The Theory Behind r.e.'s

- That’s it for the basics of Ruby
  - If you need other material for your project, you’ll either see it in discussion section, or you’ll need to learn it on your own

- Next up: How do r.e.’s really work?
  - Mixture of a very practical tool (string matching with r.e.’s) and some nice theory
  - A great computer science result

A Few Questions about Regular Expressions

- What does a regular expression represent?
  - Just a set of strings
- What are the basic components of r.e.’s?
  - E.g., we saw that $e+$ is the same as $ee^*$
- How are r.e.’s implemented?
  - We’ll see how to turn a r.e. into a program
- Can r.e.’s represent all possible languages?
  - The answer turns out to be no!
  - The languages represented by regular expressions are called, appropriately, the regular languages

Some Definitions

- An *alphabet* is a finite set of symbols
  - Usually denoted $\Sigma$
- A *string* is a finite sequence of symbols from $\Sigma$
  - $\epsilon$ is the empty string ("" in Ruby)
  - $|s|$ is the length of string $s$
    - $|Hello| = 5$, $|\epsilon| = 0$
    - Note: $\emptyset$ is the empty set (with 0 elements); $\emptyset \neq \{ \epsilon \}$
- *Concatenation* is indicated by juxtaposition
  - If $s_1 = \text{super}$ and $s_2 = \text{hero}$, then $s_1s_2 = \text{superhero}$
  - Sometimes also written $s_1 \cdot s_2$
  - For any string $s$, we have $s\epsilon = \epsilon s = s$
Languages

• A language is a set of strings over an alphabet

• Example: The set of phone numbers over the alphabet \( \Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 9, (, ), -\} \)
  – Give an example element of this language
  – Are all strings over the alphabet in the language?
  – Is there a Ruby regular expression for this language?
    • Is the Ruby regular expression over the same alphabet?

• Example: The set of all strings over \( \Sigma \)
  – Often written \( \Sigma^* \)

Operations on Languages

• Let \( \Sigma \) be an alphabet and let \( L, L_1, L_2 \) be languages over \( \Sigma \)

• Concatenation \( L_1L_2 \) is defined as
  – \( L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\} \)
  – Example: \( L_1 = \{\text{“hi”, “bye”}\}, L_2 = \{\text{“1”, “2”}\} \)
    • \( L_1L_2 = \{\text{“hi1”, “hi2”, “bye1”, “bye2”}\} \)

• Union is defined as
  – \( L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\} \)
  – Example: \( L_1 = \{\text{“hi”, “bye”}\}, L_2 = \{\text{“1”, “2”}\} \)
    • \( L_1 \cup L_2 = \{\text{“hi”, “bye”, “1”, “2”}\} \)

Languages (cont’d)

• Example: The set of all valid Ruby programs
  – Is there a Ruby regular expression for this language?

• Example: The set of strings of length 0 over the alphabet \( \Sigma = \{a, b, c\} \)
  – \( \{s \mid s \in \Sigma^* \text{ and } |s| = 0\} = \{\epsilon\} \neq \emptyset \)

Operations on Languages (cont’d)

• Define \( L^n \) inductively as
  – \( L^0 = \{\epsilon\} \)
  – \( L^n = LL^{n-1} \) for \( n > 0 \)

• In other words,
  – \( L^1 = LL^0 = L\{\epsilon\} = L \)
  – \( L^2 = LL^1 = LL \)
  – \( L^3 = LL^2 = LLL \)
  – ...
Examples of \( L^n \)

- Let \( L = \{a, b, c\} \)
- Then
  - \( L^0 = \{\varepsilon\} \)
  - \( L^1 = \{a, b, c\} \)
  - \( L^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\} \)

Operations on Languages (cont’d)

- **Kleene closure** is defined as
  \[
  L^* = \bigcup_{i \in \{0, \ldots\}} L^i
  \]
- In other words...
  \( L^* \) is the language (set of all strings) formed by concatenating together zero or more strings from \( L \)

Definition of Regexps

- Given an alphabet \( \Sigma \), the regular expressions over \( \Sigma \) are defined inductively as

<table>
<thead>
<tr>
<th>regular expression</th>
<th>denotes language</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \varepsilon )</td>
<td>( {\varepsilon} )</td>
</tr>
<tr>
<td>each element ( \sigma \in \Sigma )</td>
<td>( {\sigma} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>regular expression</th>
<th>denotes language</th>
</tr>
</thead>
<tbody>
<tr>
<td>( AB )</td>
<td>( L_A L_B )</td>
</tr>
<tr>
<td>( (A</td>
<td>B) )</td>
</tr>
<tr>
<td>( A^* )</td>
<td>( L_A^* )</td>
</tr>
</tbody>
</table>

- There are no other regular expressions for \( \Sigma \)
- We use \((\cdot)\)'s as needed for grouping
The Language Denoted by an r.e.

• For a regular expression $e$, we will write $[[e]]$ to mean the language denoted by $e$
  – $[[a]] = \{a\}$
  – $[[a(b)]] = \{a, b\}$

• If $s \in [[re]]$, we say that $re$ accepts, describes, or recognizes $s$.

Example 1

• All strings over $\Sigma = \{a, b, c\}$ such that all the $a$’s are first, the $b$’s are next, and the $c$’s last
  – Example: $aaaabbbccc$ but not $abcb$
  – Regexp: $a^*b^*c^*$
    – This is a valid regexp because...
    – $a$ is a regexp $([[a]] = \{a\})$
    – $a^*$ is a regexp $([[a^*]] = \{\varepsilon, a, aa, ...\})$
    – Similarly for $b^*$ and $c^*$
    – So $a^*b^*c^*$ is a regular expression

Example 2

• All strings over $\Sigma = \{a, b, c\}$
  – Regexp: $(a|b|c)^*$
  – Other regular expressions for the same language?
    – $(c|b|a)^*$
    – $(a^*|b^*|c^*)^*$
    – $(a^*b^*c^*)^*$
    – $(a^*b^*c^*)abc$
    – etc.

Which Strings Does $a^*b^*c^*$ Recognize?

- $aabbbcc$
  - Yes; $aa \in [[a^*]]$, $bbb \in [[b^*]]$, and $cc \in [[c^*]]$, so entire string is in $[[a^*b^*c^*]]$
- $abb$
  - Yes, $abb = abb\varepsilon$, and $\varepsilon \in [[c^*]]$
- $ac$
  - Yes
- $\varepsilon$
  - Yes
- $aacbc$
  - No
- $abcd$
  - No -- outside the language
Example 3

- All whole numbers containing the substring 330
- Regular expression: \((0|1|...|9)*330(0|1|...|9)\)*
- What if we want to get rid of leading 0's?
  - \((1|...|9)(0|1|...|9)*330(0|1|...|9)\)  \((0|1|...|9)\) *
- Any other solutions?
  - What about all whole numbers not containing the substring 330?
    - Is it recognized by a regexp?

Example 4

- What language does \((10|0)^*(10|1)^*\) denote?
  - \((10|0)^*\)
    - 0 may appear anywhere
    - 1 must always be followed by 0
  - \((10|1)^*\)
    - 1 may appear anywhere
    - 0 must always be preceded by 1
- Put together, all strings of 0’s and 1’s where every pair of adjacent 0’s precedes any pair of adjacent 1’s

What Strings are in \((10|0)^*(10|1)^*\) ?

- 00101000 110111101
  - First part in \([(10|0)^*]\)
  - Second part in \([(10|1)^*]\)
  - Notice that 0010 also in \([(10|0)^*]\)
  - But remainder of string is not in \([(10|1)^*]\)
- 0010101
  - Yes
- 101
  - Yes
- 011001
  - No

Example 5

- What language does this regular expression recognize?
  - \((1|\epsilon)(0|1|...|9)(2(0|1|2|3))\) : (0|1|...|5)(0|1|...|9)
- All valid times written in 24-hour format
  - 10:17
  - 23:59
  - 0:45
  - 8:30
Two More Examples

• \((000|00|1)^*\)
  – Any string of 0’s and 1’s with no single 0’s
• \((00|0000)^*\)
  – Strings with an even number of 0’s
  – Notice that some strings can be accepted more than one way
  • 000000 = 00-00-00 = 00·0000 = 0000·00

Regular Languages

• The languages that can be described using regular expressions are the regular languages or regular sets
• Not all languages are regular
  – Examples (without proof):
    • The set of palindromes over \(\Sigma\)
    • \(\{a^nb^n \mid n > 0\}\) \((a^n = \text{sequence of } n \text{ a’s})\)
• Almost all programming languages are not regular
  – But aspects of them sometimes are (e.g., identifiers)
  – Regular expressions are commonly used in parsing tools

Ruby Regular Expressions

• Almost all of the features we’ve seen for Ruby r.e.’s can be reduced to this formal definition
  – /Ruby/ – concatenation of single-character r.e.’s
  – /(Ruby|Regular)/ – union
  – /(Ruby)^*/ – Kleene closure
  – /(Ruby)+/ – same as \((Ruby)(Ruby)^*\)
  – /(Ruby)?/ – same as \(\epsilon(Ruby)\) \((\epsilon \text{ is } \epsilon)\)
  – /[a-z]/ – same as \(a|b|c|…|z\)
  – /[^0-9]/ – same as \(a|b|c|…\) for \(a,b,c,\ldots \in \Sigma - \{0..9\}\)
  – ^, $ – correspond to extra characters in alphabet

Implementing Regular Expressions

• We can implement regular expressions by turning them into a finite automaton
  – A “machine” for recognizing a regular language
Example

- Machine starts in *start* or *initial* state
- Repeat until the end of the string is reached:
  - Scan the next symbol \( s \) of the string
  - Take transition edge labeled with \( s \)
- The string is *accepted* if the automaton is in a *final* or *accepting* state when the end of the string is reached.

Example

- All strings over \( \{0, 1\} \) that end in 1
- What is a regular expression for this language? \((0|1)^*1\)
Formal Definition

- A deterministic finite automaton (DFA) is a 5-tuple \((\Sigma, Q, q_0, F, \delta)\) where
  - \(\Sigma\) is an alphabet
  - the strings recognized by the DFA are over this set
  - \(Q\) is a nonempty set of states
  - \(q_0 \in Q\) is the start state
  - \(F \subseteq Q\) is the set of final states
  - How many can there be?
  - \(\delta : Q \times \Sigma \rightarrow Q\) specifies the DFA’s transitions
    - What’s this definition saying that \(\delta\) is?

More on DFAs

- An FSA can have more than one final state:
  - A string is accepted as long as there is at least one path to a final state

Our Example, Formally

- \(\Sigma = \{0, 1\}\)
- \(Q = \{S0, S1\}\)
- \(q_0 = S0\)
- \(F = \{S1\}\)
- \(\delta\) is a transition function:
  - \(\delta\) specifies transitions in the DFA

Another Example

<table>
<thead>
<tr>
<th>string</th>
<th>state at end</th>
<th>accepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>aabcc</td>
<td>S2</td>
<td>Y</td>
</tr>
<tr>
<td>acc</td>
<td>S2</td>
<td>Y</td>
</tr>
<tr>
<td>bbc</td>
<td>S2</td>
<td>Y</td>
</tr>
<tr>
<td>aabbb</td>
<td>S1</td>
<td>Y</td>
</tr>
<tr>
<td>aa</td>
<td>S0</td>
<td>Y</td>
</tr>
<tr>
<td>(\varepsilon)</td>
<td>S0</td>
<td>Y</td>
</tr>
<tr>
<td>acba</td>
<td>S3</td>
<td>N</td>
</tr>
</tbody>
</table>

(a,b,c notation shorthand for three self loops)
Another Example (cont’d)

What language does this DFA accept? $a^*b^*c^*$

$S_3$ is a dead state – a nonfinal state with no transition to another state

Shorthand Notation

- If a transition is omitted, assume it goes to a dead state that is not shown

Language?

Strings over $\{0,1,2,3\}$ with alternating even and odd digits, beginning with odd digit

What Lang. Does This DFA Accept?

$a^*b^*c^*$ again, so DFAs are not unique

Nondeterministic Finite Automata (NFA)

- An NFA is a 5-tuple $(\Sigma, Q, q_0, F, \delta)$ where
  - $\Sigma$ is an alphabet
  - $Q$ is a nonempty set of states
  - $q_0 \in Q$ is the start state
  - $F \subseteq Q$ is the set of final states
  - There may be 0, 1, or many
  - $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ specifies the NFA’s transitions
  - Transitions on $\varepsilon$ are allowed – can optionally take these transitions without consuming any input
  - Can have more than one transition for a given state and symbol
- An NFA accepts $s$ if there is at least one path from its start to final state on $s$
Example DFA

- $S_0 = \text{“Haven’t seen anything yet”}$
- $S_1 = \text{“Last symbol seen was an } a\text{”}$
- $S_2 = \text{“Last two symbols seen were } ab\text{”}$
- $S_3 = \text{“Last three symbols seen were } abb\text{”}$

• Language?
  • $(a|b)^*abb$

NFA for $(a|b)^*abb$

- $ba$
  - Has paths to either $S_0$ or $S_1$
  - Neither is final, so rejected
- $babaabb$
  - Has paths to different states
  - One leads to $S_3$, so accepted

Another example DFA

• Language?
  • $(ab|aba)^*$

NFA for $(ab|aba)^*$

- $aba$
  - Has paths to states $S_0, S_1$
- $ababa$
  - Has paths to $S_0, S_1$
  - Need to use $\epsilon$-transition
Relating R.E.'s to DFAs and NFAs

- Regular expressions, NFAs, and DFAs accept the same languages!

\[ \text{DFA} \rightarrow \text{NFA} \]
\[ \text{can transform} \]
\[ \text{r.e.} \rightarrow \text{NFA} \]
\[ \text{can transform} \]
\[ \text{(we'll discuss this next)} \]

Reducing Regular Expressions to NFAs

- Goal: Given regular expression \( e \), construct NFA \( \langle e \rangle = (\Sigma, Q, q_0, F, \delta) \)
  - Remember r.e. defined recursively from primitive r.e. languages
  - Invariant: \( |F| = 1 \) in our NFAs

- Base case: \( a \)

\[ \langle a \rangle = (\{a\}, \{S0, S1\}, S0, \{S1\}, \{(S0, a, S1)\}) \]

Reduction (cont’d)

- Base case: \( \epsilon \)

\[ \langle \epsilon \rangle = (\{\epsilon\}, \{S0\}, S0, \{S0\}, \emptyset) \]

- Base case: \( \emptyset \)

\[ \langle \emptyset \rangle = (\emptyset, \{S0, S1\}, S0, \{S1\}, \emptyset) \]

Reduction (cont’d)

- Induction: \( AB \)

\[ \langle A \rangle \]
\[ \langle B \rangle \]
Reduction (cont’d)

• Induction: AB

\[<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)\]
\[<B> = (\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)\]
\[<AB> = (\Sigma_A \cup \Sigma_B, Q_A \cup Q_B, q_A, \{f_A\}, \delta_A \cup \delta_B \cup \{(f_A, \varepsilon, q_B)\})\]

Reduction (cont’d)

• Induction: (A|B)

\[<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)\]
\[<B> = (\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)\]
\[<(A|B)> = (\Sigma_A \cup \Sigma_B, Q_A \cup Q_B \cup \{S0, S1\}, S0, \{S1\}, \delta_A \cup \delta_B \cup \{(S0, \varepsilon, q_A), (S0, \varepsilon, q_B), (f_A, \varepsilon, S1), (f_B, \varepsilon, S1)\})\]

Reduction (cont’d)

• Induction: A*
Reduction (cont’d)

- Induction: \( A^* \)

\[
\langle A \rangle = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)
\]
\[
\langle A^* \rangle = (\Sigma_A, Q_A \cup \{S0,S1\}, S0, \{S1\}, \\
\delta_A \cup \{(f_A,\epsilon,S1), (S0,\epsilon,q_A), (S0,\epsilon,S1), (S1,\epsilon,S0)\})
\]

Relating R.E.’s to DFAs and NFAs

DFA can transform NFA can transform
(we’ll discuss this next)

Equivalence of DFAs and NFAs

- Let subsets of states be states in DFA
- Keep track of which subset you can be in

- Any string from \( \{A\} \) to either \( \{D\} \) or \( \{CD\} \) represents a path from \( A \) to \( D \) in the original NFA.

Reduction Complexity

- Given a regular expression \( A \) of size \( n \)... Size = # of symbols + # of operations
- How many states does \( \langle A \rangle \) have?
  - \( \mathcal{O}(n) \)
  - That’s pretty good!
- NFA to DFA reduction
  - Intuition: Build DFA where each DFA state represents a set of NFA states
  - Given NFA with \( n \) states, DFA may have \( 2^n \) states
  - Not so good, since DFAs are what we can implement easily
Implementing DFAs

It's easy to build a program which mimics a DFA.

```c
cur_state = 0;
while (1) {
    symbol = getchar();
    switch (cur_state) {
        case 0: switch (symbol) {
            case '0': cur_state = 0; break;
            case '1': cur_state = 1; break;
            case '\n': printf("rejected\n"); return 0;
            default: printf("rejected\n"); return 0;
        }
        break;
        case 1: switch (symbol) {
            case '0': cur_state = 0; break;
            case '1': cur_state = 1; break;
            case '\n': printf("accepted\n"); return 1;
            default: printf("rejected\n"); return 0;
        }
        break;
        default: printf("unknown state; I'm confused\n");
    }
}
```

Implementing DFAs (Alternative)

Alternatively, use generic table-driven DFA

```c
given components (Σ, Q, q₀, F, δ) of a DFA:
let q = q₀
while (there exists another symbol s of the input string)
    q := δ(q, s);
if q ∈ F then
    accept
else reject
```

- q is just an integer
- Represent δ using arrays or hash tables
- Represent F as a set

Relating R.E.'s to DFAs and NFAs

- Regular expressions, NFAs, and DFAs accept the same languages!

Run Time of Algorithm

- Given a string s, how long does algorithm take to decide whether s is accepted?
  - Assume we can compute δ(q₀, c) in constant time
  - Then the time per string s to determine acceptance is O(|s|)
  - Can’t get much faster!
- But recall that constructing the DFA from the regular expression A may take O(2^|A|) time
  - But this is usually not the case in practice
- So there’s the initial overhead, but then accepting strings is fast
Regular Expressions in Practice

• Regular expressions are typically “compiled” into tables for the generic algorithm
  – Can think of this as a simple byte code interpreter
  – But really just a representation of \((\Sigma, Q_A, q_A, \{f_A\}, \delta_A)\), the components of the DFA produced from the r.e.

• Regular expression implementations often have extra constructs that are non-regular
  – I.e., can accept more than the regular languages
  – Can be useful in certain cases
  – Disadvantages: nonstandard, plus can have higher complexity