Recursively enumerable languages

- *Recursively enumerable* languages form the largest group of languages that we shall study.
- They are recognized by Turing machines.
- *ONE FORM* of Turing machine extends the PDA, by having 2 stacks instead of 1 stack (a 2-stack machine).

Turing machines - I

- The basic Turing Machine (TM) starts in an initial state at the left-hand end of an *one-way infinite* tape, which contains a string from $\Sigma^*$ at the left hand end, followed by blanks ($B$)

\[ a_i a_j a_k \ldots \ldots a_r B B B \]

\[ q_0 \]
At the beginning, there can be no blanks in the input string BUT the input can be empty.

The input must begin at the leftmost tape location.

Assume the first “D” is “R.” Notice that a tape symbol \( X_1 \) is written on the tape in place of the symbol \( a_1 \).

If the first “D” had been “L” the machine would have halted.

You have a great deal of flexibility to define transitions. The transition can also use tape symbols.

There is no need to change the control state either.

We can make this TM run for ever by adding the transitions:

\[(q_2, Y) \rightarrow (q_2, Y, R)\] for any \( Y \) including the blank \( B \).
However, some of the states of the control of the TM may be designated as final states.

If a TM reaches a final state, no matter where it is on the tape, it halts and accepts the input.

If the TM runs for ever, or if it halts in a non-final state, then it is considered to not accept the input.

We may begin using simulators for finite automata and for TM’s.

Later we shall use the PDA simulator.

They are written as Java applications.

They come in a graphical form and in text-only form, the graphical form is easier to use but requires a windowing environment.

You may use bingsuns (preferably from an X-capable workstation).

To use a PC, you need to download Java (if this is not practical, contact lander@binghamton.edu, 777-2309).


Find the DOWNLOADS page.

Download the SDK for Java 2.

There are separate sets of notes on how to obtain and use the simulators, which are linked from the course home page: http://bingweb.binghamton.edu/~lander/cs573.html.

We plan to do some exercises with automata, then work on some Turing machines.
**FINITE AUTOMATA**

deterministic automata
regular languages
regular expressions
non-deterministic automata
removal of non-determinism
automata from regular expressions

---

**transitions**

- The most important part of the definition of the automaton is the set of transitions: \((q_u, a_i) \rightarrow q_v\)
- We define a transition function: \(\delta(q, a) = q'\)
- This function should be defined for all the states and inputs that participate in transitions
- Hence we have:

**definition of a DFA**

- A deterministic finite automaton (DFA) is a tuple \((Q, \Sigma, q_0, \delta, F)\), where
  - \(Q\) is a finite set of states \(\{q_0, q_1, ..., q_m\}\)
  - \(\Sigma\) is a finite alphabet \(\{a_1, a_2, ..., a_n\}\)
  - \(q_0\) is the initial state; the automaton always begins in the initial state
  - \(\delta : Q \times \Sigma \rightarrow Q\) is a partial function (it may not be defined for every state/input combination)
  - \(F \subseteq Q\) is a subset of final states

---

**FINITE AUTOMATA**

- We are given an alphabet:
  \(\Sigma = \{a_1, a_2, ..., a_n\}\)
- We need the definition of a finite automaton:
  \[
  \begin{array}{ccccccc}
  a_i & a_j & a_k & . & . & . & a_r \\
  \end{array}
  \]
  \((q_u, a_i) \rightarrow q_v\)

---

CLASS 02- 26 © L. Lander, 2000 CS 573
If $\delta$ is defined for all pairs in $Q \times \Sigma$, we say that $\delta$ is a \textit{total} function and we say that we have a \textit{full} (or \textit{complete}) automaton.

The function $\delta$ can be described in a table, called the \textit{transition table}:

<table>
<thead>
<tr>
<th></th>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$\ldots$</th>
<th>$a_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>entries are</td>
<td>the values</td>
<td>of $\delta(q_i, a_j)$</td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td>[ ... ]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ ... ]</td>
<td>[ ... ]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example transition table

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>-</td>
<td>$q_0$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_1$</td>
<td>-</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_1$</td>
<td>$q_3$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_2$</td>
<td>$q_1$</td>
<td>$q_3$</td>
</tr>
</tbody>
</table>

$\text{F = \{ q_0, q_3 \}}$

Here $\delta$ is not defined for $(q_0, a)$ and $(q_1, c)$

We can also represent a transition table using a \textit{transition diagram}:

Strings accepted by the example DFA

This automaton accepts the following strings:

- $\varepsilon$
- $b^k$ for all $k \geq 1$
- $b^k c^{2l} b^m$ for all, $k, l > 0 \ m \geq 0$
- $b^k c^{2l} b^m c^{2n+1} b c^p a b$
  for all $k, l, m, n, p \geq 0$
- and many more!
Our simulation software expects an input file that is a stylized version of the transition table plus information on the states and the alphabet.

Our simulation software creates a transition diagram and shows the state transitions that occur as the input string is processed.

Follow “simulators” link on the course home page.

We can extend the function $\delta$ from a function defined on $Q \times \Sigma$ i.e. for pairs $(q, a)$, to a function $\delta^*$ defined from $Q \times \Sigma^*$ to $Q$, i.e. for pairs $(q, w)$, where $w$ is a string:

- $\delta^*(q, \varepsilon) = q$, for state $q$
- and if $w = w_1a$,
  \[ \delta^*(q, w) = \delta^*(q, w_1a) = \delta(\delta^*(q, w_1), a) \]

This is an inductive definition.

For a single element of $Q$ and a string $a_1a_2a_3a_4$ we have:

\[ \delta^*(q, a_1a_2a_3a_4) = \{\delta(\delta(\delta(q, a_1), a_2), a_3), a_4)\} \]

Notice (for later) how a string of length 4 connects 5 states.

Special note about partial functions:

If $\delta$ is a partial function, then $\delta^*$ is also a partial function; the $\delta^*$ map may not be defined for all states $q$ and all strings $w$.

Notice (for later) how a string of length 4 connects 5 states.
Next we can extend the function \( \delta^* \) from a function defined on \( Q \times \Sigma^* \), i.e. for pairs \((q, w)\), to a function defined from \( P(Q) \times \Sigma^* \) to \( P(Q) \), where \( P(Q) \) is the power set of \( Q \), i.e. the set of subsets of \( Q \).

We shall use the notation \( \delta^* \) but the textbook writes \( \delta \).

Hence \( \delta^* : P(Q) \times \Sigma^* \to P(Q) \)

Note about \( \delta^* : P(Q) \times \Sigma^* \to P(Q) \):

We do not really need the extension to subsets in order to define the language of the DFA.

However, this extension does unify total and partial functions.

Whenever \( \delta^*(q, w) \) is not defined because \( \delta \) is not defined, we can say \( \delta^*([q], w) = \emptyset \).

In this way, \( \delta^* : P(Q) \times \Sigma^* \to P(Q) \) is a total function.

\[ \delta^*(S, w) = \bigcup_{s \in S} \{ \delta^*(s, w) \} \]

For example, if all are defined:

\[ \delta^*([q_1, q_2, q_3], w) = \{ \delta^*(q_1, w), \delta^*(q_2, w), \delta^*(q_3, w) \} \]

Thus, in this case:

\[ \delta^*([q_1, q_2, q_3], a_1a_2a_3a_4) = \{ \delta^*(q_1, a_1a_2a_3a_4), \delta^*(q_2, a_1a_2a_3a_4), \delta^*(q_3, a_1a_2a_3a_4) \} \]

and

\[ \delta^*([q_1, q_2, q_3], a_1a_2a_3a_4) = \{ \delta(\delta(\delta(q_1, a_1), a_2), a_3), a_4), \delta(\delta(\delta(q_2, a_1), a_2), a_3), a_4), \delta(\delta(\delta(q_3, a_1), a_2), a_3), a_4) \} \]

With all this notation, we can define acceptance. For a DFA \( M = (Q, \Sigma, q_0, \delta, F) \), we shall write \( L(M) \) for the language recognized by \( M \).

\[ L(M) = \{ w \in \Sigma^* : \delta^*(q_0, w) \in F \} \]

(this includes \( \delta^*(q_0, w) \) undefined), or
precise description of language recognized-VIII

- \( L(M) = \{ w \in \Sigma^* : \delta^*([q_0], w) \cap F \neq \emptyset \} \), that is, a string \( w \) is in the language recognized by \( M \) if:
  - a sequence of transitions labeled by the symbols in \( w \), form a path from state \( q_0 \) to a final state (i.e. a state in \( F \))

![Diagram of a DFA with transitions labeled by symbols and states]

Example

- The textbook shows a DFA that recognizes those strings over \( \{0, 1\} \) that contain an even number of 0’s \textit{and} an even number of 1’s:

![Diagram of a DFA with transitions labeled by symbols and states]

Regular languages

- A \textit{regular language} is any language that is recognized (or accepted) by a DFA
- Note that there may be several different DFA’s that recognize the same regular language
- Later we shall show how to find the DFA with the \textit{fewest states} for a particular regular language

Regular expressions

- Regular languages have an advantage over all other languages: there is a simple notation to describe the language
- The notation is called \textit{regular expressions}
- A regular expression is a formula that describes \textit{all} the strings in the language
Basic regular expressions

- Suppose \( \Sigma = \{ a_1, a_2, \ldots, a_n \} \) is an alphabet.
- Then the following are regular expressions:
  - \( \emptyset \) which is the empty language
  - \( \varepsilon \), which here denotes the language \( \{ \varepsilon \} \)
  - \( a_k \), which here denotes the language \( \{ a_k \} \), for \( k = 1, 2, \ldots, n \)

Operations to form compound expressions-I

- We use the notation of concatenation to form regular expressions:
  - Suppose \( r_1 \) is a regular expression denoting the language \( L_1 \) and \( r_2 \) is a regular expression denoting the language \( L_2 \)
  - Then \( r_1 r_2 \) is a new regular expression denoting \( L_1 L_2 \). Recall that \( L_1 L_2 = \{ w_1 w_2 : \text{for all } w_1 \in L_1, w_2 \in L_2 \} \)

Operations to form compound expressions-II

- We use the notation of summation to form regular expressions:
  - Suppose \( r_1 \) is a regular expression denoting the language \( L_1 \) and \( r_2 \) is a regular expression denoting the language \( L_2 \)
  - Then \( r_1 + r_2 \) is a new regular expression denoting \( L_1 \cup L_2 \).
  - Recall: \( L_1 \cup L_2 = \{ w : w \in L_1 \text{ or } w \in L_2 \} \)

Operations to form compound expressions-III

- We use the notation of Kleene closure to form regular expressions:
  - Suppose \( r \) is a regular expression denoting the language \( L \)
  - Then \( r^* \) is a new regular expression denoting \( L^* \). Recall that \( L^* = \{ w_1 w_2 \ldots w_k : k \geq 0, w_i \in L \text{ for all } i \} \)
    (when \( k = 0, w_1 w_2 \ldots w_k \) is \( \varepsilon \) )
The positive closure

- There is an extra notation (the positive closure) which is not really necessary but it is convenient:
  \[ L^+ = \{ w_1 w_2 \ldots w_k : k > 0, \ w_i \in L \text{ for all } i \} \]
- Corresponding to \( L^+ \), there is a regular expression \( r^+ \) (where \( r \) is the regular expression for \( L \))
- Note: \( r^+ = r \ r^* \) (precedence: \* and + before concatenation and concatenation before sum)

Corresponding to \( L^+ \), there is a regular expression \( r^+ \) (where \( r \) is the regular expression for \( L \))

Note: \( r^+ = r \ r^* \) (precedence: \* and + before concatenation and concatenation before sum)

Another simplifying notation & examples

- If \( r \) is the regular expression for \( L \), then \( r^k \) is the regular expression for \( L^k \), where \( k \) is some fixed positive integer and
  \[ L^k = \{ w_1 w_2 \ldots w_k : w_i \in L \text{ for all } i \} \]

Examples:
- \( 0 \) (meaning \{0\})
- \( 0 + 1^+ \)
- \( (0 + 1)^* \)
- \( (0^* + 10^*1)(0^+1^3 + 1^20^5) \)

The language of a regular expression

- As much as possible, we use a regular expression \( r \) as if it were a language ... but if we have to distinguish between them (e.g. in the definitions above) then \( L(r) \) is used for the language denoted by \( r \)

Examples
- \( L(0) = \{0\} \)
- \( L(0 + 1^+) = \{ w : w = 0 \} \) or \( w = 1^k, \ k > 0 \)
Our main goal of this section:

- The main goal of this section is to prove that regular expressions can denote any (and all) regular languages.
- If we have a regular expression $r$, we have to show how to find the DFA whose language is $L(r)$ and
- If we have a DFA $M$, we have to find a regular expression for $L(M)$.

Non-determinism:

- To achieve our goal, we need the concept of Non-deterministic Finite Automaton with $\varepsilon$-moves (NFA$\varepsilon$).
- An NFA$\varepsilon$ is a tuple $M = (Q, \Sigma, q_0, \delta, F)$, where $\delta$ is modified to be a function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(Q)$.
- There are two aspects to non-determinism:
  - one-to-many transitions
  - spontaneous transitions

one-to-many transitions:

- For an NFA$\varepsilon$, $\delta(q, a)$ is an element of $P(Q)$, i.e. a subset of $Q$, for example $\delta(q, a) = \{q_1, q_2, q_3, q_4\}$
- If we are in state $q$ and the input is $a$, we can choose to move to any one of $q_1, q_2, q_3$ or $q_4$.

$\delta(q, a)$ can be empty:

- One of the elements of $P(Q)$ is the empty set $\emptyset$ (here $\emptyset$ is the empty set of states, not the empty set of strings).
- If $\delta(q, a) = \emptyset$ and we are in state $q$, with input symbol $a$, then there is no transition possible, the NFA$\varepsilon$ halts and cannot accept the input string.
spontaneous (empty or ε-) transitions

- In an NFAε, we also define δ(q, ε)
- The most normal thing would be δ(q, ε) = {q}, i.e. if we are in state q and there is no input, then we do not move!
- However, it is very useful to allow a state change without input.
- If δ(q, ε) = {q₁, q₂, q₃} then without input we can move from q to q₁, q₂ or q₃

where do we go from state q?

- Of course, if there is no input, we can also choose to stay at state q
- Thus, if we reach state q while processing an input string, then the next transition could be from any of the states q, q₁, q₂ or q₃

Example-I

- This NFAε accepts the input abb, for example, q₀ → q₁ → q₂ → q₀ → q₁ → q₂ → q₃

Example-II

- The NFAε also accepts the input abb by q₀ → q₁ → q₂ → q₀ → q₁ → q₂ → q₃
The input $abb$ can also reach the dead state $q_1$ and halt $q_0 \xrightarrow{\varepsilon} q_2 \xrightarrow{a} q_1$.