Logic
Why Study Logic?

- A basis for computer hardware
- A basis for computer programming
- A basis for program optimization
- A basis for specification
- A basis for verification and testing
In a certain sense

Computing is Logic
Is all Logic Computing?

No, but some of it can be reduced to computing.
Flavors of Logic

- Proposition Logic
- Predicate Logic
- Temporal Logic
- Modal Logics
- Programming Logics
- Fuzzy Logic

- Studied in CS60
- Some exposure in CS80
- Some exposure in CS152 (Neural Networks)
Proposition Logic

- Also known as Switching Logic
- Basic elements are
  - 0 (false)
  - 1 (true)
- Proposition variables (take values 0 or 1)
- Either
  - Functions (functional view)
  - Connectives (expression view)
Mostly we use

- the function view
- and occasionally
  - the expression view
Proposition Logic Domain

- \{\text{false, true}\} (for purists)
  
or
  \{0, 1\} (more readable)
  
or
  \{\Box, T\} (more symmetric)
Proposition Logic Functions

- and
- or
- not
- implies
- iff (if, and only if)
- others
### and

#### Form 1 Table:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>and(x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Arguments** | **Results**
**and**

- **form 2 table:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>and(x, y)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

results
and

- rex "table":
  - and(0, 0) => 0;
  - and(0, 1) => 0;
  - and(1, 0) => 0;
  - and(1, 1) => 1;
and

- shorter rex rules (using sequential convention):
  - \text{and}(1, 1) => 1;
  - \text{and}(x, y) => 0;
common and symbols

- infix \( \land \) (mathematical logic)
- infix . (engineering)
- juxtaposition, as in \( xy \)
- infix \&\& (Java, rex, C, ...)
- infix * (a08)
- infix , (Prolog)
**form 1 table:**

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$\text{or}(x, y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
\begin{itemize}
  \item form 2 table:
\end{itemize}

\begin{center}
\begin{tabular}{c|cc}
\text{or}(x, y) & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{tabular}
\end{center}
or

- rex "table"
  - or(0, 0) => 0;
  - or(0, 1) => 1;
  - or(1, 0) => 1;
  - or(1, 1) => 1;
or

shorter rex rules:
- or(0, 0) => 0;
- or(x, y) => 1;
common or symbols

- infix  (mathematical logic)
- infix + (engineering)
- infix || (Java, rex, C, ...)
- infix + (a08)
- infix ; (Prolog)
not

**form 1 table = form 2 table:**

<table>
<thead>
<tr>
<th>x</th>
<th>not(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
not

- rex rules:
  - not(0) => 1;
  - not(1) => 0;
common *not* symbols

- prefix \( \lnot \) (mathematical logic)
- postfix \( ', \overline{ } \) (engineering)
- prefix \( ! \) (Java, rex, C, ...)
- postfix \( ' \) (a08)
- prefix \( \setminus + \) (Prolog)
implies

form 1 table:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>implies(x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
\textbf{implies}

\begin{itemize}
  \item form 2 table:
  \end{itemize}

\[
\begin{array}{c c c}
\text{implies}(x, y) & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
\end{array}
\]

\[
y
\]
implies

- rex rules:
  - implies(0, 0) => 1;
  - implies(0, 1) => 1;
  - implies(1, 0) => 0;
  - implies(1, 1) => 1;
implies

shorter rex rules (sequential):

- implies(1, 0) => 0;
- implies(x, y) => 1;
common *implies* symbols

- *infix* $\implies$ $\implies$ (mathematical logic)

- Usually $a \implies b$ can be treated as an abbreviation for $\frac{a}{b}$

- **Note**: Binding tightest to weakest: $\implies$, $\implies$, $\frac{a}{b}$
**iff**

- form 1 table:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>iff(x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
i ff

- form 2 table:

<table>
<thead>
<tr>
<th>iff(x, y)</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
iff

- rex rules:
  - iff(0, 0) => 1;
  - iff(0, 1) => 0;
  - iff(1, 0) => 0;
  - iff(1, 1) => 1;
iff

- shorter rex rules (sequential):
  - iff(x, x) => 1;
  - iff(x, y) => 0;
common \textit{iff} symbols

- \texttt{infix $\equiv$ $\equiv$ (mathematical logic)}
- \texttt{infix == (Java, rex, C)}
- \texttt{infix = (a08)}
Concise rex Summary
(sequential convention applies)

- \text{and}(1, 1) \Rightarrow 1; \quad \text{and}(x, y) \Rightarrow 0;
- \text{or}(0, 0) \Rightarrow 0; \quad \text{or}(x, y) \Rightarrow 1;
- \text{not}(0) \Rightarrow 1; \quad \text{not}(1) \Rightarrow 0;
- \text{implies}(1, 0) \Rightarrow 0; \quad \text{implies}(x, y) \Rightarrow 1;
- \text{iff}(x, x) \Rightarrow 1; \quad \text{iff}(x, y) \Rightarrow 0;
Expression Forms

- Use for greater readability of certain equalities

- Similar to ordinary discourse

- Binding order, tightest to weakest: not, and, or
Expression Forms

- Example: These mean the same thing:
  - $(a \equiv b) \quad (c \equiv d)$
  - $ab + cd'$
Logical Equivalences

- \( a \land b = b \land a \)  
  **Commutative**
- \( a \land b = b \land a \)

- \( (a \land b) \land c = a \land (b \land c) \)  
  **Associative**
- \( (a \land b) \land c = a \land (b \land c) \)

- \( (a \land b) \land c = (a \land c) \land (b \land c) \)  
  **Distributive**
- \( (a \land b) \land c = (a \land c) \land (b \land c) \)
Logical Equivalences, engineering style

- \( a \cdot b = b \cdot a \)  \hspace{1cm} \text{Commutative}
- \( a + b = b + a \)

- \( (a \cdot b) \cdot c = a \cdot (b \cdot c) \)  \hspace{1cm} \text{Associative}
- \( (a + b) + c = a + (b + c) \)

- \( (a + b) \cdot c = (a \cdot c) + (b \cdot c) \)  \hspace{1cm} \text{Distributive}
- \( (a \cdot b) + c = (a + c) \cdot (b + c) \)
More Logical Equivalences

- \((a \lor 1) = a\)  \hspace{1cm} \text{Identity}
- \((a \land 0) = a\)

- \((a \lor 0) = 0\)  \hspace{1cm} \text{Absorption}
- \((a \land 1) = 1\)
More Logical Equivalences

• \( (a \land b) = (\neg a \lor \neg b) \)
• \( (a \lor b) = (\neg a \land \neg b) \)

\{ \text{DeMorgan’s Laws} \}

\( (a \land \neg a \lor b) = a \land b \)
\( (a \lor \neg (a \land b)) = a \lor b \)

\{ \text{Worth-remembering laws} \}
More Logical Equivalences, engineering style

- \( \overline{a \cdot b} = a' + b' \)
- \( a + b = b' \cdot a' \)

\[
\begin{align*}
\text{\{DeMorgan’s Laws\}}
\end{align*}
\]

- \( (a + a' \cdot b) = a + b \)
- \( (a (a' + b)) = a \cdot b \)

\[
\begin{align*}
\text{\{Worth-remembering laws\}}
\end{align*}
\]
Logical Equivalences for Implies

- \((a \implies b) = (\neg a \lor b)\)
- \((a \implies b) = \neg (a \land \neg b)\)
- \((0 \implies b) = 1\)
- \((1 \implies b) = b\)
- \((a \implies 0) = \neg a\)
- \((a \implies 1) = 1\)
More Logical Equivalences for Implies

- \((a \implies bc) = (a \implies b) \land (a \implies c)\)

- \(((a \implies b) \land c) = (a \implies c) \land (b \implies c)\)

- \(((a \implies b) \land (b \implies c)) \implies (a \implies c)\)

- \((a \implies b) = (\neg b \lor \neg a)\)

- \((a \implies b) = (a \lor b)\)

- \((a \implies b) = (a \lor \neg b)\)
By a *truth assignment* or *combination* for a formula, we mean a function that assigns a value to every variable in the formula.

Example formula

\[((a \land b) \lor c) = (a \lor c) \land (b \lor c)\]

One combination \(a:0, b:1, c:0\)

or \([a, b, c]:[0, 1, 0]\)

A formula together with a combination has a value in the obvious way.
Counting Combinations

For a formula with n distinct variables, how many combinations are there?
A formula that has value true for every combination is called a **tautology**, or is said to be **valid**.

A formula that has value true for some (including possibly every) combination is said to be **satisfiable**.

A formula that has value true for no combination is called a **contradiction**, or is said to be **unsatisfiable**.
A Trichotomy

- Every formula is exactly one of:
  - Tautology
  - Satisfiable but not a tautology
  - Unsatisfiable

- Formula $F$ is a tautology iff $\Box F$ is unsatisfiable.
A Tautology Checker

http://www.cs.hmc.edu/~keller/javaExamples/taut/taut.html

Enter a logical expression to be checked for being a tautology

Check for tautology: \((a + b) > c) = (a > c) * (b > c)\)

is a tautology

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FALSE</td>
</tr>
<tr>
<td>1</td>
<td>TRUE</td>
</tr>
<tr>
<td>any letter</td>
<td>proposition variable</td>
</tr>
<tr>
<td>postfix '</td>
<td>NOT</td>
</tr>
<tr>
<td>infix *</td>
<td>AND</td>
</tr>
<tr>
<td>infix +</td>
<td>OR</td>
</tr>
<tr>
<td>infix &gt;</td>
<td>IMPLIES</td>
</tr>
<tr>
<td>infix =</td>
<td>IFF</td>
</tr>
</tbody>
</table>

Order of precedence is: ' * + > =. Use parentheses to enforce grouping.
Tautology Checker

http://www.cs.hmc.edu/~keller/javaExamples/taut/taut.html

Enter a logical expression to be checked for being a tautology

Check for tautology: \[(a + b) > c) = (a > c) + (b > c)\]

not a tautology, but satisfiable

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FALSE</td>
</tr>
<tr>
<td>1</td>
<td>TRUE</td>
</tr>
<tr>
<td>any letter</td>
<td>proposition variable</td>
</tr>
<tr>
<td>postfix '</td>
<td>NOT</td>
</tr>
<tr>
<td>infix *</td>
<td>AND</td>
</tr>
<tr>
<td>infix +</td>
<td>OR</td>
</tr>
<tr>
<td>infix &gt;</td>
<td>IMPLIES</td>
</tr>
<tr>
<td>infix =</td>
<td>IFF</td>
</tr>
</tbody>
</table>

Order of precedence is: ' * + > =. Use parentheses to enforce grouping.
Tautology Checker

http://www.cs.hmc.edu/~keller/javaExamples/taut/taut.html

Enter a logical expression to be checked for being a tautology

Check for tautology: a * a'

unsatisfiable

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FALSE</td>
</tr>
<tr>
<td>1</td>
<td>TRUE</td>
</tr>
<tr>
<td>any letter</td>
<td>proposition variable</td>
</tr>
<tr>
<td>postfix '</td>
<td>NOT</td>
</tr>
<tr>
<td>infix *</td>
<td>AND</td>
</tr>
<tr>
<td>infix +</td>
<td>OR</td>
</tr>
<tr>
<td>infix &gt;</td>
<td>IMPLIES</td>
</tr>
<tr>
<td>infix =</td>
<td>IFF</td>
</tr>
</tbody>
</table>

Order of precedence is: ' * + > =. Use parentheses to enforce grouping.
Checking Tautology using Full Enumeration or “Truth Table” method

- Make a table with each variable as a column header and a column for the expression to be checked.
- Enumerate all combinations for the variables.
- Evaluate the expression for each combination.
- Check whether each value is 1 (you can stop early if one isn’t.)
Example: Checking Tautology using Full Enumeration or “Truth Table” method

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>(((a \land b) \lor c) = (a \land c) \lor (b \land c))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0 1 1 1 0 1 0 1 1</td>
</tr>
</tbody>
</table>
Example: Checking Tautology using Full Enumeration or “Truth Table” method

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>((a ⊕ b) ⊕ c) = (a ⊕ c) ⊕ (b ⊕ c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 0 1 0 1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0 0 1 0 1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 0 1 0 1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0 0 1 0 1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 0 1 0 1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0 0 1 0 1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0 0 1 0 1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0 0 1 0 1 0 1 0 1 0 1 0</td>
</tr>
</tbody>
</table>
Example: Checking Tautology using Full Enumeration or “Truth Table” method

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>((a \quad b) \quad c) = ((a \quad c) \quad (b \quad c))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Example: Checking Tautology using Full Enumeration or “Truth Table” method

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>(((a \land b) \lor c) = (a \land c) \lor (b \land c))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Checking Tautology using the Boole-Shannon Principle

- Relations hold \textit{iff} they hold for every substitution of 0 and 1 for the variables (\textit{uniformly} throughout the expression)

- Therefore, a relation holds if, choosing any variable \( V \), it holds for \( V = 0 \) and for \( V = 1 \).

- But substituting 0 or 1 for a variable often yields \textit{simplifications} that make the relation obvious.
Example: Boole-Shannon Principle

- Verify \((a \oplus b) = (\neg b \oplus \neg a)\)
- Choose \(a\) as the variable.
  - Substituting 0 for \(a\):
    - \((0 \oplus b) = (\neg b \oplus \neg 0)\)
    - which simplifies to:
      - \(1 = (\neg b \oplus 1), a\) known equivalence
  - Substituting 1 for \(a\):
    - \((1 \oplus b) = (\neg b \oplus \neg 1)\)
    - which simplifies to:
      - \(b = (\neg b \oplus 0)\)
Not related: Bull Shannon of “Night Court”

Bull Shannon, as played by Richard Moll, 1984
Boole and Shannon

- **Boole**
  - Invented “Boolean algebra” (switching theory)
  - (In modern mathematics, “Boolean algebra” is a more general, abstract, system)

- **Shannon**
  - Wrote thesis on switching theory
  - Invented “Information theory”
  - Maze-solving mouse
  - Wrote first chess-playing program
  - Wrote paper on the mathematics of juggling
Boole and Shannon

George Boole (1815-1864)  Claude Shannon (1916-2001)
Binary Decision Diagrams (BDD’s)

- A way to evaluate an expression
- Used extensively in computer engineering
- Related to Boole-Shannon Principle
Binary Decision Diagrams (BDD’s)

- A BDD is a directed acyclic graph
- The leaves are either 0 or 1.
- Each non-leaf node is
  - labeled with a variable
  - has two arcs leaving it:
    - true branch
    - false branch
Example BDD
Each BDD represents some formula

\((\overline{a} \overline{b}) (a \overline{b})\)

\(a = b\)
Constructing a BDD from a Formula using the Boole-Shannon Expansion

- **ConstructBDD(F):**
  - If \( F \) contains no variable, return a node with its value (1 or 0).
  - Choose a variable \( v \) in \( F \). Let \( F_{v=0} \) mean \( F \) with all instances of \( v \) replaced with 0 (false), and similarly for \( F_{v=1} \).
  - Return a tree with root labeled \( v \), with the false branch connecting to ConstructBDD(\( F_{v=0} \)) and the true branch connecting to ConstructBDD(\( F_{v=1} \)).
Constructing a BDD from a Formula using the Boole-Shannon Expansion

- ConstructBDD((0 0 1) (0 1)):
- The basis rule applies:
  - Evaluate: 0
  - Return 0
Constructing a BDD from a Formula using the Boole-Shannon Expansion

ConstructBDD\left((\overline{a} \vee \overline{b}) \ (a \wedge b)\right):

- The basis rule does not apply.
- Choose a variable, say \(a\):
  - Construct BDD\left((\overline{0} \vee \overline{b}) \ (0 \wedge b)\right)
  - Construct BDD\left((\overline{1} \vee \overline{b}) \ (1 \wedge b)\right)
  - Connect to node labeled \(a\)
Simplifying Formulas using BDD’s

- Two nodes that are the roots of identical sub-graphs can be merged together (two references sharing a sub-graph).

- A node having both true and false branches going to the sub-graph can be replaced with the sub-graph itself.

- Re-ordering nodes sometimes enables the above simplifications.