First Order Logic

Last week we looked at Propositional Logic and the knowledge bases (KBs) we used to reason about our intelligent agents' environments.

These KBs expressed its knowledge of facts using propositional, Boolean variables and logical operators that connected them to make sentences.

Whenever we wanted to express a fact or atom of knowledge, we would then create a new propositional variable:

```
; Propositional Variables:
Let R = Whether or not it is raining,
    L = Whether or not the sun is shining,
    S = Whether or not the sidewalk is wet,
...
```

Once we had expressed some relationships between our variables (e.g., "if it is raining, then the sidewalk will be wet": R ⇒ S) we were allowed us to ask simple questions of our KB.

These questions were still propositional sentences like: α = "Is the sidewalk wet?"

Seemed pretty nifty! So what does First Order Logic do for us? Well consider the following facts we might want to express in our KB:

```
Let P = Whether or not Andrew likes Pasta,
    F = Whether or not Andrew likes Firefly,
    R = Whether or not the class likes Andrew's references,
...
```

> ⍺ What do we notice about the format of these propositional variables?

The idea of abstraction is difficult for propositional logic because with each new fact (even of a similar format to others), I need to create a new variable!

The other issue with propositional variables is with handling facts I need to model that I don't necessarily know at "compile time" (i.e., when I'm making my variables).

So, First Order Logic offers some solutions to these horrible quandries!

First Order Logic Syntax

Just as propositional logic had syntax and semantics, so does First Order Logic build upon them.

> ⍺ **Objects** are simply nouns and noun phrases that refer to... well... pretty much any noun our KB wants to talk about.

```
; Object Examples:
People
Places
The Republic of Forns
The Wet Sidewalk
Rain
Numbers
...
```

> ⍺ **Relations** are simply verbs and verb phrases that establish descriptions of (unary relations) or relationships between (n-ary relations) objects.

```
; Unary Relations in format:
; relation (object)
; AKA predicates / properties
primeNumber(3)
red(balloon)
utopia(The Republic of Forns)
usesOverheadProjector(Dyer)

; Binary Relations in format:
; relation(obj1, obj2)
greaterThan(5, 3)
teaches(Andrew, CS161)
likes(Andrew, pasta)
```

Relations like the above are called **atomic sentences** because they express a single truth in our KB.

> ⍺ We hold that atomic sentences in the KB are **true** and are used as axioms for the objects on which they establish relationships.

> ⚘ Some relations have a unique output for a given input, and are termed **functions**

```
; Functions are relations that map
; inputs to unique output
fatherOf(Andrew) ; I have one father
firstWordOf("This sentence") ; One first word
rulerOfAndDuring(England, 1200) ; King John
```

> ⚘ **Connectives** are the same logical operators we used in propositional logic and are used to create FOL sentences.

```
; Connectives (∧, ∨, ¬, ⇒)
¬hasSiblings(Andrew)
¬isFatherOf(Sean Connery, Andrew) ;_;
greaterThan(5, 3) ∧ greaterThan(3, 1)
isRoyal(Queen Elizabeth) ∨ ¬isRoyal(Lorde)
```

> ⚘ **Complex Sentences** are any FOL sentence that uses logical connectives or requires a function evaluation to ground an object.

```
; Complex Sentences
(any of the above examples)
¬adjacentTo(49thState(USA), Russia)
```

But now comes the truly expressive power of First Order Logic: the capacity to have relations with variable placeholders for objects.

> ⚘ **Variables** are placeholders for objects in a given relation that can be bound during inference. They are used to establish general facts and rules.

However, since FOL deals with objects, we want some means of expressing the notion of object collections (rather than listing individual objects and combining by some operator)!

For this reason, we never use variables on their own, but instead always accompany them with a quantifier.

```
; Philosophy 101
; For variable x:
∀x person(x) ⇒ mortal(x)
person(Socrates)

; ... you know the rest :)
```

So, the above quantification reads, "For all x, if x is a person, then x is mortal."

You can think of universal quantifiers as being used to express a conjunction of sentences such that:

```
∀x relation(...x...) → relation(...term1...) ∧ relation(...term2...) ∧ relation(...
term3...) ∧ ...
```

```
; Ground vs. Not Ground
person(Socrates) ; is a ground term
∃x brother(John Snow, x) ; is NOT a ground term (variable x)
```

```
; For variable x:
∃x likes(x, Nickelback)
; "There exists an x such that x likes Nickelback"
; (well... maybe)
```

You can think of existential quantifiers as being used to express a disjunction of sentences such that:

```
∃x relation(...x...) → relation(...term1...) ∨ relation(...term2...) ∨ relation(...t
erm3...) ∨ ...
```

> ⌐ **Nested quantifiers** can quantify multiple variables, and can either be homogenous (same quantifer) or heterogenous (different quantifiers).

```
; Homogenous nested quantifier
∃x, y marriedTo(x, y)
= ∃x ∃y marriedTo(x, y)
; "There exist an x and a y such that x is married to y"

; Heterogenous nested quantifier examples:
∀x ∃y loves(x, y)
; "Everybody loves somebody"

∀y ∃x loves(x, y)
; "Somebody loves everybody"

∀y ∃x loves(y, x)
; "There is someone who is loved by everyone"
```

One important thing to note is that ORDER MATTERS with nested quantification!

Furthermore, we observe that position within a relation is relevant as well! Just as a function expects arguments input in a certain order, so do relations.

> ⌐ The notion of **duality** gives us rules for negating sentences with quantification such that:

```
; For variable x and relation R:

∃x R ⟺ ¬∀x ¬R  ≡  ∀x R ⟺ ¬∃x ¬R
```

One way to think about duality is that if we're negating the universal quantifier, we're saying, "Not all x have relation R," which is equivalent to saying, "At least some x have relation R." (the first of the two expressions above)

Similarly, by negating the existential quantifier, we're saying "There isn't even one x without relation R," which is equivalent to saying, "All x have relation R." (the second of the two expressions above)

> **Example**
> t Express the following sentence as an existential quantification.

```
¬ ∀x, y (friends(x, y) ⇒ knows(x, y))
```

> ₫ The **equality** symbol signifies that two terms refer to the same object.

```
Father(Andrew) = Mark

; Can even use to denote uniqueness
Father(Andrew) = ¬Father(Bob)
; Meaning that Andrew and Bob are not siblings
; (they don't have the same father)
```

First Order Logic Semantics

Now that we know the syntax of FOL, we want to again move towards the goal of creating KBs with FOL sentences, and then perform inference using our new tricks!

The first issue we have to deal with is our use of variables in FOL relations, which exibit "templates" to ground objects within.

For example, we may have the template for knowledge that:

```
∀x, y (friends(x, y) ⇒ knows(x, y))
```

...but if we want to express that Andrew and Bob are friends, we need a way to **ground** our values for x and y.

> ₫ **Unification** is the process of binding variables to values such that for two sentences α and β and unifier Θ, we have:
>
> $\alpha\Theta = \beta\Theta$

Unifying variables to values allows us to perform inference on ground terms and take our general knowledge rules and apply them to our intelligent agent's environment.

The gist being that we want to make "different" logical expressions look equivalent when we want to move from general statements to specific ones.

The syntax for unification is as follows (NB: sometimes commas and semicolons are used to separate clauses in a CNF KB (as seen below)):

```
; Unification syntax:
Θ = { var1/object1, var2/object2, ... }
; Which says, "var1 is unified to object1, var2 is unified to object2, ..."

; Example 1:
∀x person(x) ⇒ mortal(x); person(Socrates)
Θ = { x/Socrates }

; Example 2:
∀x, y parent(x, motherOf(y)) ⇒ grandparent(x, y); parent(Carter, z)
; "For all x and y, if x is the parent of the mother of y, then x is the grandparent
of y."
; "Also, Carter is the parent of some child z."
Θ = { x/Carter, z/motherOf(Stewie) }

; Example 3:
Knows(John, x); Knows(y, Bill)
Θ = { x/Bill, y/John }

; Example 4:
Knows(John, x); Knows(x, Bill)
Θ = CANNOT UNIFY
; (x would need to be 2 different values to work)
; Solution: give one of the x's a unique name (see below)
```

That said, the rules for unifying universally quantified sentences are different than unifying those of existentially quantified sentences.

Because universal quantification describes a general rule true for all variables is lists, existential quantification needs but one instantiation of its variables.

So, let's start by looking at universal instantiation and then compare it to existential.

- **Universal Instantiation** says we can infer any sentence obtained by substituting a ground term for a variable in the sentence, expressed:

$$\forall v \ \alpha$$

$$SUBST(\{ v/g \}, \alpha)$$

...which simply reads, "For all instantiations of some variable v in α, unify v on some ground term g in α and add that to the Knowledge Base."

- NOTE: If we perform a substitution specified above, we remove the quantifier from the sentence, because it now represents a ground term.

- Furthermore, because our KBs are in CNF, we can list clauses in our KBs with universal quantifiers dropped BUT NOT with existential quantifiers dropped, simply because universal quantification is assumed in clauses (i.e., our clauses are general truths).

Instantiation for existentially quantified sentences is slightly more involved, but also somewhat intuitive.

We add the constraint that unified existentially quantified variables must be uniquely named (i.e., appearing nowhere else in the KB).

- The process of unifying on an existentially quantified variable is a special case of a process known as **Skolemization.**

- A **Skolem constant** is a new, unique name given to our term that appears no where else in the KB.

Why do we care that existential quantification has to unify on a Skolem constant? The book puts it nicely when it says:

> Whereas Universal Instantiation can be applied many times to produce many different consequences, Existential Instantiation can be applied once, and then the existentially quantified sentence can be discarded. For example, we no longer need ∃x Kill(x, Victim) once we have added the sentence Kill(Murderer, Victim).

An example Skolemization might use the functional notation such that:

```
; Example 1:

∀x English(x) ⇒ ∃y [Tea(y) ∧ Likes(x, y)]

; Since ∀x applies to the entire sentence, and we need a
; unique name for y (an existentially quantified variable), we
; simply replace all instances of y with a Skolem *function*, F(x):

English(x) ⇒ [Tea(F(x)) ∧ Likes(x, F(x))]
; Remember to drop the quantifiers after!


; Example 2:

∃y Dog(y) ∧ Owns(Jack, y)

; Since we DO NOT have a universal quantifier that applies to the
; whole sentence, we need not Skolemize as a function of some
; other variable. So instead, we'll replace y above with some Skolem
; *constant* name that appears nowhere else in the KB, let's say D:

Dog(D) ∧ Owns(Jack, D)
; Remember to drop the quantifiers after!
```

As such, our new unification rule for existentially quantified sentences is:

> ⍺ **Existential Instantiation** says we can infer any sentence obtained by substituting a Skolem constant for a variable in the sentence, expressed:
>
> $$∃v\ α$$
>
> ---
>
> $$\text{SUBST}(\ \{\ v/k\ \},\ α\ )$$

ά   The process by which we turn our templated FOL KB into ground terms using unification and instantiation is known as **propositionalization.**

This is an intuitive definition because we see that by providing concrete instantiations of our general rules, we're essentially transforming our FOL KB into a grounded propositional one!

This process of propositionalization is powerful because the FOL KB (with infinite generative capacity) can now benefit from the inference capacities we talked about for propositional KBs.

```
; Example KB
KB =
    1. (King(x) ∧ Greedy(x)) ⇒ Evil(x)
    2. King(John)
    3. Greedy(John)

; Then, using universal instantiation (assumed due to clauses),
; we can add the following sentence:
    4. (King(John) ∧ Greedy(John)) ⇒ Evil(John) [Θ = x/John]
```

Notice how examples like the above turn our FOL KB into what is essentially a propositional one (I could have just had propositional variables like K = Whether or not John is a king...)

There's one problem though...

ά   Every statement entailed by a FOL KB is entailed by (a finite subset of) the PL KB that is the result of propositionalizing the FOL one.

ά   **Decidability** is a property of reasoning systems that asks if any arbitrary sentence can be shown to be entailed by the KB or not.

This concept is nice and an expected application, but throws one wrench in the gears: how do we know how many nested terms were used to propositionalize a sentence from our FOL KB?

What if I had a bunch of generated sentences such that:

```
∀x Father(x) ⇒ ...
Father(John)

; Then I could generate symbols:
Father(Father(John)) ⇒ ...
Father(Father(Father(John))) ⇒ ...
Father(Father(Father(Father(John)))) ⇒ ...
...
; Where do I stop?!
```

If our FOL KB has a finite propositionalization (i.e., it propositionalizes to a finite number of unified sentences), then entailment is decidable.

However, due to the infinite generative capacity of FOL, we might not be capable of propositionalizing to some finite number of unified sentences, so entailment is said to be **semi-decidable**

> ά   The property of **semidecidability** says that our FOL inference algorithms can determine whether a sentence is entailed by our FOL KB, but it cannot also say that every non-entailed sentence it encounters is not entailed by the KB.

> **Example**
> t   There's a way to get around this possible infinite symbol generation (although the algorithm will still be semi-decidable) that is akin to a tactic we used for limiting DFS. Can you think of it?

That said, let's look at our refutation strategy from PL KBs and see how they work with FOL KBs.

First Order Inference

The rules of FOL inference are not very different from those of propositional logic (due to our propositionalization of a FOL KB).

We'll cover our two favorites from PL now:

> ά   **Generalized Modus Ponens** is an inference rule defined as:

$$p1' \wedge p2' \wedge ... \wedge pn'; \quad (p1 \wedge p2 \wedge ... \wedge pn) \Rightarrow q$$

$$SUBST(\Theta, q)$$

Generalized Modus Ponens provides a means of propositionalizing the conditions with matching terms in order to infer q.

```
; Example KB 1
KB =
    1. ∀x (King(x) ∧ Greedy(x)) ⇒ Evil(x)
    2. King(John)
    3. Greedy(John)

; Generalized Modus Ponens allows us to infer:
    4. Evil(John) [1, 2, 3; Θ = { x/John }]
; Hidden propositionalization using unifier Θ:
; (King(John) ∧ Greedy(John)) ⇒ Evil(John)

; Example KB 2
KB =
    1. ∀x (King(x) ∧ Greedy(x)) ⇒ Evil(x)
    2. King(John)
    3. ∀y Greedy(y)

; Generalized Modus Ponens allows us to infer:
    4. Evil(John) [1, 2, 3; Θ = { x/John, y/John }]
; Hidden propositionalization using unifier Θ:
; Greedy(John)
; (King(John) ∧ Greedy(John)) ⇒ Evil(John)
```

We say that Generalized Modus Ponens is **lifted** because it takes Modus Ponens (like in Propositional Logic), which consists solely of ground terms, and "lifts" it up to FOL whereby we need only make unifications where necessary for inference.

So far, we haven't touched upon the format of our FOL KB clauses, but ideally, for resolution, we'd like to keep our KB in conjunctive normal form.

As such, it pays to know how to turn any arbitrary sentence into one that can be expressed in CNF.

For FOL, this is a bit trickier than the few logical operator manipulations we saw for PL, and consists of the steps on page 346 of your text (which you should review!)

The general steps are:

1. Remove implications (get sentences with only ¬, ∧, and ∨)

2. Push negations in using DeMorgan's law and quantifier duality

3. Remove quantifiers, remembering to use Skolemization where appropriate:
   $\forall x \, (\exists y \, foo(y) \lor bar(x, y)) \rightarrow \forall x \, foo(F(X)) \lor bar(x, F(x))$

4. Drop universal quantifiers

ᵃ **FOL Resolution** is only different from PL resolution such that we unify the resultant clause added to the KB:

For any two sentences X and Y in our KB with objects and relations x_i and y_i of the format: $X = x1 \lor x2 \lor ... \lor xn$; $Y = y1 \lor y2 \lor ... \lor yn$

If X and Y contain some relation Z such that $Z \in X$ AND $\neg Z \in Y$, then we can combine the two sentences on everything except the disagreement on Z such that:

$$x1 \lor x2 \lor Z \lor ... \lor xn; \qquad y1 \lor y2 \lor \neg Z \lor ... \lor yn$$

$$SUBST(\Theta, x1 \lor x2 \lor ... \lor xn \lor y1 \lor y2 \lor ... \lor yn)$$

In other words, we're allowed to infer the sentence composed of all disjoined relations in X and Y *except* for element Z (and unify that result on Θ).

```
; Example resolution:
KB =
    1. ¬Owns(x, D) ∨ AnimalLover(x)
    2. Owns(Jack, D)
    3. Dog(D)

; Using resolution I can infer:
    4. AnimalLover(Jack) [1, 2, Θ = { x/Jack }]
```

First Order Resolution Inference

So now that we have the components, let's get back to our objective: running inference on our KB!

FOL resolution is similar to PL (requires a CNF KB) (PS. I've been wanting to get all of those initialisms in one sentence since I started writing this article; maximum confusion: success!)

> ά   FOL resolution is sound (guarantees any derived clause is entailed by KB) and refutation complete (if a contradiction is to exist in the KB, it will find it), though semi-decidable.

We'll perform the same steps that we used for PL inference proofs: proof by contradiction / refutation!

1. Convert KB to CNF if it isn't already.

2. Negate the query sentence, α, and add that negation to the KB.

3. Apply resolution to pairs of (unifiable) clauses, and add those resultant clauses to the KB

4. Repeat until either a contradiction is reached, or we're out of clauses to resolve!

Let's do one giant example where we start from the very start and end with a query answered!

> **Example**
> t   Read the following story and perform the following steps:
> 1. Convert the story into FOL sentences.
>
> 2. Convert the KB into CNF.
>
> 3. Use lifted inference tactics to answer the scandalous query question: Did curiosity kill the cat?

1. There exists a dog that Jack owns.

2. It is well known that if anyone owns some dog, then they are an animal lover.

3. It is also known that if anyone is an animal lover, then if that person ever encounters any animal, they would not kill that animal.

4. Either Jack killed Tuna or Curiosity killed Tuna.

5. Oh yeah... and Tuna is a cat.

6. Clearly for all entities that are cats, those entities are also animals.

α = Did Curiosity kill the cat?

```
; We'll start by converting the sentences from English to FOL
; HINT: Focus on quantifier-denoting words like "any," "exist,"
; "some," and "all"

; (1) There exists a dog that Jack owns.
1. ∃x Dog(x) ∧ Owns(Jack, x)

; (2) It is well known that if anyone owns some dog, then they are an animal lover.
2. ∀x [∃y Owns(x, y) ∧ Dog(y)] ⇒ AnimalLover(x)

; (3) It is also known that if anyone is an animal lover,
; then if that person ever encounters any animal, they would not kill that animal.
3. ∀x AnimalLover(x) ⇒ [∀y Animal(y) ⇒ ¬Kills(x, y)]

; (4) Either Jack killed Tuna or Curiosity killed Tuna.
4. Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)

; (5) Oh yeah... and Tuna is a cat.
5. Cat(Tuna)

; (6) Clearly for all entities that are cats, those entities are also animals
6. ∀x Cat(x) ⇒ Animal(x)

; α = Did Curiosity kill the cat?
α = Kills(Curiosity, Tuna)
```

Great! Now we've got our FOL sentences, but our KB isn't in CNF yet... let's convert our sentences to a usable CNF now!

```
1. ∃x Dog(x) ∧ Owns(Jack, x)
; Firstly, we notice that we have an existential quantifier, so to
; get rid of it, we'll Skolemize and choose the unique symbol D

  Dog(D) ∧ Owns(Jack, D)

; Next, we notice this actually consists of two clauses conjoined
; so we'll split it into two sentences in our KB

  KB1. Dog(D)
  KB2. Owns(Jack, D)
```

2. ∀x [∃y **Owns**(x, y) ∧ **Dog**(y)] ⇒ **AnimalLover**(x)
; **Firstly**, we'll get rid of that pesky implication arrow and convert
; everything into ∧ and ∨ statements:

  ∀x ¬[∃y Owns(x, y) ∧ Dog(y)] ∨ AnimalLover(x)
= ∀x ∀y ¬Owns(x, y) ∨ ¬Dog(y) ∨ AnimalLover(x)

; Since the existential quantification on y got negated to a universal
; quantification, we don't have to **Skolemize. Furthermore**, we now have
; a clause! **So** we can just drop the universal quantification, giving us:

  KB3. ¬**Owns**(x, y) ∨ ¬**Dog**(y) ∨ **AnimalLover**(x)

---

3. ∀x **AnimalLover**(x) ⇒ [∀y **Animal**(y) ⇒ ¬**Kills**(x, y)]
; **Let**'s start off by unpacking those implication arrows!

  ∀x AnimalLover(x) ⇒ [∀y ¬Animal(y) ∨ ¬Kills(x, y)]
= ∀x ¬AnimalLover(x) ∨ [∀y ¬Animal(y) ∨ ¬Kills(x, y)]
= ∀x ∀y ¬AnimalLover(x) ∨ ¬Animal(y) ∨ ¬Kills(x, y)

; We've got ourselves a clause! **Let**'s drop those universal quantifiers to get:

  KB4. ¬AnimalLover(x) ∨ ¬Animal(y) ∨ ¬Kills(x, y)

---

4. **Kills**(**Jack**, **Tuna**) ∨ **Kills**(**Curiosity**, **Tuna**)
; **That**'s already a clause! Nice!

  KB5. Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)

---

5. **Cat**(**Tuna**)
; **Also** already a clause! **Go Tuna**!

  KB6. **Cat**(**Tuna**)

```
6. ∀x Cat(x) ⇒ Animal(x)
; Get rid of that implication arrow!

   ∀x ¬Cat(x) ∨ Animal(x)

; Drop the universal quantifier, and we have our clause!

   KB7. ¬Cat(x) ∨ Animal(x)
```

```
α = Kills(Curiosity, Tuna)
; For our query, we'll need to add ¬α to the KB, so:
¬α = ¬Kills(Curiosity, Tuna)
; ...which is also a clause! Add it in!

   KB8. ¬Kills(Curiosity, Tuna)
```

Now that we have our KB in CNF, we can start performing resolution!

Let's gather all of our KB clauses and start:

```
KB =
    1. Dog(D)
    2. Owns(Jack, D)
    3. ¬Owns(x, y) ∨ ¬Dog(y) ∨ AnimalLover(x)
    4. ¬AnimalLover(x) ∨ ¬Animal(y) ∨ ¬Kills(x, y)
    5. Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)
    6. Cat(Tuna)
    7. ¬Cat(x) ∨ Animal(x)
    8. ¬Kills(Curiosity, Tuna)

; Begin inference!

    9.  ¬Owns(x, D) ∨ AnimalLover(x) [1, 3, Θ = { x/D }]
    10. AnimalLover(Jack) [2, 9, Θ = { x/Jack }]
    11. Animal(Tuna) [6, 7, Θ = { x/Tuna }]
    12. ¬AnimalLover(x) ∨ ¬Kills(x, Tuna) [4, 11, Θ = { y/Tuna }]
    13. ¬Kills(Jack, Tuna) [10, 12, Θ = { x/Jack }]
    14. Kills(Curiosity, Tuna) [5, 8]
    → ← Contradiction! [8, 14]

∴ KB ⊨ α
```

Inference Strategies

Remember when we talked about definite clauses with propositional logic?

Well, let's dive into why they're neat (for both propositional and FOL)!

As a reminder:

> ɑ **Definite clauses** are clauses with *exactly* one positive literal.

Here are two really cool properties about definite clauses:

> ɑ Any definite clause can be turned into an implication statement:

```
; #1
(¬X ∨ Y)
= (X ⇒ Y)

; #2
(¬Owns(x, y) ∨ ¬Dog(y) ∨ AnimalLover(x))
= ((Owns(x, y) ∧ Dog(y)) ⇒ AnimalLover(x))
```

So, naturally, this means that if our entire KB consisted of definite clauses, then we could perform Modus Ponens over and over for our inference strategy!

> ά Resolving on two definite clauses always produces another definite clause:

```
; #1
sent1 = (¬X ∨ Y)
sent2 = (¬Z ∨ X)
; Allows us to infer:
sent3 = (¬Z ∨ Y)

; #2
sent1 = (Y)
sent2 = (¬Y ∨ X)
; Allows us to infer:
sent3 = (X)
```

So, if our KB consists solely of definite clauses, then we gain two powerful inference techniques: backwards and forwards chaining.

> ά **Chaining** algorithms use KBs with only definite clauses to achieve efficient inference.

> ά **Forward Chaining** starts with the base facts and applies generalized Modus Ponens recursively until no clauses can be added. We succeed if we infer our query.

> ά **Backward Chaining** starts with our query sentence as the goal, and recursively adds sub-goals which, if proven, would prove our current target goals.

Meh, those are just high-level descriptions... let's look at an example and the steps that each tactic takes.

Let's start by defining a KB with only definite clauses (from the book):

> The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is an American. Is Colonel West a criminal in the eyes of America?

```
KB =
    1. (American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z)) ⇒ Criminal(x)
    2. Owns(Nono, M1)
    3. Missile(M1)
    4. (Missile(x) ∧ Owns(Nono, x)) ⇒ Sells(West, x, Nono)
    5. Missile(x) ⇒ Weapon(x)
    6. Enemy(x, America) ⇒ Hostile(x)
    7. American(West)
    8. Enemy(Nono, America)

α = Criminal(West)
```

Forward chaining begins by separating our FOL KB into two categories:

- **Rules:** statements that have an implication to be used by Generalized Modus Ponens

- **Facts / Axioms:** atomic statements that establish our system's knowledge of the environment

As such, we divide the above KB into:

```
KB =
   ; Rules:
   1. (American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z)) ⇒ Criminal(x)
   4. (Missile(x) ∧ Owns(Nono, x)) ⇒ Sells(West, x, Nono)
   5. Missile(x) ⇒ Weapon(x)
   6. Enemy(x, America) ⇒ Hostile(x)


   ; Facts:
   2. Owns(Nono, M1)
   3. Missile(M1)
   7. American(West)
   8. Enemy(Nono, America)
```

Now, since our KB consists only of definite clauses, forward chaining does the following:

1. Starting with your facts, unify over any rule that you can match completely and add those unified ground terms to your forward-chaining "proof tree"

2. Once you've unified all the rules you can using just your starting facts, take the new ground terms you found in step 1, add them to your facts, and try again.

3. Stop when you reach one of two conditions: (a) you derive the ground term equal to your query, or (b) you generate no new terms between iterations.

So, let's try this out on our example above!

```
; Forward Chaining Iteration 1

KB =
   ; Rules:
   1. (American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z)) ⇒ Criminal(x)
   4. (Missile(x) ∧ Owns(Nono, x)) ⇒ Sells(West, x, Nono)
   5. Missile(x) ⇒ Weapon(x)
   6. Enemy(x, America) ⇒ Hostile(x)

   ; Facts:
   2. Owns(Nono, M1)
   3. Missile(M1)
   7. American(West)
   8. Enemy(Nono, America)

; New facts discovered on iteration 1:
   9.  Weapon(M1) [Rule 3, Fact 5, Θ = { x/M1 }]
   10. Sells(West, M1, Nono) [Rule 4, Fact 2, Fact 3, Θ = { x/M1 }]
   11. Hostile(Nono) [Rule 6, Fact 8, Θ = { x/Nono }]
```

Notice that on our current iteration, we don't yet use any of the new facts we discover; these will be used in the next iteration.

Now, we start iteration 2 with the new facts that we got from last time!

```
; Forward Chaining Iteration 2

KB =
   ; Rules:
   1. (American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z)) ⇒ Criminal(x)
   4. (Missile(x) ∧ Owns(Nono, x)) ⇒ Sells(West, x, Nono)
   5. Missile(x) ⇒ Weapon(x)
   6. Enemy(x, America) ⇒ Hostile(x)

   ; Facts:
   2.  Owns(Nono, M1)
   3.  Missile(M1)
   7.  American(West)
   8.  Enemy(Nono, America)
   9.  Weapon(M1) [Rule 3, Fact 5, Θ = { x/M1 }]
   10. Sells(West, M1, Nono) [Rule 4, Fact 2, Fact 3, Θ = { x/M1 }]
   11. Hostile(Nono) [Rule 6, Fact 8, Θ = { x/Nono }]

 ; New facts discovered on iteration 2:
   12. Criminal(West) [Rule 1, Fact 7, Fact 9, Fact 10, Fact 11, Θ = { x/West, y/M1,
 z/Nono }]
   ; STOP - Found our query!

 ∴ KB ⊨ α
```

So, we terminate here since we found our query, meaning that KB ⊨ α

Pictorially, we might think of this forward chaining taking the steps:
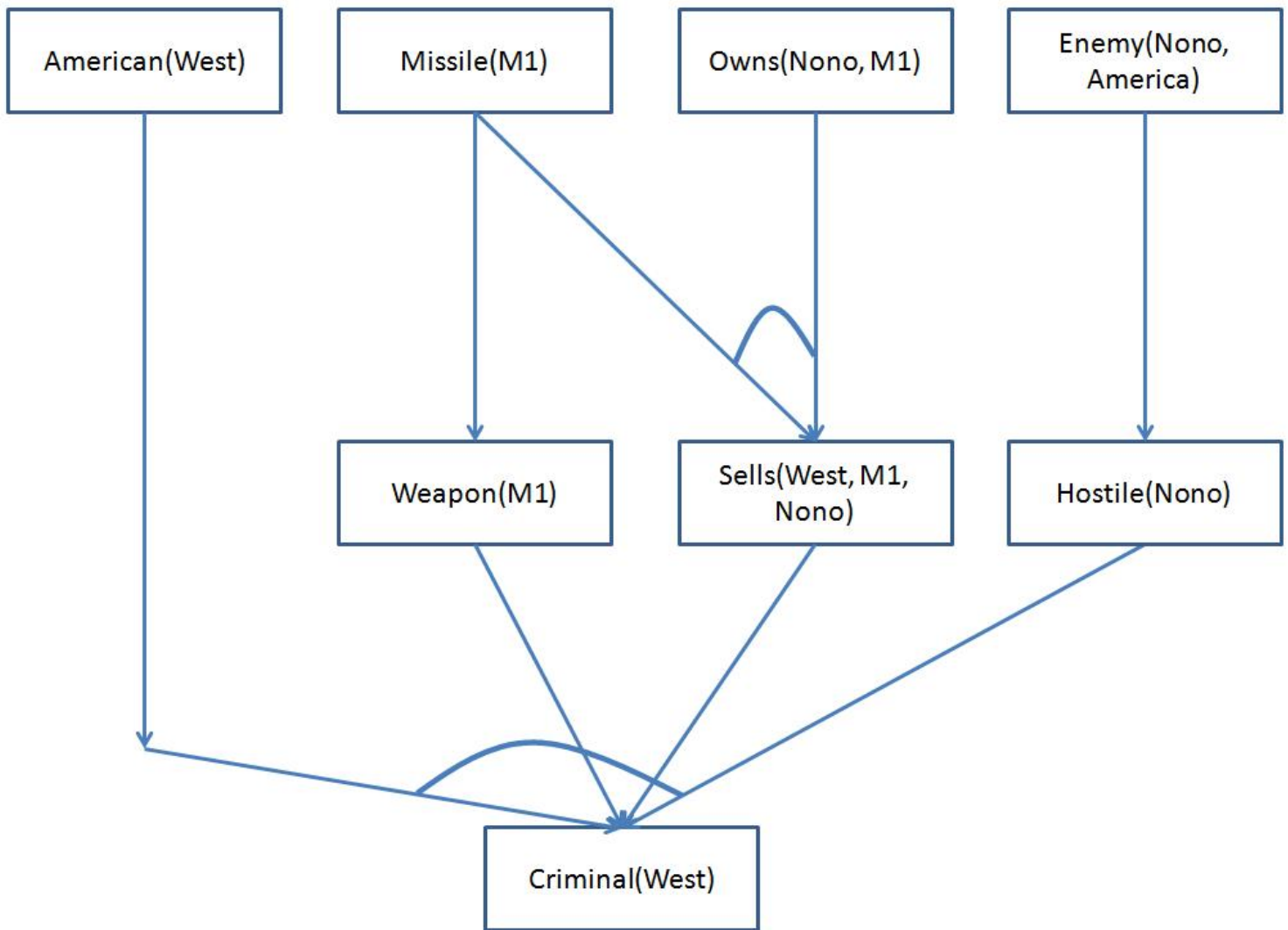
Above, the depth of our **proof tree** corresponds to the iteration level in our algorithm's steps, with depth = 1 at the top and nodes processed left-to-right.

Notice that at depth d, our gathered facts include all facts at depth < d

Arcs (as poorly illustrated as they may be) illustrate AND relationships between bindings.

Now, let's try to do the same problem over using backward chaining!

The steps of backward chaining are as follows:

1. Start with JUST our query as a goal.

2. Now, find rules in our KB whose implication's RHS contain our goals' relations.

3. Treat the LHS of any found rules' implication as our new sub-goals.

4. Recurse on our sub-goals to try and unify them to fact.

5. Any time a unification is performed for some variable, that unification holds for all other references to that variable.

6. We have succeeded to illustrate entailment of our query if all goals have been unified to ground terms.

Let's try it!

```
KB =
    1. (American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z)) ⇒ Criminal(x)
    2. Owns(Nono, M1)
    3. Missile(M1)
    4. (Missile(x) ∧ Owns(Nono, x)) ⇒ Sells(West, x, Nono)
    5. Missile(x) ⇒ Weapon(x)
    6. Enemy(x, America) ⇒ Hostile(x)
    7. American(West)
    8. Enemy(Nono, America)

α = Criminal(West)


; ============================================================================
========
; Recursive Level 1 Goals:
{ Criminal(West) } ; Start with just the query
Target goal: Criminal(West)

; Rules that contain our target goals' relations on RHS OR facts containing our goal
s:
    1. (American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z)) ⇒ Criminal(x)
       [Θ = { x/West }]

; NOTE: Unification performed above; now x/West in subsequent goal resolution

; Recurse on new goals
    New Goals: American(West), Weapon(y), Sells(West, y, z), Hostile(z)

; ============================================================================
========
; Recursive Level 2 Goals:
{ American(West), Weapon(y), Sells(West, y, z), Hostile(z) }
Target goal: American(West)

; Rules that contain our target goals' relations on RHS OR facts containing our goal
s:
    7. American(West) [Θ = {}] ; Done! Satisfied this goal! Remove it.

; ============================================================================
```

```
========
  ; Recursive Level 2 Goals:
  { Weapon(y), Sells(West, y, z), Hostile(z) }
  Target goal: Weapon(y)

  ; Rules that contain our target goals' relations on RHS OR facts containing our goal
s:
    5. Missile(x) ⇒ Weapon(x) [Θ = { x/y }]

  ; Recurse on new goals
    New Goals: Missile(y)


  ; ===============================================================================
========
  ; Recursive Level 3 Goals:
  { Missile(y) }
  Target goal: Missile(y)

  ; Rules that contain our target goals' relations on RHS OR facts containing our goal
s:
    3. Missile(M1) [Θ = { y/M1 }] ; Done! Satisfied this goal! Remove it

  ; NOTE: Unification performed above; now y/M1 in subsequent goal resolution

  ; Out of goals for this level, so return success to previous level!


  ; ===============================================================================
========
  ; Recursive Level 2 Goals:
  { Sells(West, M1, z), Hostile(z) }
  Target goal: Sells(West, M1, z)

  ; Rules that contain our target goals' relations on RHS OR facts containing our goal
s:
    5. (Missile(M1) ∧ Owns(Nono, M1)) ⇒ Sells(West, M1, Nono) [Θ = { z/Nono }]

  ; NOTE: Unification performed above; now z/Nono in subsequent goal resolution

  ; Recurse on new goals
    New Goals: Missile(M1), Owns(Nono, M1)


  ; ===============================================================================
========
  ; Recursive Level 3 Goals:
  { Missile(M1), Owns(Nono, M1) }
  Target goal: Missile(M1)
```

```
  ; Rules that contain our target goals' relations on RHS OR facts containing **our** goal
s:
    3. **Missile**(M1) [Θ = {}] ; **Done! Satisfied this** goal! **Remove** it


  ; ============================================================================
========
  ; **Recursive Level** 3 **Goals**:
  { **Owns**(**Nono**, M1) }
  **Target** goal: **Owns**(**Nono**, M1)


  ; **Rules** that contain **our** target goals' relations on RHS OR facts containing our goal
s:
    2. Owns(Nono, M1) [Θ = {}] ; Done! Satisfied this goal! Remove it

  ; Out of goals for this level, so return success to previous level!


  ; ============================================================================
========
  ; Recursive Level 2 Goals:
  { Hostile(Nono) }
  Target goal: Hostile(Nono)

  ; Rules that contain our target goals' relations on RHS OR facts containing **our** goal
s:
    6. **Enemy**(**Nono, America**) ⇒ **Hostile**(**Nono**)

  ; **Recurse** on **new** goals
    **New Goals: Enemy**(**Nono, America**)


  ; ============================================================================
========
  ; **Recursive Level** 3 **Goals**:
  { **Enemy**(**Nono, America**) }
  **Target** goal: **Enemy**(**Nono, America**)


  ; **Rules** that contain **our** target goals' relations on RHS OR facts containing our goal
s:
    8. Enemy(Nono, America) ; Done! Satisfied this goal! Remove it

  ; Out of goals for this level, so return success to previous level!


  ; ============================================================================
========
  ; Recursive Level 2 Goals:
  { }
  ; Out of goals for this level, so return success to previous level!
```

```
;  ===============================================================================
========
;  Recursive Level 1 Goals:
{  }
;  Out of goals for this level, so return success to previous level!

;  [!] There is no previous level, so we return success! Our KB DOES entail α
```
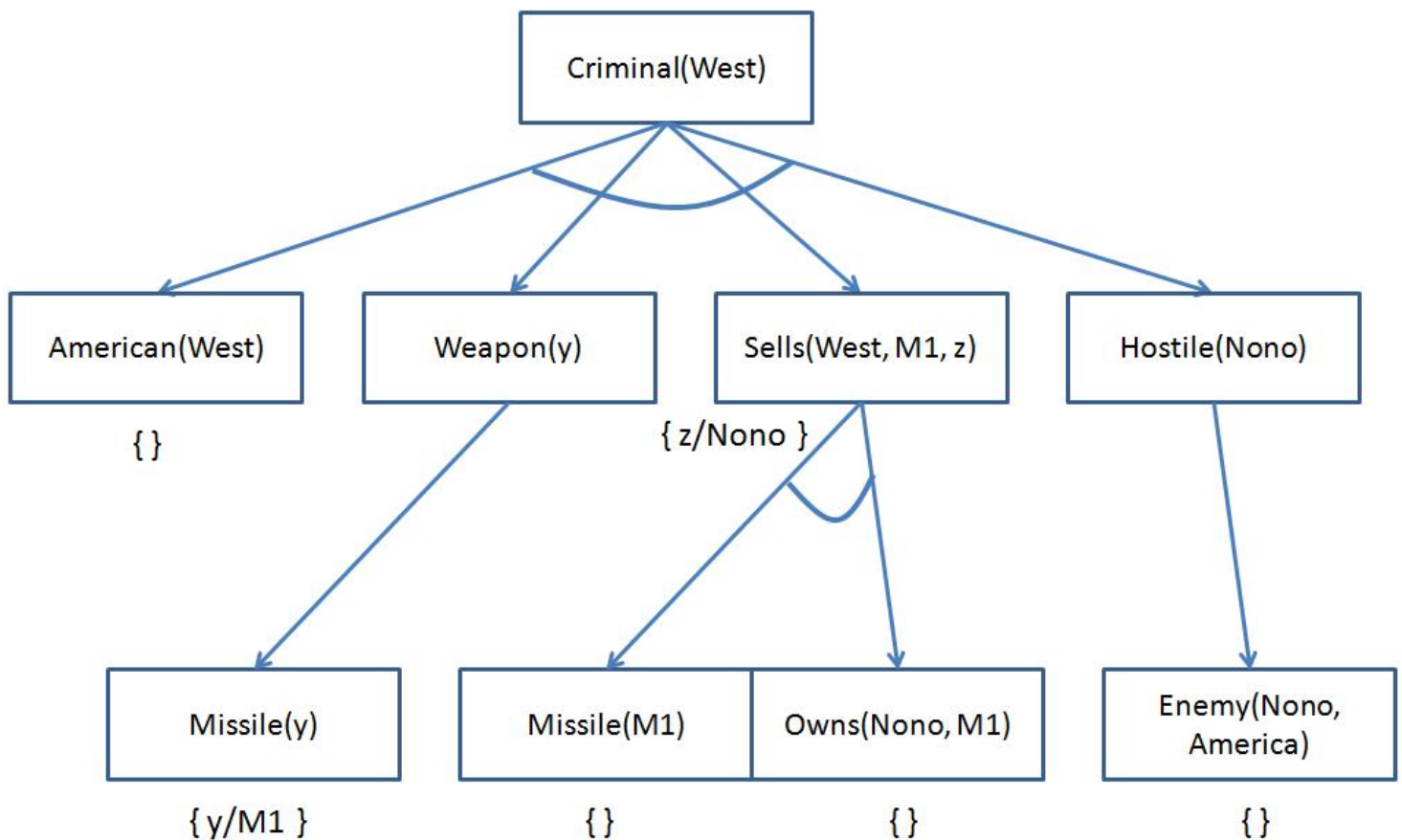
If, at any step above we failed to unify our goal, then we would return failure!

Looks complicated, but why don't we see it pictorially?

Above, the depth of our **proof tree** corresponds to the recursive level in our algorithm's steps, with depth = 1 at the top and nodes processed left-to-right.

Bindings and their locations are illustrated in brackets where they occur.

Arcs (as poorly illustrated as they may be) illustrate AND relationships between bindings.

Practice

Since forward and backward chaining are a bit of a big deal... let's run through an example together...

> **Example**
>
> [t] Using the following knowledge and query (α), perform forward and backward chaining to illustrate that KB ⊨ α

```
KB =
    ; Axioms:
    1. Queen(Liz)
    2. English(Liz)

    ; Rules:
    3. Queen(x)  ⇒ Royal(x)
    4. (Royal(x)  ∧ Likes(x, y)  ∧ Shortage(y))  ⇒ ¬Amused(x)
    5. English(x)  ⇒ ∃y [Tea(y)  ∧ Likes(x, y)]
    6. Tea(x)  ⇒ Shortage(x)

α = Amused(Liz)
```

Homework 3

At this point in discussion we'll review the newly posted homework 3!