www.myreaders.info/ , RC Chakraborty, e-mail rcchak@gmail.com , June 01, 2010

www.myreaders.info/html/artificial_intelligence.html

Knowledge Representation

Issues, Predicate Logic, Rules

Artificial Intelligence

Knowledge representation in AI, topics : knowledge progression, model, category, typology map, and relationship; Mapping between facts and representation, forward and backward representation, KR system requirements; KR schemes – relational, inheritable, inferential, declarative and procedural; Issues in KR - attributes, relationship, granularity. Knowledge representation using logic : Propositional logic - statements, variables, symbols, connective, truth value, contingencies, tautologies, contradictions, antecedent, consequent, argument; Predicate logic – predicate, logic expressions, quantifiers, formula; Representing "IsA" and "Instance" relationships; Computable functions and predicates. Knowledge representation using rules : declarative, procedural, and meta rules; Logic programming characteristics - statement, language, syntax, data objects, clause, predicate, sentence, subject, queries; Programming paradigms – models of computation, imperative, functional, and logic model; Reasoning, and conflict.

Knowledge Representation

Issues, Predicate Logic, Rules

Artificial Intelligence

Topics

(Lectures 15, 16, 17, 18, 19, 20, 21, 22 8 hours)

Knowledge Representation

Issues, Predicate Logic, Rules

How do we Represent what we know ?

- **Knowledge** is a general term.

  An answer to the question, *"how to represent knowledge"*, requires an analysis to distinguish between knowledge *"how"* and knowledge *"that".*

  - *knowing "how to do something".*

    e.g. "how to drive a car" is a **Procedural knowledge***.*
  - *knowing "that something is true or false".*

    e.g. "that is the speed limit for a car on a motorway" is a **Declarative knowledge.**

- **knowledge and Representation** *are two distinct entities. They play a central but distinguishable roles in intelligent system.*

  - Knowledge is a description of the world.

    It determines a *system's competence* by what it knows.
  - 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.

  The Knowledge Representation *models/mechanisms* are often based on:

  - **Logic**
  - **Rules**
  - **Frames**
  - **Semantic Net**

- Different types of knowledge require different kinds of *reasoning*.

1. Introduction

Knowledge is a general term.

Knowledge is a progression that starts with *data* which is of limited utility.

- By organizing or analyzing the data, we understand what the data means, and this becomes *information*.
- The interpretation or evaluation of information yield *knowledge*.
- An understanding of the principles embodied within the knowledge is *wisdom*.

● **Knowledge Progression**

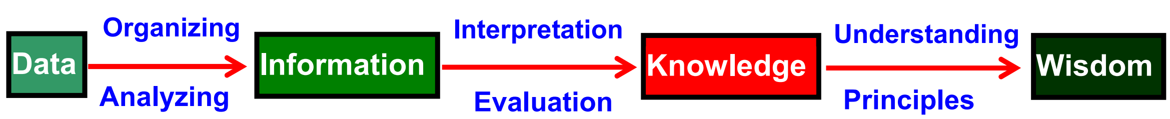


Fig 1 Knowledge Progression

- **Data** is viewed as collection of *disconnected facts.* : Example : It is raining.
- **Information** emerges when *relationships among facts* are established and understood; Provides answers to *"who", "what", "where", and "when"*. : Example : The temperature dropped 15 degrees and then it started raining.
- **Knowledge** emerges when *relationships among patterns* are identified and understood; Provides answers as *"how"* . : Example : If the humidity is very high and the temperature drops substantially, then atmospheres is unlikely to hold the moisture, so it rains.
- **Wisdom** is the pinnacle of understanding, uncovers the *principles of relationships that describe patterns*. Provides answers as *"why"* . : Example : Encompasses understanding of all the interactions that happen between raining, evaporation, air currents, temperature gradients and changes.

- **Knowledge Model** (Bellinger 1980)

A knowledge model tells, that as the degree of *"connectedness"* and *"understanding"* increases, we progress from *data* through *information* and *knowledge* to *wisdom.*

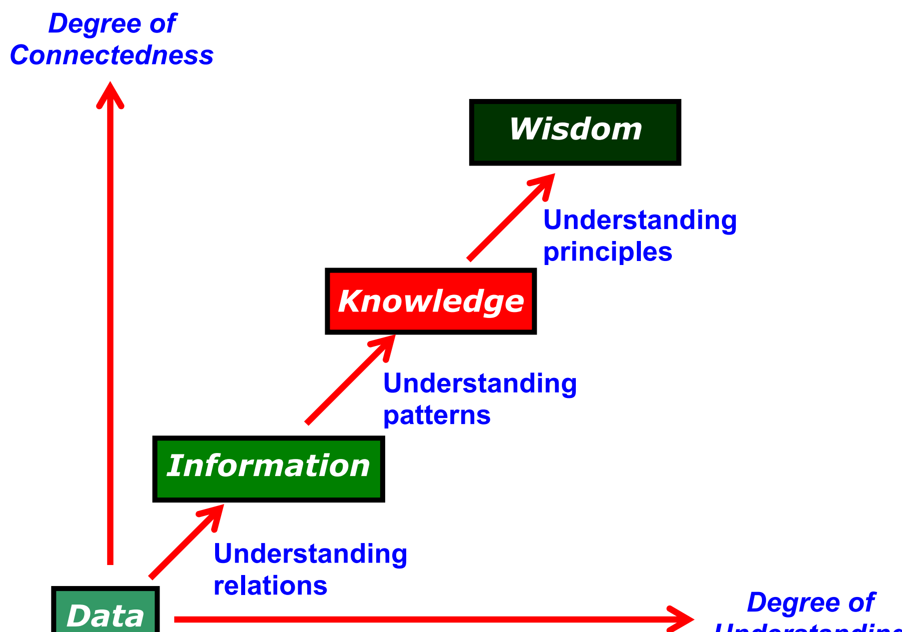


Fig. Knowledge Model

The model represents *transitions* and *understanding*.

- the **transitions** are from *data*, to *information*, to *knowledge*, and finally to *wisdom*;
- the ***understanding*** support the transitions from one stage to the next stage.

The distinctions between *data, information, knowledge*, and *wisdom* are not very discrete. They are more like shades of gray, rather than black and white (Shedroff, 2001).

- *"data"* and *"information*" deal with the past; they are based on the gathering of *facts* and adding context.
- *"knowledge*" deals with the present that enable us to perform.
- *"wisdom"* deals with the future, acquire vision for what will be, rather than for what is or was.

- **Knowledge Category**

Knowledge is categorized into two major types: *Tacit* and *Explicit*.

- term "**Tacit**" corresponds to *"informal" or "implicit"* type of knowledge,
- term "**Explicit**" corresponds to *"formal"* type of knowledge.

Tacit knowledge	**Explicit knowledge**
◇ Exists within a human being; it is embodied.	◇ Exists outside a human being; it is embedded.
◇ Difficult to articulate formally.	◇ Can be articulated formally.
◇ Difficult to communicate or share.	◇ Can be shared, copied, processed and stored.
◇ Hard to steal or copy.	◇ Easy to steal or copy
◇ Drawn from experience, action, subjective insight.	◇ Drawn from artifact of some type as principle, procedure, process, concepts.

(The next slide explains more about tacit and explicit knowledge).

Knowledge Typology Map

The map shows two types of knowledge - *Tacit* and *Explicit* knowledge.

- *Tacit knowledge* comes from *"experience", "action", "subjective" , "insight"*
- *Explicit knowledge* comes from *"principle", "procedure", "process", "concepts",* via transcribed content or artifact of some type.

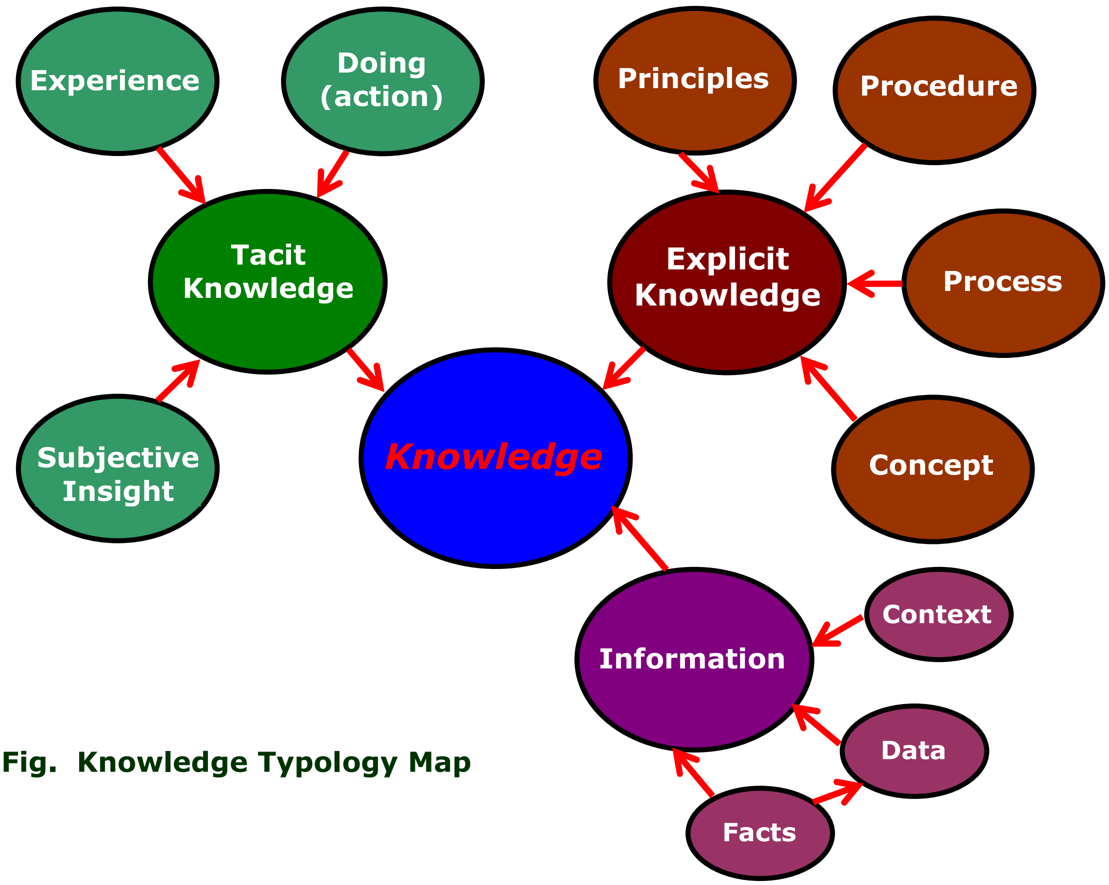


Fig. Knowledge Typology Map

- **Facts :** are data or instance that are specific and unique.
- **Concepts :** are class of items, words, or ideas that are known by a common name and share common features.
- **Processes** : are flow of events or activities that describe how things work rather than how to do things.
- **Procedures** : are series of step-by-step actions and decisions that result in the achievement of a task.
- **Principles :** are guidelines, rules, and parameters that govern; principles allow to make predictions and draw implications;

These artifacts are used in the knowledge creation process to create two types of knowledge: **declarative** and **procedural** explained below.

- **Knowledge Type**

Cognitive psychologists sort knowledge into **Declarative** and **Procedural** category and some researchers added **Strategic** as a third category.

- About *procedural knowledge*, there is some disparity in views.
  - One, it is close to Tacit knowledge, it manifests itself in the doing of something yet cannot be expressed in words; e.g., we read faces and moods.
  - Another, it is close to declarative knowledge; the difference is that a task or method is described instead of facts or things.
- All *declarative knowledge* are explicit knowledge; it is knowledge that can be and has been articulated.
- The *strategic knowledge* is thought as a subset of declarative knowledge.

Procedural knowledge	**Declarative knowledge**
◇ Knowledge about *"how to do something"*; e.g., to determine if Peter or Robert is older, first find their ages.	◇ Knowledge about *"that something is true or false". e.g.,* A car has four tyres; Peter is older than Robert;
◇ Focuses on tasks that must be performed to reach a particular objective or goal.	◇ Refers to representations of objects and events; knowledge about facts and relationships;
◇ Examples *:* procedures, rules, strategies, agendas, models.	◇ Example *:* concepts, objects, facts, propositions, assertions, semantic nets, logic and descriptive models.

- **Relationship among Knowledge Type**

The relationship among *explicit*, *implicit*, *tacit*, *declarative* and *procedural knowledge* are illustrated below.

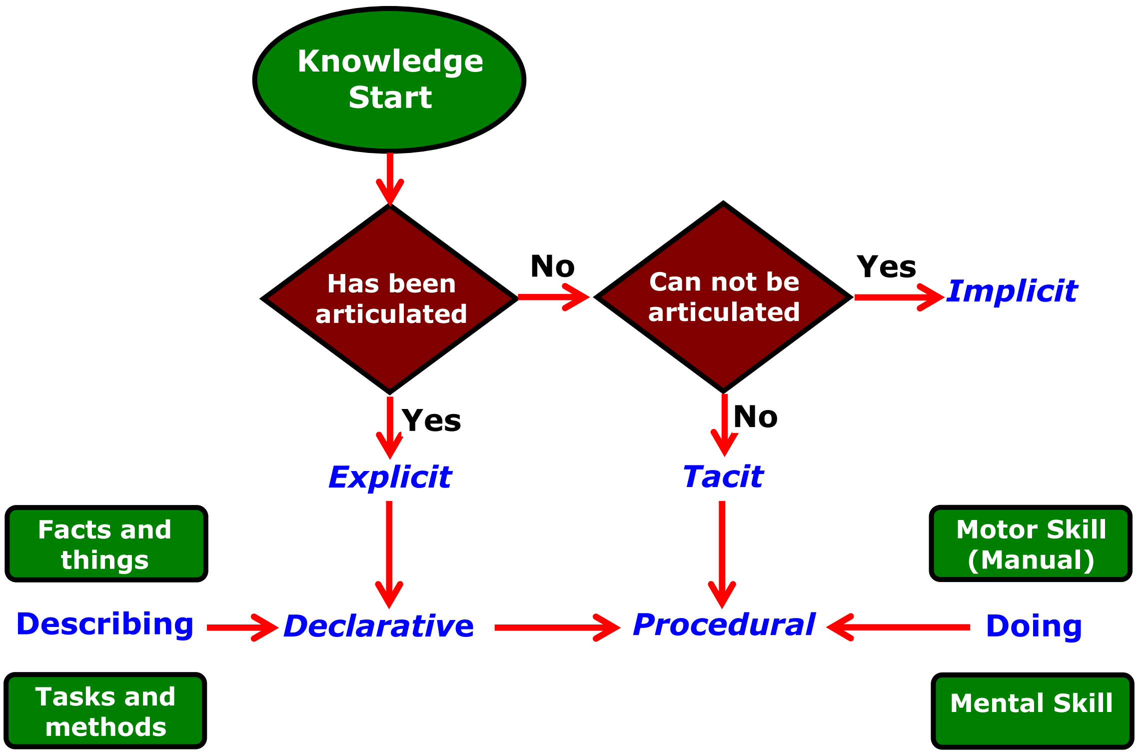


Fig. Relationship among types of knowledge

The Figure shows :

Declarative knowledge is tied to *"describing"* and

Procedural knowledge is tied to *"doing."*

Vertical arrows connecting *explicit* with *declarative* and *tacit* with *procedural,* indicate the strong relationships exist among them.

Horizontal arrow connecting *declarative* and *procedural* indicates that we often develop *procedural knowledge* as a result of starting with *declarative knowledge*. i.e., we often "know about" before we "know how".

Therefore, we may view :

– all procedural knowledge as tacit knowledge, and

– all declarative knowledge as explicit knowledge.

1.1 Framework of Knowledge Representation (Poole 1998)

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

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. The computer can then use an algorithm to compute an answer. This process is illustrated below.

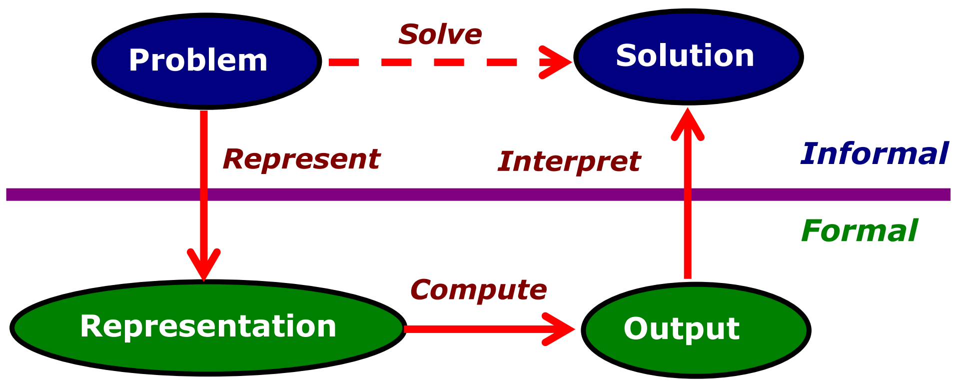


Fig. Knowledge Representation Framework

The steps are

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

Note : The Problem solving requires

- formal knowledge representation, and
- conversion of informal knowledge to formal knowledge , that is conversion of implicit knowledge to explicit knowledge.

- **Knowledge and Representation**

Problem solving requires large amount of knowledge and some mechanism for manipulating that knowledge.

The Knowledge and the Representation are distinct entities, play a central but distinguishable roles in intelligent system.

- **Knowledge** is a description of the world;
  it determines a *system's competence* by what it knows.
- **Representation** is the way knowledge is encoded;
  it defines the *system's performance* in doing something.

In simple words, we *:*

- need to know about *things we want to represent* , and
- need some means by which *things we can manipulate*.

◇ know things to represent	‡ Objects	- facts about objects in the domain.
	‡ Events	- actions that occur in the domain.
	‡ Performance	- knowledge about how to do things
	‡ Meta-knowledge	- knowledge about what we know
◇ need means to manipulate	‡ Requires some formalism	- to what we represent ;

Thus, knowledge representation can be considered at two levels :

(a) *knowledge level* at which facts are described, and

(b) *symbol level* at which the representations of the objects, defined in terms of symbols, can be manipulated in the programs.

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

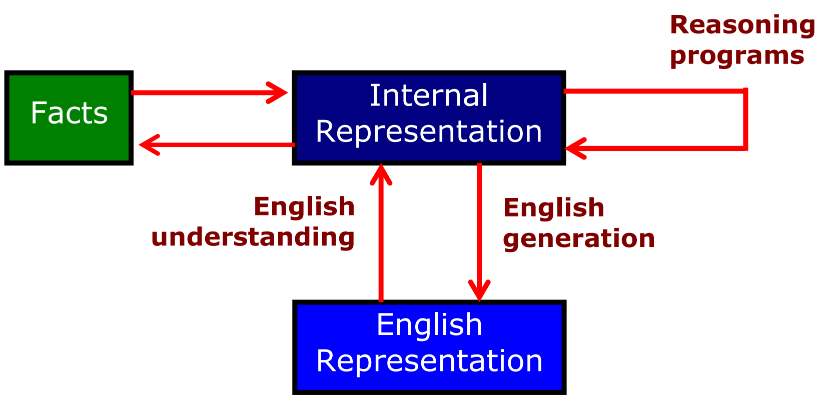
- **Mapping between Facts and Representation**

Knowledge is a collection of *"facts"* from some domain.

We need a representation of *"facts"* that can be manipulated by a program. Normal English is insufficient, too hard currently for a computer program to draw inferences in natural languages.
Thus some symbolic representation is necessary.

Therefore, we must be able to map *"facts to symbols"* and *"symbols to facts"* using *forward and backward representation mapping.*

Example : Consider an English sentence



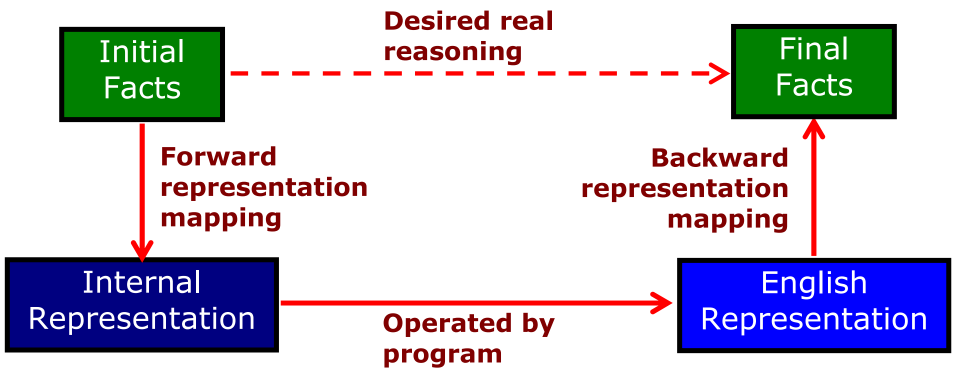
Facts	Representations
◇ **Spot is a dog**	A *fact* represented in *English sentence*
◇ **dog (Spot)**	Using *forward mapping function* the above *fact* is represented in *logic*
◇ $\forall$ **x : dog(x) $\rightarrow$ hastail (x)**	A *logical representation* of the *fact* that *"all dogs have tails"*

Now using ***deductive mechanism*** we can generate a new representation of object :

◇ **hastail (Spot)**	A new object representation
◇ **Spot has a tail** ***[it is new knowledge]***	Using *backward mapping function* to generate English sentence

Forward and Backward Representation

The forward and backward representations are elaborated below :



- The doted line on top indicates the *abstract reasoning* process that a program is intended to model.

- The solid lines on bottom indicates the *concrete reasoning* process that the program performs.

- **KR System Requirements**

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

A knowledge representation system should have following properties.

◇ Representational Adequacy	The ability to represent all kinds of knowledge that are needed in that domain.
◇ Inferential Adequacy	The ability to manipulate the representational structures to derive new structure corresponding to new knowledge inferred from old .
◇ 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.
◇ Acquisitional Efficiency	The ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

Note : To date no single system can optimizes all of the above properties**.**

1.2 Knowledge Representation Schemes

There are four types of Knowledge representation :

Relational, Inheritable, Inferential, and Declarative/Procedural.

- **Relational Knowledge :**
  - provides a framework to compare two objects based on equivalent attributes.
  - any instance in which two different objects are compared is a relational type of knowledge.
- **Inheritable Knowledge**
  - is obtained from associated objects.
  - it prescribes a structure in which new objects are created which may inherit all or a subset of attributes from existing objects.
- **Inferential Knowledge**
  - is inferred from objects through relations among objects.
  - e.g., a word alone is a simple syntax, but with the help of other words in phrase the reader may infer more from a word; this inference within linguistic is called semantics.
- **Declarative Knowledge**
  - a statement in which knowledge is specified, but the use to which that knowledge is to be put is not given.
  - e.g. laws, people's name; these are facts which can stand alone, not dependent on other knowledge;

  Procedural Knowledge
  - a representation in which the control information, to use the knowledge, is embedded in the knowledge itself.
  - e.g. computer programs, directions, and recipes; these indicate specific use or implementation;

These KR schemes are detailed in next few slides

- **Relational Knowledge :**

This knowledge associates elements of one domain with another domain.

- Relational knowledge is made up of objects consisting of attributes and their corresponding associated values.
- The results of this knowledge type is a mapping of elements among different domains.

The table below shows a simple way to store facts.

- The facts about a set of objects are put systematically in columns.
- This representation provides little opportunity for inference.

Table - Simple Relational Knowledge

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 simple question such as :

" Who is the heaviest player ? ".

but if a procedure for finding heaviest player is provided, then these facts will enable that procedure to compute an answer.

‡ We can ask things like who "bats – left" and "throws – right".

- **Inheritable Knowledge :**

Here the knowledge elements inherit attributes from their parents.

The knowledge is 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 be prescribed to the child elements.

The *inheritance* is a powerful form of inference, but not adequate. The basic KR needs to be augmented with inference mechanism.

The KR in hierarchical structure, shown below, is called *"semantic network"* or a collection of *"frames" or "slot-and-filler structure".* The structure shows property inheritance and way for insertion of additional knowledge.

Property inheritance : The objects or elements of specific classes inherit attributes and values from more general classes. The classes are organized in a generalized hierarchy.

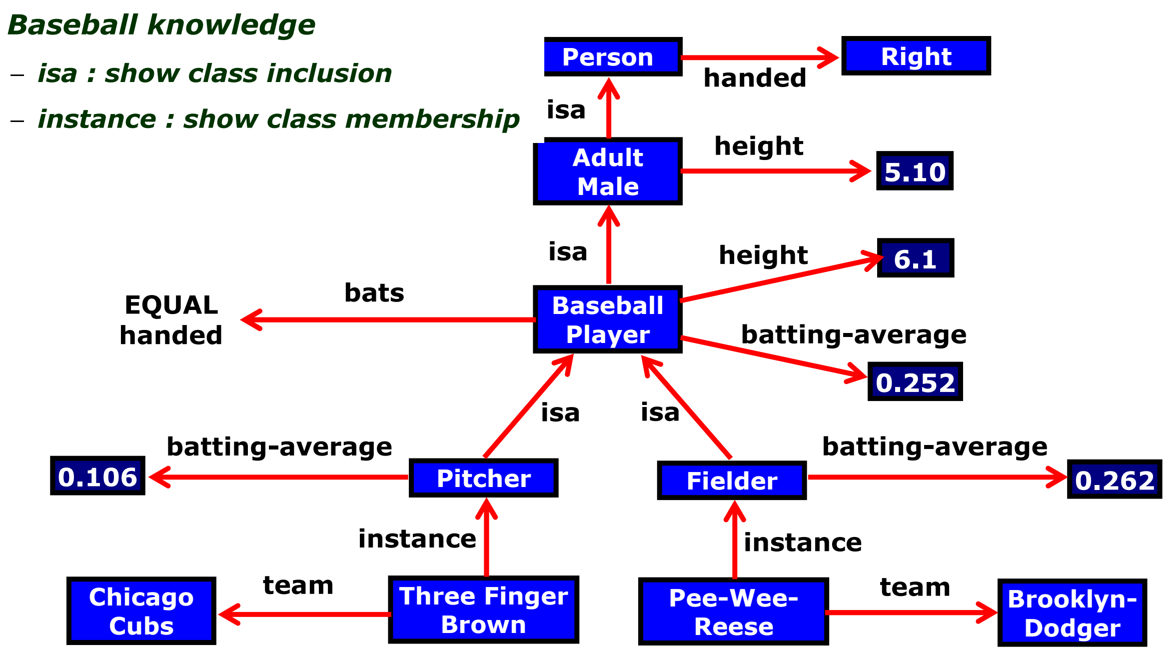


Fig. Inheritable knowledge representation (KR)

- The directed arrows represent *attributes* (*isa, instance, team*) originates at object being described and terminates at object or its value.
- The box nodes represents *objects* and *values* of the attributes.

[Continued in the next slide]

[from previous slide – example]

- **Viewing a node as a frame**

  Example : Baseball-player

isa :	*Adult-Male*
Bates :	*EQUAL handed*
Height :	*6.1*
Batting-average :	*0.252*

- **Algorithm :** Property Inheritance

  Retrieve a value **V** for an attribute **A** of an instance object **O**.

  Steps to follow:

  1. Find object **O** in the knowledge base.
  2. If there is a value for the attribute **A** then report that value.
  3. Else, if there is a value for the attribute instance; If not, then fail.
  4. Else, move to the node corresponding to that value and look for a value for the attribute **A**; If one is found, report it.
  5. Else, do until there is no value for the "**isa**" attribute or until an answer is found :
     (a) Get the value of the "**isa**" attribute and move to that node.
     (b) See if there is a value for the attribute **A**; If yes, report it.

  This algorithm is simple. It describes the basic mechanism of inheritance. It does not say what to do if there is more than one value of the instance or "**isa**" attribute.

  This can be applied to the example of knowledge base illustrated, in the previous slide, to derive answers to the following queries :

  – team (Pee-Wee-Reese) = Brooklyn–Dodger
  – batting–average(Three-Finger-Brown) = 0.106
  – height (Pee-Wee-Reese) = 6.1
  – bats (Three Finger Brown) = right

  [For explanation - refer book on AI by Elaine Rich & Kevin Knight, page 112]

- **Inferential Knowledge :**

This knowledge generates new information from the given information.

This new information does not require further data gathering form 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) is used to infer from a set of attributes.
- inference through predicate logic uses a set of logical operations to relate individual data.
- the symbols used for the logic operations are :

  " → " (implication), " ¬ " (not), " V " (or), " Λ " (and),
  " ∀ " (for all), " ∃ " (there exists).

Examples of predicate logic statements :

1. *"Wonder"* is a name of a dog : **dog (wonder)**
2. All dogs belong to the class of animals : **∀ x : dog (x) → animal(x)**
3. All animals either live on land or in water : **∀ x : animal(x) → live (x, land) V live (x, water)**

From these three statements we can infer that :

" *Wonder* lives either on land or on water."

Note : If more information is made available about these objects and their relations, then more knowledge can be inferred.

- **Declarative/Procedural Knowledge**

Differences between Declarative/Procedural knowledge is not very clear.

Declarative knowledge :

Here, the knowledge is based on declarative facts about *axioms* and *domains* .

- axioms are assumed to be true unless a counter example is found to invalidate them.
- domains represent the physical world and the perceived functionality.
- axiom and domains thus simply exists and serve as declarative statements that can stand alone.

Procedural knowledge:

Here, the knowledge is a mapping process between domains that specify **"what to do when"** and the representation is of **"how to make it"** rather than *"what it is"*. The procedural knowledge :

- may have inferential efficiency, but no inferential adequacy and acquisitional efficiency.
- are represented as small programs that know how to do specific things, how to proceed.

Example : A parser in a natural language has the knowledge that a noun phrase may contain articles, adjectives and nouns. It thus accordingly call routines that know how to process articles, adjectives and nouns.

1.3 Issues in Knowledge Representation

The fundamental goal of Knowledge Representation is to facilitate inferencing (conclusions) from knowledge.

The issues that arise while using KR techniques are many. Some of these are explained below.

- Important Attributes :
  Any attribute of objects so basic that they occur in almost every problem domain ?

- Relationship among attributes:
  Any important relationship that exists among object attributes ?

- Choosing Granularity :
  At what level of detail should the knowledge be represented ?

- Set of objects :
  How sets of objects be represented ?

- Finding Right structure :
  Given a large amount of knowledge stored, how can relevant parts be accessed ?

Note : These issues are briefly explained, referring previous example, Fig. Inheritable KR. For detail readers may refer book on AI by Elaine Rich & Kevin Knight- page 115 – 126.

- **Important Attributes** : *(Ref. Example - Fig. Inheritable KR)*

  There are attributes that are of general significance.

  There are two attributes **"instance"** and **"isa"**, that are of general importance. These attributes are important because they support *property inheritance*.

- **Relationship among Attributes** : *(Ref. Example- Fig. Inheritable KR)*

  The attributes to describe objects are themselves entities they represent.

  The relationship between the attributes of an object, independent of specific knowledge they encode, may hold properties like:
  Inverses, existence in an isa hierarchy, techniques for reasoning about values and single valued attributes.

  ◇ **Inverses :**
  This is about *consistency check*, while a value is added to one attribute. The entities are related to each other in many different ways. The figure shows attributes *(isa, instance, and team)*, each with a directed arrow, originating at the object being described and terminating either at the object or its value.
  There are two ways of realizing this:
  - first, represent two relationships in a *single representation*; e.g., a logical representation, ***team(Pee-Wee-Reese, Brooklyn–Dodgers)***, that can be interpreted as a statement about Pee-Wee-Reese or Brooklyn–Dodger.
  - second, use attributes that focus on a *single entity but use them in pairs*, one the inverse of the other; for e.g., one, ***team = Brooklyn–Dodgers*** , and the other, ***team = Pee-Wee-Reese, ...***

  This second approach is followed in semantic net and frame-based systems, accompanied by a knowledge acquisition tool that guarantees the consistency of inverse slot by checking, each time a value is added to one attribute then the corresponding value is added to the inverse.

◊ **Existence in an "isa" hierarchy :**

This is about *generalization-specialization*, like, classes of objects and specialized subsets of those classes. There are attributes and specialization of attributes.

Example: the attribute *"height"* is a specialization of general attribute *"physical-size"* which is, in turn, a specialization of *"physical-attribute"*. These generalization-specialization relationships for attributes are important because they support inheritance.

◊ **Techniques for reasoning about values :**

This is about *reasoning values of attributes* not given explicitly.

Several kinds of information are used in reasoning, like,

height : must be in a unit of length,

age : of person can not be greater than the age of person's parents.

The values are often specified when a knowledge base is created.

◊ **Single valued attributes :**

This is about a *specific attribute* that is guaranteed to take a unique value.

Example : A baseball player can at time have only a single height and be a member of only one team. KR systems take different approaches to provide support for single valued attributes.

- **Choosing Granularity**

  What level should the knowledge be represented and

  what are the primitives ?

  – Should there be a small number or should there be a large number of low-level primitives or High-level facts.

  – High-level facts may not be adequate for inference while Low-level primitives may require a lot of storage.

  Example of Granularity :

  – Suppose we are interested in following facts

  John spotted Sue.

  – This could be represented as

  Spotted (agent(John), object (Sue))

  – Such a representation would make it easy to answer questions such are

  Who spotted Sue ?

  – Suppose we want to know

  Did John see Sue ?

  – Given only one fact, we cannot discover that answer.

  – We can add other facts, such as

  Spotted (x , y) $\rightarrow$ saw (x , y)

  – We can now infer the answer to the question.

- **Set of Objects**

Certain properties of objects that are true as member of a set but not as individual;

Example : Consider the assertion made in the sentences

"there are more *sheep* than *people* in Australia", and

"*English* speakers can be found all over the world."

To describe these facts, the only way is to attach assertion to the sets representing people, sheep, and English**.**

The reason to represent sets of objects is :

If a property is true for all or most elements of a set,

then it is more efficient to associate it once with the set

rather than to associate it explicitly with every elements of the set .

This is done in different ways :

- in logical representation through the use of *universal quantifier*, and
- in hierarchical structure where node represent sets, the *inheritance propagate* set level assertion down to individual.

Example: assert **large (elephant)**;

Remember to make clear distinction between,

- whether we are asserting some property of the set itself,
  means, **the set of elephants is large**, or

- asserting some property that holds for individual elements of the set ,
  means, **any thing that is an elephant is large**.

There are three ways in which sets may be represented :

(a) Name, as in the example – Ref Fig. Inheritable KR, the node - Baseball-Player and the predicates as Ball and Batter in logical representation.

(b) Extensional definition is to list the numbers, and

(c) In tensional definition is to provide a rule, that returns true or false depending on whether the object is in the set or not.

[Readers may refer book on AI by Elaine Rich & Kevin Knight- page 122 - 123]

- **Finding Right Structure**

  Access to right structure for describing a particular situation.

  It requires, selecting an initial structure and then revising the choice.
  While doing so, it is necessary to solve following problems :

  - how to perform an initial selection of the most appropriate structure.
  - how to fill in appropriate details from the current situations.
  - how to find a better structure if the one chosen initially turns out not to be appropriate.
  - what to do if none of the available structures is appropriate.
  - when to create and remember a new structure.

  There is no good, general purpose method for solving all these problems.
  Some knowledge representation techniques solve some of them.

  [Readers may refer book on AI by Elaine Rich & Kevin Knight- page 124 - 126]

2. KR Using Predicate Logic

In the previous section much has been illustrated about knowledge and KR related issues. This section, illustrates :

How knowledge can be represented as *"symbol structures"* that characterize bits of knowledge about objects, concepts, facts, rules, strategies;

Examples : **"red"** represents *colour red*;
"car1" represents *my car* ;
"red(car1)" represents fact that *my car is red*.

Assumptions about KR :

- *Intelligent Behavior* can be achieved by manipulation of symbol structures.
- *KR languages* are designed to facilitate operations over symbol structures, have precise syntax and semantics;

  Syntax tells which expression is legal ?,

  e.g., red1(car1), red1 car1, car1(red1), red1(car1 & car2) ?; and

  Semantic tells what an expression means ?

  e.g., property "dark red" applies to my car.
- *Make Inferences, draw new conclusions* from existing facts.

To satisfy these assumptions about KR, we need formal notation that allow automated inference and problem solving. One popular choice is use of **logic.**

- **Logic**

Logic is concerned with the truth of statements about the world.

Generally each statement is either *TRUE* or *FALSE*.

Logic includes : *Syntax* , *Semantics* *and* *Inference Procedure*.

- **Syntax :**

  Specifies the *symbols* in the language about how they can be combined to form sentences. The facts about the world are represented as sentences in logic.

- **Semantic :**

  Specifies how to assign a truth value to a sentence based on its *meaning* in the world. It Specifies what facts a sentence refers to.

  A fact is a claim about the world, and it may be *TRUE* or *FALSE*.

- **Inference Procedure :**

  Specifies *methods* for computing new sentences from the existing sentences.

Note

Facts : are claims about the world that are *True* or *False*.

Representation : is an expression (sentence), stands for the *objects* and *relations*.

*Sentence*s : can be encoded in a computer program.

- **Logic as a KR Language**

  Logic is a language for reasoning, a collection of rules used while doing logical reasoning. Logic is studied as KR languages in artificial intelligence.

  - **Logic** is a formal system in which the formulas or sentences have true or false values.

  - **Problem of designing KR language** is a tradeoff between that which is

    (a) *Expressive* enough to represent important objects and relations in a problem domain.

    (b) *Efficient* enough in reasoning and answering questions about implicit information in a reasonable amount of time.

  - **Logics are of different types :** *Propositional logic, Predicate logic, Temporal logic, Modal logic, Description logic* etc;
    They represent things and allow more or less efficient inference.

  - **Propositional logic and Predicate logic are fundamental to all logic.**
    Propositional Logic is the study of statements and their connectivity.
    Predicate Logic is the study of individuals and their properties.

2.1 Logic Representation

Logic can be used to represent simple facts.

The *facts* are claims about the world that are *True* or *False*.

To build a Logic-based representation :

- User defines a set of primitive *symbols* and the associated *semantics.*
- Logic defines ways of putting symbols together so that user can define legal *sentences* in the language that represent *TRUE* facts.
- Logic defines ways of inferring *new sentences* from existing ones.
- Sentences - either *TRUE* or *false* but not both are called *propositions.*
- A declarative sentence expresses a *statement* with a proposition as content; example:

  the declarative **"snow is white"** expresses that **snow is white**;

  further, **"snow is white"** expresses that *snow is white* is **TRUE***.*

In this section, first **Propositional Logic (PL)** is briefly explained and then the **Predicate logic** is illustrated in detail.

- **Propositional Logic (PL)**

A proposition is a statement, which in English would be a declarative sentence. Every proposition is either TRUE or FALSE.

Examples: **(a) The sky is blue., (b) Snow is cold. , (c) 12 * 12=144**

‡ Propositions are "sentences" , either true or false but not both.

‡ A sentence is smallest unit in propositional logic.

‡ If proposition is true, then truth value is "true" .
If proposition is false, then truth value is "false" .

Example :

Sentence	Truth value	Proposition (Y/N)
"Grass is green"	"true"	Yes
"2 + 5 = 5"	"false"	Yes
"Close the door"	-	No
"Is it hot out side ?"	-	No
"x > 2" where x is variable	-	No (since x is not defined)
"x = x"	-	No (don't know what is "x" and "="; "3 = 3" or "air is equal to air" or "Water is equal to water" has no meaning)

- Propositional logic is fundamental to all logic.
- Propositional logic is also called Propositional calculus, Sentential calculus, or Boolean algebra.
- Propositional logic tells the ways of joining and/or modifying entire propositions, statements or sentences to form more complicated propositions, statements or sentences, as well as the logical relationships and properties that are derived from the methods of combining or altering statements.

Statement, Variables and Symbols

These and few more related terms, such as, *connective, truth value, contingencies, tautologies, contradictions, antecedent, consequent, argument* are explained below.

Statement

Simple statements (sentences), TRUE or FALSE, that does not contain any other statement as a part, are basic propositions; lower-case letters, **p, q, r,** are symbols for simple statements.

Large, compound or complex statement are constructed from basic propositions by combining them with connectives.

Connective or Operator

The connectives join simple statements into compounds, and joins compounds into larger compounds.

Table below indicates, the *basic connectives* and their symbols :

- listed in decreasing order of operation priority;
- operations with higher priority is solved first.

Example of a formula : $((((a \land \neg b) \lor c \to d) \leftrightarrow \neg (a \lor c))$

Connectives and Symbols in decreasing order of operation priority

Connective	***Symbols***					**Read as**
assertion	**P**					**"p is true"**
negation	**$\neg$p**	**~**	**!**		**NOT**	**"p is false"**
conjunction	**p $\land$ q**	**·**	**&&**	**&**	**AND**	**"both p and q are true"**
disjunction	**P v q**	**$\Vert$**	**$\vert$**		**OR**	**"either p is true, or q is true, or both "**
implication	**p $\to$ q**	**$\supset$**	**$\Rightarrow$**		**if ..then**	**"if p is true, then q is true" " p implies q "**
equivalence	**$\leftrightarrow$**	**$\equiv$**	**$\Leftrightarrow$**		**if and only if**	**"p and q are either both true or both false"**

Note : The propositions and connectives are the basic elements of propositional logic.

◇ Truth Value

The truth value of a statement is its *TRUTH* or *FALSITY* ,

Example :

p is either *TRUE* or *FALSE*,

~p is either *TRUE* or *FALSE*,

p v q is either *TRUE* or *FALSE*, and so on.

use " **T** " or " **1** " to mean *TRUE*.

use " **F** " or " **0** " to mean *FALSE*

Truth table **defining the basic *connectives* :**

p	q	¬p	¬q	p ∧ q	p v q	p→q	p ↔ q	q→p
T	T	F	F	T	T	T	T	T
T	F	F	T	F	T	F	F	T
F	T	T	F	F	T	T	F	F
F	F	T	T	F	F	T	T	T

[The next slide shows the truth values of a group of propositions, called tautology, contradiction, contingency, antecedent, consequent. They form argument where one proposition claims to follow logically other proposition]

◊ Tautologies

A proposition that is always true is called a "tautology".

e.g., **(P v ¬P)** is always true regardless of the truth value of the proposition **P**.

◊ Contradictions

A proposition that is always false is called a "contradiction".

e.g., **(P ∧ ¬P)** is always false regardless of the truth value of the proposition **P**.

◊ Contingencies

A proposition is called a "contingency" , if that proposition is neither a *tautology* nor a *contradiction* .

e.g., **(P v Q)** is a contingency.

◊ Antecedent, Consequent

These two are parts of conditional statements.

In the conditional statements, $\mathbf{p \rightarrow q}$, the

1st statement or **"if - clause"** (here **p**) is called **antecedent** ,

2nd statement or **"then - clause"** (here **q**) is called **consequent.**

◇ Argument

An argument is a demonstration or a proof of some statement.

Example : **"That bird is a crow; therefore, it's black."**

Any argument can be expressed as a compound statement.

In logic, an argument is a set of one or more meaningful declarative sentences (or "propositions") known as the *premises* along with another meaningful declarative sentence (or "proposition") known as the *conclusion*.

- **Premise** is a proposition which gives reasons, grounds, or evidence for accepting some other proposition, called the conclusion.
- **Conclusion** is a proposition, which is purported to be established on the basis of other propositions.

Take all the premises, conjoin them, and make that conjunction the antecedent of a conditional and make the conclusion the consequent. This implication statement is called the corresponding *conditional of the argument.*

Note : Every argument has a corresponding conditional, and every implication statement has a corresponding argument. Because the corresponding conditional of an argument is a statement, it is therefore either a tautology, or a contradiction, or a contingency.

‡ An argument is **valid**
"if and only if" its corresponding conditional is a *tautology*.

‡ Two statements are **consistent**
"if and only if" their conjunction is not a contradiction.

‡ Two statements are **logically equivalent**
"if and only if" their truth table columns are identical;
"if and only if" the statement of their equivalence using **"** $\equiv$ **"** is a tautology.

Note **:** The truth tables are adequate to test *validity, tautology, contradiction, contingency, consistency, and equivalence*.

- **Predicate Logic**

The propositional logic, is not powerful enough for all types of assertions;

Example : The assertion **"x > 1"**, where **x** is a variable, is not a proposition because it is neither true nor false unless value of **x** is defined.

For **x > 1** to be a proposition ,

- either we substitute a specific number for **x** ;
- or change it to something like

  "There is a number x for which x > 1 holds";
- or **"For every number x, x > 1 holds"**.

Consider example :

" All men are mortal.
Socrates is a man.
Then Socrates is mortal" ,

These cannot be expressed in propositional logic as a finite and logically valid argument (formula).

We need languages : that allow us to describe properties (*predicates*) of objects, or a relationship among objects represented by the variables .

Predicate logic satisfies the requirements of a language.

- *Predicate logic* is powerful enough for expression and reasoning.
- *Predicate logic* is built upon the ideas of *propositional logic.*

- **Predicate :**

Every complete "sentence" contains two parts : a "subject" and a "predicate".

The *subject* is what (or whom) the sentence is about.

The *predicate* tells something about the subject;

Example :

A *sentence* **"Judy {runs}".**

The subject is **Judy** and the predicate is **runs .**

Predicate, always includes verb, tells something about the subject.

Predicate is a verb phrase template that describes a property of objects, or a relation among objects represented by the variables.

Example:

"The car Tom is driving *is blue*" ;

"The sky *is blue*" ;

"The cover of this book *is blue*"

Predicate is **"is blue"** , describes property.

Predicates are given names; Let **'B'** is name for predicate **"is_blue"**.

Sentence is represented as **"B(x)"** , read as **"x is blue"**;

Symbol **"x"** represents an arbitrary Object .

- **Predicate Logic Expressions :**

  The propositional operators combine predicates, like

  If (p(....) && (!q(....) || r (....)))

  Logic operators :

  Examples of disjunction (**OR**) and conjunction (**AND**).

  Consider the expression with the respective logic symbols **||** and **&&**

  x < y || (y < z && z < x)

  which is **true || (true && true)** ;

  Applying truth table, found **True**

  Assignment for **<** are **3, 2, 1** for **x, y, z** and then

  the value can be *FALSE* or *TRUE*

  3 < 2 || (2 < 1 && 1 < 3)

  It is **False**

■ Predicate Logic Quantifiers

As said before, **x > 1** is not proposition and why ?

Also said, that for **x > 1** to be a proposition what is required ?

Generally, a predicate with variables (is called atomic formula) that can be made a proposition by applying one of the following two operations to each of its variables :

1. Assign a value to the variable; e.g., **x > 1**, if **3** is assigned to **x** becomes **3 > 1** , and it then becomes a true statement, hence a proposition.
2. Quantify the variable using a quantifier on formulas of predicate logic (called **wff** well-formed formula), such as **x > 1** or **P(x)**, by using Quantifiers on variables.

Apply Quantifiers on Variables

‡ **Variable** **x**

- **x > 5** is not a proposition, its truth depends upon the value of variable **x**
- to reason such statements, **x** need to be declared

‡ **Declaration** **x : a**

- **x : a** declares variable **x**
- **x : a** read as **"x is an element of set a"**

‡ **Statement** **p** is a statement about **x**

- **Q $\underbrace{\textbf{x : a}}$ • p** is quantification of statement

  statement

  declaration of variable **x** as element of set a

  quantifier

- Quantifiers are two types :

  universal quantifiers , denoted by symbol $\forall$ and

  existential quantifiers , denoted by symbol $\exists$

Note : The next few slide tells more on these two Quantifiers.

- **Universe of Discourse**

  The universe of discourse, also called domain of discourse or universe.

  This indicates :

  - a *set of entities* that the quantifiers deal.
  - *entities* can be set of real numbers, set of integers, set of all cars on a parking lot, the set of all students in a classroom etc.
  - *universe* is thus the domain of the (individual) variables.
  - *propositions* in the predicate logic are statements on objects of a universe.

  The universe is often left implicit in practice, but it should be obvious from the context.

  Examples:

  - About "natural numbers" **forAll x, y (x < y or x = y or x > y)**, there is no need to be more precise and say **forAll x, y in N**, because **N** is implicit, being the universe of discourse.
  - About a property that holds for natural numbers but not for real numbers, it is necessary to qualify what the allowable values of **x** and **y** are.

- **Apply Universal Quantifier ∀ " For All "**

  Universal Quantification allows us to make a statement about a collection of objects.

  ‡ Universal quantification: **∀ x : a • p**

  * read " for all **x** in **a** , **p** holds "
  * **a** is universe of discourse
  * **x** is a member of the domain of discourse.
  * **p** is a statement about **x**

  ‡ In propositional form it is written as : **∀x P(x)**

  * read " for all **x, P(x)** holds "
    " for each **x, P(x)** holds " or
    " for every **x, P(x)** holds "
  * where **P(x)** is predicate,
    ∀x means all the objects **x** in the universe
    P(x) is true for every object **x** in the universe

  ‡ Example : English language to Propositional form

  * **"All cars have wheels"**
    ∀x : car • x has wheel
  * **x P(x)**

    where **P (x)** is predicate tells : **'x has wheels'**

    x is variable for object **'cars'** that populate universe of discourse

- **Apply Existential Quantifier ∃** " There Exists "

  Existential Quantification allows us to state that an object does exist without naming it.

  ‡ Existential quantification: **∃ x : a • p**

  * read " there exists an **x** such that **p** holds "
  * **a** is universe of discourse
  * **x** is a member of the domain of discourse.
  * **p** is a statement about **x**

  ‡ In propositional form it is written as : **∃ x P(x)**

  * read " there exists an **x** such that **P(x)** " or
    " there exists at least one **x** such that **P(x)** "
  * Where **P(x)** is predicate
    ∃ x means at least one object **x** in the universe
    P(x) is true for least one object **x** in the universe

  ‡ Example : English language to Propositional form

  * **" Someone loves you "**

    ∃ x : Someone • x loves you

  * **x P(x)**

    where **P(x)** is predicate tells : **' x loves you '**
    x is variable for object **' someone '** that populate universe of discourse

■ Formula

In mathematical logic, a formula is a type of abstract object.

A token of a formula is a symbol or string of symbols which may be interpreted as any meaningful unit in a formal language.

‡ Terms

Defined recursively as variables, or constants, or functions like **$f(t_1, \ldots, t_n)$**, where **f** is an n-ary function symbol, and **$t_1, \ldots, t_n$** are *terms*. Applying predicates to terms produces *atomic formulas*.

‡ Atomic formulas

An atomic formula (or simply atom) is a formula with no deeper propositional structure, i.e., a formula that contains no logical connectives or a formula that has no strict sub-formulas.

- **Atoms** are thus the simplest well-formed formulas of the logic.
- **Compound formulas** are formed by combining the atomic formulas using the logical connectives.
- **Well-formed formula** ("**wiff**") is a symbol or string of symbols (a formula) generated by the formal grammar of a formal language.

An atomic formula is one of the form :

- **$t_1 = t_2$,** where **t_1** and **t_2** are terms, or
- **$R(t_1, \ldots, t_n)$,** where **R** is an n-ary relation symbol, and **$t_1, \ldots, t_n$** are terms.
- **$\neg a$** is a formula when **a** is a formula.
- **$(a \wedge b)$** and **$(a \vee b)$** are formula when **a** and **b** are formula

‡ Compound formula : example

$$((((a \wedge b) \wedge c) \vee ((\neg a \wedge b) \wedge c)) \vee ((a \wedge \neg b) \wedge c))$$

2.2 Representing " IsA " and " Instance " Relationships

Logic statements, containing *subject*, *predicate*, and *object*, were explained. Also stated, two important attributes "*instance*" and *"isa",* in a hierarchical structure (Ref. Fig. Inheritable KR).

Attributes " **IsA** " and " **Instance** " support property inheritance and play important role in knowledge representation.

The ways these two attributes "instance" and "isa", are logically expressed are shown in the example below :

- **Example :** A simple sentence like **"Joe is a musician"**
  - Here **"is a"** (called **IsA**) is a way of expressing what logically is called a *class-instance relationship* between the subjects represented by the terms "**Joe**" and "**musician**".
  - "**Joe**" is an *instance of the class of things* called "**musician**". "**Joe**" plays the role of *instance*, "**musician**" plays the role of *class* in that sentence.
  - Note : In such a sentence, while for a human there is no confusion, but for computers each relationship have to be defined explicitly. This is specified as: **[Joe]** **IsA** **[Musician]**

    i.e., **[Instance]** **IsA** **[Class]**

2.3 Computable Functions and Predicates

The objective is to define class of functions **C** computable in terms of **F.**

This is expressed as **C { F }** is explained below using two examples :
(1) "evaluate factorial n" and (2) "expression for triangular functions".

- **Example 1 :** A conditional expression to define ***factorial n*** ie **n!**
  - Expression

    " if p_1 then e_1 else if p_2 then e_2 . . . else if p_n then e_n".

    ie. (p1 → e1, p2 → e_2, ... pn → en)

    Here p_1, p_2, ... p_n are propositional expressions taking the values **T** or **F** for true and false respectively.
  - The value of **($p_1 \rightarrow e_1$, $p_2 \rightarrow e_2$, ...$p_n \rightarrow e_n$)** is the value of the ***e*** corresponding to the first **p** that has value **T**.
  - The expressions defining **n! , n= 5,** recursively are :

    $$n! = n \times (n-1)! \text{ for } n \geq 1$$
    $$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$
    $$0! = 1$$

    The above definition incorporates an instance that :
    if the product of no numbers ie **0! = 1** ,
    then only, recursive relation **(n + 1)! = (n+1) x n!** works for **n = 0**
  - Use of the above conditional expressions to define functions **n!** recursively is $n! = (n = 0 \rightarrow 1,\ n \neq 0 \rightarrow n \,.\, (n - 1)!)$
  - **Example:** Evaluate **2!** according to above definition.

    $$\begin{aligned} 2! &= (2 = 0 \rightarrow 1,\ 2 \neq 0 \rightarrow 2 \,.\, (2 - 1)!) \\ &= 2 \times 1! \\ &= 2 \times (1 = 0 \rightarrow 1,\ 1 \neq 0 \rightarrow 1 \,.\, (1 - 1)!) \\ &= 2 \times 1 \times 0! \\ &= 2 \times 1 \times (0 = 0 \rightarrow 1,\ 0 \neq 0 \rightarrow 0 \,.\, (0 - 1)!) \\ &= 2 \times 1 \times 1 \\ &= 2 \end{aligned}$$

■ **Example 2 :** A conditional expression for triangular functions

◇ The graph of a well known triangular function is shown below

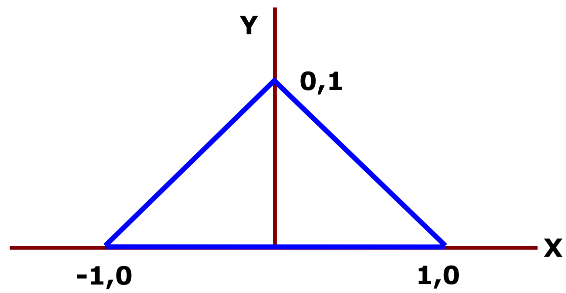


Fig. A Triangular Function

the conditional expressions for triangular functions are

$$|x| = (x < 0 \rightarrow -x \,, \quad x \geq 0 \rightarrow x)$$

◇ the triangular function of the above graph is represented by the conditional expression

$$\text{tri}\ (x) = (x \leq -1 \rightarrow 0, \ x \leq 0 \rightarrow -x, \ x \leq 1 \rightarrow x, \ x > 1 \rightarrow 0)$$

2.4 Resolution

Resolution is a procedure used in proving that arguments which are expressible in predicate logic are correct.

Resolution is a procedure that produces proofs by refutation or contradiction.

Resolution lead to refute a theorem-proving technique for sentences in propositional logic and first-order logic.

- Resolution is a rule of inference.
- Resolution is a computerized theorem prover.
- Resolution is so far only defined for Propositional Logic. The strategy is that the Resolution techniques of Propositional logic be adopted in Predicate Logic.

3. KR Using Rules

In the earlier slides, the Knowledge representations using predicate logic have been illustrated. The other popular approaches to Knowledge representation are called *production rules* , *semantic net* and *frames*.

Production rules, sometimes called **IF-THEN** rules are most popular KR.

- production rules are simple but powerful forms of KR.
- production rules provide the flexibility of combining declarative and procedural representation for using them in a unified form.

Examples of production rules :

- IF condition THEN action
- IF premise THEN conclusion
- IF proposition p_1 and proposition p_2 are true THEN proposition p_3 is true

Advantages of production rules :

- they are modular,
- each rule define a small and independent piece of knowledge.
- new rules may be added and old ones deleted
- rules are usually independently of other rules.

The production rules as knowledge representation mechanism are used in the design of many **"Rule-based systems"** also called **"Production systems"** .

- **Types of Rules**

  Three types of rules are mostly used in the Rule-based production systems.

  - **Knowledge Declarative Rules :**

    These rules state all the facts and relationships about a problem.

    Example :

    IF inflation rate declines

    THEN the price of gold goes down.

    These rules are a part of the knowledge base.

  - **Inference Procedural Rules**

    These rules advise on how to solve a problem, while certain facts are known.

    Example :

    IF the data needed is not in the system

    THEN request it from the user.

    These rules are part of the inference engine.

  - **Meta rules**

    These are rules for making rules. Meta-rules reason about which rules should be considered for firing.

    Example :

    IF the rules which do not mention the current goal in their premise,

    AND there are rules which do mention the current goal in their premise,

    THEN the former rule should be used in preference to the latter.

    – Meta-rules direct reasoning rather than actually performing reasoning.

    – Meta-rules specify which rules should be considered and in which order they should be invoked.

3.1 Procedural versus Declarative Knowledge

These two types of knowledge were defined in earlier slides.

- **Procedural Knowledge** : **knowing 'how to do'**

  Includes : *rules, strategies, agendas, procedures, models*.

  These explains *what to do* in order to reach a certain conclusion.

  Example

  Rule: To determine if Peter or Robert is older, first find their ages.

  It is knowledge about *'how to do'* something. It manifests itself in the doing of something, e.g., manual or mental skills cannot reduce to words. It is held by individuals in a way which does not allow it to be communicated directly to other individuals.

  Accepts a description of the steps of a task or procedure. It Looks similar to declarative knowledge, except that tasks or methods are being described instead of facts or things.

- **Declarative Knowledge** : **knowing 'what', knowing 'that'**

  Includes : *concepts, objects, facts, propositions, assertions, models.*

  It is knowledge about *facts* and *relationships, that*

  – can be expressed in simple and clear statements,

  – can be added and modified without difficulty.

  Examples : A car has four tyres; Peter is older than Robert.

  Declarative knowledge and explicit knowledge are articulated knowledge and may be treated as synonyms for most practical purposes.

  Declarative knowledge is represented in a format that can be manipulated, decomposed and analyzed independent of its content.

■ **Comparison :**

Comparison between Procedural and Declarative Knowledge :

Procedural Knowledge	**Declarative Knowledge**
• Hard to debug	• Easy to validate
• Black box	• White box
• Obscure	• Explicit
• Process oriented	• Data - oriented
• Extension may effect stability	• Extension is easy
• Fast , direct execution	• Slow (requires interpretation)
• Simple data type can be used	• May require high level data type
• Representations in the form of sets of rules, organized into routines and subroutines.	• Representations in the form of production system, the entire set of rules for executing the task.

- **Comparison :**

Comparison between Procedural and Declarative Language :

Procedural Language	Declarative Language
• Basic, C++, Cobol, etc.	• SQL
• Most work is done by interpreter of the languages	• Most work done by Data Engine within the DBMS
• For one task many lines of code	• For one task one SQL statement
• Programmer must be skilled in translating the objective into lines of procedural code	• Programmer must be skilled in clearly stating the objective as a SQL statement
• Requires minimum of management around the actual data	• Relies on SQL-enabled DBMS to hold the data and execute the SQL statement .
• Programmer understands and has access to each step of the code	• Programmer has no interaction with the execution of the SQL statement
• Data exposed to programmer during execution of the code	• Programmer receives data at end as an entire set
• More susceptible to failure due to changes in the data structure	• More resistant to changes in the data structure
• Traditionally faster, but that is changing	• Originally slower, but now setting speed records
• Code of procedure tightly linked to front end	• Same SQL statements will work with most front ends Code loosely linked to front end.
• Code tightly integrated with structure of the data store	• Code loosely linked to structure of data; DBMS handles structural issues
• Programmer works with a pointer or cursor	• Programmer not concerned with positioning
• Knowledge of coding tricks applies only to one language	• Knowledge of SQL tricks applies to any language using SQL

3.2 Logic Programming

Logic programming offers a formalism for specifying a computation in terms of logical relations between entities.

- logic program is a collection of logic statements.
- programmer describes all relevant logical relationships between the various entities.
- computation determines whether or not, a particular conclusion follows from those logical statements.

- **Characteristics of Logic program**

  Logic program is characterized by set of relations and inferences.

  - program consists of a set of axioms and a goal statement.
  - rules of inference determine whether the axioms are sufficient to ensure the truth of the goal statement.
  - execution of a logic program corresponds to the construction of a proof of the goal statement from the axioms.
  - programmer specify basic logical relationships, does not specify the manner in which inference rules are applied.

    Thus **Logic + Control = Algorithms**

- **Examples of Logic Statements**

  - Statement

    A grand-parent is a parent of a parent.
  - Statement expressed in more closely related logic terms as

    A person is a grand-parent if she/he has a child and that child is a parent.
  - Statement expressed in first order logic as

    (for all) x: grandparent (x, y) :- parent (x, z), parent (z, y)

    read as x is the grandparent of y
    if x is a parent of z and z is a parent of y

- **Logic Programming Language**

A programming language includes :

- the *syntax*
- the *semantics* of programs and
- the *computational model*.

There are many ways of organizing computations. The most familiar paradigm is **procedural**. The program specifies a computation by saying *"how"* it is to be performed. *FORTRAN, C,* and *Object-oriented languages* fall under this general approach.

Another paradigm is **declarative.** The program specifies a computation by giving the properties of a correct answer. *Prolog and logic data language (LDL)* are examples of declarative languages, emphasize the logical properties of a computation.

Prolog and LDL are called logic programming languages.

PROLOG (PROgramming LOGic) is the most popular Logic programming language rose within the realm of Artificial Intelligence (AI). It became popular with AI researchers, who know more about "what" and "how" intelligent behavior is achieved.

- **Syntax and Terminology** (relevant to Prolog programs)

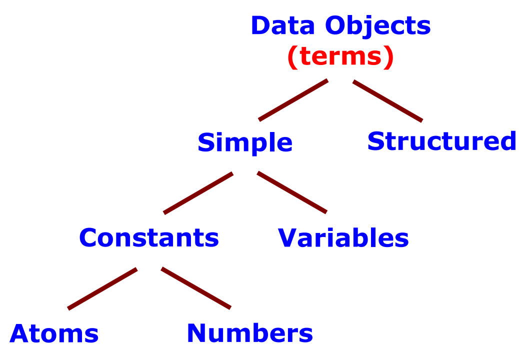
  In any language, the formation of components (expressions, statements, etc.), is guided by syntactic rules.

  The components are divided into two parts:

  (A) data components and (B) program components.

(A) Data components :

Data components are collection of data objects that follow hierarchy.



Data object of any kind is also called a *term.* A term is a constant, a variable or a compound term.

Simple data object is not decomposable; e.g. *atoms, numbers, constants, variables*.

Syntax distinguishes the data objects, hence no need for declaring them.

Structured data object are made of several components.

All these data components are explained in next slide.

(a) Data Objects :

The data objects of any kind is called a *term.*

- **Term** : Examples

  ‡ **Constants:**

  Denote elements such as *integers, floating point, atoms*.

  ‡ **Variables:**

  Denote a single but unspecified element; *symbols* for variables begin with an uppercase letter or an underscore.

  ‡ **Compound terms:**

  Comprise a *functor* and sequence of one or more compound terms called *arguments*.

  ► **Functor**: is characterized by its name and number of arguments; name is an ***atom***, and number of arguments is ***arity***.

$$f/n = f(t_1, t_2, \ldots t_n)$$

where f is name of the *functor* and is of *arity* **n**

t_i's are the *argument*

f/n denotes *functor* f of *arity* **n**

Functors with same name but different arities are distinct.

  ‡ **Ground and non-ground**:

  Terms are ground if they contain no variables (only constant signs); otherwise they are non-ground.

  Goals are atoms or compound terms, and are generally non-ground.

(b) Simple Data Objects : *Atoms, Numbers, Variables*

- **Atoms**
  - a lower-case letter, possibly followed by other letters of either case, digits, and underscore character.

    e.g. **a** **greaterThan** **two_B_or_not_2_b**
  - a string of special characters such as: + - * / \ = ^ < > : ~ # $ &

    e.g. **<>** **##&&** **::=**
  - a *string* of any characters enclosed within single quotes.

    e.g. **'ABC'** **'1234'** **'a<>b'**
  - following are also *atoms* **!** **;** **[]** **{}**

- **Numbers**
  - applications involving heavy numerical calculations are rarely written in Prolog.
  - *integer* representation: **e.g. 0 -16 33 +100**
  - *real numbers* written in standard or scientific notation,

    e.g. **0.5 -3.1416 6.23e+23 11.0e-3 -2.6e-2**

- **Variables**
  - begins by a capital letter, possibly followed by other letters of either case, digits, and underscore character.

    e.g. **X25** **List** **Noun_Phrase**

(c) Structured Data Objects : *General Structures , Special Structures*

- **General Structures**
  - ‡ a structured term is syntactically formed by a functor and a list of arguments.
  - ‡ functor is an atom.
  - ‡ list of arguments appear between parentheses.
  - ‡ arguments are separated by a comma.
  - ‡ each argument is a term (i.e., any Prolog data object).
  - ‡ the number of arguments of a structured term is called its *arity.*
  - ‡ e.g. **greaterThan(9, 6)** **f(a, g(b, c), h(d))** **plus(2, 3, 5)**

  Note : a structure in Prolog is a mechanism for combining terms together, like integers 2, 3, 5 are combined with the functor *plus*.

- **Special Structures**
  - ‡ In Prolog an ordered collection of terms is called a *list* .
  - ‡ Lists are structured terms and Prolog offers a convenient notation to represent them:
    - * Empty list is denoted by the atom **[ ]**.
    - * Non-empty list carries element(s) between square brackets, separating elements by comma.

      e.g. **[bach, bee]** **[apples, oranges, grapes]**

(B) Program Components

A Prolog program is a collection of predicates or rules.

A predicate establishes a relationships between objects.

(a) Clause, Predicate, Sentence, Subject

- Clause is a collection of grammatically-related words .
- Predicate is composed of one or more clauses.
- Clauses are the building blocks of sentences; every sentence contains one or more clauses.
- A Complete Sentence has two parts: subject and predicate.
  - subject is what (or whom) the sentence is about.
  - predicate tells something about the subject.
- Example 1 : **"cows eat grass"**.
  It is a clause, because it contains
  the **subject "cows"** and
  the **predicate "eat grass."**
- Example 2 : **"cows eating grass are visible from highway"**
  This is a complete clause.
  the **subject "cows eating grass"** and
  the **predicate "are visible from the highway"** makes complete thought.

(b) Predicates & Clause

Syntactically a predicate is composed of one or more clauses.

- The general form of clauses is

  <left-hand-side> :- <right-hand-side>.

  where LHS is a single goal called *"goal"* and RHS is composed of one or more goals, separated by commas, called *"sub-goals"* of the goal on left-hand side.

  The symbol **" :- "** is pronounced as "it is the case" or "such that"

- The structure of a clause in logic program

$$\underbrace{\overbrace{\textbf{pred (functor(var1, var2))}}^{\textbf{head}} \quad \textbf{:-} \quad \overbrace{\underbrace{\textbf{pred(var1)}}_{\textbf{literal}}\textbf{ , } \underbrace{\textbf{pred(var2)}}_{\textbf{literal}}}^{\textbf{body}}}_{\textbf{clause}}$$

  Literals represent the possible choices in primitive types the particular language. Some of the choices of types of literals are often integers, floating point, Booleans and character strings.

- Example : **grand_parent (X, Z) :- parent(X, Y), parent(Y, Z).**
  parent (X, Y) :- mother(X, Y).
  parent (X, Y) :- father(X, Y).

  Read as if x is mother of y then x is parent of y

[Continued in next slide]

[Continued from previous slide]

‡ Interpretation:

* A clause specifies the conditional truth of the goal on the LHS; goal on LHS is assumed to be true if the sub-goals on RHS are all true. A predicate is true if at least one of its clauses is true.

* An **individual "X"** is the **grand-parent of "Z"** if a **parent** of that same **"X"** is **"Y"** and **"Y"** is the **parent** of that **"Z".**

$$\begin{tikzcd} X \arrow[r, "\text{(X is parent of Y)}"] \arrow[rr, bend right, "\text{(X is grand parent of Z)}"'] & Y \arrow[r, "\text{(Y is parent of Z)}"] & Z \end{tikzcd}$$

* An **individual "X"** is a **parent of "Y"** if **"Y" is the mother of "X"**

$$\begin{tikzcd} X \arrow[r, shift left, "\text{(X is parent of Y)}"] \arrow[r, shift right, "\text{(X is mother of Y)}"'] & Y \end{tikzcd}$$

* An **individual "X"** is a **parent of "Y"** if **"Y" is the father of "X"**.

$$\begin{tikzcd} X \arrow[r, shift left, "\text{(X is parent of Y)}"] \arrow[r, shift right, "\text{(X is father of Y)}"'] & Y \end{tikzcd}$$

(c) Unit Clause - a special Case

Unlike the previous example of conditional truth, one often encounters unconditional relationships that hold.

- In Prolog the clauses that are unconditionally true are called *unit clause* or *fact* .
- Example : Unconditionally relationships say

  'X' is the father of 'Y' is unconditionally true.

  This relationship as a *Prolog clause* is

  father(X, Y) :- true.

  Interpreted as relationship of father between **X** and **Y** is always true; or simply stated as **X** is father of **Y** .
- Goal true is built-in in Prolog and always holds.
- Prolog offers a simpler syntax to express unit clause or fact

  father(X, Y)

  ie the " **:- true** " part is simply omitted.

(d) Queries

In Prolog the queries are statements called directive.

A special case of directives, are called queries.

‡ Syntactically, directives are clauses with an empty left-hand side.

Example : **? - grandparent(Q, Z).**

This query **Q** is interpreted as : **Who is a grandparent of Z ?**

By issuing queries **Q**, Prolog tries to establish the validity of specific relationships.

The answer from previous slides is **(X is grand parent of Z)**

‡ The result of executing a query is either success or failure

Success, means the goals specified in the query holds according to the facts and rules of the program.

Failure, means the goals specified in the query does not hold according to the facts and rules of the program.

- **Programming Paradigms :** Models of Computation

A complete description of a programming language includes the *computational model, syntax, semantics, and pragmatic considerations* that shape the language.

Models of Computation :

A computational model is a collection of values and operations, while computation is the application of a sequence of operations to a value to yield another value.

There are three basic computational models :
(a) Imperative, (b) Functional, and (c) Logic.

In addition to these, there are two programming paradigms :
(a) concurrent (b) object-oriented programming .
While, these two are not models of computation, but they rank in importance with computational models.

(a) Imperative Model

The Imperative model of computation, consists of a state and an operation of assignment which is used to modify the state.

- Programs consist of sequences of commands.
- Computations are changes in the state.

Example : Linear function

A linear function **y = 2x + 3** can be written as

$$Y := 2 * X + 3$$

The implementation requires to determines the value of **X** in the state and then creates a new state which differs from the old state.

New State: X = 3, Y = 9,

The imperative model is closest to the hardware model on which programs are executed, that makes it most efficient model in terms of execution time.

(b) Functional model

The Functional model of computation, consists of a set of values, functions, and the operation of functions. The functions may be named and composed with other functions. It can take other functions as arguments and return results.

- Programs consist of definitions of functions.
- Computations are application of functions to values.

‡ Example 1 : Linear function

A linear function y = 2x + 3 can be defined as :

f (x) = 2 ∗ x + 3

‡ Example 2 : Determine a value for Circumference.

Assign a value to Radius, that determines a value for Circumference.

Circumference = 2 × pi × radius , where pi = 3.14

Generalize Circumference with the variable "radius" ie

Circumference(radius) = 2 × pi × radius , where pi = 3.14

Functional models are developed over many years. The notations and methods form the base upon which problem solving methodologies rest.

(c) Logic Model

The logic model of computation is based on relations and logical inference.

- Programs consist of definitions of relations.
- Computations are inferences (is a proof).

‡ Example 1 **:** Linear function

A linear function **y = 2x + 3** can be represented as :

f (X , Y) if Y is 2 ∗ X + 3.

Here the function represents the relation between **X** and **Y**.

‡ Example 2: Determine a value for Circumference.

The circumference computation can be represented as:

Circle (R , C) if Pi = 3.14 and **C = 2 ∗ pi ∗ R.**

Here the function is represented as the relation between radius **R** and circumference **C**.

‡ Example 3: Determine the mortality of Socrates and Penelope.

The program is to determine the mortality of Socrates and Penelope.

The fact given that Socrates and Penelope are human.

The rule is that all humans are mortal, that is

for all X, if X is human then X is mortal.

To determine the mortality of Socrates or Penelope, make the assumption that there are no mortals, that is **¬ mortal (Y)**

[logic model continued in the next slide]

[logic model continued in the previous slide]

- The equivalent form of the facts and rules stated before are

  human (Socrates)

  mortal (X) if human (X)

- To determine the mortality of Socrates and Penelope, we made the assumption that there are no mortals i.e. **¬ mortal (Y)**

- Computation (proof) that Socrates is mortal

1. (a)	**human(Socrates)**	**Fact**
2.	**mortal(X) if human(X)**	**Rule**
3	**¬mortal(Y)**	**assumption**
4.(a)	**X = Y**	
4.(b)	**¬human(Y)**	**from 2 & 3 by unification and modus tollens**
5.	**Y = Socrates**	**from 1 and 4 by unification**
6.	**Contradiction**	**5, 4b, and 1**

- Explanation :
  - The 1st line is the statement "Socrates is a man."
  - The 2nd line is a phrase "all human are mortal" into the equivalent "for all X, if X is a man then X is mortal".
  - The 3rd line is added to the set to determine the mortality of Socrates.
  - The 4th line is the deduction from lines 2 and 3. It is justified by the inference rule modus tollens which states that if the conclusion of a rule is known to be false, then so is the hypothesis.
  - Variables X and Y are unified because they have same value.
  - By unification, Lines 5, 4b, and 1 produce contradictions and identify Socrates as mortal.
  - Note that, resolution is an inference rule which looks for a contradiction and it is facilitated by unification which determines if there is a substitution which makes two terms the same.

Logic model formalizes the reasoning process. It is related to relational data bases and expert systems.

3.3 Forward versus Backward Reasoning

Rule-Based system architecture consists *a set of rules*, *a set of facts*, and an *inference engine*. The need is to find what new facts can be derived.

Given a set of rules, there are essentially two ways to generate new knowledge: one, forward chaining and the other, backward chaining.

- **Forward chaining** : also called data driven.
  It starts with the facts, and sees what rules apply.

- **Backward chaining** : also called goal driven.
  It starts with something to find out, and looks for rules that will help in answering it.

■ **Example 1**

Rule R1 :	IF hot AND smoky	THEN fire
Rule R2 :	IF alarm_beeps	THEN smoky
Rule R3 :	IF fire	THEN switch_on_sprinklers
Fact F1 :	alarm_beeps	[Given]
Fact F2 :	hot	[Given]

■ **Example 2**

Rule R1 :	IF hot AND smoky	THEN ADD fire
Rule R2 :	IF alarm_beeps	THEN ADD smoky
Rule R3 :	IF fire	THEN ADD switch_on_sprinklers
Fact F1 :	alarm_beeps	[Given]
Fact F2 :	hot	[Given]

■ **Example 3 : A typical Forward Chaining**

Rule R1 :	IF hot AND smoky	THEN ADD fire	
Rule R2 :	IF alarm_beeps	THEN ADD smoky	
Rule R3 :	If fire	THEN ADD switch_on_sprinklers	
Fact F1 :	alarm_beeps	[Given]	
Fact F2 :	hot	[Given]	
Fact F4 :	smoky	[from F1 by R2]	
Fact F2 :	fire	[from F2, F4 by R1]	
Fact F6 :	switch_on_sprinklers	[from F2 by R3]	

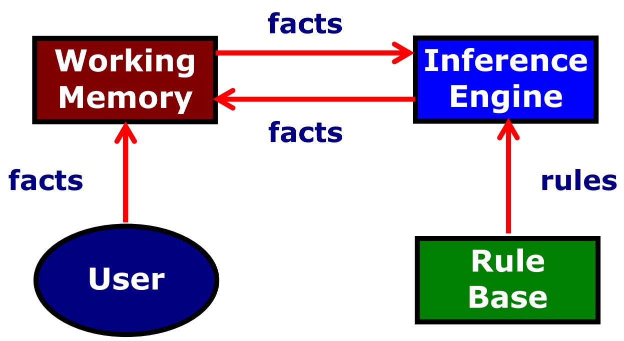
■ **Example 4 : A typical Backward Chaining**

Rule R1 :	IF hot AND smoky	THEN fire
Rule R2 :	IF alarm_beeps	THEN smoky
Rule R3 :	If _fire	THEN switch_on_sprinklers
Fact F1 :	hot	[Given]
Fact F2 :	alarm_beeps	[Given]
Goal :	Should I switch sprinklers on?	

- **Forward Chaining**

The Forward chaining system, properties , algorithms, and conflict resolution strategy are illustrated.

■ **Forward chaining system**



‡ facts are held in a working memory

‡ condition-action rules represent actions to be taken when specified facts occur in working memory.

‡ typically, actions involve adding or deleting facts from the working memory.

■ **Properties of Forward Chaining**

‡ all rules which can fire do fire.

‡ can be inefficient - lead to spurious rules firing, unfocused problem solving

‡ set of rules that can fire known as conflict set.

‡ decision about which rule to fire is conflict resolution.

■ Forward chaining algorithm - I

Repeat

- ‡ Collect the rule whose condition matches a fact in WM.
- ‡ Do actions indicated by the rule.
  (add facts to WM or delete facts from WM)

Until problem is solved or no condition match

Apply on the Example 2 extended (adding 2 more rules and 1 fact)

Rule R1 :	IF hot AND smoky	THEN ADD fire	
Rule R2 :	IF alarm_beeps	THEN ADD smoky	
Rule R3 :	If fire	THEN ADD switch_on_sprinklers	
Rule R4 :	IF dry	THEN ADD switch_on_humidifier	
Rule R5 :	IF sprinklers_on	THEN DELETE dry	
Fact F1 :	alarm_beeps	[Given]	
Fact F2 :	hot	[Given]	
Fact F2 :	Dry	[Given]	

Now, two rules can fire (R2 and R4)

Rule R4	ADD humidifier is on	[from F2]
Rule R2	ADD smoky	[from F1]
[followed by sequence of actions]	ADD fire	[from F2 by R1]
	ADD switch_on_sprinklers	[by R3]
	DELEATE dry, ie humidifier is off **a conflict !**	[by R5]

■ Forward chaining algorithm - II (applied to example 2 above)

Repeat

- ‡ Collect the rules whose conditions match facts in WM.
- ‡ If more than one rule matches as stated above then
  - ◇ Use *conflict resolution strategy* to eliminate all but one
- ‡ Do actions indicated by the rules
  (add facts to WM or delete facts from WM)

Until problem is solved or no condition match

Conflict Resolution Strategy

Conflict set is the set of rules that have their conditions satisfied by working memory elements.

Conflict resolution normally selects a single rule to fire.

The popular conflict resolution mechanisms are :

Refractory, Recency, Specificity.

- **Refractory**
  - a rule should not be allowed to fire more than once on the same data.
  - discard executed rules from the conflict set.
  - prevents undesired loops.

- **Recency**
  - rank instantiations in terms of the recency of the elements in the premise of the rule.
  - rules which use more recent data are preferred.
  - working memory elements are time-tagged indicating at what cycle each fact was added to working memory.

- **Specificity**
  - rules which have a greater number of conditions and are therefore more difficult to satisfy, are preferred to more general rules with fewer conditions.
  - more specific rules are 'better' because they take more of the data into account.

- **Alternative to Conflict Resolution –** Use Meta Knowledge

  Instead of conflict resolution strategies, sometimes we want to use knowledge in deciding which rules to fire. **Meta-rules** reason about which rules should be considered for firing. They direct reasoning rather than actually performing reasoning.

  - Meta-knowledge : knowledge about knowledge to guide search.
  - Example of meta-knowledge

    IF conflict set contains any rule (c , a) such that
    a = "animal is mammal''
    THEN fire (c , a)

  - This example says meta-knowledge encodes knowledge about how to guide search for solution.
  - Meta-knowledge, explicitly coded in the form of rules with "object level" knowledge.

- **Backward Chaining**

Backward chaining system and the algorithm are illustrated.

- **Backward chaining system**

  ‡ Backward chaining means reasoning from goals back to facts. The idea is to focus on the search.

  ‡ Rules and facts are processed using backward chaining interpreter.

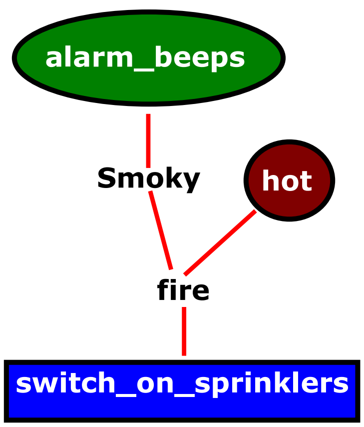
  ‡ Checks hypothesis, e.g. *"should I switch the sprinklers on?"*

- **Backward chaining algorithm**

  ‡ Prove goal **G**

  If **G** is in the initial facts , it is proven.

  Otherwise, find a rule which can be used to conclude **G**, and try to prove each of that rule's conditions.



Encoding of rules

Rule R1 :	IF hot AND smoky	THEN fire	
Rule R2 :	IF alarm_beeps	THEN smoky	
Rule R3 :	If fire	THEN switch_on_sprinklers	
Fact F1 :	hot	[Given]	
Fact F2 :	alarm_beeps	[Given]	
Goal :	Should I switch sprinklers on?		

- **Forward vs Backward Chaining**

  ‡ Depends on problem, and on properties of rule set.

  ‡ Backward chaining is likely to be better if there is clear hypotheses.
  Examples : Diagnostic problems or classification problems, Medical expert systems

  ‡ Forward chaining may be better if there is less clear hypothesis and want to see what can be concluded from current situation;
  Examples : Synthesis systems - design / configuration.

3.4 Control Knowledge

An algorithm consists of : logic component, that specifies the knowledge to be used in solving problems, and *control component*, that determines the problem-solving strategies by means of which that knowledge is used.

Thus **Algorithm = Logic + Control .**

The logic component determines the meaning of the algorithm whereas the control component only affects its efficiency.

An algorithm may be formulated in different ways, producing same behavior. One formulation, may have a clear statement in logic component but employ a sophisticated problem solving strategy in the control component. The other formulation, may have a complicated logic component but employ a simple problem-solving strategy.

The efficiency of an algorithm can often be improved by improving the control component without changing the logic of the algorithm and therefore without changing the meaning of the algorithm.

The trend in databases is towards the separation of logic and control. The programming languages today do not distinguish between them. The programmer specifies both logic and control in a single language. The execution mechanism exercises only the most rudimentary problem-solving capabilities.

Computer programs will be more often correct, more easily improved, and more readily adapted to new problems when programming languages separate logic and control, and when execution mechanisms provide more powerful problem-solving facilities of the kind provided by intelligent theorem-proving systems.

4. References : Textbooks

1. *"Artificial Intelligence", by Elaine Rich and Kevin Knight, (2006), McGraw Hill companies Inc., Chapter 4-6, page 105-192.*

2. *"Artificial Intelligence: A Modern Approach" by Stuart Russell and Peter Norvig, (2002), Prentice Hall, Chapter 7-10, page 194-369.*

3. *"Computational Intelligence: A Logical Approach", by David Poole, Alan Mackworth, and Randy Goebel, (1998), Oxford University Press, Chapter 5-7, page 169-278.*

4. *"Artificial Intelligence: Structures and Strategies for Complex Problem Solving", by George F. Luger, (2002), Addison-Wesley, Chapter 2, 7, page 35-77, 227-273.*

5. *"AI: A New Synthesis", by Nils J. Nilsson, (1998), Morgan Kaufmann Inc., Chapter 13, Page 217-229.*

6. *"Artificial Intelligence: Theory and Practice", by Thomas Dean, (1994), Addison-Wesley, Chapter 3, 6, Page 71-130, 255-296.*

7. *Related documents from open source, mainly internet. An exhaustive list is being prepared for inclusion at a later date.*