Deterministic Finite Automaton:

- **DFA**: a finite-state machine that *accepts* and *rejects* finite strings of symbols and produces a unique computation of the automaton for each input string.
  - *Deterministic*: the uniqueness of the computation
    - For each pair of states and possible input chars, there is a unique next state
  - *Start state*: where computations begin
  - *Accept states*: define when a computation is successful
- Represented by *state diagrams*

Formal Definition: a deterministic finite automaton $M$ is a 5-tuple, $(S, A, \partial, s_0, F)$ consisting of
- A finite set of states ($S$)
- A finite set of input symbols (the alphabet $A$)
- A transition function ($\partial : S \times A \rightarrow S$)
- An initial or start state ($s_0$)
- A set of accept states ($F \subseteq Q$)

The automaton $M$ accepts a string $w = a_1a_2…a_n$ if a sequence of states $r_0r_1…r_n$ exists in $A$ with the following conditions:

1) $r_0 = s_0$ (the machine starts in the start state $s_0$)
2) $r_{i+1} = \partial(r_i, a_{i+1})$ for $i = 0, …, n-1$ (given each character of string $w$, the machine will transition from state to state according to the transition function $\partial$)
3) $r_n \in F$ (the machine accepts $w$ if the last input of $w$ causes the machine to halt in one of the accepting states)

Regular Expressions

- Finite automata are used to *recognize* patterns of strings; regular expressions are used to *generate* patterns of strings

Operands can be:
- *Characters* from the alphabet over which the regular expression is defined
- *Variables* whose values are any pattern defined by a regular expression
- *Epsilon* which denotes the empty string containing no characters
- *Null* which denotes the empty set of strings

Operators include:
- Union: if $R_1$ and $R_2$ are regular expressions, the $R_1 + R_2$ is also a regular expression
- Concatenation: if $R_1$ and $R_2$ are regular expressions, then $R_1 . R_2$ is also a regular expression
- Kleene closure: if $R_1$ is a regular expression, then $R_1*$ is also a regular expression

Constructing a RE from a FA

Examples:

<table>
<thead>
<tr>
<th>Regular Expressions</th>
<th>Regular Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(0 + 10^*)$</td>
<td>$L = {0, 1, 10, 100, 1000, \ldots}$</td>
</tr>
<tr>
<td>$(0^<em>10^</em>)$</td>
<td>$L = {1, 01, 10, 010, 0010, \ldots}$</td>
</tr>
<tr>
<td>$(0 + \varepsilon)\ (1 + \varepsilon)$</td>
<td>$L = {\varepsilon, 0, 1, 01}$</td>
</tr>
<tr>
<td>$(a + b)^*abb$</td>
<td>$L = {abb, aabb, babb, aaabb, ababb, \ldots}$</td>
</tr>
<tr>
<td>$(11)^*$</td>
<td>$L = {a, 11, 111, 11111, \ldots}$</td>
</tr>
<tr>
<td>$(aa)*((bb)*b$</td>
<td>$L = {b, aab, aabbb, aabbb, aabbb, \ldots}$</td>
</tr>
</tbody>
</table>
A language is regular if it is denoted by a regular expression:

<table>
<thead>
<tr>
<th>Set</th>
<th>Regular?</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L = {0^m \mid m \geq 0} )</td>
<td>Yes</td>
<td>( 0^* )</td>
</tr>
<tr>
<td>( L = {0^n1000 \mid n \geq 10} )</td>
<td>Yes</td>
<td>( 0^{10}0^*1100 )</td>
</tr>
<tr>
<td>( L = {01^{n}001^m \mid n,m \geq 11} )</td>
<td>Yes</td>
<td>( 01^{11}1<em>001^{11}1^</em> )</td>
</tr>
<tr>
<td>( L = {0^n1^m \mid n \geq 1} )</td>
<td>No</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Knuth-Morris-Pratt Algorithm**
- Searches for occurrences of a word \( W \) within a main text string \( P \)
- See lecture notes for pseudocode

Efficiency of the Search Algorithm:
- \( O(n) \), where \( n \) is the length of \( s \)

**Failure Function**
- Allow the algorithm not to match any character of \( S \) more than once
- “Pre-search” the pattern and compile a list of all possible fallback positions
- \( f[i] \) is the length of the longest proper initial segment of \( p \) that is also a segment of the substring ending at \( p[i-1] \)

Example:

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p[i] )</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>( f[i] )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Efficiency of Failure Function:
- \( O(k) \), where \( k \) is the length of \( p \)