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

Proposition Logic, Predicate Logic: Studied in CS60

Temporal Logic, Modal Logics, Programming Logics: Some exposure in CS80

Fuzzy Logic: 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

- {false, true} (for purists)

  or

  {0, 1} (more readable)

  or

  {□, T } (more symmetric)

Proposition Logic Functions

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

and

form 1 table:

x	y	and(x, y)
0	0	0
0	1	0
1	0	0
1	1	1

arguments results

and

- form 2 table:

and(x, y)		y = 0	y = 1
x	0	0	0
x	1	0	1

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):
  - and(1, 1) => 1;
  - and(x, y) => 0;

common *and* symbols

- infix □ (mathematical logic)
- infix . (engineering)
- juxtaposition, as in xy
- infix && (Java, rex, C, ...)
- infix * (a08)
- infix , (Prolog)

or

- form 1 table:

x	y	or(x, y)
0	0	0
0	1	1
1	0	1
1	1	1

or

- form 2 table:

or(x, y)	0	1
0	0	1
1	1	1

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:

x	not(x)
0	1
1	0

not

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

common ***not*** symbols

- prefix ¬ (mathematical logic)
- postfix ' , $\overline{\text{overbar}}$ (engineering)
- prefix ! (Java, rex, C, ...)
- postfix ' (a08)
- prefix \+ (Prolog)

implies

- form 1 table:

x	y	implies(x, y)
0	0	1
0	1	1
1	0	0
1	1	1

implies

- form 2 table:

implies(x, y)		y = 0	y = 1
x	0	1	1
x	1	0	1

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 □ □ (mathematical logic)
- Usually a □ b can be treated as an abbreviation for □a b
- **Note**: Binding tightest to weakest: □, □,

iff

- form 1 table:

x	y	iff(x, y)
0	0	1
0	1	0
1	0	0
1	1	1

iff

- form 2 table:

iff(x, y)	0	1
0	1	0
1	0	1

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 *iff* symbols

- infix ≡ □ □ (mathematical logic)
- infix == (Java, rex, C)
- infix = (a08)

Concise rex Summary

(sequential convention applies)

- and(1, 1) => 1; and(x, y) => 0;
- or(0, 0) => 0; or(x, y) => 1;
- not(0) => 1; not(1) => 0;
- implies(1, 0) => 0; implies(x, y) => 1;
- iff(x, x) => 1; iff(x, y) => 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 □ b)   (c □ □ d)
  - ab + cd′

Logical Equivalences

- $a \wedge b = b \wedge a$ — Commutative
- $a \vee b = b \vee a$

- $(a \wedge b) \wedge c = a \wedge (b \wedge c)$ — Associative
- $(a \vee b) \vee c = a \vee (b \vee c)$

- $(a \vee b) \wedge c = (a \wedge c) \vee (b \wedge c)$ — Distributive
- $(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$

Logical Equivalences, engineering style

- a b = b a — Commutative
- a + b = b + a

- (a b) c = a (b c) — Associative
- (a + b) + c = a + (b + c)

- (a + b) c = (a c) + (b c) — Distributive
- (a b) + c = (a + c) (b + c)

More Logical Equivalences

- $(a \wedge 1) = a$
- $(a \vee 0) = a$

Identity

- $(a \wedge 0) = 0$
- $(a \vee 1) = 1$

Absorption

More Logical Equivalences

- □(a□ b) = (□ a   □ b)
- □(a   b) = (□ a □ □ b)

DeMorgan's Laws

- (a   □a b)   = a   b
- (a □ (□ a   b)) = a □ b

Worth-remembering laws

More Logical Equivalences, engineering style

- $\overline{a\ b} = a' + b'$
- $\overline{a + b} = b'\ a'$

} DeMorgan's Laws

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

} Worth-remembering laws

Logical Equivalences for Implies

- $(a \rightarrow b) = (\neg a \vee b)$
- $(a \rightarrow b) = \neg(a \wedge \neg b)$

- $(0 \rightarrow b) = 1$
- $(1 \rightarrow b) = b$
- $(a \rightarrow 0) = \neg a$
- $(a \rightarrow 1) = 1$

More Logical Equivalences for Implies

- (a □ bc) = (a □ b) □ (a □ c)
- ((a b) □ c) = (a □ c) □ (b □ c)
- ((a □ b) □ (b □ c)) □ (a □ c)
- (a □ b) = (□b □ □ a)
- □(a □ b) = (a □□b)

Truth Assignment or "Combination"

- 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

  $$((a \quad b) \,\square\; c) = (a \,\square\; c) \quad (b \,\square\; c)$$

- **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?

Tautologies, etc.

- 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 $\neg F$ is unsatifiable.

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)

Clear All   is a tautology

Symbol	Meaning
0	FALSE
1	TRUE
any letter	proposition variable
postfix '	NOT
infix *	AND
infix +	OR
infix >	IMPLIES
infix =	IFF

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)

Clear All

not a tautology, but satisfiable

Symbol	Meaning
0	FALSE
1	TRUE
any letter	proposition variable
postfix '	NOT
infix *	AND
infix +	OR
infix >	IMPLIES
infix =	IFF

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'

Clear All

unsatisfiable

Symbol	**Meaning**
0	FALSE
1	TRUE
any letter	proposition variable
postfix '	NOT
infix *	AND
infix +	OR
infix >	IMPLIES
infix =	IFF

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

a	b	c	((a b) □ c) = (a □ c) □ (b □ c)
0	0	0	0 0 0 0 0 0 0
			0 1 1
			1 1
			1

Example: Checking Tautology using Full Enumeration or "Truth Table" method

a	b	c	((a b) □ c) = (a □ c) □ (b □ c)
0	0	0	0 0 1 0 1 0 1 0 1 0 1 0
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Example: Checking Tautology using Full Enumeration or "Truth Table" method

a	b	c	((a b) □ c) = (a □ c) □ (b □ c)
0	0	0	1
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Example: Checking Tautology using Full Enumeration or "Truth Table" method

a	b	c	((a b) □ c) = (a □ c) □ (b □ c)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

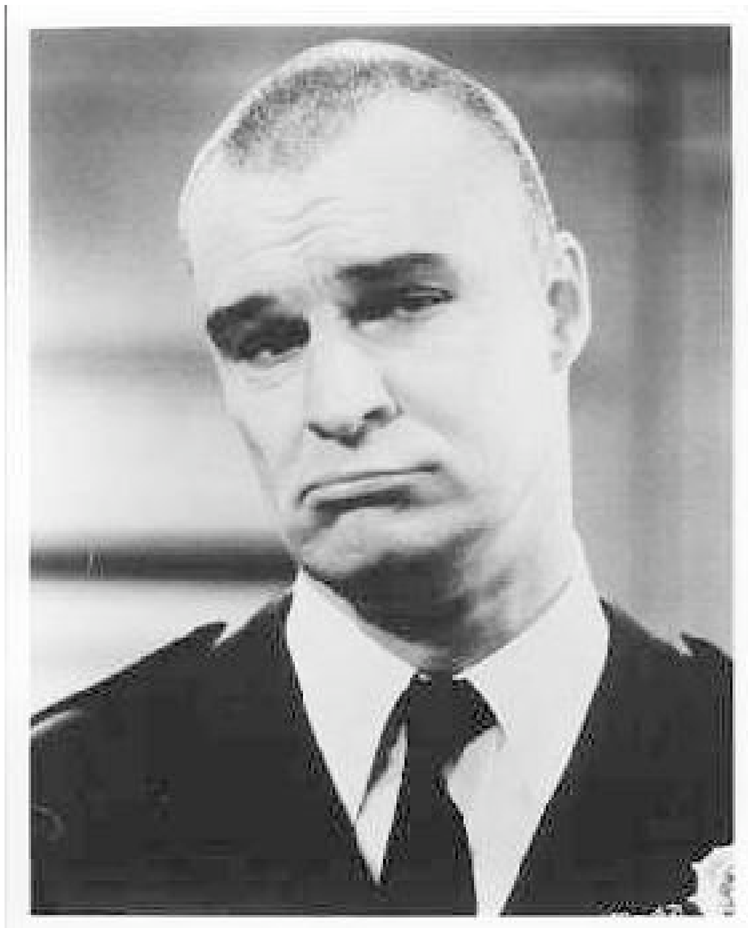
Checking Tautology using the **Boole-Shannon Principle**

- Relations hold **iff** they hold for **every substitution** of 0 and 1 for the variables (**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 **simplifications** that make the relation obvious.

Example: Boole-Shannon Principle

- Verify $(a \Rightarrow b) = (\neg b \Rightarrow \neg a)$
- Choose a as the variable.
  - Substituting 0 for a:
    - $(0 \Rightarrow b) = (\neg b \Rightarrow \neg 0)$
  - which simplifies to:
    - $1 = (\neg b \Rightarrow 1)$, a known equivalence
  - Substituting 1 for a:
    - $(1 \Rightarrow b) = (\neg b \Rightarrow \neg 1)$
  - which simplifies to:
    - $b = (\neg b \Rightarrow 0)$

Not related: Bull Shannon of "Night Court"



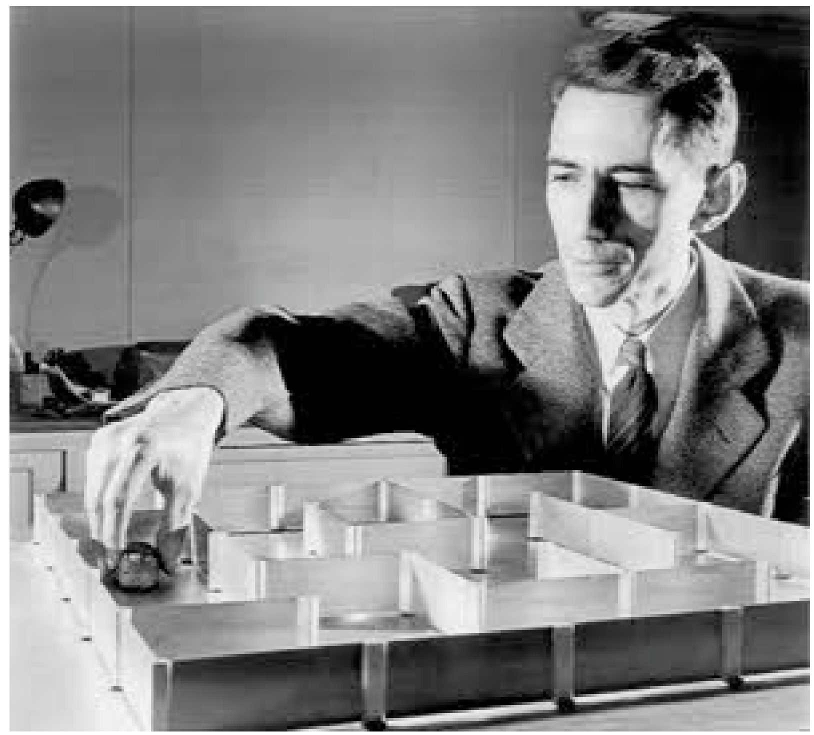
Bull Shannon, as played by Richard Moll, 1984

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)

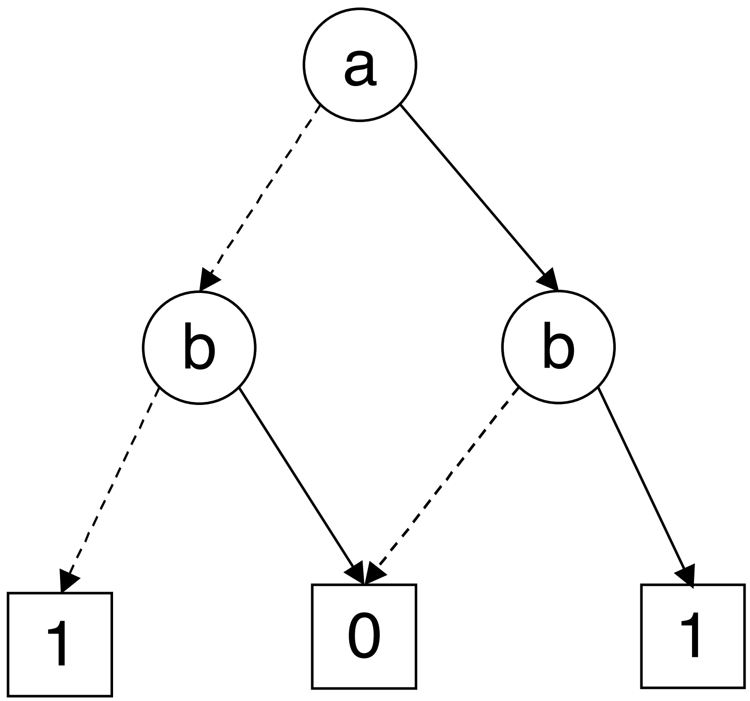
Binary Decision Diagrams (BDD's)

- 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 - - - - -

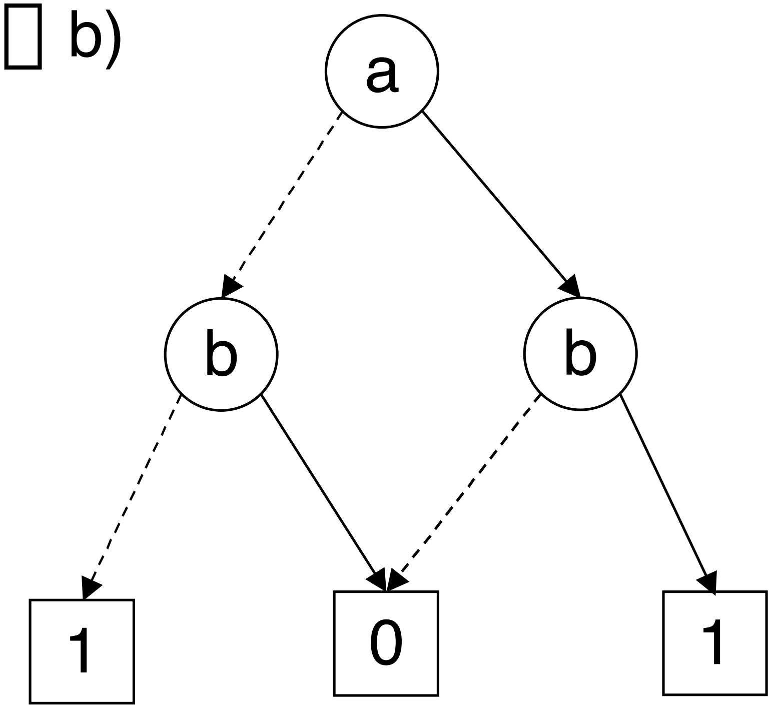
Example BDD



Each BDD represents some formula

(□a □ □b) (a □ 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

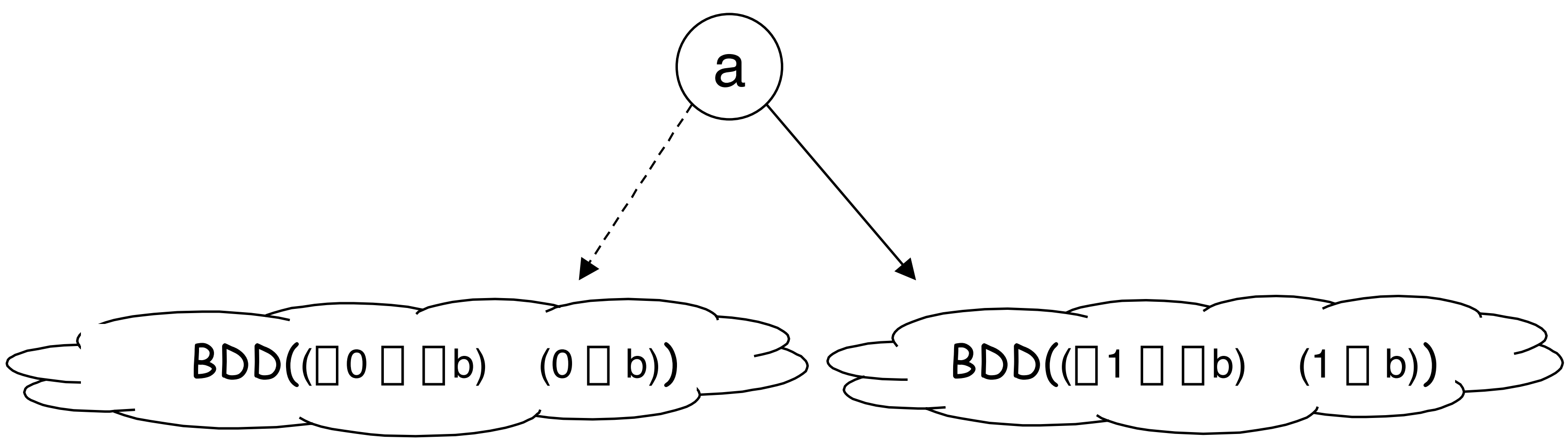
- ConstructBDD((□0 □ □1) (0 □ 1)):
- The basis rule applies:
  - Evaluate: 0
  - Return

0

Constructing a BDD from a Formula using the Boole-Shannon Expansion

ConstructBDD((□a □ □b) (a □ b)):

- The basis rule does not apply.
- Choose a variable, say a:
  - Construct BDD((□0 □ □b) (0 □ b))
  - Construct BDD((□1 □ □b) (1 □ b))
  - 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.