CSE 311: Foundations of Computing

Lecture 5: DNF, CNF and Predicate Logic
Administrative

• HW1 due today
  – Submit via Gradescope by 11:00 pm
  – EC1 extra credit submitted separately

• Tomorrow:
  – HW2 out
  – Quiz sections
  – 390Z/ZA sign-up still available
    Loew 113 Thursday 3:30-5:00
Last Class: 1-bit Binary Adder

\[
\begin{array}{cccc}
A & + & B & S \\
\text{(C}_{\text{OUT}}) & & & (\text{C}_{\text{OUT}})
\end{array}
\]

\[
\begin{array}{c}
0 + 0 = 0 \text{ (with } C_{\text{OUT}} = 0) \\
0 + 1 = 1 \text{ (with } C_{\text{OUT}} = 0) \\
1 + 0 = 1 \text{ (with } C_{\text{OUT}} = 0) \\
1 + 1 = 0 \text{ (with } C_{\text{OUT}} = 1)
\end{array}
\]

Idea: These are chained together, with a carry-in
Design Process:

1. Write down a function table showing desired 0/1 inputs
2. Construct a Boolean algebra expression
   • term for each 1 in the column
   • sum (or) them to get all 1s
3. Simplify the expression using equivalences
4. Translate Boolean algebra expression to a circuit
# Last Class: 1-bit Binary Adder

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

![Truth Table and Logic Equation](image)

<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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>

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

\[ C_{\text{OUT}} = A' \cdot B \cdot C_{\text{IN}} + A \cdot B' \cdot C_{\text{IN}} + A \cdot B \cdot C_{\text{IN}}' + A \cdot B \cdot C_{\text{IN}} \]
The theorems of Boolean algebra can simplify expressions
– e.g., full adder’s carry-out function

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

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

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

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

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

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

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

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

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

Adding extra copies of the same term lets us reuse it for simplification.
1-Bit Adder with XOR gates allowed

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

No Boolean algebra simplifications possible … but $\text{Sum} \equiv (A \oplus B) \oplus C_{IN}$
Given a truth table:

2. Write the Boolean expression
3. Simplify (“minimize”) the Boolean expression
4. Draw as gates
5. Map to available gates

\[
F = A'BC' + A'BC + AB'C + ABC \\
= A'B(C' + C) + AC(B' + B) \\
= A'B + AC
\]
Multi-bit Ripple-Carry Adder

1-Bit Adder

A → B → Cout
A → B → Sum
A → B → Cin

Sum

Cout → Cin

A2 → B2 → Sum2
A1 → B1 → Sum1
A0 → B0 → Sum0

Cout → Cin
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
   - 001 → A’B’C
   - 011 → A’BC
   - 101 → AB’C
   - 110 → ABC’
   - 111 → ABC

Don’t simplify!
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)

<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>

\[ F = \text{Read } F \text{ rows off truth table} \rightarrow \text{Negate all bits} \rightarrow \text{Multiply the maxterms together} \rightarrow \text{Convert to Boolean Algebra} \]

F = 0 0 0 0 0 1 1 1 0 0 1 1 1 0 1 1
Product-of-Sums Canonical Form

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

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

1. Read F rows off truth table
2. Negate all bits
3. Convert to Boolean Algebra
4. Don’t simplify!

<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: Why does this procedure work?

Useful Facts:

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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>

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

Useful Facts:

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

<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>0</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>

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

Taking the complement of both sides...
\[ (F')' = (A'B'C' + A'BC' + AB'C')' \]

And using DeMorgan/Complement...
\[ F = (A'B'C')' (A'BC')' (AB'C')' \]
\[ = (A'' + B'' + C'')(A'' + B' + C'')(A' + B'' + C'') \]
\[ = (A + B + C)(A + B' + C)(A' + B + C) \]
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>
</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+B’+C</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A+B’+C’</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A’+B+C</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A’+B+C’</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A’+B’+C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A’+B’+C’</td>
</tr>
</tbody>
</table>

F in canonical form:

\[ F(A, B, C) = (A + B + C) (A + B’ + C) (A’ + B + C) \]

canonical form ≠ minimal form

\[ F(A, B, C) = (A + B + C) (A + B’ + C) (A’ + B + C) \]

\[ = (A + B + C) (A’ + B + C) \]

\[ = (A + C) (B + C) \]
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 $x^3 + y^3 \neq z^3$.”
• **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
Predicate

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

\[
\begin{align*}
\text{Cat}(x) & \iff \text{“x is a cat”} \\
\text{Prime}(x) & \iff \text{“x is prime”} \\
\text{HasTaken}(x, y) & \iff \text{“student x has taken course y”} \\
\text{LessThan}(x, y) & \iff \text{“x < y”} \\
\text{Sum}(x, y, z) & \iff \text{“x + y = z”} \\
\text{GreaterThan5}(x) & \iff \text{“x > 5”} \\
\text{HasNChars}(s, n) & \iff \text{“string s has length n”}
\end{align*}
\]

Predicates can have varying numbers of arguments and input types.
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”
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 \( x \)”
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></th>
<th>Integers</th>
<th>Odd Integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall x \text{ Odd}(x) )</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>( \forall x \text{ LessThan4}(x) )</td>
<td>True</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></th>
<th>{1, 3, -1, -27}</th>
<th>Integers</th>
<th>Positive Multiples of 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists x ) Odd(x)</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>( \exists x ) LessThan4(x)</td>
<td>True</td>
<td>True</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>Positive Integers</th>
</tr>
</thead>
</table>

Predicate Definitions

<table>
<thead>
<tr>
<th>Even(x)</th>
<th>“x is even”</th>
<th>Greater(x, y)</th>
<th>“x &gt; y”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odd(x)</td>
<td>“x is odd”</td>
<td>Equal(x, y)</td>
<td>“x = y”</td>
</tr>
<tr>
<td>Prime(x)</td>
<td>“x is prime”</td>
<td>Sum(x, y, z)</td>
<td>“x + y = z”</td>
</tr>
</tbody>
</table>
Determine the truth values of each of these statements:

\[ \exists x \text{ Even}(x) \]

\[ \forall x \text{ Odd}(x) \]

\[ \forall x (\text{Even}(x) \lor \text{Odd}(x)) \]

\[ \forall x \text{ Greater}(x+1, x) \]

\[ \exists x (\text{Even}(x) \land \text{Prime}(x)) \]
<table>
<thead>
<tr>
<th>Domain of Discourse</th>
<th>Positive Integers</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Predicate Definitions</th>
<th>Even(x) ::= “x is even”</th>
<th>Greater(x, y) ::= “x &gt; y”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Odd(x) ::= “x is odd”</td>
<td>Equal(x, y) ::= “x = y”</td>
</tr>
<tr>
<td></td>
<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:

- ∃x Even(x)  \[\text{T}\] e.g. 2, 4, 6, ...
- ∀x Odd(x)  \[\text{F}\] e.g. 2, 4, 6, ...
- ∀x (Even(x) ∨ Odd(x))  \[\text{T}\] every integer is either even or odd
- ∃x (Even(x) ∧ Odd(x))  \[\text{F}\] no integer is both even and odd
- ∀x Greater(x+1, x)  \[\text{T}\] adding 1 makes a bigger number
- ∃x (Even(x) ∧ Prime(x))  \[\text{T}\] Even(2) is true and Prime(2) is true
Statements with Quantifiers

<table>
<thead>
<tr>
<th>Domain of Discourse</th>
<th>Positive Integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate Definitions</td>
<td></td>
</tr>
<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>

Translate the following statements to English

∀x ∃y Greater(y, x)

∀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”</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

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

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

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

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

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

For every positive integer \( x \), there is a pos. int. \( y \) such that \( y > x \) and \( y \) is prime.

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

For each positive integer \( x \), if \( x \) is prime, then \( x = 2 \) or \( x \) is odd.

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

There exist positive integers \( x \) and \( y \) such that \( x + 2 = y \) and \( x \) and \( y \) are prime.
Statements with Quantifiers (Natural Translations)

<table>
<thead>
<tr>
<th>Domain of Discourse</th>
<th>Positive Integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate Definitions</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>
</tbody>
</table>

Translate the following statements to English

∀ x ∃ y Greater(y, x)  
There is no greatest positive integer.

∀ x ∃ y Greater(x, y)  
There is no least positive integer.

∀ x ∃ y (Greater(y, x) ∧ Prime(y))  
For every positive integer there is a larger number that is prime.

∀ x (Prime(x) → (Equal(x, 2) ∨ Odd(x)))  
Every prime number is either 2 or odd.

∃ x ∃ y (Sum(x, 2, y) ∧ Prime(x) ∧ Prime(y))  
There exist prime numbers that differ by two.”
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”

“Red cats like tofu”

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

Domain of Discourse

Mammals

Predicate Definitions

<table>
<thead>
<tr>
<th>Cat(x) ::= “x is a cat”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red(x) ::= “x is red”</td>
</tr>
<tr>
<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

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

**Domain of Discourse**
- Mammals

- “Red cats like tofu”
  - When putting two predicates together like this, we use an “and”.
  - When there’s no leading quantification, it means “for all”.

- “Some red cats don’t like tofu”
  - “Some” means “there exists”.
  - When restricting to a smaller domain in a “for all” we use implication.
  - When restricting to a smaller domain in an “exists” we use and.
Negations of Quantifiers

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

(*) \( \forall x \text{ PurpleFruit}(x) \) (“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”

(*) ∀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”

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

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”

\[ \neg \exists x \ \forall y \ (x \geq y) \]

\[ \equiv \ \forall x \ \neg \forall y \ (x \geq y) \]

\[ \equiv \ \forall x \ \exists y \ (x < y) \]

“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 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:**  
\[ \text{NotLargest}(x) \equiv \exists y \text{ Greater (y, x)} \equiv \exists z \text{ Greater (z, x)} \]

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

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