Knowledge-Based Systems

The Problem
The declarative paradigm
KT Applications
Data, Information & Knowledge
Knowledge-Based Systems
Modelling Tools
Logic programming
How do we compute with declarative knowledge?

Eg “if the dollar is high and falling and inflation is low then gold is worth looking at”
  • declarative
  • not a command.

Knowledge processing is concerned with computation using the expertise in declarative statements.
Non-imperative statements.

• “workers are supervised by the manager of the department in which they work, and departmental managers are supervised by the General Manager”

• “in Sydney in January a very hot day is usually followed by a cold wet day”
  • “very hot”, “usually followed”

• true or false to some degree
Applications of Artificial Intelligence

• Knowledge-Based Systems

The Problem

⇒ The declarative paradigm

KT Applications

Data, Information & Knowledge

Knowledge-Based Systems

Modelling Tools

Logic programming
The declarative paradigm

• representation of, and computation with, both pure and modal statements.
“workers are supervised by the manager of the department in which they work, and departmental managers are supervised by the General Manager”

“to find the supervisor of X”

• “to find the list of all people supervised by Y”.
Knowledge processing

- representation of “expert knowledge” or “expertise”
- to provide the basis for computation without descending to explicit representation of imperative aspects
Knowledge processing language

• capable of representing statements of fact
• to enable computations to be derived from those statements of fact
Applications of Artificial Intelligence

- Knowledge-Based Systems

The Problem
The declarative paradigm
- KT Applications
Data, Information & Knowledge
Knowledge-Based Systems
Modelling Tools
Logic programming
Applications of Artificial Intelligence

Knowledge technology focuses on the high level expertise in an application.

When assessing an application: ‘how far down’ does it go?

It may require that a large amount of common sense.

For example, a system which selected the location for summer vacation

A perceived block of knowledge is “the tip of an iceberg”.

How big is the “iceberg”?

“Bottom up” applications, and

“Top down” applications.
Applications of Artificial Intelligence

Bottom up application: we initially know the foundation on which the system will be built and have yet to assess the extent of the expertise which is to be added.

Eg. a corporate databases as the foundation for a knowledge base.

In database systems the knowledge has not been properly represented, analysed and implemented.

Most database design methodologies do not attempt to deal with ‘rules’.

When building a corporate database into a corporate knowledge base: proceed ‘bottom up’ by systematically adding more and more expertise.
Top down application:

we have initially identified a block of expertise and have yet to assess the extent of the foundation on which the system will be built.

require careful analysis to ensure that what initially appeared to be feasible does not turn into a “bottomless pit” when the analyst starts to ‘dig’ into the application.

To determine whether knowledge systems are the appropriate paradigm for an application we consider a number of factors.
• Can the problem be solved effectively by conventional algorithmic systems?

  If ‘yes’ then knowledge technology may not be appropriate.

  Knowledge technology may be appropriate for ill-structured problems: for which there is no efficient algorithmic solution.

  Eg. weather forecasting.
Applications of Artificial Intelligence

- Is the application well-bounded?

  Eg. to diagnose faults in cars

    Either limited amount of practical expertise

    Or include a ‘deep’ system which contained expertise from mechanical engineering, electrical engineering, aerodynamics and so on.
Applications of Artificial Intelligence

• Does the application depend on ‘common sense’?

  “Suppose that Aristotle were to come back to life; how long would it take to train him to do this task?”

  Form a view on the amount of knowledge required to do the task at hand.

• Is there a need and a desire for a knowledge-based system?

  Highly significant for knowledge-based systems applications.
Applications of Artificial Intelligence

• Is there an effective knowledge source?
  Is a human expert willing and able?
  Multiple sources compatible?

• Is the raw expertise in a form that can be understood by the knowledge engineer?
  Can human expert articulate the expertise?
    Eg. how you move a finger.
    Some experts find it very difficult to explain how they do what they do.
Applications of Artificial Intelligence

- Is the raw expertise of an appropriate form for implementation in a knowledge-based system?
  
  Depends on a diagram or a scene?

  The “telephone test”: “could the expertise be applied effectively over a telephone?”

- Does the application contain a significant amount of heuristic knowledge and uncertain knowledge?

  If so then knowledge technology may be particularly appropriate.

  Eg. if the basis of the knowledge is experience.
Knowledge in a database

- Knowledge
- Information and data

Procedural code

- Host language
- DBMS

Database
Applications of Artificial Intelligence

Three generations of systems

- **Decisions**
  - DSS (Strategic)
- **Control**
  - MIS (Middle)
- **Do**
  - DP (Supervise)

Management Level

Activity

Slide 19 © J.K. Debenham, 2003
Applications of Artificial Intelligence

Knowledge technology and IDSSs

1980 1990 2000

Knowledge Technology
Database Systems
Operations Research
Statistics
Simulation
Mathematical Modelling

IDSS
Applications of Artificial Intelligence

Approaches to decision making

- Human Expertise
- Managerial Decision
- Mathematical Model

Top down: Knowledge technology
Bottom up: Statistics, OR Simulation, Mathematical modelling
Maintenance is a key issue for KBs.

Mid 1980’s “expert systems” were being built by small teams of clever, intuitive and resourceful personnel who had substantial experience in programming, systems analysis and in database and at least one expert systems “shell”.

Systems were constructed moderately quickly, but maintenance became an increasing problem particularly with the integrity of the knowledge base.

Why then is the maintenance of KBs a problem?
1960’s consideration of the maintenance of traditional programs led to structured programming.

1970’s consideration of the maintenance of databases led to database design techniques.

1980’s consideration of the maintenance of knowledge bases led to knowledge base design methodologies.

The reason for the emergence of maintenance is a key issue was the early approaches to systems construction were “software led”.

The solution to the maintenance problem is “design”.
Applications of Artificial Intelligence

The methodology trap

Design

then

Maintain

© J.K. Debenham, 2003
A methodology should address:

- knowledge documentation
- requirements specification
- knowledge acquisition
- knowledge modelling
- knowledge analysis
- knowledge implementation
- maintenance
Knowledge Technology Myth 1:

- KT is the “last word” in computing. By this we mean that because knowledge represents the wisdom in an application sometimes we forget that wisdom alone is not sufficient to lead to efficient calculations.

Result: in-efficient systems

Moral: understand limitations of technology
Knowledge Technology Myth 2:

- KT is as “end user” activity. By this we mean that as shells are so easy to use we can distribute knowledge processing to the end user. Knowledge technology is highly technical; no technology should be distributed until it is well understood.

Result: un-maintainable systems and uncontrolled development

Moral: distribute technology only when fully understood.
Knowledge Technology Myth 3:

- KT as “stand alone” systems. This myth is generated by many knowledge processing products which do not provide convenient links into other investments in systems.

Result: high cost, low performance systems and isolated systems

Moral: exploit existing investments to develop low cost high gain systems
Knowledge-based systems have been constructed in the following application areas:

### Applications of Artificial Intelligence

<table>
<thead>
<tr>
<th>Problem-Solving Paradigm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Governing system behaviour to meet specifications</td>
</tr>
<tr>
<td>Design</td>
<td>Configuring objects under constraint</td>
</tr>
<tr>
<td>Diagnosis</td>
<td>Inferring system malfunctions from observables</td>
</tr>
<tr>
<td>Instruction</td>
<td>Diagnosing, debugging, and repairing student behaviour</td>
</tr>
<tr>
<td>Interpretation</td>
<td>Inferring situation description from data</td>
</tr>
<tr>
<td>Monitoring</td>
<td>Comparing observations to expectations</td>
</tr>
<tr>
<td>Planning</td>
<td>Designing actions</td>
</tr>
<tr>
<td>Prediction</td>
<td>Inferring likely consequences of given situations</td>
</tr>
<tr>
<td>Prescription</td>
<td>Recommending solution to system malfunction</td>
</tr>
<tr>
<td>Selection</td>
<td>Identifying best choice from a list of possibilities</td>
</tr>
<tr>
<td>Simulation</td>
<td>Modelling the interaction between system components</td>
</tr>
</tbody>
</table>
Control systems adaptively govern the behaviour of a given system. Eg, controlling a manufacturing process.

Obtains data on the system's operation, interprets the data to form an understanding of the state of the system or a prediction of its future state, and determines and executes needed adjustments.

Must also perform monitoring and interpretation tasks to track system behaviour over time.

Some systems include prediction and planning tasks that allow them to formulate plans to avoid anticipated problems.
Design systems
configure objects under a set of problem constraints.

Eg, designing a computer system.
usually perform their tasks following a series of steps, each with its own specific constraints.
these steps are usually dependent upon each other
these types of systems are often built using a non-monotonic reasoning technique.
Applications of Artificial Intelligence

Diagnosis systems

infer system malfunctions or faults from observable information.

often have knowledge of possible fault conditions with means to infer whether the fault exists from information on the system observable behaviour.

Eg, diagnosing a given disease.

A more recent trend in the field relies on a model-based reasoning approach

Most diagnosis systems include a prescription task that offers a remedy to the detected fault.
Instruction systems guide the education of a student in a given topic. treat the student as a system that must be diagnosed and repaired.

begin by interacting with the student to form a model of the student's understanding of the topic.

compare this student model with an “ideal” model to uncover weaknesses in the student's understanding.

followed by remedial instruction to correct any misunderstandings.
Applications of Artificial Intelligence

Interpretation systems

produce an understanding of a situation from available information.

Typically this information consists of data from such sources as sensors, instruments, test results, etc.

Eg, machine monitoring sensors, imaging systems or speech analysis results.

translate the raw data into symbolic form that describes the situation.

often need to work with noisy, incomplete or unreliable data that requires inexact or statistical reasoning.
Monitoring systems

compare observable information on the behaviour of a system with system states that are considered crucial to its operation.

usually interpret signals from sensors and compare the information with known crucial states.

When a crucial state is detected, an established sequence of tasks is performed.
Planning systems
form actions to achieve a given goal under problem constraints.

Eg, planning the different tasks performed by a robot to accomplish a given work function.

must have the flexibility to change the series of planned tasks when they obtain new problem information.

need the ability to backtrack and reject a current line of reasoning in favour of exploring a better one.

usually require non-monotonic reasoning.
Prediction systems
infer likely consequences from a given situation.

attempt to predict future events using available information and a model of the problem.

often must be able to reason about time or ordered events.

Models must be available to infer how some given action influences future events.

Eg, predicting the expected damage to a crop from an invading insect.

Intelligent simulation models are often used in these types of systems.
Prescription systems recommend solutions to a given system malfunction.

usually first incorporate a diagnostic task to determine the nature of the malfunction.

most rely on a “canned” prescription for each known fault.

More advanced systems incorporate planning and prediction techniques for creating tailored remedies.
Selection systems

identify the best choice from a list of possibilities.

work from problem specifications defined by the user and attempt to find a solution that most closely matches these specifications.

usually employ an inexact reasoning technique or a matching evaluation function when forming their selections.
Simulation systems

model a process or system to permit operational studies under various conditions.

model the various components of the system and their interactions.

Users are usually permitted to make adjustments to the model to account for either existing or hypothetical conditions.

can be used to predict operating conditions for the real system.
Applications of Artificial Intelligence

• Knowledge-Based Systems

The Problem
The declarative paradigm
KT Applications

→ Data, Information & Knowledge
Knowledge-Based Systems
Modelling Tools
Logic programming
Objective: to present a design methodology for knowledge-based systems.

Data thing: a fundamental, indivisible thing in that application.

• Can be represented naturally by populations and labels.
Associations between things.

If an association can be described by a succinct, computable rule it is called an *explicit association*.

If an association can not be described by a succinct, computable rule it is called an *implicit association*.

An *information thing* is an implicit association between the data things.

A *knowledge thing* is an explicit association between the data things or information things.
Associations are often “functional”; called a *functional association*.

If an information thing is functional then it can be represented as a relation with a key.

If a knowledge thing is functional then it can be represented as a clause group.
Applications of Artificial Intelligence

Relational database: “functional dependency” and “key”.

branch/department/manager(

branch-no., dept-no., manager-name )

A relation can represent a functional dependency between items of data.

Cannot give a succinct definition of the above function.

*Implicit* functional association.
“To convert from degrees Fahrenheit to degrees Celsius, subtract 32 and divide by 1.8”.

Function between two items of data.

We can actually specify the function:

\[ f : (\text{deg F}) \rightarrow (\text{deg C}) \]

This function can be described by a succinct, computable rule:

\[ f(x) = \left( x - 32 \right) \div 1.8 \]

*Explicit* functional association.
“Selling price is 1.25 times buying price” represented by the clause:

\[
\text{item/sale-price}(x, y) \leftarrow \\
\text{item/cost-price}(x, z), \quad y = 1.25 \times z
\]

In functional form from the relation:

\text{item/cost-price}

to the relation:

\text{item/sale-price}

Explicit, succinct and computable.

Explicit functional association between information things.
In an application:

- *data* is the fundamental, indivisible things in that application.

- *information* is the implicit associations between data things in that application.

- *knowledge* is the explicit associations between items of information things and/or data things in that application.
Distinction between implicit and explicit associations is hard to draw precisely.
Knowledge-Based Systems

The Problem
The declarative paradigm
KT Applications
Data, Information & Knowledge
→ Knowledge-Based Systems
Modelling Tools
Applications of Artificial Intelligence

Not possible to define knowledge-based systems solely on the basis of the way in which a system is implemented.

A *knowledge-based system* is a system which represents an application containing a significant amount of real knowledge, and has been designed, implemented and possibly maintained with due regard for the nature and structure of the data, information and knowledge.
Expert systems are thought of as prototype knowledge-based systems.

An *expert system* is a system in which the knowledge and has been deliberately represented “as it is”.

In particular, in an expert system the represented knowledge is often intended to solve problems *in the same way* as the expert solved the problems.
Applications of Artificial Intelligence

• Expert systems are often built to perform in the manner of a particular trained human expert. A knowledge-based system is not constrained in this way.

• Expert systems do not usually interact with large corporate databases. Knowledge-based systems belong on the corporate system platform and should be integrated with all principal, corporate resources.
Applications of Artificial Intelligence

- Expert systems usually perform tasks which are clearly “contained”; a knowledge-based system should be based on carefully modelled and normalised knowledge which should enable it to expand across boundaries between formally separated tasks.

Expert systems are often being associated with the knowledge of a particular expert and have something of a pioneering flavour.

Knowledge-based systems have something of a systems architectural flavour.
Design

A design methodology is a method that given an application produces a design for a computer system for that application and is sufficiently prescriptive for it to be useful in a teamwork situation.

A design methodology has given due regard for the nature and structure of knowledge if the structure of the raw knowledge has been faithfully represented and has been preserved as much as possible during the design process.
The design technique:

• has faithfully represented raw structure of the knowledge if all of the functional interpretations of the raw expertise have been represented in a cohesive way.

• has preserved the raw structure of the knowledge if as the design process proceeds the explicit associations are not decomposed into component functional associations until it is absolutely necessary to do so.
Implementation

An implementation has given due regard for the nature and structure of the knowledge if the raw structure of the explicit associations has been preserved as much as possible by the implementation.

This does not necessary mean that the explicit associations should be implemented in some declarative knowledge processing language.
Taxonomy of computer system implementations is based on:

- the way in which the real knowledge is stored;
- the way in which the real information is stored;
- the way in which the real data is stored;
- the amount of the system (knowledge, information and data) which is designed to accommodate updates, and
- the amount of the system (knowledge, information and data) which represents queries which the system has been designed to respond to.
The amount of the system, including knowledge, information and data, which is designed to accommodate updates is called the *update scope*.

The amount of the system, including knowledge, information and data, which represents queries which the system has been designed to respond to is called the *query scope*.

Beyond the scope of the queries and updates with which the system is designed to cope there will be more substantial “*maintenance operations*”. 
Applications of Artificial Intelligence

Data processing implementation

Information and knowledge in conventional programming language

Data and information in simple storage technology

Update scope

Query scope
Applications of Artificial Intelligence

Database implementation

Knowledge in conventional programming language

Information and data in database management system

Update scope

Query scope
Expert systems implementation

Knowledge in conventional programming language

Information and data in database management system

Knowledge, information and data in expert systems shell

Possible access from auxiliary database

Update scope

Query scope
Deductive database implementation

Knowledge in knowledge language

Information and data in database management system

Limited update scope

Update scope

Query scope
Knowledge-based implementation

Knowledge in knowledge language

Information and data in database management system

Update scope

Query scope
Applications of Artificial Intelligence

**Maintenance**

A system which “has been maintained with due regard for the nature and structure of the data, information and knowledge”:

- the recognition by the maintenance procedure of the relationship between real knowledge things and their implementation, and
- the recognition by the maintenance procedure of the interrelationships between the representations of pairs of knowledge things.
The relationship between real knowledge things and their implementation.

“the sale price of spare parts is the cost price marked up by a universal mark-up rate”.

Can be represented as:

- one cluster
- three groups
- a number of imperative programs

The maintenance procedure should recognise the relationship between the raw expertise and the implementation of that expertise.
The recognition by the maintenance procedure of the interrelationships between the representations of pairs of knowledge things.

“the profit on a part is the product of the cost price of the part and the mark-up factor of the part less 1”.

This may be represented as the clause:

\[
\text{part/profit}(x, y) \leftarrow \text{part/cost-price}(x, w), \\
\text{part/mark-up-factor}(x, u), \\
y = w \times (u - 1) \]
“the tax payable on a part is 10% of the product of the cost-price of the part and the mark-up factor of the part”.

This may be represented as the clause:

\[
\text{part/tax}(x, y) \leftarrow \text{part/cost-price}(x, w), \quad \text{part/mark-up-factor}(x, u), \quad y = (w \times u) \times 0.1 \quad [B]
\]
Both [A] and [B] have buried within them the sub-rule “the selling price of a part is the cost price of that part multiplied by the mark-up factor of the part”.

This sub-rule may be represented as the clause:

\[
\text{part/sale-price}(x, y) \leftarrow \text{part/cost-price}(x, z), \\
\text{part/mark-up-factor}(x, w), \quad y = (z \times w)
\]  

[C]

Now if the expertise represented in clause [C] should change then both clause [A] and clause [B] will have to be modified.
The maintenance procedure should be able to recognise the interrelationships between the representations of pairs of knowledge things.

[A] can be re-expressed so that they do not share a sub-rule.

“the profit on a part is the sale price of the part less the cost price of the part”.

This may be represented as the clause:

\[
\text{part/profit}( x, y ) \leftarrow \text{part/sale-price}( x, z ), \text{part/cost-price}( x, w ), y = z - w \ [D]
\]
If [B] had been expressed as

“the tax payable on a part is 10% of the sale price of that part”.

This may be represented as the clause:

\[
\text{part/tax( } x, y \text{ ) } \leftarrow \text{part/sale-price( } x, z \text{ ), } y = z \times 0.1 \quad [E]
\]

Clauses [D] and [C] together imply [A] and clauses [E] and [C] together imply [B].

[D] and [E] do not share a sub-rule.
Knowledge-Based Systems

The Problem
The declarative paradigm
KT Applications
Data, Information & Knowledge
Knowledge-Based Systems
  ➔ Modelling Tools
Logic programming
Applications of Artificial Intelligence

Structure of methodology

- Lifecycle
- Steps
- Tasks
- Sub-tasks
- Methods, guidelines and algorithms
- Deliverables
Applications of Artificial Intelligence

Knowledge Engineering Process

Requirements Specification

Requirements Model

Conceptual Model

Functional Model

Internal Model

System Analysis

System Engineering

System Layout

System Implementation

Physical Model
This methodology employs BR (ER) modelling to represent data and information, and a hybrid model to represent knowledge.

The hierarchic structure of data, information and knowledge are exploited:

- the things which are identified by BR modelling are viewed as predicates in terms of which the knowledge must be expressed.
Thus BR modelling makes a substantial contribution to the analysis of the knowledge.

BR modelling identifies the procedure names in terms of which the knowledge will be expressed.

ER may also be used.
The conceptual model is a representation of the expertise required to perform the tasks; it does not contain a representation of the system requirements.

The *conceptual model* consists of:

- an ER or BR model of the data and information in the application;
- a set of predicates which are extracted from the ER or BR model, and
- a model of the knowledge using clusters, dependency diagrams and logic.
Applications of Artificial Intelligence

Conventional database design

Application

BR analysis

Informal model of rules

Implement

Procedural code

Programming language

BR model

Implement

Conventional database

DBMS
Applications of Artificial Intelligence

Knowledge-based system modelling

Application

Knowledge analysis

BR analysis

Formal model of knowledge

Predicates

BR model

Implement

Implement
Applications of Artificial Intelligence

Data and Information Representation

We represent a thing-population (or NOLOT) by an oval shape.

The identifying population is shown in parentheses beneath the thing-population name inside the oval shape.
If all labels associated with population A are also associated with population B, then population A is a *sub-type* of population B.

The collection of all sub-type relationships is called the *type hierarchy*.

The type hierarchy will have a lattice structure in general.
An *information* thing is an implicit association between two or more data things in the application.

<table>
<thead>
<tr>
<th>part-number</th>
<th>dollar-amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>1.23</td>
</tr>
<tr>
<td>2468</td>
<td>2.34</td>
</tr>
<tr>
<td>3579</td>
<td>3.45</td>
</tr>
<tr>
<td>8642</td>
<td>4.56</td>
</tr>
<tr>
<td>7531</td>
<td>5.67</td>
</tr>
<tr>
<td>1470</td>
<td>6.78</td>
</tr>
</tbody>
</table>
BR analysis: “parts have a cost”

part (part number) have a of a cost ($s)

car part

Show any sub-type relationships

part (part number) have a of a cost ($s)
Applications of Artificial Intelligence

**B-R diagram for non-binary relationship**

- **Student** (student number)
- **Enrolment**
- **Subject** (code)
- **Scores** of an
- **Grade** (score%)
B-R diagram for arithmetic sum

integer pair

integer

has sum

sum of

integer

integer pair

integer
“the selling price of a part is the cost price of that part marked up by the mark-up rate for that part”
“the selling price of a part is the cost price of that part marked up by a universal mark-up rate”

“the selling price of a part is the cost price of that part marked up by a universal mark-up rate”
Knowledge Representation

Knowledge things are represented by clusters, fields and instances; the set of instances associated with a cluster at any particular time is called the value set of that cluster.

<table>
<thead>
<tr>
<th>part/sale-price</th>
<th>part/cost-price</th>
<th>mark-up rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>part-number</td>
<td>$-amount</td>
<td>factor-%</td>
</tr>
<tr>
<td>1234</td>
<td>1.48</td>
<td></td>
</tr>
<tr>
<td>2468</td>
<td>2.81</td>
<td></td>
</tr>
<tr>
<td>3579</td>
<td>4.14</td>
<td></td>
</tr>
<tr>
<td>8642</td>
<td>5.47</td>
<td></td>
</tr>
<tr>
<td>7531</td>
<td>6.80</td>
<td></td>
</tr>
<tr>
<td>1470</td>
<td>8.14</td>
<td></td>
</tr>
<tr>
<td>1234</td>
<td>1.23</td>
<td>1.2</td>
</tr>
<tr>
<td>2468</td>
<td>2.34</td>
<td></td>
</tr>
<tr>
<td>3579</td>
<td>3.45</td>
<td></td>
</tr>
<tr>
<td>8642</td>
<td>4.56</td>
<td></td>
</tr>
<tr>
<td>7531</td>
<td>5.67</td>
<td></td>
</tr>
<tr>
<td>1470</td>
<td>6.78</td>
<td></td>
</tr>
</tbody>
</table>
Knowledge analysis is a methodology for representing the knowledge things in an application.

It has three main steps:

• apply BR analysis to build a BR model of the data and information;

• extract a set of predicates from the BR model, and

• build a model of the knowledge expressed in terms of that set of predicates.
Consider:

This diagram contains a representation of four data things and two information things.
Six predicates are thus extracted:

is-a[x:part-number] which means “x is a valid part number”

is-a[x:mark-up-rate-%] which means “x is the universal mark-up rate”

is-a[x:cost-$s] which means “x is a valid cost price”

is-a[x:sale-price-$s] which means “x is a valid sale price”

part/sale-price(x, y) which means “y is the sale price identified with part x”

part/cost-price(x, y) which means “y is the cost price identified with part x”
The fundamental modelling principle

• the knowledge must be expressed in terms of these predicates.

If this is not possible then one reason for this may be that the BR model is incomplete.

There are two distinct notations for knowledge representation. The first, called “cluster diagrams”, is a less detailed notation than the second, called “dependency diagrams”.
A *cluster diagram* is an undirected graph containing a node for each predicate, and there is an additional node to which the other nodes are joined by an arc. The cluster is given an identifying number.
A dependency diagram is a directed graph containing a node for each predicate, and there is an additional node to which those nodes are joined by arcs. A directed arc is drawn to the node which is the head predicate.
Knowledge-Based Systems

The Problem
The declarative paradigm
KT Applications
Data, Information & Knowledge
Knowledge-Based Systems
Modelling Tools
→ Logic programming
“Logic for Knowledge Representation”

Classical Logic of Zero-Order

Proposition

a statement which asserts or denies and is therefore true or false (or ‘unknown’)

Zero-order analysis

identify each proposition

Also ‘propositional calculus’
replace each proposition by a formal propositions:

\[ p, q, r, s, p_1, q_1, r_1, s_1, p_2, q_2, \ldots \ldots \ldots, \]

replace connectives:

- if .... then .... by \( \rightarrow \)
- and by \( \land \)
- or by \( \lor \)
- not by \( \neg \)
Premises:

(1) \( p \rightarrow q \)

(2) \( p \)

Conclusion:

\( q \)

where:

‘\( p \)’ stands for “the world is round”

‘\( q \)’ stands for “the ancient Greeks were clever”
The validity of arguments in $L^0$ may be established by truth tables.

$L^0$ is very weak because the internal structure of propositions is lost.

“Peter is late and Peter is worried”

becomes

$p \land q$
First- and Higher-Order Logics

Zero-order analysis:

the proposition “Socrates is a man” is simply represented by ‘p’, say.

First-order analysis:

splits the proposition up into subject and predicate form. Let

‘a’ represent “Socrates” (the subject)

‘P’ the property of “being a man” (the predicate)

P(a), to be read “a has property P”.
First-order analysis:
based on a division between
‘objects’, and
‘properties’ or ‘sets of objects’
and thus is restrictive
P(a): ‘P’ a property, ‘a’ an object

Higher-order logic
unrestricted
Higher-order logic

inherently inconsistent

Def of predicate \( P \)

\[ P(X) \text{ if and only if } \sim X(X) \]

Problem

\[ P(P) \text{ if and only if } \sim P(P) \]
In arithmetic “$a=b$”;
• has two subjects, ‘a’ and ‘b’,
• represented as ‘$E(a,b)$’.

Predicates may be needed having 3, 4, 5,…. or any finite number of arguments.
‘a × b = -c’

- two subjects could be ‘a × b’ and ‘-c’
- function ‘f’ where ‘f(a,b)’ stands for ‘a × b’
- function ‘g’ where ‘g(c)’ stands for ‘-c’
- ‘E(f(a,b), g(c))’.

Hence we will need functions of 1, 2, 3, ... and, in general, of n-arguments.

- three subjects ‘a’, ‘b’ and ‘c’
- predicate ‘F(a, b, c)’ to mean ‘a × b = c’
- ‘F(a, b, g(c))’.
Computational Logic

First-order logic:

• first computed in the mid 1960's.
• chosen as the kernel programming language for the Japanese FGCS project.
• will be the target language for knowledge systems on FGCS machines.

We will use the language of logic to represent “knowledge”.
A *knowledge language* is a formalism that:

- admits a “declarative semantics”
- can be interpreted both as:
  - a programming language and
  - a database language.
We repeatedly decompose “things” until we identify the “fundamental, indivisible, real objects” which are represented by a label.

variable

quantifiers

universal quantifier

“For all values of the variable named x” written as “(∀x)”

existential quantifier

“For at least one value of the variable named y” written as “(∃y)”.
function
$f : \times^n T \rightarrow T$

where $T$ is the set of “terms”.

A term is defined by:

- the atomic symbols are terms
- if the set $\{t_i\}_{i=1,..,n}$ are n terms and $f$ is an n-adic function then $f(t_1,t_2,..,t_n)$ is a term.

Eg.

$f( 2, 3 )$

$f( f( 3, 1 ), f( -4, f( 5, 6 )))$
Lists:
list( a , list( b , list ( c , φ ) ) )

We use the notation “x.y”.
List structures find a natural place in logic.
Applications of Artificial Intelligence

Trees:

\[ t(t(t(\emptyset, c, \emptyset), b, t(\emptyset, d, \emptyset)), a, t(\emptyset, e, \emptyset)) \]

[2.2]
Predicate:

\[ P : x^n T \rightarrow \{ \text{TRUE}, \text{FALSE} \} \]

If a predicate could reasonably be stored as a table it is called a *relation*.

Eg. “the cost of item number #1234 is $12”

\[
\text{cost12('#1234')}
\]

\[
\text{item/cost:$( '#1234' , '12' )}
\]
Applications of Artificial Intelligence

Eg. “all items cost $12”: 
\[(\forall x) \text{item/cost:}$( x, '12' )\]
Eg. “at least one item costs $12”: 
\[(\exists x) \text{item/cost:}$( x, 12 )\]
Eg. “Every item has a cost”: 
\[(\forall x)(\exists y) \text{item/cost:}$( x, y )\]
Introduce “Skolem functions”: 
\[(\forall x) \text{item/cost:}$( x, f(x) )\]
Connectives:

- meaning “not”
- meaning “and”
- meaning “or”
- meaning “implies”
- meaning “is implied by”

Eg. “all items costing $12 are out of stock”:

item/cost:$( x , 12 ) \rightarrow \sim \text{instock}( x )
Language of first-order logic:

- “predicates”,
- “labels”,
- “variables”,
- “functions” and
- “connectives”.

Applications of Artificial Intelligence
Does not admit:

for all properties P, \( P(x) \implies P(x) \)

\( P( P( x ) ) \)

"it is possible that...."

"go home"

Classic problem:

To determine whether \( T \) is a logical consequence of \{A1, A2,..., An\}.

Theorem.

No algorithm — "semi-algorithms"
First construct the goal

Ans ⇐ T

answer predicate.

Deducing T from:

\{A1, A2, ..., An\}

equivalent to

deducing “Ans” from:

\{A1, A2, ..., An, Ans ⇐ T \}
Special forms are decidable.

Eg.

\[ A \lor \neg B \lor \neg C \lor \neg D \]

which is logically equivalent to:

\[ A \leftarrow B, C, D \]

or

\[ A \leftarrow (B \land C \land D) \]

where A, B, C and D are predicates.
Eg.

\[
\begin{align*}
\text{Animal}(x) & \leftarrow \text{Human}(x) \\
\text{Human('Socrates')} & \leftarrow \\
\text{then:} \\
\text{Animal('Socrates')} & \leftarrow \\
& \quad \text{Human('Socrates')} \\
\text{Human('Socrates')} & \leftarrow \\
\text{hence:} \\
\text{Animal('Socrates')} & \leftarrow
\end{align*}
\]
Eg. 

\[ P( x , f(x) ) \leftarrow Q( x ) , R( x , a ) \] 
\[ S( y ) \leftarrow V( a , y ) , P( b , y ) , U( f(b) ) \] 

then 

\[ P( b , f(b) ) \leftarrow Q( b ) , R( b , a ) \] 
\[ S( f(b) ) \leftarrow V( a , f(b) ) , P( b , f(b) ) , U( f(b) ) \] 

hence 

\[ S( f(b) ) \leftarrow V( a , f(b) ) , Q( b ) , R( b , a ) , U( f(b) ) \]
Resolvant

- Unification
- Resolution

First choose:

- which two clauses to resolve, then
- which two predicates to unify.

Eg.

\[ P( x, f(x) ) \leftarrow Q( a ), R( a, x ) \]
\[ S( x ) \leftarrow P( f(a), x ), P( a, f(a) ) \]
Control strategy

LUSH

SL

SLD
SL first focuses on the goal sentence, \( \text{Ans} \not\leftarrow T \)

then it takes the leftmost predicate in the body of the sentence in focus and attempts to resolve it with the head of one of the clauses \{A1, A2,\ldots, An\} in that order.

If successful

then the resulting clause becomes the new sentence in focus

else

the system backtracks.

Procedure halts when:

\( \text{Ans} \not\leftarrow \)

has been constructed
Applications of Artificial Intelligence

Eg.

1. \( P(a) \leftarrow P(b) \)
2. \( P(b) \leftarrow \)
3. \( Q(b) \leftarrow \)
4. \( \text{Ans} \leftarrow P(x), Q(x) \)

hence:

5. \( \text{Ans} \leftarrow P(b), Q(a) \)
6. \( \text{Ans} \leftarrow Q(a) \)

Backtrack.

5. \( \text{Ans} \leftarrow Q(b) \)
6. \( \text{Ans} \leftarrow \)
Tree for this problem:

SL search strategy searches this tree

Admissible search strategy will find the solution node eventually.

Applications of Artificial Intelligence
Applications of Artificial Intelligence

Eg.

item/cost:$\{ \#1234, 12 \} \leftarrow$

item/cost:$\{ \#2468, 25 \} \leftarrow$

item/cost:$\{ \#3579, 8 \} \leftarrow$

item-list/price-list(∅, ∅) \leftarrow

item-list/price-list(x.y, u.v) \leftarrow

item/cost(x, u), item-list/price-list(y, v)

and

Ans \leftarrow item/cost:$\{ \#3579, 8 \}$

or

Ans \leftarrow item-list/price-list(\#1234.\#3579, 12.8)
Late 1960's:

- Computational logic to mechanise the proof of mathematical theorems.
- Not been achieved.
- Combinatorial explosion.

Research produced some practical applications.

Concerned with clausal logic as a programming language and database language.

Both derived from clausal logic as a question-answering language.
Question-answering systems

$$\text{Ans}(x) \leftarrow \text{item-list/price-list}(\#1234.\#3579, x)$$

then

$$\text{Ans}(\$12.\$8) \leftarrow$$

Interpret as a simple computer program which executes the command:

“calculate the price list for items 1234 and 3579”

Pure logic programming

Enhancements:

the provision of specific computational features
the provision of control features
Naive, but note:

demonstration of “partial correctness”
is trivial

non-deterministic

not “goal dependent”

Eg.

Ans( x ) ← item-list/price-list( x , $25.$8 )

Ans( x ) ← item/cost( x , x )
Interpreted as simple database.

\[
\text{Ans}( x ) \leftarrow \text{item-list/price-list}( \#1234.\#3579, x )
\]

with

\[
\text{item-list/price-list}( \emptyset, \emptyset ) \leftarrow
\]

\[
\text{item-list/price-list}( x.y, u.v ) \leftarrow
\]

\[
\text{item/cost}( x, u), \text{item-list/price-list}( y, v )
\]

then

\[
\text{Ans}( $12.8 ) \leftarrow
\]

where:

“item/cost” is the name of a relation, and

“item-list/price-list” is the name of a procedure
The goal sentence:

“retrieve the price list for items numbered 1234 and 3579”

*pure logic database*

Enhancements:

an efficient mechanism for storage and retrieval of the actual data

implement instructions such as “find all but without repetition”
Important properties of logic as a database language

Knowledge represented in one place.

No difference between the retrieval of stored information and the calculation of information.

Real relation

Virtual relation
Ans will find the “first” answer
“find-one predicate”

Eg.
\[ \text{Ans}(x, y) \leftarrow \text{item/cost:}\$(x, y) \]

will return
\[ \text{Ans}(#1234, 12) \leftarrow \]
find-all predicate.

Eg.

tuple/find-all-list( (x), y )

When control is passed to this predicate the value of tuple (x) is added to the list y, the predicate then “fails” and the search “backtracks” to look for the next tuple (x) until the search for new tuples (x) is unsuccessful.
Eg.

\[
\text{Ans}( z ) \leftrightarrow \text{item/cost:$( x, y )}, \quad \text{tuple/find-all-list( $( x, y ), z\text{) retrieves:} }
\]

<table>
<thead>
<tr>
<th>x</th>
<th>y:($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1234</td>
<td>12</td>
</tr>
<tr>
<td>#2468</td>
<td>25</td>
</tr>
<tr>
<td>#3579</td>
<td>8</td>
</tr>
</tbody>
</table>
Note:

\((\forall x) \sim P(x) \iff \sim (\exists x) P(x)\)

so

“that it is not the case that there exists an \(x\) such that \(P(x)\)”

is equivalent to

“that for all \(x\) it is not the case that \(P(x)\)”
Logic as a Programming Language

Horn clause logic extended to a practical programming language by

introducing a set of computational procedures and

providing additional control facilities
Eg. arithmetic facilities are made available by inbuilt predicates.

\[
\text{plus}( x, y, z ) \\
\text{times}( x, y, z )
\]

Eg.

\[
\text{Ans}( x ) \leftarrow \text{plus}( 2, x, 7 )
\]

then

\[
\text{plus}( 2, 5, 7 ) \leftarrow
\]

would be assumed and thus

\[
\text{Ans}( 5 ) \leftarrow
\]

Also

\[
\text{less-than}( x, y )
\]
Eg.

item/sell( x, y ) ⇐ item/cost( x, z ), times( z, 1.25, y )

invoice( ∅, 0) ⇐ 

invoice( x.y, z ) ⇐ invoice( y, w ),

item/sell( x, v ), plus( v, w, z )

Note the business rule:

“that all items are marked up by 25%”

Note partial correctness may be easily proved
SL is too simple.

either devise more intelligent control mechanisms

or provide control instructions.

Practical implementations of logic contain many in-built predicates and other features.

In Prolog there is a “cut” operator.

No standard has been generally agreed to

Note the difference between “Horn clause logic” and “Prolog”

Note that the Japanese adopted Horn clause logic, and not Prolog
Use of Logic as a Programming Language

When we introduce a new predicate we state what that predicate “means”; sufficient to demonstrate partial correctness

NB: no need for loop invariants

Logic admits a *declarative semantics*, ie. statements may be interpreted as statements of fact, in addition to *imperative semantics* as a sequence of commands determined by the control

Also non-imperative

Language with declarative semantics cannot contain any purely imperative statement forms.

Thus no assignment statement in logic programming
Logic programming:
- admits trivial correctness proofs,
- is non-deterministic,
- is non-goal dependent,
- is very high level,
- is very simple,
- is very powerful,
- no distinction between program and data,
- no distinction between input and output,
- logic variable not the name for a location,
- imperative semantics “totally defined”
Logic programming is based on positive logic

Negation introduced by:

the provision of a “negative” predicate, or

an extension to the control mechanism
Applications of Artificial Intelligence

Eg.
not[item/cost](x,y) ⇐ x = 1234, x = 2468, x = 3579
not[item/cost](x,y) ⇐ x = 1234, x = 2468, y = 8
not[item/cost](x,y) ⇐ x = 1234, y = 25, x = 3579
not[item/cost](x,y) ⇐ x = 1234, y = 25, y = 8
not[item/cost](x,y) ⇐ y = 12, x = 2468, x = 3579
not[item/cost](x,y) ⇐ y = 12, x = 2468, y = 8
not[item/cost](x,y) ⇐ y = 12, y = 25, x = 3579
not[item/cost](x,y) ⇐ y = 12, y = 25, y = 8
Note that:

\[ \text{not[item/cost]}(\ #6, \ 18 \ ) \leftarrow \]

is true because there is no such thing as item number 6, and, in this example, there is no such thing as an item which costs $18.

Often not practical to construct negative predicates.
“Negation as failure”: a statement is assumed to be “false” if an attempt to prove it “true” fails

Eg.

\[ \text{Ans} \leftarrow \text{item/cost:}$( \#4321 \ , \ 16 ) \]

hence “it is not the case that item number 4321 costs $16”

Eg. to demonstrate that “no invoice could cost $19”:

\[ \text{Ans}(x) \leftarrow \text{invoice}( \ x \ , \ 19 ) \]

would not fail; would compute forever
Logic as a Database Language

Eg.

item/cost:$\{ \#1234, 12 \} \leftarrow$

item/cost:$\{ \#2468, 25 \} \leftarrow$

item/cost:$\{ \#3579, 8 \} \leftarrow$

item-list/price-list( $\emptyset$, $\emptyset$ ) $\leftarrow$

item-list/price-list( x.y, u.v ) $\leftarrow$

item/cost( x, u ), item-list/price-list( y, v )

item/sell( x, y ) $\leftarrow$ item/cost( x, z ), times( z, 1.25, y )

invoice( $\emptyset$, 0 ) $\leftarrow$

invoice( x.y, z ) $\leftarrow$ invoice( y, w ),

item/sell( x, v ), plus( v, w, z )
Pure Horn clause logic may be extended to a database language by providing fundamental database features and additional control facilities.

Storage mechanism may be
simple, or
have standard database features.

The storage mechanism employed is the *database engine* or DBE.

The DBE may be
part of an integrated logic dbms, or
an auxiliary system linked to the logic system.
When the DBE is simple

it may be coupled naturally by providing item/cost as a built in predicate.

Data can be stored using the special “STORE-TUPLE” predicate:

STORE-TUPLE(item/cost( #1470 , 3:$ ) )

Also

“DELETE-TUPLE”, and

“REPLACE-TUPLE”
Important built in predicate is the “is-a” predicate:

\[ \text{is-a}[\text{<descriptor>}](x) \]

Eg. to find a single, valid item number:

\[ \text{Ans}(x) \leftarrow \text{item/cost}(x, y) \]

or

\[ \text{Ans}(x) \leftarrow \text{is-a[item]}(x) \]

Eg.

\[ \text{Ans}(x) \leftarrow \text{is-the[today's-date]}(x) \]
When the DBE is supported by standard database features which include database integrity checks.

\[
\text{REPLACE-TUPLE}( \text{item/cost:$( #1470 , 3 )} , \\
\text{item/cost:$( #1470 , 4) , R } )
\]

where R is a flag-predicate.

Also

\[
\text{BEFORE}( \text{event1 , event2 } )
\]
If the logic is permitted to interact with the DBE without restriction then anomalies can occur.

Eg.

\[ \text{Ans}(x) \leftarrow \text{REPLACE}( \text{item/cost:}$$(x, 8)$$, \text{item/cost:}$$(x, 9)$$), \ x < 1999 \]

When the SL strategy is applied it will fail. However, in the process of the calculation the left-most predicate in the goal will have been satisfied.
Eg.

Ans( x ) ← REPLACE( item/cost:$( x, 8 ) ,
item/cost:$( x, 9 ) ), x > 3500

would succeed, would have returned the answer “3579”, and, as a result, the cost of item number #3579 would have been increased from $8 to $9
Eg.

$$\text{Ans}(x) \leftarrow \text{item/cost:}$(x, 12)$, \text{STORE-TUPLE}(\text{item/cost:}$(x, 13)$), \text{P}(x)$$

becomes:

$$\text{Ans} (#1234) \leftarrow \text{STORE-TUPLE(} \text{item/cost:}$(1234, 13)$), \text{P} (#1234)$$

This goal should fail and the calculation should backtrack. This can be enforced by “locking” the record:

$$\text{item/cost:}$(#1234, 12)$$$

A locked record cannot be altered until it is “unlocked”
Eg.

\[
\text{item/cost:$( #1234, 12 )}$
\]

would be locked

This record will then remain locked until either the goal is satisfied or the strategy backtracks to the point of the above resolution
Data in a complex DBE would certainly have some internal structure.

Suppose that a major requirement had been identified for retrieving the list of items with a given cost:

\[
\text{cost:$/item-list( x , y )}
\]

this in-built predicate will be identified at design time

An *inverse predicate*, not to be treated as an “ordinary” predicate. It will only operate efficiently with the first argument as “input”
Eg.

\[ \text{Ans}(x) \leftarrow \text{cost:}\$/\text{item-list}(3, x) \]

Useful for the goals:

- to show that there are only three items which cost $17
- there is no item costing $19
Three choices for the implementation of an inverse predicate:

- if the inverse predicate is a relation then its values may be stored as a relation in the ordinary way

- may be defined in terms of its “original” predicate using the tuple/find-all-list predicate. Eg.

  \[
  \text{cost:$/item-list( x, y ) \leftarrow item/cost:$( z, x ),
  \text{tuple/find-all-list( (z), y )}
  \]

- may be defined in terms of the other predicates in the system, some of which may themselves need to be inverted
Logic is a universal database language:
• data description,
• data manipulation,
• transaction specification and
• integrity checking
• can all be represented in logic, which also
  provides
• the essential syntax of interface languages

Note if database management system has
restart and recovery
Use of Logic as a Database Language

Predicates may be represented as either real or virtual relations; this will be determined by the functional requirements of the system.

Eg.

\[
\text{item/sell}( x, y ) \leftarrow \text{item/cost}( x, z ),
\text{times}( z, 1.25, y )
\]

implemented in five different ways
(1) item/cost could be stored as a real relation, in which case the rule

\[
\text{item/sell}(x, y) \leftrightarrow \text{item/cost}(x, z), \\
\text{times}(z, 1.25, y)
\]

defines the virtual relation item/sell

(2) The rule could be inverted to define item/cost in terms of item/sell by:-

\[
\text{item/cost}(x, z) \leftrightarrow \text{item/sell}(x, y), \\
\text{times}(z, 1.25, y)
\]

in which case the rule defines the virtual relation item/cost in terms of the real relation item/sell
If both item/sell and item/cost are stored then:

(3) Either the rule in (1) or the rule in (2) could be used as an integrity constraint to check that item/sell and item/cost are consistent after an update.

(4) The rule in (1) could be used to update item/sell when item/cost is updated. In which case, some form of locking of files would be required.

(5) The rule in (2) could be used to update item/cost when item/sell is updated. Again, file locking would be required.
Within a logic database implementation, logic may also be used as a powerful programming language for performing complex calculations on or with the data.

A logic database may have substantial real relations stored in the DBE; can yield very expensive searches.
Eg.

Ans ← item/cost( x , 3:$ ),
    times( z , 10 , x )

This could initiate a very expensive search of the whole of the item/cost relation.

One way of reducing the cost of searches such as this within the logic itself, is to use inverse relations.
Writing Logic Programs

Substantial experience with conventional programming languages can be a hindrance.

- unfamiliarity of the data structures natural too logic programs
- logic programs admit a non-imperative interpretations

Eg:

\[ x = 1 \]

Can find no place in a logic programmer’s thinking
The logic programmer must be able to think non-imperatively.

“Non-imperative thinking”

is not contradictory to “algorithmic thinking”

“Algorithmic thinking” is necessary for the construction of good programs in predicate logic

The difficulty is to see algorithms in non-imperative terms

think of logic programs as procedures which check whether or not a proposed solution is in fact correct
Eg a program to add together a list of numbers. 

Predicate SUM(X,Y) to be “the arithmetic sum of the elements in the list X is the number Y”.

Goal statement:

\[ \text{ANSWER}(X) \leftarrow \text{SUM}(<\text{input list}>,X) \]

Next define SUM(X,Y) if the list is non-empty:

\[ \text{SUM}(\text{list}(A,Z), Y) \leftarrow \text{SUM}(Z,W), A + W = Y \]

if the list is empty:

\[ \text{SUM}(\text{nil},0) \leftarrow \]
Eg Factorial.

The predicate Fac(x,y) to mean “the factorial of x is y”:

\[
\begin{align*}
\text{Fac}(0,1) & \iff \\
\text{Fac}(x,y) & \iff x > 0, \text{Fac}(z,w), z=x-1, y = x \times w \\
\text{Fac}(x, "undefined") & \iff x < 0
\end{align*}
\]
Calculate the Factorial of given integer ‘a’:

Answer(x) ← Fac(a,x)

Calculate the inverse factorial function

calculate the number whose factorial is ‘a’:

Answer(x) ← Fac(x,a)

Find x such that x! = 6 × x:

Answer (x) ← Fac(x,y), y = 6 × x
Compare with the program:

```plaintext
function Factorial (n)
begin
  if x < 0 then return "undefined"
  if x = 0 then return "1"
  if x > 0 then
    begin
      m ← 1
      for k = 1, n do m ← m × k
      return m
    end
  end
end
```
Properties

- logic programs are not goal dependent
- statements in a logic program are not (necessarily) order sensitive.
- logic programs bear a very natural relation to the wisdom in the problem
- logic programs are able to “compute” a range of problems.
Example

Translate thinking from an Pascal-like language to Logic programming.

Consider Quicksort. Using the predicates:

Sort(X,Y) to mean “Y is a sorted version of list X”

Part(W,X,Y,Z) to mean “list X is partitioned into two lists Y and Z. Y contains those elements less than or equal to element W and Z contains the remainder”.

Cat (X,Y,Z) to mean “list X and Y have been catenated to give list Z”.
The Quicksort algorithm may be written:

```plaintext
procedure Sort (x,y)
begin
  if x = nil
    then begin
      atom x1; list x2, u1, u2, v1, v2
      x1 ← head(x); x2 ← tail(x)
      Part(x1, x2, u1, u2)
      Sort(u1, v1); Sort (u2, v2)
      Cat(v1, list (x1, v2), y)
    end
  end

if x = nil then y ← nil
end
```
First step: admit pattern matching, this makes the functions ‘head’ and ‘tail’ redundant:

```
procedure Sort (x,y)
begin atom x1; list x2, u1, u2, v1, v2
  if x = list (x1,x2)
    then begin Part(x1, x2, u1, u2)
      Sort(u1, v1); Sort (u2, v2)
      Cat(v1, list (x1, v2), y)
    end
  if x = nil then y ← nil
end
```
Second step: observe that the variable x is essentially redundant; by removing it we extract the imperative aspects from the program:

```pascal
procedure Sort (list (x1, x2), y)
begin list u1, u2, v1, v2
   Part(x1, x2, u1, u2)
   Sort(u1, v1); Sort(u2, v2)
   Cat(v1, list (x1, v2), y)
end

procedure Sort (nil, nil)
```
Third step: translate the program into logic programming:

\[
\text{Sort}(\text{list}(x_1, x_2), y) \leftarrow \text{Part}(x_1, x_2, u_1, u_2), \\
\text{Sort}(u_1, v_1), \text{Sort}(u_2, v_2), \\
\text{Cat}(v_1, \text{list}(x_1, v_2), y) \\
\text{Sort}(\text{nil}, \text{nil}) \leftarrow
\]
Fourth step: add definition of the procedures Part and Cat:

\[
\begin{align*}
\text{Part} & (x_1, \text{list}(z, x_2), \text{list}(z, u_1), u_2) & \leftarrow & \langle z, x_1 \rangle, \\
& - & \text{Part} & (x_1, x_2, u_1, u_2) \\
& - & \text{Part} & (x_1, \text{list}(z, x_2), u_1, \text{list}(z, u_2)) & \leftarrow & \langle x_1, z \rangle \\
& - & \text{Part} & (x_1, x_2, u_1, u_2) \\
& - & \text{Part} & (x, \text{nil}, \text{nil}, \text{nil}, \text{nil}) & \leftarrow \\
& - & \text{Cat} & (\text{list}(u, x), y, \text{list}(u, z)) & \leftarrow & \text{Cat} & (x, y, z) \\
& - & \text{Cat} & (\text{nil}, y, y) & \leftarrow
\end{align*}
\]
Logic programming

- is a very high level language
- is non-deterministic
- is non-goal dependent
- admits declarative and imperative semantics.
- is an exceptionally simple language
- contains no distinction between input and output
- contains no distinction between data and program
Logic programming

It is not correct to think of a logic programming variable as a name for a machine storage location.

Conceptual problems of “when is what assigned to which” are not the concern of the logic programmer.

The procedural or imperative semantics of Logic programming is totally defined.

It is impossible for a syntactically correct program to perform an illegal or undefined operations.