Knowledge representation in first order logic

The state space search algorithms covered earlier had a relatively general formulation, but required the problem to be represented in a specific format. This format included the definition of the state space, the set of state transition operators, and a heuristic state evaluation function.

Generally, the structure and format of knowledge representation are highly important and affect the efficiency — or even the ability — of searching for the solution.

There exist a number of paradigms for knowledge representation in artificial intelligence. These knowledge representation paradigms usually come with associated with them algorithms for **reasoning**, ie. making subsequent findings ultimately leading to determining the solution to the problem.

One of the most powerful and popular knowledge representation schemes is the language of mathematical logic.

Why is mathematical logic a good representation language in artificial intelligence?

On the one hand, it is close to the way people think about the world and express their thought in natural language. People even view their way of thinking as "logical". The categories by which people think and speak include such constructs as: objects and relations between them, simple and complex assertions, sentences, connectives, conditionals, and even quantifiers.

On the other hand, the mathematical logic offers a precise apparatus for reasoning, based on theorem proving. People, likewise, use logical reasoning in their thinking, so mathematical logic seems to be a good representation platform for the knowledge base of an intelligent agent, whose way of expressing facts and reasoning should be similar to the human's.

Example: the wumpus world

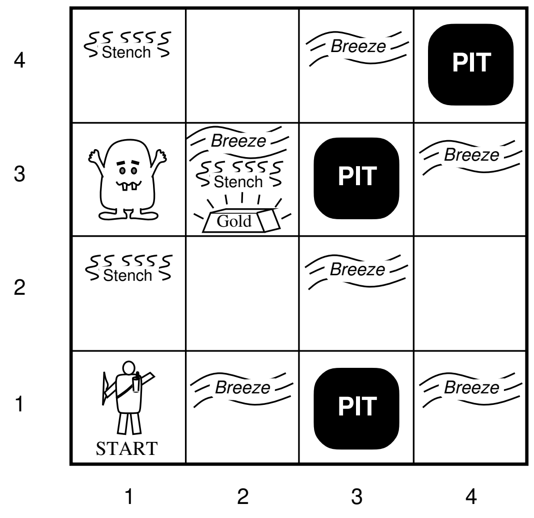
It is useful to have a good testing environment for verifying the methods being developed. This environment needs to be simple enough to allow developing intuitions and quickly discovering properties, but at the same time rich enough to pose some significant demands of the problem solving abilities, and allow to formulate problems of various degree of difficulty.

One of such "textbook" testing environment is the **wumpus world**.[1] An intelligent agent moves around this environment in search for gold, which she intends to carry out safely. The agent is however faced with some dangers, such as the *pits*), into which she may fall, and the title *wumpus* monster, which may eat the agent.

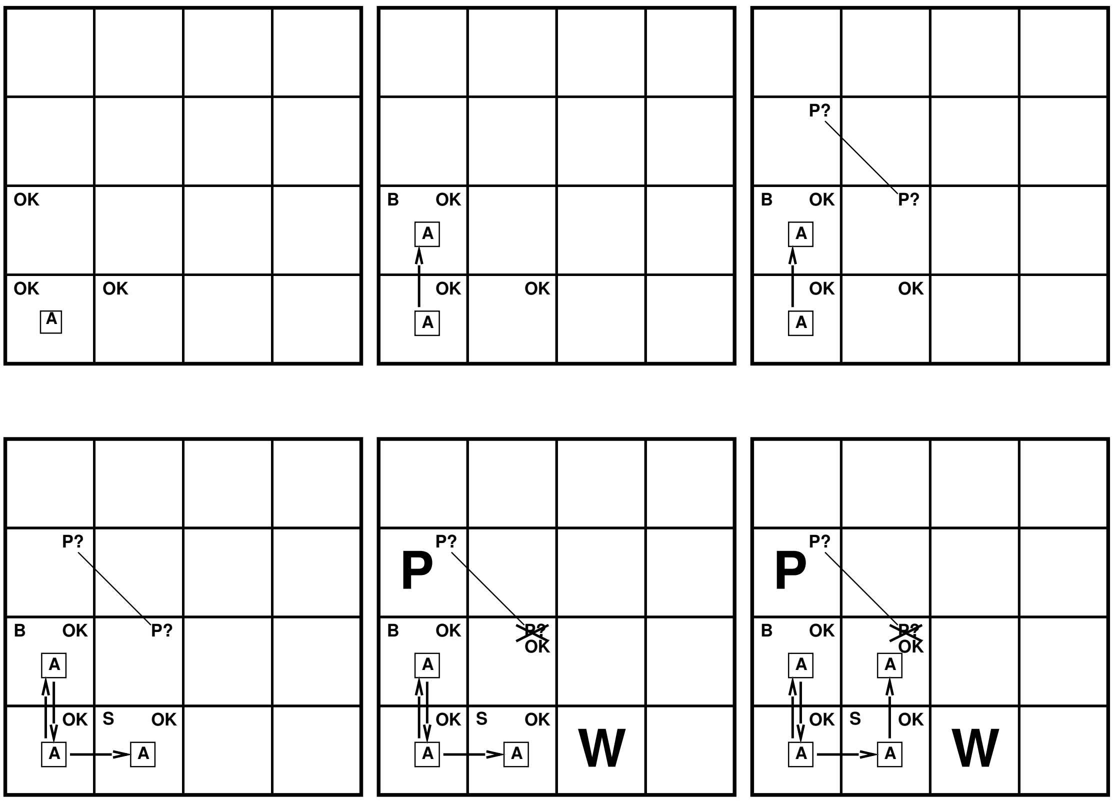
The agent may only turn right or left, move forward by one step, shoot a single arrow from a bow (ahead), pick up gold, and leave the environment when she is in the starting position.

[1]The examples and diagrams of the wumpus world presented here are borrowed from the textbook by Russell and Norvig "Artificial Intelligence A Modern Approach" and the materials provided on Stuart Russell's Web page.

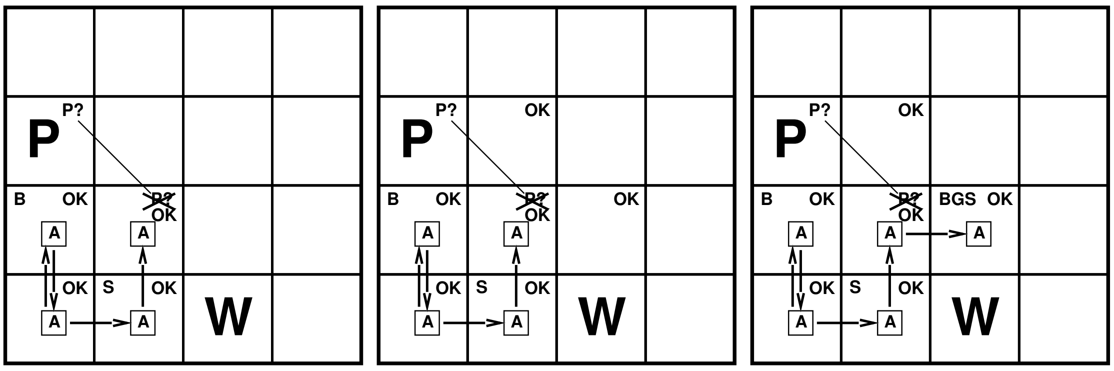
The agent receives some information about her environment (the data obtained by the agent by her perception are called the **percepts**). She can smell the wumpus *stench* and feel the *breeze* from the pits, but only in the fields directly neighboring the wumpus or the pits. She can also detect the *gold*, but only when she enters the field it is in. She cannot determine her absolute position (*à la* GPS), but she can remember her position and covered trail. She can sense the walls only by trying to enter them, which results in getting bumped back.



Example: moving around the wumpus world

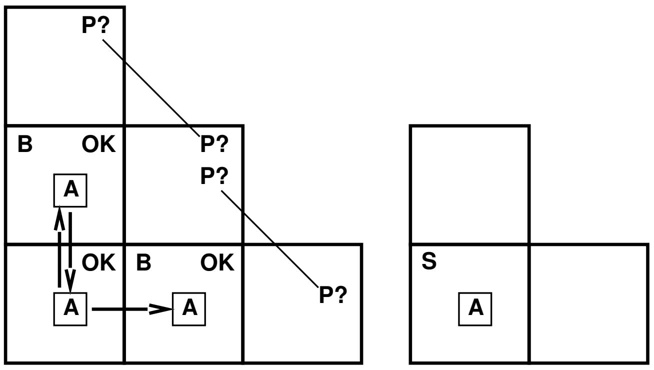


Example: moving around the wumpus world (cntd.)



However, it is not always possible to act so efficiently in the wumpus world by using only logical reasoning.

In some cases the only solution is to "shoot", ie. blindly select a move, and analyze the outcome. Provided that we survive!!



First order predicate calculus — terms

The **terms** represent objects in the language of logic and may be: constants (denoting a specific object), variables (can assume the values of various objects), or functions (determine an object from the value of their object argument(s), or map some objects into some others).

Examples of terms: A, 123, x, $f(A)$, $f(g(x))$, $+(x, 1)$

By convention, we will write constant terms in capital letters, and variables in lowercase.

Let us make a note, that the last term in the above examples is an indirect notation of the subsequent value for x, and not a subtraction. In pure logic there is no arithmetic. We will see the consequences of this often.

First order predicate calculus — predicates

The **predicates** represent relations over the set of terms. We can treat them as functions assuming the values of true or false (1 or 0), assigning 1 to each vector of n terms satisfying the relation, and 0 to each vector of n terms not satisfying the relation.

A predicate symbol written with the set of terms is called an **atomic formula**.

Examples of atomic formulas: P, $Q(A)$, $R(x, f(A))$, $> (x, 10)$

The expression $> (x, 10)$ is the functional equivalent of $x > 10$. In arithmetic we treat such an expression as inequality and we could solve it. But as a logical formula we can only **evaluate** it, meaning determine its truth value. But if a formula contains a variable then often its truth value cannot be determined.

Representing facts with logical formulas

What is the purpose of the predicate language?

We could use it to write the facts we want to express, like:

$$\begin{array}{l} At(Wumpus, 2, 2) \\ At(Agent, 1, 1) \\ At(Gold, 3, 2) \end{array}$$

The selection of the set of symbols, by which we intend to describe the objects and relations of some world is called **conceptualization**. For example, an alternative conceptualization for the above facts could be the following:

$$\begin{array}{l} AtWumpus(loc(2, 2)) \\ AtAgent(loc(1, 1)) \\ AtGold(loc(3, 2)) \end{array}$$

These two conceptualizations are similar, but have different properties. For example, in the latter the wumpus, agent and gold are not mentioned directly. In general, the accepted conceptualization has influence on the ease or even the ability to express different facts about the problem domain.

Representing facts with logical formulas (cntd.)

A problem with the conceptualization of the wumpus world is the description of the presence and location of the pits. We could give the pits full citizenship and identity:

$$At(Pit_4, 3, 3)$$

In this way it would be easy to describe the "bird's eye view" of the wumpus world, by giving different pits some names (constant terms). But from the point of view of the agent acting in the wumpus world this conceptualization is very uncomfortable. It would be hard to describe the world as it is gradually learned, when at first the agent does not even know the total number of pits. The presence of a pit at some location would have to be described by a variable:

$$At(x, 3, 3)$$

Unfortunately, this description does not indicate that x is a pit so this requires further descriptions. A comfortable alternative is to view the pits as anonymous, and only denote the presence or absence of pits at specific locations:

$$\begin{array}{l} PitAt(3, 3) \\ NoPitAt(1, 1) \end{array}$$

Logical connectives and complex formulas

Complex formulas can be constructed from atomic formulas using the **logical connectives**: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$. As a special case, an atomic formula or a negation of an atomic formula is called a **literal**.
Examples of complex formulas (the first one is a single literal):

$$\neg At(Wumpus, 1, 1)$$
$$PitAt(2, 1) \vee PitAt(1, 2)$$
$$[At(Agent, 1, 1) \wedge PitAt(2, 1)] \Rightarrow Percept(Breeze)$$

Let us observe that the last formula is of a different nature. The first two could be a fragment of a world description obtained or constructed by the intelligent agent during her activity in the wumpus world. But the last one expresses one of the laws of this world. The agent knows this law and can hold such a formula in her knowledge base.

The facts generally true in a problem domain are called the **axioms of the world**. The facts describing a specific instance of the problem are called **incidental**.

It is further worth noting, that the $\Rightarrow$ and $\Leftrightarrow$ are just logical connectives, transforming any two formulas into a complex formula. They are not part of the reasoning processes, which we will consider later.

Quantifiers

The complex formulas can also be built using the **quantifiers**: $\forall, \exists$, which **bind** variables in formulas. The general scheme for the formula with a quantifier is:

$$\forall x P(x)$$

A variable not bound by a quantifier in a formula is called **free**. The formula:

$$\exists y Q(x, y)$$

contains two variables, one free (x) and one bound by a quantifier (y).

A **sentence**, or a **closed** formula is a formula without free variables.

Examples:

$$\exists x, y\ At(Gold, x, y)$$
$$\forall x, y\ [At(Wumpus, x, y) \land At(Agent, x, y)] \Rightarrow AgentDead$$
$$\forall x, y\ [At(Wumpus, x, y) \land At(Agent, -(x, 1), y)] \Rightarrow Percept(Stench)$$

Let's note that the $-(x, 1)$ is an indirect notation of the column left of x, and not a subtraction. There is no subtracting in logic.

Short review

1. Work out a complete representation for the wumpus world in the first order predicate calculus. That is: introduce term symbols (constants and functions), and predicate symbols necessary to describe problem instances for the domain.

   Note: we do not consider the process of searching for the solution, analyzing alternative moves and their consequences, describing sequences of steps etc. We only seek a scheme for describing static snapshots of a problem instance.

2. Using the representation worked out in the previous question, describe a problem instance given on page 4.

3. Try to write the axioms for the wumpus world, that is, the general rules for this domain.

Semantics

The definition of the predicate language introduced only one component of a representation, which is its **syntax**. The second component is the **semantics**, which is the apparatus for describing the meaning.

It might seem that some formulas are self-explanatory. So far we intuitively guessed the meaning of $At(Agent, 2, 2)$. In general, however, this is insufficient. Even restricted to the wumpus world, we do not know which instance and which state of the game the formula refers to.

An **interpretation** is an assignment of objects from some domain to the syntactic elements (terms and predicates) of the predicate language.

Obviously, a formula written with a few symbols can refer to different domain. For example, the formula $At(Agent, 2, 2)$ can refer to some episode of the wumpus world, a scene from a James Bond movie, some real world agent, of something different still. There could be very many interpretations for a given formula.

Models

Let us note, that the interpretation determines the truth value of atomic formulas. If in the domain of the interpretation a relation holds between some objects, then the atomic formula describing this using appropriate terms is true under this interpretation.

Also conversely, having any written formula, we can determine its truth value by checking whether in the domain of the interpretation the relation described by the formula holds. (However, if a formula contains free variables then its truth value may not be determined by an interpretation.)

Consequently, from the definitions of the logical connectives and quantifiers we can determine the truth value of any (closed) formula for a given interpretation. We can say that interpretations assign truth values to formulas (at least closed).

An interpretation assigning a formula the value true is called a satisfying interpretation, or a **model** of this formula.

Satisfiability

A formula is called **satisfiable** if there exists a satisfying interpretation for it (in other words: there exists its model). A formula is **unsatisfiable** if there exists no satisfying interpretation (model) for it. If a formula has the truth value of 1 for every interpretation, then it is called a **tautology**, or a **valid** formula.

Consider examples:

$$At(Wumpus, 2, 2)$$
$$\exists x, y\ At(Gold, x, y)$$
$$\forall x, y\ [At(Wumpus, x, y) \wedge At(Agent, x, y)] \Rightarrow AgentDead$$

All above formulas are satisfiable, as we can easily make up an instance of the wumpus world, where the corresponding facts would hold. None of them are tautologies though. The latter two formulas hold for every episode of the wumpus world adhering to the rules of this world. However, it is easy to think of another, similar world, where they would not hold.

An example of a valid formula (tautology) can be the less interesting, though probably fascinating from the mathematical point of view formula: P∨¬P, where P is any 0-argument predicate, or, in other words, a statement of some fact.

Short review

1. For the predicate calculus formulas given below answer whether the formula is: satisfiable, unsatisfiable, a tautology.
   (a) P
   (b) $P(x)$
   (c) $\forall x\ P(x)$
   (d) $\exists x\ P(x)$
   (e) $[P \Rightarrow Q] \wedge P \wedge \neg Q$
   (f) $[P \Rightarrow Q] \Leftrightarrow [\neg P \vee Q]$

Logical following

In many situations an agent would like to conduct some reasoning. A typical case is to determine some facts, of which the agent has no information, but which can follow from the information she does have. We saw these cases in the wumpus world examples. Another case might be of the agent trying to determine the possible consequences of her actions; the desired as well as undesired.

We would like to have an effective way of verifying if some facts follow from others. However, logic only allows to determine the following of formulas.

A formula φ **logically follows** from the set of formulas Δ if every interpretation satisfying all the formulas of Δ also satisfies the formula φ. This is denoted by:

$$\Delta \models \varphi$$

Why do we define the following this way? Because we want to have a universal logical apparatus, for reasoning correct in all possible interpretations. While working with formulas, we want to be sure that our reasoning process is correct also for the specific problem domain of the agent.

A digression — the propositional calculus

There is a logical formalism simpler than the predicate calculus, called the **propositional calculus**. It does not have the terms, so the predicates are reduced to 0-argument predicates, called propositions. The atomic formulas are single relation symbols, and the complex formulas can be constructed using the connectives. There are no quantifiers.

The semantics of the propositional calculus is very simple. Instead of considering all possible interpretations for the given formula, it suffices to divide them into groups, which assign truth or false values to the specific propositional symbols. In fact, instead of considering interpretations, we can just consider all combinations of truth and false values for all propositional symbols. A simple procedure for this uses truth tables.

Such procedure does not work for the predicate calculus, as the truth values of an atomic formula depends on the values of the argument terms. Unfortunately, the propositional calculus is too weak (not **expressive** enough) for practical applications.

Inference rules

However, verifying the logical following by interpretations and unsatisfiability can be tedious. This is because all possible interpretations need to be examined, and there could be very many of them.

In mathematics, instead of checking the logical following, a different approach of **theorem proving** is used. It is done by introducing **inference rules**, which create new formulas from the existing ones by way of syntactic transformations.

For example, one of the most basic inference rules is the following, called the *modus ponens*, or the derivation rule:

$$\frac{\varphi, \varphi \Rightarrow \psi}{\psi}$$

For example, if the agent had the facts: $Drought$ i $Drought \Rightarrow LowCrops$, then inserting them into the above schema, she would be allowed to inference a new fact: $LowCrops$.

Theorem proving

The **proof** of the formula φ for the set of formulas Δ, called the **axioms**, we will call a sequence of formulas of which the last one is φ, where each formula in the sequence satisfies one of the following conditions:

1. is a tautology,
2. is one of the axioms,
3. is a formula derived from the formulas preceding it in the proof (located to the left of it) using one of the inference rules.

A **theorem** of a set of formulas Δ we will call each formula φ which has a proof for Δ. We the say that the formula φ can be **inferred** from the set of axioms Δ and denote this by:

$$\Delta \vdash \varphi$$

The set of all theorems of the set of formulas Δ is called the **theory** of this set, and is denoted by $\mathcal{T}(\Delta)$.

Reasoning by proving theorems

We initially introduced the logical following ($\Delta \models \varphi$) as a way of reasoning, ie. determining the fact following. However, we lacked an efficient procedure to verify this.

Theorem proving ($\Delta \vdash \varphi$) offers a potentially good reasoning procedure for an agent using logic as a representation language. Wanting to prove some fact from the set of axioms we can at worst systematically generate all finite sequences of formulas, satisfying the definition of the proof of this fact. If a proof exists, and has the length N, then it will be generated by this procedure. This is significantly better than checking all interpretations, some of which are even hard to imagine.

But is theorem proving as good a method for checking the fact following for any real problem domain? This is by no means obvious.

Proof systems

A number of **proof systems** have been defined in mathematical logic, which introduce alternative sets of inference rules, along with certain initial formulas (axioms). We will not cover the construction of such systems in this course. However, we need to know and understand their properties, since we will learn to use one of such systems.

An inference rule is **sound** if it allows to infer the false formula only from an unsatisfiable set of premises.

A set of inference rules is **complete** if it allows to infer the false formula from any unsatisfiable set of premises.

In a proof system which is sound and complete, false formula can be inferred from a set of clauses if and only if this set is unsatisfiable (provided these are closed clauses).

What would be needed in artificial intelligence, is a sound and complete proof system, with a computationally efficient proof generator.

Short review

1. Explain why the reasoning of an agent about her world should be based on logical following.

2. What is theorem proving and what role in it have the inference rules?

3. When can we apply theorem proving for determining properties of some problem domain?

The disjunctive normal form (DNF)

The automatic processing of logical formulas requires writing them in some uniform **normal forms**.

We will call **literals** any atomic formulas, and their negations, eg.: $P, \neg Q(x, f(a))$.

A formula is in the **Disjunctive Normal Form (DNF)** if it has the form of an alternative of conjunctions of literals. Both alternative and conjunctions here will typically be n-way rather than binary connectives, thanks to associativity.

For any logical formula there exists a formula in DNF form, which is logically equivalent to it. For example, the formula $(\neg P \wedge Q) \vee R$ is already in DNF form, while the formula $\neg P \wedge (Q \vee R)$ could be transformed to an equivalent DNF form using the law of the distributivity of conjunction over alternative: $(\neg P \wedge Q) \vee (\neg P \wedge R)$.

There could be many different DNF forms of a given formula.

Converting formulas to DNF

One specific construction of the DNF form can be obtained from the truth table of a formula in terms of its component atomic formulas.

Example:

$$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$$

P	Q	$P \Rightarrow Q$	$Q \Rightarrow P$	$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	1	1	1

By selecting the rows of the truth table for which the formula is true (one in the last column), we can construct the DNF formula:

$$(\neg P \wedge \neg Q) \vee (P \wedge Q)$$

The conjunctive normal form (CNF)

A formula in the form of a conjunction of alternatives of literals is said to be in the **Conjunctive Normal Form (CNF)**. Formulas, which are alternatives of literals are called **clauses**. So a formula in the CNF form is a conjunction of clauses. (For this reason CNF could also be spelled out as Clausal Normal Form.)

An example of a formula in the CNF form: $(P \vee Q \vee \neg R) \wedge (P \vee \neg Q \vee R)$. CNF is analogous, dual to the DNF form of writing formulas. It may seem less intuitive at first, but is, in fact, much more useful in the automatic theorem proving systems.

For example, it is modular, ie. if we wanted to add a new fact (in CNF) to the already existing formula (also in CNF), then it is a syntactically trivial operation, and does not require any transformations to either of the formulas, unlike in the case of DNF.

Converting formulas to CNF

Consider again an example formula and its truth table:

$$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$$

P	Q	$P \Rightarrow Q$	$Q \Rightarrow P$	$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	1	1	1

Analogous to the algorithm for constructing the canonical DNF of the formula, we can construct the CNF by taking the rows from the table with a zero in the last column, and writing clauses eliminating these rows:

$$(\neg P \vee Q) \wedge (P \vee \neg Q)$$

Empty conjunctions and empty clauses

We can talk of single literals as of 1-ary (unary) conjunctions of clauses, or of unary clauses (alternatives of literals). Moreover, we allow empty (0-ary) clauses, and empty conjunctions of clauses.

$$\begin{array}{rcl} p_1 \wedge p_2 \wedge \ldots \wedge p_n & = & \wedge(p_1, p_2, \ldots, p_n) \\ p \wedge q & = & \wedge(p, q) \\ p & = & \wedge(p) \\ \sqcup & = & \wedge() \end{array} \quad \begin{array}{rcl} p_1 \vee p_2 \vee \ldots \vee p_n & = & \vee(p_1, p_2, \ldots, p_n) \\ p \vee q & = & \vee(p, q) \\ p & = & \vee(p) \\ \sqcup & = & \vee() \end{array}$$

While the truth value of any nonempty conjunction or clause depends on the truth values of its components, the empty formulas should have a constant logical interpretation. By a simple generalization of the definition of the logical values of these connectives we can obtain that the empty conjunction is a valid formula (tautology), while the empty clause is a false (unsatisfiable) formula.

When writing the logical formulas as sets, or lists, the empty conjunctions and clauses appear as empty sets $\{\}$, or lists: () or NIL. Additionally, in the logical notation, the symbol $\Box$ is used to denote the empty clause.

Example:

Let's consider the formula $(P \wedge Q)$ written as a set of clauses (of single literals): $\{P, Q\}$. Adding to this set the empty conjunction: $\{P, Q\} \cup \{\}$ corresponds to taking the following conjunction in the logical notation: $(P \wedge Q) \wedge T \equiv (P \wedge Q)$ (where T represents the truth). This confirms the correctness of interpreting the empty conjunction as tautology.

Analogously, we can write the clause $(P \vee Q)$ as a set of literals: $\{P, Q\}$. Adding to this set the empty clause: $\{P, Q\} \cup \{\}$ in the logical notation corresponds to: $(P \vee Q) \vee F \equiv (P \vee Q)$ (where F represents falsity). So the interpretation of the empty clause as a false formula is also correct.

Rewriting logical formulas as sets of clauses

A variable-free formula can be converted to a set of clauses, also called the **prenex** form, where all quantifiers are written in front of the formula:

(i) rename the variables bound by quantifiers to be unique,
(ii) replace all logical connectives with only conjunctions and alternatives,
(iii) move negations to inside the formulas (to predicate symbols),
(iv) extract the quantifiers outside the formula,
(v) convert the formula to CNF,
(vi) replace all existential quantifiers with Skolem functions.

The first five steps are logically equivalent transformations (as long as the right order of the extracted quantifiers is maintained in step (iv)). The (vi) step, called **skolemization**, converts all the formulas of the form:

$$\forall x_1 \forall x_2 ... \forall x_n \exists y \; \Phi(x_1, x_2, ..., x_n, y)$$

with:

$$\forall x_1 \forall x_2 ... \forall x_n \; \Phi(x_1, x_2, ..., x_n, f_y(x_1, x_2, ..., x_n))$$

where f_y is a newly introduced functional symbol called the **Skolem function**. In case there are no universal quantifiers $\forall$ this will be a **Skolem constant**.

Skolem's theorem

The last step in the algorithm for the conversion of formulas into the prenex form is not a logically equivalent transformation. That means, that for the original formula Φ and the resulting prenex formula Φ', in general $\Phi \not\equiv \Phi'$.

However, the following property, called the **Skolem theorem** holds: for a closed formula Φ, if Φ' is its prenex form, then Φ is satisfiable if and only if Φ' is satisfiable.

Therefore, while we cannot <u>in general</u> use the derived prenex form Φ' for any logical reasoning instead of Φ, we can use it <u>for proving satisfiability</u> (or unsatisfiability).

There exists an extremely useful inferencing scheme, using formulas in prenex form, often written as sets (or lists) of clauses, with clauses written as sets (or lists) of literals.

Short review

Convert to prenex form the following first order predicate calculus formulas:

1. $\forall x\ [(P(x) \Rightarrow Q(x)) \wedge (P(x) \Rightarrow R(x))]$
2. $\forall x\ [(P(x) \wedge Q(x)) \vee (R(x) \wedge S(x))]$
3. $\forall x \exists y\ [P(x) \Rightarrow Q(x,y)]$
4. $\exists x \forall y\ [P(x,y) \Rightarrow Q(A,x)]$
5. $\forall x \exists y\ [P(x,y) \Rightarrow Q(y,f(y))]$

Resolution — the ground clauses case

The **resolution** for two ground (fully instantiated) clauses: if there is a common literal occurring in these clauses with opposite signs, then the resolution constructs a new clause, called the **resolvent**, which is a union of all the literals from both clauses excluding the common literal.

For an example, for the clauses:

$$P \vee Q(A) \text{ and } \neg S \vee \neg Q(A)$$

the resolution created the resolvent: $P \vee \neg S$.

For the pair of clauses: $(P \vee Q(A)), (\neg S \vee Q(A))$ and likewise for: $(P \vee Q(A)), (\neg S \vee \neg Q(B))$ there do not exist common literals with opposite signs, so executing resolution for these pairs of clauses is not possible.

Fact: the resolvent always logically follows from the conjunction of the original clauses, so resolution is a sound inference rule.

Some interesting special cases of resolution (the $\leadsto$ symbol denotes here the possibility of executing the resolution and obtaining the result indicated):

P	and	$\neg P \vee Q$	$\leadsto$	Q	modus ponens
$P \vee Q$	and	$\neg P \vee Q$	$\leadsto$	Q	a stronger version
$P \vee Q$	and	$\neg P \vee \neg Q$	$\leadsto$	$P \vee \neg P$	tautology
$P \vee Q$	and	$\neg P \vee \neg Q$	$\leadsto$	$Q \vee \neg Q$	-"-
P	and	$\neg P$	$\leadsto$	NIL	contradiction
$\neg P \vee Q$	and	$\neg Q \vee R$	$\leadsto$	$\neg P \vee R$	transitivity of
$(P \Rightarrow Q)$		$(Q \Rightarrow R)$		$(P \Rightarrow R)$	implication

Short review

For the following set of formulas, write all possible to obtain resolvents.
If it is not possible to compute a resolution, then give a short explanation.
Compare the computed resolvents with logical consequences you can derive intuitively from the formulas given.
Pay attention to commas, to correctly identify formulas in sets.

1. $\{ P \vee Q \vee R \ , \ \neg P \vee \neg Q \vee \neg R \}$
2. $\{ P \vee Q \ , \ P \vee \neg Q \ , \ \neg P \vee Q \}$
3. $\{ P \Rightarrow (Q \vee R) \ , \ \neg Q \wedge \neg R \}$
4. $\{ P \Rightarrow Q \ , \ R \Rightarrow Q \ , \ P \vee R \}$

Substituting variables in formulas

We shall consider transformations of formulas consisting in replacing variable occurrences with other expressions (terms). Since the variables in prenex formulas are implicitly bound with universal quantifiers, replacing variables with other terms means taking specific cases of the formula.

We will call a **substitution** a set of mappings indicating terms to be substituted for specific variables. The term may not contain the variable it is to replace. An example of a substitution: $s = \{x \mapsto A, y \mapsto f(z)\}$.

Applying a substitution works by syntactically replacing all the occurrences of a given variable within a formula with its associated term. All replacements are done simultaneously, so eg. by applying the substitution $s = \{x \mapsto y, y \mapsto A\}$ to the term $f(x, y)$ the result will be the term $f(y, A)$.

Note that this way it does not matter in which order the variables are substituted, even though a substitution is a set (unordered).

A **composition** of substitutions s_1 and s_2 (written as: $s_1 s_2$) is called a substitution obtained by applying the substitution s_2 on terms from s_1, and appending to the resulting set all the pairs from s_2 with variables not in s_1.

$$\begin{aligned} \Phi s_1 s_2 &= (\Phi s_1) s_2 \\ s_1(s_2 s_3) &= (s_1 s_2) s_3 \end{aligned}$$

Unification

Unification is the procedure of finding a substitution of terms to variables in a set of formulas, to reduce it to a singleton set (or to logically equivalent formulas, see explanation below).

A **unifier** of a set of formulas is a substitution reducing it to a singleton set. A set of formulas is **unifiable** if there exists a unifier for it.

For example, the set $\{P(x), P(A)\}$ is unifiable, and its unifier is $s = \{x \mapsto A\}$.

Likewise, the set $\{P(x), P(y), P(A)\}$ is unifiable, and its unifier is $s = \{x \mapsto A, y \mapsto A\}$.

The set $\{P(A), P(B)\}$ is not unifiable, and neither is $\{P(A), Q(x)\}$.

Unification (cntd.)

While unification is a general procedure, here we will compute it only on sets of clauses. Consider the following example clause sets:

$$\begin{aligned}
\Phi &= \{P \vee Q(x), P \vee Q(A), P \vee Q(y)\} \\
\Psi &= \{P \vee Q(x), P \vee Q(A), P \vee Q(f(y))\} \\
\Omega &= \{P \vee Q(x), P \vee Q(A) \vee Q(y)\}
\end{aligned}$$

The set Φ is unifiable, its unifier is: $s = \{x \mapsto A, y \mapsto A\}$, and the unified set is the singleton set: $\Phi s = \{P \wedge Q(A)\}$).

The set Ψ is not unifiable.

The set Ω is a more complex case. By applying a purely **syntactic unification**, it is not unifiable, since after applying the substitution the formulas are not the same. However, by applying a **semantic unification**, the set is unifiable, since the formulas after applying the substitution are logically equivalent. We will allow semantic unification using associativity and commutativity of the alternative.

Most general unifier (mgu)

The **most general unifier (mgu)** of a unifiable set of formulas is the simplest (minimal) unifier for that set.

For a unifiable set of formulas there always exists its mgu, and any other unifier for this set can be obtained by composing the mgu with some additional substitution. The **unification algorithm** computes the mgu of a set of formulas.

Short review

For the following set of clauses answer if each set is unifiable.
If so, then write its unifier. Try to give both the mgu, and another unifier, which is not mgu. If the set is not unifiable, then explain why.
Pay attention to commas, to correctly identify formulas in sets.

1. $\{P(x)\ , P(f(x))\}$
2. $\{P(x,y)\ , P(y,x)\}$
3. $\{P(x,y)\ , P(y,f(x))\}$
4. $\{P(x,y)\ , P(y,f(y))\}$
5. $\{P(x,y)\ , P(y,z)\ , P(z,A)\}$

Resolution — the general case

Resolution in the general case: if for two clauses (sets of literals): $\{L_i\}$ and $\{M_i\}$ there exist respective subsets of literals: $\{l_i\}$ and $\{m_i\}$, called the **collision literals** such, that the set: $\{l_i\} \cup \{\neg m_i\}$ is unifiable and s is its mgu, then their resolvent is the set: $[\{L_i\} - \{l_i\}]s \cup [\{M_i\} - \{m_i\}]s$.

There can exist different resolvents for given clauses, by different selection of collision literals. For example, consider the following clauses:

$$P[x, f(A)] \vee P[x, f(y)] \vee Q(y) \quad \text{and} \quad \neg P[z, f(A)] \vee \neg Q(z)$$

By choosing $\{l_i\} = \{P[x, f(A)]\}$ and $\{m_i\} = \{\neg P[z, f(A)]\}$ we obtain the resolvent:

$$P[z, f(y)] \vee \neg Q(z) \vee Q(y)$$

But by choosing $\{l_i\} = \{P[x, f(A)], P[x, f(y)]\}$ and $\{m_i\} = \{\neg P[z, f(A)]\}$ we obtain:

$$Q(A) \vee \neg Q(z)$$

Short review

For the following set of clauses, write all possible to obtain resolvents.
For each resolvent, note which clauses it was derived from, and what substitution was used. If it is not possible to compute a resolution, then give a short explanation.
Pay attention to commas, to correctly identify formulas in sets.

1. $\{\neg P(x) \vee Q(x)\ ,\ P(A)\}$

2. $\{\neg P(x) \vee Q(x)\ ,\ \neg Q(x)\}$

3. $\{\neg P(x) \vee Q(x)\ ,\ P(f(x))\ ,\ \neg Q(x)\}$

Resolution as an inference rule

Resolution is a sound inference rule, since a clause obtained from a pair of clauses by resolution is their logical consequence. It is, however, not complete, ie. we cannot derive by resolution just any conclusion φ of a given formula Δ, such that $\Delta \vdash \varphi$.

For example, for $\Delta = \{P, Q\}$ we cannot derive by resolution the formula $P \vee Q$ or $P \wedge Q$, and for $\Delta = \{\forall x R(x)\}$ we cannot derive the formula $\exists x R(x)$.

However, if resolution is used in the **refutation** proof procedure, ie. by negating the thesis and deriving falsity, represented by the empty clause (denoted by $\Box$), then any theorem can be proved by resolution. So resolution is said to be **refutation complete**.

Consider the above examples. For: $\Delta = \{P, Q\}$ negating the formula $P \vee Q$ gives the clauses $\neg P$ and $\neg Q$ and each of them immediately gives the empty clause with the corresponding clause from Δ. The negation of $P \wedge Q$ is the clause $\neg P \vee \neg Q$ and the empty clause can be derived in two resolution steps. For $\Delta = \{\forall x R(x)\}$ the negation of $\exists x R(x)$ is $\neg R(y)$, which unifies with the clause $R(x)$ derived from Δ and derives the empty clause in one resolution step.

Theorem proving based on resolution

The basic reasoning scheme based on resolution, when we have a set of axioms Δ and want to derive from it the formulaφ, is the following. We make a union of the sets of clauses obtained from Δ and $\neg\varphi$, and we try to derive falsity (the empty clause) from it, generating subsequent resolvents from the selected pairs of clauses. At each step we add the newly obtained resolvent to the main set of clauses, and repeat the procedure.

The main result from the mathematical logic being used here is the following two facts. If resolution is executed on a set of clauses obtained from an unsatisfiable formula, with some systematic algorithm of generating resolvents, then we will obtain the empty clause at some point. And the other way around, if the empty clause can be generated from a set of clauses obtained from some formula, then this set of clauses, but also the original formula, are both unsatisfiable. This applies as well to the clauses created by skolemization, so is a confirmation of the correctness of the whole procedure.

Theorem proving: an example

We know that:

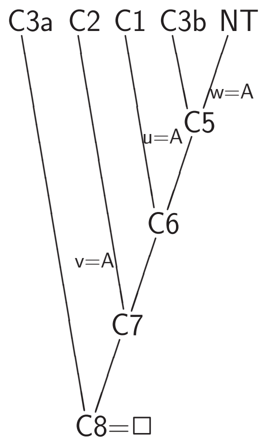
1.	Whoever can read is literate.	$(\forall x)[R(x) \Rightarrow L(x)]$
2.	Dolphins are not literate.	$(\forall x)[D(x) \Rightarrow \neg L(x)]$
3.	Some dolphins are intelligent.	$(\exists x)[D(x) \wedge I(x)]$

We need to prove the statement:

4.	Some who are intelligent cannot read.	$(\exists x)[I(x) \wedge \neg R(x)]$

After converting the statements to the prenex CNF form we obtain the clauses:

C1:	$\neg R(u) \vee L(u)$	from the first axiom
C2:	$\neg D(v) \vee \neg L(v)$	from the second axiom
C3a:	$D(A)$	from the third axiom, p.1
C3b:	$I(A)$	from the third axiom, p.2
NT:	$\neg I(w) \vee R(w)$	from the negation of the theorem



From the subsequent resolution steps we obtain:

C5:	$R(A)$	resolvent of clauses C3b and NT
C6:	$L(A)$	resolvent of clauses C5 and C1
C7:	$\neg D(A)$	resolvent of clauses C6 and C2
C8:	$\Box$	resolvent of clauses C7 and C3a

Theorem proving: an example from mathematics

Let us consider the following example from mathematics.[2] We would like to prove that the intersection of two sets is contained in either one of them. We start from writing the axioms which are necessary for this reasoning. In this case they are the definitions of the set intersection and inclusion.

$$\forall x \forall s \forall t \;\; (x \in s \;\wedge\; x \in t) \Leftrightarrow x \in s \cap t$$
$$\forall s \forall t \;\; (\forall x \; x \in s \Rightarrow x \in t) \Leftrightarrow s \subseteq t$$

Theorem to be proved:

$$\forall s \forall t \;\; s \cap t \subseteq s$$

[2] This example is borrowed from the book "Logical Foundations of Artificial Intelligence" by Genesereth and Nilsson.

After converting the formulas to prenex CNF sets of clauses:

1.	$\{x \notin s, x \notin t, x \in s \cap t\}$	from the definition of intersection
2.	$\{x \notin s \cap t, x \in s\}$	from the definition of intersection
3.	$\{x \notin s \cap t, x \in t\}$	from the definition of intersection
4.	$\{F(s,t) \in s, s \subseteq t\}$	from the definition of inclusion
5.	$\{F(s,t) \notin t, s \subseteq t\}$	from the definition of inclusion
6.	$\{A \cap B \not\subseteq A\}$	from the negation of the theorem

Let us note the Skolem functions in clauses 4 and 5, and Skolem constants in clause 6. Below is the proof sequence leading quite directly to the empty clause.

7.	$\{F(A \cap B, A) \in A \cap B\}$	from clauses 4. and 6.
8.	$\{F(A \cap B, A) \notin A\}$	from clauses 5. and 6.
9.	$\{F(A \cap B, A) \in A\}$	from clauses 2. and 7.
10.	$\{\}$	from clauses 8. and 9.

This is it. Theorem proved. But somehow, it is hard to feel the satisfaction which typically accompanies completing a real mathematical proof. Furthermore, in case one wanted to review the proof, or verify it, one has to do some nontrivial extra work, although in this particular case it is still relatively simple.

Short review

For the following axiom sets Δ and theorems φ, try proving $\Delta \vdash \varphi$ using resolution refutation.

1. $\Delta = \{\forall x(\mathsf{Likes}(x, \mathsf{Wine}) \Rightarrow \mathsf{Likes}(\mathsf{Dick}, x)), \mathsf{Likes}(\mathsf{Ed}, \mathsf{Wine})\}$
   $\varphi = \mathsf{Likes}(\mathsf{Dick}, \mathsf{Ed})$

2. $\Delta = \{\forall x(\mathsf{Likes}(x, \mathsf{Dick}) \Rightarrow \mathsf{Likes}(\mathsf{Dick}, x)), \neg\mathsf{Likes}(\mathsf{wife}(\mathsf{Ed}), \mathsf{Dick})\}$
   $\varphi = \mathsf{Likes}(\mathsf{Dick}, \mathsf{wife}(\mathsf{Ed}))$

3. $\Delta = \{\forall x(\mathsf{Likes}(x, \mathsf{Wine}) \Rightarrow \mathsf{Likes}(\mathsf{Dick}, x)), \mathsf{Likes}(\mathsf{Ed}, \mathsf{Wine})\}$
   $\varphi = (\mathsf{Likes}(\mathsf{Dick}, \mathsf{Ed}) \vee \mathsf{Likes}(\mathsf{Dick}, \mathsf{wife}(\mathsf{Ed})))$

4. $\Delta = \{\forall x(\mathsf{Likes}(x, \mathsf{Wine}) \Rightarrow \mathsf{Likes}(\mathsf{Dick}, x)), \mathsf{Likes}(\mathsf{Ed}, \mathsf{Wine})\}$
   $\varphi = (\mathsf{Likes}(\mathsf{Dick}, \mathsf{Ed}) \wedge \mathsf{Likes}(\mathsf{Dick}, \mathsf{wife}(\mathsf{Ed})))$

Knowledge engineering

The presented formalism of first order predicate logic, along with resolution as a method of theorem proving, constitute a technique for building intelligent agents capable of solving problems presented to them. The construction of such an agent, however, requires an efficient design of a representation, which can be formulated as the following process termed **knowledge engineering**:

problem identification
: Define the scope of questions which the agent would have to be able to answer, the type of facts, which she will be able to use, etc. For example, in relation to the wumpus world, we must declare whether the agent should be able to plan activities, or, for example, only create the representation of the state of the world identified by previous actions.

knowledge acquisition
: The developer of the agent software (knowledge engineer) may not understand all the nuances of the described world, and must cooperate with experts to obtain all the necessary knowledge.

definition of the representation dictionary
: The concepts and objects of the problem domain must be described with logical formulas. It is necessary to define a vocabulary of predicates and terms, ie term functions and constants. This stage may prove crucial for the ability to effectively solve problems, eg. in the wumpus world, would pits better be represented as objects, or properties of locations.

encoding general knowledge
: Encode axioms containing general knowledge about the problem domain, the rules governing this world, existing heuristics, etc.

encoding specific knowledge
: The statement of a specific problem to be solved by the agent, including the facts about all specific objects, as well as the question to answer, or, more generally, the theorem to prove.

submit queries to the reasoning device
: Run the theorem proving procedure on the knowledge base constructed.

debug the knowledge base

Unfortunately, as is also the case with normal programs, rarely designed system will immediately work properly. There may occur such problems as the lack of some key axioms, or axioms imprecisely stated, that permit proving too strong assertions.

Dealing with equality

One very special relation occurring in logical formulas is the equality (identity) of terms.

Example:
$\Delta = \{$=(wife(Ed), Meg), Owns(wife(Ed), alfa-8c)$\}$.
Does it mean that Meg owns an Alfa 8c Competizione?
Can we prove it using resolution?
Owns(Meg, alfa-8c)?

Unfortunately not. The resolution proof procedure does not treat the equality predicate in any special way, and will not take advantage of the term equality information it has. For the resolution proof in the above example to succeed, we would have to formulate an appropriate equality axiom:

$$\forall x, y, z\ [\mathsf{Owns}(x, y) \wedge \mathsf{=}(x, z) \Rightarrow \mathsf{Owns}(z, y)]$$

Using the above axiom, connecting Owns with equality, we can prove that Meg owns the Alfa, as well as any other facts and equalities of the owners. However, to likewise extend the reasoning to the equality of the objects owned, an additional axiom has to be introduced:

$$\forall x, y, z\ [\mathsf{Owns}(x, y) \wedge \mathsf{=}(y, z) \Rightarrow \mathsf{Owns}(x, z)]$$

Worse yet, for the system to properly handle all the facts of the term identities with respect to all relations, similar axioms would have to be written for all the predicate symbols. Unfortunately, in the first order predicate language it is not possible to express this in one general formula, like:

$$\forall P, y, z[P(y) \wedge \mathsf{=}(y, z) \Rightarrow P(z)]$$

An alternative solution would be to incorporate the processing of term equality in the theorem proving process. Several solutions exist: a formula reduction rule with respect to term equality, called **demodulation**, a generalized resolution rule called **paramodulation**, and an extended unification procedure which handles equalities.

Answer extraction from the proof tree

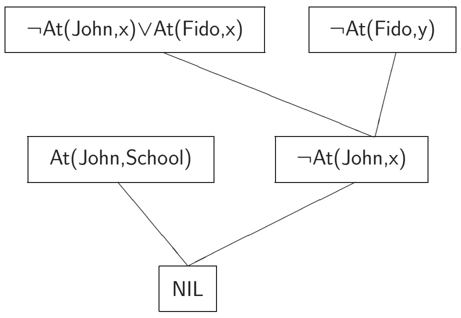
Consider a simple example, we know:

1. Where is John, there is Fido. $(\forall x)[\text{At}(John, x) \Rightarrow \text{At}(Fido, x)]$
2. John is at school. $\text{At}(John, School)$

The question we need to answer is:

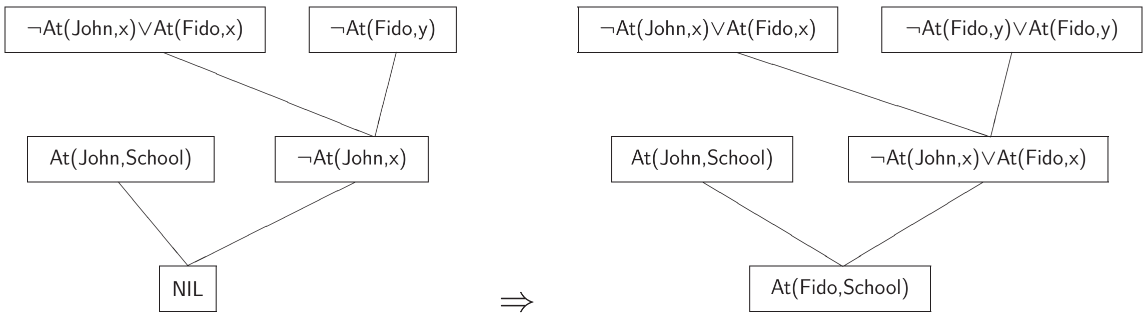
3. Where is Fido? $(\exists x)[\text{At}(Fido, x)]$

The logical version of the original question is $\neg\text{At}(Fido, x)$, and the proof is generated easily:



Unfortunately, it does not provide an answer to the original question.

Answer extraction from the proof tree (cntd.)



- The basic procedure converts the refutation proof to a direct proof of the theorem.
- If the theorem contains alternatives (which become conjunctions after negation) then the resulting formula may be complex and hard to interpret.
- If the theorem contains a universal quantifier then after negation it contains Skolem functions or constants, which are carried to the result formula, but can be converted to a universally quantified variable.

Resolution speedup strategies

In proving theorems using the resolution refutation procedure, we aim to generate the empty clause, indicating a contradiction. To be sure that the empty clause obtains, assuming that this is at all possible, we need to generate the resolvents in some systematic way, for example, using the breadth-first search. But with larger databases, this can lead to generating a large number of conclusions, of which most may not have anything to do with the theorem being proved.

It would be useful to have some **speedup strategies**, which would cut off this search and prevent generating at least some resolvents. These strategies can be **complete**, ie such that will always find a solutions (contradiction) if at all possible, or **incomplete** ie giving no such a guarantee (but typically much more effective).

Speedup strategies:

- single literal preference (by itself incomplete, but complete if used as a preference)
- set of support: allow only resolution steps using one clause from a certain set, initially equal to the negated theorem clauses (complete)
- input resolution permits only resolution steps using an original input clause (complete only in some cases, eg. for Horn clause databases)
- linear resolution (incomplete)
- repetition and subsumption reduction (complete)

Undecidability of predicate calculus

The predicate calculus seems well suited to expressing facts and reasoning in artificial intelligence systems. However, we need to be aware of some of its fundamental limitations, which constrain its practical applications.

Church's theorem (1936, of the undecidability of predicate calculus): there does not exist a **decision procedure**, which could test the validity of any predicate calculus formula. We say that the predicate calculus is **undecidable**.

This property significantly restricts what can be inferred in the predicate calculus. However, for a number of classes of formulas, there does exist a decision procedure. Furthermore, the predicate calculus has the property of **semidecidability**, which means that there exists a procedure which can determine, in a finite number of steps, that a formula is unsatisfiable, if it is so. However, for satisfiable formulas such procedure may not terminate.

Incompleteness in predicate calculus

One could think, that the undecidability of predicate calculus can be circumvented by taking advantage of its semidecidability. Attempting to derive a formula φ from an axiom set Δ, we start two simultaneous proof procedures: $\Delta \vdash \varphi$ and $\Delta \vdash \neg\varphi$. By semidecidability, we could expect that at least one of these proofs should terminate. Unfortunately, this is not so.

Gödel's theorem (1931, of incompleteness): in predicate calculus one can formulate **incomplete** theories: theories with (closed) formulas, which cannot be proved true or false. What's more, such theories are quite simple and common, eg. the theory of natural numbers is one such theory.

A theory $\mathcal{T}$ is called **decidable** if there exists an algorithm that for any closed formula φ can test whether $\varphi \in \mathcal{T}$, or $\varphi \notin \mathcal{T}$. Incomplete theories are obviously undecidable.

The effect of the Gödel's theorem is, that if, after some number of steps of a proof $\Delta \vdash \varphi$ (and, perhaps a simultaneous proof $\Delta \vdash \neg\varphi$), there is still no derivation, then we still cannot be certain whether the proof will eventually terminate (or at least one of them will), or that we have an incomplete theory.

Representing changes

The predicate calculus works well as a representation language for static domains, ie. such where nothing ever changes, and whatever is true, stays so forever. Unfortunately, the real world is not like this.

For example, if the formula: $At(John, School)$ correctly describes the current state of the morning of some weekday, then, unfortunately, we must accept that John will go home eventually. If the axioms correctly describe the effects of agents' actions, then the system might be able to derive a new fact: $At(John, Home)$. Unfortunately, the fact database will then contain a contradiction, which for a logical system is a disaster. A proof system containing a false formula among its axioms can prove any theorem!

Exercise: assume the axiom set Δ contains, among others, two clauses: P and $\neg P$. Give a proof of an arbitrary formula Q. Hint: prove first that $P \vee \neg P \vee Q$ is a tautology (sentence always true) for any P and Q. This can be accomplished constructing a proof: $\models (P \vee \neg P \vee Q)$, this is, proving the thesis with an empty set of axioms. Next add such a tautology to the Δ set and proceed to prove Q.

Temporal logics

To solve the problem of representation of the changes, a number of special logic theories, called **temporal logics** were created. Ordinary facts expressed in these logics occur at specific points in time. However, the time, its properties, and special inference rules concerning its passage, are built into the theory (instead of being represented explicitly, along with other properties of the world).

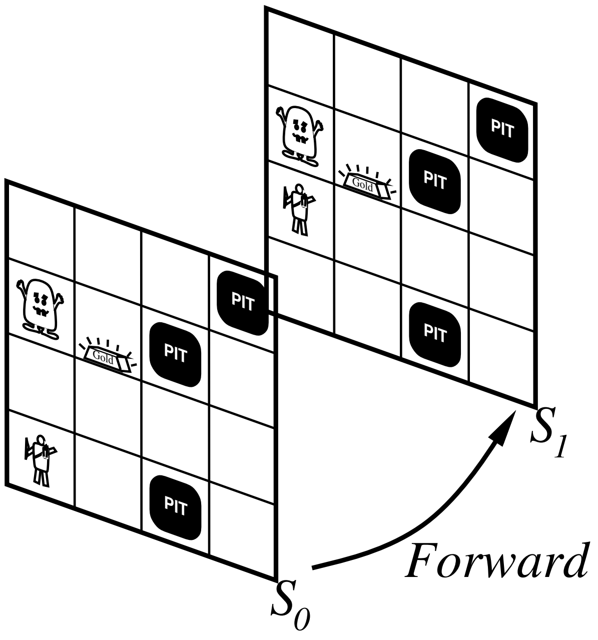
One of the main issues, which these theories treat differently, is the representation of the time itself. Time can be discrete or continuous, may be given in the form of points or intervals, may be bounded or unbounded, etc. Moreover, time can be a linear concept or can have branches. Usually it should be structured, although there exist circular representations of time.

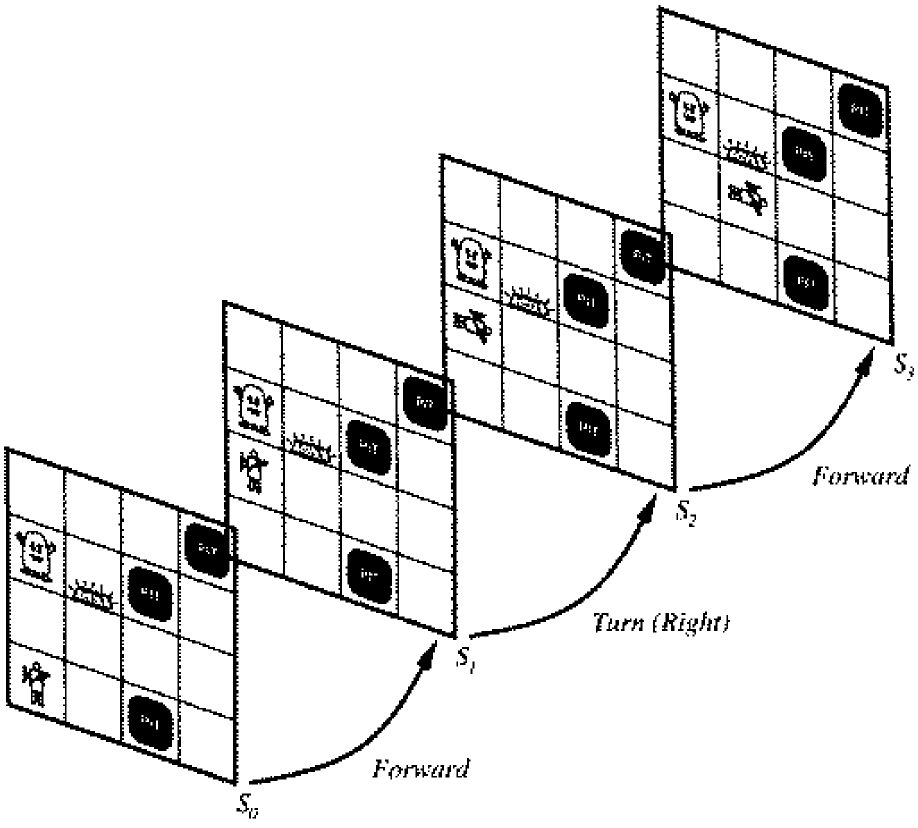
For each of these temporal logics, to be able to effectively reason about formulas created, which represent the phenomena that an intelligent agent faces, there must exist a proof procedure. The construction of such a procedure may be based on projections of the given theory to first-order predicate logic.

The situation calculus

An alternative to temporal logics is a direct recording of time moments in the representation language. An example of such an approach is the **situation calculus**:

$At(Agent, [1,1], S_0) \wedge At(Agent, [1,2], S_1) \wedge S_1 = Result(Forward, S_0)$





The situation calculus (cntd.)

The situation calculus uses the concepts of: **situations**, **actions**, and **fluents**:

situations: a situation is the initial state s_0, and for any situation s and action a a situation is also $Result(a, s)$; situations correspond to sequences of actions, and are thus different from states, ie. an agent may be in some state through different situations,

fluents: functions and relations which can vary from one situation to the next are called fluents; by convention their last argument is the situation argument,

possibility axioms: describe preconditions of actions, eg. for action $Shoot$: $Have(Agent, Arrow, s) \Rightarrow Poss(Shoot, s)$

successor-state axioms: describe for each fluent what happens depending on the action taken, eg. for action $Grab$ the axiom should assert, that after properly executing the action the agent will end up holding whatever she grabbed; but we must also remember about situations when the fluent was unaffected by some action:

$$
\begin{aligned}
&Poss(a, s) \Rightarrow \\
&\quad (Holding(Agent, g, Result(a, s)) \Leftrightarrow \\
&\qquad a = Grab(g) \vee (Holding(Agent, g, s) \wedge a \neq Release(g))).
\end{aligned}
$$

unique action axioms: because of the presence of the action inequality clauses on the successor-state axioms, we must enable to agent to effectively derive such facts, by adding the unique action axioms; for each pair of action symbols A_i and A_j we must state the (seemingly obvious) axiom $A_i \neq A_j$; also, for actions with parameters we must also state:

$$A_i(x_1, ..., x_n) = A_j(y_1, ..., y_n) \Leftrightarrow x_1 = y_1 \wedge ... \wedge x_n = y_n$$

Example: monkey and bananas — the axioms

- general knowledge of the world and operators (partial and simplified):

A1: $\forall p \forall p_1 \forall s$ [At(box, p, s) $\Rightarrow$ At(box, p, goto(p_1, s))]
A2: $\forall p \forall p_1 \forall s$ [At(bananas, p, s) $\Rightarrow$ At(bananas, p, goto(p_1, s))]
A3: $\forall p \forall s$ [At(monkey, p, goto(p, s))]
A4: $\forall p \forall p_1 \forall s$ [At(box, p, s) $\wedge$ At(monkey, p, s) $\Rightarrow$ At(box, p_1, move(box, p, p_1, s))]
A5: $\forall p \forall p_1 \forall p_2 \forall s$ [At(bananas, p, s) $\Rightarrow$ At(bananas, p, move(box, p_1, p_2, s))]
A6: $\forall p \forall p_1 \forall s$ [At(monkey, p, s) $\Rightarrow$ At(monkey, p_1, move(box, p, p_1, s))]
A7: $\forall s$ [Under(box, bananas, s) $\Rightarrow$ Under(box, bananas, climb(box, s))]
A8: $\forall p \forall s$ [At(box, p, s) $\wedge$ At(monkey, p, s) $\Rightarrow$ On(monkey, box, climb(box, s))]
A9: $\forall s$ [Under(box, bananas, s) $\wedge$ On(monkey, box, s) $\Rightarrow$ Havebananas(grab(bananas, s))]
A10: $\forall p \forall s$ [At(box, p, s) $\wedge$ At(bananas, p, s) $\Rightarrow$ Under(box, bananas, s)]

- specific case data:

A11: [At(monkey, P_1, S_0) $\wedge$ At(box, P_2, S_0) $\wedge$ At(bananas, P_3, S_0)]

- theorem to prove:

$\exists s$(Havebananas(s))

[2]The solution to the monkey and bananas problem presented here is based on an example in book „Artificial Intelligence" by Philip C. Jackson Jr.

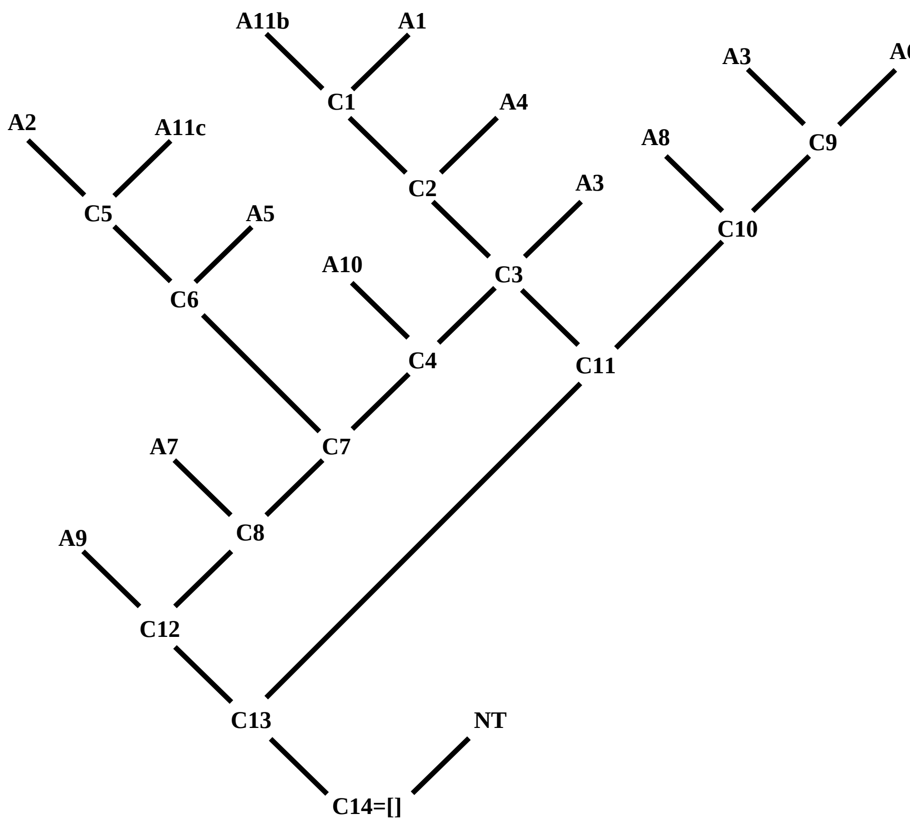
Example: monkey and bananas — the clauses

A1: {¬At(box, p, s_1), At(box, p, goto(p_1, s_1))}
A2: {¬At(bananas, q, s_2), At(bananas, q, goto(q_1, s_2))}
A3: {At(monkey, r, goto(r, s_3))}
A4: {¬At(box, u, s_4), ¬At(monkey, u, s_4), At(box, u_1, move(box, u, u_1, s_4))}
A5: {¬At(bananas, t, s_5), At(bananas, t, move(box, t_2, t_3, s_5))}
A6: {¬At(monkey, v_1, s_6), At(monkey, v_2, move(box, v_1, v_2, s_6))}
A7: {¬Under(box, bananas, s_7), Under(box, bananas, climb(box, s_7))}
A8: {¬At(monkey, w, s_8), ¬At(box, w, s_8), On(monkey, box, climb(box, s_8))}
A9: {¬Under(box, bananas, s_9), ¬On(monkey, bananas, s_9),
Havebananas(grab(bananas, s_9))}
A10: {¬At(box, p, s_{10}), ¬At(bananas, p, s_{10}), Under(box, bananas, s_{10})}
A11a: {At(monkey, P_1, S_0)}
A11b: {At(box, P_2, S_0)}
A11c: {At(bananas, P_3, S_0)}
NT: {¬Havebananas(z)}

Example: monkey and bananas — the proof

C1(A1,A11b)	$\{\text{At}(\text{box}, P_2, \text{goto}(p_1, S_0))\}$
C2(C1,A4)	$\{\neg\text{At}(\text{bananas}, P_2, \text{goto}(p_1, S_0)),$ $\text{At}(\text{box}, u_1, \text{move}(\text{box}, P_2, u_1, \text{goto}(p_1, S_0)))\}$
C3(C2,A3)	$\{\text{At}(\text{box}, u_1, \text{move}(\text{box}, P_2, u_1, \text{goto}(P_2, S_0)))\}$
C4(C3,A10)	$\{\neg\text{At}(\text{bananas}, u_1, \text{move}(\text{box}, P_2, u_1, \text{goto}(P_2, S_0))),$ $\text{Under}(\text{box}, \text{bananas}, \text{move}(\text{box}, P_2, u_1, \text{goto}(P_2, S_0)))\}$
C5(A2,A11c)	$\{\text{At}(\text{bananas}, P_3, \text{goto}(q_1, S_0))\}$
C6(C5,A5)	$\{\text{At}(\text{bananas}, P_3, \text{move}(\text{box}, t_2, t_3, \text{goto}(q_1, S_0)))\}$
C7(C6,C4)	$\{\text{Under}(\text{box}, \text{bananas}, \text{move}(\text{box}, P_2, P_3, \text{goto}(P_2, S_0)))\}$
C8(C7,A7)	$\{\text{Under}(\text{box}, \text{bananas}, \text{climb}(\text{box}, \text{move}(\text{box}, P_2, P_3, \text{goto}(P_2, S_0))))\}$
C9(A3,A6)	$\{\text{At}(\text{monkey}, v_2, \text{move}(\text{box}, r, v_2, \text{goto}(r, r_1)))\}$
C10(C9,A8)	$\{\text{At}(\text{box}, v_2, \text{move}(\text{box}, r, v_2, \text{goto}(r, r_1))),$ $\text{On}(\text{monkey}, \text{box}, \text{climb}(\text{box}, \text{move}(\text{box}, r, r_2, \text{goto}(r, r_1))))\}$
C11(C10,C3)	$\{\text{On}(\text{monkey}, \text{box}, \text{climb}(\text{box}, \text{move}(\text{box}, P_2, u_1, \text{goto}(P_2, S_0))))\}$
C12(C8,A9)	$\{\neg\text{On}(\text{monkey}, \text{box}, \text{climb}(\text{box}, \text{move}(\text{box}, P_2, P_3, \text{goto}(P_2, S_0)))),$ $\text{Havebananas}(\text{grab}(\text{bananas},$ $\text{climb}(\text{box}, \text{move}(\text{box}, P_2, P_3, \text{goto}(P_2, S_0)))))\}$
C13(C11,C12)	$\{\text{Havebananas}(\text{grab}(\text{bananas},$ $\text{climb}(\text{box}, \text{move}(\text{box}, P_2, P_3, \text{goto}(P_2, S_0)))))\}$
C14(C13,NT)	$\{\}$

Example: monkey and bananas — the resolution tree



The frame problem

As we could see in the wumpus and the monkey and bananas examples, a correct logical description of a problem requires explicitly stating the axioms for the effects of actions on the environment, as well as other effect (like rain). It is also necessary to write the axioms to conclude the lack of change:

$$\forall a, x, s \; Holding(x, s) \land (a \neq Release) \Rightarrow Holding(x, Result(a, s))$$
$$\forall a, x, s \; \neg Holding(x, s) \land (a \neq Grab) \Rightarrow \neg Holding(x, Result(a, s))$$

Unfortunately, in a world more complex than the wumpus world, there will be many fluents, and the description must represent their changes, as well as invariants, both as direct and indirect consequences of the actions.

These axioms, called the **frame axioms**, are hard to state in a general way, and they significantly complicate the representation.

Of course, during the course of work, the agent must state and answer many questions, and prove theorems. The multiplicity of axioms causes a rapid expansion of her database, which slows down further reasoning, and can result in a total paralysis.

Short review

1. Write a situation calculus based representation for the wumpus world, as described at the beginning of this document.

Problems with the lack of information

The logic-based methods presented so far assumed that all information necessary to carry out logical reasoning is available to the agent. Unfortunately, this is not a realistic assumption.

One problem is that of incomplete information. Agent may not have full information about the problem, allowing him to draw categorical conclusions. She may, however, have partial information, such as:

- „typical" facts,
- „possible" facts,
- „probable" facts,
- exceptions to the generally valid facts.

Having such information is often crucial for making the right decisions. Unfortunately, the classical predicate logic cannot make any use of them.

Another problem is the uncertainty of information. An agent may have data from a variety of not fully reliable sources. In the absence of certain information, those unreliable data should be used. She should reason using the best available data, and estimate the reliability of any conclusions obtained this way. Again, classical logic does not provide such tools.

Common sense reasoning

Consider what information a human knows for sure, making decisions in everyday life. Getting up in the morning, her intention is to go to work. But what if there is a large-scale failure of public transportation? She should, in fact, get up much earlier, and first check whether the buses are running. The day before, she bought products to make breakfast. But can she be sure that her breakfast salad is still in the refrigerator, or if it did not spoil, or perhaps if someone has not sneaked to steal it, etc.

Conclusion: a logically reasoning agent needs 100% certain information to conduct her actions, and sooner or later she will be paralyzed by the perfect correctness of her inference system. In the real world she will never be able to undertake any action, until she has full information about the surrounding world.

However, people perform quite well in a world full of uncertain and incomplete information, defaults facts, and exceptions. How do they do it? We must conclude that, in their reasoning, the humans use a slightly different logic than the rigorous mathematical logic. One could generally call this hypothetical inferencing mechanism a logic of **common sense reasoning**.

Nonmonotonic logics

Part of the blame for the problems of inference using classical logic bears its property known as **monotonicity**. In classical logic, the more we know, the more we can deduce using inferencing.

Humans use a different model of inference, much more flexible, utilizing typical facts, default facts, possible facts, and even lack of information. This kind of reasoning seems not to have the monotonicity property.

For example, lacking good information about a situation a human is likely to make assumptions and derive some conclusions. After having acquired more complete information, she might not be able to conduct the same reasoning and work out the same solutions.[3]

Hence, different models of inference, designed to overcome these problems, and following a more flexible reasoning model similar to that of humans, are collectively called **nonmonotonic logics**.

[3]The solution produced earlier, in the absence of information, turns out to be wrong now, but perhaps it was better than the lack of any action. But not necessarily.

Nonmonotonic logics — example

Minsky's challenge: to design a system, which would permit to correctly describe a well-known fact, that the birds can fly.

$$\forall x[\mathsf{bird}(x) \rightarrow \mathsf{canfly}(x)]$$

In order to accommodate the exceptions, eg. ostriches, the preceding formula must be modified per case.

$$\forall x[\mathsf{bird}(x) \wedge \neg\mathsf{ostrich}(x) \rightarrow \mathsf{canfly}(x)]$$

But there are more exceptions: birds bathed in spilled crude oil, wingless birds, sick birds, dead birds, painted birds, abstract birds, ...

An idea: we introduce a modal operator **M**:

$$\forall x[\mathsf{bird}(x) \wedge \mathbf{M}\ \mathsf{canfly}(x) \rightarrow \mathsf{canfly}(x)]$$

Now the exceptions can be introduced modularly:

$$\forall x[\mathsf{ostrich}(x) \rightarrow \neg\mathsf{canfly}(x)]$$

For the following set of facts:

$$\Delta = \{\text{bird}(\mathit{Tweety}), \text{bird}(\mathit{Sam}), \text{ostrich}(\mathit{Sam})\}$$

we can deduce: $\neg\text{canfly}(\mathit{Sam})$
so it should not be possible to derive:

$$\mathbf{M}\ \text{canfly}(\mathit{Sam}) \qquad \text{nor} \qquad \text{canfly}(\mathit{Sam})$$

However, using the normal proof procedure we cannot prove the *Tweety*'s ability to fly:

$$\mathbf{M}\ \text{canfly}(\mathit{Tweety}), \text{canfly}(\mathit{Tweety})$$

In order to do this, a proof procedure is needed, capable of effective (and automatic) proving theorems in predicate logic extended with the modal operator **M**, consistent with the following inference rule:

$$\frac{\text{Not}(\vdash \neg p)}{\mathbf{M}\ p}$$

Nonmonotonic logics — what proof procedure?

Leaving aside the restrictions resulting from the reference to the proof procedure in the above definition, such a procedure may be neither effective computationally, nor decidable nor even semidecidable, as are the proof procedures for the predicate logic.

The premise of the above inference rule contains the statement, that some formula is impossible to prove. To start, this may not be possible to determine at all. And to find a positive confirmation of this fact, it will certainly be necessary to carry out global inferencing over the entire database. For how else we could say that something can not be proved.

In contrast, proofs in first-order predicate calculus are local in nature. If, for example, we are lucky to choose the appropriate premises, we can obtain the proof in several steps, even if the data base contains thousands of facts.

Problems with logic-based methods

The approach to knowledge representation based on first-order logic at one time created much excitement and hope for building powerful and universal systems of artificial intelligence. There are, however, important considerations which significantly limit the practical applications of this approach:

- **combinatorial explosion** of the proof procedure; while there exist speedup strategies, they do not help much; at the same time it is hard to incorporate in the proof procedure any heuristic information available
- **undecidability** and the Gödel's **incompleteness** of the predicate calculus
- reasoning about **changes** — situation calculus, temporal logics
  - reasoning about change exhibits the **frame problem** — besides determining what has changed, is is essential to keep track of what has not
- reasoning with **incomplete** and **uncertain information**, truly challenging for the formal approaches, but seems crucial for the human reasoning
  - taking into account uncertain information leads to **nonmonotonic** reasoning, a property of the human reasoning, while the traditional (mathematical) logic is strictly monotonic

Applications of the logic-based methods

The above problems with logic-based methods significantly burden their application as a platform for implementing intelligent agents. Nevertheless, the first order predicate language itself is commonly used in artificial intelligence for representing facts.

Still, in some specific applications it is possible to use this methodology, and the above problems are not critical. Some of these applications are:

- computer program synthesis and verification, software engineering
- design and verification of computing hardware, including the VLSI design
- theorem proving in mathematics; which help seek proofs for any postulated theorems, for which efforts failed to find a proof in the traditional way