



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

## 5.4 REGULAR EXPRESSIONS

---

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



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

## 5.4 REGULAR EXPRESSIONS

---

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

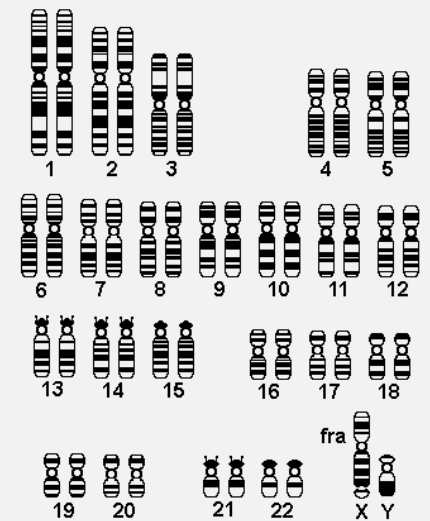
# Pattern matching

**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`

# 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 a regular expression here

Browse

- [by documentation entry](#)
- [by ProRule description](#)
- [by taxonomic scope](#)
- [by number of positive hits](#)

<http://prosite.expasy.org>

# Google code search

---

## Search public source code

Search Code

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

type a regular expression here

### Search Options

### In Search Box

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


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

# 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.
- Validate forms (dates, email, URL, credit card).
- Filter text (spam, NetNanny, Carnivore, malware).
- ...



Form Validation

First name:

Last name:

Username:

E-mail:

Password:

Phone:

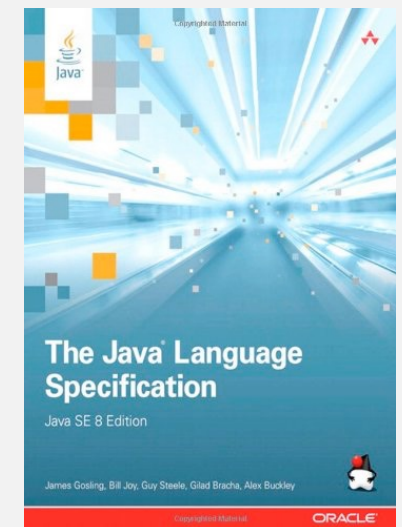
Date:

Address:

Powered by [QuiteWeb.com](http://QuiteWeb.com) ©

## Parse text files.

- Compile a Java program.
- Crawl and index the Web.
- Read 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.

↑  
typically infinite

operation	order	example RE	matches	does not match
concatenation	3	A A B A A B	A A B A A B	<i>every other string</i>
or	4	A A   B A A B	A A B A A B	<i>every other string</i>
closure	2	A B * A	A A A B B B B B B B A	A B A B A B A
parentheses	1	A ( A   B ) A A B	A A A A B A B A A B	<i>every other string</i>
		( A B ) * A	A A B A B A B A B A B A	A A A B B A



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

- 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 C A B D



# Regular expression shortcuts

---

Additional operations further extend the utility of REs.

operation	example RE	matches	does not match
wildcard	.U.U.U.	CUMULUS JUGULUM	SUCCUBUS TUMULTUOUS
character class	[A-Za-z][a-z]*	word Capitalized	camelCase 4illegal
one or more	A(BC)+DE	ABCDE ABCBCDE	ADE BCDE
exactly k	[0-9]{5}-[0-9]{4}	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.



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

- (1) alphabet is { A, C, G, T }
- (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|G|T)(A|C|G|T)(A|C|G|T))* (TAG|TAA|TTG)`
- B.** `ATG([ACGT]{3})* (TAG|TAA|TTG)`
- C.** Both A and B.
- D.** Neither A nor B.

# Regular expressions to the rescue



# Can the average programmer learn to use REs?

## Perl RE for valid RFC822 email addresses

[illegible]

# 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; using them can be amazingly complex and error-prone.



<https://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.**

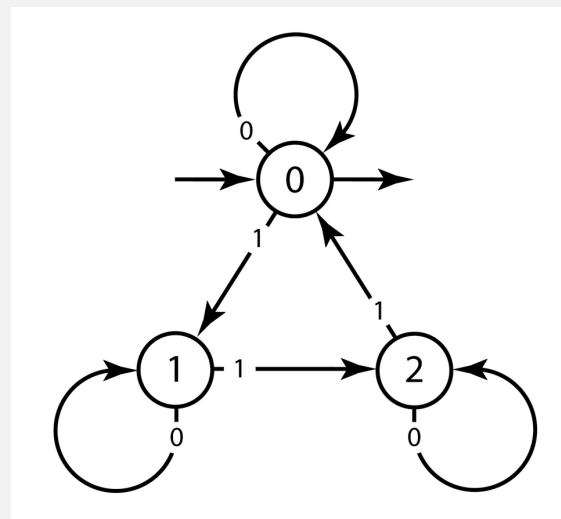
- For any DFA, there exists a RE that describes the same set of strings.
- For any RE, there exists a DFA that recognizes the same set of strings.

**RE**

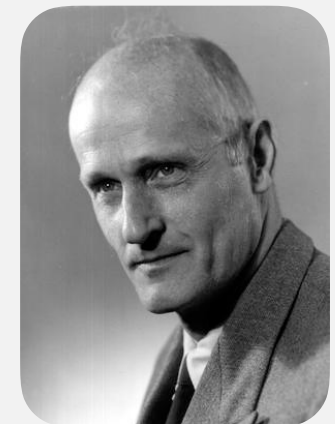
$0^* \mid (0^*10^*10^*10^*)^*$

number of 1s is a multiple of 3

**DFA**



number of 1s 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.

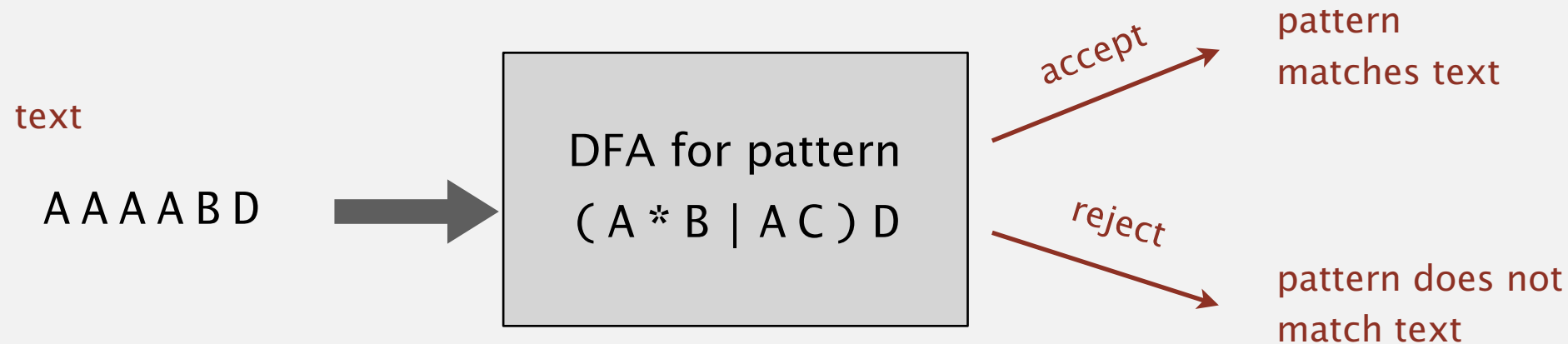


Ken Thompson  
Turing Award '83

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).

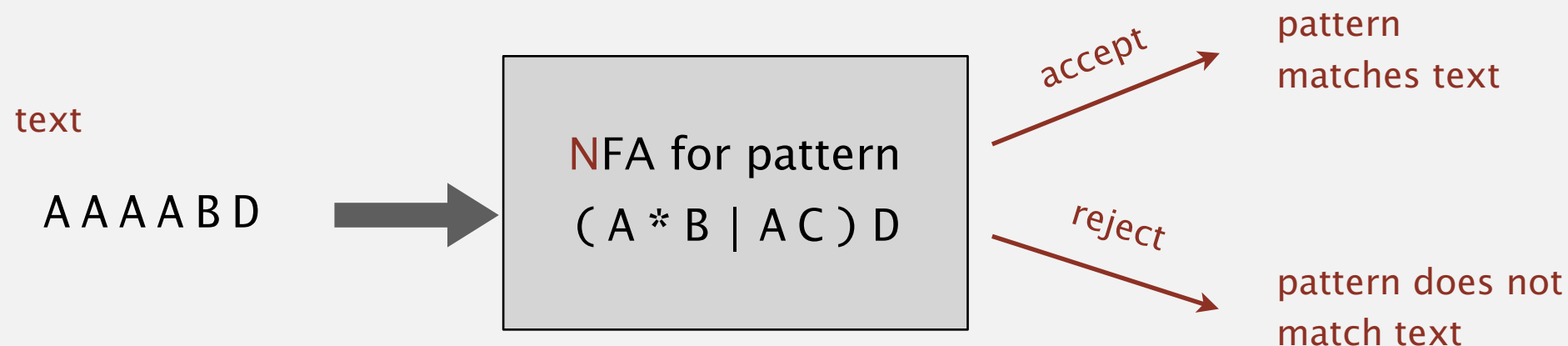


Ken Thompson  
Turing Award '83

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

Basic plan. [apply Kleene's theorem]

- Build **N**FA from RE.
- Simulate **N**FA with text as input.



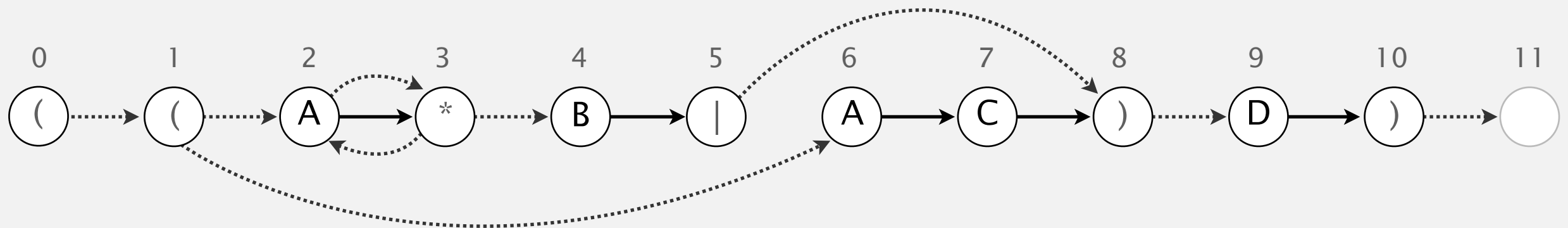
Q. What is an NFA?

# Nondeterministic finite-state automata

## Regular-expression-matching NFA.

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

after scanning all text characters

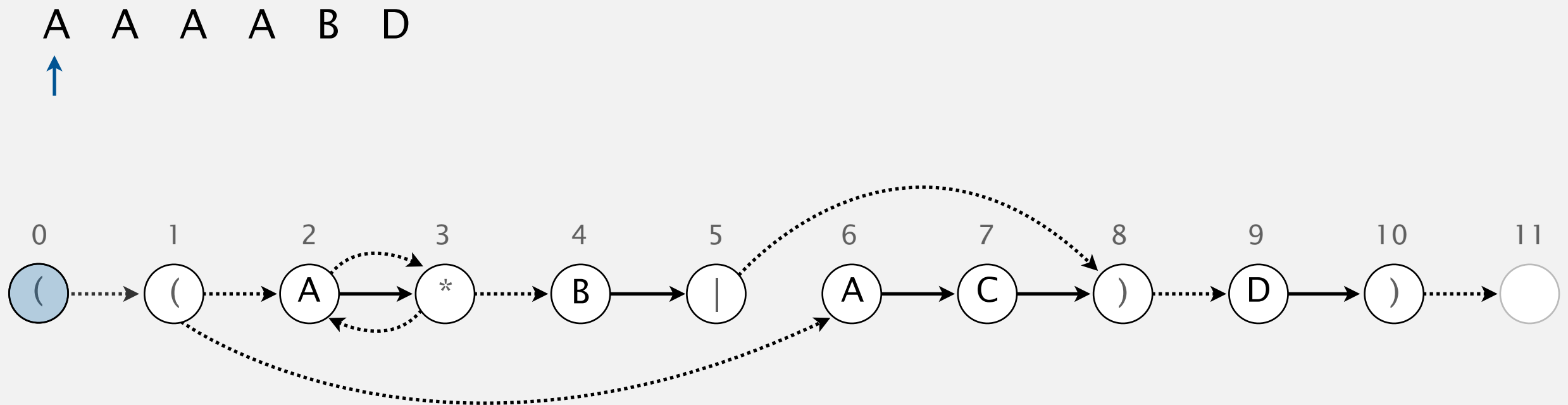
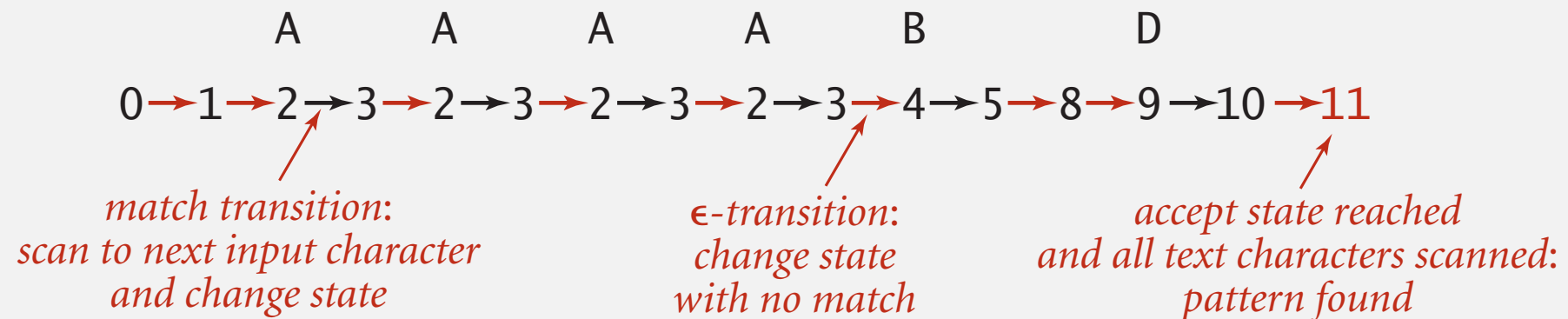


NFA corresponding to the pattern  $((A * B | A C) D)$

# Nondeterministic finite-state automata

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

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

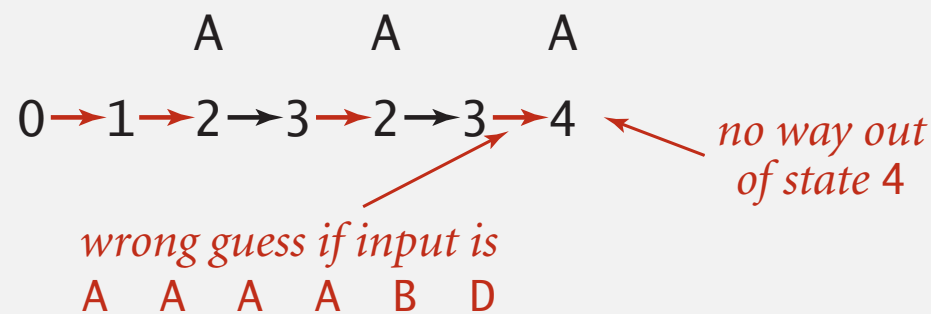


NFA corresponding to the pattern  $((A^*B|AC)D)$

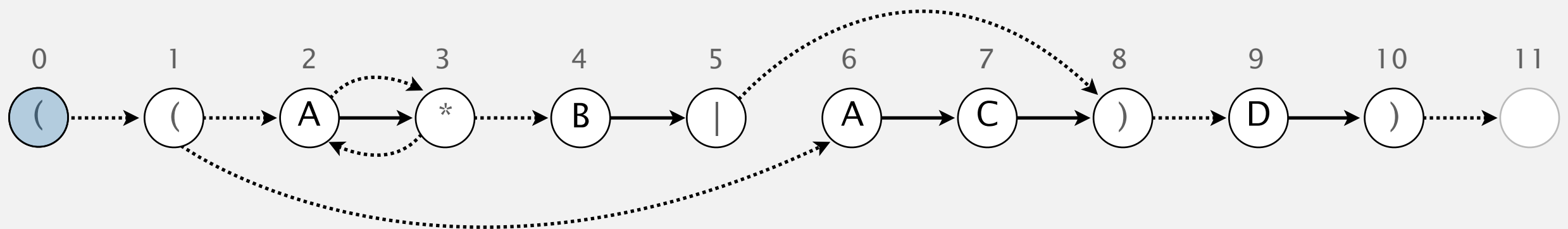
# Nondeterministic finite-state automata

Q. Is A 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 ]



A A A A B D  
↑



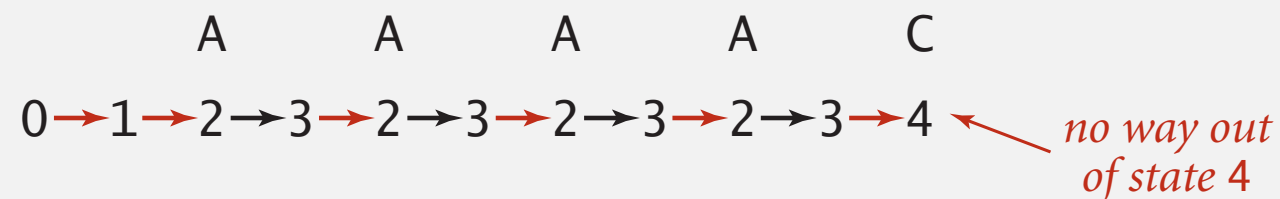
NFA corresponding to the pattern ( ( A \* B | A C ) D )

# Nondeterministic finite-state automata

Q. Is A A A C matched by NFA?

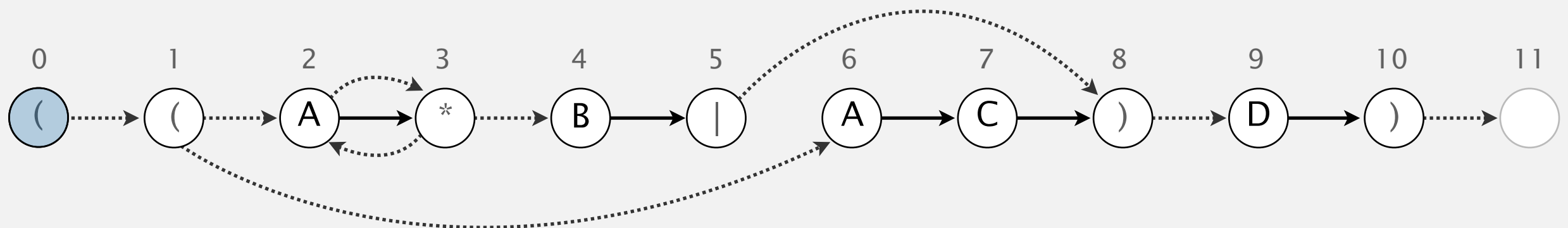
A. No, because **no** sequence of legal transitions ends in state 11.

[ must argue about all possible sequences (not just the one below) ]



A   A   A   A   C

↑

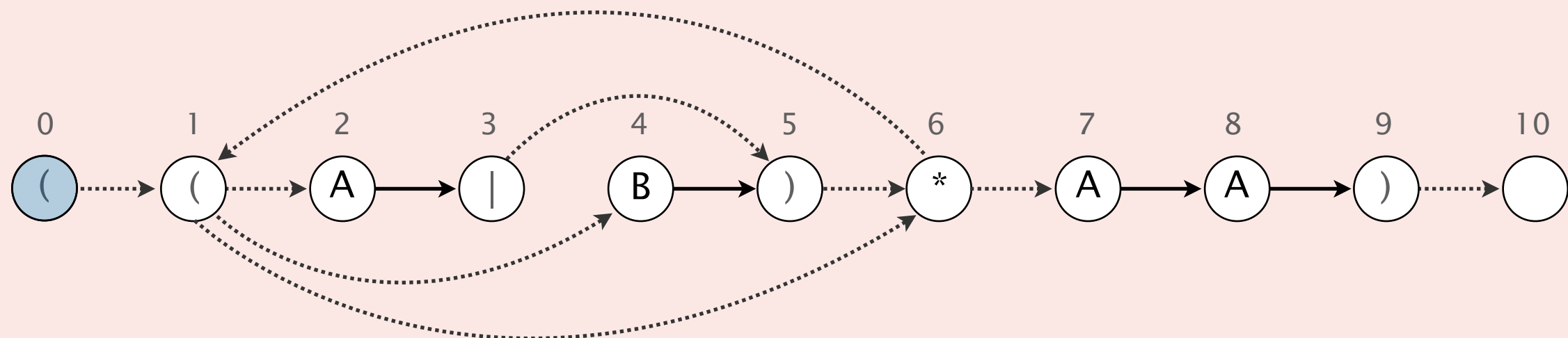


NFA corresponding to the pattern ( ( A \* B | A C ) D )



Which of the following strings are matched by the NFA?

- A. B A A A A
- B. A A B A A B A A
- C. Both A and B.
- D. Neither A nor B.



# Nondeterminism

---

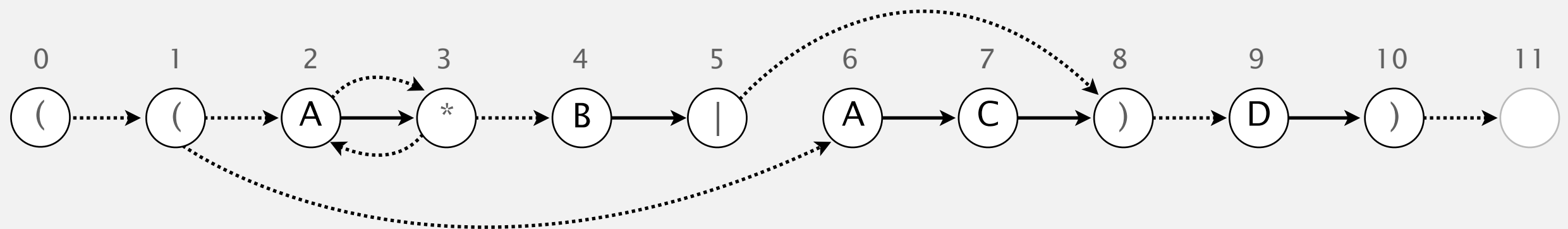
**Q.** How to determine whether a string is matched by an automaton?

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

**NFA.** Nondeterministic  $\Rightarrow$  hard (at each step, can be several applicable transitions; machine “guesses” the correct one!)

**Q.** How to simulate NFA?

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



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





<https://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  $m$ .

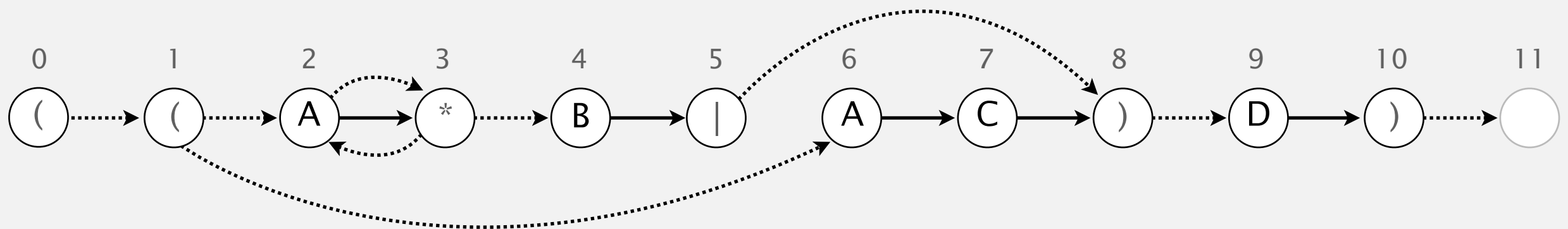
number of symbols in RE

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**  $G$ .

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



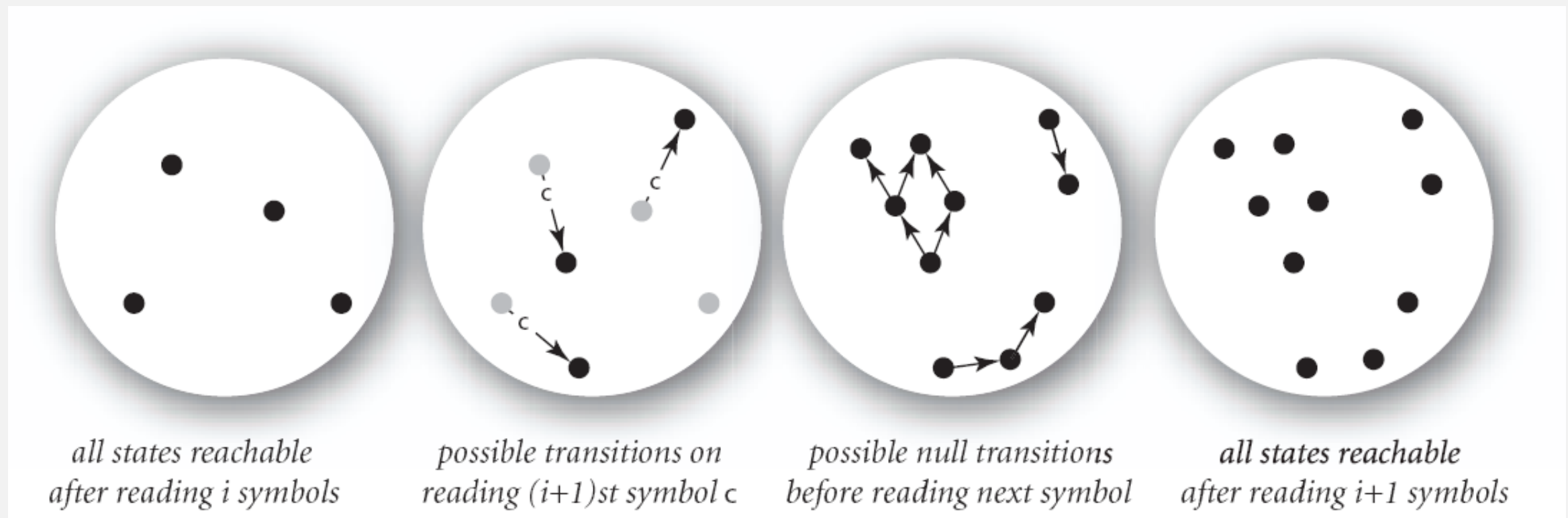
NFA corresponding to the pattern `(( A * B | A C ) D )`

# 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.

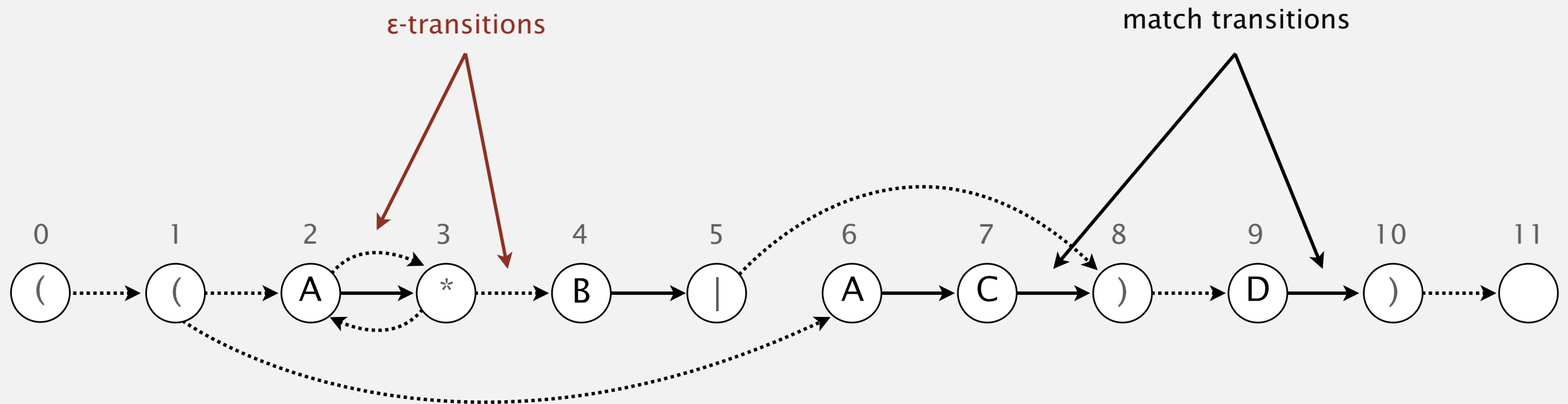
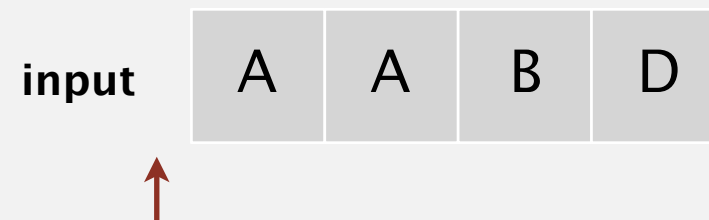
one step in simulating an NFA



- 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 **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(); ← stay tuned
    }

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

    public Digraph buildEpsilonTransitionDigraph()
    { /* 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  $m$

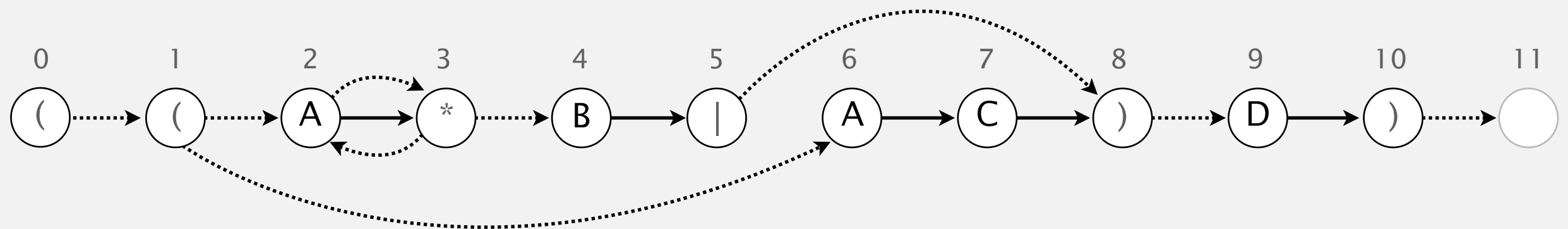
```
}
```

# 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  $m n$  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  $\varepsilon$ -transitions.

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



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





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

## 5.4 REGULAR EXPRESSIONS

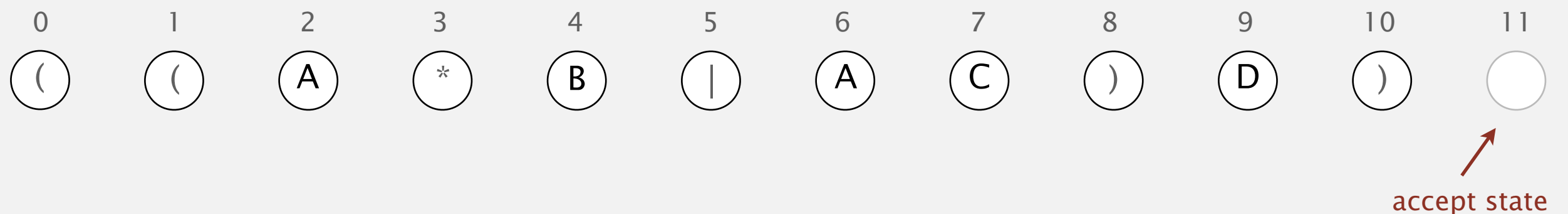
---

- ▶ *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  $((A * B | A C ) D )$

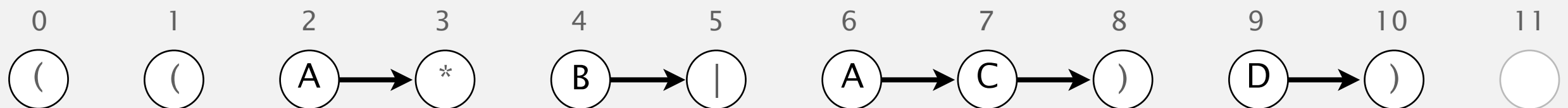
# Building an NFA corresponding to an RE

---

**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  $\varepsilon$ -transition edge from parentheses to next state.

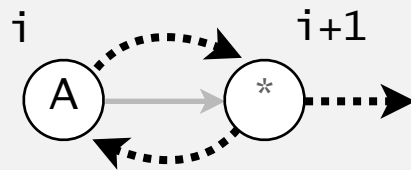


NFA corresponding to the pattern  $((A*B|AC)D)$

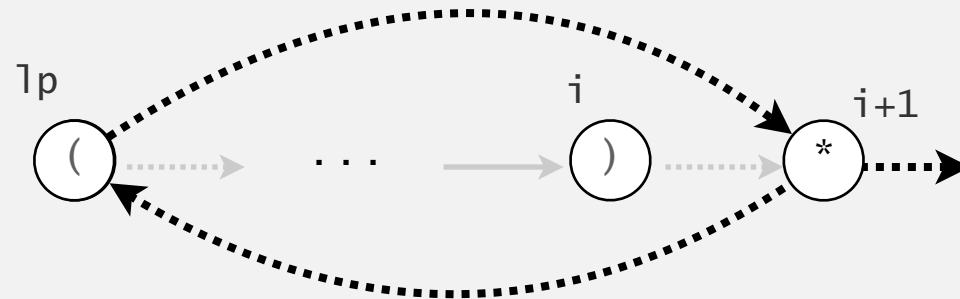
# Building an NFA corresponding to an RE

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

**singe-character closure**



**closure expression**

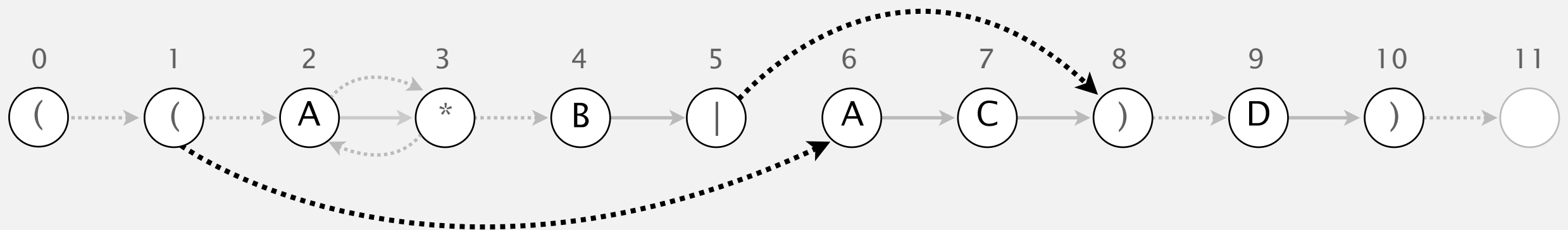
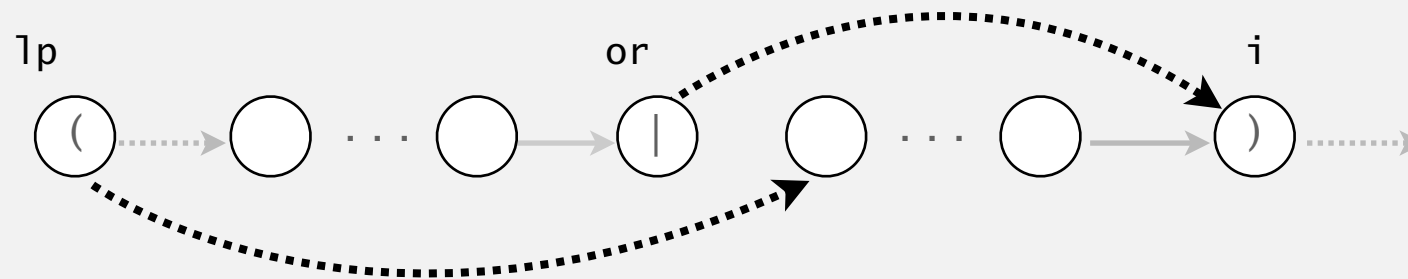


NFA corresponding to the pattern  $( ( A * B | A C ) D )$

# Building an NFA corresponding to an RE

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

2-way or expression



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

# Building an NFA corresponding to an RE

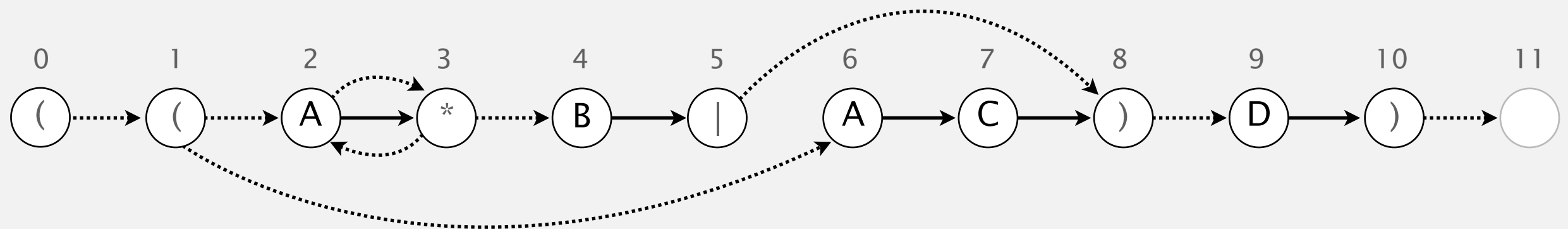
**States.** Include a state for each symbol in the RE, plus an accept state.

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

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

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

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



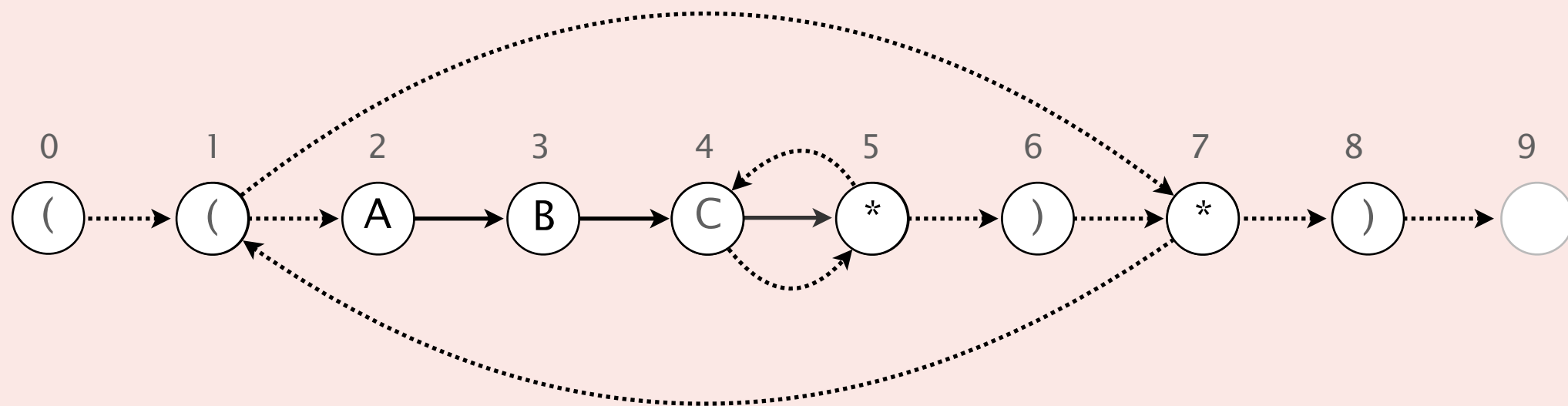
NFA corresponding to the pattern  $((A * B | A C) D)$



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. None of the above.

one or more occurrence



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



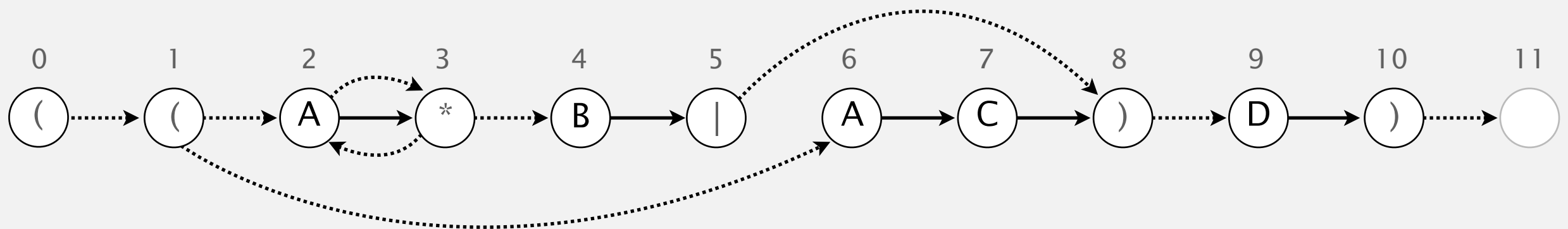
# NFA construction: implementation

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

**Challenges.** Remember left parentheses to implement closure and 2-way or; remember | symbols to implement 2-way or.

**Solution.** Maintain a stack.

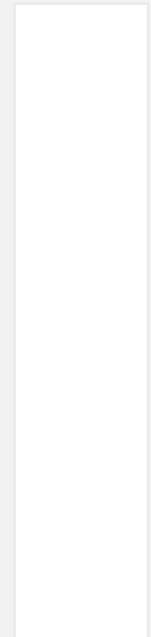
- ( symbol: push ( onto stack.
- | symbol: push | onto stack.
- ) symbol: pop corresponding ( and any intervening |; add  $\varepsilon$ -transition edges for closure/or.



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

# NFA construction demo

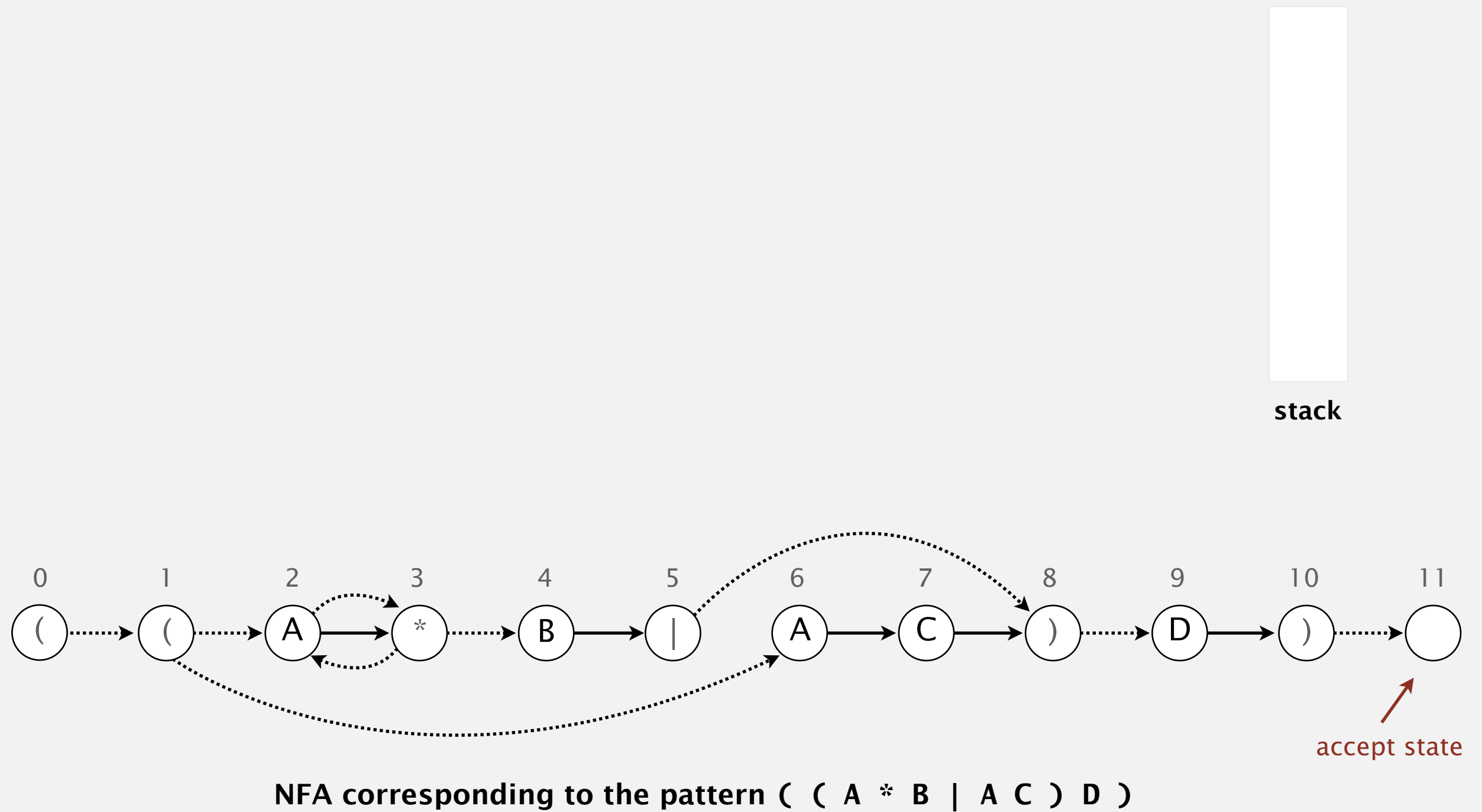
---



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);

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

        if (i < m-1 && re[i+1] == '*') {
            G.addEdge(lp, i+1);
            G.addEdge(i+1, lp);
        }

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

← left parentheses and |

← 2-way or

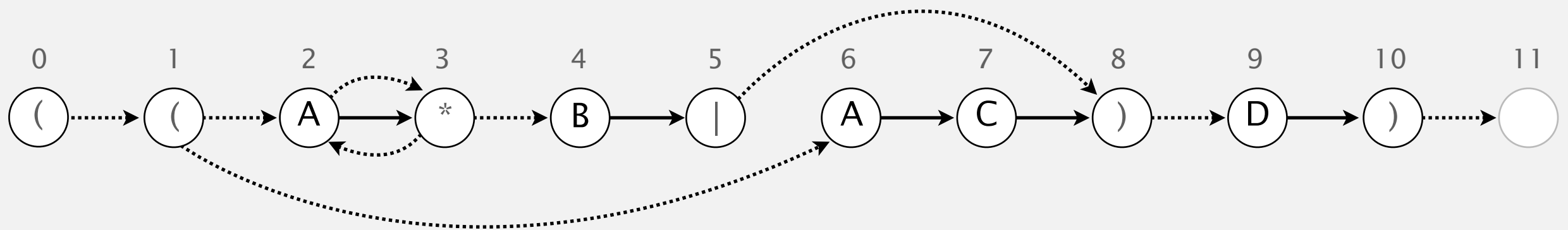
← closure  
(needs 1-character lookahead)

← metasympols

# 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  $((A * B | A C) D)$



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

## 5.4 REGULAR EXPRESSIONS

---

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



# Regular expressions in Java

---

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

**Java string library.** Use `input.matches(regex)` 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(regexp));
    }
}
```

```
% 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
```

```
GCGCGGCGGCGGCGGCGGCTG
```

```
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 regexp = args[0];
```

```
        In in = new In(args[1]);
```

```
        String input = in.readAll();
```

```
        Pattern pattern = Pattern.compile(regexp);
```

```
        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()`



What is the worst-case running time of Java's `matches()` method?

A.  $m + n$

B.  $m n$

C.  $m n^2$

D.  $2^n$

$m$  = pattern length

$n$  = text length

## matches

```
public boolean matches(String regex)
```

Tells whether or not this string matches the given `regular expression`.

An invocation of this method of the form `str.matches(regex)` yields exactly the same result as the expression

```
Pattern.matches(regex, str)
```

### Parameters:

`regex` - the regular expression to which this string is to be matched

### Returns:

true if, and only if, this string matches the given regular expression

# 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

---

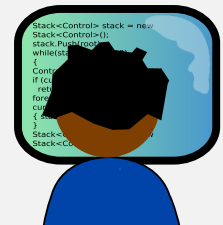
## Theoretician.

- 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.



## Programmer.

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



## You.

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



## Example of essential paradigm in computer science.

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