Lecture 5: DNF, CNF and Predicate Logic

xkcd
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

\[ A + B \]

<table>
<thead>
<tr>
<th>( A )</th>
<th>( B )</th>
<th>( S )</th>
<th>(( C_{OUT} ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(with ( C_{OUT} = 0 ))</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(with ( C_{OUT} = 0 ))</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(with ( C_{OUT} = 0 ))</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(with ( C_{OUT} = 1 ))</td>
</tr>
</tbody>
</table>

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

<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>

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

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

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

\[
\begin{align*}
&= A' B \text{ Cin} + A B' \text{ Cin} + A B \text{ Cin}' + A B \text{ Cin} \\
&= A' B \text{ Cin} + A B \text{ Cin} + A B' \text{ Cin} + A B \text{ Cin}' \\
&= (A' + A) B \text{ Cin} + A B' \text{ Cin} + A B \text{ Cin}' + A B \text{ Cin} \\
&= (1) B \text{ Cin} + A B' \text{ Cin} + A B \text{ Cin}' + A B \text{ Cin} \\
&= B \text{ Cin} + A B' \text{ Cin} + A B \text{ Cin}' + A B \text{ Cin} \\
&= B \text{ Cin} + A B' \text{ Cin} + A B \text{ Cin} + A B \text{ Cin}' + A B \text{ Cin} \\
&= B \text{ Cin} + A (B' + B) \text{ Cin} + A B \text{ Cin}' + A B \text{ Cin} \\
&= B \text{ Cin} + A (1) \text{ Cin} + A B \text{ Cin}' + A B \text{ Cin} \\
&= B \text{ Cin} + A \text{ Cin} + A B (\text{ Cin}' + \text{ Cin}) \\
&= B \text{ Cin} + A \text{ Cin} + A B (1) \\
&= B \text{ Cin} + A \text{ Cin} + A B
\end{align*}
\]
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 \( \text{Sum} \equiv (A \oplus B) \oplus C_{IN} \)
Mapping Truth Tables to Logic Gates – extra step

Given a truth table:

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

<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>0</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>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>0</td>
</tr>
<tr>
<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
Multi-bit Ripple-Carry Adder
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

Don’t simplify!

Add the (min)terms together

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

<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>

1. Read T rows off truth table
2. Convert to Boolean Algebra
3. Add the (min)terms together

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

F =

1. Read F rows off truth table
2. Negate all bits
3. Convert to Boolean Algebra
4. Multiply the maxterms together

F =
# 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. **Multiply the maxterms 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>

Don’t simplify!
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 \(F'\)

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

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

Taking the complement of both sides...

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

And using DeMorgan/Comp....

\[
F = (A'B'C')' (A'BC')' (AB'C')' \\
= (A'' + B'' + C'')(A'' + B' + C'')(A' + B'' + C'') \\
= (A + B + C)(A + B' + C)(A' + B + C)
\]
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>
<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>= (A + B + C) (A + B’ + C) (A’ + B + C)</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>= (A + B + C) (A + B’ + C)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A’+B+C’</td>
<td>(A + B + C) (A’ + B + C)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A’+B’+C</td>
<td>= (A + C) (B + C)</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
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.
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>Trues</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Trues</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

- \( \forall x \; \text{Odd}(x) \)
- \( \forall x \; \text{LessThan4}(x) \)
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></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>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>
</tbody>
</table>
Statements with Quantifiers

Predicate Definitions

<table>
<thead>
<tr>
<th>Domain of Discourse</th>
<th>Even(x) ::= “x is even”</th>
<th>Greater(x, y) ::= “x &gt; y”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Integers</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:

\[ \exists x \text{ Even}(x) \]
\[ \forall x \text{ Odd}(x) \]
\[ \forall x (\text{Even}(x) \lor \text{Odd}(x)) \]
\[ \exists x (\text{Even}(x) \land \text{Odd}(x)) \]
\[ \forall x \text{ Greater}(x + 1, x) \]
\[ \exists x (\text{Even}(x) \land \text{Prime}(x)) \]
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>

Predicate Definitions

Determine the truth values of each of these statements:

- **∃x Even(x)**
  - T
  - e.g. 2, 4, 6, ...

- **∀x Odd(x)**
  - F
  - e.g. 2, 4, 6, ...

- **∀x (Even(x) ∨ Odd(x))**
  - T
  - every integer is either even or odd

- **∃x (Even(x) ∧ Odd(x))**
  - F
  - no integer is both even and odd

- **∀x Greater(x+1, x)**
  - T
  - adding 1 makes a bigger number

- **∃x (Even(x) ∧ Prime(x))**
  - 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>
</table>

<table>
<thead>
<tr>
<th>Predicate Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Even(x) ::= “x is even”</td>
</tr>
<tr>
<td>Odd(x) ::= “x is odd”</td>
</tr>
<tr>
<td>Prime(x) ::= “x is prime”</td>
</tr>
<tr>
<td>Greater(x, y) ::= “x &gt; y”</td>
</tr>
<tr>
<td>Equal(x, y) ::= “x = y”</td>
</tr>
<tr>
<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 all \( x \) there is a \( y \) that is greater than \( x \)
- For every positive integer there is a larger integer

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

- False
- For every positive integer there is a smaller integer

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

- False
- For every positive integer there is a larger integer that is prime

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

- True
- Every prime is either 2 or is odd

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

- False
- There is no prime number such that there is another prime exactly 2 larger
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>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>

Translate the following statements to English

\(\forall x \, \exists y \) Greater(y, x)

There is no greatest positive integer.

\(\forall x \, \exists y \) Greater(x, y)

There is no least positive integer.

\(\forall x \, \exists y \) (Greater(y, x) \land Prime(y))

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

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

Every prime number is either 2 or odd.

\(\exists x \, \exists y \) (Sum(x, 2, y) \land Prime(x) \land Prime(y))

There exist prime numbers that differ by two.”
“Red cats like tofu”
\[ \forall x \left( (\text{Cat}(x) \land \text{Red}(x)) \rightarrow \text{LikesTofu}(x) \right) \]

“Some red cats don’t like tofu”