

KNOWLEDGE REPRESENTATION

KNOWLEDGE REPRESENTATION:-

For the purpose of solving complex problems encountered in AI, we need both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions to new problems. A variety of ways of representing knowledge (facts) have been exploited in AI programs. In all variety of knowledge representations, we deal with two kinds of entities.

A. Facts: Truths in some relevant world. These are the things we want to represent.

B. Representations of facts in some chosen formalism. These are things we will actually be able to manipulate.

One way to think of structuring these entities is at two levels: (a) the knowledge level, at which facts are described, and (b) the symbol level, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

The facts and representations are linked with two-way mappings. This link is called representation mappings. The forward representation mapping maps from facts to representations. The backward representation mapping goes the other way, from representations to facts.

One common representation is natural language (particularly English) sentences. Regardless of the representation for facts we use in a program, we may also need to be concerned with an English representation of those facts in order to facilitate getting information into and out of the system. We need mapping functions from English sentences to the representation we actually use and from it back to sentences.

Representations and Mappings

- In order to solve complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions.
- Knowledge and Representation are two distinct entities. They play central but distinguishable roles in the intelligent system.
- Knowledge is a description of the world. It determines a system's competence by what it knows.
- Moreover, Representation is the way knowledge is encoded. It defines a system's performance in doing something.
- Different types of knowledge require different kinds of representation.

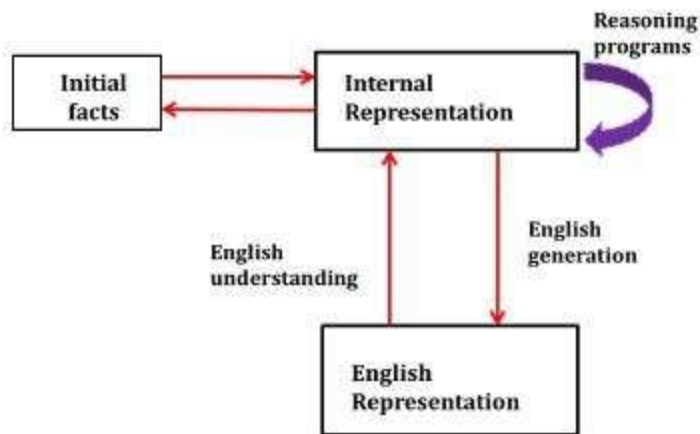


Fig: Mapping between Facts and Representations

The Knowledge Representation models/mechanisms are often based on:

- Logic
- Rules
- Frames
- Semantic Net

Knowledge is categorized into two major types:

1. Tacit corresponds to “informal” or “implicit“
 - Exists within a human being;
 - It is embodied.
 - Difficult to articulate formally.
 - Difficult to communicate or share.
 - Moreover, Hard to steal or copy.
 - Drawn from experience, action, subjective insight
2. Explicit formal type of knowledge, Explicit
 - Explicit knowledge
 - Exists outside a human being;
 - It is embedded.
 - Can be articulated formally.
 - Also, Can be shared, copied, processed and stored.
 - So, Easy to steal or copy
 - Drawn from the artifact of some type as a principle, procedure, process, concepts.

A variety of ways of representing knowledge have been exploited in AI programs.

There are two different kinds of entities, we are dealing with.

1. Facts: Truth in some relevant world. Things we want to represent.
2. Also, Representation of facts in some chosen formalism. Things we will actually be able to manipulate.

These entities structured at two levels:

1. The knowledge level, at which facts described.
2. Moreover, The symbol level, at which representation of objects defined in terms of symbols that can manipulate by programs

Framework of Knowledge Representation

- The computer requires a well-defined problem description to process and provide a well-defined acceptable solution.

- Moreover, To collect fragments of knowledge we need first to formulate a description in our spoken language and then represent it in formal language so that computer can understand.
- Also, The computer can then use an algorithm to compute an answer.

So, This process illustrated as,

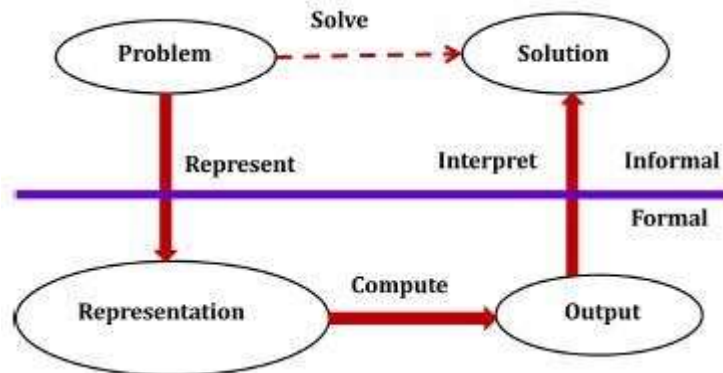


Fig: Knowledge Representation Framework

The steps are:

- The informal formalism of the problem takes place first.
- It then represented formally and the computer produces an output.
- This output can then represented in an informally described solution that user understands or checks for consistency.

The Problem solving requires,

- Formal knowledge representation, and
- Moreover, Conversion of informal knowledge to a formal knowledge that is the conversion of implicit knowledge to explicit knowledge.

Mapping between Facts and Representation

- Knowledge is a collection of facts from some domain.
- Also, We need a representation of “facts“ that can manipulate by a program.
- Moreover, Normal English is insufficient, too hard currently for a computer program to draw inferences in natural languages.
- Thus some symbolic representation is necessary.

A good knowledge representation enables fast and accurate access to knowledge and understanding of the content.

A knowledge representation system should have following properties.

1. **Representational Adequacy**
 - The ability to represent all kinds of knowledge that are needed in that domain.
2. **Inferential Adequacy**
 - Also, The ability to manipulate the representational structures to derive new structures corresponding to new knowledge inferred from old.
3. **Inferential Efficiency**
 - The ability to incorporate additional information into the knowledge structure that can be used to focus the attention of the inference mechanisms in the most promising direction.
4. **Acquisitional Efficiency**
 - Moreover, The ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

Knowledge Representation Schemes

Relational Knowledge

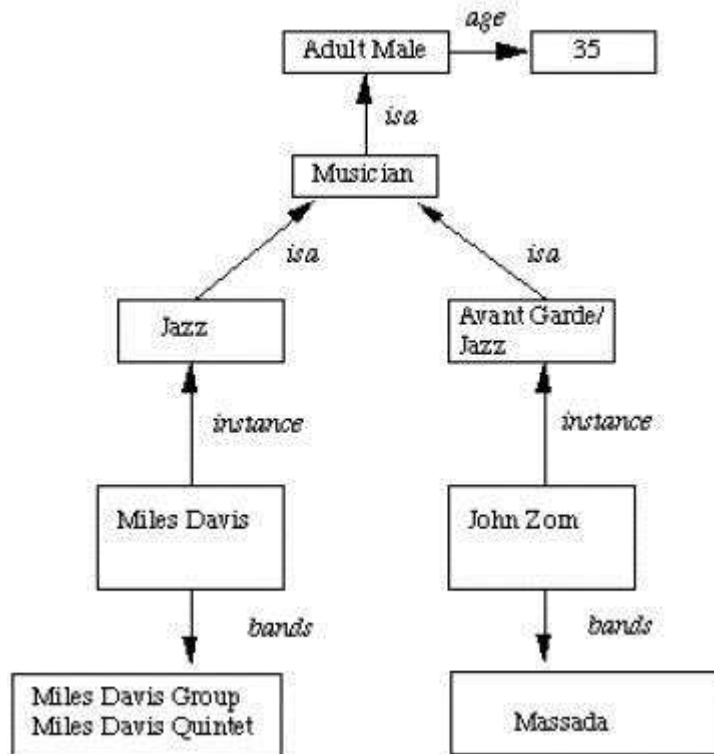
- The simplest way to represent declarative facts is a set of relations of the same sort used in the database system.
- Provides a framework to compare two objects based on equivalent attributes. o Any instance in which two different objects are compared is a relational type of knowledge.
- The table below shows a simple way to store facts.
 - Also, The facts about a set of objects are put systematically in columns.
 - This representation provides little opportunity for inference.

Player	Height	Weight	Bats - Throws
Aaron	6-0	180	Right - Right
Mays	5-10	170	Right - Right
Ruth	6-2	215	Left - Left
Williams	6-3	205	Left - Right

- Given the facts, it is not possible to answer a simple question such as: “Who is the heaviest player?”
- Also, But if a procedure for finding the heaviest player is provided, then these facts will enable that procedure to compute an answer.
- Moreover, We can ask things like who “bats – left” and “throws – right”.

Inheritable Knowledge

- Here the knowledge elements inherit attributes from their parents.
- The knowledge embodied in the design hierarchies found in the functional, physical and process domains.
- Within the hierarchy, elements inherit attributes from their parents, but in many cases, not all attributes of the parent elements prescribed to the child elements.
- Also, The inheritance is a powerful form of inference, but not adequate.
- Moreover, The basic KR (Knowledge Representation) needs to augment with inference mechanism.
- Property inheritance: The objects or elements of specific classes inherit attributes and values from more general classes.
- So, The classes organized in a generalized hierarchy.



- Boxed nodes — objects and values of attributes of objects.
- Arrows — the point from object to its value.
- This structure is known as a slot and filler structure, semantic network or a collection of frames.

The steps to retrieve a value for an attribute of an instance object:

1. Find the object in the knowledge base
2. If there is a value for the attribute report it
3. Otherwise look for a value of an instance, if none fail
4. Also, Go to that node and find a value for the attribute and then report it
5. Otherwise, search through using is until a value is found for the attribute.

Inferential Knowledge

- This knowledge generates new information from the given information.
- This new information does not require further data gathering from source but does require analysis of the given information to generate new knowledge.
- Example: given a set of relations and values, one may infer other values or relations. A predicate logic (a mathematical deduction) used to infer from a set of attributes. Moreover, Inference through predicate logic uses a set of logical operations to relate individual data.
- Represent knowledge as formal logic: All dogs have tails $\forall x: dog(x) \rightarrow hastail(x)$
- Advantages:
 - A set of strict rules.
 - Can use to derive more facts.
 - Also, Truths of new statements can be verified.
 - Guaranteed correctness.
- So, Many inference procedures available to implement standard rules of logic popular in AI systems. e.g Automated theorem proving.

Procedural Knowledge

- A representation in which the control information, to use the knowledge, embedded in the knowledge itself. For example, computer programs, directions, and recipes; these indicate specific use or implementation;
- Moreover, Knowledge encoded in some procedures, small programs that know how to do specific things, how to proceed.
- Advantages:
 - Heuristic or domain-specific knowledge can represent.
 - Moreover, Extended logical inferences, such as default reasoning facilitated.
 - Also, Side effects of actions may model. Some rules may become false in time. Keeping track of this in large systems may be tricky.
- Disadvantages:
 - Completeness — not all cases may represent.
 - Consistency — not all deductions may be correct. e.g If we know that Fred is a bird we might deduce that Fred can fly. Later we might discover that Fred is an emu.
 - Modularity sacrificed. Changes in knowledge base might have far-reaching effects.
 - Cumbersome control information.

USING PREDICATE LOGIC

Representation of Simple Facts in Logic

Propositional logic is useful because it is simple to deal with and a decision procedure for it exists.

Also, In order to draw conclusions, facts are represented in a more convenient way as,

1. Marcus is a man.
 - `man(Marcus)`
2. Plato is a man.
 - `man(Plato)`
3. All men are mortal.
 - `mortal(men)`

But propositional logic fails to capture the relationship between an individual being a man and that individual being a mortal.

- How can these sentences be represented so that we can infer the third sentence from the first two?
- Also, Propositional logic commits only to the existence of facts that may or may not be the case in the world being represented.
- Moreover, It has a simple syntax and simple semantics. It suffices to illustrate the process of inference.
- Propositional logic quickly becomes impractical, even for very small worlds.

Predicate logic

First-order Predicate logic (FOPL) models the world in terms of

- Objects, which are things with individual identities
- Properties of objects that distinguish them from other objects
- Relations that hold among sets of objects

- Functions, which are a subset of relations where there is only one “value” for any given “input”

First-order Predicate logic (FOPL) provides

- Constants: a, b, dog33. Name a specific object.
- Variables: X, Y. Refer to an object without naming it.
- Functions: Mapping from objects to objects.
- Terms: Refer to objects
- Atomic Sentences: in(dad-of(X), food6) Can be true or false, Correspond to propositional symbols P, Q.

A well-formed formula (*wff*) is a sentence containing no “free” variables. So, That is, all variables are “bound” by universal or existential quantifiers.

$(\forall x)P(x, y)$ has x bound as a universally quantified variable, but y is free.

Quantifiers

Universal quantification

- $(\forall x)P(x)$ means that P holds for all values of x in the domain associated with that variable
- E.g., $(\forall x) \text{dolphin}(x) \rightarrow \text{mammal}(x)$

Existential quantification

- $(\exists x)P(x)$ means that P holds for some value of x in the domain associated with that variable
- E.g., $(\exists x) \text{mammal}(x) \wedge \text{lays-eggs}(x)$

Also, Consider the following example that shows the use of predicate logic as a way of representing knowledge.

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. Also, All Pompeians were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

The facts described by these sentences can be represented as a set of well-formed formulas (*wffs*) as follows:

1. Marcus was a man.
 - $\text{man}(\text{Marcus})$
2. Marcus was a Pompeian.
 - $\text{Pompeian}(\text{Marcus})$
3. All Pompeians were Romans.
 - $\forall x: \text{Pompeian}(x) \rightarrow \text{Roman}(x)$
4. Caesar was a ruler.
 - $\text{ruler}(\text{Caesar})$
5. All Pompeians were either loyal to Caesar or hated him.
 - inclusive-or
 - $\forall x: \text{Roman}(x) \rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar})$
 - exclusive-or
 - $\forall x: \text{Roman}(x) \rightarrow (\text{loyalto}(x, \text{Caesar}) \wedge \neg \text{hate}(x, \text{Caesar})) \vee$
 - $(\neg \text{loyalto}(x, \text{Caesar}) \wedge \text{hate}(x, \text{Caesar}))$

6. Everyone is loyal to someone.
 - $\forall x: \exists y: \text{loyalto}(x, y)$
7. People only try to assassinate rulers they are not loyal to.
 - $\forall x: \forall y: \text{person}(x) \wedge \text{ruler}(y) \wedge \text{tryassassinate}(x, y)$
 - $\rightarrow \neg \text{loyalto}(x, y)$
8. Marcus tried to assassinate Caesar.
 - $\text{tryassassinate}(\text{Marcus}, \text{Caesar})$

Now suppose if we want to use these statements to answer the question: **Was Marcus loyal to Caesar?**

Also, Now let's try to produce a formal proof, reasoning backward from the desired goal: $\neg \text{loyalto}(\text{Marcus}, \text{Caesar})$

In order to prove the goal, we need to use the rules of inference to transform it into another goal (or possibly a set of goals) that can, in turn, transformed, and so on, until there are no unsatisfied goals remaining.

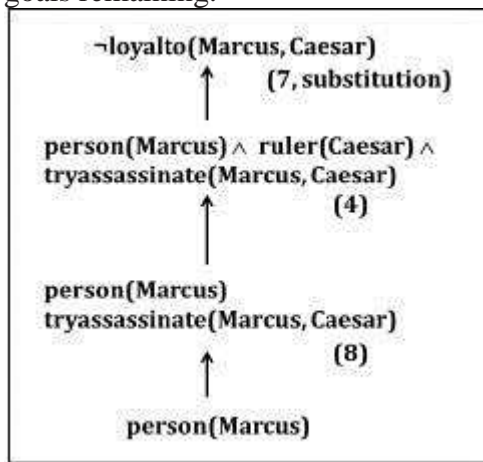


Figure: An attempt to prove $\neg \text{loyalto}(\text{Marcus}, \text{Caesar})$.

- The problem is that, although we know that Marcus was a man, we do not have any way to conclude from that that Marcus was a person. Also, We need to add the representation of another fact to our system, namely: $\forall \text{man}(x) \rightarrow \text{person}(x)$
- Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar.
- Moreover, From this simple example, we see that three important issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones:
 1. Many English sentences are ambiguous (for example, 5, 6, and 7 above). Choosing the correct interpretation may be difficult.
 2. Also, There is often a choice of how to represent the knowledge. Simple representations are desirable, but they may exclude certain kinds of reasoning.
 3. Similarly, Even in very simple situations, a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand. In order to be able to use a set of statements effectively. Moreover, It is usually necessary to have access to another set of statements that represent facts that people consider too obvious to mention.

Representing Instance and ISA Relationships

- Specific attributes **instance** and **isa** play an important role particularly in a useful form of reasoning called property inheritance.
- The predicates instance and isa explicitly captured the relationships they used to express, namely class membership and class inclusion.
- 4.2 shows the first five sentences of the last section represented in logic in three different ways.
- The first part of the figure contains the representations we have already discussed. In these representations, class membership represented with unary predicates (such as Roman), each of which corresponds to a class.
- Asserting that $P(x)$ is true is equivalent to asserting that x is an instance (or element) of P .
- The second part of the figure contains representations that use the **instance** predicate explicitly.

<ol style="list-style-type: none"> 1. Man(Marcus). 2. Pompeian(Marcus). 3. $\forall x: \text{Pompeian}(x) \rightarrow \text{Roman}(x).$ 4. ruler(Caesar). 5. $\forall x: \text{Roman}(x) \rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar}).$
<ol style="list-style-type: none"> 1. instance(Marcus, man). 2. instance(Marcus, Pompeian). 3. $\forall x: \text{instance}(x, \text{Pompeian}) \rightarrow \text{instance}(x, \text{Roman}).$ 4. instance(Caesar, ruler). 5. $\forall x: \text{instance}(x, \text{Roman}). \rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar}).$
<ol style="list-style-type: none"> 1. instance(Marcus, man). 2. instance(Marcus, Pompeian). 3. isa(Pompeian, Roman) 4. instance(Caesar, ruler). 5. $\forall x: \text{instance}(x, \text{Roman}). \rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar}).$ 6. $\forall x: \forall y: \forall z: \text{instance}(x, y) \wedge \text{isa}(y, z) \rightarrow \text{instance}(x, z).$

Figure: Three ways of representing class membership: ISA Relationships

- The predicate **instance** is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs.
- But these representations do not use an explicit **isa** predicate.
- Instead, subclass relationships, such as that between Pompeians and Romans, described as shown in sentence 3.
- The implication rule states that if an object is an instance of the subclass Pompeian then it is an instance of the superclass Roman.
- Note that this rule is equivalent to the standard set-theoretic definition of the subclass-superclass relationship.
- The third part contains representations that use both the **instance** and **isa** predicates explicitly.
- The use of the **isa** predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided.

Computable Functions and Predicates

- To express simple facts, such as the following greater-than and less-than relationships:
 $gt(1,0)$ $lt(0,1)$ $gt(2,1)$ $lt(1,2)$ $gt(3,2)$ $lt(2,3)$
- It is often also useful to have computable functions as well as computable predicates.
Thus we might want to be able to evaluate the truth of $gt(2 + 3,1)$
- To do so requires that we first compute the value of the plus function given the arguments 2 and 3, and then send the arguments 5 and 1 to gt .

Consider the following set of facts, again involving Marcus:

- 1) Marcus was a man.
 $man(Marcus)$
- 2) Marcus was a Pompeian.
 $Pompeian(Marcus)$
- 3) Marcus was born in 40 A.D.
 $born(Marcus, 40)$
- 4) All men are mortal.
 $x: man(x) \rightarrow mortal(x)$
- 5) All Pompeians died when the volcano erupted in 79 A.D.
 $erupted(volcano, 79) \wedge \forall x : [Pompeian(x) \rightarrow died(x, 79)]$
- 6) No mortal lives longer than 150 years.
 $x: t1: At2: mortal(x) \wedge born(x, t1) \wedge gt(t2 - t1, 150) \rightarrow died(x, t2)$
- 7) It is now 1991.
 $now = 1991$

So, Above example shows how these ideas of computable functions and predicates can be useful. It also makes use of the notion of equality and allows equal objects to be substituted for each other whenever it appears helpful to do so during a proof.

- So, Now suppose we want to answer the question “Is Marcus alive?”
- The statements suggested here, there may be two ways of deducing an answer.
- Either we can show that Marcus is dead because he was killed by the volcano or we can show that he must be dead because he would otherwise be more than 150 years old, which we know is not possible.
- Also, As soon as we attempt to follow either of those paths rigorously, however, we discover, just as we did in the last example, that we need some additional knowledge. For example, our statements talk about dying, but they say nothing that relates to being alive, which is what the question is asking.

So we add the following facts:

- 8) Alive means not dead.
 $x: t: [alive(x, t) \rightarrow \neg dead(x, t)] [\neg dead(x, t) \rightarrow alive(x, t)]$
- 9) If someone dies, then he is dead at all later times.
 $x: t1: At2: died(x, t1) \wedge gt(t2, t1) \rightarrow dead(x, t2)$

So, Now let's attempt to answer the question “Is Marcus alive?” by proving: $\neg alive(Marcus, now)$

Resolution

Propositional Resolution

1. Convert all the propositions of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:
 1. Select two clauses. Call these the parent clauses.
 2. Resolve them together. The resulting clause, called the resolvent, will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and $\neg L$ such that one of the parent clauses contains L and the other contains $\neg L$, then select one such pair and eliminate both L and $\neg L$ from the resolvent.
 3. If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of classes available to the procedure.

The Unification Algorithm

- In propositional logic, it is easy to determine that two literals cannot both be true at the same time.
- Simply look for L and $\neg L$ in predicate logic, this matching process is more complicated since the arguments of the predicates must be considered.
- For example, $\text{man}(\text{John})$ and $\neg \text{man}(\text{John})$ is a contradiction, while the $\text{man}(\text{John})$ and $\neg \text{man}(\text{Spot})$ is not.
- Thus, in order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical.
- There is a straightforward recursive procedure, called the unification algorithm, that does it.

Algorithm: Unify(L_1, L_2)

1. If L_1 or L_2 are both variables or constants, then:
 1. If L_1 and L_2 are identical, then return NIL.
 2. Else if L_1 is a variable, then if L_1 occurs in L_2 then return {FAIL}, else return (L_2/L_1).
 3. Also, Else if L_2 is a variable, then if L_2 occurs in L_1 then return {FAIL}, else return (L_1/L_2).
 - d. Else return {FAIL}.
2. If the initial predicate symbols in L_1 and L_2 are not identical, then return {FAIL}.
3. If L_1 and L_2 have a different number of arguments, then return {FAIL}.
4. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all the substitutions used to unify L_1 and L_2 .)
5. For $I \leftarrow 1$ to the number of arguments in L_1 :
 1. Call Unify with the i^{th} argument of L_1 and the i^{th} argument of L_2 , putting the result in S .
 2. If S contains FAIL then return {FAIL}.
 3. If S is not equal to NIL then:
 2. Apply S to the remainder of both L_1 and L_2 .
 3. SUBST: = APPEND(S , SUBST).
6. Return SUBST.

Resolution in Predicate Logic

We can now state the resolution algorithm for predicate logic as follows, assuming a set of given statements F and a statement to be proved P:

Algorithm: Resolution

1. Convert all the statements of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in 1.
3. Repeat until a contradiction found, no progress can make, or a predetermined amount of effort has expanded.
 1. Select two clauses. Call these the parent clauses.
 2. Resolve them together. The resolvent will be the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals T1 and $\neg T2$ such that one of the parent clauses contains T2 and the other contains T1 and if T1 and T2 are unifiable, then neither T1 nor T2 should appear in the resolvent. We call T1 and T2 Complementary literals. Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should omit from the resolvent.
 3. If the resolvent is an empty clause, then a contradiction has found. Moreover, If it is not, then add it to the set of clauses available to the procedure.

Resolution Procedure

- Resolution is a procedure, which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form.
- Resolution produces proofs by refutation.
- In other words, *to prove a statement (i.e., to show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (i.e., that it is unsatisfiable).*
- The resolution procedure is a simple iterative process: at each step, two clauses, called the parent clauses, are compared (resolved), resulting in a new clause that has inferred from them. The new clause represents ways that the two parent clauses interact with each other. Suppose that there are two clauses in the system:

winter \vee summer

\neg winter \vee cold

- Now we observe that precisely one of winter and \neg winter will be true at any point.
- If winter is true, then cold must be true to guarantee the truth of the second clause. If \neg winter is true, then summer must be true to guarantee the truth of the first clause.
- Thus we see that from these two clauses we can deduce *summer \vee cold*
- This is the deduction that the resolution procedure will make.
- Resolution operates by taking two clauses that each contains the same literal, in this example, *winter*.
- Moreover, The literal must occur in the positive form in one clause and in negative form in the other. The resolvent obtained by combining all of the literals of the two parent clauses except the ones that cancel.
- If the clause that produced is the empty clause, then a contradiction has found.

For example, the two clauses

winter

\neg winter
will produce the empty clause.

Natural Deduction Using Rules

Testing whether a proposition is a tautology by testing every possible truth assignment is expensive—there are exponentially many. We need a **deductive system**, which will allow us to construct proofs of tautologies in a step-by-step fashion.

The system we will use is known as **natural deduction**. The system consists of a set of **rules of inference** for deriving consequences from premises. One builds a proof tree whose root is the proposition to be proved and whose leaves are the initial assumptions or axioms (for proof trees, we usually draw the root at the bottom and the leaves at the top).

For example, one rule of our system is known as **modus ponens**. Intuitively, this says that if we know P is true, and we know that P implies Q , then we can conclude Q .

$$\frac{P \quad P \Rightarrow Q}{Q} \text{ (modus ponens)}$$

The propositions above the line are called **premises**; the proposition below the line is the **conclusion**. Both the premises and the conclusion may contain metavariables (in this case, P and Q) representing arbitrary propositions. When an inference rule is used as part of a proof, the metavariables are replaced in a consistent way with the appropriate kind of object (in this case, propositions).

Most rules come in one of two flavors: **introduction** or **elimination** rules. Introduction rules introduce the use of a logical operator, and elimination rules eliminate it. Modus ponens is an elimination rule for \Rightarrow . On the right-hand side of a rule, we often write the name of the rule. This is helpful when reading proofs. In this case, we have written (modus ponens). We could also have written (\Rightarrow -elim) to indicate that this is the elimination rule for \Rightarrow .

Rules for Conjunction

Conjunction (\wedge) has an introduction rule and two elimination rules:

$$\frac{P \quad Q}{P \wedge Q} \text{ (\wedge-intro)} \qquad \frac{P \wedge Q}{P} \text{ (\wedge-elim-left)} \qquad \frac{P \wedge Q}{Q} \text{ (\wedge-elim-right)}$$

Rule for T

The simplest introduction rule is the one for T . It is called "unit". Because it has no premises, this rule is an **axiom**: something that can start a proof.

$$\frac{}{T} \text{ (unit)}$$

Rules for Implication

In natural deduction, to prove an implication of the form $P \Rightarrow Q$, we assume P , then reason under that assumption to try to derive Q . If we are successful, then we can conclude that $P \Rightarrow Q$.

In a proof, we are always allowed to introduce a new assumption P , then reason under that assumption. We must give the assumption a name; we have used the name x in the example below. Each distinct assumption must have a different name.

$$\frac{}{[x : P]} \text{ (assum)}$$

Because it has no premises, this rule can also start a proof. It can be used as if the proposition P were proved. The name of the assumption is also indicated here.

However, you do not get to make assumptions for free! To get a complete proof, all assumptions must be eventually *discharged*. This is done in the implication introduction rule. This rule introduces an implication $P \Rightarrow Q$ by discharging a prior assumption $[x : P]$. Intuitively, if Q can be proved under the assumption P , then the implication $P \Rightarrow Q$ holds without any assumptions. We write x in the rule name to show which assumption is discharged. This rule and modus ponens are the introduction and elimination rules for implications.

$$\frac{\begin{array}{c} [x : P] \\ \vdots \\ Q \end{array}}{P \Rightarrow Q} \quad (\Rightarrow\text{-intro}/x) \qquad \frac{P \quad P \Rightarrow Q}{Q} \quad (\Rightarrow\text{-elim, modus ponens})$$

A proof is valid only if every assumption is eventually discharged. This must happen in the proof tree below the assumption. The same assumption can be used more than once.

Rules for Disjunction

$$\frac{P}{P \vee Q} \quad (\vee\text{-intro-left}) \qquad \frac{Q}{P \vee Q} \quad (\vee\text{-intro-right}) \qquad \frac{P \vee Q \quad P \Rightarrow R \quad Q \Rightarrow R}{R} \quad (\vee\text{-elim})$$

Rules for Negation

A negation $\neg P$ can be considered an abbreviation for $P \Rightarrow \perp$:

$$\frac{P \Rightarrow \perp}{\neg P} \quad (\neg\text{-intro}) \qquad \frac{\neg P}{P \Rightarrow \perp} \quad (\neg\text{-elim})$$

Rules for Falsity

$$\frac{\begin{array}{c} [x : \neg P] \\ \vdots \\ \perp \end{array}}{P} \quad (\text{reductio ad absurdum, RAA}/x) \qquad \frac{\perp}{P} \quad (\text{ex falso quodlibet, EFQ})$$

Reductio ad absurdum (RAA) is an interesting rule. It embodies proofs by contradiction. It says that if by assuming that P is false we can derive a contradiction, then P must be true. The assumption x is discharged in the application of this rule. This rule is present in classical logic but not in **intuitionistic** (constructive) logic. In intuitionistic logic, a proposition is not considered true simply because its negation is false.

Excluded Middle

Another classical tautology that is not intuitionistically valid is the **the law of the excluded middle**, $P \vee \neg P$. We will take it as an axiom in our system. The Latin name for this rule is *tertium non datur*, but we will call it *magic*.

$$\frac{}{P \vee \neg P} \quad (\text{magic})$$

Proofs

A proof of proposition P in natural deduction starts from axioms and assumptions and derives P with all assumptions discharged. Every step in the proof is an instance of an inference rule with metavariables substituted consistently with expressions of the appropriate syntactic class.

Example

For example, here is a proof of the proposition $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)$.

$$\begin{array}{c}
 \frac{\overline{[y : A \wedge B]} \text{ (A)}}{A} \text{ (\wedge E)} \quad \frac{\overline{[x : A \Rightarrow B \Rightarrow C]} \text{ (A)}}{B \Rightarrow C} \text{ (\Rightarrow E)} \quad \frac{\overline{[y : A \wedge B]} \text{ (A)}}{B} \text{ (\wedge E)} \\
 \frac{B \Rightarrow C \quad B}{C} \text{ (\Rightarrow I, y)} \\
 \frac{A \wedge B \Rightarrow C}{(A \Rightarrow B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)} \text{ (\Rightarrow I, x)}
 \end{array}$$

The final step in the proof is to derive $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)$ from $(A \wedge B \Rightarrow C)$, which is done using the rule $(\Rightarrow\text{-intro})$, discharging the assumption $[x : A \Rightarrow B \Rightarrow C]$. To see how this rule generates the proof step, substitute for the metavariables P, Q, x in the rule as follows: $P = (A \Rightarrow B \Rightarrow C)$, $Q = (A \wedge B \Rightarrow C)$, and $x = x$. The immediately previous step uses the same rule, but with a different substitution: $P = A \wedge B$, $Q = C$, $x = y$.

The proof tree for this example has the following form, with the proved proposition at the root and axioms and assumptions at the leaves.



A proposition that has a complete proof in a deductive system is called a **theorem** of that system.

Soundness and Completeness

A measure of a deductive system's power is whether it is powerful enough to prove all true statements. A deductive system is said to be **complete** if all true statements are theorems (have proofs in the system). For propositional logic and natural deduction, this means that all tautologies must have natural deduction proofs. Conversely, a deductive system is called **sound** if all theorems are true. The proof rules we have given above are in fact sound and complete for propositional logic: every theorem is a tautology, and every tautology is a theorem. Finding a proof for a given tautology can be difficult. But once the proof is found, checking that it is indeed a proof is completely mechanical, requiring no intelligence or insight whatsoever. It is therefore a very strong argument that the thing proved is in fact true.

We can also make writing proofs less tedious by adding more rules that provide reasoning shortcuts. These rules are sound if there is a way to convert a proof using them into a proof using the original rules. Such added rules are called **admissible**.

Procedural versus Declarative Knowledge

We have discussed various search techniques in previous units. Now we would consider a set of rules that represent,

1. Knowledge about relationships in the world and
2. Knowledge about how to solve the problem using the content of the rules.

Procedural vs Declarative Knowledge

Procedural Knowledge

- A representation in which the control information that is necessary to use the knowledge is embedded in the knowledge itself for e.g. computer programs, directions, and recipes; these indicate specific use or implementation;
- The real difference between declarative and procedural views of knowledge lies in where control information reside.

For example, consider the following

Man (Marcus)

Man (Caesar)

Person (Cleopatra)

$\forall x: \text{Man}(x) \rightarrow \text{Person}(x)$

Now, try to answer the question. *?Person(y)*

The knowledge base justifies any of the following answers.

Y=Marcus

Y=Caesar

Y=Cleopatra

- We get more than one value that satisfies the predicate.
- If only one value needed, then the answer to the question will depend on the order in which the assertions examined during the search for a response.
- If the assertions declarative then they do not themselves say anything about how they will be examined. In case of procedural representation, they say how they will examine.

Declarative Knowledge

- A statement in which knowledge specified, but the use to which that knowledge is to be put is not given.
- For example, laws, people's name; these are the facts which can stand alone, not dependent on other knowledge;
- So to use declarative representation, we must have a program that explains what is to do with the knowledge and how.
- For example, a set of logical assertions can combine with a resolution theorem prover to give a complete program for solving problems but in some cases, the logical assertions can view as a program rather than data to a program.
- Hence the implication statements define the legitimate reasoning paths and automatic assertions provide the starting points of those paths.
- These paths define the execution paths which is similar to the 'if then else' in traditional programming.
- So logical assertions can view as a procedural representation of knowledge.

Logic Programming – Representing Knowledge Using Rules

- Logic programming is a programming paradigm in which logical assertions viewed as programs.
- These are several logic programming systems, PROLOG is one of them.
- ***A PROLOG program consists of several logical assertions where each is a horn clause i.e. a clause with at most one positive literal.***
- Ex : $P, P \vee Q, P \rightarrow Q$
- The facts are represented on Horn Clause for two reasons.
 1. Because of a uniform representation, a simple and efficient interpreter can write.
 2. The logic of Horn Clause decidable.

- Also, The first two differences are the fact that PROLOG programs are actually sets of Horn clause that have been transformed as follows:-
 1. If the Horn Clause contains no negative literal then leave it as it is.
 2. Also, Otherwise rewrite the Horn clauses as an implication, combining all of the negative literals into the antecedent of the implications and the single positive literal into the consequent.
- Moreover, This procedure causes a clause which originally consisted of a disjunction of literals (one of them was positive) to be transformed into a single implication whose antecedent is a conjunction universally quantified.
- But when we apply this transformation, any variables that occurred in negative literals and so now occur in the antecedent become existentially quantified, while the variables in the consequent are still universally quantified.

For example the PROLOG clause $P(x) : - Q(x, y)$ is equal to logical expression $\forall x: \exists y: Q(x, y) \rightarrow P(x)$.

- The difference between the logic and PROLOG representation is that the PROLOG interpretation has a fixed control strategy. And so, the assertions in the PROLOG program define a particular search path to answer any question.
- But, the logical assertions define only the set of answers but not about how to choose among those answers if there is more than one.

Consider the following example:

1. Logical representation

$\forall x : pet(x) \sqcap small(x) \rightarrow apartmentpet(x)$
 $\forall x : cat(x) \sqcap dog(x) \rightarrow pet(x)$
 $\forall x : poodle(x) \rightarrow dog(x) \sqcap small(x)$
poodle (fluffy)

2. Prolog representation

apartmentpet(x) : pet(x), small(x)
pet(x): cat(x)
pet(x): dog(x)
dog(x): poodle(x)
small(x): poodle(x)
poodle (fluffy)

Forward versus Backward Reasoning

Forward versus Backward Reasoning

A search procedure must find a path between initial and goal states.

There are two directions in which a search process could proceed.

The two types of search are:

1. Forward search which starts from the start state
2. Backward search that starts from the goal state

The production system views the forward and backward as symmetric processes.

Consider a game of playing 8 puzzles. The rules defined are

Square 1 empty and square 2 contains tile n. \rightarrow

- Also, Square 2 empty and square 1 contains the tile n .

Square 1 empty Square 4 contains tile n . \rightarrow

- Also, Square 4 empty and Square 1 contains tile n .

We can solve the problem in 2 ways:

1. Reason forward from the initial state

- Step 1. Begin building a tree of move sequences by starting with the initial configuration at the root of the tree.
- Step 2. Generate the next level of the tree by finding all rules **whose left-hand side matches** against the root node. The right-hand side is used to create new configurations.
- Step 3. Generate the next level by considering the nodes in the previous level and applying it to all rules whose left-hand side match.

2. Reasoning backward from the goal states:

- Step 1. Begin building a tree of move sequences by starting with the goal node configuration at the root of the tree.
- Step 2. Generate the next level of the tree by finding all rules **whose right-hand side matches** against the root node. The left-hand side used to create new configurations.
- Step 3. Generate the next level by considering the nodes in the previous level and applying it to all rules whose right-hand side match.
- So, The same rules can use in both cases.
- Also, In forwarding reasoning, the left-hand sides of the rules matched against the current state and right sides used to generate the new state.
- Moreover, In backward reasoning, the right-hand sides of the rules matched against the current state and left sides are used to generate the new state.

There are four factors influencing the type of reasoning. They are,

1. Are there more possible start or goal state? We move from smaller set of sets to the length.
2. In what direction is the branching factor greater? We proceed in the direction with the lower branching factor.
3. Will the program be asked to justify its reasoning process to a user? If, so then it is selected since it is very close to the way in which the user thinks.
4. What kind of event is going to trigger a problem-solving episode? If it is the arrival of a new factor, the forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

Example 1 of Forward versus Backward Reasoning

- It is easier to drive from an unfamiliar place from home, rather than from home to an unfamiliar place. Also, If you consider a home as starting place an unfamiliar place as a goal then we have to backtrack from unfamiliar place to home.

Example 2 of Forward versus Backward Reasoning

- Consider a problem of symbolic integration. Moreover, The problem space is a set of formulas, which contains integral expressions. Here START is equal to the given formula with some integrals. GOAL is equivalent to the expression of the formula without any integral. Here we start from the formula with some integrals and proceed to an integral free expression rather than starting from an integral free expression.

Example 3 of Forward versus Backward Reasoning

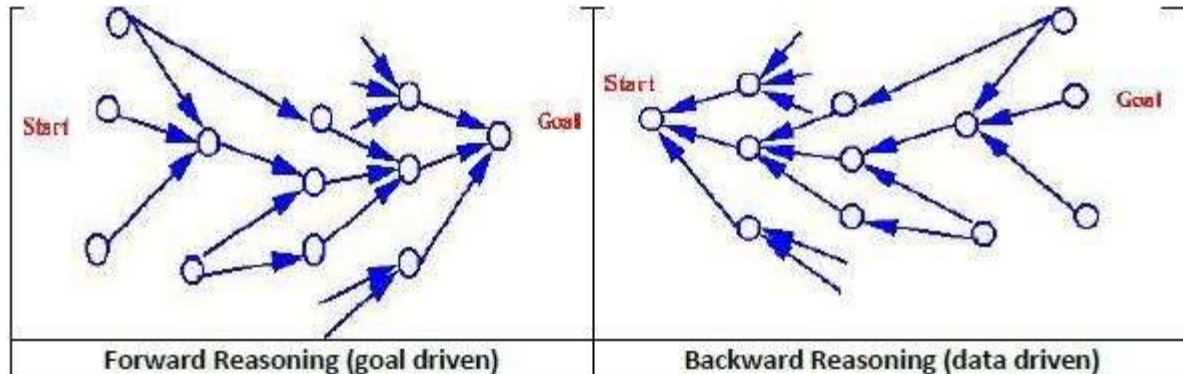
- The third factor is nothing but deciding whether the reasoning process can justify its reasoning. If it justifies then it can apply. For example, doctors are usually unwilling to accept any advice from diagnostics process because it cannot explain its reasoning.

Example 4 of Forward versus Backward Reasoning

- Prolog is an example of backward chaining rule system. In Prolog rules restricted to Horn clauses. This allows for rapid indexing because all the rules for deducing a given fact share the same rule head. Rules matched with unification procedure. Unification tries to find a set of bindings for variables to equate a sub-goal with the head of some rule. Rules in the Prolog program matched in the order in which they appear.

Combining Forward and Backward Reasoning

- Instead of searching either forward or backward, you can search both simultaneously.
- Also, That is, start forward from a starting state and backward from a goal state simultaneously until the paths meet.
- This strategy called Bi-directional search. The following figure shows the reason for a Bidirectional search to be ineffective.



Forward versus Backward Reasoning

- Also, The two searches may pass each other resulting in more work.
- Based on the form of the rules one can decide whether the same rules can apply to both forward and backward reasoning.
- Moreover, If left-hand side and right of the rule contain pure assertions then the rule can reverse.
- And so the same rule can apply to both types of reasoning.
- If the right side of the rule contains an arbitrary procedure then the rule cannot reverse.
- So, In this case, while writing the rule the commitment to a direction of reasoning must make.

Symbolic Reasoning Under Uncertainty

Symbolic Reasoning

- The reasoning is the act of deriving a conclusion from certain properties using a given methodology.
- The reasoning is a process of thinking; reasoning is logically arguing; reasoning is drawing the inference.
- *When a system is required to do something, that it has not been explicitly told how to do, it must reason. It must figure out what it needs to know from what it already knows.*

- Many types of Reasoning have been identified and recognized, but many questions regarding their logical and computational properties still remain controversial.
- The popular methods of Reasoning include abduction, induction, model-based, explanation and confirmation. All of them are intimately related to problems of belief revision and theory development, knowledge absorption, discovery, and learning.

Logical Reasoning

- Logic is a language for reasoning. It is a collection of rules called Logic arguments, we use when doing logical reasoning.
- The logic reasoning is the process of drawing conclusions from premises using rules of inference.
- The study of logic divided into formal and informal logic. The formal logic is sometimes called symbolic logic.
- Symbolic logic is the study of symbolic abstractions (construct) that capture the formal features of logical inference by a formal system.
- The formal system consists of two components, a formal language plus a set of inference rules.
- The formal system has axioms. Axiom is a sentence that is always true within the system.
- Sentences derived using the system's axioms and rules of derivation called theorems.
- The Logical Reasoning is of our concern in AI.

Approaches to Reasoning

- There are three different approaches to reasoning under uncertainties.
 1. Symbolic reasoning
 2. Statistical reasoning
 3. Fuzzy logic reasoning

Symbolic Reasoning

- The basis for intelligent mathematical software is the integration of the “power of symbolic mathematical tools” with the suitable “proof technology”.
- Mathematical reasoning enjoys a property called monotonicity, that says, “If a conclusion follows from given premises A, B, C... then it also follows from any larger set of premises, as long as the original premises A, B, C.. included.”
- Moreover, Human reasoning is not monotonic.
- People arrive at conclusions only tentatively; based on partial or incomplete information, reserve the right to retract those conclusions while they learn new facts. Such reasoning non-monotonic, precisely because the set of accepted conclusions have become smaller when the set of premises expanded.

Formal Logic

Moreover, The Formal logic is the study of inference with purely formal content, i.e. where content made explicit.

Examples – Propositional logic and Predicate logic.

- Here the logical arguments are a set of rules for manipulating symbols. The rules are of two types,
 1. Syntax rules: say how to build meaningful expressions.
 2. Inference rules: say how to obtain true formulas from other true formulas.
- Moreover, Logic also needs semantics, which says how to assign meaning to expressions.

Uncertainty in Reasoning

- The world is an uncertain place; often the Knowledge is imperfect which causes uncertainty.
- So, Therefore reasoning must be able to operate under uncertainty.
- Also, AI systems must have the ability to reason under conditions of uncertainty.

Monotonic Reasoning

- A reasoning process that moves in one direction only.
- Moreover, The number of facts in the knowledge base is always increasing.
- The conclusions derived are valid deductions and they remain so.

A monotonic logic cannot handle

1. Reasoning by default: because consequences may derive only because of lack of evidence to the contrary.
2. Abductive reasoning: because consequences only deduced as most likely explanations.
3. Belief revision: because new knowledge may contradict old beliefs.

Introduction to Nonmonotonic Reasoning

Non-monotonic Reasoning

The definite clause logic is **monotonic** in the sense that anything that could be concluded before a clause is added can still be concluded after it is added; adding knowledge does not reduce the set of propositions that can be derived.

A logic is **non-monotonic** if some conclusions can be invalidated by adding more knowledge.

The logic of definite clauses with negation as failure is non-monotonic. Non-monotonic reasoning is useful for representing defaults. A **default** is a rule that can be used unless it overridden by an exception.

For example, to say that b is normally true if c is true, a knowledge base designer can write a rule of the form

$$b \leftarrow c \wedge \sim ab_a.$$

where ab_a is an atom that means abnormal with respect to some aspect a . Given c , the agent can infer b unless it is told ab_a . Adding ab_a to the knowledge base can prevent the conclusion of b .

Rules that imply ab_a can be used to prevent the default under the conditions of the body of the rule.

Example 5.27: Suppose the purchasing agent is investigating purchasing holidays. A resort may be adjacent to a beach or away from a beach. This is not symmetric; if the resort was adjacent to a beach, the knowledge provider would specify this. Thus, it is reasonable to have the clause $away_from_beach \leftarrow \sim on_beach$.

This clause enables an agent to infer that a resort is away from the beach if the agent is not told it is adjacent to a beach.

A **cooperative system** tries to not mislead. If we are told the resort is on the beach, we would expect that resort users would have access to the beach. If they have access to a beach, we would expect them to be able to swim at the beach. Thus, we would expect the following defaults:

$$beach_access \leftarrow on_beach \wedge \sim ab_{beach_access}.$$

$$swim_at_beach \leftarrow beach_access \wedge \sim ab_{swim_at_beach}.$$

A cooperative system would tell us if a resort on the beach has no beach access or if there is no swimming. We could also specify that, if there is an enclosed bay and a big city, then there is no swimming, by default:

$$ab_{swim_at_beach} \leftarrow enclosed_bay \wedge big_city \wedge \sim ab_{no_swimming_near_city}.$$

We could say that British Columbia is abnormal with respect to swimming near cities:

$ab_{no_swimming_near_city} \leftarrow in_BC \wedge \sim ab_{BC_beaches}$.

Given only the preceding rules, an agent infers *away_from_beach*. If it is then told *on_beach*, it can no longer infer *away_from_beach*, but it can now infer *beach_access* and *swim_at_beach*. If it is also told *enclosed_bay* and *big_city*, it can no longer infer *swim_at_beach*. However, if it is then told *in_BC*, it can then infer *swim_at_beach*.

By having defaults of what is normal, a user can interact with the system by telling it what is abnormal, which allows for economy in communication. The user does not have to state the obvious.

One way to think about non-monotonic reasoning is in terms of **arguments**. The rules can be used as components of arguments, in which the negated abnormality gives a way to undermine arguments. Note that, in the language presented, only positive arguments exist that can be undermined. In more general theories, there can be positive and negative arguments that attack each other.

Implementation Issues

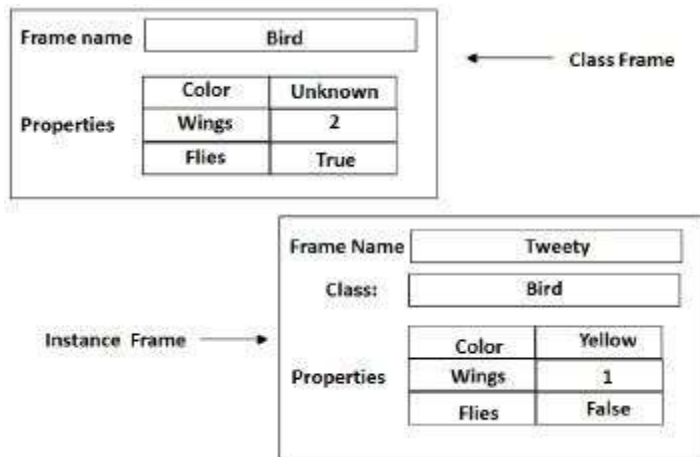
Weak Slot and Filler Structures

Evolution Frames

- As seen in the previous example, there are certain problems which are difficult to solve with Semantic Nets.
- Although there is no clear distinction between a semantic net and frame system, more structured the system is, more likely it is to be termed as a frame system.
- A frame is a collection of attributes (called slots) and associated values that describe some entities in the world. Sometimes a frame describes an entity in some absolute sense;
- Sometimes it represents the entity from a particular point of view only.
- A single frame taken alone is rarely useful; we build frame systems out of collections of frames that connected to each other by virtue of the fact that the value of an attribute of one frame may be another frame.

Frames as Sets and Instances

- The set theory is a good basis for understanding frame systems.
- Each frame represents either a class (a set) or an instance (an element of class)
- Both *isa* and *instance* relations have inverse attributes, which we call subclasses & all instances.
- As a class represents a set, there are 2 kinds of attributes that can be associated with it.
 1. Its own attributes &
 2. Attributes that are to be inherited by each element of the set.



Frames as Sets and Instances

- Sometimes, the difference between a set and an individual instance may not be clear.
- Example: Team India is an instance of the class of Cricket Teams and can also think of as the set of players.
- Now the problem is if we present Team India as a subclass of Cricket teams, then Indian players automatically become part of all the teams, which is not true.
- So, we can make Team India a subclass of class called Cricket Players.
- To do this we need to differentiate between regular classes and meta-classes.
- Regular Classes are those whose elements are individual entities whereas Meta-classes are those special classes whose elements are themselves, classes.
- The most basic meta-class is the class *CLASS*.
- It represents the set of all classes.
- All classes are instances of it, either directly or through one of its subclasses.
- The class *CLASS* introduces the attribute cardinality, which is to inherited by all instances of *CLASS*. Cardinality stands for the number.

Other ways of Relating Classes to Each Other

- We have discussed that a class1 can be a subset of class2.
- If Class2 is a meta-class then Class1 can be an instance of Class2.
- Another way is the ***mutually-disjoint-with*** relationship, which relates a class to one or more other classes that guaranteed to have no elements in common with it.
- Another one is, ***is-covered-by*** which relates a class to a set of subclasses, the union of which is equal to it.
- If a class is-covered-by a set S of mutually disjoint classes, then S called a partition of the class.

Slots as Full-Fledged Objects (Frames)

Till now we have used attributes as slots, but now we will represent attributes explicitly and describe their properties.

Some of the properties we would like to be able to represent and use in reasoning include,

- The class to which the attribute can attach.
- Constraints on either the type or the value of the attribute.
- A default value for the attribute. Rules for inheriting values for the attribute.
- To be able to represent these attributes of attributes, we need to describe attributes (slots) as frames.

- These frames will organize into an *isa* hierarchy, just as any other frames, and that hierarchy can then be used to support inheritance of values for attributes of slots.
- Now let us formalize what is a slot. A slot here is a relation.
- It maps from elements of its domain (the classes for which it makes sense) to elements of its range (its possible values).
- A relation is a set of ordered pairs.
- Thus it makes sense to say that relation R1 is a subset of another relation R2.
- In that case, R1 is a specialization of R2. Since a slot is a set, the set of all slots, which we will call SLOT, is a meta-class.
- Its instances are slots, which may have sub-slots.

Frame Example

In this example, the frames Person, Adult-Male, ML-Baseball-Player (corresponding to major league baseball players), Pitcher, and ML-Baseball-Team (for major league baseball team) are all classes.

```

Person
  isa : Mammal
  cardinality : 6,000,000,000
  * handed : Right
Adult-Male
  isa : Person
  cardinality : 2,000,000,000
  * height : 5-10
ML-Baseball-Player
  isa : Adult-Male
  cardinality : 624
  * height : 6-1
  * bats : equal to handed
  * batting-average : .252
  * team :
  * uniform-color :
Fielder
  isa : ML-Baseball-Player
  cardinality : 376
  * batting-average : .262
Pee-Wee-Reese
  instance : Fielder
  height : 5-10
  bats : Right
  batting-average : .309
  team : Brooklyn-Dodgers
  uniform-color : Blue
ML-Baseball-Team
  isa : Team
  cardinality : 26
  * team-size : 24
  * manager :
Brooklyn-Dodgers
  instance : ML-Baseball-Team
  team-size : 24
  manager : Leo-Durocher
  players : {Pee-Wee-Reese,...}

```

- The frames Pee-Wee-Reese and Brooklyn-Dodgers are instances.
- The *isa* relation that we have been using without a precise definition is, in fact, the subset relation. The set of adult males is a subset of the set of people.
- The set of major league baseball players subset of the set of adult males, and so forth.
- Our instance relation corresponds to the relation element-of. Pee Wee Reese is an element of the set of fielders.
- Thus he is also an element of all of the supersets of fielders, including major league baseball players and people. The transitivity of *isa* follows directly from the transitivity of the subset relation.

- Both the isa and instance relations have inverse attributes, which we call subclasses and all instances.
- Because a class represents a set, there are two kinds of attributes that can associate with it.
- Some attributes are about the set itself, and some attributes are to inherited by each element of the set.
- We indicate the difference between these two by prefixing the latter with an asterisk (*).
- For example, consider the class ML-Baseball-Player, we have shown only two properties of it as a set: It is a subset of the set of adult males. And it has cardinality 624.
- We have listed five properties that all major league baseball players have (height, bats, batting average, team, and uniform-color), and we have specified default values for the first three of them.
- By providing both kinds of slots, we allow both classes to define a set of objects and to describe a prototypical object of the set.
- Frames are useful for representing objects that are typical of stereotypical situations.
- The situation like the structure of complex physical objects, visual scenes, etc.
- A commonsense knowledge can represent using default values if no other value exists. Commonsense is generally used in the absence of specific knowledge.

Semantic Nets

- Inheritance property can represent using **isa** and **instance**
- Monotonic Inheritance can perform substantially more efficiently with such structures than with pure logic, and non-monotonic inheritance is also easily supported.
- The reason that makes Inheritance easy is that the knowledge in slot and filler systems is structured as a set of entities and their attributes.

These structures turn out to be useful as,

- It indexes assertions by the entities they describe. As a result, retrieving the value for an attribute of an entity is fast.
- Moreover, It makes easy to describe properties of relations. To do this in a purely logical system requires higher-order mechanisms.
- It is a form of object-oriented programming and has the advantages that such systems normally include modularity and ease of viewing by people.

Here we would describe two views of this kind of structure – Semantic Nets & Frames.

Semantic Nets

- There are different approaches to knowledge representation include semantic net, frames, and script.
- The semantic net describes both objects and events.
- In a semantic net, information represented as a set of nodes connected to each other by a set of labeled arcs, which represents relationships among the nodes.
- It is a directed graph consisting of vertices which represent concepts and edges which represent semantic relations between the concepts.
- It is also known as associative net due to the association of one node with other.
- The main idea is that the meaning of the concept comes from the ways in which it connected to other concepts.
- We can use inheritance to derive additional relations.

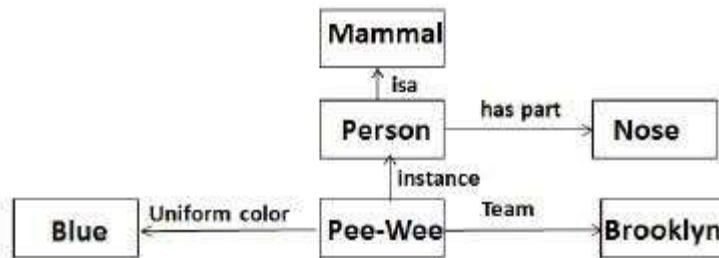


Figure: A Semantic Network

Intersection Search Semantic Nets

- We try to find relationships among objects by spreading activation out from each of two nodes. And seeing where the activation meets.
- Using this we can answer the questions like, what is the relation between India and Blue.
- It takes advantage of the entity-based organization of knowledge that slot and filler representation provides.

Representing Non-binary Predicates Semantic Nets

- Simple binary predicates like *isa(Person, Mammal)* can represent easily by semantic nets but other non-binary predicates can also represent by using general-purpose predicates such as *isa* and *instance*.
- Three or even more place predicates can also convert to a binary form by creating one new object representing the entire predicate statement and then introducing binary predicates to describe a relationship to this new object.

Conceptual Dependency

Introduction to Strong Slot and Filler Structures

- The main problem with semantic networks and frames is that they lack formality; there is no specific guideline on how to use the representations.
- In frame when things change, we need to modify all frames that are relevant – this can be time-consuming.
- Strong slot and filler structures typically represent links between objects according to more rigid rules, specific notions of what types of object and relations between them are provided and represent knowledge about common situations.
- Moreover, We have types of strong slot and filler structures:
 1. Conceptual Dependency (CD)
 2. Scripts
 3. Cyc

Conceptual Dependency (CD)

Conceptual Dependency originally developed to represent knowledge acquired from natural language input.

The goals of this theory are:

- To help in the drawing of the inference from sentences.
- To be independent of the words used in the original input.
- That is to say: For any 2 (or more) sentences that are identical in meaning there should be only one representation of that meaning.

Moreover, It has used by many programs that portend to understand English (MARGIE, SAM, PAM).

Conceptual Dependency (CD) provides:

- A structure into which nodes representing information can be placed.
- Also, A specific set of primitives.
- A given level of granularity.

Sentences are represented as a series of diagrams depicting actions using both abstract and real physical situations.

- The agent and the objects represented.
- Moreover, The actions are built up from a set of primitive acts which can modify by tense.

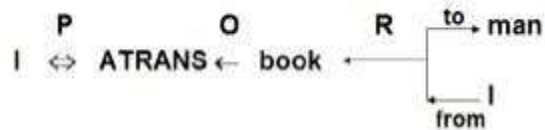
CD is based on events and actions. Every event (if applicable) has:

- an ACTOR or an ACTION performed by the Actor
- Also, an OBJECT that the action performs on
- A DIRECTION in which that action is oriented

These are represented as slots and fillers. In English sentences, many of these attributes left out.

A Simple Conceptual Dependency Representation

For the sentence, "I have a book to the man" CD representation is as follows:



Where the symbols have the following meaning.

- Arrows indicate directions of dependency.
- Moreover, The double arrow indicates the two-way link between actor and action.
- O — for the object case relation
- R – for the recipient case relation
- P – for past tense
- D – destination

Primitive Acts of Conceptual Dependency Theory

ATRANS

- Transfer of an abstract relationship (i.e. give)

PTRANS

- Transfer of the physical location of an object (e.g., go)

PROPEL

- Also, Application of physical force to an object (e.g. push)

MOVE

- Moreover, Movement of a body part by its owner (e.g. kick)

GRASP

- Grasping of an object by an action (e.g. throw)

INGEST

- Ingesting of an object by an animal (e.g. eat)

EXPEL

- Expulsion of something from the body of an animal (e.g. cry)

MTRANS

- Transfer of mental information (e.g. tell)

MBUILD

- Building new information out of old (e.g. decide)

SPEAK

- Producing of sounds (e.g. say)

ATTEND

- Focusing of a sense organ toward a stimulus (e.g. listen)

There are four conceptual categories. These are,

ACT

- Actions {one of the CD primitives}

PP

- Also, Objects {picture producers}

AA

- Modifiers of actions {action aiders}

PA

- Modifiers of PP's {picture aiders}

Advantages of Conceptual Dependency

- Using these primitives involves fewer inference rules.
- So, Many inference rules already represented in CD structure.
- Moreover, The holes in the initial structure help to focus on the points still to established.

Disadvantages of Conceptual Dependency

- Knowledge must decompose into fairly low-level primitives.
- Impossible or difficult to find the correct set of primitives.
- Also, A lot of inference may still require.
- Representations can be complex even for relatively simple actions.
- Consider: Dave bet Frank five pounds that Wales would win the Rugby World Cup.
- Moreover, Complex representations require a lot of storage.

Scripts

Scripts Strong Slot

- A script is a structure that prescribes a set of circumstances which could be expected to follow on from one another.
- It is similar to a thought sequence or a chain of situations which could be anticipated.
- It could be considered to consist of a number of slots or frames but with more specialized roles.

Scripts are beneficial because:

- Events tend to occur in known runs or patterns.
- Causal relationships between events exist.
- Entry conditions exist which allow an event to take place
- Prerequisites exist for events taking place. E.g. when a student progresses through a degree scheme or when a purchaser buys a house.

Script Components

Each script contains the following main components.

- Entry Conditions: Must be satisfied before events in the script can occur.
- Results: Conditions that will be true after events in script occur.
- Props: Slots representing objects involved in the events.
- Roles: Persons involved in the events.
- Track: the Specific variation on the more general pattern in the script. Different tracks may share many components of the same script but not all.

- Scenes: The sequence of events that occur. Events represented in conceptual dependency form.

Advantages and Disadvantages of Script

Advantages

- Capable of predicting implicit events
- Single coherent interpretation may be build up from a collection of observations.

Disadvantage

- More specific (inflexible) and less general than frames.
- Not suitable to represent all kinds of knowledge.

To deal with inflexibility, smaller modules called memory organization packets (MOP) can combine in a way that appropriates for the situation.

Script Example

Script : Play in theater	Various Scenes
Track: Play in Theater Props: <ul style="list-style-type: none"> • Tickets • Seat • Play Roles: <ul style="list-style-type: none"> • Person (who wants to see a play) – P • Ticket distributor – TD • Ticket checker – TC Entry Conditions: <ul style="list-style-type: none"> • P wants to see a play • P has a money Results: <ul style="list-style-type: none"> • P saw a play • P has less money • P is happy (optional if he liked the play) 	Scene 1: Going to theater <ul style="list-style-type: none"> • P PTRANS P into theater • P ATTEND eyes to ticket counter
	Scene 2: Buying ticket <ul style="list-style-type: none"> • P PTRANS P to ticket counter • P MTRANS (need a ticket) to TD • TD ATRANS ticket to P
	Scene 3: Going inside hall of theater and sitting on a seat <ul style="list-style-type: none"> • P PTRANS P into Hall of theater • TC ATTEND eyes on ticket POSS_by P • TC MTRANS (showed seat) to P • P PTRANS P to seat • P MOVES P to sitting position
	Scene 4: Watching a play <ul style="list-style-type: none"> • P ATTEND eyes on play • P MBUILD (good moments) from play
	Scene 5: Exiting <ul style="list-style-type: none"> • P PTRANS P out of Hall and theater

- It must activate based on its significance.
- If the topic important, then the script should open.
- If a topic just mentioned, then a pointer to that script could hold.
- For example, given “John enjoyed the play in theater”, a script “Play in Theater” suggested above invoke.
- All implicit questions can answer correctly.

Here the significance of this script is high.

- Did John go to the theater?
- Also, Did he buy the ticket?
- Did he have money?

If we have a sentence like “John went to the theater to pick his daughter”, then invoking this script will lead to many wrong answers.

- Here significance of the script theater is less.

Getting significance from the story is not straightforward. However, some heuristics can apply to get the value.

CYC

What is CYC?

- An ambitious attempt to form a very large knowledge base aimed at capturing commonsense reasoning.
- Initial goals to capture knowledge from a hundred randomly selected articles in the Encyclopedia Britannica.
- Also, Both Implicit and Explicit knowledge encoded.
- Moreover, Emphasis on study of underlying information (assumed by the authors but not needed to tell to the readers).

Example: Suppose we read that Wellington learned of Napoleon's death • Then we (humans) can conclude Napoleon never new that Wellington had died.

How do we do this?

So, We require special implicit knowledge or commonsense such as:

- We only die once.
- You stay dead.
- Moreover, You cannot learn anything when dead.
- Time cannot go backward.

Why build large knowledge bases:

1. Brittleness
 - Specialised knowledge bases are brittle. Hard to encode new situations and non-graceful degradation in performance. Commonsense based knowledge bases should have a firmer foundation.
2. Form and Content
 - Moreover, Knowledge representation may not be suitable for AI. Commonsense strategies could point out where difficulties in content may affect the form.
3. Shared Knowledge
 - Also, Should allow greater communication among systems with common bases and assumptions.

How is CYC coded?

- By hand.
- Special CYCL language:
- LISP-like.
- Frame-based
- Multiple inheritances
- Slots are fully fledged objects.
- Generalized inheritance — any link not just *isa* and *instance*.

Module 2

Game Playing:

Game Playing

- Charles Babbage, the nineteenth-century computer architect thought about programming his analytical engine to play chess and later of building a machine to play tic-tac-toe.
- There are two reasons that games appeared to be a good domain.
 1. They provide a structured task in which it is very easy to measure success or failure.
 2. They are easily solvable by straightforward search from the starting state to a winning position.
- The first is true for all games but the second is not true for all, except simplest games.
- For example, consider chess.
- The average branching factor is around 35. In an average game, each player might make 50.
- So in order to examine the complete game tree, we would have to examine 35^{100}
- Thus it is clear that a simple search is not able to select even its first move during the lifetime of its opponent.
- It is clear that to improve the effectiveness of a search based problem-solving program two things can do.
 1. Improve the generate procedure so that only good moves generated.
 2. Improve the test procedure so that the best move will recognize and explored first.
- If we use legal-move generator then the test procedure will have to look at each of them because the test procedure must look at so many possibilities, it must be fast.
- Instead of the legal-move generator, we can use plausible-move generator in which only some small numbers of promising moves generated.
- As the number of lawyers available moves increases, it becomes increasingly important in applying heuristics to select only those moves that seem more promising.
- The performance of the overall system can improve by adding heuristic knowledge into both the generator and the tester.
- In game playing, a goal state is one in which we win but the game like chess. It is not possible. Even we have good plausible move generator.
- The depth of the resulting tree or graph and its branching factor is too great.
- It is possible to search tree only ten or twenty moves deep then in order to choose the best move. The resulting board positions must compare to discover which is most advantageous.
- This is done using static evaluation function, which uses whatever information it has to evaluate individual board position by estimating how likely they are to lead eventually to a win.
- Its function is similar to that of the heuristic function h' in the A* algorithm: in the absence of complete information, choose the most promising position.

MINIMAX Search Procedure

- The minimax search is a depth-first and depth limited procedure.
- The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions.

- Now we can apply the static evolution function to those positions and simply choose the best one.
- After doing so, we can back that value up to the starting position to represent our evolution of it.
- Here we assume that static evolution function returns larger values to indicate good situations for us.
- So our goal is to maximize the value of the static evaluation function of the next board position.
- The opponents' goal is to minimize the value of the static evaluation function.
- **The alternation of maximizing and minimizing at alternate ply when evaluations are to be pushed back up corresponds to the opposing strategies of the two players is called MINIMAX.**
- It is the recursive procedure that depends on two procedures
 - MOVEGEN(position, player)— The plausible-move generator, which returns a list of nodes representing the moves that can make by Player in Position.
 - STATIC(position, player)— static evaluation function, which returns a number representing the goodness of Position from the standpoint of Player.
- With any recursive program, we need to decide when recursive procedure should stop.
- There are the variety of factors that may influence the decision they are,
 - Has one side won?
 - How many plies have we already explored? Or how much time is left?
 - How stable is the configuration?
- We use DEEP-ENOUGH which assumed to evaluate all of these factors and to return TRUE if the search should be stopped at the current level and FALSE otherwise.
- It takes two parameters, position, and depth, it will ignore its position parameter and simply return TRUE if its depth parameter exceeds a constant cut off value.
- One problem that arises in defining MINIMAX as a recursive procedure is that it needs to return not one but two results.
 - The backed-up value of the path it chooses.
 - The path itself. We return the entire path even though probably only the first element, representing the best move from the current position, actually needed.
- We assume that MINIMAX returns a structure containing both results and we have two functions, VALUE and PATH that extract the separate components.
- Initially, It takes three parameters, a board position, the current depth of the search, and the player to move,
 - MINIMAX(current,0,player-one) If player –one is to move
 - MINIMAX(current,0,player-two) If player –two is to move

Adding alpha-beta cutoffs

- Minimax procedure is a depth-first process. One path is explored as far as time allows, the static evolution function is applied to the game positions at the last step of the path.
- The efficiency of the depth-first search can improve by branch and bound technique in which partial solutions that clearly worse than known solutions can abandon early.
- It is necessary to modify the branch and bound strategy to include two bounds, one for each of the players.
- This modified strategy called alpha-beta pruning.