5.4 Regular Expressions

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

Algorithms, 4th Edition · Robert Sedgewick and Kevin Wayne · Copyright © 2002–2010 · April 11, 2011 7:57:28 AM

- **regular expressions**
- 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.
- Human genome contains triplet repeats of CGG or AGG, bracketed by GCG at the beginning and CTG at the end.
- Number of repeats is variable, and correlated with syndrome.

pattern	`GCG(CGG|AGG)*CTG`
text	`GCGGCGTGTGTGCGAGAGAGTGGGTTTAAAGCTGGCGCGGAGGCGGCTGGCGCGGAGGCTG`

Pattern matching: applications

Test if a string matches some pattern.

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

Parse text files.

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

Regular expressions

A regular expression is a notation to specify a (possibly infinite) set of strings.

↑ a "language"

operation	example RE	matches	does not match
concatenation	AABAAB	AABAAB	every other string
or	AA | BAAB	AA BAAB	every other string
closure	AB*A	AA ABBBBBBBBA	AB ABABA
parentheses	A(A|B)AAB	AAAAB ABAAB	every other string
	(AB)*A	A ABABABABABA	AA ABBA

Regular expression shortcuts

Additional operations are often added for convenience.

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

operation	example RE	matches	does not match
wildcard	.U.U.U.	CUMULUS JUGULUM	SUCCUBUS TUMULTUOUS
at least 1	A(BC)+DE	ABCDE ABCBCDE	ADE BCDE
character classes	[A-Za-z][a-z]*	word Capitalized	camelCase 4illegal
exactly k	[0-9]{5}-[0-9]{4}	08540-1321 19072-5541	111111111 166-54-111
complement	[^AEIOU]{6}	RHYTHM	DECADE

Regular expression examples

Notation is surprisingly expressive

regular expression	matches	does not match
.*SPB.* *(contains the trigraph spb)*	RASPBERRY CRISPBREAD	SUBSPACE SUBSPECIES
[0-9]{3}-[0-9]{2}-[0-9]{4} *(Social Security numbers)*	166-11-4433 166-45-1111	11-55555555 8675309
[a-z]+@([a-z]+\.)+(edu|com) *(valid email addresses)*	wayne@princeton.edu rs@princeton.edu	spam@nowhere
[$_A-Za-z][$_A-Za-z0-9]* *(valid Java identifiers)*	ident3 PatternMatcher	3a ident#3

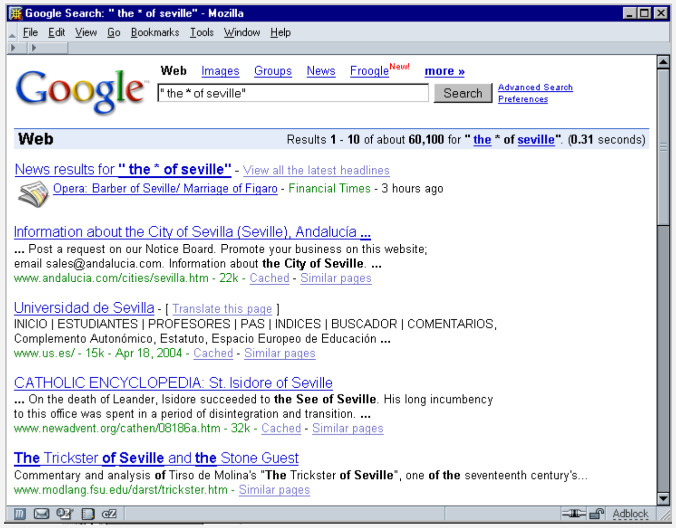
and plays a well-understood role in the theory of computation.

Regular expressions to the rescue

http://xkcd.com/208

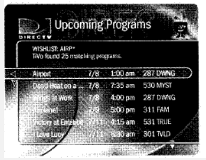
Can the average web surfer learn to use REs?

Google. Supports * for full word wildcard and | for union.



Can the average TV viewer learn to use REs?

TiVo. WishList has very limited pattern matching.



Using * in WishList Searches. To search for similar words in Keyword and Title WishList searches, use the asterisk (*) as a special symbol that replaces the endings of words. For example, the keyword *AIRP** would find shows containing "airport," "airplane," "airplanes," as well as the movie "Airplane!" To enter an asterisk, press the SLOW ( ) button as you are spelling out your keyword or title.

The asterisk can be helpful when you're looking for a range of similar words, as in the example above, or if you're just not sure how something is spelled. Pop quiz: is it "irresistible" or "irresistable?" Use the keyword *IRRESIST** and don't worry about it! Two things to note about using the asterisk:

- It can only be used at a word's end; it cannot be used to omit letters at the beginning or in the middle of a word. (For example, *AIR*NE* or **PLANE* would not work.)

Reference: page 76, Hughes DirectTV TiVo manual

Can the average programmer learn to use REs?

Perl RE for valid RFC822 email addresses

http http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html

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.

- regular expressions
- **NFAs**
- NFA simulation
- NFA construction
- applications

Pattern matching implementation: basic plan (first attempt)

Ken Thompson
Turing Award '83

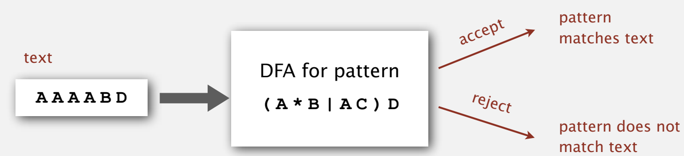
Overview is the same as for KMP.

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

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 number of states).

Pattern matching implementation: basic plan (revised)

Ken Thompson
Turing Award '83

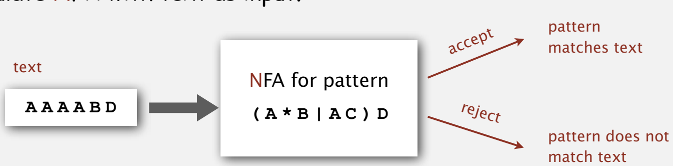
Overview is similar to KMP.

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

Underlying abstraction. Nondeterministic finite state automata (NFA).

Basic plan. [apply Kleene's theorem]

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



Q. What exactly is an NFA?

Duality

RE. Concise way to describe a set of strings.

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

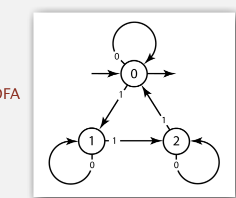
Kleene's theorem.

- 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* | (0*10*10*10*)*`

number of 1's is a multiple of 3

DFA



number of 1's is a multiple of 3

Good news. Basic plan works in theory.

Bad news. Basic plan fails in practice.

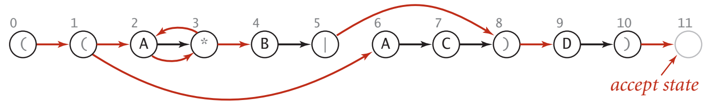
Nondeterministic finite-state automata

Regular-expression-matching NFA.

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

Nondeterminism.

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

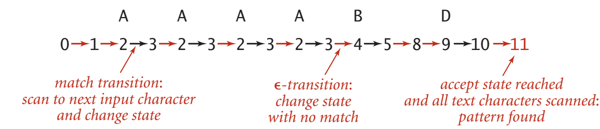


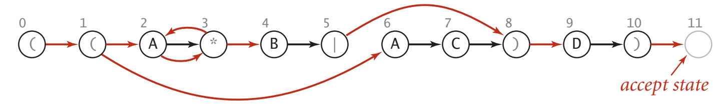
NFA corresponding to the pattern ((A * B | A C) D)

Nondeterministic finite-state automata

Q. Is AAAABD matched by NFA?

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



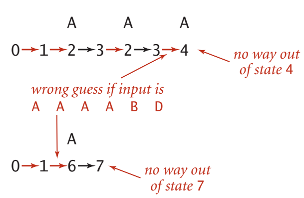


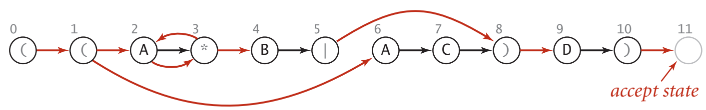
NFA corresponding to the pattern ((A * B | A C) D)

Nondeterministic finite-state automata

Q. Is AAAABD matched by NFA?

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



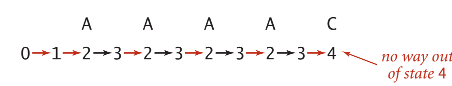


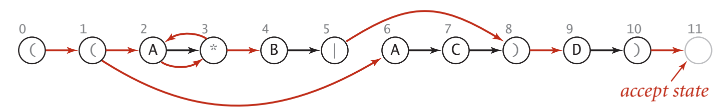
NFA corresponding to the pattern ((A * B | A C) D)

Nondeterministic finite-state automata

Q. Is AAAAC 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)

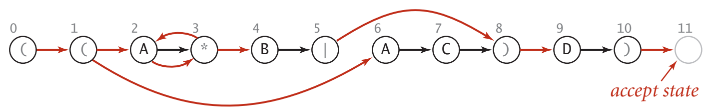
Nondeterminism

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

DFA. Deterministic ⇒ exactly one applicable transition.

NFA. Nondeterministic ⇒ can be several applicable transitions; need to select the right one!

Q. How to simulate NFA?
A. Systematically consider all possible transition sequences.



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

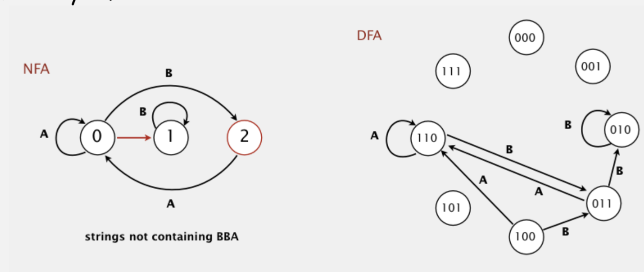
Partial proof of Kleene's theorem (RE ⇒ DFA)

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

- Given an RE, construct an NFA (stay tuned)
- Given an NFA, construct a DFA (see construction below)

To construct a DFA that recognizes the same language as a given NFA:

- create a DFA state for every set of NFA states
- systematically infer transitions



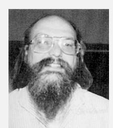
Problem: N states in NFA ⇒ 2^N states in DFA

Insight: Need to consider all possible transitions to simulate NFA

Pattern matching implementation: basic plan (revised)

Overview is similar to KMP.

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

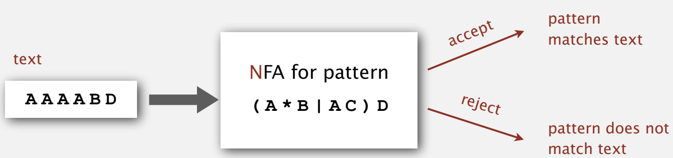


Ken Thompson
Turing Award '83

Underlying abstraction. Nondeterministic finite state automata (NFA).

Basic plan. [apply Kleene's theorem]

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



Q. How to construct NFA and how to efficiently simulate NFA?

- regular expressions
- NFAs
- **NFA simulation**
- NFA construction
- applications

NFA representation

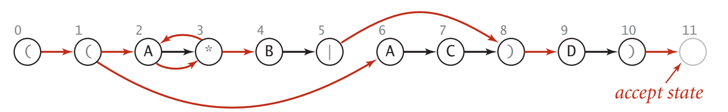
State names. Integers from 0 to M.

number of symbols in RE

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

ε-transitions. Store in a digraph G.

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

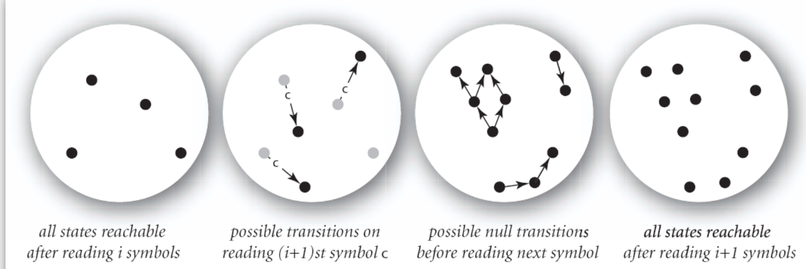


NFA corresponding to the pattern ((A * B | A C) D)

NFA simulation

Q. How to efficiently simulate an NFA?

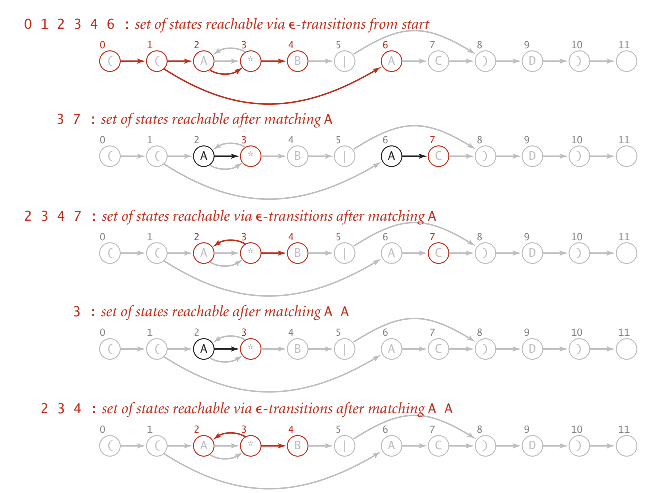
A. Maintain set of all possible states that NFA could be in after reading in the first i text characters.



One step in simulating an NFA

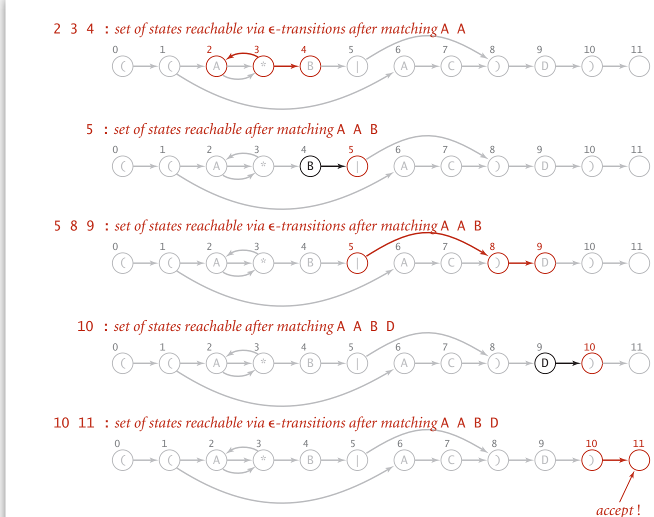
Q. How to perform reachability?

NFA simulation example



Simulation of ((A * B | A C) D) NFA for input A A B D

NFA simulation example (continued)



Simulation of ((A * B | A C) D) NFA for input A A B D

Digraph reachability

Recall Section 4.2. Find all vertices reachable from a given set of vertices.

```
public class DirectedDFS

        DirectedDFS(Digraph G, int s)                   find vertices reachable from s

        DirectedDFS(Digraph G,
                    Iterable<Integer> sources)          find vertices reachable from sources

boolean marked(int v)                                   is v reachable from source(s)?
```


NFA simulation: Java implementation

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

   public NFA(String regexp)
   {
      M  = regexp.length();
      re = regexp.toArray();
      G = buildEpsilonTransitionsGraph();       ← stay tuned
   }

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

}
```

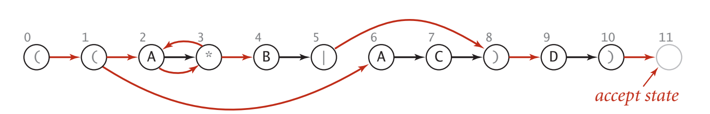

NFA simulation: Java implementation

```
public boolean recognizes(String txt)
{
   Bag<Integer> pc = new Bag<Integer>();            ← states reachable from
   DirectedDFS dfs = new DirectedDFS(G, 0);           start by ε-transitions
   for (int v = 0; v < G.V(); v++)
      if (dfs.marked(v)) pc.add(v);

   for (int i = 0; i < txt.length(); i++)
   {
      Bag<Integer> match = new Bag<Integer>();     ← states reachable after
      for (int v : pc)                               scanning past txt.charAt(i)
      {
         if (v == M) continue;
         if ((re[v] == txt.charAt(i)) || re[v] == '.')
            match.add(v+1);
      }

      dfs = new DirectedDFS(G, match);             ← follow ε-transitions
      pc = new Bag<Integer>();
      for (int v = 0; v < G.V(); v++)
         if (dfs.marked(v)) pc.add(v);
   }

   for (int v : pc)
      if (v == M) return true;                     ← accept iff ends in state M
   return false;
}
```


NFA simulation: analysis

Proposition. Determining whether an N-character text string 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 ε-transitions. (The NFA construction we consider ensures the number of edges in $G \le 3M$.)

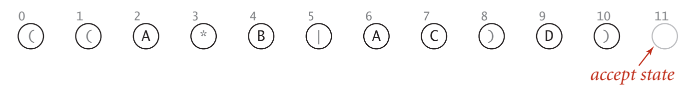


NFA corresponding to the pattern ((A * B | A C) D)

- regular expressions
- 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.

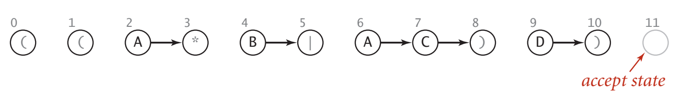


NFA corresponding to the pattern ((A * B | A C) D)

Building an NFA corresponding to an RE

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

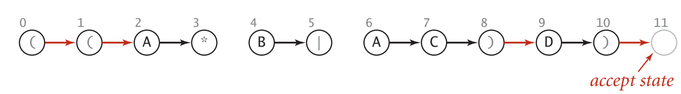
Alphabet. A B C D
Metacharacters. ( ) . * |



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

Building an NFA corresponding to an RE

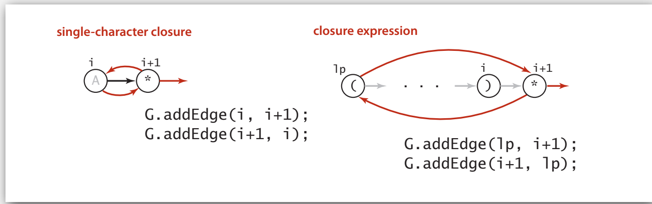
Parentheses. Add ε-transition edge from parentheses to next state.

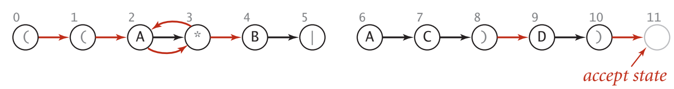


NFA corresponding to the pattern ((A * B | A C) D)

Building an NFA corresponding to an RE

Closure. Add three ε-transition edges for each `*` operator.

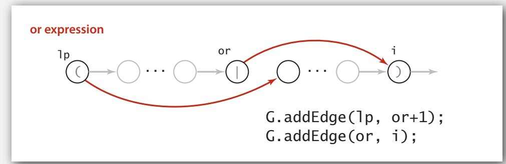


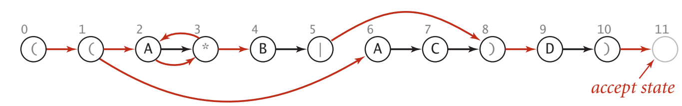


NFA corresponding to the pattern `( ( A * B | A C ) D )`

Building an NFA corresponding to an RE

Or. Add two ε-transition edges for each `|` operator.





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

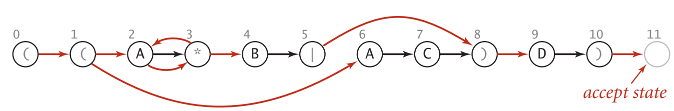
NFA construction: implementation

Goal. Write a program to build the ε-transition digraph.

Challenges. Need to remember left parentheses to implement closure and or; also need to remember `|` to implement or.

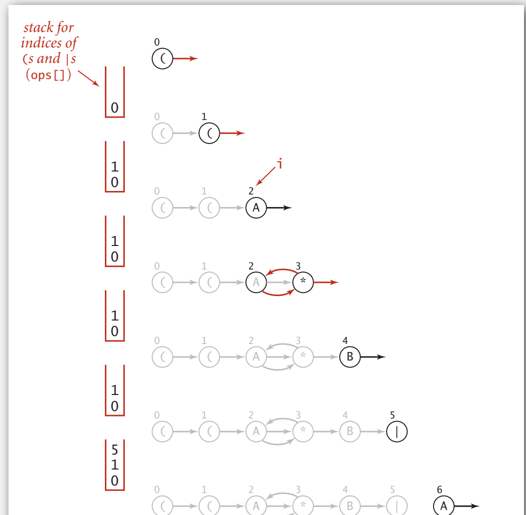
Solution. Maintain a stack.

- `(` symbol: push `(` onto stack.
- `|` symbol: push `|` onto stack.
- `)` symbol: pop corresponding `(` and possibly intervening `|`; add ε-transition edges for closure/or.



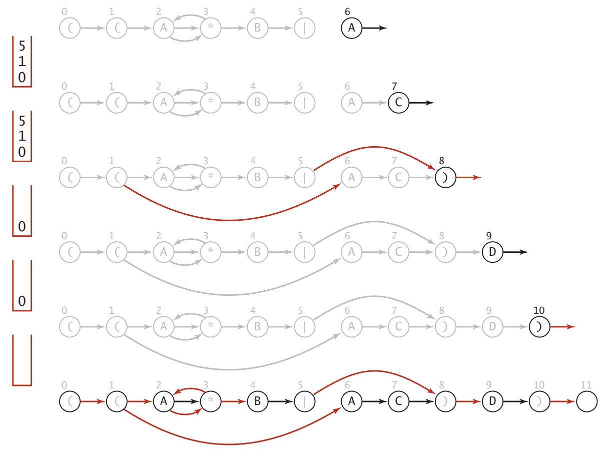
NFA corresponding to the pattern `( ( A * B | A C ) D )`

NFA construction: example



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

NFA construction: example



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

NFA construction: Java implementation

```
private Digraph buildEpsilonTransitionGraph() {
   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 |

or

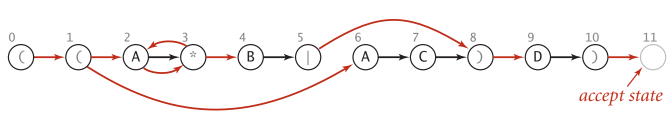
closure
(needs lookahead)

metasymbols

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 ε-transitions and execute at most two stack operations.



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

- regular expressions
- NFAs
- NFA simulation
- NFA construction
- **applications**

Generalized regular expression print

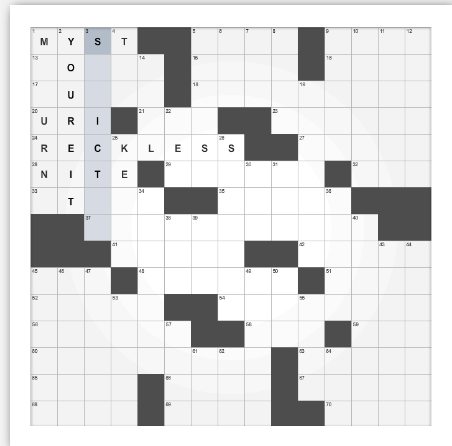
Grep. Take a RE as a command-line argument and print the lines from standard input having some substring that is matched by the RE.

```
public class GREP
{
   public static void main(String[] args)
   {
      String regexp = "(.*" + args[0] + ".*)";
      NFA nfa = new NFA(regexp);
      while (StdIn.hasNextLine())
      {
         String line = StdIn.readLine();
         if (nfa.recognizes(line))
            StdOut.println(line);
      }
   }
}
```

find lines containing RE as a substring

Bottom line. Worst-case for grep (proportional to $M N$) is the same as for elementary exact substring match.

Typical grep application: crossword puzzles



```
% more words.txt
a
aback
abacus
abalone
abandon
...
```

dictionary (standard in Unix) also on booksite

```
% grep 's..ict..' words.txt
constrictor
stricter
stricture
```


Industrial-strength grep implementation

To complete the implementation:

- Add character classes.
- Handle metacharacters.
- 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>` ?

reluctant reluctant

`<blink>text</blink>some text<blink>more text</blink>`

greedy

Regular expressions in other languages

Broadly applicable programmer's tool.

- Originated in Unix in the 1970s.
- Many languages support extended regular expressions.
- Built into grep, awk, emacs, Perl, PHP, Python, JavaScript.

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

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

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

PERL. Practical Extraction and Report Language.

```
% 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
```

print all words that start with uppercase letter

do for each line

Regular expressions in Java

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

Java string library. Use `input.matches(regexp)` 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

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

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

legal Java identifier

valid email address (simplified)

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

harvest links from website

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


Harvesting information

RE pattern matching is implemented in Java's `Pattern` and `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()`

Algorithmic complexity attacks

Warning. Typical implementations do not guarantee performance!

Unix grep, Java, Perl

```
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac              1.6 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac            3.7 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac          9.7 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac       23.2 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac     62.2 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 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.
- Troublemaker can use such addresses to DOS a mail server.

Not-so-regular expressions

Back-references.

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

```
% java Harvester "\b(.+)\1\b"  words.txt
beriberi
couscous
```

word boundary

Some non-regular languages.

- Set of strings of the form ww for some string w: **beriberi**.
- Set of bitstrings with an equal number of 0s and 1s: `01110100`.
- Set of 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 ⇒ DFA.
- `grep` RE ⇒ NFA.
- `javac` Java language ⇒ 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 and REs have limitations.

You. Practical application of core CS principles.

Example of essential paradigm in computer science.

- Build intermediate abstractions.
- Pick the right ones!
- Solve important practical problems.