CMSC 330: Organization of Programming Languages

Regular Expressions and Finite Automata
How do regular expressions work?

- What we’ve learned
  - What regular expressions are
  - What they can express, and cannot
  - Programming with them

- What’s next: how they work
  - A great computer science result
Languages and Machines

- Turing Machines
- Unrestricted grammars
- PDAs
- CFGs
- Regular Languages
- Reg exps
- FSMs
- Context-Free Languages
- Recursive Languages
- Recursively Enumerable Languages
A Few Questions About REs

- How are REs implemented?
  - Implementing a one-off RE is not so hard
    - How to do it in general?
A Few Questions About REs

- How are REs implemented?
  - Implementing a one-off RE is not so hard
    - How to do it in general?

- What are the basic components of REs?
  - Can implement some features in terms of others
    - E.g., we saw that e+ is the same as ee*
A Few Questions About REs

- How are REs implemented?
  - Implementing a one-off RE is not so hard
    - How to do it in general?

- What are the basic components of REs?
  - Can implement some features in terms of others
    - E.g., we saw that $e^+$ is the same as $ee^*$

- What does a regular expression represent?
  - Just a set of strings
    - This observation provides insight on how we go about our implementation
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)
    - $\emptyset \neq \{ \varepsilon \} \neq \varepsilon$
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: String concatenation

- String concatenation is indicated by juxtaposition
  
  \[ s_1 = \text{super} \quad s_1s_2 = \text{superhero} \]
  
  \[ s_2 = \text{hero} \]

- Sometimes also written \( s_1 \cdot s_2 \)
Definition: String concatenation

- String **concatenation** is indicated by juxtaposition

- For any string $s$, we have $se = es = s$
- 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 $L$ is a set of strings over an alphabet.
- Example: The set of phone numbers over the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 9, (, ), -\}$
Definition: Language

- A language \( L \) 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\}/\)
Definition: Language

- A **language** \( L \) 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?
    \[
    \text{/
    (\d{3,3})\ \d{3,3}-(\d{4,4})/}
    \]

- **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 $\Sigma = \{a, b, c\}$
  - $L = \{ s \mid s \in \Sigma^* \text{ and } |s| = 0 \} = \{\varepsilon\} \neq \emptyset$
Definition: Language (cont.)

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

- Example: The set of all valid Ruby programs
  - Is there a Ruby regular expression for this language?
Definition: Language (cont.)

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

- 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 using REs \{ { { … } } } \}
Definition: Language (cont.)

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

- 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 using REs \( \{ \{ \{ \ldots \} \} \} \)

- REs represent languages, but not all of them
  - The languages represented by regular expressions are called, appropriately, the regular languages
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 $\Sigma$.
Operations on Languages

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

Concatenation \( L_1 L_2 \) is defined as

\[ L_1 L_2 = \{ xy \mid x \in L_1 \text{ and } y \in L_2 \} \]
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
  - $L_1L_2 = \{ xy | x \in L_1 \text{ and } y \in L_2 \}$
- Union is defined as
  - $L_1 \cup L_2 = \{ x | x \in L_1 \text{ or } x \in L_2 \}$
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
  - $L_1L_2 = \{ xy \mid x \in L_1 \text{ and } y \in L_2 \}$

- **Union** is defined as
  - $L_1 \cup L_2 = \{ x \mid x \in L_1 \text{ or } x \in L_2 \}$

- **Kleene closure** is defined as
  - $L^* = \{ x \mid x = \varepsilon \text{ or } x \in L \text{ or } x \in LL \text{ or } x \in LLL \text{ or } \ldots \}$
Regular Expressions Denote Languages

- By applying operations on constants
  - Generates a set of strings (i.e., a language)
  - Examples
    - \( a \rightarrow \{ \text{“a”} \} \)
    - \( a|b \rightarrow \{ \text{“a”} \} \cup \{ \text{“b”} \} = \{ \text{“a”}, \text{“b”} \} \)
    - \( a^* \rightarrow \{ \varepsilon \} \cup \{ \text{“a”} \} \cup \{ \text{“aa”} \} \cup ... = \{ \varepsilon, \text{“a”}, \text{“aa”}, ... \} \)

- If \( s \in \text{language generated by a RE } r \), we say that \( r \) accepts, describes, or recognizes string \( s \)
Precedence

Order in which operators are applied

- 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., \( a(b|c), (ab)^*, (a|b)^* \)
Regular Languages

- The languages that can be described using regular expressions are the **regular languages** or **regular sets**
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 $\Sigma$
    - $\{a^n b^n \mid n > 0 \}$ ($a^n$ = sequence of $n$ a’s)
    - $a^* b^*$
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 $\Sigma$
    - $\{a^n b^n \mid n > 0 \}$ (where $a^n$ = 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
Implementing Regular Expressions

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

```
“String”
“String”
“String”
“String”
```

Yes
No
Finite Automata

\[ \Sigma = \{ 0, 1 \} \]

States:
- Start: \( S_0 \)
- Final: \( S_1 \)

Transitions:
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

\begin{itemize}
  \item \textbf{S\_match}: \((/ (0|1)* 0/)\)
  \item \textbf{Input}: 001011
  \item \textbf{Transition States}: S0 \rightarrow S1, S1 \rightarrow S0
\end{itemize}

\[
\begin{align*}
0 &\rightarrow 0 \\
1 &\rightarrow 1 \\
0 &\rightarrow 1 \\
1 &\rightarrow 0
\end{align*}
\]

\begin{itemize}
  \item \textbf{Accepted?}: Yes
\end{itemize}
Finite Automaton: Example 2

 Accepted?
What Language is This?
What Language is This?

- All strings over \( \{0, 1\} \) that end in 1
- What is a regular expression for this language?
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)
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>
</tbody>
</table>
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>accc</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
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>accc</td>
<td>S2</td>
<td>Y</td>
</tr>
</tbody>
</table>
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>bbbc</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
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>bbbc</td>
<td>S2</td>
<td>Y</td>
</tr>
</tbody>
</table>
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>aacbbb</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
Finite Automaton: Example 3

(a,b,c notation shorthand for three self loops)
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>ε</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
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>$\varepsilon$</td>
<td>S0</td>
<td>Y</td>
</tr>
</tbody>
</table>
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>acba</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
Finite Automaton: Example 3

(a, b, c notation shorthand for three self loops)
What language does this FA accept?
Finite Automaton: Example 3 (cont.)

What language does this FA accept?

a*b*c*
Finite Automaton: Example 3 (cont.)

What language does this FA 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.

\[ S0 \xrightarrow{1,3} S1 \xrightarrow{0,2} S0 \]

\[ S0 \xrightarrow{1,3} S1 \xrightarrow{0,2} S2 \xrightarrow{0,1,2,3} S0 \]

is short for
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.
Finite Automaton: Example 4

Diagram: Finite automaton with states S0, S1, S2, S3, S4, and transitions labeled with symbols a, b, c, and a, b, c.
Finite Automaton: Example 4

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

again, so DFAs are not unique
Finite Automaton: Example 5

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”**
Finite Automaton: Example 5

Language?
Finite Automaton: Example 5

Language?

- $(a|b)^*abb$
Examples

- Give a DFA that accepts binary numbers with an odd number of 1’s over \{0,1\}
Examples

- Construct a DFA to accept a string containing even number of zeroes and any number of ones. Alphabet = \{0,1\}

(assume 0 is not an even number)
Examples

- Construct a DFA to accept all strings containing odd number of zeroes and odd number of ones
  Alphabet = \{0, 1\}
Examples

- Construct a DFA to accept all strings DO NOT contain odd number of zeroes and odd number of ones Alphabet =\{0,1\}