CMSC 330: Organization of Programming Languages

Theory of Regular Expressions
Introduction

- That’s it for the basics of Ruby
  - If you need other material for your project, come to office hours or check out the documentation

- Next up: How do regular expressions (REs) really work?
  - Mixture of a very practical tool (string matching with REs) and some nice theory
  - A great computer science result
A Few Questions About REs

► What does a regular expression represent?
  • Just a set of strings

► What are the basic components of REs?
  • E.g., we saw that $e^+$ is the same as $ee^*$

► How are REs implemented?
  • We’ll see how to build a structure to parse REs
Definition: Alphabet

- An alphabet is a finite set of symbols
  - Usually denoted $\Sigma$

Example alphabets:
- Binary: $\Sigma = \{0, 1\}$
- Decimal: $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Alphanumeric: $\Sigma = \{0-9, a-z, A-Z\}$
Definition: String

- A string is a finite sequence of symbols from $\Sigma$.
  - $\varepsilon$ is the empty string (""") in Ruby.
  - $|s|$ is the length of string $s$.
    - $|\text{Hello}| = 5$, $|\varepsilon| = 0$
  - Note
    - $\emptyset$ is the empty set (with 0 elements)
    - $\emptyset \neq \{ \varepsilon \} \neq \varepsilon$

- Example strings:
  - $0101 \in \Sigma = \{0,1\}$ (binary)
  - $0101 \in \Sigma = \text{decimal}$
  - $0101 \in \Sigma = \text{alphanumeric}$
Definition: Concatenation

- **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 \( s\varepsilon = \varepsilon s = s \)
- You can concatenate strings from different alphabets; then the new alphabet is the union of the originals:
  - If \( s_1 = \text{super} \in \Sigma_1 = \{s,u,p,e,r\} \) and \( s_2 = \text{hero} \in \Sigma_2 = \{h,e,r,o\} \), then \( s_1s_2 = \text{superhero} \in \Sigma_3 = \{e,h,o,p,r,s,u\} \)
Definition: Language

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: \((123) \ 456-7890\)
- Are all strings over the alphabet in the language? No
- Is there a Ruby regular expression for this language? \\
  \( /\(\d{3,3}\)\) \d{3,3}-\d{4,4}/\)

Example: The set of all strings over \( \Sigma \)

- Often written \( \Sigma^* \)
Definition: Language (cont.)

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

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

  No. Matching (an arbitrary number of) brackets so that they are balanced is impossible. {$ \{ \{ \{ \ldots \} \} \}$}

- Can REs represent all possible languages?
  - The answer turns out to be no!
  - The languages represented by regular expressions are called, appropriately, the regular languages
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: $L_1 = \{“hi”, “bye”\}, L_2 = \{“1”, “2”\}$
  - $L_1L_2 = \{“hi1”, “hi2”, “bye1”, “bye2”\}$

Union is defined as

- $L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\}$
- Example: $L_1 = \{“hi”, “bye”\}, L_2 = \{“1”, “2”\}$
  - $L_1 \cup L_2 = \{“hi”, “bye”, “1”, “2”\}$
Operations on Languages (cont.)

- Define $L^n$ inductively as
  - $L^0 = \{\epsilon\}$
  - $L^n = LL^{n-1}$ for $n > 0$

- In other words,
  - $L^1 = LL^0 = L\{\epsilon\} = L$
  - $L^2 = LL^1 = LL$
  - $L^3 = LL^2 = LLL$
  - $\ldots$
Examples of $L^n$

- Let $L = \{a, b, c\}$
- Then
  - $L^0 = \{\varepsilon\}$
  - $L^1 = \{a, b, c\}$
  - $L^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$
- How many elements will $L^3$ have? $L^n$? How long will each element be in these languages?
Operations on Languages (cont.)

- **Kleene closure** is defined as
  \[ L^* = \bigcup_{i \in \mathbb{N}} L^i \]
  \((\mathbb{N} = \{0,1,2,\ldots\} \text{ is the set of natural numbers})\)

- In other words...
  \(L^*\) is the language (set of all strings) formed by concatenating together zero or more instances of strings from \(L\)
Definition: Regular Expressions

- 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>

Constants
Definition: Regular Expressions (cont.)

- Let $A$ and $B$ be regular expressions denoting languages $L_A$ and $L_B$, respectively.

<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>

Operations

- There are no other regular expressions over $\Sigma$. 
Precedence

- Order in which operators are applied
  - In arithmetic
    - Multiplication × > addition +
    - \(2 \times 3 + 4 = (2 \times 3) + 4 = 10\)
  - In regular expressions
    - Kleene closure * > concatenation > union |
    - \(ab|c = (a b) | c = \{“ab”, “c”\}\)
    - \(ab^* = a (b^*) = \{“a”, “ab”, “abb”...\}\)
    - \(a|b^* = a | (b^*) = \{“a”, “”, “b”, “bb”, “bbb”...\}\)
  - Can change order using parentheses ( )
    - E.g., \(a(b|c), (ab)^*, (a|b)^*\)
The Language Denoted by an RE

- For a regular expression $e$, we will write $[[e]]$ to mean the language denoted by $e$
  - $[[a]] = \{a\}$
  - $[[a|b]] = \{a, b\}$

- If $s \in [[\text{RE}]]$, 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: aaabbbbbccc but not abcb

- Regexp: $a^*b^*c^*$
  - This is a valid regexp because:
    - a is a regexp ($[[a]] = \{a\}$)
    - $a^*$ is a regexp ($[[a^*]] = \{\epsilon, a, aa, \ldots\}$)
    - Similarly for $b^*$ and $c^*$
    - So $a^*b^*c^*$ is a regular expression
      (Remember that we need to check this way because regular expressions are defined inductively.)
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, $abb = abb\epsilon$, and $\epsilon \in [c^*]$

- **ac**
  - Yes

- **$\epsilon$**
  - Yes

- **aacbc**
  - No

- **abcd**
  - No -- outside the language
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.
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?
  \(\ ((1|...|9)(0|1|...|9)^*330(0|1|...|9)^* | 330(0|1|...|9)^* )\)
- Any other solutions?

- Challenge: What about all whole numbers not containing the substring 330?
  - Is it recognized by a regexp? Yes. We’ll see how to find it later…
Example 4

What is the English description for the language that \((10|0)^*(10|1)^*\) denotes?

- \((10|0)^*\)
  - 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
  - i.e., no 00 may appear after 11
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?
  - \((1|\epsilon)(0|1|\ldots|9) \mid (2(0|1|2|3))\) : (0|1|\ldots|5)(0|1|\ldots|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
    - \(000000 = 00 \cdot 00 \cdot 00 = 00 \cdot 0000 = 0000 \cdot 00\)
  - How else could we express this language?
    - \((00)^*\)
    - \((00|000000)^*\)
    - \((00|0000|000000)^*\)
    - etc…
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 $\Sigma$
      - reads the same backward or forward
    - $\{a^n b^n \mid n > 0 \}$ (where $a^n$ is a sequence of $n$ 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 REs can be reduced to this formal definition:

- `/Ruby/` – concatenation of single-character REs
- `/(Ruby|Regular)/` – union
- `/(Ruby)*/` – Kleene closure
- `/(Ruby)+/` – same as `(Ruby)(Ruby)*`
- `/(Ruby)?/` – same as `(ε|Ruby))` (// is ε)
- `/[a-z]/` – same as `(a|b|c|...|z)`
- `/[^0-9]/` – same as `(a|b|c|...)` for `a,b,c,... ∈ Σ - {0..9}`
- `^, $` – correspond to extra characters in alphabet
Summary

- Languages
  - Sets of strings
  - Operations on languages

- Regular expressions
  - Constants
  - Operators
  - Precedence
Implementing Regular Expressions

- We can implement a regular expression by turning it into a finite automaton
  - A “machine” for recognizing a regular language
Finite Automata

- 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 \( s \)
- String is accepted if automaton is in final state when end of string reached
Finite Automata: States

- **Start state**
  - State with incoming transition from no other state
  - Can have only 1 start state

- **Final states**
  - States with double circle
  - Can have 0 or more final states
  - Any state, including the start state, can be final
Finite Automaton: Example 1

0 0 1 0 1 1

accepted
Finite Automaton: Example 2

0 0 1 0 1 0
not accepted
What Language is This?

- All strings over \{0, 1\} that end in 1
- What is a regular expression for this language?
  
  \((0|1)^*1\)
Finite Automaton: Example 3

(a,b,c notation shorthand for three self loops)

<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>ε</td>
<td>S0</td>
<td>Y</td>
</tr>
<tr>
<td>acba</td>
<td>S3</td>
<td>N</td>
</tr>
</tbody>
</table>
Finite Automaton: Example 3 (cont.)

What language does this DFA accept?

\[ a^*b^*c^* \]

S3 is a dead state – a nonfinal state with no transition to another state
Dead 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 FA Accept?

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

Give the English descriptions and the DFA or regular expression of the following languages:

- \(((0|1)(0|1)(0|1)(0|1)(0|1))\)^*  
  - All strings with length a multiple of 5

- \((01)^*|(10)^*|(01)^*0|(10)^*1\)  
  - All alternating binary strings

- All binary strings containing the substring “11”
Finite Automaton: Example 4

Description for each state

- **S0** = “Haven’t seen anything yet” OR “seen zero or more b’s” OR “Last symbol seen was a b”
- **S1** = “Last symbol seen was an a”
- **S2** = “Last two symbols seen were ab”
- **S3** = “Last three symbols seen were abb”

Language?

- \((a|b)^*abb\)
Practice

Give the regular expressions and finite automata for the following languages

• You and your neighbors’ names
• All protein-coding DNA strings (including only ATCG and appearing in multiples of 3)
• All binary strings containing an even length substring of all 1’s
• All binary strings containing exactly two 1’s
• All binary strings that start and end with the same number
Types of Finite Automata

- Deterministic Finite Automata (DFA)
  - Exactly one sequence of steps for each string
  - All examples so far

- Nondeterministic Finite Automata (NFA)
  - May have many sequences of steps for each string
  - More compact
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?
- A DFA accepts \(s\) if it stops at a final state on \(s\)
Formal Definition: Example

- $\Sigma = \{0, 1\}$
- $Q = \{S0, S1\}$
- $q_0 = S0$
- $F = \{S1\}$

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
<td>S0</td>
<td>S1</td>
</tr>
</tbody>
</table>
DFA Requirement

- Cannot have more than one transition leaving a state on the same symbol
  - I.e., transition function must be a valid function

- NFAs do not have this requirement!
  - Can have any number of transitions leaving a state with the same symbol
  - DFA is a special case of NFA
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
  - $\delta \subseteq Q \times \Sigma \times Q$ specifies the NFA's transitions
    - 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 $s$
NFA for \((a\mid b)^*abb\)

- **ba**
  - Has paths to either S0 or S1
  - Neither is final, so rejected

- **babaabb**
  - Has paths to different states
  - One leads to S3, so accepted
Another example DFA

Language?
• \((ab|aba)^*\)
NFA for \((ab|aba)^*\)

- **aba**
  - Has paths to states S0, S1
- **ababa**
  - Has paths to S0, S1
NFA-ε

- There are versions of NFAs that allow ε-transitions (transitions that consume no symbol).
- They are equivalent to NFAs, DFAs, regular expressions.
- NFA-ε for \((ab|aba)^*\):
Relating REs to DFAs and NFAs

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

- **Goal**: Given regular expression $e$, construct NFA: $\langle e \rangle = (\Sigma, Q, q_0, F, \delta)$

- **Approach**
  - Remember regular expressions are defined recursively from primitive RE languages
  - We will define two recursive predicates on REs
    - $e \checkmark$ (Holds of RE $e$ if $\varepsilon \in [[e]]$)
    - $e \xrightarrow{a} e'$ (Holds if RE $e$ can “process” $a$ then behave like $e'$)
  - These relations will enable us to think of REs as being like programs that accept languages!
How Do We Use These Relations?

- We will be able to do the following when we have defined these relations:
  
  \[(ab|aba)^* \xrightarrow{a} b(ab|aba)^* \]
  
  \[- \xrightarrow{b} (ab|aba)^* \]
  
  \[- \xrightarrow{a} ba(ab|aba)^* \]
  
  \[- \xrightarrow{b} a(ab|aba)^* \]
  
  \[- \xrightarrow{a} (ab|aba)^* \checkmark \]

- It will turn out that \(ababa \in [(ab|aba)^*]\)

- To get \(\langle (ab|aba)^* \rangle\):
  - States are regular expressions involved in transitions
  - Transitions defined by \(\xrightarrow{a}\)
  - Final states defined by \(\checkmark\)
Defining ✓

- ✓ will defined inductively (i.e. recursively on the structure of REs)
- The definition is given via rules

\[
\begin{array}{c|c|c}
H_1 & \ldots & H_n \\
\hline
C
\end{array}
\]

- Idea: if the conditions $H_1 \ldots H_n$ ("hypotheses") above the line are true, the rule says the condition $C$ ("conclusion") below the line follows
- Hypotheses typically involve subexpressions in conclusion
- If $n=0$ (no hypotheses) then the conclusion is automatically true and is called an **axiom**
  - Often a “-” is written in the hypothesis list in this case
## Rules for $✓$

### Primitive REs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-$</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
</tbody>
</table>

### Composite REs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Left Result</th>
<th>Right Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$</td>
<td>$e_1 \varepsilon$</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>$(e_1 \mid e_2)$</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>$e_1 \cdot e_2$</td>
</tr>
<tr>
<td>$-$</td>
<td>$e^*$</td>
<td>$e^*$</td>
</tr>
</tbody>
</table>

$(\varepsilon) \quad (\varepsilon)$

$(\varepsilon | L) \quad (\varepsilon | R)$

$(\cdot) \quad (\cdot)$

$(\cdot)$

$(\cdot)$
Using the Rules for ✓

- For any e, e ✓ is defined to hold if and only if you can build a proof!
- Proofs start with axioms, involve applications of rules
- No proof: no ✓

Example Proof

… that ((ab)* | b(a*)) ✓

- (ab)* ✓

((ab)* | b(a*)) ✓
Another Example Proof: \((ab)^*(ba)^* \checkmark\)

\[
\begin{array}{c}
\_ \\
\_ \\
(ab)^* \checkmark \\
\_ \\
(ba)^* \checkmark \\
\_ \\
(ab)^*(ba)^* \checkmark 
\end{array}
\]
Goal-Directed Proof Construction

- Start with what you want to prove …
- … and work backwards!
  - Applicable rules for a given conclusion are determined by top-most operator in regular expression
  - Select an applicable rule
  - Use rule to generate subgoals
  - Repeat until you reach axioms everywhere (proof attempt succeeds) or you get stuck (proof attempt fails)
Example: \( ab^* \ | \ (ba^* \ | \ \varepsilon) \)
Example: An Unsuccessful Proof

\[
\begin{align*}
\ _ & \ _ \\
| & | \\
a & b^* \checkmark \\
\hline
ab^* \\
\hline
ab^* | (ba^* | \varepsilon)
\end{align*}
\]
Goal-Directed Disproof of ✓

- The goal-directed proof strategy gives a way of showing when e✓ fails
  - Try each proof rule that is applicable to e to generate subgoals, then try to prove subgoals
  - If one proof succeeds: e✓ holds
  - If all proofs fail: e✓ does not hold

- Examples where e✓ fails
  - a
  - (ab*) | (ba*)
Algorithm for Computing ✓

- The previous discussion suggests the following method for computing if $e ✓$ holds or not

```plaintext
algorithm check (e)
    when e is
        a, $\emptyset$: return false  // primitive cases where answer is false
        $\varepsilon$, $f^*$: return true  // cases where answer is true based on axioms
    $e_1 | e_2$:  // apply two rules for |
        if check($e_1$) then return true
        else return check($e_2$)
    $e_1 e_2$:  // concatenation rule
        if check($e_1$) then return check($e_2$)
        else return false
```

- Facts
  - $e ✓$ holds if and only if $\varepsilon \in [[e]]$.
  - check(e) returns true if and only if $e ✓$ holds.
Defining \( \rightarrow \)

- Like \( \checkmark \), defined inductively using rules
- Intuition: if \( e \rightarrow a \rightarrow e' \) then \( e \) can “process” symbol \( a \), then behave like \( e' \)
### Rules for --a-->

<table>
<thead>
<tr>
<th>Primitive REs</th>
<th>Composite REs</th>
</tr>
</thead>
</table>
| \( \epsilon \) | \( e_1 \rightarrow a \rightarrow e_1' \)  
| \( a \rightarrow a \rightarrow \epsilon \) | \( e_1 | e_2 \rightarrow a \rightarrow e_1' \)  
| | \( e_2 \rightarrow a \rightarrow e_2' \)  
| | \( e_1 | e_2 \rightarrow a \rightarrow e_2' \)  
| | \( e_1 \rightarrow a \rightarrow e_1' \)  
| | \( e_1 | e_2 \rightarrow a \rightarrow e_1' e_2 \)  
| | \( e_1 \rightarrow a \rightarrow e_1' \)  
| | \( e_1 e_2 \rightarrow a \rightarrow e_1' e_2 \)  
| | \( e_1 \rightarrow a \rightarrow e_1' \)  
| | \( e_1 \rightarrow a \rightarrow e_1' e_2 \)  
| | \( e \rightarrow a \rightarrow e' \)  
| | \( e^* \rightarrow a \rightarrow e'e^* \)  

\( \rightarrow \epsilon \)  
\( \rightarrow 1 \)  
\( \rightarrow 2 \)  
\( \rightarrow 1 \)  
\( \rightarrow 2 \)  
\( \rightarrow \)  
\( \rightarrow \)
Example: \(ab \rightarrow_a \epsilon b\)

\[
\begin{align*}
- \\
\hline
a \rightarrow_a \epsilon \\
\hline
ab \rightarrow_a \epsilon b
\end{align*}
\]

Note:
- For any RE \(e\), \([e] = [\epsilon e]\)
- We will often suppress these leading \(\epsilon\) in proofs
- In the above proof, we could write the conclusion thusly:
  - \(ab \rightarrow_a b\)
Another Example: $a^* --a-- \rightarrow a^*$

$\begin{align*}
\_ \\
\_ \\
a --a-- \rightarrow \epsilon \\
a^* --a-- \rightarrow a^*
\end{align*}$
One More Example: \((ab|ba)^* \rightarrow b \rightarrow a(ab|ba)^*\)

\[
\begin{align*}
\rightarrow \\
\rightarrow b \rightarrow \epsilon \\
\rightarrow ba \rightarrow a \\
\rightarrow ab|ba \rightarrow a \\
\rightarrow (ab|ba)^* \rightarrow b \rightarrow a((ab|ba)^*)
\end{align*}
\]
Final Example: \((a^*b^*)^* --b-- \rightarrow b^*(a^*b^*)^*\)

- \((a^*b^*)^* \rightarrow a^*\)
- \(b^* \rightarrow b^*\)
- \(a^*b^* \rightarrow b^*\)

Non-obvious fact: \([[(a^*b^*)^*]] = [[(a|b)^*]]\) (Why?)
Algorithm for Computing \( \rightarrow a \rightarrow \)

- Using goal-directed proof ideas from \( \checkmark \), we can define a routine for computing \( \{<a,e'> | e \rightarrow a \rightarrow e'\} \), given \( e \) as input.

```plaintext
algorithm trans (e)
  when e is
    \( \emptyset, \varepsilon \): return \{\}
    a: return \{<a, e'>\}
    e_1 \mid e_2:
      return (trans(e_1) \cup trans(e_2))
  e_1 e_2:
    if check(e_1) then s = trans(e_2) else s = \{\}
    return (\{<a,e'e_2> | \langle a,e' \rangle \in trans(e_1) \} \cup s)
  f*:
    return (\{<a,f'f*> | \langle a,f' \rangle \in trans(f)\})
```
Reducing REs to NFAs

- Given RE \( e \), \(<e>\) defined by
  - States: REs reachable from \( e \) via \(--b-->\) sequences
  - Start state: \( e \)
  - Transitions: \(--b-->\)
  - Final states: \( \checkmark \)
NFA for \((ab|ba)^*\)
NFA for \((ab|aba)^*\)
Reduction Algorithm

algorithm red (e)

\[ \delta = \Sigma = \{ \} \] // initialize transitions, alphabet
\[ Q = N = \{ e \} \] // initialize state set, aux. set N of states needing processing
\[ q_0 = e \] // create initial state

while N \neq \{ \} do // repeat until no states need processing
    choose f \in N // pick a state to process, remove it from N
    N = N - \{ f \}
    if check(f) then F = F \cup \{ f \} // if state should be final, add it to F
    foreach \langle a, f' \rangle in trans(f) do // compute transitions, update Q,N,\Sigma,\delta
        if f' \notin Q then // if transition target not seen before, add to Q, N
            Q = Q \cup \{ f' \}
            N = N \cup \{ f' \}
            \Sigma = \Sigma \cup \{ a \} // update alphabet
            \delta = \delta \cup \{ f, a, f' \} // update transition function
    return \langle \Sigma, Q, q_0, F, \delta \rangle
Reduction Complexity

- Given a regular expression $A$ of size $n$...
  
  Size = # of symbols + # of operations

- How many states does $<A>$ have?
  
  • $\leq n$
  
  • That’s pretty good!
Practice

- Draw NFAs for the following regular expressions and languages
  - \((0|1)^*110^*\)
  - \(101^*|111\)
  - all binary strings ending in 1 (odd numbers)
  - all alphabetic strings which come after “hello” in alphabetic order
  - \((ab^*c|d^*a|ab)d\)
Summary

- **Finite automata**
  - Deterministic (DFA)
  - Non-deterministic (NFA)

- **Questions**
  - How are DFAs and NFAs different?
  - When does an NFA accept a string?
  - How to convert regular expression to an NFA?
How NFAs Work

- When NFA processes a string
  - NFA may be in several possible states
    - Multiple transitions with same label

- Example
  - After processing “a”
    - NFA may be in states
      S1
      S2
Reducing NFA to DFA

- NFA may be reduced to DFA
  - By explicitly tracking the set of NFA states

- Intuition
  - Build DFA where
    - Each DFA state represents a set of NFA states

- Example
Reducing NFA to DFA (cont.)

- Reduction applied using the subset construction
  - DFA state is a subset of set of all NFA states

- Algorithm
  - Input
    - NFA ($\Sigma, Q, q_0, F_n, \delta$)
  - Output
    - DFA ($\Sigma, R, r_0, F_d, \delta$)
  - Using
    - $\text{move}(p, a)$
Calculating move(p,a)

- move(p,a)
  - Set of states reachable from p using exactly one transition on a
    - Set of states q such that \( \{p, a, q\} \in \delta \)
    - move(p,a) = \( \{q \mid <p, a, q> \in \delta\} \)
  
- Note move(p,a) may be empty \( \emptyset \)
  - If no transition from p with label a
move(a,p) : Example

Following NFA
- \( \Sigma = \{ a, b \} \)

Move
- \( \text{move}(S1, a) = \{ S2, S3 \} \)
- \( \text{move}(S1, b) = \emptyset \)
- \( \text{move}(S2, a) = \emptyset \)
- \( \text{move}(S2, b) = \{ S3 \} \)
- \( \text{move}(S3, a) = \emptyset \)
- \( \text{move}(S3, b) = \emptyset \)
NFA $\rightarrow$ DFA Reduction Algorithm

- **Input** NFA ($\Sigma$, $Q$, $q_0$, $F_n$, $\delta$), **Output** DFA ($\Sigma$, $R$, $r_0$, $F_d$, $\delta$)

- **Algorithm** ("subset construction")
  
  
  $r_0 = \{q_0\}; \ R = \{r_0\}$ \hspace{1cm} // DFA start state
  
  while $\exists$ an unmarked state $r \in R$ do
    
    mark $r$
    
    foreach $a \in \Sigma$
      
      $S = \{s \mid q \in r \& s \in \text{move}(q,a)\}$
      
      if $S \notin R$ then
        
        $R = \{S\} \cup R$
        
        $\delta = \delta \cup \{<r, a, S>\}$
        
    $F_d = \{r \mid \exists s \in r \text{ with } s \in F_n\}$
    
    state in $F_n$
NFA $\rightarrow$ DFA Example 1

- $r_0 = \{S1\}$
- $R = \{r_0\} = \{\{S1\}\}$
- $r \in R = \{S1\}$
- $move(\{S1\}, a) = \{S2, S3\}$
  - $R = R \cup \{\{S2, S3\}\} = \{\{S1\}, \{S2, S3\}\}$
  - $\delta = \delta \cup \{<\{S1\}, a, \{S2, S3\}>\}$
- $move(\{S1\}, b) = \emptyset$
  - $R = R \cup \emptyset$
    - $= \{\emptyset, \{S1\}, \{S2, S3\}\}$
  - $\delta = \delta \cup \{<\{S1\}, b, \emptyset>\}$
NFA → DFA Example 1 (cont.)

• \( R = \{ \{S1\}, \{S2, S3\}, \emptyset \} \)
• \( r \in R = \{S2, S3\} \)
• \( \text{move}\{S2, S3\}, a\} = \emptyset \)
• \( \text{move}\{S2, S3\}, b\} = \{S3\} \)
  - \( R = R \cup \{S3\} = \{ \{S1\}, \{S2, S3\} \}, \emptyset, \{S3\} \) 
  - \( \delta = \delta \cup \{(S2, S3), a, \emptyset \}, (S2, S3), b, \{S3\}\)
NFA → DFA Example 1 (cont.)

- \( R = \{ \{S1\}, \{S2,S3\}, \emptyset, \{S3\} \} \)
- \( r \in R = \emptyset \)
- \( \text{move}(\emptyset, a) = \emptyset \)
- \( \text{move}(\emptyset, b) = \emptyset \)
  - \( R = \{ \{S1\}, \{S2,S3\} \}, \emptyset, \{S3\} \} \)
  - \( \delta = \delta \cup \{< \emptyset, a, \emptyset >, <\emptyset, b, \emptyset>\} \)
NFA → DFA Example 1 (cont.)

- \( R = \{ \{ S1 \}, \{ S2, S3 \}, \emptyset, \{ S3 \} \} \)
- \( r \in R = \{ S3 \} \)
- \( \text{move}(\{ S3 \}, a) = \emptyset \)
- \( \text{move}(\{ S3 \}, b) = \emptyset \)
  - \( R = \{ \{ S1 \}, \{ S2, S3 \} \}, \emptyset, \{ S3 \} \}\)
  - \( \delta = \delta \cup \{ <S3, a, \emptyset>, <S3, b, \emptyset> \} \)
- \( F_d = \{ \{ S2, S3 \}, \{ S3 \} \} \)
  - Since \( S3 \in F_n \)
- Done!
NFA → DFA Example 2
Equivalence of DFAs and NFAs

- Any string from \{A\} to either \{D\} or \{CD\}
  - Represents a path from A to D in the original NFA
Equivalence of DFAs and NFAs (cont.)

- Can reduce any NFA to a DFA using subset alg.
- How many states in the DFA?
  - Each DFA state is a subset of the set of NFA states
  - Given NFA with \( n \) states, DFA may have \( 2^n \) states
    - Since a set with \( n \) items may have \( 2^n \) subsets
  - Corollary
    - Reducing a NFA with \( n \) states may be \( O(2^n) \)
Minimizing DFA

Result from CS theory
- Every regular language is recognizable by a minimum-state DFA that is unique up to state names

In other words
- For every DFA, there is a unique DFA with minimum number of states that accepts the same language
- Two minimum-state DFAs have same underlying shape
Minimizing DFA: Hopcroft Reduction

- **Intuition**
  - Look for states that are distinguishable from each other
  - End up in different accept / non-accept state with identical input

- **Algorithm**
  - Construct initial partition
    - Accepting & non-accepting states
  - Iteratively refine partitions (until partitions remain fixed)
    - Split a partition if members in partition have transitions to different partitions for same input
      - Two states \( x, y \) belong in same partition if and only if for all symbols in \( \Sigma \) they transition to the same partition
  - Update transitions & remove dead states
Splitting Partitions

- No need to split partition \{S,T,U,V\}
  - All transitions on \(a\) lead to identical partition \(P_2\)
  - Even though transitions on \(a\) lead to different states
Splitting Partitions (cont.)

- Need to split partition \{S,T,U\} into \{S,T\}, \{U\}
  - Transitions on a from S,T lead to partition P2
  - Transition on a from R lead to partition P3
Resplitting Partitions

- Need to reexamine partitions after splits
  - Initially no need to split partition \{S,T,U\}
  - After splitting partition \{X,Y\} into \{X\}, \{Y\}
  - Need to split partition \{S,T,U\} into \{S,T\}, \{U\}
DFA Minimization Algorithm (1)

- **Input** DFA \((\Sigma, Q, q_0, F_n, \delta)\), **Output** DFA \((\Sigma, R, r_0, F_d, \delta)\)

- **Algorithm**
  
  Let \(p_0 = F_n, p_1 = Q - F\)  // initial partitions = final, nonfinal states
  
  Let \(R = \{ p | p \in \{p_0,p_1\} \text{ and } p \neq \emptyset \}, P = \emptyset\) // add \(p\) to \(R\) if nonempty
  
  **While** \(P \neq R\) **do**  // while partitions changed on prev iteration
  
  Let \(P = R, R = \emptyset\)  // \(P =\) prev partitions, \(R =\) current partitions
  
  **For each** \(p \in P\)  // for each partition from previous iteration
  
  \(\{p_0,p_1\} = \text{split}(p,P)\)  // split partition, if necessary
  
  \(R = R \cup \{ p | p \in \{p_0,p_1\} \text{ and } p \neq \emptyset \}\) // add \(p\) to \(R\) if nonempty
  
  \(r_0 = p \in R\) where \(q_0 \in p\)  // partition w/ starting state
  
  \(F_d = \{ p | p \in R \text{ and exists } s \in p \text{ such that } s \in F_n \}\) // partitions w/ final states
  
  \(\delta(p,c) = q\) when \(\delta(s,c) = r\) where \(s \in p\) and \(r \in q\)  // add transitions
DFA Minimization Algorithm (2)

Algorithm for split(p,P)

Choose some r ∈ p, let q = p − {r}, m = {} // pick some state r in p
For each s ∈ q // for each state in p except for r
    For each c ∈ Σ // for each symbol in alphabet
        If δ(r,c) = q₀ and δ(s,c) = q₁ and // q’s = states reached for c
            there is no p₁ ∈ P such that q₀ ∈ p₁ and q₁ ∈ p₁ then
                m = m ∪ {s} // add s to m if q’s not in same partition

Return p − m, m // m = states that behave differently than r
               // m may be Ø if all states behave the same
               // p − m = states that behave the same as r
Minimizing DFA: Example 1

- DFA

- Initial partitions
  - Accept \{ R \} → P1
  - Reject \{ S, T \} → P2

- Split partition? → Not required, minimization done
  - move(S,a) = T → P2
  - move(S,b) = R → P1
  - move(T,a) = T → P2
  - move(T,b) = R → P1

After cleanup
Minimizing DFA: Example 2

- **DFA**

- **Initial partitions**
  - Accept \{ R \} → P1
  - Reject \{ S, T \} → P2

- **Split partition? → Not required, minimization done**
  - \text{move}(S,a) = T → P2
  - \text{move}(S,b) = R → P1
  - \text{move}(T,a) = S → P2
  - \text{move}(T,b) = R → P1
Minimizing DFA: Example 3

- **DFA**

- **Initial partitions**
  - Accept  \(\{ R \}\)  \(\rightarrow\)  \(P_1\)
  - Reject  \(\{ S, T \}\)  \(\rightarrow\)  \(P_2\)

- **Split partition?**  \(\rightarrow\) Yes, different partitions for \(B\)
  - \(move(S,a) = T \rightarrow P_2\)
  - \(move(S,b) = T \rightarrow P_2\)
  - \(move(T,a) = T \rightarrow P_2\)
  - \(move(T,b) = R \rightarrow P_1\)

- DFA already minimal
Complement of DFA

- Given a DFA accepting language $L$
  - How can we create a DFA accepting its complement?
  - Example DFA
    - $\Sigma = \{a,b\}$
Complement of DFA (cont.)

- **Algorithm**
  - Change every accepting state to a non-accepting state & every non-accepting state to an accepting state

- **Note this only works with DFAs**
  - Why not with NFAs?
Practice

Make the DFA which accepts the complement of the language accepted by the DFA below.
Reducing DFAs to REs

- General idea
  - Remove states one by one, labeling transitions with regular expressions
  - When two states are left (start and final), the transition label is the regular expression for the DFA
Relating REs to DFAs and NFAs

Why do we want to convert between these?
  • Can make it easier to express ideas
  • Can be easier to implement
Implementing DFAs

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

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");
        break;
    }
}

It's easy to build a program which mimics a DFA
Implementing DFAs (Alternative)

Alternatively, use generic table-driven DFA

given components \((\Sigma, Q, q_0, F, \delta)\) of a DFA:

let \(q = q_0\)

while (there exists another symbol \(s\) of the input string)

\(q := \delta(q, s);\)

if \(q \in F\) then

accept

else reject

• \(q\) is just an integer
• Represent \(\delta\) using arrays or hash tables
• Represent \(F\) as a set
Run Time of DFA

- How long for DFA to decide to accept/reject string $s$?
  - Assume we can compute $\delta(q, c)$ in constant time
  - Then time to process $s$ is $O(|s|)$
    - Can’t get much faster!
- Constructing DFA for RE $A$ may take $O(2^{|A|})$ time
  - But usually not the case in practice
- So there’s the initial overhead
  - But then processing 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 RE

- 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
Practice

- Convert to a DFA

- Convert to an NFA and then to a DFA
  - $(0|1)^*11|0^*$
  - Strings of alternating 0 and 1
  - $aba^*|(ba|b)$
Summary of Regular Expression Theory

- Finite automata
  - DFA, NFA

- Equivalence of RE, NFA, DFA
  - RE $\rightarrow$ NFA
    - ✓, --a--
  - NFA $\rightarrow$ DFA
    - subset construction

- DFA
  - Minimization, complementation
  - Implementation