

MODULE -4

Subject:ARTIFICIAL INTELLIGENCE

Subject Code: BCS515B



Contents

- **First Order Logic:** Representation Revisited, Syntax and Semantics of First Order logic, Using First Order logic, Knowledge Engineering In First-Order Logic
- **Inference in First Order Logic:** Propositional Versus First Order Inference, Unification, Forward Chaining

Drawbacks of Propositional Logic

- Propositional logic is very simple and declarative, in which knowledge and inference are separate, and inference is entirely domain independent
- Propositional logic has lack of data structure in programming.
- Propositional logic is not sufficient for complex or natural language sentences.
- E.g. Some students in KEC are intelligent
- Propositional logic has very limited expressive power
- E.g., cannot say "pits cause breezes in adjacent squares" except by writing one sentence for each square

First Order Logic (FOL)

- First-order logic is also known as Predicate logic or First-order predicate logic.
- First-order logic, like natural language has well defined syntax and semantics.
- It assumes the world contains
- Objects: people, houses, numbers, colors, baseball games, wars, ...
- Relations: red, round, prime, brother of, bigger than, part of, comes between, ...
- Functions: father of, best friend, one more than, plus,

Properties of FOL

- It has ability to represent facts about some or all of the objects and relations in the universe
- Represent law and rules extracted from real world
- Useful language for Mathematic , Philosophy and AI
- Represent facts in realistic manner rather than just true / false
- Makes ontological commitment

Syntax of FOL: Basic Elements

- **Constants** KingJohn, 2, NUS,...
- **Predicates** Brother, >,...
- **Functions** Sqrt, LeftLegOf, ...
- **Variables** x, y, a, b,...
- **Connectives** $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$
- **Equality** =
- **Quantifiers** $\forall$, $\exists$
- **Sentences** Atom / complex sentences
- **Atom** True / False / AP (atomic proposition)
- **Complex sentence** (sentence) / connective sentence / $\neg$ sentence

Atomic sentences

- Atomic sentence = *predicate* ($term_1, \dots, term_n$) or $term_1 = term_2$
- Term = *function* ($term_1, \dots, term_n$) or *constant* or *variable*
- Example.
- *Brother(KingJohn, RichardTheLionheart)*
- *> (Length(LeftLegOf(Richard)), Length(LeftLegOf(KingJohn)))*

Complex sentences

- Complex sentences are made from atomic sentences using five logical connectives
- S_1 and S_2 are two atomic statements then
- $\neg S_1, S_1 \wedge S_2, S_1 \vee S_2, S_1 \Rightarrow S_2, S_1 \Leftrightarrow S_2,$

E.g. $Sibling(KingJohn, Richard) \Leftrightarrow Sibling(Richard, KingJohn)$

$>(1,2) \vee \leq (1,2)$

$>(1,2) \wedge \neg >(1,2)$

Truth in FOL

- 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 $\text{predicate}(\text{term}_1, \dots, \text{term}_n)$ is true if the objects referred by $\text{term}_1, \dots, \text{term}_n$ are in the relation referred by predicate

A) Syntax and Semantics of First Order logic

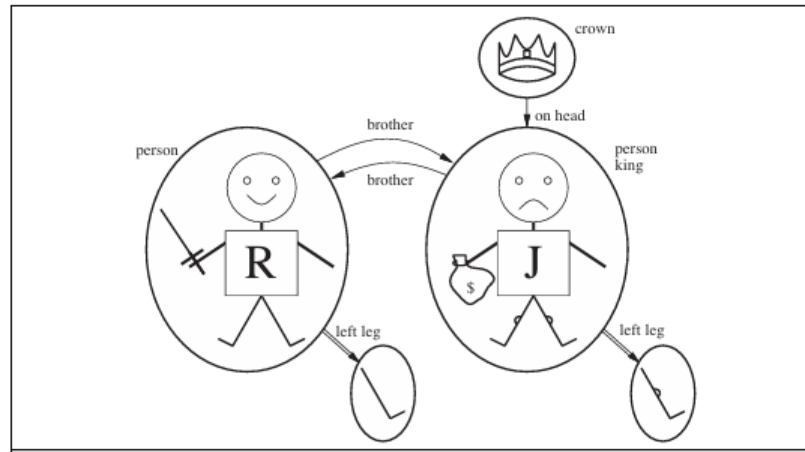


Figure 4.2: A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

- The objects in the model may be related in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation is just the set of tuples of objects that are related.
- (A tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.) Thus, the brotherhood relation in this model is the set {Richard the Lionheart, King John, King John, Richard the Lionheart }.

- For example, each person has one left leg, so the model has a unary “left leg” function that includes the following mappings:
- {Richard the Lionheart} $\rightarrow$ Richard’s left leg(A)
{King John} $\rightarrow$ John’s left leg.(B)
- Strictly speaking, models in first-order logic require total functions, that is, there must be a value for every input tuple. Thus, the crown must have a left leg and so must each of the left legs. There is a technical solution to this awkward problem involving an additional “invisible”

B) Symbols and interpretations

```

Sentence → AtomicSentence | ComplexSentence
AtomicSentence → Predicate | Predicate(Term,...) | Term = Term
ComplexSentence → ( Sentence ) | [ Sentence ]
                | ¬ Sentence
                | Sentence ∧ Sentence
                | Sentence ∨ Sentence
                | Sentence ⇒ Sentence
                | Sentence ⇔ Sentence
                | Quantifier Variable,... Sentence

Term → Function(Term,...)
      | Constant
      | Variable

Quantifier → ∀ | ∃
Constant → A | X1 | John | ...
Variable → a | x | s | ...
Predicate → True | False | After | Loves | Raining | ...
Function → Mother | LeftLeg | ...

OPERATOR PRECEDENCE : ¬, =, ∧, ∨, ⇒, ⇔
  
```

Figure 4.3: The syntax of first-order logic with equality, specified in Backus–Naur form Operator precedence's are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

- **For example**, we might use the constant symbols Richard and John; the predicate symbols Brother, On Head, Person, King, and Crown; and the function symbol LeftLeg.
- As with proposition symbols, the choice of names is entirely up to the user. Each predicate and function symbol comes with an arity that fixes the number of arguments.
- As in **propositional logic**, every model must provide the information required to determine if any given sentence is true or false.
- Thus, in addition to its objects, relations, and functions, each model includes an interpretation that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols. One possible interpretation for **our example**—which a logician would call the intended interpretation—is as follows:

- Richard refers to Richard the Lion heart and John refers to the evil King John. Brother refers to the brotherhood relation, that is, the set of tuples of objects given in as mentioned in Equation (A); On Head refers to the “on head” relation that holds between the crown and King John; Person, King, and Crown refer to the sets of objects that are persons, kings, and crowns.
- LeftLeg refers to the “left leg” function, that is, the mapping given in Equation (B).

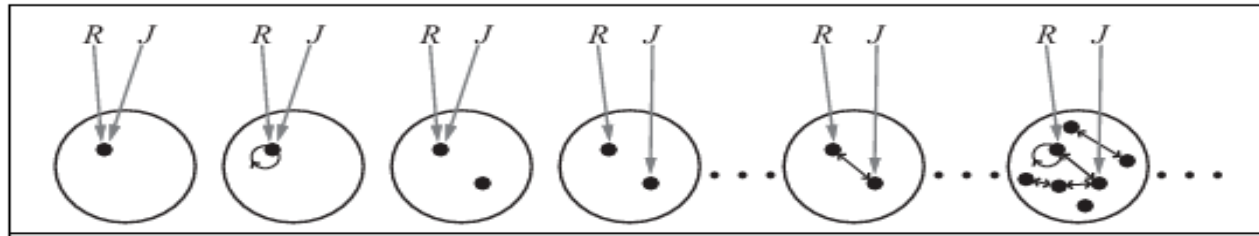


Figure 4.4: Some members of the set of all models for a language with two constant symbols, R and J , and one binary relation symbol. The interpretation of each constant symbol is shown by a gray arrow. Within each model, the related objects are connected by arrows.

4.2.3 Terms

- A term is a logical expression that refers to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object. For example, in English we might use the expression “King John’s left leg” rather than giving a name to his leg.

4.2.4 Atomic sentences

- Now that we have both terms for referring to objects and predicate symbols for referring to relations, we can put them together to make atomic sentences that state facts. An atomic sentence (or atom for short) is formed from a predicate symbol optionally followed by parenthesized list of terms, such as Brother(Richard,John).

4.2.5 Complex sentences

- We can use logical connectives to construct more complex sentences, with the same syntax and semantics as in propositional calculus. Here are four sentences that are true in the model of Figure 4.2 under our intended interpretation:
- $\neg \text{Brother}(\text{LeftLeg}(\text{Richard}), \text{John})$
- $\text{Brother}(\text{Richard}, \text{John}) \wedge \text{Brother}(\text{John}, \text{Richard})$
- $\text{King}(\text{Richard}) \vee \text{King}(\text{John})$
- $\neg \text{King}(\text{Richard}) \Rightarrow \text{King}(\text{John})$.

4.2.6 Quantifiers

- Once we have a logic that allows objects, it is only natural to want to express properties of entire collections of objects, instead of enumerating the objects by name. Quantifiers let us do this.
- First-order logic contains two standard quantifiers, called universal and existential.

1) Universal quantification ($\forall$)

- The expression of general rules in propositional logic. Rules such as “Squares neighboring the wumpus are smelly” and “All kings are persons” are the bread and butter of first-order logic. The second rule, “All kings are persons,” is written in first-order logic as $\forall x \text{ King}(x) \Rightarrow \text{Person}(x)$. $\forall$ is usually pronounced “For all ...”. (Remember that the upside-down A stands for “all.”)
- We can extend the interpretation in five ways:
- $x \rightarrow \text{Richard the Lionheart}$,
- $x \rightarrow \text{KingJohn}$,
- $x \rightarrow \text{Richard's left leg}$,
- $x \rightarrow \text{John's left leg}$,
- $x \rightarrow \text{thecrown}$.

2) Existential quantification ($\exists$)

- Universal quantification makes statements about every object. Similarly, we can make a statement about some object in the universe without naming it, by using an existential quantifier. To say, for example, that King John has a crown on his head, we write
- $\exists x \text{ Crown}(x) \wedge \text{OnHead}(x, \text{John})$.
- $\exists x$ is pronounced “There exists an x such that ...” or “For some x ...”.
- Intuitively, the sentence $\exists x P$ says that P is true for at least one object x . More precisely, $\exists x P$ is true in a given model if P is true in at least one extended interpretation that assigns x to a domain element. That is, at least one of the following is true:
 - *Richard the Lionheart is a crown $\wedge$ Richard the Lionheart is on John’s head;*
 - *King John is a crown $\wedge$ King John is on John’s head;*
 - *Richard’s left leg is a crown $\wedge$ Richard’s left leg is on John’s head;*
 - *John’s left leg is a crown $\wedge$ John’s left leg is on John’s head;*
 - *The crown is a crown $\wedge$ the crown is on John’s head.*

3) Nested quantifiers

- We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, “Brothers are siblings” can be written as

$$\forall x \forall y \text{ Brother}(x,y) \Rightarrow \text{Sibling}(x,y).$$

- Consecutive quantifiers of the same type can be written as one quantifier with several variables. For example, to say that siblinghood is a symmetric relationship, we can write

$$\forall x,y \text{ Sibling}(x,y) \Leftrightarrow \text{Sibling}(y,x) .$$

- In other cases we will have mixtures. “Everybody loves somebody” means that for every person, there is someone that person loves:

$$\forall x \exists y \text{ Loves}(x,y) .$$

- On the other hand, to say “There is someone who is loved by everyone,” we write

$$\exists y \forall x \text{ Loves}(x,y)$$

4) Connections between $\forall$ and $\exists$

- The two quantifiers are actually intimately connected with each other, through negation. As inserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

$\forall x \neg Likes(x, Parsnips)$ is equivalent to $\neg \exists x Likes(x, Parsnips)$

- We can go one step further: “Everyone likes ice cream” means that there is no one who does not like ice cream:

$\forall x Likes(x, IceCream)$ is equivalent to $\neg \exists x \neg Likes(x, IceCream)$.

- Because $\forall$ is really a conjunction over the universe of objects and $\exists$ is a disjunction, it should not be surprising that they obey De Morgan’s rules. The De Morgan rules for quantified and unquantified sentences are as follows: Thus, we do not really need both $\forall$ and $\exists$, just as we do not really need both $\wedge$ and $\vee$. Still, readability is more important than parsimony, so we will keep both of the quantifiers.

4.2.7 Equality

- First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. We can use the equality symbol to signify that two terms refer to the same object. For example,

Father(John)=Henry

- To say that Richard has at least two brothers, we would write

$\exists x, y \text{ Brother}(x, \text{Richard}) \wedge \text{Brother}(y, \text{Richard}) \wedge \neg(x=y)$.

The sentence

$\exists x, y \text{ Brother}(x, \text{Richard}) \wedge \text{Brother}(y, \text{Richard})$

does not have the intended meaning. In particular, it is true in the model of Figure 4.2, where Richard has only one brother. To see this, consider the extended interpretation in which both x and y are assigned to King John. The addition of $\neg(x=y)$ rules out such models. The notation $x=y$ is sometimes used as an abbreviation for $\neg(x=y)$

4.2.8 An alternative semantics?

- Suppose that we believe that Richard has two brothers, John and Geoffrey. 8 Can we capture this state of affairs by asserting $Brother(John, Richard) \wedge Brother(Geoffrey, Richard)$?
- Not quite. First, this assertion is true in a model where Richard has only one brother— we need to add $John = Geoffrey$. Second, the sentence doesn't rule out models in which Richard has many more brothers besides John and Geoffrey. Thus, the correct translation of "Richard's brothers are John and Geoffrey" is as follows:
- $Brother(John, Richard) \wedge Brother(Geoffrey, Richard) \wedge John = Geoffrey$
- $\wedge \forall x Brother(x, Richard) \Rightarrow (x = John \vee x = Geoffrey)$.
- For many purposes, this seems much more cumbersome than the corresponding natural language expression

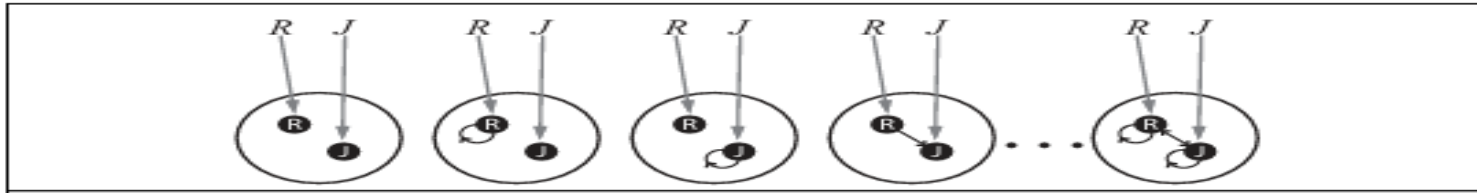


Figure 4.5: Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol, under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

- One proposal that is very popular in database systems works as follows.
- **First**, we insist that every constant symbol refer to a distinct object—the so-called unique-names assumption.
- **Second**, we assume that atomic sentences not known to be true are in fact false—the closed-world assumption.
- **Finally**, we invoke domain closure, meaning that each model contains no more domain elements than those named by the constant symbols.
- Under the resulting semantics, which we call database semantics to distinguish it from the standard semantics of first-order logic, the sentence Equation (B) does indeed state that Richard's two brothers are John and Geoffrey.

4.3 Using First Order Logic

- We begin with a brief description of the TELL/ASK interface for first-order knowledge bases. Then we look at the domains of family relationships, numbers, sets, and lists, and at the wumpus world.

1. Assertions and queries in first-order logic

- Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called assertions. For example, we can assert that John is a king, Richard is a person, and all kings are persons:
 - TELL(KB, King(John)) .
 - TELL(KB, Person(Richard)) .
 - TELL(KB, $\forall x \text{ King}(x) \Rightarrow \text{Person}(x)$) .



- We can ask questions of the knowledge base using ASK. For example, $\text{ASK}(\text{KB}, \text{King}(\text{John}))$ returns true.
- Questions asked with ASK are called queries or goals. Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively.
- For example, given the two preceding assertions, the query $\text{ASK}(\text{KB}, \text{Person}(\text{John}))$ should also return true. We can ask quantified queries, such as $\text{ASK}(\text{KB}, \exists x \text{ Person}(x))$

4.3.2 The kinship domain

- The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as “Elizabeth is the mother of Charles” and “Charles is the father of William” and rules such as “One’s grandmother is the mother of one’s parent.” Clearly, the objects in our domain are people.
- We have two unary predicates, Male and Female.
- **Kinship relations**—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: Parent, Sibling, Brother, Sister, Child, Daughter, Son, Spouse, Wife, Husband, Grandparent, Grandchild, Cousin, Aunt, and Uncle. We use functions for Mother and Father, because every person has exactly one of each of these (at least according to nature’s design).

- We can go through each function and predicate, writing down what we know in terms of the other symbols. For example, one's mother is one's female parent:
- $\forall m, c \text{ Mother}(c)=m \Leftrightarrow \text{Female}(m) \wedge \text{Parent}(m, c).$
- One's husband is one's male spouse:
- $\forall w, h \text{ Husband}(h, w) \Leftrightarrow \text{Male}(h) \wedge \text{Spouse}(h, w).$
- Male and female are disjoint categories:
- $\forall x \text{ Male}(x) \Leftrightarrow \neg \text{Female}(x).$
- Parent and child are inverse relations:
- $\forall p, c \text{ Parent}(p, c) \Leftrightarrow \text{Child}(c, p) .$
- A grandparent is a parent of one's parent:
- $\forall g, c \text{ Grandparent}(g, c) \Leftrightarrow \exists p \text{ Parent}(g, p) \wedge \text{Parent}(p, c).$
- A sibling is another child of one's parents:
- $\forall x, y \text{ Sibling}(x, y) \Leftrightarrow x \neq y \wedge \exists p \text{ Parent}(p, x) \wedge \text{Parent}(p, y).$

4.3.3 Numbers, sets, and lists

- Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We describe here the theory of natural numbers or non-negative integers. We need a predicate NatNum that will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, S (successor). The Peano axioms define natural numbers and addition. Natural numbers are defined recursively:
 - $\text{NatNum}(0)$.
 - $\forall n \text{ NatNum}(n) \Rightarrow \text{NatNum}(S(n))$
 - That is, 0 is a natural number, and for every object n , if n is a natural number, then $S(n)$ is a natural number. So the natural numbers are 0, $S(0)$, $S(S(0))$, and so on. We also need axioms to constrain the successor function:
 - $\forall n 0 \neq S(n)$.
 - $\forall m, n m = n \Rightarrow S(m) = S(n)$.
 - Now we can define addition in terms of the successor function:
 - $\forall m \text{ NatNum}(m) \Rightarrow +(0, m) = m$.
 - $\forall m, n \text{ NatNum}(m) \wedge \text{NatNum}(n) \Rightarrow +(S(m), n) = S(+(m, n))$.

- We will use the normal vocabulary of set theory as syntactic sugar. The empty set is a constant written as $\{\}$. There is one unary predicate, Set , which is true of sets. The binary predicates are $x \in s$ (x is a member of set s) and $s_1 \subseteq s_2$ (set s_1 is a subset, not necessarily proper, of set s_2). The binary functions are $s_1 \cap s_2$ (the intersection of two sets), $s_1 \cup s_2$ (the union of two sets), and $\{x | s\}$ (the set resulting from adjoining element x to set s). One possible set of axioms is as follows:
 - 1. The only sets are the empty set and those made by adjoining something to a set:
 - $\forall s \text{ Set}(s) \Leftrightarrow (s = \{\}) \vee (\exists x, s_2 \text{ Set}(s_2) \wedge s = \{x | s_2\})$.
 - 2. The empty set has no elements adjoined into it. In other words, there is no way to decompose $\{\}$ into a smaller set and an element:
 - $\neg \exists x, s \{x | s\} = \{\}$.
 - 3. Adjoining an element already in the set has no effect:
 - $\forall x, s x \in s \Leftrightarrow s = \{x | s\}$.
 - 4. The only members of a set are the elements that were adjoined into it. We express this

recursively, saying that x is a member of s if and only if s is equal to some set s_2 adjoined with some element y , where either y is the same as x or x is a member of s_2 :

- $\forall x, s \ x \in s \Leftrightarrow \exists y, s_2 \ (s = \{y/s_2\} \wedge (x = y \vee x \in s_2)).$

5. A set is a subset of another set if and only if all of the first set's members are members of the second set:

- $\forall s_1, s_2 \ s_1 \subseteq s_2 \Leftrightarrow (\forall x \ x \in s_1 \Rightarrow x \in s_2).$

6. Two sets are equal if and only if each is a subset of the other:

- $\forall s_1, s_2 \ (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1).$

7. An object is in the intersection of two sets if and only if it is a member of both sets:

- $\forall x, s_1, s_2 \ x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \wedge x \in s_2).$

8. An object is in the union of two sets if and only if it is a member of either set:

- $\forall x, s_1, s_2 \ x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \vee x \in s_2).$

LIST

- Lists are similar to sets. The differences are that lists are ordered and the same element can appear more than once in a list.
- We can use the vocabulary of Lisp for lists: Nil is the constant list with no elements; Cons, Append, First, and Rest are functions; and Find is the predicate that does for lists what Member does for sets. List? is a predicate that is true only of lists.
- As with sets, it is common to use syntactic sugar in logical sentences involving lists. The empty list is []. The term Cons(x,y), where y is a nonempty list, is written [x|y]. The term Cons(x,Nil) (i.e., the list containing the element x) is written as [x]. A list of several elements, such as [A,B,C], corresponds to the nested term Cons(A,Cons(B,Cons(C,Nil))).

4.3.4 The wumpus world

- The first order axioms in this section are much more concise, capturing in a natural way exactly what we want to say. Recall that the wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what. We use integers for time steps. A typical percept sentence would be
- *Percept([Stench,Breeze,Glitter,None,None],5)* .
- Here, Percept is a binary predicate, and Stench and so on are constants placed in a list. The actions in the wumpus world can be represented by logical terms:

- *Turn(Right), Turn(Left), Forward, Shoot, Grab, Climb* .
- To determine which is best, the agent program executes the query
- *ASKVARS($\exists a$ BestAction($a,5$))* ,
- which returns a binding list such as {*a/Grab*}. The agent program can then return *Grab* as the action to take. The raw percept data implies certain facts about the current state. For example:
- $\forall t,s,g,m,c$ *Percept*(*[s,Breeze,g,m,c]*,*t*) $\Rightarrow$ *Breeze*(*t*) ,
- $\forall t,s,b,m,c$ *Percept*(*[s,b,Glitter,m,c]*,*t*) $\Rightarrow$ *Glitter*(*t*) ,
- and so on. These rules exhibit a trivial form of the reasoning process called perception, Notice the quantification over time *t*. In propositional logic, we would need copies of each sentence for each time step. Simple “reflex” behavior can also be implemented by quantified implication sentences. For example, we have
- $\forall t$ *Glitter*(*t*) $\Rightarrow$ *BestAction*(*Grab*,*t*) .

- Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion $\text{BestAction}(\text{Grab}, 5)$ —that is, Grab is the right thing to do. We have represented the agent's inputs and outputs; now it is time to represent the environment itself. Let us begin with objects. Obvious candidates are squares, pits, and the wumpus. We could name each square—Square1,2 and soon—but then the fact that Square1,2 and Square1,3 are adjacent would have to be an “extra” fact, and we would need one such fact for each pair of squares. It is better to use a complex term in which the row and column appear as integers; for example, we can simply use the list term [1,2]. Adjacency of any two squares can be defined as
- $$\forall x, y, a, b \text{ Adjacent}([x, y], [a, b]) \Leftrightarrow$$
$$(x = a \wedge (y = b - 1 \vee y = b + 1)) \vee (y = b \wedge (x = a - 1 \vee x = a + 1)).$$

- We could name each pit, but this would be inappropriate for a different reason: there is no reason to distinguish among pits. It is simpler to use a unary predicate *Pit* that is true of squares containing pits. Finally, since there is exactly one wumpus, a constant *Wumpus* is just as good as a unary predicate (and perhaps more dignified from the wumpus's viewpoint). The agent's location changes over time, so we write $At(Agent, s, t)$ to mean that the agent is at square *s* at time *t*. We can fix the wumpus's location with $\forall t At(Wumpus, [2,2], t)$. We can then say that objects can only be at one location at a time:
 - $\forall x, s1, s2, t \quad At(x, s1, t) \wedge At(x, s2, t) \Rightarrow s1 = s2$.
 - Given its current location, the agent can infer properties of the square from properties of its current percept. For example, if the agent is at a square and perceives a breeze, then that square is breezy:
 - $\forall s, t \quad At(Agent, s, t) \wedge Breeze(t) \Rightarrow Breezy(s)$.

- It is useful to know that a square is breezy because we know that the pits cannot move about.
- Notice that Breezy has no time argument. Having discovered which places are breezy (or smelly) and, very important, not breezy (or not smelly), the agent can deduce where the pits are (and where the wumpus is). Whereas propositional logic necessitates a separate axiom for each square and would need a different set of axioms for each geographical layout of the world, first-order logic just needs one axiom:
- $\forall s \text{ Breezy}(s) \Leftrightarrow \exists r \text{ Adjacent}(r,s) \wedge \text{Pit}(r)$. Similarly, in first-order logic we can quantify over time, so we need just one successor-state axiom for each predicate, rather than a different copy for each time step. For example, the axiom for the arrow becomes
- $\forall t \text{ HaveArrow}(t+1) \Leftrightarrow (\text{HaveArrow}(t) \wedge \neg \text{Action}(\text{Shoot}, t))$.
- From these two example sentences, we can see that the first-order logic formulation is no less concise than the original English-language.

4.4 Knowledge Engineering In First- Order Logic

- A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain.
- We illustrate the knowledge engineering process in an electronic circuit domain that should already be fairly familiar, so that we can concentrate on the representational issues involved.
- The approach we take is suitable for developing special-purpose knowledge bases whose domain is carefully circumscribed and whose range of queries is known in advance. General-purpose knowledge bases, which cover a broad range of human knowledge and are intended to support tasks such as natural language understanding.

4.4.1 The knowledge-engineering process

- Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:
 - 1. Identify the task.** The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance. For example, does the wumpus knowledge base need to be able to choose actions or is it required to answer questions only about the contents of the environment? Will the sensor facts include the current location? The task will determine what knowledge must be represented in order to connect problem instances to answers.

2. Assemble the relevant knowledge. The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know—a process called knowledge acquisition. At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works. For the wumpus world, which is defined by an artificial set of rules, the relevant knowledge is easy to identify.

3. Decide on a vocabulary of predicates, functions, and constants. That is, translate the important domain-level concepts into logic-level names. This involves many questions of knowledge-engineering style. Like programming style, this can have a significant impact on the eventual success of the project. For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? Once the choices have been made, the result is a vocabulary that is known as the ontology of the domain

4. Encode general knowledge about the domain. The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.

5. Encode a description of the specific problem instance. If the ontology is well thought out, this step will be easy. It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors, whereas a “disembodied” knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.

6. Pose queries to the inference procedure and get answers. This is where the reward is: we can let the inference procedure operate on the axioms and problem- specific facts to derive the facts we are interested in knowing.

Thus, we avoid the need for writing an application-specific solution algorithm.

7. Debug the knowledge base. Alas, the answers to queries will seldom be correct on the first try. More precisely, the answers will be correct for the knowledge base as written, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting. For example, if an axiom is missing, some queries will not be answerable from the knowledge base.

- $\forall s \text{ Smelly}(s) \Rightarrow \text{Adjacent}(\text{Home}(\text{Wumpus}), s)$,

instead of the biconditional, then the agent will never be able to prove the absence of wumpuses. Incorrect axioms can be identified because they are false statements about the world. For example, the sentence

- $\forall x \text{ NumOfLegs}(x, 4) \Rightarrow \text{Mammal}(x)$

is false for reptiles, amphibians, and, more importantly, tables. The falsehood of this sentence can be determined independently of the rest of the knowledge base. In contrast, a typical error in a program looks like this:

4.4.2 The electronic circuits domain

- We will develop an ontology and knowledge base that allow us to reason about digital circuits of the kind shown in Figure 4.6. We follow the seven-step process for knowledge engineering.

1) Identify the task

- There are many reasoning tasks associated with digital circuits. At the highest level, one analyzes the circuit's functionality. For example, does the circuit in Figure 4.6 actually add properly? If all the inputs are high, what is the output of gate A2? Questions about the circuit's structure are also interesting. For example, what are all the gates connected to the first input terminal? Does the circuit contain feedback loops? These will be our tasks in this section. There are more detailed levels of analysis, including those related to timing delays, circuit area, power consumption, production cost, and so on. Each of these levels would require additional knowledge.

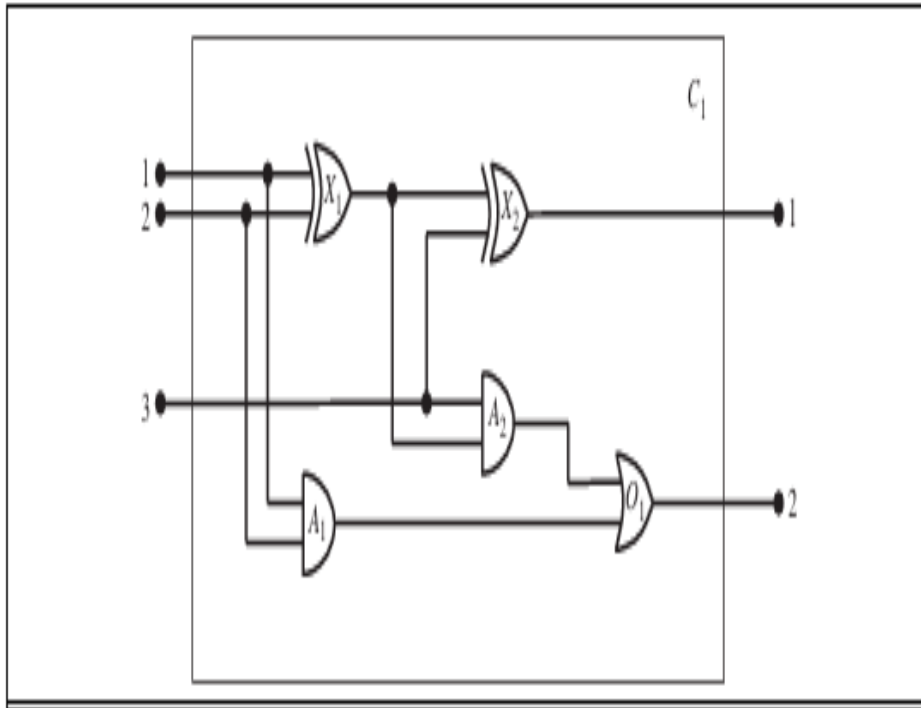


Figure 4.6: A digital circuit C_1 , purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates, and one OR gate.

2) Assemble the relevant knowledge
What do we know about digital circuits? For our purposes, they are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a signal on the output terminal that flows along another wire. To determine what these signals will be, we need to know how the gates transform their input signals. There are four types of gates: AND, OR, and XOR gates have two input terminals, and NOT gates have one. All gates have one output terminal. Circuits, like gates, have input and output terminals.



ATME

College of Engineering

GOLD

QS I-GAUGE
INDIAN COLLEGE RATINGS





A T M E
College of Engineering

GOLD

QS I-GAUGE
INDIAN COLLEGE RATINGS





ATME

College of Engineering

GOLD

QS I-GAUGE
INDIAN COLLEGE RATINGS



3) Decide on a vocabulary

- We now know that, we want to talk about circuits, terminals, signals, and gates. The next step is to choose functions, predicates, and constants to represent them.
- First, we need to be able to distinguish gates from each other and from other objects. Each gate is represented as an object named by a constant, about which we assert that it is a gate with,
- say, $\text{Gate}(X1)$. The behavior of each gate is determined by its type: one of the constants AND, OR, XOR, or NOT. Because a gate has exactly one type, a function is appropriate: $\text{Type}(X1) = \text{XOR}$. Circuits, like gates, are identified by a predicate: $\text{Circuit}(C1)$. Next we consider terminals, which are identified by the predicate $\text{Terminal}(x)$. A gate or circuit can have one or more input terminals and one or more output terminals.
- We use the function $\text{In}(1, X1)$ to denote the first input terminal for gate $X1$. A similar function Out is used for output terminals. The function $\text{Arity}(c, i, j)$ says that circuit c has i input and j output terminals. The connectivity between gates can be represented by a predicate, Connected , which takes two terminals as arguments, as in $\text{Connected}(\text{Out}(1, X1), \text{In}(1, X2))$. Finally, we need to know whether a signal is on or off.

4) Encode general knowledge of the domain

- One sign that we have a good ontology is that we require only a few general rules, which can be stated clearly and concisely. These are all the axioms we will need:

1. If two terminals are connected, then they have the same signal:

$$\forall t_1, t_2 \text{ Terminal}(t_1) \wedge \text{Terminal}(t_2) \wedge \text{Connected}(t_1, t_2) \Rightarrow \text{Signal}(t_1) = \text{Signal}(t_2)$$

2. The signal at every terminal is either 1 or 0:

$$\forall t \text{ Terminal}(t) \Rightarrow \text{Signal}(t) = 1 \vee \text{Signal}(t) = 0 .$$

3. Connected is commutative:

$$\forall t_1, t_2 \text{ Connected}(t_1, t_2) \Leftrightarrow \text{Connected}(t_2, t_1) .$$

4. There are four types of gates:

$$\forall g \text{ Gate}(g) \wedge k = \text{Type}(g) \Rightarrow k = \text{AND} \vee k = \text{OR} \vee k = \text{XOR} \vee k = \text{NOT} .$$

5. An AND gate's output is 0 if and only if any of its inputs is 0:

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{AND} \Rightarrow \text{Signal}(\text{Out}(1, g)) = 0 \Leftrightarrow \exists n \text{ Signal}(\text{In}(n, g)) = 0 .$$

6. An OR gate's output is 1 if and only if any of its inputs is 1:

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{OR} \Rightarrow$$

$$\text{Signal}(\text{Out}(1,g)) = 1 \Leftrightarrow \exists n \quad \text{Signal}(\text{In}(n,g)) = 1$$

7. An XOR gate's output is 1 if and only if its inputs are different:

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{XOR} \Rightarrow$$

$$\text{Signal}(\text{Out}(1,g)) = 1 \Leftrightarrow \text{Signal}(\text{In}(1,g)) \neq \text{Signal}(\text{In}(2,g)) .$$

8. A NOT gate's output is different from its input:

$$\forall g \text{ Gate}(g) \wedge (\text{Type}(g) = \text{NOT}) \Rightarrow$$

$$\text{Signal}(\text{Out}(1,g)) \neq \text{Signal}(\text{In}(1,g)) .$$

9. The gates (except for NOT) have two inputs and one output.

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) \neq \text{NOT} \Rightarrow \text{Arity}(g, 1, 1) .$$

$$\forall g \text{ Gate}(g) \wedge k = \text{Type}(g) \wedge (k = \text{AND} \vee k = \text{OR} \vee k = \text{XOR}) \Rightarrow$$

$$\text{Arity}(g, 2, 1)$$

10. A circuit has terminals, up to its input and output arity, and nothing beyond its arity:

- $\forall c, i, j \text{ Circuit}(c) \wedge \text{Arity}(c, i, j) \Rightarrow$
- $\forall n (n \leq i \Rightarrow \text{Terminal}(\text{In}(c, n))) \wedge (n > i \Rightarrow \text{In}(c, n) = \text{Nothing}) \wedge$
- $\forall n (n \leq j \Rightarrow \text{Terminal}(\text{Out}(c, n))) \wedge (n > j \Rightarrow \text{Out}(c, n) = \text{Nothing})$

11. Gates, terminals, signals, gate types, and Nothing are all distinct.

- $\forall g, t \text{ Gate}(g) \wedge \text{Terminal}(t) \Rightarrow$
- $g = t = 1 = 0 = \text{OR} = \text{AND} = \text{XOR} = \text{NOT} = \text{Nothing} .$

12. Gates are circuits.

- $\forall g \text{ Gate}(g) \Rightarrow \text{Circuit}(g)$

Inference in First Order Logic

4.5 Propositional Versus First Order Inference

- The idea of unification, showing how it can be use to construct inference rules that work directly with first-order sentences. We then discuss three major families of first-order inference algorithms.
- We begin with some simple inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers. These rules lead naturally to the idea that first-order inference can be done by converting the knowledge base to propositional logic and using propositional inference, which we already know how to do.
- The next section points out an obvious shortcut, leading to inference methods that manipulate first-order sentences directly.

- The rule of Universal Instantiation (UI for short) says that we can infer any sentence obtained by substituting a ground term (a term without variables) for the variable.1 To write out the inference rule formally. Let $SUBST(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α . Then the rule is written
- for any variable v and ground term g . For example, the three sentences given earlier are obtained with the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

In the rule for Existential Instantiation, the variable is replaced by a single new constant symbol. The formal statement is as follows: for any sentence α , variable v , and constant symbol k that does not appear elsewhere in the knowledge base,

For example, from the sentence
 $\exists x \text{Crown}(x) \wedge \text{OnHead}(x, John)$
we can infer the sentence
 $\text{Crown}(C1) \wedge \text{OnHead}(C1, John)$

4.5.2 Reduction to propositional inference

- Once we have rules for inferring non quantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference. The first idea is that, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of all possible instantiations. For example, suppose our knowledge base contains just the sentences

- $\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$
- $\text{King}(\text{John})$
- $\text{Greedy}(\text{John})$
- $\text{Brother}(\text{Richard}, \text{John})$.

Then we apply UI to the first sentence using all possible ground-term substitutions from the

vocabulary of the knowledge base—in this case, $\{x/\text{John}\}$ and $\{x/\text{Richard}\}$. We obtain
 $\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John})$
 $\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \Rightarrow \text{Evil}(\text{Richard})$,

and we discard the universally quantified sentence. Now, the knowledge base is essentially

propositional if we view the ground atomic sentences— $\text{King}(\text{John})$, $\text{Greedy}(\text{John})$, and so on—as proposition symbols.

4.6 Unification

- The preceding section described the understanding of first-order inference that existed up to the early 1960s. The sharp-eyed reader (and certainly the computational logicians of the
- early 1960s) will have noticed that the propositionalization approach is rather inefficient. For example, given the query $\text{Evil}(x)$ and the knowledge base, it seems per verse to generate sentences such as $\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \Rightarrow \text{Evil}(\text{Richard})$.
- Indeed, the inference of $\text{Evil}(\text{John})$ from the sentences
- $\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$
- $\text{King}(\text{John})$
- $\text{Greedy}(\text{John})$
- Seems completely obvious to a human being. We now show how to make it completely obvious to a computer

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called unification and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a unifier for them if one exists:

$\text{UNIFY}(p, q) = \vartheta$ where $\text{SUBST}(\vartheta, p) = \text{SUBST}(\vartheta, q)$.

Let us look at some examples of how UNIFY should behave. Suppose we have a query $\text{AskVars}(\text{Knows}(\text{John}, x))$: whom does John know? Answers to this query can be found by finding all sentences in the knowledge base that unify with $\text{Knows}(\text{John}, x)$. Here are the results of unification with four different sentences that might be in the knowledge base:

4.7 Forward Chaining

- The idea is simple: start with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made. Here, we explain how the algorithm is applied to first-order definite clauses. Definite clauses such as Situation $\Rightarrow$ Response are especially useful for systems that make inferences in response to newly arrived information. Many systems can be defined this way, and forward chaining can be implemented very efficiently.

4.7.1 First-order definite clauses

- First-order definite clauses closely resemble propositional definite clauses (page 256): they are disjunctions of literals of which exactly one is positive. A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses:
 - $\text{King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$.
 - $\text{King}(\text{John})$.
 - $\text{Greedy}(y)$

- Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified. (Typically, we omit universal quantifiers when writing definite clauses.) Not every knowledge base can be converted into a set of definite clauses because of the single- positive-literal restriction, but many can. Consider the following problem: 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 American.

We will prove that west is a criminal. First, we will represent these facts as first-order definite clauses. The next section shows how the forward-chaining algorithm solves the problem.

“...it is a crime for an American to sell weapons to hostile nations”:

$American(x) \wedge Weapon(y) \wedge Sells(x,y,z) \wedge Hostile(z) \Rightarrow Criminal(x)$.

“Nono ...has some missiles.” The sentence $\exists x Owns(Nono,x) \wedge Missile(x)$ is transformed into two definite clauses by Existential Instantiation, introducing a new constant M1:

$Owns(Nono,M1)$

$Missile(M1)$

- “All of its missiles were sold to it by Colonel West”:
 $\text{Missile}(x) \wedge \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono})$.
- We will also need to know that missiles are weapons:
 $\text{Missile}(x) \Rightarrow \text{Weapon}(x)$
- and we must know that an enemy of America counts as “hostile”:
 $\text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x)$.
- “West, who is American ...”:
- $\text{American}(\text{West})$.
- “The country Nono, an enemy of America ...”:
 $\text{Enemy}(\text{Nono}, \text{America})$.
- This knowledge base contains no function symbols and is therefore an instance of the class of Datalog knowledge bases. Datalog is a language that is restricted to first-order definite clauses with no function symbols. Datalog gets its name because it can represent the type of statements typically made in relational databases.



A T M E
College of Engineering

GOLD

QS I-GAUGE
INDIAN COLLEGE RATINGS



THANK YOU