        Bag<Integer> states = new Bag<Integer>();
        for (int v : pc)
        {
            if (v == M) continue;
            if ((re[v] == txt.charAt(i)) || re[v] == '.')
                states.add(v+1);
        }
```

← set of states reachable after scanning past `txt.charAt(i)`

← not necessarily a match (RE needs to match full text)

```
        dfs = new DirectedDFS(G, states);
        pc = new Bag<Integer>();
        for (int v = 0; v < G.V(); v++)
            if (dfs.marked(v)) pc.add(v);
```

← follow  $\epsilon$ -transitions

```
    }
    for (int v : pc)
        if (v == M) return true;
    return false;
```

← accept if can end in state M

```
}
```

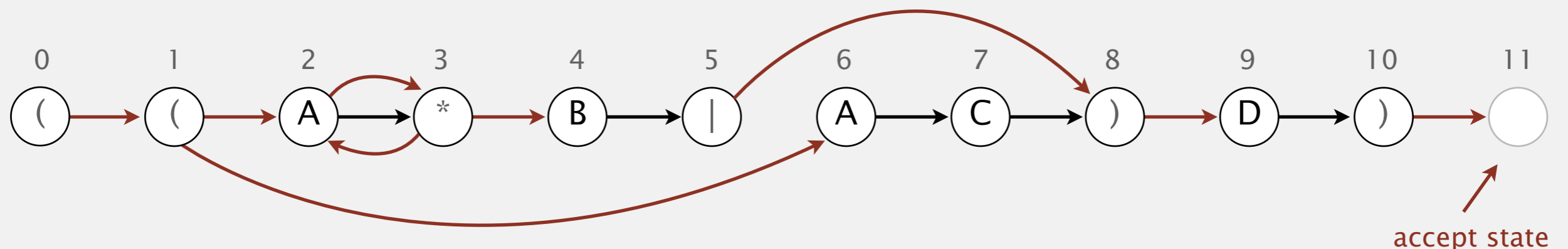


# NFA simulation: analysis

**Proposition.** Determining whether an  $N$ -character text is recognized by the NFA corresponding to an  $M$ -character pattern takes time proportional to  $MN$  in the worst case.

**Pf.** For each of the  $N$  text characters, we iterate through a set of states of size no more than  $M$  and run DFS on the graph of  $\epsilon$ -transitions.

[The NFA construction we will consider ensures the number of edges  $\leq 3M$ .]



NFA corresponding to the pattern  $( ( A * B | A C ) D )$



<http://algs4.cs.princeton.edu>

## 5.4 REGULAR EXPRESSIONS

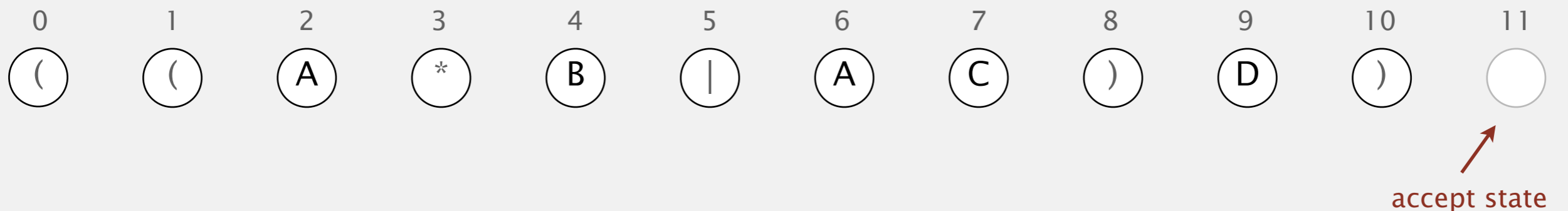
---

- ▶ *regular expressions*
- ▶ *REs and NFAs*
- ▶ *NFA simulation*
- ▶ ***NFA construction***
- ▶ *applications*

# Building an NFA corresponding to an RE

---

**States.** Include a state for each symbol in the RE, plus an accept state.



NFA corresponding to the pattern  $((A * B | A C ) D )$

# Building an NFA corresponding to an RE

---

**Concatenation.** Add match-transition edge from state corresponding to characters in the alphabet to next state.

**Alphabet.** A B C D

**Metacharacters.** ( ) . \* |

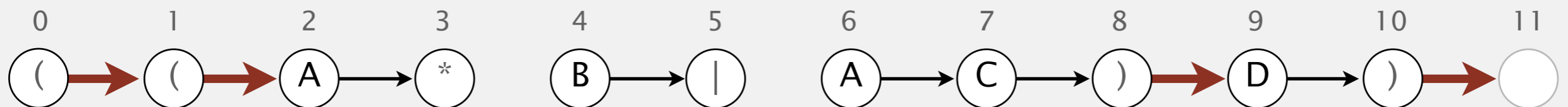


**NFA corresponding to the pattern ( ( A \* B | A C ) D )**

# Building an NFA corresponding to an RE

---

**Parentheses.** Add  $\epsilon$ -transition edge from parentheses to next state.

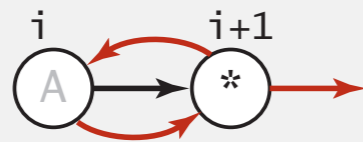


**NFA corresponding to the pattern  $((A * B | A C) D)$**

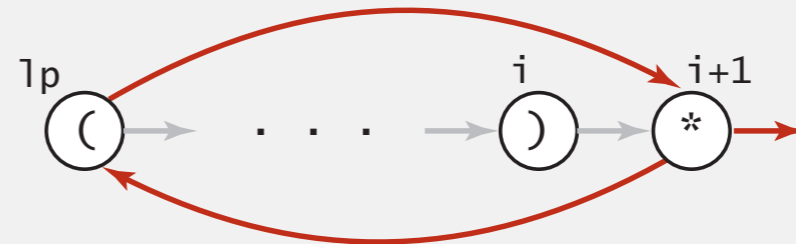
# Building an NFA corresponding to an RE

**Closure.** Add three  $\epsilon$ -transition edges for each  $*$  operator.

single-character closure



closure expression

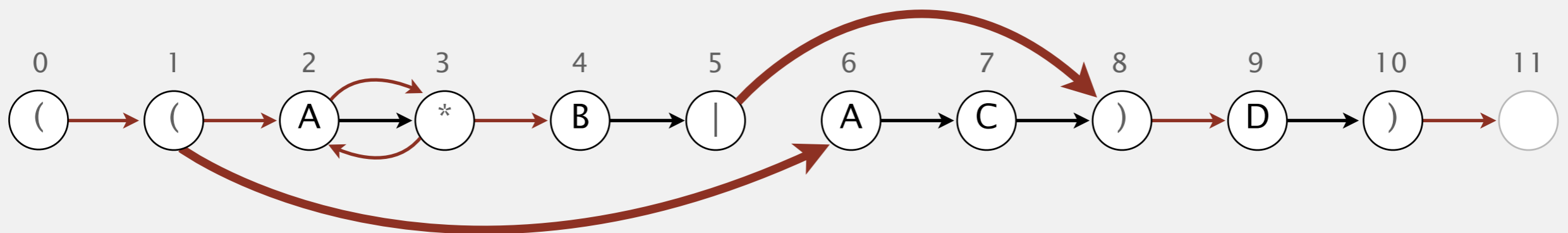
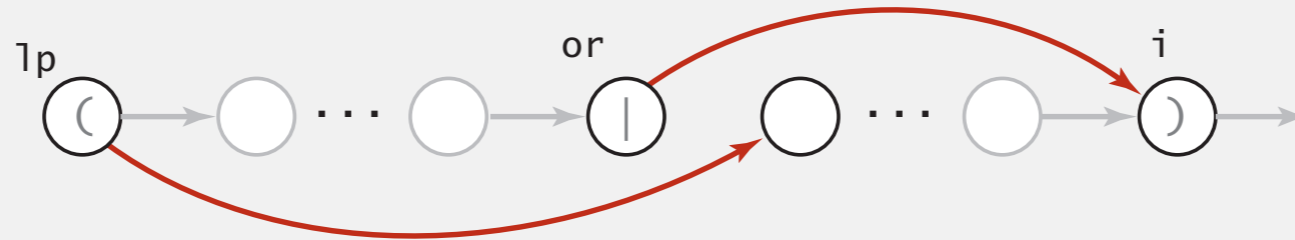


NFA corresponding to the pattern  $((A*B|AC)D)$

# Building an NFA corresponding to an RE

2-way or. Add two  $\epsilon$ -transition edges for each | operator.

or expression



NFA corresponding to the pattern  $((A * B | A C ) D )$

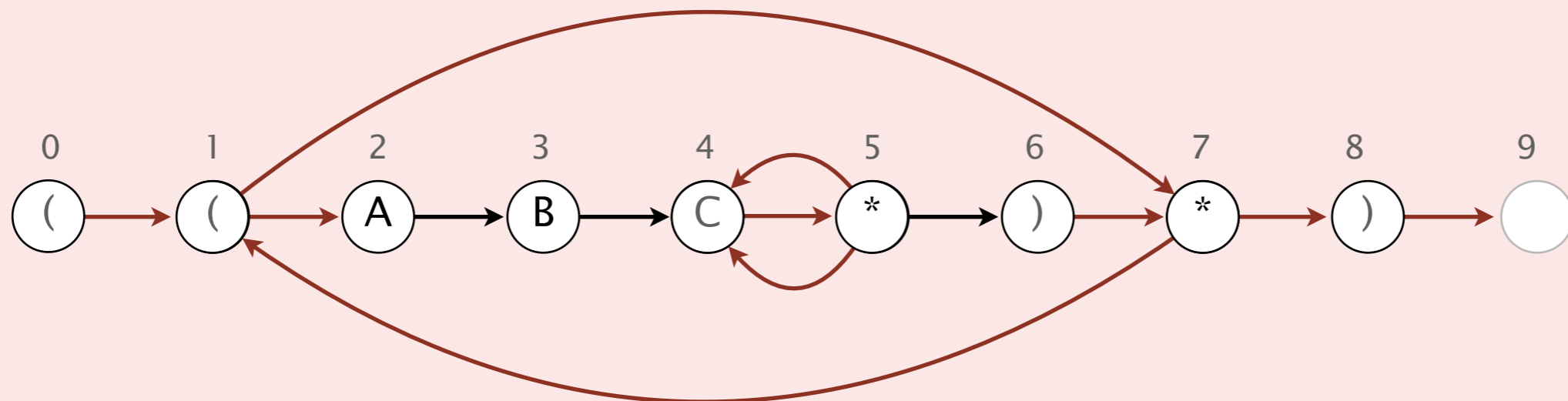

# Regular expression: quiz 4

---

How would you modify the NFA below to match  $((ABC^*)^+)$  ?

- A. Remove  $\epsilon$ -transition edge  $1 \rightarrow 7$ .
- B. Remove  $\epsilon$ -transition edge  $7 \rightarrow 1$ .
- C. Remove  $\epsilon$ -transition edges  $1 \rightarrow 7$  and  $7 \rightarrow 1$ .
- D. *I don't know.*

one or more occurrence



NFA corresponding to the pattern  $((ABC^*)^+)$



# NFA construction: implementation

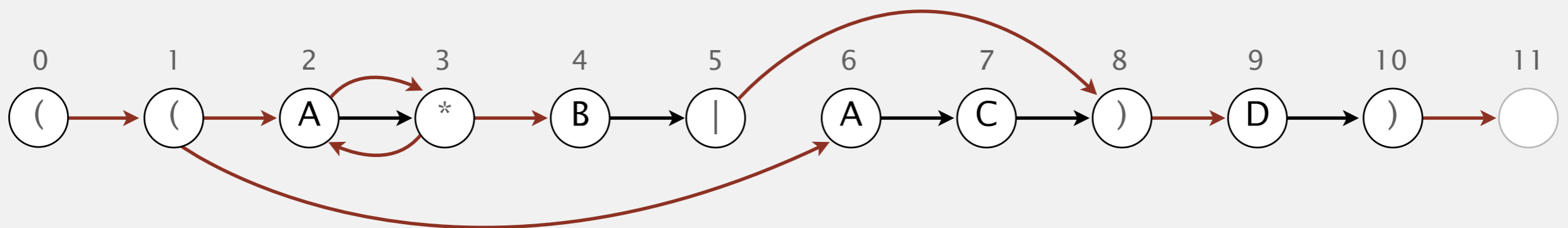
---

**Goal.** Write a program to build the  $\epsilon$ -transition digraph.

**Challenges.** Remember left parentheses to implement closure and or; remember | symbols to implement or.

**Solution.** Maintain a stack.

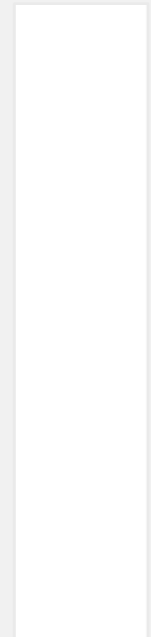
- ( symbol: push ( onto stack.
- | symbol: push | onto stack.
- ) symbol: pop corresponding ( and any intervening |; add  $\epsilon$ -transition edges for closure/or.



NFA corresponding to the pattern  $((A * B | A C) D)$

# NFA construction demo

---

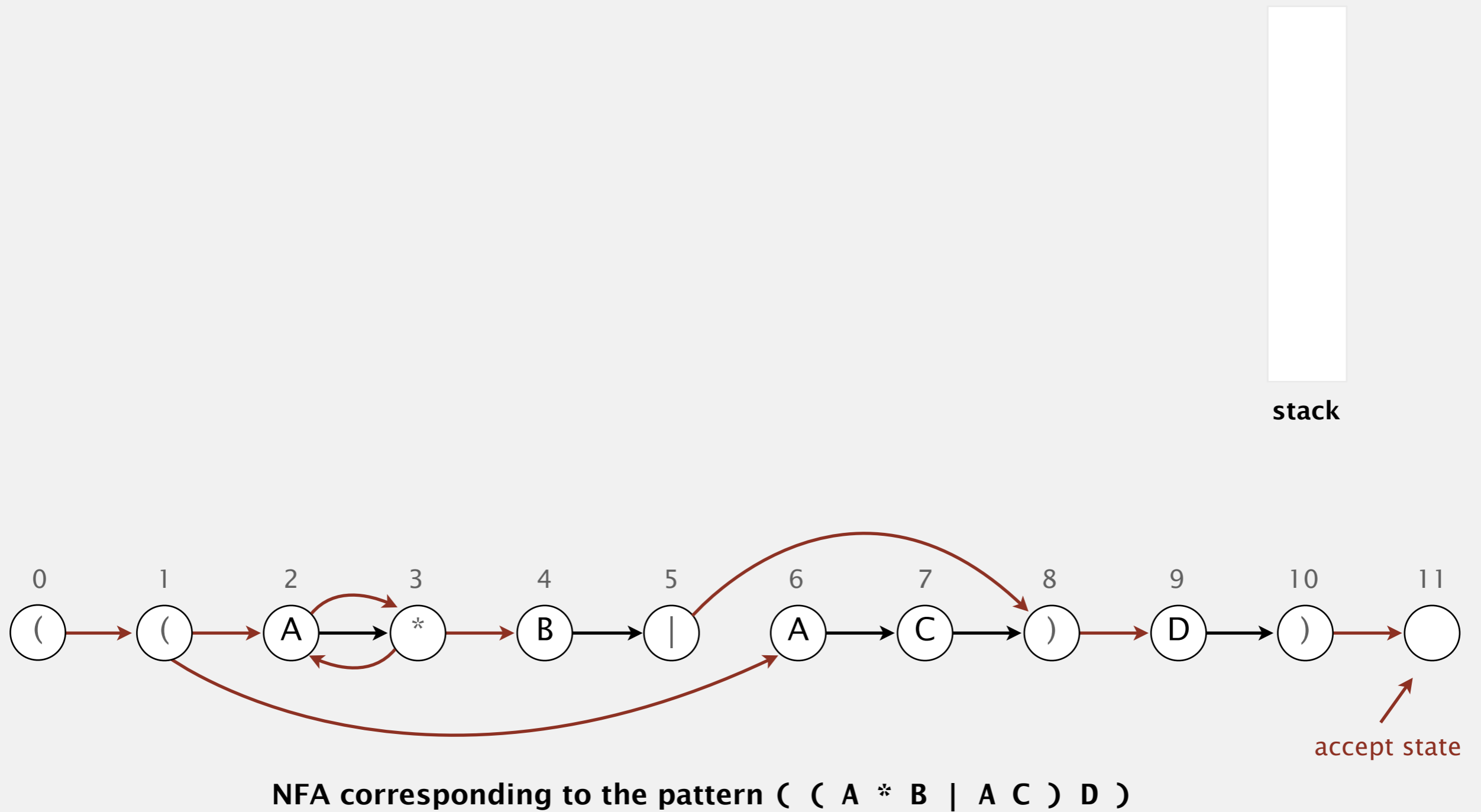


stack

( ( A \* B | A C ) D )

# NFA construction demo

---



# NFA construction: Java implementation

```
private Digraph buildEpsilonTransitionDigraph() {
    Digraph G = new Digraph(M+1);
    Stack<Integer> ops = new Stack<Integer>();
    for (int i = 0; i < M; i++) {
        int lp = i;

        if (re[i] == '(' || re[i] == '|') ops.push(i); ← left parentheses and |

        else if (re[i] == ')') {
            int or = ops.pop();
            if (re[or] == '|') { ← 2-way or
                lp = ops.pop();
                G.addEdge(lp, or+1);
                G.addEdge(or, i);
            }
            else lp = or;
        }

        if (i < M-1 && re[i+1] == '*') { ← closure
            G.addEdge(lp, i+1);
            G.addEdge(i+1, lp);
            (needs 1-character lookahead)
        }

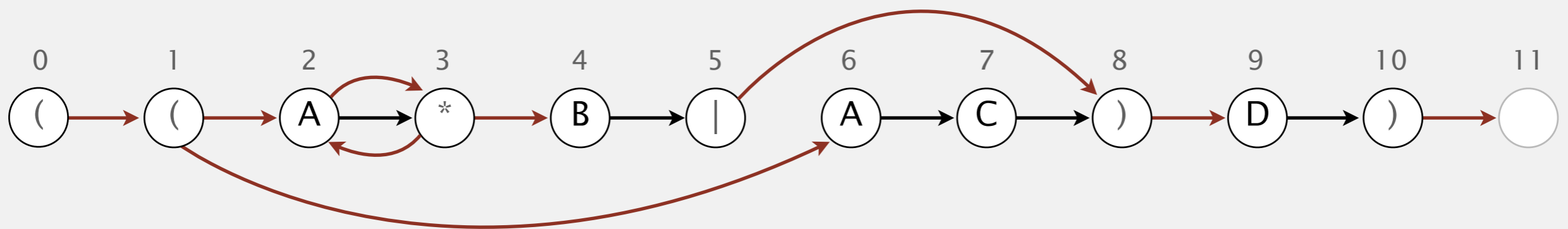
        if (re[i] == '(' || re[i] == '*' || re[i] == ')') ← metasymbols
            G.addEdge(i, i+1);
    }
    return G;
}
```

# NFA construction: analysis

---

**Proposition.** Building the NFA corresponding to an  $M$ -character RE takes time and space proportional to  $M$ .

**Pf.** For each of the  $M$  characters in the RE, we add at most three  $\epsilon$ -transitions and execute at most two stack operations.



NFA corresponding to the pattern  $( ( A * B | A C ) D )$



<http://algs4.cs.princeton.edu>

## 5.4 REGULAR EXPRESSIONS

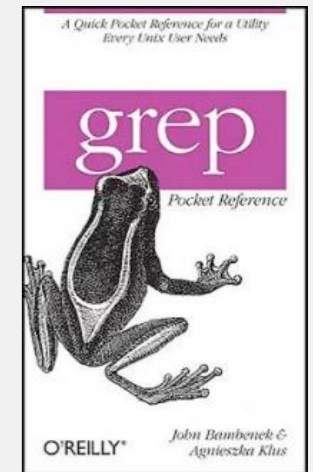
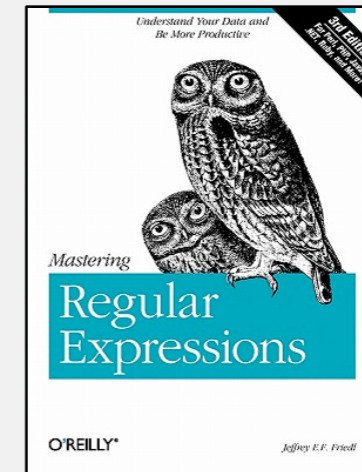
---

- ▶ *regular expressions*
- ▶ *REs and NFAs*
- ▶ *NFA simulation*
- ▶ *NFA construction*
- ▶ *applications*

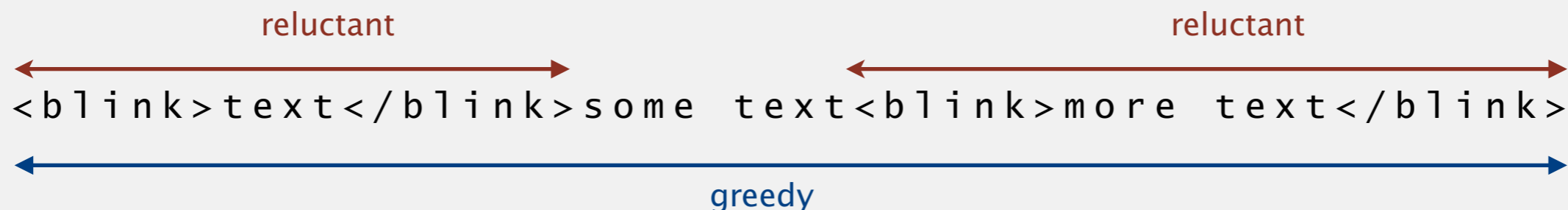
# Industrial-strength grep implementation

## To complete the implementation:

- Add multiway or.
- Handle metacharacters.
- Support character classes.
- Add capturing capabilities.
- Extend the closure operator.
- Error checking and recovery.
- Greedy vs. reluctant matching.



Ex. Which substring(s) should be matched by the RE `<blink>.*</blink>` ?



# Regular expressions in the wild

---

## Broadly applicable programmer's tool.



- Originated in Unix in the 1970s.
- Built in to many tools: grep, egrep, emacs, ....

```
% grep 'NEWLINE' */*.java
```

← print all lines containing NEWLINE which occurs in any file with a .java extension

```
% egrep '^[qwertyuiop]*[zxcvbnm]*$' words.txt | egrep '.....'  
typewritten
```

- Built in to many languages: awk, Perl, PHP, Python, JavaScript, ....

```
% perl -p -i -e 's|from|to|g' input.txt
```

← replace all occurrences of from with to in the file input.txt

```
% perl -n -e 'print if /^[A-Z][A-Za-z]*$/' words.txt
```

↑ do for each line

← print all words that start with uppercase letter



# Regular expressions in Java

---

**Validity checking.** Does the input match the re?

**Java string library.** Use `input.matches(re)` for basic RE matching.

```
public class Validate
{
    public static void main(String[] args)
    {
        String regexp = args[0];
        String input = args[1];
        StdOut.println(input.matches(re));
    }
}
```

```
% java Validate "[$_A-Za-z][$_A-Za-z0-9]*" ident123
true
```

← legal Java identifier

```
% java Validate "[a-z]+@([a-z]+\.)+(edu|com)" rs@cs.princeton.edu
true
```

← valid email address  
(simplified)

```
% java Validate "[0-9]{3}-[0-9]{2}-[0-9]{4}" 166-11-4433
true
```

← Social Security number

# Harvesting information

---

**Goal.** Print all substrings of input that match a RE.

```
% java Harvester "gcg(cgg|agg)*ctg" chromosomeX.txt
```

```
gcgcggcggcggcggcggcggctg
```

```
gcgctg
```

```
gcgctg
```

```
gcgcggcggcggaggcggaggcggctg
```



harvest patterns from DNA



```
% java Harvester "http://(\w+\.)*(\w+)" http://www.cs.princeton.edu
```

```
http://www.w3.org
```

```
http://www.cs.princeton.edu
```

```
http://drupal.org
```

```
http://csguide.cs.princeton.edu
```

```
http://www.cs.princeton.edu
```

```
http://www.princeton.edu
```

# Harvesting information

---

RE pattern matching is implemented in Java's `java.util.regex.Pattern` and `java.util.regex.Matcher` classes.

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class Harvester
{
    public static void main(String[] args)
    {
        String regex = args[0];
        In in = new In(args[1]);
        String input = in.readAll();
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(input);
        while (matcher.find())
        {
            StdOut.println(matcher.group());
        }
    }
}
```

`compile()` creates a Pattern (NFA) from RE

`matcher()` creates a Matcher (NFA simulator) from NFA and text

`find()` looks for the next match

`group()` returns the substring most recently found by `find()`

# Algorithmic complexity attacks

---

**Warning.** Typical implementations do **not** guarantee performance!



Unix grep, Java, Perl, Python

```
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 1.6 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 3.7 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 9.7 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 23.2 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 62.2 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 161.6 seconds
```

SpamAssassin regular expression.

```
% java RE "[a-z]+@[a-z]+([a-z\.]+\.)+[a-z]+" spammer@x.....
```

- Takes exponential time on pathological email addresses.
- Attacker can use such addresses to DOS a mail server.

# Not-so-regular expressions

---

## Back-references.

- `\1` notation matches subexpression that was matched earlier.
- Supported by typical RE implementations.

```
(.+)\1          // beriberi couscous  
1?$|^(11+?)\1+ // 1111 111111 1111111111
```

## Some non-regular languages.

- Strings of the form  $ww$  for some string  $w$ : beriberi.
- Unary strings with a composite number of 1s: 111111.
- Bitstrings with an equal number of 0s and 1s: 01110100.
- Watson-Crick complemented palindromes: atttcggaaat.

**Remark.** Pattern matching with back-references is intractable.

# Context

---

## Abstract machines, languages, and nondeterminism.

- Basis of the theory of computation.
- Intensively studied since the 1930s.
- Basis of programming languages.

**Compiler.** A program that translates a program to machine code.

- KMP string  $\Rightarrow$  DFA.
- grep RE  $\Rightarrow$  NFA.
- javac Java language  $\Rightarrow$  Java byte code.

	KMP	grep	Java
pattern	string	RE	program
parser	unnecessary	check if legal	check if legal
compiler output	DFA	NFA	byte code
simulator	DFA simulator	NFA simulator	JVM

# Summary of pattern-matching algorithms

---

## Programmer.

- Implement substring search via DFA simulation.
- Implement RE pattern matching via NFA simulation.



## Theoretician.

- RE is a compact description of a set of strings.
- NFA is an abstract machine equivalent in power to RE.
- DFAs, NFAs, and REs have limitations.



## You.

- Core CS principles provide useful tools that you can exploit now.
- REs and NFAs provide introduction to theoretical CS.

## Example of essential paradigm in computer science.

- Build the right intermediate abstractions.
- Solve important practical problems.