5.4 Regular Expressions

- regular expressions
- REs and NFAs
- NFA simulation
- NFA construction
- applications
• 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 \textit{cgg} or \textit{agg}, bracketed by \textit{gcg} at the beginning and \textit{ctg} at the end.
- Number of repeats is variable, and correlated with syndrome.

\[
\text{pattern} \quad \text{GCG (CGG | AGG) * CTG} \\
\text{text} \quad \text{(highlighted text)}
\]
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.

<table>
<thead>
<tr>
<th>operation</th>
<th>example RE</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td><strong>AABAAB</strong></td>
<td><strong>AABAAB</strong></td>
<td>every other string</td>
</tr>
<tr>
<td>or</td>
<td>**AA</td>
<td>BAAB**</td>
<td><strong>AA</strong></td>
</tr>
<tr>
<td>closure</td>
<td><strong>AB*A</strong></td>
<td><strong>AA</strong></td>
<td><strong>AB</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ABBBBBBBBBBA</strong></td>
<td><strong>ABABA</strong></td>
</tr>
<tr>
<td>parentheses</td>
<td>**A (A</td>
<td>B) AAB**</td>
<td><strong>AAAAB</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ABAAB</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*<em>(AB) <em>A</em></em></td>
<td><strong>A</strong></td>
<td><strong>AA</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ABABABABABABA</strong></td>
<td><strong>ABBA</strong></td>
</tr>
</tbody>
</table>
Regular expression shortcuts

Additional operations are often added for convenience.

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

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

Notation is surprisingly expressive

<table>
<thead>
<tr>
<th>regular expression</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>.<em>SPB..</em></td>
<td>RASPBERRY</td>
<td>SUBSPACE</td>
</tr>
<tr>
<td></td>
<td>(contains the trigraph spb)</td>
<td>CRISPBREAD</td>
</tr>
<tr>
<td>[0-9]{3}-[0-9]{2}-[0-9]{4}]</td>
<td>166-11-4433 166-45-1111</td>
<td>11-55555555 8675309</td>
</tr>
<tr>
<td></td>
<td>(Social Security numbers)</td>
<td></td>
</tr>
<tr>
<td>[a-z]+@[(a-z)+.]+(edu</td>
<td>com)</td>
<td><a href="mailto:wayne@princeton.edu">wayne@princeton.edu</a> <a href="mailto:rs@princeton.edu">rs@princeton.edu</a></td>
</tr>
<tr>
<td></td>
<td>(valid email addresses)</td>
<td></td>
</tr>
<tr>
<td>$_A-Za-z[$_A-Za-z0-9]*</td>
<td>ident3</td>
<td>3a</td>
</tr>
<tr>
<td></td>
<td>(valid Java identifiers)</td>
<td>PatternMatcher</td>
</tr>
</tbody>
</table>

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
Perl RE for valid RFC822 email addresses

Can the average programmer learn to use REs?

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)

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).
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**?
Nondeterministic finite-state automata

Regular-expression-matching NFA.
• RE enclosed in parentheses.
• One state per RE character (start = 0, accept = M).
• Red $\varepsilon$-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 \(( ( A \ast B | A C ) D )\)
Q. Is aaaabd matched by NFA?
A. Yes, because some sequence of legal transitions ends in state 11.
**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 ]
Q. Is $aaac$ matched by NFA?
A. No, because no sequence of legal transitions ends in state 11. [ but need to argue about all possible sequences ]
**Nondeterminism**

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

DFA. Deterministic $\Rightarrow$ exactly one applicable transition.

NFA. Nondeterministic $\Rightarrow$ 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 $$( ( A \ast B \mid A C ) D )$$. 

accept state
Pattern matching implementation: basic plan (revised)

Overview is similar to KMP.
• No backup in text input stream.
• Quadratic-time guarantee (linear-time typical).

Underlying abstraction. Non-deterministic 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 \( M \).

**Match-transitions.** Keep regular expression in array \( re[] \).

**\( \varepsilon \)-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 \( ( ( A * B \mid A C ) D ) \)
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.

Q. How to perform reachability?
NFA simulation example

Simulation of \(( ( A \ast B \mid A \ C ) \ D )\) NFA for input \(A \ A \ B \ D\)
NFA simulation example (continued)

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

- **Set of states reachable via $\epsilon$-transitions after matching $A A$:**
  - States: $2 3 4$
  - Transition: $0 \rightarrow 4$

- **Set of states reachable after matching $A A$:**
  - States: $5$
  - Transition: $4 \rightarrow 5$

- **Set of states reachable via $\epsilon$-transitions after matching $A A B$:**
  - States: $5 8 9$
  - Transition: $5 \rightarrow 8$

- **Set of states reachable after matching $A A B$:**
  - States: $10$
  - Transition: $9 \rightarrow 10$

- **Set of states reachable via $\epsilon$-transitions after matching $A A B D$:**
  - States: $10 11$
  - Transition: $10 \rightarrow 11$

*accept!*
## Digraph reachability

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public class DirectedDFS</code></td>
<td></td>
</tr>
<tr>
<td><code>DirectedDFS(Digraph G, int s)</code></td>
<td>find vertices reachable from s</td>
</tr>
<tr>
<td><code>DirectedDFS(Digraph G, Iterable&lt;Integer&gt; sources)</code></td>
<td>find vertices reachable from sources</td>
</tr>
<tr>
<td><code>boolean marked(int v)</code></td>
<td>is v reachable from source(s)?</td>
</tr>
</tbody>
</table>
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.toCharArray();
        G = buildEpsilonTransitionsGraph();
    }

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

   for (int i = 0; i < txt.length(); i++)
   {
      Bag<Integer> match = new Bag<Integer>();
      for (int v : pc)
      {
         if (v == M) continue;
         if ((re[v] == txt.charAt(i)) || re[v] == '.')
            match.add(v+1);
      }
      dfs = new DirectedDFS(G, match);
      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;
   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 $MN$ 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 $\varepsilon$-transitions. (The NFA construction we consider ensures the number of edges in $G \leq 3M$.)
- 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.** ( ) . * |
Building an NFA corresponding to an RE

**Parentheses.** Add \( e \)-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 $\varepsilon$-transition edges for each * operator.

The NFA corresponding to the pattern $( ( A * B | A C ) D )$ is shown in the diagram below.
Building an NFA corresponding to an RE

Add two $\varepsilon$-transition edges for each $|$ operator.

NFA corresponding to the pattern $((A \ast B \mid A \cdot C) \cdot D)$
NFA construction: implementation

**Goal.** Write a program to build the $\varepsilon$-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 $\varepsilon$-transition edges for closure/or.

![NFA diagram](image)
NFA construction: example

Building the NFA corresponding to \(( ( A \ast B \mid A C ) D )\)
NFA construction: example

Building the NFA corresponding to \(( ( A \ast B \mid A \cdot C ) \cdot D )\)
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;
        } 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;
        } else if (re[i] == '(' || re[i] == '*' || re[i] == ')')
            G.addEdge(i, i+1);
        if (i < M-1 && re[i+1] == '*') {
            G.addEdge(lp, i+1);
            G.addEdge(i+1, lp);
        }
    }
    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 $\varepsilon$-transitions and execute at most two stack operations.

![Diagram of NFA corresponding to the pattern ( ( A * B | A C ) D )](image)
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.

```java
public class GREP {
   public static void main(String[] args)
   {
      String regexp = "(.*) + args[0] + ".*"; // find lines containing RE as a substring
      NFA nfa = new NFA(regexp);
      while (StdIn.hasNextLine())
      {
         String line = StdIn.readLine();
         if (nfa.recognizes(line))
            StdOut.println(line);
      }
   }
}
```

**Bottom line.** Worst-case for grep (proportional to $MN$) is the same as for elementary exact substring match.
Typical grep application: crossword puzzles

% more words.txt
a
aback
abacus
abalone
abandon
...
% grep 's..ict..' words.txt
constrictor
stricter
stricture

dictionary
(standard in Unix)
also on booksite
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>`?

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

```java
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
Harvesting information

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

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

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

harvest patterns from DNA

harvest links from website
RE pattern matching is implemented in Java’s Pattern and Matcher classes.

```java
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!

SpamAssassin regular expression.

- 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.

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: atttcggaat.

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.
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.