IMPORTANT INSTRUCTIONS

1. The following Lectures will not be repeated and would be part of mid-term & final exam as well.

2. Attendance of two lectures will be marked on the basis of timely submission of assignment

3. Submit your assignment on LMS on time. late submission will result as late attendance and deduction of marks

Week # 5 Lecture #9&10

KNOWLEDGE REPRESENTATION

RELATED VIDEOS ON YOUTUBE			
Sr.	**Channel**	**Video Name**	**URL**
1	Dr Umair Abdullah	(Solution of Assignment 1& QA) Solution Semantic Net Assignment	https://youtu.be/yhvYrTbMMuc https://youtu.be/d-_ZEIEeJdM
2	Dr Umair Abdullah	Production Rules : Introduction, Rule-based programming, Circle Area Example	https://youtu.be/NPrwSLeClZg
3	Dr Umair Abdullah	Calculating factorial in OPS-5	https://youtu.be/XBNN8H61azI
4	Dr Umair Abdullah	Other KR techniques	https://youtu.be/3Vr6W-q_C1Q

Learning Objectives

1. Questions Answers of last lectures, solution of assignment # 1
2. Representing knowledge as production rules
3. Rule-based programming vs. conventional programming
4. Circle area example
5. Tracing of factorial example in OPS-5 syntax.
6. Other KR techniques and reference material

Questions and Answers of Last Week

Logic:
1: (setf (get block-1 'color)' Red)

2: (color block-1 red)

which of them is okay to be followed for making logic? Because in our lecture Sir you have taught the 2 but in the YT video you discussed 1. Moreover, the colon used after the round-brackets is making me confused shouldn't it be before
closing the bracket I mean like this (setf (get block-1 'color') Red)??

ANSWER:
Very good question, use the format no 2. Format 1 is of last semester. You may ignore the colon, no need to get confused.

Question #2:
How much should we extend an entity like in the hands-out it is written that Pakistan is a country, although there are some other distinct attributes belong to Pakistan I mean how can we determine, that much information regarding any entity is enough for semantic network to be valid.
e.g. Pakistan is a country, Pakistan is also a Muslim country and a lot more distinct attributes belong to Pakistan. Such case can occur for every Entity...

ANS: you are very right. We may extend a network at some appropriate level (1 to 3 necessary links are enough)

Question 3: Terminated is also an action so, is it OKAY to draw such lines to represent it?
ANS: Yes absolutely ok. We just need to keep the network visually understandable

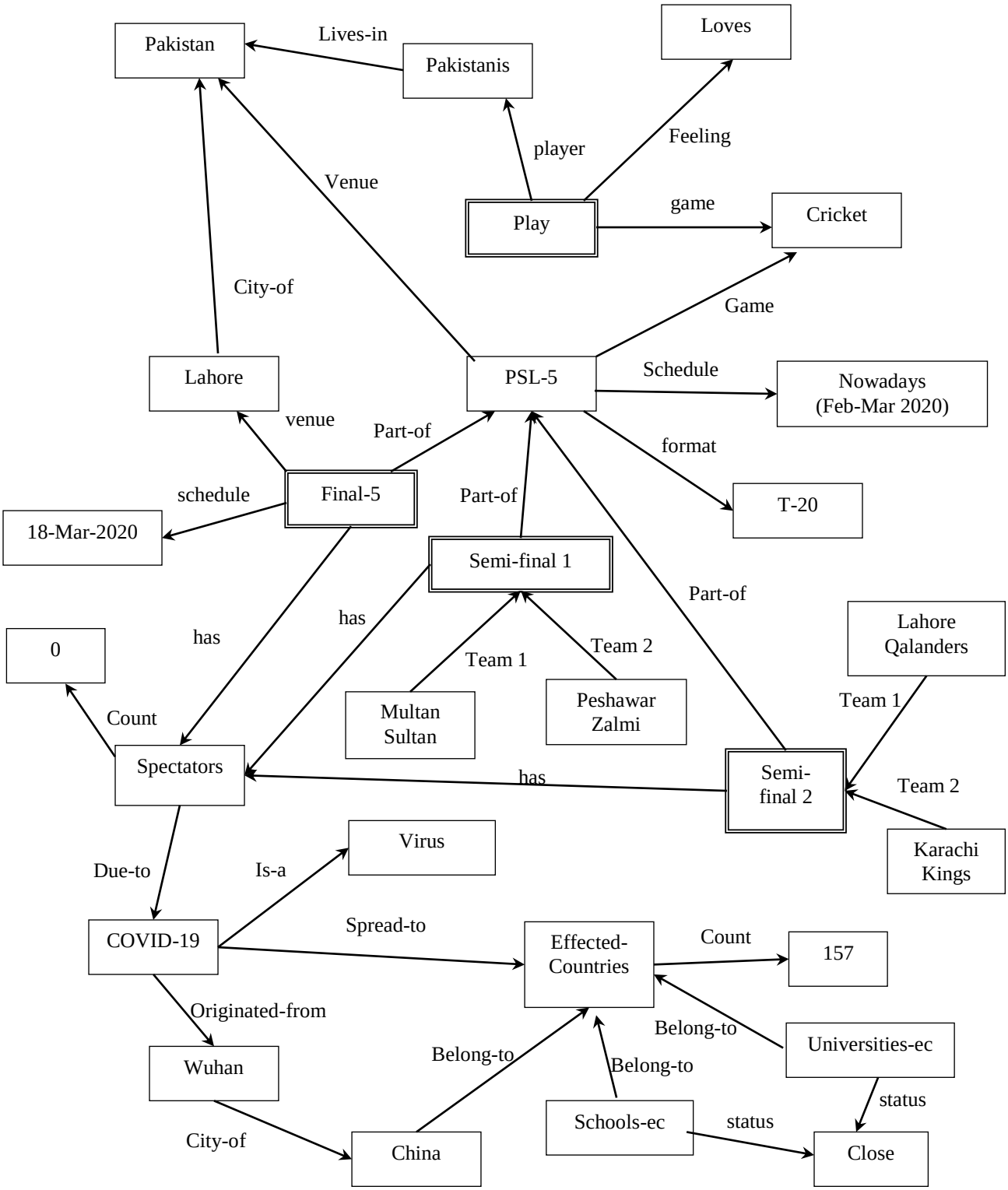
Guidelines toDraw Semantic Network

1. Avoid generic terms
2. Assign IDs
3. Elaborate the generic terms (e.g. today)
4. All links should be labeled
5. Two nodes and a link should be self-standing (i.e. understandable without other nodes)
6. Visually understandable, (avoid crossing of lines, avoid multiple nodes of same entity

SOLUTION (Assignment # 1)

DRAW THE SEMANTIC NETWORK OF THE FOLLOWING INFORMATION AND ALSO REPRESENT THE GIVEN KNOWLEDGE USING LOGIC

"Pakistanis like to play cricket. Nowadays PSL20 of T-20 matches are being played in Pakistan. Final of the PSL will be played in Lahore on 18 March 2020. First semi-final will be played between Multan Sultan and Peshawar Zalmi, while the second semi-final will be played between Karachi kings and Lahore Qalanders. Unfortunately, due to corona virus, these matches will be played without spectators. Corona virus (COVID-19) is a contagious disease, originated from Wuhan China and now has spread to 157 countries. Schools and universities have been closed in many countries due to this virus"



Representation in LOGIC

(lives_in Pakistanis Pakistan)
(player Play Pakistanis)
(feeling Play love)
(game Play cricket)
(venue PSL-5 Pakistan)
(game PSL-5 cricket)
(schedule PSL-5 Feb-Mar-2020)
(format PSL-5 T-20)
(part-of Final-5 PSL-5)
(part-of Semi-Final-1 PSL-5)
(part-of Semi-Final-2 PSL-5)
(schedule Final-5 18-Mar-2020)
(venue Final-5 Lahore)
(city-of Lahore Pakistan)
(has Final-5 spectators)
(has Semi-Final-1 spectators)
(has Semi-Final-2 spectators)
(count spectators 0)
(due-to spectators COVID-19)
(team-1 Semi-Final-1 Multan-Sultan)
(team-2 Semi-Final-1 Peshawar-Zalmi)
(team-1 Semi-Final-2 Lahore-Qalanders)
(team-2 Semi-Final-2 Karachi-Kings)
(belong-to schools-ec effected-countries)
(belong-to universities-ec effected-countries)
(status schools-ec closed)
(status universities-ec closed)
(count effected-countries 157)
(spread-to COVID-19 effected-countries)
(originated-from COVID-19 Wuhan)
(city-of Wuhan China)
(belong-to China effected-countries)
(Is-a COVID-19 virus)

Note: Sequence of logic statements does not matter. Similarly, these statements are not case-sensitive. Moreover, you can replace any of these symbols with more suitable (meaningful with respect to the real world)

Assignment #2

Dear students read the given lectures carefully as per the course objectives mentioned on the top and carryout the assignment as per following instructions

1. **Submission Date: Before next class**
2. **You must prepare handwritten solution of the assignment and submit on LMS only.**
3. **Attempt the Assignment after reading the lecture notes given below, watching the videos (mentioned above), and doing self-learning of the topic from web.**

Q1: Write collection of production rules to input a number from user and display its multiplication table (first ten rows). Also show the status of working memory by tracing the rules if user enters 5 as input.

Knowledge as Production Rules

Operational knowledge can be better described in the form of production rules, which has been used for developing rule-based expert systems. Programming with the help of production rules can be termed as rule based programming. Following section describes the difference of rule based programming and conventional programming;

Rule-Based Programming

This section introduces the underlying technology of developing rule-based system, which is a non-conventional software, although initiated in 1970s in healthcare domain but nowadays frequently available in healthcare domain[2]. Rule-based programming is used to develop rule-based systems that can achieve change flexibility, hence can maintain quality of data with the passage of time, which utlimately contributes to better patient care.

Systems developed using rule-based programming comprises of three basic components namely knowledge base (a collection of 'rules'), working memory and inference engine. Inference engines are those specified applications which have the capabilities to read a set of rules from the knowledge base, on input conditions identify the appropriate rule and execute the conforming actions[2]. Rules are made up of if-then structures having diverse properties with respect to representation and computation. In rule-based programming, the term 'rule' (also known as 'production rule') refers to 'if-then' structure whereas 'if' represents condition and 'then' is followed by the action part.

IF <*condition*> THEN <*action*>

Conceptually, 'production rule' is the same as 'if' statement in conventional programming i.e. having condition portion and an action part. Action is performed when condition is true. However a major difference between a 'rule' and 'if-statement' is that a 'rule' is part of data, while 'if' statement is part of code. Changing a portion of code requires programming skill, recompilation of code, which leads to extremely high maintenance costs.

Conventional Programming

In conventional programming, checks are encoded in the form of compiled codes written in some programming language like Java, C++, C#, etc. With new innovations in healthcare new checks and conditions are requisite to implement on the monthly or quarterly basis, which in turn may induce new bugs in the software; and excessive time is wasted to eradicate those bugs. In conventional software 'if' statements are used (instead of production rules) as part of code and only persons with programming skills can change the code. Therefore, in case of conventional programming based software, professionals of healthcare domain are not able to incorporate updated knowledge in software by themselves. Some aspects of rule-based programming and conventional programming have been summarized in the Table 1 given below;

Table 1: Comparison of Rule-Based programming and conventional programming

Characteristics	**Rule Based Programming**	**Conventional Programming**
Type of Programming	Declarative	Procedural
Development Tools	Special Artificial intelligence languages (LISP, PROLOG) or in specific tools like OPS-5, KEE, SQL^2, Clips, Jess, Drools, ILOG Rules, G2. etc.	Compiled languages like Java, C++, C#, VB, etc. are used to develop conventional software.
Implementation of Conditions and Actions	Production rules	If-else statements, nested if-else statements, switch statement
Part of	Data	Code
System Specification	Flexible and dynamic	Complex, static
Processing zone	Centralized on database. Checks are executed on data stored in the database which updates the database directly.	Decentralized (mostly client-server model). Processing can be done on both client and server side.
Efficiency	More efficient because of direct implementation at the database.	Less efficient due to excessive network traffic
Database Server	No special requirements.	Server task is only to fetch and update data, so it depends upon size of data in the server.
User Friendliness	Easy to use and understand	Difficult to use and understand as software development skill and access code are indispensible to update software

Example 2: Rules for Area of Circle

```
(P  inputrule
        (start)
        →
        (write "Enter Radius:")
        (read  <x>)
        (insert  'radius  <x>)
        (remove  1)
)
(P  calculateRule
        (radius  <r>)
        →
        (insert  'area  3.14* <r> * <r> )
        (remove 1)
)

(P  outputRule
        (area  <a>)
        →
        (write "Area of Circle=")
        (write <a>)
        (remove  1)
)
```

Conventional Programming

```
#include <iostream>
using namespace std;
int  input()
{
        cout<<"Enter Radius:";
        cin>> x;
        return x;
}
float  calculate( )
{       int r;
        float a;
        r = input();
        a = 3.14 * r * r;
        return a;
}
voidoutput( )
{       float a;
        a = calculate( );
        cout<< " Area of Circle ="<<a
                <<endl;
}
void main ( )
{
        area( );
        system("pause");
}
```

Tracing of both versions;

Production Rules in OPS-5 Syntax

Example 1: For calculating Factorial

```
(P  inputRule
        (start)
        →
        (write "enter a number:")
        (read  <x>)
        (insert  'num  <x>)
        (remove 1)                          )
(P   fact1
        (num   <x> == 1)
        →
        (insert  'fact  <x> )
        (insert  'factorial   1   1)
        (remove 1)        )
(P   fact2
        (num   <x> != 1)
        →
        (insert  'num  (<x>  - 1) )
        (insert  'fact  <x> )
        (remove 1)        )

(P   fact3
        (fact   <x> )
        (factorial  <x><y>)
        →
        (insert  'factorial  <x> + 1      <x> * <y> )
        (remove 1 ) (remove 2)  )

(P   result
        (factorial   <x><y>)
        - (fact  <x>)
        →
        (write  <x> - 1)
        (write  " ! = ")
        (write   <y> )
        (remove 1))
```

Command prompt (output window)

(insert 'start)
(run)

enter number: 4
4 != 24

~~(start)~~ **Memory**
~~(num 4)~~
~~(num 3)~~
~~(fact 4)~~
~~(num 2)~~
~~(fact 3)~~
~~(num 1)~~
~~(fact 2)~~
~~(fact 1)~~
~~(factorial 1 1)~~
~~(factorial 2 1)~~
~~(factorial 3 2)~~
~~(factorial 4 6)~~
~~(factorial 5 24)~~

Trace the above OPS-5 example by showing the status of working memory, if user enters 4 as the input.

FRAMES:

Frame based systems are hierarchical structures that contain a number of slots or named attributes which together describe a concept, object or event. Each frame lower down in the structure can inherit the attributes of a higher frame.

- Semantic nets cannot handle application specific knowledge. It needs to be built into the program.
- Not natural in expressing hierarchical relations
- Local & global information problem.

```
[frame : flat
        [isa      :       dwelling]
        [no-of-rooms    :       [range  :       1 to 20]
                                [default :      4]
        ]
        [wall-color     :       [default : white]]
        [floor  :       [range : wood,  stone, brick, plastic]]
]

[frame  :       flat-at-sixth-road
        [isa      :       flat]
        [no-of-rooms    :       [value  :       6]]
        [street-name    :       [value  :       sixth-road]]
]
```

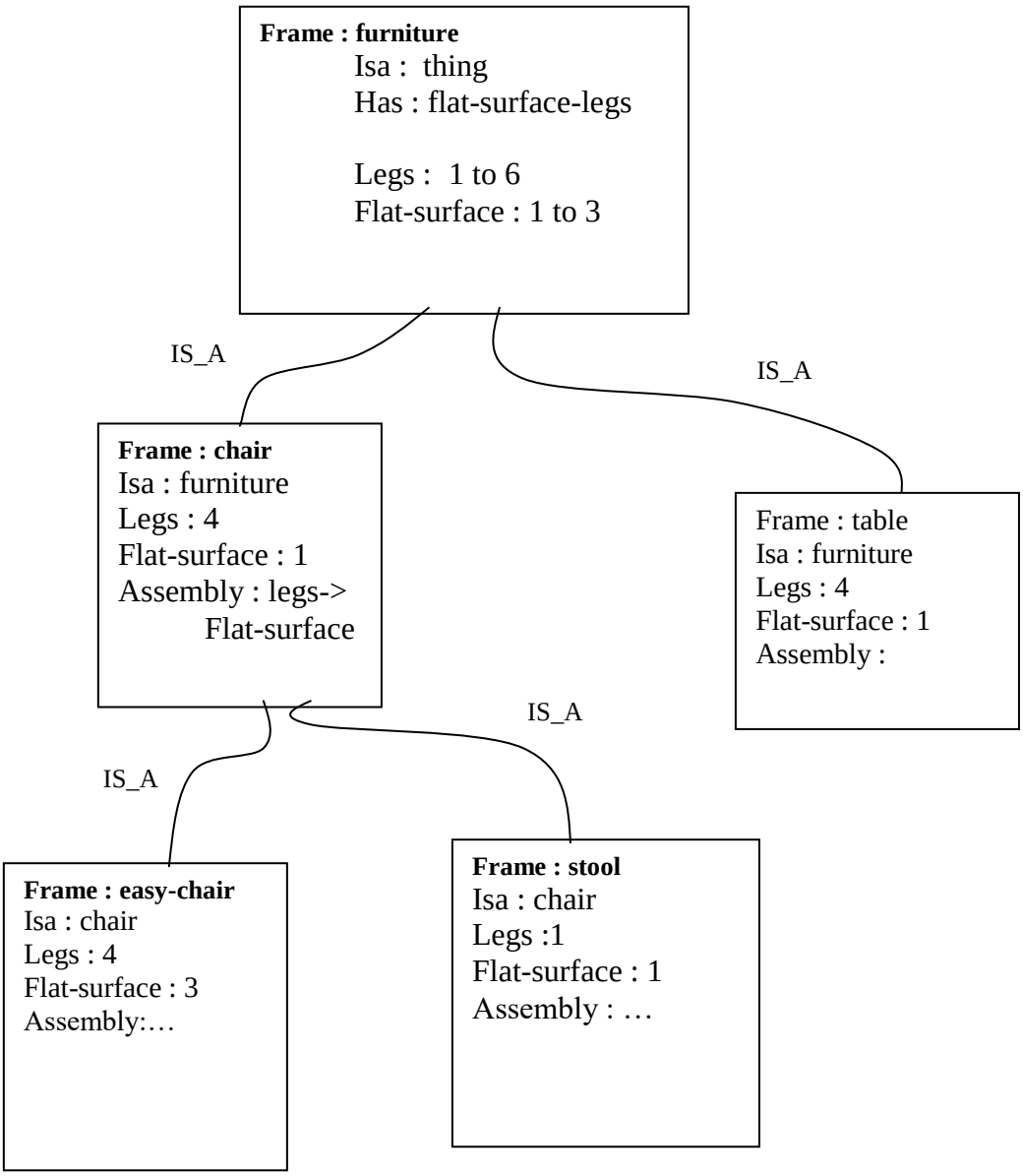
- Inheritance
- Defaults
- Demons
- Procedural attachments/methods

```
[frame :        vehicle
        [isa      :       thing]
        [motion-by      :       [range :        wheel tracks  wings]
                                [default :      wheel]]]
        [seats          :       [range :        1 to 20]
                                [default : 4]]
        [engine         :       [range :        none petrol  diesel  jet ]
                                [default :      petrol]]
]

[frame :        car
        [isa      :       vehicle]
        [wheels :       4]
]

[frame :        corolla
        [isa      :       car]
        [engine :       petrol]
```

Furniture Example:



Vehicle example (make by yourself)

SCRIPTS:

Frame like structures used to represent commonly occurring experiences such as going to movies, shopping in a supermarket, eating in a restaurant, or visiting a dentist. Like a script in a play, script structure is described in terms of actors, events, roles, props and scenes.

- Sequence of events
- Acts, Scenes

- Entry conditions
- Roles : people in script
- Props : objects in script
- Scenes
- Results

Example 1

Script name: restaurant
Role: Customer
Waiter
Cook
Owner

Entry condition : customer is hungry
Props : food
Table
Money

Scene1 : Enter Restaurant
Customer Goes into rest
Customer Move to sit
Go to scene2

Scene2 : ORDERING
Customer Signals to waiter,
Waiter Goes to customer
Customer Orders 'Bring me food' to waiter
Waiter Goes to cook
Waiter Give Customer's order to cook
Cook do (Prepare food script)

Scene3 : EATING
Scene4 : Payment Money
Customer Goes to the Counter
Owner tells the bill amount
Customer Pay money to Owner
(go to scene 5)
Scene5: Exiting
Customer leaves the restaurant

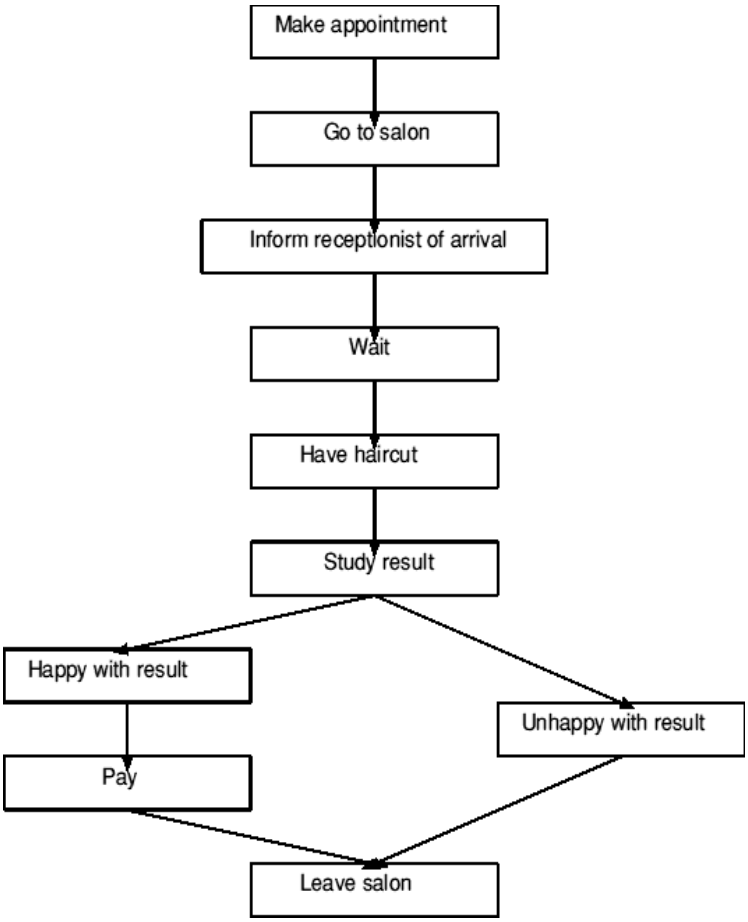
Result : Customer not hungry
rest. has less food
customer has less money
owner has more money.

An Other Example

The following example is adapted from *Charniak and McDermott, "Introduction to Artificial Intelligence", Addison Wesley, 1987.*

"Sally needed a haircut. She made an appointment to have one later that day. When she arrived at the salon she told the receptionist and was disappointed to find that her preferred hair dresser Andre was away. After a short wait, Sally had her hair cut. When she looked in the mirror, she was very disappointed with the result and angrily left without paying."

This script can be partially represented like this:



Purchasing Book Script (Attempt Yourself)

Knowledge Representation Reference Study

The function of any representation scheme is to capture the essential features of a problem domain and make that information accessible to a problem-solving procedure. It is obvious that a representation language must allow the programmer to express the knowledge needed for a problem solution. *Abstraction*, the representation of only that information needed for a given purpose, is an essential tool for managing complexity. Inheritance is an important abstraction technique. It is also important that the resulting programs be computationally efficient. *Expressiveness* and *efficiency* are major dimensions for evaluating knowledge representation languages. Many highly expressive representations are too inefficient for use in certain classes of problems. Sometimes, expressiveness must be sacrificed to improve efficiency. This must be done without limiting the representation's ability to capture essential problem-solving knowledge. Optimizing this trade-off is a major task for designers of intelligent systems.

Knowledge representation languages are also tools for helping humans solve problems. As such, a representation should provide a *natural* frame work for expressing problem-solving knowledge; it should make that knowledge available to the computer and assist the programmer in its organization.

The computer representation of floating-point numbers illustrates these trade-offs. In general, real numbers require an infinite string of digits to be fully described; this cannot be accomplished on a finite device, indeed a finite-state machine. The best answer to this dilemma is to represent the number in two pieces; its *significant* digits and the location within those digits of the decimal point. Although it is not possible to actually store a real number in a computer, it is possible to create a representation that functions adequately in most practical applications.

Floating-point representation thus sacrifices full expressive power to make the representation efficient, in this case to make it possible. The representation allows algorithms for multiple-precision arithmetic, giving effectively infinite precision by limiting round-off error to any prespecified tolerance. It also guarantees well-behaved round-off errors. Like all representations, it is only an abstraction a symbol pattern that designates a desired entity and not the entity itself.

The array is another representation common in computer science. For many problems, it is more natural and efficient than the memory architecture implemented in computer hardware. This gain in naturalness and efficiency involves compromises in expressiveness, as illustrated by the following example from image processing.

The visual scene is made up of a number of picture points. Each picture point, or *pixel*, has both a location and a number value representing its gray level. It is natural, then, to collect the entire scene into a two-dimensional array where the row and column address gives the location of a pixel (X&Y coordinates) and the content of the array element is the gray level at that point. Algorithms are designed to perform operations like looking for isolated points to remove noise from the image, finding threshold levels for discerning objects and edges, summing contiguous elements to determine size or density and in various other ways transforming the picture point data. Implementing these algorithms is straightforward, given the array representation and the

FORTRAN language. This task would be quite cumbersome using other representations such as the predicate calculus, records or assembly code, because these do not have a natural fit with the material being represented.

When we represent the picture as an array of pixel points, we sacrifice fineness of resolution (compare a photo in a newspaper to the original print of the same picture). In addition, pixel arrays cannot express the deeper semantic organization of the image. For example, a pixel array cannot represent the organization of chromosomes in a single cell nucleus, their genetic function, or the role of metaphase in cell division. This knowledge is more easily captured using a representation such as predicate calculus or semantic networks.

A representational scheme should:

1. Be adequate to express all of the necessary information.
2. Support efficient execution of the resulting code.
3. Provide a natural scheme for expressing the required knowledge.

McDermott and others have stated that the key to writing a successful knowledge-based program is the selection of appropriate representational tools. Often, lower-level language (BASIC, FORTRAN etc.) programmers fail in building expert systems simply because these languages do not provide the representational power and modularity required for knowledge-based programming (McDermott, 1981).

In general, the problems AI attempts to solve do not lend themselves to the representations offered by more traditional formalisms such as arrays. Artificial intelligence is concerned with qualitative rather than quantitative problem solving, with reasoning rather than calculation, with organizing large and varied amounts of knowledge rather than implementing a single, well-defined algorithm. To support these needs, an AI representation language must:

A. Handle qualitative knowledge.
B. Allow new knowledge to be inferred from a set of facts and rules.
C. Allow representation of general principles as well as specific situations.
D. Capture complex semantic meaning.
E. Allow for meta-level reasoning.

These topics make up the remainder of our discussion of knowledge representation.

A. Handle qualitative knowledge.

Artificial intelligence programming requires a means of capturing and reasoning about the *qualitative* aspects of a problem.

One way to represent this blocks arrangement would be to use a Cartesian coordinate system, giving the X & Y coordinates of each vertex of a block. Although this approach certainly describes the blocks world and is the correct representation for many tasks, it fails to capture directly the properties and relations required for qualitative reasoning. These include determining which blocks are stacked on other blocks and

which blocks have clear tops so that they can be picked up. *Predicate calculus,* the blocks world could be described by the logical assertions:

```
clear(c).
clear(a).
ontable(a).
ontable(b).
on(c,b).
cube(b).
cube(a).
pyramid(c).
```

The first word of each expression (on, ontable, etc.) is a *predicate* denoting some property or relationship among its arguments (appearing within parentheses). The arguments are symbols denoting objects (blocks) in the domain. The collection of logical clauses describes the important properties and relationships of the blocks world. Predicate calculus provides artificial intelligence programmers with a well-defined language for describing and reasoning about qualitative aspects of a system. Although it is not the only approach to representation, it is probably the best understood. In addition, it is sufficiently general to provide a foundation for other formal models of knowledge representation.

B. Allow new knowledge to be inferred from a set of facts and rules.

The ability to infer additional knowledge from a world description is essential to any intelligent entity. Humans, for example, do not store an inflexible description of every situation encountered; this would be impossible. Rather, we are able to formulate and reason from abstract descriptions of classes of objects and situations. A knowledge representation language must provide this capability.

In the blocks world example, we might like to define a test to determine whether a block is *clear*, that is, has nothing stacked on top of it. This is important if a robot hand is to pick it up or stack another block on top of it. It is not necessary to explicitly add this to our description for all such blocks. Instead, we define a general rule that allows the system to infer this information from the given facts. In predicate calculus, the rule can be written

Or, “for all X, S is clear if there does not exist a Y such that Y is on X.” This rule can be applied to a variety of situations by substituting different values for X & Y. By letting the programmer form general rules of inference, predicate calculus allows much greater economy of representation, as well as the possibility of designing system that are flexible and general enough to respond intelligently to a range of situations.

C. Allow representation of general principles as well as specific situations.

In addition to demonstrating the use of logical rules to infer additional knowledge from basic facts, the blocks example introduced the used of variables in the predicate calculus. Because an intelligent system must be as general as possible, any useful representation language needs variables. The requirements of qualitative reasoning make the use and implementation of variables subtly different from their treatment in traditional programming languages. The assignment, type and scope rules we find in calculation-oriented languages are too restrictive for reasoning systems. Good knowledge representation languages handle the bindings of variable names, objects and values in a highly dynamic fashion.

D. Capture complex semantic meaning.

Many artificial intelligence problem domains require large amounts of highly structured interrelated knowledge. It is not sufficient to describe a car by listing its component parts; a valid description must also describe the ways in which those parts are combined and the interactions between them. This view of structure is essential to a range of situations including taxonomic information, such as the classification of plants by genus and species, or a description of complex objects such as a diesel engine or a human body in terms of their component parts. In our blocks example, the interactions among predicates were the basis of a complete description of the arrangement.

Semantic relationships are also important in describing the causal relationships between events occurring over time. These are necessary to understand simple narratives such as a child's story or to represent a robot's plan of action as a sequence of atomic actions that must be executed in order.

Though all of these situations can ultimately be represented as collections of predicates or similar formalisms, some higher-level notion of structure is desirable to help both program and programmer deal with complex concepts in a coherent fashion. For example a simple description of a bluebird might be "a bluebird is a small blue-colored bird and a bird is a feathered flying vertebrate." This may be represented as a set of logical predicates:

```
hassize(bluebird, small).
hascovering(bird, feathers).
hascolor(bluebird, blue).
hasproperty(bird, flies).
isa(bluebird, bird).
isa(bird, vertebrate).
```

This description cold also be represented graphically by using the *arcs* (*or links*) in a graph instead of predicates to indicate relations. This description, called a *semantic network*, is a fundamental technique for representing semantic meaning.

Because relationships are explicitly denoted by the links of a graph, an algorithm for reasoning about the domain could make relevant associations simply by following links. In the bluebird illustration, a system need only follow two links in order to determine that a bluebird is a vertebrate. This is more efficient than exhaustively searching a data base of predicate calculus descriptions of the form isa(X,Y).

In addition, knowledge may be organized to reflect the natural class-instance structure of the domain. Certain links in a semantic network indicate class membership and allow properties attached to a class description to be *inherited* by all members of the class. This inheritance mechanism is built into the language itself and allows knowledge to be stored at the highest possible level of abstraction.

Knowledge Representation Language

Inheritance is a natural tool for representing taxonomically structured information and ensures that all members of a class share common properties. Because of these gains in efficiency and naturalness of expression and because they make the power of graph theory available for reasoning about the structural organization of a knowledge base, semantic nets are an important alternative to predicate calculus.

We have just seen an example of a single problem description having two or more representations. Selecting the proper representation is a critical task for AI programmers. The benefits of a higher-level representation such as a semantic network are not unlike the advantages of a higher-level programming language such as Pascal over low-level machine code: although every Pascal program is ultimately compiled down to machine code, a higher-level language is almost essential for the creation of well-designed, coherent programs. Continuing this simile, predicate calculus or LISP s-expressions may be thought of as machine languages for AI representation. AI knowledge-structuring techniques such as frames, semantic networked, scripts and plans provide more powerful constructs for organizing knowledge.

E. Allow for meta-level reasoning.

An intelligent system not only should know things but also should know what it knows. It should be able not only to solve problems but also to explain how it solved the problems and why it made certain decisions. It should be able to describe its knowledge in both specific and general terms, recognize the limitations of its knowledge and learn from its interactions with the world. This "knowing about what you know" constitutes a higher level of knowledge called *meta-knowledge,* and its automation is essential to the design and development of truly intelligent systems.

The problem of formalizing meta-knowledge was first explored by Bertrand Russell in his theory of logical types. Briefly, if sets are allowed to be members of other sets (a situation analogous to having knowledge), it is possible to have sets that are members of themselves. As Russell discovered, this leads to irresolvable paradoxes, Russell disallowed these paradoxes by classifying sets as being of different types depending on whether they were sets of individuals, sets of sets of individuals, etc. Sets could not be members of sets of a smaller or equal type number. This corresponds to the distinctions between knowledge and meta-knowledge. As Russell discovered, numerous difficulties arise in attempting to formally describe this reasoning (Whitehead and Russell, 1950).

The ability to learn from examples, from experience, or even from high-level instructions (as opposed to being programmed) depends on the application of meta-knowledge. The representational techniques developed for AI programming offer the flexibility and modifiability required of learning systems and form a basis for this research.

To meet the needs of symbolic computing, artificial intelligence has developed representation languages such as the predicate calculus, semantic networks, frames, and objects. LISP and PROLOG are languages for implementing these and other representations. All of these tools are examined in detail throughout the text.

In building a knowledge base, a programmer must select the significant objects and relations in the domain and map these into a formal language. The resulting program must contain sufficient knowledge to solve problems in the domain, it must make correct inferences from this knowledge and it must do so efficiently.

We can think of a knowledge base in terms of a mapping between the objects and relations in a problem domain and the computational objects and relations in a program (Bobrow, 1975). The results of inferences on the knowledge base should correspond to the results of actions or observations in the world. The computational objects, relations and inferences available to programmers are determined by the knowledge representation language they select. The proper language can help the programmer acquire, organize and debug the knowledge base.

There are general principles of knowledge organization that apply across a variety of domains and can be directly supported by a representation language. For example, class hierarchies are found in both scientific and commonsense classification systems. How may we provide a general mechanism for representing them? How may we represent definitions? What is the relation between a rule and its exceptions? When should a knowledge-based system make default assumptions about missing information and how should it adjust its reasoning if these assumptions prove wrong? How may we best represent time? Causality? Uncertainty? Progress in knowledge-based systems depends on discovering the principles of knowledge organization and supporting them in higher-level representational tools.

It is useful to distinguish between a representational *scheme* and the *medium* of its implementation (Hayes, 1974). This is similar to the distinction between data structures and programming languages. Programming languages are the *medium* of implementation; the data structure is the *scheme*. Generally, knowledge representation languages are more constrained than predicate calculus or programming languages such as LISP & PROLOG. The constraints take the form of explicit structures for representing categories of knowledge. The medium in which they are implemented might be PROLOG, LISP, or a conventional language such as C, C^{++} or ADA.

Over the past 25 years, numerous representational schemes have been proposed and implemented, each of them having its own strengths and weaknesses. Mylopoulos and Levesque (1984) have classified these into four categories:

1. **Logical representation schemes.** This class of representations uses expressions in formal logic to represent a knowledge base. Inference rules and proof procedures apply this knowledge to problem instances. First-order predicate calculus is the most widely used logical representation scheme, but it is only one of a number of logical representations (Turner, 1984). PROLOG is an ideal programming language for implementing logical representation schemes.
2. **Procedural representation scheme.** Procedural schemes represent knowledge as a set of instructions for solving a problem. This contrasts with the declarative representations

provided by logic and semantic networks. In a rule-based system, for example, an if …… then ……. Rule may be interpreted as a procedure for solving a goal in a problem domain, by solving the premises in order. A production system may be seen as an example of a procedural representation scheme.

3. **Network representation schemes.** Network representations capture knowledge as a graph in which the nodes represent objects or concepts in the problem domain and the arcs represent relations or associations between them. Examples of network representations include *semantic network, conceptual dependencies* and *conceptual graphs.*
4. **Structured representation schemes.** Structured representation languages extend networks by allowing each node to be a complex data structure consisting of named slots with attached values. These values may be simple numeric or symbolic data, pointers to other frames, or even procedures for performing a particular task. Examples of structured representations include *scripts, frames,* and *objects.*

What Have Been Represented:

Let us consider what kinds of knowledge have been represented so far in above discussion:

Objects: -- Facts about objects in our world domain. *e.g.* Guitars have strings, trumpets are brass instruments.

Events: -- Actions that occur in our world. *e.g.* Steve Vai played the guitar in Frank Zappa's Band.
Performance:-- A behavior like *playing the guitar* involves knowledge about how to do things.
Meta-knowledge: knowledge about what we know. *e.g.* Bobrow's Robot who plan's a trip. It knows that it can read street signs along the way to find out where it is. In AI we must represent knowledge and there are two entities to deal with:
Facts :-- truths about the real world and what we represent, this can be regarded as the *knowledge level.*
Representation of the facts: which we manipulate. This can be regarded as the *symbol level* since we usually define the representation in terms of symbols that can be manipulated by programs.

Issues of Knowledge Representation

The following properties should be possessed by a knowledge representation system.

Representational Adequacy:the ability to represent the required knowledge i.e. adequate information is available.

Inferential Adequacy:the ability to manipulate the knowledge represented to produce new knowledge corresponding to that inferred from the original;

Inferential Efficiency:the ability to efficiently execute/direct the inferential mechanisms into the most productive directions by storing appropriate guides;

Acquisitional Efficiency:the ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.