Lecture 5: DNF, CNF and Predicate Logic
1-bit Binary Adder

\[
\begin{align*}
A + B & : \quad 0 + 0 = 0 \text{ (with } C_{\text{OUT}} = 0) \\
& \quad 0 + 1 = 1 \text{ (with } C_{\text{OUT}} = 0) \\
S & \quad 1 + 0 = 1 \text{ (with } C_{\text{OUT}} = 0) \\
(C_{\text{OUT}}) & \quad 1 + 1 = 0 \text{ (with } C_{\text{OUT}} = 1) 
\end{align*}
\]

**Idea:** These are chained together, with a carry-in.
1-bit Binary Adder

- **Inputs:** A, B, Carry-in
- **Outputs:** Sum, Carry-out

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C\textsubscript{IN}</th>
<th>C\textsubscript{OUT}</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
1-bit Binary Adder

- **Inputs:** A, B, Carry-in
- **Outputs:** Sum, Carry-out

### Truth Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C\text{IN}</th>
<th>C\text{OUT}</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Derive an expression for S:

\[ S = A' \cdot B' \cdot C_{\text{IN}} + A' \cdot B \cdot C_{\text{IN}}' + A \cdot B' \cdot C_{\text{IN}}' + A \cdot B \cdot C_{\text{IN}} \]
1-bit Binary Adder

- **Inputs:** A, B, Carry-in
- **Outputs:** Sum, Carry-out

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C\textsubscript{IN}</th>
<th>C\textsubscript{OUT}</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Derive an expression for $C\text{OUT}$

$$C\text{OUT} = A' \cdot B \cdot C\text{IN} + A \cdot B' \cdot C\text{IN}' + A \cdot B \cdot C\text{IN}' + A \cdot B' \cdot C\text{IN}$$

$$S = A' \cdot B' \cdot C\text{IN} + A' \cdot B \cdot C\text{IN}' + A \cdot B' \cdot C\text{IN}' + A \cdot B \cdot C\text{IN}$$
1-bit Binary Adder

- **Inputs:** A, B, Carry-in
- **Outputs:** Sum, Carry-out

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C_{IN}</th>
<th>C_{OUT}</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
S = A' \cdot B' \cdot C_{IN} + A' \cdot B \cdot C_{IN}' + A \cdot B' \cdot C_{IN}' + A \cdot B \cdot C_{IN}
\]

\[
C_{OUT} = A' \cdot B \cdot C_{IN} + A \cdot B' \cdot C_{IN} + A \cdot B \cdot C_{IN}' + A \cdot B \cdot C_{IN}
\]
Apply Theorems to Simplify Expressions

The theorems of Boolean algebra can simplify expressions – e.g., full adder’s carry-out function

\[
\text{Cout} = A' \cdot \text{B Cin} + \text{A B' Cin} + \text{A B Cin' + A B Cin}
\]

\[
= A' \cdot \text{B Cin} + \text{A B' Cin} + \text{A B Cin'} + \text{A B Cin} + \text{A B Cin}
\]

\[
= (A' + A) \cdot \text{B Cin} + \text{A B' Cin} + \text{A B Cin'} + \text{A B Cin}
\]

\[
= (1) \cdot \text{B Cin} + \text{A B' Cin} + \text{A B Cin'} + \text{A B Cin}
\]

\[
= \text{B Cin} + \text{A B' Cin} + \text{A B Cin'} + \text{A B Cin} + \text{A B Cin}
\]

\[
= \text{B Cin} + \text{A (B' + B) Cin} + \text{A B Cin'} + \text{A B Cin}
\]

\[
= \text{B Cin} + \text{A (1) Cin} + \text{A B Cin'} + \text{A B Cin}
\]

\[
= \text{B Cin} + \text{A Cin} + \text{A B (Cin' + Cin)}
\]

\[
= \text{B Cin} + \text{A Cin} + \text{A B (1)}
\]

\[
= \text{B Cin} + \text{A Cin} + \text{A B}
\]

adding extra terms creates new factoring opportunities
A 2-bit Ripple-Carry Adder

Uses the fact that

\[ \text{Sum} = A' \cdot B' \cdot C_{\text{IN}} + A' \cdot B \cdot C_{\text{IN}}' + A \cdot B' \cdot C_{\text{IN}}' + A \cdot B \cdot C_{\text{IN}} \]

is equivalent to \( \text{Sum} = (A \oplus B) \oplus C_{\text{IN}} \)
Mapping Truth Tables to Logic Gates

Given a truth table:

1. Write the Boolean expression
2. Minimize the Boolean expression
3. Draw as gates
4. Map to available gates

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

F = A′BC′+A′BC+AB′C+ABC

= A′B(C′+C)+AC(B′+B)

= A′B+AC

1. Write the Boolean expression

2. Minimize the Boolean expression

3. Draw as gates

4. Map to available gates
Canonical Forms

• Truth table is the unique signature of a Boolean Function

• The same truth table can have many gate realizations
  – We’ve seen this already
  – Depends on how good we are at Boolean simplification

• Canonical forms
  – Standard forms for a Boolean expression
  – We all come up with the same expression
Sum-of-Products Canonical Form

• AKA **Disjunctive Normal Form (DNF)**
• AKA **Minterm Expansion**

$$F = A'B'C + A'BC + AB'C + ABC' + ABC'$$

1. Read T rows off truth table
2. Convert to Boolean Algebra
3. Add the minterms together

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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>0</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>0</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>

$$F$$
Sum-of-Products Canonical Form

Product term (or minterm)

- ANDed product of literals – input combination for which output is true
- each variable appears exactly once, true or inverted (but not both)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A'B'C'</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A'B'C</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A'BC'</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A'BC</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>AB'C'</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>AB'C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ABC'</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ABC</td>
</tr>
</tbody>
</table>

F in canonical form:

\[
F(A, B, C) = A'B'C + A'BC + AB'C + ABC' + ABC
\]

canonical form ≠ minimal form

\[
F(A, B, C) = A'B'C + A'BC + AB'C + ABC + ABC'
= (A'B' + A'B + AB' + AB)C + ABC'
= ((A' + A)(B' + B))C + ABC'
= C + ABC'
= ABC' + C
= AB + C
\]
Product-of-Sums Canonical Form

- **AKA** Conjunctive Normal Form (CNF)
- **AKA** Maxterm Expansion

\[ F = \text{Multiply the maxterms together} \]

1. **Read F rows off truth table**
2. **Negate all bits**
3. **Convert to Boolean Algebra**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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>0</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>0</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>
Product-of-Sums Canonical Form

- AKA **Conjunctive Normal Form (CNF)**
- AKA **Maxterm Expansion**

F = (A + B + C)(A + B' + C)(A' + B + C)

Read F rows off truth table

Negate all bits

Convert to Boolean Algebra

Multiply the maxterms together
Product-of-Sums: Why does this procedure work?

Useful Facts:

- We know \((F')' = F\)
- We know how to get a minterm expansion for \(F'\)

\[
F' = A'B'C' + A'BC' + AB'C'
\]
Product-of-Sums: Why does this procedure work?

Useful Facts:

- We know \((F')' = F\)
- We know how to get a minterm expansion for \(F'\)

\[
\begin{array}{c|c|c|c|c}
A & B & C & F \\
\hline
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

\[F' = A'B'C' + A'BC' + AB'C'\]

Taking the complement of both sides...

\[(F')' = (A'B'C' + A'BC' + AB'C')'\]

And using DeMorgan/Comp....

\[F = (A'B'C')' (A'BC')' (AB'C')'\]

\[F = (A + B + C)(A + B' + C)(A' + B + C)\]
Product-of-Sums Canonical Form

Sum term (or maxterm)
- ORed sum of literals – input combination for which output is false
- each variable appears exactly once, true or inverted (but not both)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>maxterms</th>
<th>F in canonical form:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A+B+C</td>
<td>F(A, B, C) = (A + B + C) (A + B’ + C) (A’ + B + C)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A+B+C’</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A+B’+C</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A+B’+C’</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A’+B+C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A’+B’+C’</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A’+B’+C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A’+B’+C’</td>
<td></td>
</tr>
</tbody>
</table>

canonical form ≠ minimal form

F(A, B, C) = (A + B + C) (A + B’ + C) (A’ + B + C)
= (A + B + C) (A + B’ + C)
= (A + B + C) (A’ + B + C)
= (A + C) (B + C)
Predicate Logic

• **Propositional Logic**
  “If you take the high road and I take the low road then I’ll arrive in Scotland before you.”

• **Predicate Logic**
  “All positive integers $x$, $y$, and $z$ satisfy $x^3 + y^3 \neq z^3$.”
Predicate Logic

• **Propositional Logic**
  
  – Allows us to analyze complex propositions in terms of their simpler constituent parts (a.k.a. atomic propositions) joined by connectives

• **Predicate Logic**
  
  – Lets us analyze them at a deeper level by expressing how those propositions depend on the objects they are talking about
Predicate Logic

Adds two key notions to propositional logic
   – Predicates

   – Quantifiers
Predicates

Predicate

– A function that returns a truth value, e.g.,

Cat(x) ::= “x is a cat”
Prime(x) ::= “x is prime”
HasTaken(x, y) ::= “student x has taken course y”
LessThan(x, y) ::= “x < y”
Sum(x, y, z) ::= “x + y = z”
GreaterThan5(x) ::= “x > 5”
HasNChars(s, n) ::= “string s has length n”

Predicates can have varying numbers of arguments and input types.
Domain of Discourse

For ease of use, we define one “type”/“domain” that we work over. This set of objects is called the “domain of discourse”.

For each of the following, what might the domain be?
(1) “x is a cat”, “x barks”, “x ruined my couch”

(2) “x is prime”, “x = 0”, “x < 0”, “x is a power of two”

(3) “student x has taken course y” “x is a pre-req for z”
Domain of Discourse

For ease of use, we define one “type”/“domain” that we work over. This set of objects is called the “domain of discourse”.

For each of the following, what might the domain be?

(1) “\(x\) is a cat”, “\(x\) barks”, “\(x\) ruined my couch”
   “mammals” or “sentient beings” or “cats and dogs” or ...

(2) “\(x\) is prime”, “\(x = 0\)”, “\(x < 0\)”, “\(x\) is a power of two”
   “numbers” or “integers” or “integers greater than 5” or ...

(3) “student \(x\) has taken course \(y\)” “\(x\) is a pre-req for \(z\)”
   “students and courses” or “university entities” or ...
Quantifiers

We use quantifiers to talk about collections of objects.

\[ \forall x \ P(x) \]

\( P(x) \) is true for every \( x \) in the domain
read as “for all \( x, P \) of \( x \)”

\[ \exists x \ P(x) \]

There is an \( x \) in the domain for which \( P(x) \) is true
read as “there exists \( x, P \) of \( x \)”
Quantifiers

We use quantifiers to talk about collections of objects.

Universal Quantifier (“for all”): \( \forall x \ P(x) \)

\( P(x) \) is true for every \( x \) in the domain

read as “for all \( x \), \( P \) of \( x \)”

Examples: Are these true?

• \( \forall x \ \text{Odd}(x) \)

• \( \forall x \ \text{LessThan5}(x) \)
Quantifiers

We use *quantifiers* to talk about collections of objects.

**Universal Quantifier ("for all"):** \( \forall x \ P(x) \)

- \( P(x) \) is true for every \( x \) in the domain
- read as “for all \( x \), \( P \) of \( x \)”

**Examples:** Are these true? It depends on the domain. For example:

<table>
<thead>
<tr>
<th>( {1, 3, -1, -27} )</th>
<th>Integers</th>
<th>Odd Integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
We use quantifiers to talk about collections of objects.

Existential Quantifier ("exists"):  \( \exists x \ P(x) \)

*There is* an \( x \) in the domain for which \( P(x) \) is true

read as "*there exists* \( x \), \( P \) of \( x \)

Examples:  

- \( \exists x \ \text{Odd}(x) \)

- \( \exists x \ \text{LessThan5}(x) \)
Quantifiers

We use quantifiers to talk about collections of objects.

Existential Quantifier (“exists”): \( \exists x \ P(x) \)

There is an \( x \) in the domain for which \( P(x) \) is true

read as “there exists \( x \), \( P \) of \( x \)”

Examples: Are these true? It depends on the domain. For example:

<table>
<thead>
<tr>
<th>( \exists x ) Odd(x)</th>
<th>( {1, 3, -1, -27} )</th>
<th>Integers</th>
<th>Positive Multiples of 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
Statements with Quantifiers

Just like with propositional logic, we need to define variables (this time predicates) before we do anything else. We must also now define a domain of discourse before doing anything else.

<table>
<thead>
<tr>
<th>Domain of Discourse</th>
<th>Predicate Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Integers</td>
<td>Even(x) ::= “x is even” Greater(x, y) ::= “x &gt; y”</td>
</tr>
<tr>
<td></td>
<td>Odd(x) ::= “x is odd”  Equal(x, y) ::= “x = y”</td>
</tr>
<tr>
<td></td>
<td>Prime(x) ::= “x is prime” Sum(x, y, z) ::= “x + y = z”</td>
</tr>
</tbody>
</table>
Statements with Quantifiers

<table>
<thead>
<tr>
<th>Domain of Discourse</th>
<th>Predicate Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Integers</td>
<td>Even(x) ::= “x is even” Greater(x, y) ::= “x &gt; y”</td>
</tr>
<tr>
<td></td>
<td>Odd(x) ::= “x is odd”  Equal(x, y) ::= “x = y”</td>
</tr>
<tr>
<td></td>
<td>Prime(x) ::= “x is prime” Sum(x, y, z) ::= “x + y = z”</td>
</tr>
</tbody>
</table>

Determine the truth values of each of these statements:

∃x Even(x)

∀x Odd(x)

∀x (Even(x) ∨ Odd(x))

∃x (Even(x) ∧ Odd(x))

∀x Greater(x+1, x)

∃x (Even(x) ∧ Prime(x))
Statements with Quantifiers

Predicate Definitions

<table>
<thead>
<tr>
<th>Domain of Discourse</th>
<th>Positive Integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Even(x) ::= “x is even”</td>
<td>Greater(x, y) ::= “x &gt; y”</td>
</tr>
<tr>
<td>Odd(x) ::= “x is odd”</td>
<td>Equal(x, y) ::= “x = y”</td>
</tr>
<tr>
<td>Prime(x) ::= “x is prime”</td>
<td>Sum(x, y, z) ::= “x + y = z”</td>
</tr>
</tbody>
</table>

Determine the truth values of each of these statements:

- $\exists x \text{ Even}(x)$
  - T e.g. 2, 4, 6, ...

- $\forall x \text{ Odd}(x)$
  - F e.g. 2, 4, 6, ...

- $\forall x (\text{Even}(x) \lor \text{Odd}(x))$
  - T every integer is either even or odd

- $\exists x (\text{Even}(x) \land \text{Odd}(x))$
  - F no integer is both even and odd

- $\forall x \text{ Greater}(x+1, x)$
  - T adding 1 makes a bigger number

- $\exists x (\text{Even}(x) \land \text{Prime}(x))$
  - T Even(2) is true and Prime(2) is true
Statements with Quantifiers

<table>
<thead>
<tr>
<th>Domain of Discourse</th>
<th>Predicate Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Integers</td>
<td>Even(x) ::= “x is even”</td>
</tr>
<tr>
<td></td>
<td>Odd(x) ::= “x is odd”</td>
</tr>
<tr>
<td></td>
<td>Prime(x) ::= “x is prime”</td>
</tr>
<tr>
<td></td>
<td>Greater(x, y) ::= “x &gt; y”</td>
</tr>
<tr>
<td></td>
<td>Equal(x, y) ::= “x = y”</td>
</tr>
<tr>
<td></td>
<td>Sum(x, y, z) ::= “x + y = z”</td>
</tr>
</tbody>
</table>

Translate the following statements to English

∀x ∃y Greater(y, x)

∀x ∃y Greater(x, y)

∀x ∃y (Greater(y, x) ∧ Prime(y))

∀x (Prime(x) → (Equal(x, 2) ∨ Odd(x)))

∃x ∃y (Sum(x, 2, y) ∧ Prime(x) ∧ Prime(y))
Statements with Quantifiers (Literal Translations)

<table>
<thead>
<tr>
<th>Domain of Discourse</th>
<th>Predicate Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Integers</td>
<td>Even(x) ::= “x is even” Greater(x, y) ::= “x &gt; y”</td>
</tr>
<tr>
<td></td>
<td>Odd(x) ::= “x is odd”  Equal(x, y) ::= “x = y”</td>
</tr>
<tr>
<td></td>
<td>Prime(x) ::= “x is prime” Sum(x, y, z) ::= “x + y = z”</td>
</tr>
</tbody>
</table>

Predicate Definitions

Translate the following statements to English

∀x ∃y Greater(y, x)
   For every positive integer x, there is a positive integer y, such that y > x.

∀x ∃y Greater(x, y)
   For every positive integer x, there is a positive integer y, such that x > y.

∀x ∃y (Greater(y, x) ∧ Prime(y))
   For every positive integer x, there is a pos. int. y such that y > x and y is prime.

∀x (Prime(x) → (Equal(x, 2) ∨ Odd(x)))
   For each positive integer x, if x is prime, then x = 2 or x is odd.

∃x ∃y (Sum(x, 2, y) ∧ Prime(x) ∧ Prime(y))
   There exist positive integers x and y such that x + 2 = y and x and y are prime.
Translate the following statements to English

\(\forall x \exists y \text{ Greater}(y, x)\)

There is no greatest positive integer.

\(\forall x \exists y \text{ Greater}(x, y)\)

There is no least positive integer.

\(\forall x \exists y (\text{Greater}(y, x) \land \text{Prime}(y))\)

For every positive integer there is a larger number that is prime.

\(\forall x (\text{Prime}(x) \rightarrow (\text{Equal}(x, 2) \lor \text{Odd}(x)))\)

Every prime number is either 2 or odd.

\(\exists x \exists y (\text{Sum}(x, 2, y) \land \text{Prime}(x) \land \text{Prime}(y))\)

There exist prime numbers that differ by two.”
“Red cats like tofu”

“Some red cats don’t like tofu”
English to Predicate Logic

<table>
<thead>
<tr>
<th>Domain of Discourse</th>
<th>Predicate Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mammals</td>
<td>Cat(x) ::= “x is a cat”</td>
</tr>
<tr>
<td></td>
<td>Red(x) ::= “x is red”</td>
</tr>
<tr>
<td></td>
<td>LikesTofu(x) ::= “x likes tofu”</td>
</tr>
</tbody>
</table>

“Red cats like tofu”

∀x ((Red(x) ∧ Cat(x)) → LikesTofu(x))

“Some red cats don’t like tofu”

∃y ((Red(y) ∧ Cat(y)) ∧ ¬LikesTofu(y))
English to Predicate Logic

**Domain of Discourse**

| Mammals |

**Predicate Definitions**

- Cat(x) ::= “x is a cat”
- Red(x) ::= “x is red”
- LikesTofu(x) ::= “x likes tofu”

“When putting two predicates together like this, we use an “and”.

“Red cats like tofu”

“When there’s no leading quantification, it means “for all”.

“Some red cats don’t like tofu”

“When restricting to a smaller domain in a “for all” we use implication.

“Some” means “there exists”.

“When restricting to a smaller domain in an “exists” we use and.”
Negations of Quantifiers

Predicate Definitions

| PurpleFruit(x) ::= “x is a purple fruit” |

(*) \( \forall x \) PurpleFruit(x) (“All fruits are purple”)

What is the negation of (*)?

(a) “there exists a purple fruit”
(b) “there exists a non-purple fruit”
(c) “all fruits are not purple”

Try your intuition! Which one “feels” right?

Key Idea: In every domain, exactly one of a statement and its negation should be true.
Negations of Quantifiers

Predicate Definitions

PurpleFruit(x) ::= “x is a purple fruit”

(\forall x \text{ PurpleFruit}(x) \ (“All fruits are purple”))

What is the negation of (\text{"*"})?

(a) “there exists a purple fruit”
(b) “there exists a non-purple fruit”
(c) “all fruits are not purple”

Key Idea: In every domain, exactly one of a statement and its negation should be true.

Domain of Discourse

Domain of Discourse

Domain of Discourse

\{plum\} \hspace{1cm} \{apple\} \hspace{1cm} \{plum, apple\}

The only choice that ensures exactly one of the statement and its negation is (b).
De Morgan’s Laws for Quantifiers

\[ \neg \forall x \ P(x) \equiv \exists x \ \neg P(x) \]
\[ \neg \exists x \ P(x) \equiv \forall x \ \neg P(x) \]
De Morgan’s Laws for Quantifiers

\[ \neg \forall x \ P(x) \equiv \exists x \ \neg P(x) \]
\[ \neg \exists x \ P(x) \equiv \forall x \ \neg P(x) \]

“There is no largest integer”
\[ \neg \exists x \ \forall y \ (x \geq y) \]
\[ \equiv \ \forall x \ \neg \forall y \ (x \geq y) \]
\[ \equiv \ \forall x \ \exists y \ \neg (x \geq y) \]
\[ \equiv \ \forall x \ \exists y \ (x < y) \]

“For every integer there is a larger integer”
Scope of Quantifiers

\[ \exists x \ (P(x) \land Q(x)) \quad \text{vs.} \quad \exists x \ P(x) \land \exists x \ Q(x) \]
Scope of Quantifiers

\[ \exists x \ (P(x) \land Q(x)) \quad \text{vs.} \quad \exists x \ P(x) \land \exists x \ Q(x) \]

This one asserts P and Q of the same \( x \).

This one asserts P and Q of potentially different \( x \)’s.
Scope of Quantifiers

Example:  NotLargest(x) ≡ ∃ y Greater (y, x)  
          ≡ ∃ z Greater (z, x)

truth value:

doesn’t depend on y or z “bound variables”
does depend on x “free variable”

quantifiers only act on free variables of the formula they quantify  
       ∀ x (∃ y (P(x,y) → ∀ x Q(y, x))
Quantifier “Style”

\[ \forall x (\exists y (P(x,y) \rightarrow \forall x Q(y,x))) \]

This isn’t “wrong”, it’s just horrible style. Don’t confuse your reader by using the same variable multiple times...there are a lot of letters...