When we discussed propositional logic, we ran into a roadblock representing repeating data.
Consider the Wumpus World:
Propositional: Each square needs a similar set of variables to represent all possible states
First Order: $\forall$r Pit(r) $\implies$ [$\forall$ s Adjacent(r, s) $\implies$ Breezy(s)]
First order logic is thus much more expressive and succinct; Our main challenge will be computational.

Consider the representation of a world:
Propositional Logic:

- Variables {A, ...}
- Values {T, F}

First Order Logic:

- Objects with Properties
- Relationships Between Objects
- Functions Mapping Objects to Objects

How would we represent each of the following in first-order logic?

1. One plus one equals two
   - Objects: One, One plus one, two
   - Properties: None
   - Relations: equals
   - Functions: plus
2. Squares adjacent to the Wumpus are smelly
   - Objects: Squares, Squares adjacent to the Wumpus, Wumpus
   - Properties: Smelly
   - Relations: adjacent
   - Functions: None

Syntax

The syntax of first order logic is much nicer than that of propositional logic

- constants — uppercase words that represent objects
  Examples include Z, Jack, UCLA, etc
- predicates — lowercase words that represent relations
  Examples include adjacent(), at(), etc
- property — single argument predicate
- equality — a key subset of predicates
- functions — lowercase words that give a value for each input
  Examples include leftLeg(), father(), etc

These are domain specific and form our "vocabulary".
Our domain-independent vocabulary is as follows:

- variables: x, y, z
- connectives: $\vee \wedge \neg \implies \iff$
- quantifiers: $\forall$ $\exists$

We can use all of these to define atomic sentences.
These are of the form: predicate (Term1, . . . , TermN)
Term $\equiv$ a constant, variable, or function
Ground term $\equiv$ term with no variables

The new operators in first-order logic are called **quantifiers**. Quantification comes in two forms

1. **UNIVERSAL QUANTIFICATION**
   FORM: $\forall$ variables sentence
   ex. $\forall x at(x, UCLA) \implies smart(x)$ is a predicate – at(x, UCLA) is a relation – smart(x) is a property
   This forms a conjunction of the instantiations of the predicate, and often appears with '$\implies$'
   $\rightarrow$ [at(John, UCLA $\implies$ smart(John)] $\wedge$ [at(fatherOf(John), UCLA) $\implies$ smart(fatherOf(John))]

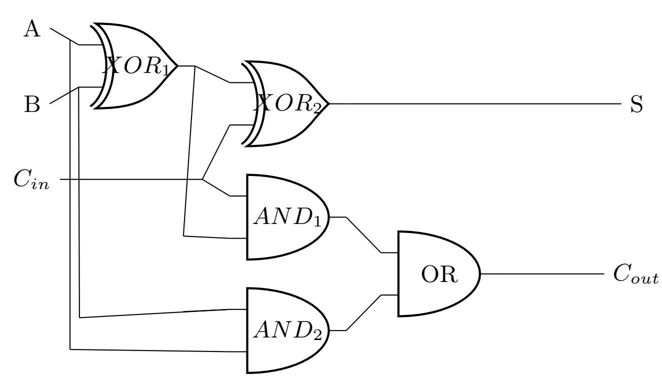
2. **EXISTENTIAL QUANTIFICATION**
   FORM: ∃ variables statement
   ex ∃ x at(x, UCLA) ∧ tall(x)
   This forms a disjunction of the instantiations of the predicate, and often appears with '∧'
   → [at(John, UCLA) ∧ tall(John)] ∨ [at(fatherOf(John), UCLA) ∧ tall(fatherOf(John))]

Quantification is not always commutative:
$\exists\ x\ \exists\ y = \exists\ y\ \exists\ x$
$\forall\ x\ \forall\ y = \forall\ y\ \forall\ x$
$\forall\ x\ \exists\ y! = \exists y\ \forall\ x$
Why? consider:
$\forall\ x\ \exists$ y loves(y, x) means "everyone in the world has at least one person who loves them"
$\exists\ y\ \forall$ x loves(y, x) means "there is at least one person who loves everyone in the world"
We can, however, use one operator to simulate the other:
¬∀ x likes (x, IceCream) = $\exists x\neg$likes (x, IceCream)

Asserting a number of unifications tends to be a bit trickier
"Spot has two sisters"
→ $\exists\ x\ \exists$ y sister(x, spot) ∧ sister(y, spot) ∧ x != y
"Spot has exactly two sisters"
→ We use the above statement, plus ∀ z sister(z, spot) $\implies$ ((z = x) ∨ (z = y))
Which can also be written as ¬(∃ z sister(z, spot) ∧ ((z = x) ∨ (z = y)))
Some people make this cleaner by using to represent "exists a unique"
∃! x king(x) = [∃ x king(x)] ∧ [∀ y king(y) $\implies$ (x = y)]
(this is an error in actuality, since the x on the left is out of scope)

We can see it is critical to develop well formed formulas with no free variables.

Consider the 1-bit adder:
We want to derive the output given the input.
This is hard to do with circuits, but this is easy with first order logic.



Our vocabulary consists of:
- domain:
  - constants: AND, OR, NOT, XOR, 0, 1
  - functions: type(g), signal(i, o), in(g), out(g)
  - predicates: connected(g_1, g_2)
- instance (the specific layout of this circuit):
  - constants: $XOR_1, XOR_2, AND_1, AND_2, OR_1$

We then define our knowledge base:
- Domain:
  - ∀ t1, t2 connected(t1, t2) $\implies$ (signal(t1) = signal(t2))
  - ∀ t1, t2 connected(t1, t2) $\iff$ connected(t2, t1)
  - ∀ g type(g) = OR $\implies$ [signal(out(1, g)) = 1 $\iff$ ∃ n signal(in(n, g)) = 1]
    - (similar rules for other gates are omitted for the sake of brevity)
  - (the most general part follows)
  - ∀ t signal(t) = 1 ∨ signal(t) = 0
  - ¬1 = 0
- Instance:
  - type(XOR_1) = XOR, type(XOR_2) = XOR, ...
  - connected(out(1, XOR_1), in(2, AND_2)), ...

This is all we need to begin queries!

$\exists i_1, i_2, i_3$ signal(in(1, adder)) $= 1$
$= i_1 \wedge$ signal(in(2, adder))
$= i_2 \wedge$ signal(in(3, adder))
$= i_3 \wedge$ signal(out(1, adder))
$= 0 \wedge$ signal(out(2, adder))
$\rightarrow$ {(1, 1, 0), (1, 0 1), (0, 1, 1)}

We could expand this to check if a circuit is functioning & to diagnose errors with:
ok(g) — represents whether the circuit is ok
stuck(g) — represents whether a gate is always off and stuck on 0
We may even want to include wires if that is what we're testing.
All of this is called Knowledge Engineering.