First-Order Logic
Semantics (& no inference, but including Unification)

Reading: Chapter 8, 9.1-9.2, ~~9.5.1-9.5.5~~

FOL Syntax and Semantics read: 8.1-8.2

FOL Knowledge Engineering read: 8.3-8.5

FOL Inference read: Chapter 9.1-9.2, ~~9.5.1-9.5.5~~

(Please read lecture topic material before and after each lecture on that topic)

You will be expected to know

- Semantics, Worlds, and Interpretations
- Nested quantifiers
  - Difference between "$\forall$ x $\exists$ y P(x, y)" and "$\exists$ x $\forall$ y P(x, y)"
  - $\forall$ x { $\exists$ y Person(x) $\Rightarrow$ Likes(x, y) }
    - = Every person (every person x) likes something (likes some y).
    - Can be a different y for each x (think about variable scope!)
  - $\exists$ x { $\forall$ y Person(x) $\wedge$ Likes(x, y) }
    - = There is some person (some x) that likes everything (every y).
    - Must be the same x for every y (think about variable scope!)
- Translate simple English sentences to FOPC and back
  - $\forall$ x $\exists$ y Likes(x, y) = "Every person has some person that they like."
  - $\exists$ x $\forall$ y Likes(x, y) = "There is some person who likes every person."
  - Technically, there should be Person(x) & Person(y) predicates above
- Unification: Given two FOL terms containing variables
  - Find the most general unifier if one exists.
  - Else, explain why no unification is possible.
  - See Section 9.2 and Figure 9.1 in your textbook.

Outline

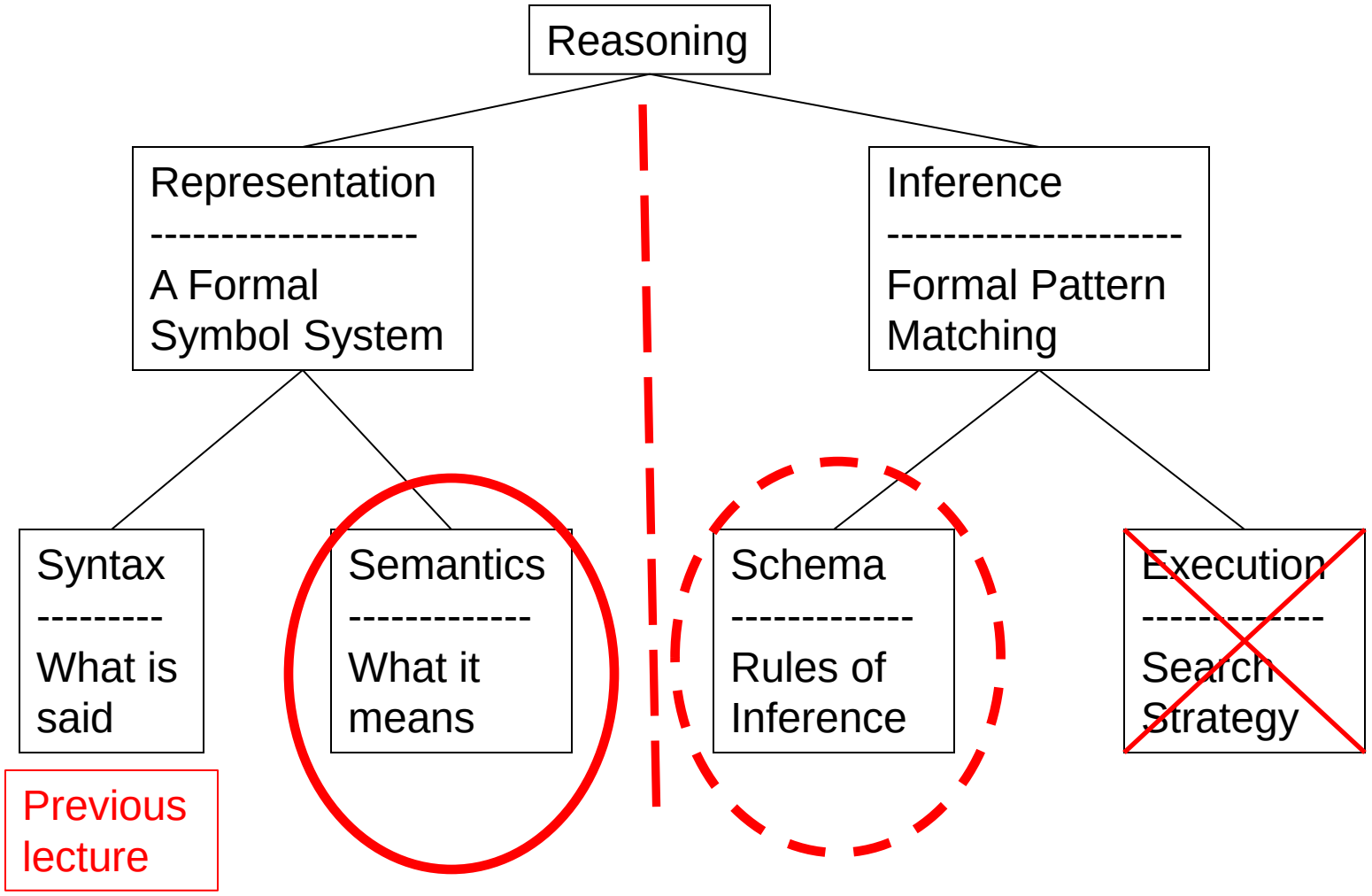
- Review: KB |= S is equivalent to |= (KB $\Rightarrow$ S)
  - So what does {} |= S mean?
- Review: Follows, Entails, Derives
  - Follows: "Is it the case?"
  - Entails: "Is it true?"
  - Derives: "Is it provable?"
- Semantics of FOL (FOPC)
  - Model, Interpretation
- Unification

FOL (or FOPC) Ontology:

What kind of things exist in the world?

What do we need to describe and reason about?

Objects --- with their relations, functions, predicates, properties, and general rules.



Review: KB |= S means |= (KB ⇒ S)

- KB |= S is read "KB entails S."
  - Means "S is true in every world (model) in which KB is true."
  - Means "In the world, S follows from KB."

- KB |= S is equivalent to |= (KB ⇒ S)
  - Means "(KB ⇒ S) is true in every world (i.e., is valid)."

- And so: {} |= S is equivalent to |= ({} ⇒ S)

- So what does ({} ⇒ S) mean?
  - Means "True implies S."
  - Means "S is valid."
  - In Horn form, means "S is a fact." p. 256 (3rd ed.; p. 281, 2nd ed.)

- **Why does {} mean True here, but means False in resolution proofs?**

Review: (True ⇒ S) means "S is a fact."

- By convention,
  - The null conjunct is "syntactic sugar" for True.
  - The null disjunct is "syntactic sugar" for False.
  - Each is assigned the truth value of its identity element.
    - For conjuncts, True is the identity: $(A \wedge \text{True}) \equiv A$
    - For disjuncts, False is the identity: $(A \vee \text{False}) \equiv A$

- A KB is the conjunction of all of its sentences.
  - So in the expression: { } |= S
    - We see that { } is the null conjunct and means True.
  - The expression means "S is true in every world where True is true."
    - I.e., "S is valid."
  - Better way to think of it: { } does not exclude any worlds (models).

- In Conjunctive Normal Form each clause is a disjunct.
  - So in, say, KB = { (P Q) ($\neg$Q R) ( ) (X Y $\neg$Z) }
    - We see that ( ) is the null disjunct and means False.

Side Trip: Functions AND, OR, and null values (Note: These are "syntactic sugar" in logic.)

function AND(*arglist*) **returns** a truth-value
    return ANDOR(*arglist,* True)

function OR(*arglist*) **returns** a truth-value
    return ANDOR(*arglist,* False)

function ANDOR(*arglist, nullvalue*) **returns** a truth-value
    /* *nullvalue* is the identity element for the caller. */
    if (*arglist* = {})
        then return *nullvalue*
    if (FIRST(*arglist*) = NOT(*nullvalue*))
        then return NOT(*nullvalue*)
    return ANDOR(REST(*arglist*), *nullvalue*)

Side Trip: We only need one logical connective. (Note: AND, OR, NOT are "syntactic sugar" in logic.)

Both NAND and NOR are logically complete.

- **NAND is also called the "Sheffer stroke"**
- **NOR is also called "Pierce's arrow"**

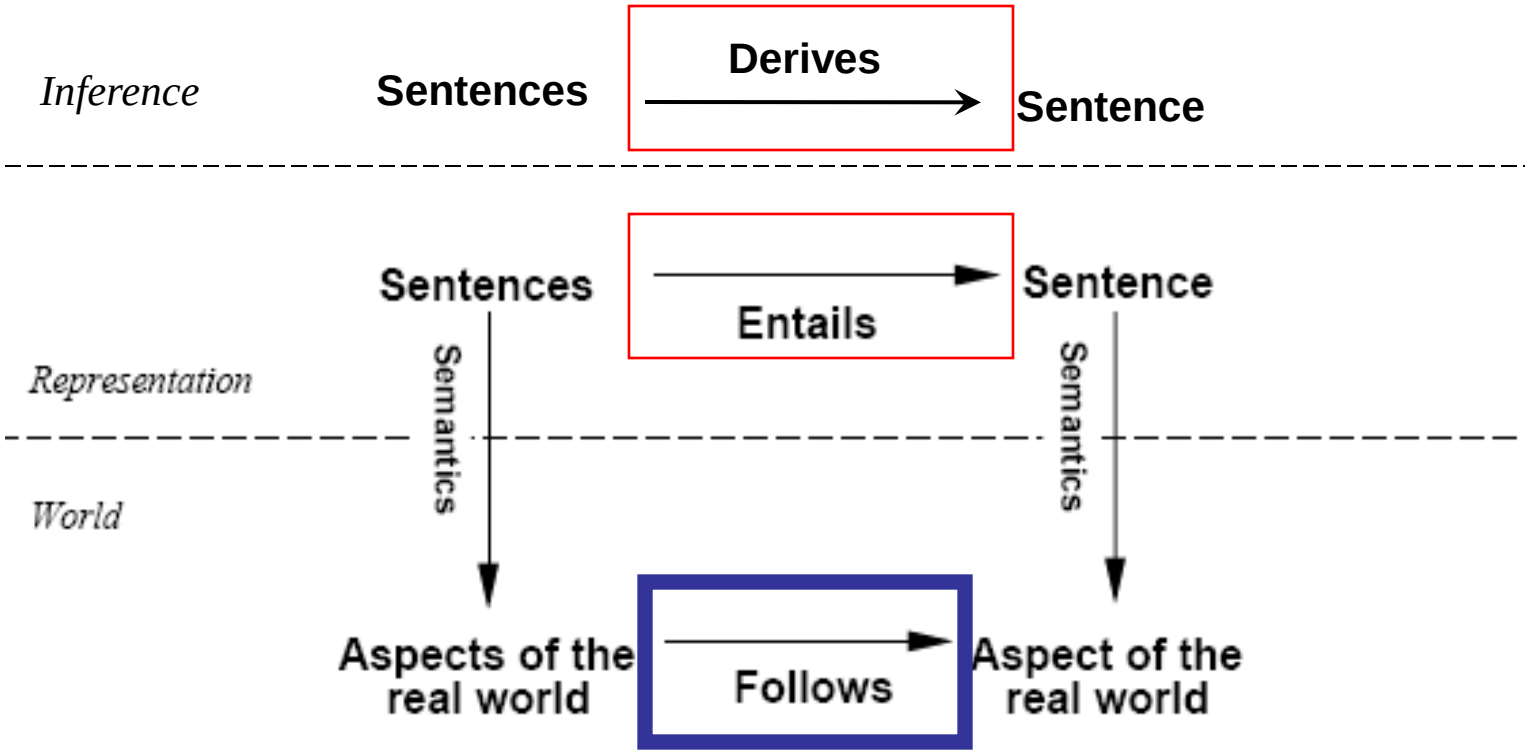
(NOT A) = (NAND A TRUE) = (NOR A FALSE)

(AND A B) = (NAND TRUE (NAND A B))
= (NOR (NOR A FALSE) (NOR B FALSE))

(OR A B) = (NAND (NAND A TRUE) (NAND B TRUE))
=(NOR FALSE (NOR A B))

This fact is exploited by, e.g., VLSI semiconductor fabrication, which often provide a single NAND/NOR gate for efficiency.

Review: Schematic for Follows, Entails, and Derives



If KB is true in the real world,

then any sentence α ***entailed*** *by KB*

and any sentence α ***derived*** *from KB*

by a sound inference procedure

is also true in the real world.

Schematic Example: Follows, Entails, and Derives

Inference

"Mary is Sue's sister and Amy is Sue's daughter."

"An aunt is a sister of a parent."

Derives →

Is it provable?

"Mary is Amy's aunt."

Need also to know:
Daughter(x,y) => Parent(y,x)

Representation

"Mary is Sue's sister and Amy is Sue's daughter."

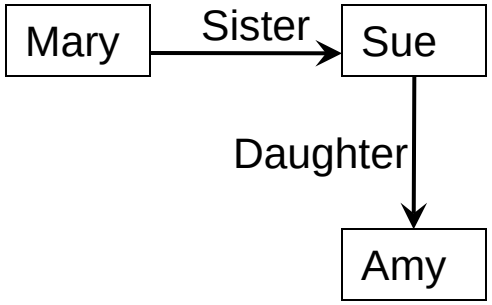
"An aunt is a sister of a parent."

Entails →

Is it true?

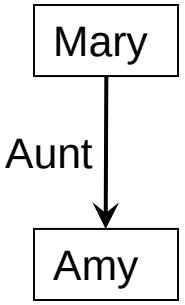
"Mary is Amy's aunt."

World



Follows →

Is it the case?



Review: Models (and in FOL, Interpretations)

- Models are formal worlds within which truth can be evaluated
- Interpretations map symbols in the logic to the world
  - Constant symbols in the logic map to objects in the world
  - n-ary functions/predicates map to n-ary functions/predicates in the world

- We say *m* is a model given an interpretation i of a sentence α if and only if α is true in the world *m* under the mapping *i.*

- Your job, as knowledge engineers, is to ensure that *only* your intended worlds and interpretations make your KB true.

- In the circuit world, you Tell it: "(1 not= 0)".

- In the biology world, you Tell it : "$\forall x$ (Cat(x) $\Rightarrow$ Mammal(x))"

- If you fail to Tell it these facts, then it will make stupid inferences that you will have to come back later and debug, to fix your KB.

- You know *all* these things. It doesn't know any of them. Stupid…

Review: Models (and in FOL, Interpretations)

- Models are formal worlds within which truth can be evaluated
- Interpretations map symbols in the logic to the world
  - Constant symbols in the logic map to objects in the world
  - n-ary functions/predicates map to n-ary functions/predicates in the world

- We say *m* is a model given an interpretation i of a sentence α if and only if α is true in the world *m* under the mapping *i.*

- *M(α)* is the set of all models of α

- Then KB $\models$ α iff *M(KB)* $\subseteq$ *M(*α)
  - E.g. *KB,* = "Mary is Sue's sister and Amy is Sue's daughter."
  - α = "Mary is Amy's aunt." (Must Tell it about mothers/daughters)

- Think of KB and α as constraints, and models as states.
- M(KB) are the solutions to KB and M(α) the solutions to α.
- Then, KB $\models$ α, i.e., $\models$ (KB $\Rightarrow$ a) ,
  when all solutions to KB are also solutions to α.

Review: Semantics of Worlds

- **The** world **consists of** objects **that have** properties**.**
  - **There are** relations **and** functions **between these objects**
  - **Objects in the world, individuals:** people, houses, numbers, colors, baseball games, wars, centuries
    - Clock A, John, 7, the-house in the corner, Tel-Aviv, Ball43
  - **Functions** on individuals:
    - father-of, best friend, third inning of, one more than
  - **Relations**:
    - brother-of, bigger than, inside, part-of, has color, occurred after
  - **Properties (a relation of arity 1)**:
    - red, round, bogus, prime, multistoried, beautiful

Semantics: Interpretation

- An **interpretation** of a sentence (wff) is an assignment that maps
  - Object constant symbols to objects in the world,
  - n-ary function symbols to n-ary functions in the world,
  - n-ary relation symbols to n-ary relations in the world

- Given an interpretation, an atomic sentence has the value "true" if it denotes a relation that holds for those individuals denoted in the terms. Otherwise it has the value "false."
  - Example: Kinship world:
    - Symbols = Ann, Bill, Sue, Married, Parent, Child, Sibling, …
  - World consists of individuals in relations:
    - Married(Ann,Bill) is false, Parent(Bill,Sue) is true, …

- Your job, as a Knowledge Engineer, is to construct KB so it is true ***exactly*** for your world and intended interpretation.

Truth in first-order logic

- Sentences are true with respect to a model and an interpretation
- Model contains objects (domain elements) and relations among them
- Interpretation specifies referents for

constant symbols	→	objects
predicate symbols	→	relations
function symbols	→	functional relations

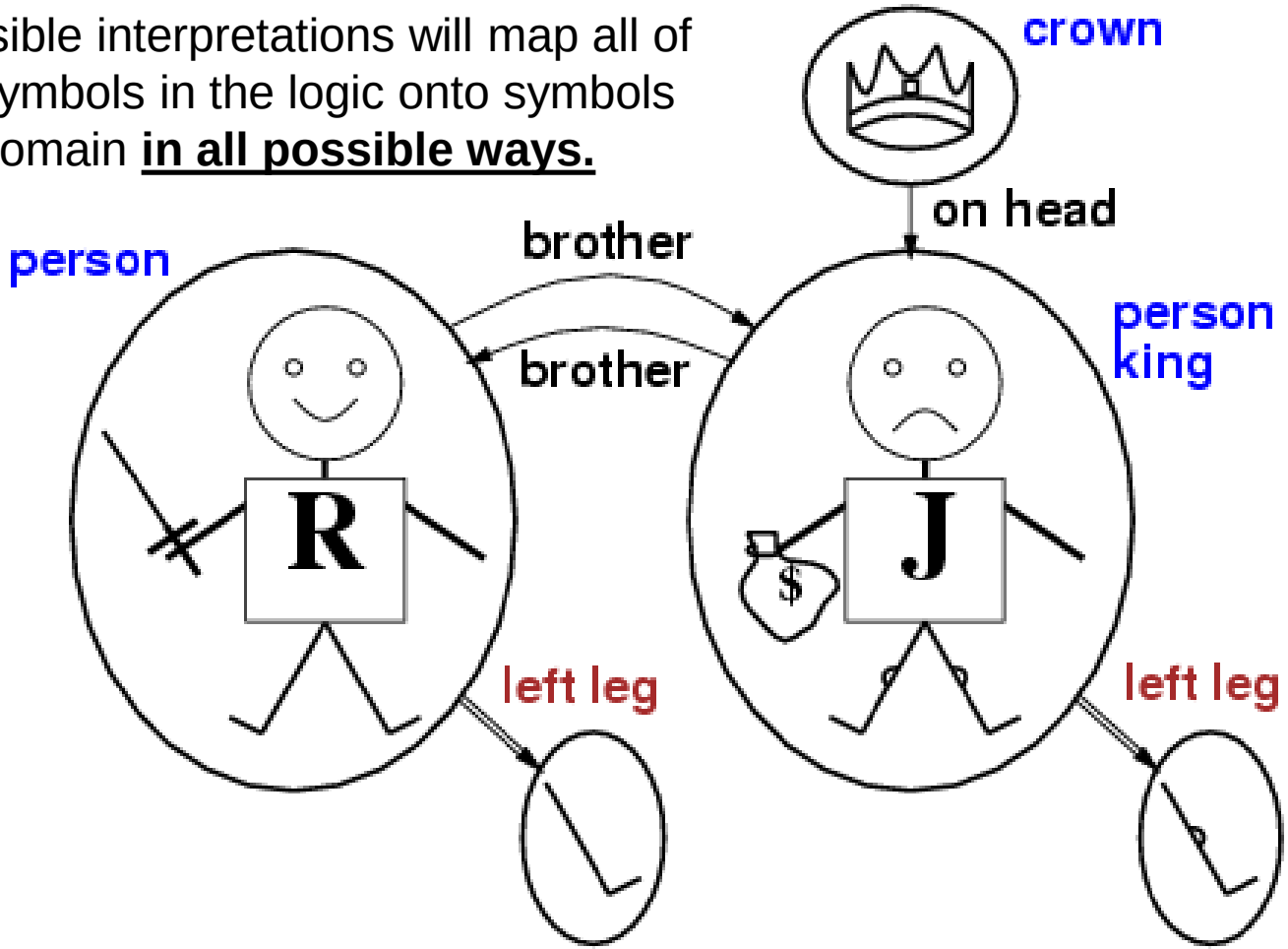
- An atomic sentence *predicate*$(term_1,\ldots,term_n)$ is true iff the objects referred to by $term_1,\ldots,term_n$ are in the relation referred to by *predicate*

Semantics: Models and Definitions

- An interpretation and possible world satisfies a wff (sentence) if the wff has the value “true” under that interpretation in that possible world.
- Model: A domain and an interpretation that satisfies a wff is a model of that wff
- Validity: Any wff that has the value “true” in all possible worlds and under all interpretations is valid.
- Any wff that does not have a model under any interpretation is inconsistent or unsatisfiable.
- Any wff that is true in at least one possible world under at least one interpretation is satisfiable.
- If a wff w has a value true under all the models of a set of sentences KB then KB logically entails w.

Models for FOL: Example

All possible interpretations will map all of these symbols in the logic onto symbols in the domain **in all possible ways.**



An interpretation maps all symbols in KB onto matching symbols in a possible world. All possible interpretations gives a combinatorial explosion of mappings. Your job, as a Knowledge Engineer, is to write the axioms in KB so **they are satisfied only under the intended interpretation in your own real world.**

Summary of FOL Semantics

- A well-formed formula ("wff") FOL is true or false with respect to a world and an interpretation (a model).
- The world has objects, relations, functions, and predicates.
- The interpretation maps symbols in the logic to the world.
- The wff is true if and only if (iff) its assertion holds among the objects in the world under the mapping by the interpretation.
- Your job, as a Knowledge Engineer, is to write sufficient KB axioms that ensure that KB is true in your own real world under your own intended interpretation.
  - **The KB axioms must rule out other worlds and interpretations.**

Summary of Conversion to CNF in FOL

- In order to cover less material with more thoroughness, I no longer cover conversion to CNF and resolution in FOL.
  - Intuitions from Propositional Logic are a good base for FOL.

- The major different points are these:
- Existential quantification is replaced by a "Skolem constant"
  - For example,

    $\exists$ x Lives_in(John, Castle(x)) "John lives in somebody's castle."

    is replaced by Lives_in(John, Castle(Symbol_97))

    "John lives in Symbol_97's castle"
- So, all remaining variables must be universally quantified
  - For example, $\forall$ x King(x) => Person(x) becomes

    ($\neg$ King(x_73) $\vee$ Person(x_73))
  - All variables always given a different name ("standardizing apart")
- **Resolution on predicate terms after unification (substitution)**
  - For example, "($\neg$ King(x_73) Person(x_73))" with "King(John)"

    yields "Person(John)" because of the substitution x_73/John

Unification

- Recall: Subst(θ, p) = result of substituting θ into sentence p

- Unify algorithm: takes 2 sentences p and q and returns a unifier if one exists

  Unify(p,q) = θ where Subst(θ, p) = Subst(θ, q)

- Example:

  p = Knows(John,x)

  q = Knows(John, Jane)

  Unify(p,q) = {x/Jane}

Unification examples

- simple example: query = Knows(John,x), i.e., who does John know?

p	q	θ
Knows(John,x)	Knows(John,Jane)	{x/Jane}
Knows(John,x)	Knows(y,OJ)	{x/OJ,y/John}
Knows(John,x)	Knows(y,Mother(y))	{y/John,x/Mother(John)}
Knows(John,x)	Knows(x,OJ)	{fail}

- Last unification fails: only because x can't take values John and OJ at the same time
  - But we know that if John knows x, and everyone (x) knows OJ, we should be able to infer that John knows OJ
- Problem is due to use of same variable x in both sentences
- Simple solution: Standardizing apart eliminates overlap of variables, e.g., Knows(z,OJ)

Unification

- To unify *Knows(John,x)* and *Knows(y,z)*,

  θ = {y/John, x/z } or θ = {y/John, x/John, z/John}

- The first unifier is more general than the second.

- There is a single most general unifier (MGU) that is unique up to renaming of variables.

  MGU = { y/John, x/z }

- General algorithm in Figure 9.1 in the text

Unification Algorithm

function UNIFY(x, y, θ) **returns** a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
    y, a variable, constant, list, or compound expression
    θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure **then return** failure
  else if $x = y$ **then return** θ
  else if VARIABLE?(x) **then return** UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) **then return** UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) **and** COMPOUND?(y) **then**
    return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) **and** LIST?(y) **then**
    return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure

---

function UNIFY-VAR(var, x, θ) **returns** a substitution

  if $\{var/val\} \in \theta$ **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** failure
  else return add $\{var/x\}$ to θ

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Unification Algorithm

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure
```

If we have failed or succeeded, then fail or succeed.

```
function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Unification Algorithm

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure
```

If we can unify a variable then do so.

```
function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Unification Algorithm

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution
  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

If we already have bound variable *var* to a value, try to continue on that basis.

Figur There is an implicit assumption that "{var/val} ∈ θ", if it
of the succeeds, binds val to the value that allowed it to succeed,
up alo
that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Unification Algorithm

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

If we already have bound *x* to a value, try to continue on that basis.

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Unification Algorithm

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

If *var* occurs anywhere within *x*, then no substitution will succeed.

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Unification Algorithm

function UNIFY(x, y, θ) **returns** a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
    y, a variable, constant, list, or compound expression
    θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure **then return** failure
  else if $x = y$ **then return** θ
  else if VARIABLE?(x) **then return** UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) **then return** UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) **and** COMPOUND?(y) **then**
    return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) **and** LIST?(y) **then**
    return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure

---

function UNIFY-VAR(var, x, θ) **returns** a substitution

  if $\{var/val\} \in \theta$ **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** failure
  else return add $\{var/x\}$ to θ

Else, try to bind *var* to *x*, and recurse.

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Unification Algorithm

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure
```

If a predicate/function, unify the arguments.

```
function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Unification Algorithm

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure
```

If unifying arguments, unify the remaining arguments.

```
function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Unification Algorithm

function UNIFY(x, y, θ) **returns** a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
    y, a variable, constant, list, or compound expression
    θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure **then return** failure
  else if $x = y$ **then return** θ
  else if VARIABLE?(x) **then return** UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) **then return** UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) **and** COMPOUND?(y) **then**
    return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) **and** LIST?(y) **then**
    return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure Otherwise, fail.

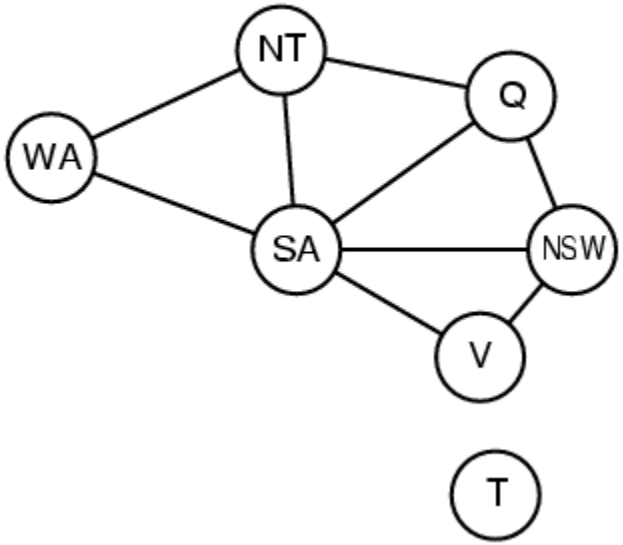
---

function UNIFY-VAR(var, x, θ) **returns** a substitution

  if $\{var/val\} \in \theta$ **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** failure
  else return add $\{var/x\}$ to θ

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Hard matching example



$Diff(wa,nt) \wedge Diff(wa,sa) \wedge Diff(nt,q) \wedge Diff(nt,sa) \wedge Diff(q,nsw) \wedge Diff(q,sa) \wedge Diff(nsw,v) \wedge Diff(nsw,sa) \wedge Diff(v,sa) \Rightarrow Colorable()$

$Diff(Red,Blue)$ $Diff(Red,Green)$
$Diff(Green,Red)$ $Diff(Green,Blue)$
$Diff(Blue,Red)$ $Diff(Blue,Green)$

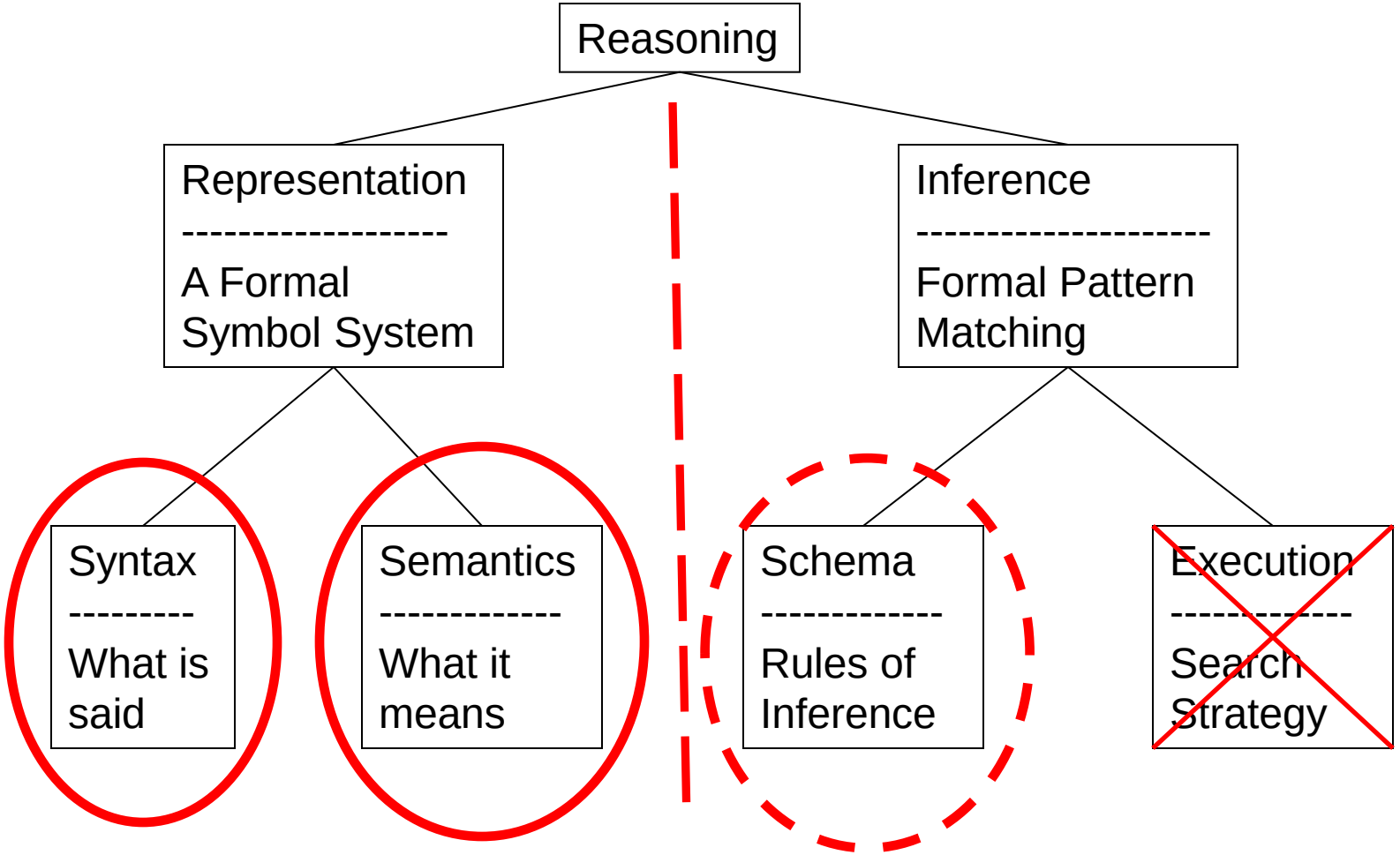
- To unify the grounded propositions with premises of the implication you need to solve a CSP!
- *Colorable*() is inferred iff the CSP has a solution
- CSPs include 3SAT as a special case, hence matching is NP-hard

FOL (or FOPC) Ontology:

What kind of things exist in the world?

What do we need to describe and reason about?

Objects --- with their relations, functions, predicates, properties, and general rules.



Summary

- First-order logic:
  - Much more expressive than propositional logic
  - Allows objects and relations as semantic primitives
  - Universal and existential quantifiers
- Syntax: constants, functions, predicates, equality, quantifiers
- Nested quantifiers
- Translate simple English sentences to FOPC and back
- Semantics: correct under any interpretation and in any world
- Unification: Making terms identical by substitution
  - The terms are universally quantified, so substitutions are justified.