Representing Knowledge in First-Order Logic

Much of the knowledge that we can informally express in ordinary language can be formally represented in first-order logic (FOL)

The previous lecture mentioned various examples of the many sorts of knowledge that people have – and that intelligent computers need in a form enabling not only fluent conversation, but also complex reasoning and planning.

Why logic?

The question then is how to *represent* this knowledge in a form that enables us to clearly understand its meaning (relationship to the things it is *about*), and hence to study methods of reasoning and planning that are well-founded. With this in mind, we will focus on first-order logic, which meets these criteria.

We may well ask, however, "Why not use ordinary language, e.g., English", since we're considering knowledge expressible in language?". From the above perspective, i.e., enabling the analysis of the meaning of a representation, and its use for sound reasoning and planning, it is clear that informal language won't do – it is too ambiguous and context-dependent.

Consider trying to formally anylyze the relationship of the words and phrases of the following sentence to the world, and specifying under what conditions it is true:

I saw the moons of Jupiter with a telescope last night.

We can see the following problems in attempting such a semantic analysis, and consequently, in figuring out how this sentence could be used for reasoning in a way that comes with formal guarantees of soundness:

- What "I" refers to in the world isn't fixed - it depends on who uttered the sentence.
- What "last night" refers to in the world isn't fixed – it depends when the sentence was uttered.
- Also, since the sentence as such is just a sequence of words, with no explicit structure, on what basis (other than intuition) can we assume that"last night" modifies the seeing-event, rather than the word "telescope" immediately preceding it? (Compare with, "I saw the headline of the D&C about the *accident last night*")?

- Similarly, how could we be sure that "with a telescope" modifies "saw" rather than "moons"? (I.e., it might be that some moons of Jupiter have telescopes on them, and those are the ones I saw!)
- The verb "saw" could refer to the action of "sawing", even though it makes no sense to use present tense for an action last night!
- In a different conext, "Jupiter" could refer to the god, rather than the planet.

We might say that FOL is a formalized, unambiguous form of ordinary language that avoids problems like the above. Its invention is one of the most important mathematical achievements of the 19th and 20th centuries, and has had an enormous impact on mathematics, philosophy, and AI.

Before proceeding to describe how FOL avoids such ambiguities (and thus enables forms of argumentation and planning that are demonstrably sound), we should note that the advent of LLMs has changed how we view the role of logic in AI. The remarkable property of LLMs is that they actually do figure out the *intended* meaning of sentences like the above. At least so it seems, since they generally behave reasonably in conversation and in other linguistic tasks assigned to them. But the problem of reliable reasoning and planning remains, so we would like to combine the context-aware linguistic fluency of LLMs with the kind of reliable reasoning enabled by FOL. We'll see quite a bit more about this "neuro-symbolic" integration problem later in the course, but for now, we need to learn how FOL makes precision possible. Once we undersand that, we can try to tackle the problem of bringing this kind of precision to DNN-based AI, without giving up the linguistic subtlety that these new machines can handle.

Basically, the way FOL avoids problems like the above is by

- using brackets and fixed operator-operand ordering to avoid syntactic ambiguity;
- allowing any one symbol (like "I" or "saw") to have just one interpretation (where "interpretation" is a precisely defined notion);[1]
- using formalized versions of *and, or, not, implies, is equivalent to,* and *is identical to*, as a means of combining information, where these operators have fixed, well-defined meanings;
- using *quantifiers* $\exists$, $\forall$ and *variables* like x, y, z to talk about an existing, but unspecified individual, or about all individuals.

In the early, somewhat arrogant days of AI, people working on KR often thought they could safely ignore logic as a representation, since after all they were working within a completely new "paradigm" – a new way of looking at reasoning and intelligence, namely in terms of *symbolic information processing.* They came up with many supposedly new

[1] More generally, logic avoids *context-dependent* meanings, evident in such sentences as *"He got one too"*, or questions like *"What about you?"*.

representations inspired by the computational needs of question-answering, and simple inferencing. But ultimately these representations turned out to be very, very close to FOL, but with a new terminology, and new ways of writing things down or drawing them as graphs. There *were* some genuinely new features; for example, graphical representations (and corresponding data structures) more clearly indicate how knowledge can be *accessed and used*; but in terms of *content* of the knowledge representation, the new notations tended to be a re-invention of FOL – but less precise!

In retrospect, this is not really surprising. *Any* representation for particular and general facts about the world surely needs at least the following devices:

- A way of referring to individuals that we want to say something about, and a way of saying that an individual has a certain property, or that certain individuals are related in a particular way (e.g., that Mary is single, or that she is married to John, or that Barack Obama was a president of the US);
- Ways of saying that a certain statement is true *and* a certain other statement is true as well; that one *or* another of two statements are true; or that a statement does *not* hold;
- A way of saying that all individuals (of a specified type) have a certain property (e.g., that all birds have wings).
- A way of saying that two things are identical (e.g., that 2 plus 2 equals 4, or that the Morning Star is the planet Venus)
- A way of referring to entities that are functionally determined by other entities (e.g., the *sum* of 2 and 2; the *weight* of an object; the *surface* of an object)

But taken together, these requirements virtually force you to adopt (at least) FOL as a representation! In fact, the last requirement is really technically redundant, as is the requirement for "or" ("and" together with "not" can express "A or B", viz., "not ((not A) and (not B))").

So, better to learn it than reinvent it! A lot is known about how to precisely characterize *meaning* in FOL, and how to perform inferences (see below), and it is both very difficult and a waste of time to keep rediscovering these things, in slightly different notations. That is not to say "FOL is all you need"; it isn't. But it's an important, fundamental part of what you need!

The syntax of first-order logic

The list of requirements above very directly motivates the syntax of FOL. The syntax allows us to form formulas similar to English sentences, such as Dog(Odie) ("Odie is a dog"), or Loves(Romeo,Juliet), or ($\forall$x (Dog(x) $\Rightarrow \neg$Can-talk(x))) ("For any object x, if x is a dog, it cannot talk", i.e., "Dogs can't talk"). We build such formulas systematically from smaller pieces, as follows.

Terms

First, the *basic terms* used to refer to individuals are

individual constants: A, B, C, USA, CSC244, John, Mary, Block1, Block2, ...[2]

and

individual variables: x, y, z, x1, y1, block1, block2, ...

(Since there are no other variables in FOL, we often just say "variables".) Two points should be noted about the interpretation of individual constants (informally speaking). First, we can let them refer to *any* individuals we please. So *A* or *John* (or both) could refer to John or to the US or to the number 17 or to the chair you are sitting on, etc. (assuming that these are things you wish to be able to talk about in your knowledge base). In this respect logic is unlike a natural language – in English we cannot use *John* to refer to countries, numbers, or chairs, except perhaps by special prior agreement, as some sort of secret code or joke. The second point is that since in logic meanings are unambiguous, a constant or variable denotes exactly one thing. This is also unlike ordinary language, since a name like *John* can refer to any number of individuals with that name, depending on context.

It can also be quite useful to allow numerals

0, 1, 2, ..., 10, 11, 12, ...

as individual constants.[3] This raises certain questions that we'll get back to later. For now let's just note that (given the above "freedom of interpretation") numerals could refer to people or countries, or anything else; in fact, 0 could refer to the number 17, etc. So at some point we presumably want to impose some constraints on the meanings of numerals.

For referring to the values of functions, we introduce

function constants: f, g, h, weight, father-of, surface-of, sum, ...

For each function constant, we have to decide how many arguments it takes – this is called its *arity* (or *adicity*). Thus *weight* might be a unary (or, monadic) function constant (arity 1), and *sum* might be a binary (or, dyadic) function constant (arity 2). But note that function constants can be interpreted to mean anything we want them to mean, just as in the case of individual constants. For instance, *weight* might actually refer to the surface area of an object, and *sum* to the distance between two given objects; however, we usually do not choose our constants so perversely! Using functions, we can form

function terms: weight(John), sum(weight(x),weight(y)), ...

[2]Brachman & Levesque instead use lower case here, perhaps influenced by Prolog conventions, but we will follow our natural instinct to capitalize names, and use lower case for variables!

[3]Genesereth & Nilsson allow this, while Brachman & Levesque don't – though in ch. 4 they use 0, 1, 2, ... to abbreviate *Zero, succ*(*Zero*)*, succ*(*succ*(*Zero*))*, ...*

In general, we can recursively define **terms** as being either individual constants, or individual variables, or functions applied to an appropriate number of terms. Terms containing no variables are called **ground terms**, while others are **non-ground terms**.

An important technical point here is that we assume that functions always have values, no matter what objects they are applied to. (They are *total* functions.)

weight(17), father-of(Pacific-Ocean)

are presumed to have values, perhaps chosen in some arbitrary way since we don't really expect to refer to these values.

In addition to ordinary function constants like those above, some texts (G & N) also allow certain mathematical function symbols, such as

$., +, -, *, /, \uparrow, \cup, \cap.$

These constants have certain "intended interpretations", and can be written in infix rather than prefix form; e.g.,

(A . x), (A + 2), etc.,

rather than .(A,x), +(A,2), etc. (“.” is intended to refer to addition of an initial element to a sequence, like the Lisp *cons* function.) However, unless we somehow define or constrain the interpretations of these function constants, they can in principle refer to any functions at all. (We'll come back to that.)

Formulas

Formulas (also often called *sentences*) describe properties and relationships (including identity) of objects. Thus, besides terms referring to objects, we also need symbols referring to properties and relations. These are called

predicate constants: A, B, C, Dog, Person, Loves, Married-to, Smokes, ...

These again have to have a fixed arity, such as 1 in the case of *Dog* and 2 in the case of *Loves* (or perhaps 3, allowing for a time or situation where the relation holds). And again these predicate constants can refer to any properties and relationships we like, not necessarily the ones we would expect from their resemblance to English.

Note that we allow any capitalized (or completely upper-case) symbols for both individual constants and predicate constants. However, any one symbol can be used in only one way. Similarly a lower-case symbol can be used as a variable, individual constant, or function constant, but not simultaneously. Also note that we need not explicitly specify whether something is an individual constant, function, or predicate – it can be inferred from the way we *use* these symbols (if, in fact, we do use them consistently). Similarly, the arity of functions and predicates is evident from the way we use them.

We should note here that we also allow the *equality predicate*, =, but rather than

being freely interpretable, this predicate has a fixed meaning: it holds only for a pair of arguments that refer to the same object. As in the case of binary mathematical functions, we use infix form (possibly surrounded by brackets, if there is ambiguity)

A = B, etc.,

for equality statements, rather than prefix form =(A,B), etc. The presence of the equality predicate distinguishes FOL from the *First-Order Predicate Calculus* (FOPC), i.e., FOL is FOPC plus equality. We may also provide some additional mathematical predicates (as G & N but not B & L do), namely

$<,\ >,\ \leq,\ \geq,\ \in,\ \subset,\ \supset,\ \subseteq,\ \supseteq$.

But again we should not be misled by the "familiar look" of these predicates – unlike =, they are in principle freely interpretable. They won't automatically "mean what we think they mean" (unless we constrain their meanings somehow).

Using predicates and terms, we can now form

atomic formulas: Dog(Fido), Loves(Romeo,Juliet), f(x) = y, etc.

Formulas containing no variables are called **ground formulas**.

There are two general ways of forming complex formulas from atomic formulas, namely, by use of logical connectives and by use of quantifiers. We use the following

logical connectives: $\neg,\ \wedge,\ \vee,\ \Rightarrow,\ \Leftarrow,\ \Leftrightarrow$.

Intuitively these mean *not, and, or, implies, implied by,* and *if and only if* (*equivalent to*) respectively. "$\Rightarrow$" may also be written as "$\supset$" and "$\Leftrightarrow$" as "$\equiv$" (as is done in B & L).[4] $\neg$ is a unary (1-place) logical connective, while the others are binary (2-place) logical connectives, and as such allow formation of compound sentences such as

$\neg$Likes(Merkel,Putin), Loves(Romeo,Juliet)$\wedge$ Loves(Juliet,Romeo),
At-home(Mary)$\vee$ At-work(Mary), Poodle(Fifi)$\Rightarrow$ Dog(Fifi),
(A > B)$\Leftrightarrow$ (B < A).

Finally, we introduce the two

quantifiers: $\forall,\ \exists$.

Their use is illustrated by the following examples:

($\forall$x Entity(x)), ($\forall$x (Poodle(x) $\Rightarrow$ Dog(x))),
($\exists$x Robot(x)), ($\exists$x (Robot(x) $\wedge$ Smart(x))),
($\forall$x (Robot(x) $\Rightarrow$ ($\exists$y Built(y,x)))),
($\exists$x (Robot(x) $\wedge$ ($\forall$y (Robot(y) $\Rightarrow$ x=y)))).

These can be read respectively as *Everything is an entity*; *Every poodle is a dog*; *There*

[4] We prefer "$\Rightarrow$" and "$\Leftrightarrow$" because they are ascii-printable and can be Lisp atoms; also "$\supset$" can be confused with "superset", if we want to use such a relation.

exists a robot; *Some robot is smart* (Note that this requires $\wedge$ rather than $\Rightarrow$!); *For every robot, there is someone (or something) that built this robot*; and, *There is only one (i.e., exactly one) robot.*

As you see, a quantifier is always followed immediately by a variable, and is said to *bind* that variable. The quantifier and variable are immediately followed by a formula, called the *scope* of the quantifier. The quantifier, variable, and scope are enclosed in parentheses, except where no ambiguity can arise.

An occurrence of a variable ν in a formula is said to be a **free** occurrence if it is not in the scope of any $\forall$- or $\exists$-quantifier that binds ν. A formula that contains no free variables is called a **closed formula**. The term **sentence** is also frequently reserved for closed formulas (though as noted it is commonly used for arbitrary formulas as well). Note that *ground formulas* are always closed, but some closed formulas are not ground formulas, because they contain quantifiers.

BNF notation; first-order logic and first-order languages

We can now summarize and formalize these syntactic devices as follows, using "BNF" notation (ignoring the infixed, mathematical functions and relations, and also ignoring correct correspondence between the arity of function and predicate constants, and the number of arguments to which they are applied):

⟨**individual constant**⟩ ::= A | B | C | USA | CSC244 | John | Block1 | Block2 | ...
⟨**variable**⟩ ::= x | y | z | x1 | x2 | block1 | ...
⟨**function constant**⟩ ::= f | g | h | weight | sum | mother-of | ...
⟨**term**⟩ ::= ⟨**individual constant**⟩ | ⟨**variable**⟩ |
⟨**function constant**⟩ (⟨**term**⟩ , ..., ⟨**term**⟩)
⟨**predicate constant**⟩ ::= A | B | C | Dog | Loves | Owes | ...
⟨**binary connective**⟩ ::= $\wedge$ | $\vee$ | $\Rightarrow$ | $\Leftarrow$ | $\Leftrightarrow$
⟨**formula**⟩ ::= ⟨**predicate constant**⟩ (⟨**term**⟩ , ..., ⟨**term**⟩) | (⟨**term**⟩ = ⟨**term**⟩)
| $\neg$⟨**formula**⟩ | (⟨**formula**⟩ ⟨**binary connective**⟩ ⟨**formula**⟩)
| ($\forall$⟨**variable**⟩ ⟨**formula**⟩) | ($\exists$⟨**variable**⟩ ⟨**formula**⟩)

Note that expressions of form ⟨**....**⟩ above are *metalinguistic* symbols – they vary over expressions in the formalism being defined, rather than being part of that formalism.

Also, we are still not being perfectly precise: function and predicate constants should really be sorted into 1-place, 2-place, 3-place, ... constants (also called modadic, dyadic, triadic, ..., or unary, binary, ternary, ...). Furthermore, a given alphanumeric string may only be used as one type of constant, with a fixed adicity (arity). In other words, the sets comprising the individual constants, the 1-place function constants, the 2-place function constants, ..., the 1-place predicate constants, the 2-place predicate constants, etc., are *disjoint sets.*

As a final point, observe that depending on exactly what individual constants, function

constants and predicate constants we choose, we can get various **first-order languages** using the above schema. For instance, we might limit these constants to some finite sets, rather than having an unlimited supply of them. So FOL, as such, is not itself a unique *language*, but rather it is a *formalism* that allows for many first-order languages.

What FOL can and can't do

We've indicated that logical representations provide an unambiguous way of capturing *factual* knowledge, of the sort we can readily express in words. However, though it is derivative from natural language, it is much less flexible in the forms it allows, in avoiding idioms and metaphors, and also misses some important semantic capabilities (like talk about beliefs or intentions that are common in language – and are added in some extensions of FOL.) But it's an excellent place to start if we're interested in how well-founded reasoning is possible. While our next major task is to spell out the semantics (meaning) of FOL, much of the rest of the course will be concerned with the *use* of FOL for reasoning.

Factual knowledge and types of inference

When logicians talk about FOL, they generally have in mind the above syntax (perhaps leaving out some inessential features), the semantics we shall discuss, and the "deductive proof theory". The deductive proof theory concerns the rules by which we derive true conclusions from true premises, and we will spend a good deal of time on this. However, it would be a mistake to think that in using FOL we are somehow restricted to using deduction for inference (a mistake quite often made by people who argue against the use of logical representations!) There are other important forms of inference besides deduction that FOL lends itself to, and it is worth looking at a list of these before we delve into semantics. We will eventually learn something about several of these, though with the emphasis on the first:

- *Deduction*: deriving logically necessary consequences
  - Mary has a poodle named Fifi. Therefore (given that all poodles are dogs), Mary has a dog.
  - Fifi's mother is Lulu. Therefore (given that offspring are younger than their parents, and that mothers are parents), Fifi is younger than Lulu.
  - Fifi ate a cookie. Therefore (given certain facts about eating, which you can fill in), the cookie went inside Fifi.
- *Uncertain and nonmonotonic inference*: deriving likely consequences in the absence of information to the contrary
  - Given (only) that Tweety is a canary, tentatively conclude that Tweety flies.

  - Given that John is sniffling and has a sore throat but no fever, tentatively conclude that he has a cold. (In either example, further evidence may reverse the conclusion – that's what "nonmonotonic" means.)

- *Analogical, pattern-based "mimetic" inference*: People do it with just a few or even a single example; Deep neural nets generally need thousands or millions of examples, but end up doing it very well.
  - Suppose a 2-year-old encounters her first dog, and it comes up to her and wags its tail. When she encounters another similar-looking dog later, she anticipates that it will act similarly;
  - In "The Princess Bride", the hero says to the villain: "Hello, my name is Inigo Montoya. You killed my father. Prepare to die." He fails at first, but after a drawn-out battle, has the villain at his mercy and says, "My name is Inigo Montoya. You killed ..." how does this continue? Yes, you guessed it.
  - Arguably, our general world knowledge consists largely of *schemas* describing commonly encountered patterns of events and relationships, and we use these for prediction and retrodiction. For example, we can all describe the actions and entities involved in dining at a restaurant, brushing one's teeth, sweeping a floor, ordering from Amazon, tanking up a car, attending a class, etc. This knowledge tells us both how to act to achieve goals, and how to predict or retrodict events when observing others engaged in such routine activities.

- *Abduction* (and *induction*): formulating general hypotheses to account for multiple particular observations or facts; (induction: confirming or disconfirming such hypotheses based on observations)
  - Given that all the crows I've seen are black, conjecture that all crows are black. (Induction: gradually confirm the hypothesis after seeing more and more examples, and no counterexamples.)
  - Upon noticing that it is possible to "fit" ellipses to observations of the motions of several planets, conjecture that all planets follow elliptical orbits (Kepler).

- *Explanation*: postulating particular facts which, when combined with other available knowledge, can account for new observed facts (this is sometimes counted as a special case of abduction; it also shades over into uncertain, nonmonotonic inference, or schema-based inference).
  - Given that my parked car has disappeared, conclude that it was stolen or towed.
  - Given the observation of a slight, periodic wobble in the position of a certain star, conclude that a large planet is orbiting it.

- *Planning and plan recognition*: formulating plans to achieve given goals; and conjecturing other agents' plans based on their actions or stated desires; again this relates to schemas, which include routine plans one can employ to achieve one's recurring goals.

– Given her goal of being in Baltimore in early November, a professor decides to book a flight, arrange for a TA to sub for her class, etc.
– As I am about to pull out of my parking spot at a busy bank, I see another car stop and wait behind me. I infer that the driver wants to park in my spot, and (probably) go to the bank.

"Know-how" and the limits of FOL

Does the above list of inference modes cover the full range of reasoning styles that people are capable of?

It seems clear that the answer is "no": we possess various specialized skills and know-how, including many motor skills and sensory processing abilities, that appear to have nothing to do with reasoning. Rather, they are best viewed as procedures operating on specialized data structures and I/O streams, and perhaps running on very specialized computational architectures. Imagine trying to implement any of the following activities in a humanoid robot using logical representations and operations as a basis:

- whistling
- riding a bike, catching a ball, using chopsticks, tying shoelaces
- interpreting retinal images
- speaking grammatically (well, this example is debatable...)
- learning a new language, a new skill, ...

Even if we could come up with logical descriptions of how we do these things (or how a humanoid robot could do them), such descriptions probably could not be used in any direct way in implementing them.

It would be wrong to conclude, though, that there is a sharp division between knowledge of how to do things – *procedural* knowledge – and factual knowledge. (This has tended to be the view taken by both sides in the "proceduralist-declarativist controversy" – a controversy that started in the late 60's, with one side defending procedural representations and the other logical representations of knowledge.) For instance, consider the following kinds of procedural knowledge:

- cooking recipes
- assembly and installation instructions (for furniture, electronic equipment, etc.)
- emergency procedures (on airplanes, in medical emergencies, etc.)

etc. These are just as natural to express in language – and FOL – as facts about food, furniture, airplanes, etc. The difference is really just one of "mood": declarative (or "indicative") *vs.* imperative. Declarative sentences express facts while imperative sentences

express what to do, but apart from that, they use the same grammatical constituents (noun phrases, verb phrases, etc.), with the same meanings.

In fact in a broad sense, standard programming languages such as C++ can be viewed as logics, but ones that make (mostly) imperative rather than declarative statements.[5] Why, then, is it that programs in most programming languages look so little like FOL? Well, there are several reasons. First, the objects and operations they talk about tend to be ones inside the computer (storage locations, variables, assignment) or abstract mathematical ones (numbers, symbolic objects, subroutines), instead of being real-world things like food and cooking and furniture and so on. Second, the concern in procedural languages is very much with sequencing in time, while FOL makes no special provision for this (any more than for, say, sequencing in a spatial dimension – though it is entirely possible to "talk about" time and space). Third (a point related to the second), they use syntactic devices not seen in FOL – e.g., *begin–end* blocks, *if–then* statements, procedure declarations, loops, lambda-abstraction, etc. And at the same time, they tend not to have quantifiers $\forall$ or $\exists$. Still, the meaning of programs can be studied in much the same way as the meaning of declarative logics.[6] As well, certain languages have been developed, called *dynamic logics* and *executable temporal logics*, which have a more FOL-like syntax and can be used as high-level programming languages (or to describe what programs written in other programming languages do); e.g., see H. Barringer, M. Fisher, D. Gabbay, R. Owens and M. Reynolds, *The Imperative Future: Principles of Executable Temporal Logic*, Research Studies Press LTd., 1996.

As a final comment, it should be noted again that FOL is not expressive enough to cover everything we can easily express in ordinary language. For instance, it lacks good means for expressing modifers such as "very", "fake" (as in "fake blood"), "almost" (as in "he almost fell"), and temporal modifiers (as in "John woke up repeatedly during the night"), or for expressing belief (as in "Mary believes that John is an insomniac"), or for expressing "reification" – turning a concept or proposition into a thing (as in "Climbing rock walls without ropes is foolhardy", or "That there is water on the Moon is surprising"). We may look at some useful extensions to FOL later in the course.

[5] Prolog gives the illusion of being declarative, but is still interpreted imperatively, i.e., there is an "understood" way of executing a prolog program.

[6] E.g., see E. Stoy's book *Denotational Semantics*.