Finite-State Machines

State Machines in General

- "Mathematical" (as opposed to mechanical) machines
  - Turing Machines (potentially infinite-state)
  - Finite-state machines
  - Other categories (cf. CS 142, Theory of Computation)

What are Finite-State Machines?

- A primitive computational model, related to many facets of computing:
  - A severely-restricted type of Turing machine
  - Model of switching circuits with memory
  - The building blocks for most real-life computers
  - Parsing for a limited family of languages (called "regular" languages or finite-state languages)
  - Regular expressions: used for textual pattern matching
  - Real-time software applications

FSM as a “crippled” TM

Can move in one direction only; Symbol written is never again changed.

FSM with separate I/O

FSM as head-stationary, tape-moving, device
**FSM as Sequence Transducer**

- **Input sequence**: $x_1, x_2, x_3, \ldots$
- **Output sequence**: $y_1, y_2, y_3, \ldots$
- **Finite set of internal states**

**FSM as a Sequence Classifier**

- **Classification**: Output associated with current state
- **Input sequence**: $x_1, x_2, x_3, \ldots$
- **Finite set of internal states**

**Edge-Detector Example**

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td>0111</td>
<td>0100</td>
</tr>
<tr>
<td>01110</td>
<td>01001</td>
</tr>
</tbody>
</table>

**Transducer: An Edge Detector**

- The state is recording the previous input.
- Whenever the current input differs, an edge has been detected.
- The first input is not considered an edge.

**Transducer Transcribed to a rex Program**

```plaintext
edgeDetector(input) = f(input);
f(0) => [0];
f(0 [ remainder]) => [0 | g(remainder)];
f(1 [ remainder]) => [0 | h(remainder)];
g(0) => [0];
g(0 [ remainder]) => [0 | g(remainder)];
g(1 [ remainder]) => [1 | h(remainder)];
h(0) => [1];
h(0 [ remainder]) => [1 | g(remainder)];
h(1 [ remainder]) => [0 | h(remainder)];
test(edgeDetector([0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1],
                [0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0]));
```

**A related Classifier:**

- **How many edges were there so far (0, 1, more)**

<table>
<thead>
<tr>
<th>input</th>
<th>output (class)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>more</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>1</td>
</tr>
<tr>
<td>k</td>
<td>1</td>
</tr>
<tr>
<td>l</td>
<td>0</td>
</tr>
<tr>
<td>m</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
</tr>
<tr>
<td>o</td>
<td>more</td>
</tr>
</tbody>
</table>
Classifier Transcribed to a rex Program

edgeClassifier(input) = a(input);
a([]) => 0;
a([0 | remainder]) => b(remainder);
a([1 | remainder]) => c(remainder);
b([]) => 0;
b([0 | remainder]) => b(remainder);
b([1 | remainder]) => d(remainder);
c([]) => 0;
c([0 | remainder]) => e(remainder);
c([1 | remainder]) => c(remainder);
d([]) => 1;
d([0 | remainder]) => f(remainder);
d([1 | remainder]) => d(remainder);
e([]) => 1;
e([0 | remainder]) => e(remainder);
e([1 | remainder]) => f(remainder);
f([]) => "more";
f([0 | remainder]) => f(remainder);
f([1 | remainder]) => f(remainder);

Finite-State Acceptors (FSAs)

- Acceptors are Classifiers with only 2 classes: accepted and rejected.
- Rejection is not necessarily final: additional input can convert to accepted.
- Typically acceptors show accepting states with double outline, rejecting states have single outline.

A related Acceptor:
Accept sequences with exactly one edge

Acceptor Transcribed to a rex Program

Conversions

- Every classifier can be represented as a "gang" of acceptors, by encoding the class.
- Every transducer can be represented as an "equivalent" classifier.
- Therefore, studying acceptors, the simplest model, yields insight for all finite-state machines.

What can an Acceptor Do?

- The Pepsi Machine near B101.
- Coins of 5, 10, and 25 cents can be entered (referred to by input symbols n, d, q, respectively).
- Accepts when a total of 40 cents (or more ☺) has been entered.
The specification is "partial" because we have not said what happens when q is input to states 20, 25, 30, 35, etc.

Types of Acceptor Problems

- **Analysis**: What does a given acceptor do (e.g., in English)?
- **Synthesis**: Construct an acceptor that performs according to a specification.
- **Realization**: Show the logic for a switching circuit that realizes an acceptor.
- **Abstraction** ("reverse engineering"): From a switching circuit, give the corresponding acceptor.

Languages for FSAs

- A convenient way to characterize an acceptor is by its language, the set of all input sequences it accepts.
- Typically the language will be infinite, although there are also cases of finite languages.

Languages for FSAs

- The set of all strings over \{0, 1\} such that every 0, if followed by any symbol, is followed by a 1.
- The set of all strings over \{0, 1\} such that the number of symbols is a multiple of 4.
- The set of all binary numerals that, m.s.b. first, are multiples of 3: \{0, 11, 110, 1001, 1100, 1111, \ldots\} (corresponding to 0, 3, 6, 9, 12, 15, \ldots\)

Language Examples

- The set of all strings over \{0, 1\} such that every 0, if followed by any symbol, is followed by a 1.
- The set of all strings over \{0, 1\} such that the number of symbols is a multiple of 4.
Language Examples

- The set of all binary numerals that, m.s.b. first, are multiples of 3:
  (0, 11, 110, 1001, 1100, 1111, ...)
  (corresponding to 0, 3, 6, 9, 12, 15, ...)

Correctness of the Multiples-of-3 Example

Let $n$ be the numeral input so far. For every $n$, there is a $k$ and an $r < 3$, such that $n = 3k+r$ ($r = n \% 3$). States, other than the starting state, are identified with $r$.

Inputting a 0 takes $n$ to $2n$, and inputting a 1 takes $n$ to $2n+1$.
So inputting a 0 takes $3k+r$ to $6k+2r$, while inputting a 1 takes $3k+r$ to $6k+2r+1$.

Acceptor as Entered in JFLAP

(Applet-based tool by Susan Rodger:
http://www.cs.duke.edu/~rodger/tools/jflap)

Characterization of Finite-State Machines by "Regular Expressions"

- Regular expressions are a machine-independent way of specifying a language.
- They are often used in textual pattern-matching applications.
- They are closely related to grammars, but the form of recursion is limited to "iterative" forms only.

Regular Expressions

- Kleene was also a principal developer of the field of recursion (computability) theory.

About Mr. Kleene

Kleene pronounced his last name /klay'nee/.
/klee'nee/ and /kleen/ are extremely common mispronunciations. His first name is /steev'n/, not /stef'n/.

His son, Ken Kleene <kenneth.kleene@umb.edu>, wrote:
"As far as I am aware this pronunciation is incorrect in all known languages. I believe that this novel pronunciation was invented by my father."
Regular Expressions Defined

- A regular expression (RE) is always defined with respect to a finite alphabet of symbols, \( \Sigma \). The definition is inductive:
  - Basis:
    - Any symbol in \( \Sigma \) is an RE.
    - The special symbol \( \lambda \) is an RE (often \( \epsilon \) is used instead of \( \lambda \)).
    - The special symbol \( \emptyset \) is an RE.
  - Induction step: If \( R \) and \( S \) are RE's, then so are:
    - \( RS \)
    - \( R | S \)
    - \( R^* \)

Regular Expression Examples

- Take \( \Sigma = \{0, 1\} \).
  - Basis:
    - Any symbol in \( \Sigma \) is an RE: \( 0 \) \( 1 \)
    - The special symbol \( \lambda \) is an RE: \( \lambda \)
    - The special symbol \( \emptyset \) is an RE: \( \emptyset \)
  - Induction step: If \( R \) and \( S \) are RE's, then so are:
    - \( RS: 00 \) \( 01 \) \( 001 \) \( 1010 \) \( 1(00 | 11)^*0 \)
    - \( R | S: 00 \) \( 11 \) \( 0 \) \( 1 \) \( \lambda \)
    - \( R^*: 0^* \) \( 01^*0 \) \( (00 | 11)^* \)

Meaning of Regular Expressions (1)

- Each regular expression \( R \) denotes a language (set of strings) \( L(R) \) over its alphabet:
  - Basis:
    - A symbol \( \sigma \) in \( \Sigma \) denotes the language of one string of one letter: \( L(\sigma) = \{\sigma\} \).
    - The special symbol \( \lambda \) denotes the empty string (no letters): \( L(\lambda) = \{\lambda\} \).
    - The special symbol \( \emptyset \) denotes the empty set (no strings): \( L(\emptyset) = \emptyset \)

Meaning of Regular Expressions (2)

- Induction step: Suppose \( R \) and \( S \) are regular expressions and \( L(R) \) and \( L(S) \) have been defined. Then (concatenation of two strings)
  - \( L(RS) = \{xy \mid x \in L(R) \text{ and } y \in L(S)\} \)
  - \( L(R | S) = L(R) \cup L(S) \)
  - \( L(R^*) = \{\lambda\} \cup L(R) \cup L^2(R) \cup L^3(R) \ldots \)
    where \( L^k(R) \) means the language formed by concatenating \( k \) strings, each one from \( L(R) \).

Similarity to Grammar Rules

- Suppose that we have a grammar in which auxiliary symbol \( r \) derives the strings in \( L(R) \) and auxiliary symbol \( s \) derives the strings in \( L(S) \). Then:
  - Adding \( t \rightarrow r \ s \) would make \( t \) derive the strings in \( L(RS) \).
  - Adding \( t \rightarrow r \ | \ s \) would make \( t \) derive the strings in \( L(R | S) \).
  - Adding \( t \rightarrow r \) would make \( t \) derive the strings \( L(R^*) \).

Note on Precedence in Regular Expressions

- It is common to omit parentheses.
- The binding order is:
  - \( * \) binds most tightly
  - \( | \) is next
  - \( \) binds most weakly
Examples of RE’s, with Meanings

- \(0101\)
The set of one string “0101”.
- \(0101 | 1010\)
The set of two strings, “0101” and “1010”.
- \(10101 | 10100\)
The set of two strings, “101010” and “101000”.
- \(01^*0\)
The set of strings that begin and end with 0 and contain a continuous run of 1’s (of length 0 or more).

Examples of RE’s, with Meanings

- \(0*1^*\)
The set of strings in which no 1 is followed by a 0.
- \(0^*1^*0^*1^*\)
The set of strings in which at most one 1 is followed by a 0.
- \(0^*(100^*)^*\)
The set of strings in which every one is followed by a 0.

Try These

- \((0*10*1)^*0^*\)
- \(((0 | 1)(0 | 1))^*\)
- \(0^*10^* | 1^*01^*\)
- \((0^*1)^*\)

Give Regular Expressions (over alphabet \(\{0, 1\}\)) for

- The set of strings with at most two 0’s
- The set of strings with more than two 0’s
- The set of strings in which 0’s and 1’s strictly alternate

Kleene’s Remarkable Result

- The languages accepted by finite-state acceptors and the languages denoted by regular expressions are the same thing.

In other words:

- Part I: The language accepted by any finite-state acceptor can be expressed as a regular expression.
- Part II: For every regular expression, there is a finite state acceptor that accepts the language denoted by the expression.
Proof of Part I

- The language accepted by any finite-state acceptor can be expressed as a regular expression.

Idea of Part I:
(analogous to Gaussian elimination)

- Eliminate states, replacing paths with regular expressions that represent those paths.

Original:

\[ q_0 \xrightarrow{0} 0 \xrightarrow{1} 1 \xrightarrow{0} 2 \xrightarrow{1} q_1 \]

Replacement:

\[ q_0 \xrightarrow{0} 0 \xrightarrow{1} 0 \xrightarrow{1} 1 \xrightarrow{0} 0 \]

Replacement (2):

\[ q_0 \xrightarrow{0} 0 \xrightarrow{1} 1 \xrightarrow{0} 0 \]

Final: \( (0 | 1(01^*0)^*1)(0 | 1(01^*0)^*1)^* \)

Sanity-Preserving Technique for Elimination

- This helps deal with cases in which initial and accepting states overlap or are involved in loops.
- Introduce new states for initial and accepting.
- Connect new initial state to original initial state by \( \lambda \) transition.
- Connect all accepting states to a single new accepting state via \( \lambda \) transitions.
- The original initial and accepting states are now ordinary states.
- Eliminate, in succession, all nodes other than the new initial and accepting states.
- The regular expression for the acceptor is the one connecting initial to accepting.
Proof of Part II

For every regular expression $R$, there is an FSA that accepts $L(R)$, the language denoted by $R$.

Non-Deterministic FSAs

The easiest way to prove part II is to appeal to the idea of a non-deterministic finite-state acceptor (NFSA):

- **Part IIa**: For every regular expression $R$, there is an NFSA that accepts $L(R)$.

- **Part IIb**: For every NFSA $N$ there is a (deterministic) finite-state acceptor that accepts $L(N)$.

Non-Deterministic FSAs

A non-deterministic finite-state acceptor (NFSA) is a finite-state acceptor with free-choice of transitions:

- A given state may have more than one transition leaving with the same symbol, or

- A state may be left spontaneously via a $\lambda$ transition.
Non-Deterministic FSAs

- A given state may have more than one (or even no) transition leaving with a given symbol.

The machine gets to choose which one to take.

Non-Deterministic FSAs

- A state may be left spontaneously via a $\lambda$ transition.

The machine can leave state a spontaneously and go to b, or it can absorb input 0 and go to c.

Acceptance Notion for NFSAs

- An NFSA accepts an input sequence iff there is some path from some initial state (an NFSA can have more than one) to some accepting state.

This machine accepts $\{01\}$.

Proof of Part IIa

- Part IIa: For every regular expression R, there is an NFSA that accepts $L(R)$.

- This proof is by structural induction on the formation of regular expressions.

- Basis:
  - Any symbol in $\Sigma$ is an RE.
  - The special symbol $\lambda$ is an RE.
  - The special symbol $\emptyset$ is an RE.

- Induction step: If $R$ and $S$ are RE's, then so are:
  - $RS$
  - $R | S$
  - $R^*$

Proof of Part IIa (1)

- We construct an accepting NFSA for each RE introduced in the definition.

- Basis:
  - Any symbol in $\Sigma$ is an RE.

- Induction step: If $R$ and $S$ are RE's, then so are:
  - $RS$
  - $R | S$
  - $R^*$

Proof of Part IIa (2)

- We construct an accepting NFSA for each RE introduced in the definition.

- Induction step: If $R$ and $S$ are RE's, then so are:
  - $RS$
  - $R | S$
  - $R^*$

- We assume that NFSAs exist for $R$ and $S$, and construct them for those three cases:
  - $RS$
  - $R | S$
  - $R^*$

NFSA for $R$ and $S$
Proof of Part IIa (3)

- We assume that NFSA's exist for R and S, and construct them for these three cases:
  - R | S
  - NFSA for R
  - NFSA for S

Proof of Part IIa (4)

- We assume that NFSA's exist for R and S, and construct them for these three cases:
  - R
  - NFSA for R
  - NFSA for R

Proof of Part IIb (1)

- For every NFSA N there is a (deterministic) FSA that accepts L(N).
- The idea is that for an NFSA N we can construct a FSA D accepting L(N) by "simulating in parallel" all the choices the NFSA could make. An input sequence is accepted iff any of those choices led to acceptance in N.

Proof of Part IIb (2)

- To simulate an NFSA, we construct D to have as its states subsets of the states of N. The transitions of D emulate all transitions for N "in parallel". For example, suppose that {0, 1, 2} is the alphabet.

Definition of f, the state transition for D:

\[ f(S, \sigma) = \{ q' | (\exists q \in S) q \rightarrow q' \text{ in } N \} \]

Proof of Part IIb (3)

- An accepting state in D is any that has an accepting state of N as a member.

Proof of Part IIb (4)

- The initial state in D is the set of all states reachable from some initial state in N by the empty sequence (i.e., including \( \lambda \) transitions).
The Complete Construction for a Simple Example

```
N: 0 0 a b c d e 1 1
D: (a, c) (b, c) (d, b) (d, e) (b, c) (e, d) 0 0 1 1
```

This Completes the Proof of Kleene's Theorem

- We now know that the following are equivalent:
  - L is a language denoted by some regular expression.
  - L is a language accepted by an NFSA.
  - L is a language accepted by an FSA.

Example:
Regular Expression to FSA (1)

- Construct an FSA for the RE $01^*0 \mid 0^*10^*$
- By inspection we can do NFSA's for 01*0 and 0*10*:

Example:
Regular Expression to FSA (2)

Below, any unspecified transitions go to {}.

Regular Expressions in Everyday Practice:
- e.g. Unix `grep`
- used for searching for lines containing matching strings in files
- Do `man regexp` to get this information on turing:
  - Most single characters match themselves (exceptions: . * [ ] ^ $)
  - matches any character, except new-line
  - ^ matches beginning of line (must occur first)
  - $ matches end of line (must occur last)
- Examples:
  - `grep 'elle' filename`
  - `grep 'll.*ll' filename` (is like Σ*)
  - `grep 'll$' filename`
  - `grep '^Ll' filename`
  - `grep 'aa|bb|cc' filename`
  - `grep '(aa|bb)c' filename`