7. Theory of Computation

Introduction to Theoretical CS

Q. What can a computer do?
Q. What can a computer do with limited resources?

General approach:
- Don’t talk about specific machines or problems.
- Consider minimal abstract machines.
- Consider general classes of problems.

Pioneering work in the 1930s.
- Princeton == center of universe.
- Automata, languages, computability, universality, complexity, logic.

Why Learn Theory?

In theory ...
- Deeper understanding of what is a computer and computing.
- Foundation of all modern computers.
- Pure science.
- Philosophical implications.

In practice ...
- Web search: theory of pattern matching.
- Sequential circuits: theory of finite state automata.
- Compilers: theory of context free grammars.
- Cryptography: theory of computational complexity.
- Data compression: theory of information.

"In theory there is no difference between theory and practice. In practice there is." – Yogi Berra
Regular Expressions

Describing a Pattern

**PROSITE.** Huge database of protein families and domains.

**Q.** How to describe a protein motif?

**Ex.** [signature of the C2H2-type zinc finger domain]

- **C**
  - Between 2 and 4 amino acids.
- **C**
  - 3 more amino acids.
  - One of the following amino acids: LIVMFYWCX.
  - 8 more amino acids.
- **H**
  - Between 3 and 5 more amino acids.

**Pattern Matching Applications**

- Test if a string matches some pattern.
  - Process natural language.
  - Scan for virus signatures.
  - Access information in digital libraries.
  - Search-and-replace in a word processors.
  - Filter text (spam, NetNanny, ads, 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 TOY input file format.
  - Automatically create Java documentation from Javadoc comments.

**Regular Expressions: Basic Operations**

**Regular expression.** Notation to specify a set of strings.

<table>
<thead>
<tr>
<th>operation</th>
<th>regular expression</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td>aabaab</td>
<td>aabaab</td>
<td>every other string</td>
</tr>
<tr>
<td>wildcard</td>
<td>.u.u.u.</td>
<td>cumulus jugulum</td>
<td>succubus tumultuous</td>
</tr>
<tr>
<td>union</td>
<td>aa</td>
<td>baab</td>
<td>aa</td>
</tr>
<tr>
<td>closure</td>
<td>ab*a</td>
<td>aa</td>
<td>ababa</td>
</tr>
<tr>
<td>parentheses</td>
<td>a(ab)sab</td>
<td>aaab</td>
<td>aaab</td>
</tr>
<tr>
<td></td>
<td>(ab)*a</td>
<td>a</td>
<td>aa abababa</td>
</tr>
</tbody>
</table>
Regular Expressions: Examples

Regular expression. Notation is surprisingly expressive.

**Table: Regular Expressions**

<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 crispbread</td>
<td>subspace subspecies</td>
</tr>
<tr>
<td>a*</td>
<td>(a<em>ba</em>ba*)*</td>
<td>multiple of three V's</td>
</tr>
<tr>
<td>.*0....</td>
<td>1000234 98701234</td>
<td>111111111 403992772</td>
</tr>
<tr>
<td>gcg(cgg</td>
<td>agg)*ctg</td>
<td>gcgcggctg gcgcgggcgcggctg gcgcggctg gcgcggaggctg</td>
</tr>
</tbody>
</table>

**Examples**

- `a* | (a*ba*ba*)*` is shorthand for `(a|b|c|d|e)(a|b|c|d|e)*`.
- `\s` is shorthand for “any whitespace character” (space, tab, ...).

**Table: Generalized Regular Expressions**

<table>
<thead>
<tr>
<th>operation</th>
<th>regular expression</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>one or more</td>
<td>a(bc)+de</td>
<td>abode</td>
<td>ade bcde</td>
</tr>
<tr>
<td>character class</td>
<td>[A-Za-z][a-z]*</td>
<td>lowercase Capitalized</td>
<td>camelCase illegal</td>
</tr>
<tr>
<td>exactly k</td>
<td>[0-9]{5}-[0-9]{4}</td>
<td>08540-1321 19072-5541</td>
<td>111111111 166-54-1111</td>
</tr>
<tr>
<td>negation</td>
<td><img src="image.png" alt="image" /></td>
<td>rhythm</td>
<td>decade</td>
</tr>
</tbody>
</table>

**Examples in Java**

Validity checking. Is input in the set described by the re?

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

**String Searching Methods**

```java
String s = StdIn.readString();
```

replace each sequence of at least one whitespace character with a single space
String Searching Methods

public class String  (Java's String library)

boolean matches(String re)  does this string match the given regular expression
String replaceAll(String re, String str)  replace all occurrences of regular expression with the replacement string
int indexOf(String r, int from)  return the index of the first occurrence of the string r after the index from
String[] split(String re)  split the string around matches of the given regular expression

String s = StdIn.readAll();
String[] words = s.split("\s");

create an array of the words in StdIn

Solving the Pattern Match Problem

Regular expressions are a concise way to describe patterns.
• How would you implement the method matches()?
• Hardware: build a deterministic finite state automaton (DFA).
• Software: simulate a DFA.

DFA: simple machine that solves a pattern match problem.
• Different machine for each pattern.
• Accepts or rejects string specified on input tape.
• Focus on true or false questions for simplicity.

Deterministic Finite State Automaton (DFA)

Simple machine with N states.
• Begin in start state.
• Read first input symbol.
• Move to new state, depending on current state and input symbol.
• Repeat until last input symbol read.
• Accept input string if last state is labeled Y.
DFA and RE Duality

RE. Concise way to describe a set of strings.
DFA. Machine to recognize whether a given string is in a given set.

Duality.
• 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.

Practical consequence of duality proof: to match RE
• build DFA
• simulate DFA on input string.

Application: Harvester

Harvest information from input stream.
• Harvest patterns from DNA.

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

• Harvest email addresses from web for spam campaign.

```
% java Harvester "[a-z]+@[a-z]+\.(edu|com)" http://www.princeton.edu/~cos126
rs@cs.princeton.edu
mail@cs.princeton.edu
doug@cs.princeton.edu
wayne@cs.princeton.edu
```

Implementing a Pattern Matcher

Problem. Given a RE, create program that tests whether given input is in set of strings described.

Step 1. Build the DFA.
• A compiler!
• See COS 226 or COS 320.

Step 2. Simulate it with given input.

```
State state = start;
while (!StdIn.isEmpty())
{
  char c = StdIn.readChar();
  state = state.next(c);
}
StdOut.println(state.accept());
```

Application: Harvester

Harvest information from input stream.
• Use Pattern data type to compile regular expression to NFA.
• Use Matcher data type to simulate NFA.

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

public class Harvester
{
  public static void main(String[] args)
  {
    String re = args[0];
    In in = new In(args[1]);
    String input = in.readAll();
    Pattern pattern = Pattern.compile(re);
    Matcher matcher = pattern.matcher(input);

    look for next match
    Matcher matcher = pattern.matcher(input);

    while (matcher.find())
      StdOut.println(matcher.group());
  }
}
```
Application: Parsing a Data File

**Ex:** parsing an NCBI genome data file.

```
LOCUS AC146846 128142 bp DNA linear HTG 13-NOV-2003
DEFINITION Ornithorhynchus anatinus clone CLM1-393H9,
ACCESSION AC146846
VERSION AC146846.2 GI:38304214
KEYWORDS HTG; HTGS_PHASE2; HTGS_DRAFT.
SOURCE Ornithorhynchus anatinus (platypus)
ORIGIN
1 tgtatttcat ttgaccgtgc tgttttttcc cggtttttca gtacggtgtt agggagccac
61 gtgattctgt ttgttttatg ctgccgaata gctgctcgat gaatctctgc atagacagct // a comment
121 gccgcaggga gaaatgacca gtttgtgatg acaaaatgta ggaaagctgt ttcttcataa...
128101 ggaaatgcga cccccacgct aatgtacagc ttctttagat tg
```

**Goal.** Extract the data as a single *actg* string.

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

public class ParseNCBI {
  public static void main(String[] args) {
    String re = "\[\s*\[0-9\]+(\[actg\]*)\].*";
    Pattern pattern = Pattern.compile(re);
    In in = new In(args[0]);
    String data = "";
    while (!in.isEmpty()) {
      String line = in.readLine();
      Matcher matcher = pattern.matcher(line);
      if (matcher.find())
        data += matcher.group(1).replaceAll(" ", "");
    }
    System.out.println(data);
  }
}
```

**Summary**

**Programmer.**
- Regular expressions are a powerful pattern matching tool.
- Implement regular expressions with finite state machines.

**Theoretician.**
- RE is a compact description of a set of strings.
- DFA is an abstract machine that solves RE pattern match problem.

**You.** Practical application of core CS principles.
Q. Are there patterns that cannot be described by any RE?
A. Yes.
• Bit strings with equal number of 0s and 1s.
• Strings that represent legal REs.
• Decimal strings that represent prime numbers.
• DNA strings that are Watson-Crick complemented palindromes.

Q. Are there languages that cannot be recognized by any DFA?
A. Yes.
• Bit strings with equal number of 0s and 1s.
• Strings that represent legal REs.
• Decimal strings that represent prime numbers.
• DNA strings that are Watson-Crick complemented palindromes.

Fundamental problem: DFA lacks memory.
Fundamental Questions

Q. Are there machines that are more powerful than a DFA?  
A. Yes.

A 1-stack DFA can recognize:  
• Bit strings with equal number of 0s and 1s.  
• Legal REs.  
• Watson-Crick complemented palindromes.

Fundamental Questions

Q. Are there machines that are more powerful than a 1-stack DFA?  
A. Yes.

A 2-stack DFA can recognize:  
• Prime numbers.  
• Legal Java Programs.

Fundamental Questions

Q. Are there machines that are more powerful than a 2-stack DFA?  
A. No! Not even a supercomputer!

2-stack DFAs are equivalent to Turing machines [stay tuned].